Chapter 1
Why SYSTEMC: ESL and TLM

1.1 Introduction

The goal of this chapter is to explain why it is important for you to learn SystemC. If you already know why you are studying SystemC, then you can jump ahead to Chapter 2. If you are learning SystemC for a college course or because your boss says you must, then you may benefit from this chapter. If your boss doesn't know why you need to spend your time learning SystemC, then you may want to show your boss this chapter.

SystemC is a system design and modeling language. This language evolved to meet a system designer's requirements for designing and integrating today's complex electronic systems very quickly while assuring that the final system will meet performance expectations.

Typically, today's systems contain both application-specific hardware and software. Furthermore, the hardware and software are usually co-developed on a tight schedule with tight real-time performance constraints and stringent requirements for low power. Thorough functional (and architectural) verification is required to avoid expensive and sometimes catastrophic failures in the device. In some cases, these failures result in the demise of the company or organization designing the errant system. The prevailing name for this concurrent and multidisciplinary approach to the design of complex systems is electronic system-level design or ESL.

The drive for concurrent engineering through ESL has side effects that affect more than the design organizations of a company. ESL affects the basic business model of a company and how companies interact with their customers and with their suppliers.

ESL happens by modeling systems at higher levels of abstraction than traditional methods used in the past. Portions of the system model are subsequently iterated and refined, as needed. A set of techniques has evolved called Transaction-Level Modeling or TLM to aide with this task.

ESL and TLM impose a set of requirements on a language that is different than the requirements for hardware description languages (HDLs) or the requirements

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_1, © Springer Science+Business Media, LLC 2010

for traditional software languages like C, C++[1], or Java. The authors believe that SystemC is uniquely positioned to meet these requirements.

We will discuss all these topics in more detail in the following sections.

1.2 ESL Overview

ESL techniques evolved in response to increasing design complexity and increasingly shortened design cycles in many industries. Systems, Integrated Circuits (ICs), and Field Programmable Gate Arrays (FPGAs) are becoming large. Many more multi-disciplinary trade-offs are required to optimize the hardware, the software, and the overall system performance.

1.2.1 Design Complexity

The primary driver for an ESL methodology is the same driver that drove the evolution of previous design methodologies: increasing design complexity.

Modern electronic systems consist of many subsystems and components. ESL focuses primarily on hardware, software, and algorithms at the architectural level. In modern systems, each of these disciplines has become more complex. Likewise, the interaction has become increasingly complex.

Interactions imply that trade-offs between the domains are becoming more important for meeting customer requirements. System development teams find themselves asking questions like:

- Should this function be implemented in hardware, software, or with a better algorithm?
- Does this function use less power in hardware or software?
- Do we have enough interconnect bandwidth for our algorithm?
- What is the minimum precision required for our algorithm to work?

These questions are not trivial and the list could go on and on. Systems are so complex, just deriving specifications from customer requirements has become a daunting task. Hence, this task brings the need for higher levels of abstraction and executable specifications or virtual system prototypes.

Figure 1.1 illustrates the complexity issues for just the hardware design in a large system-on-a-chip (SoC) design. The figure shows three sample designs from three generations: yesterday, today, and tomorrow. In reality, tomorrow's systems are being designed today. The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design:

- Architecture
- Behavioral
- RTL
- Gates

[1] We will see later that SystemC is actually a C++ class library that "sits on top" of C++.

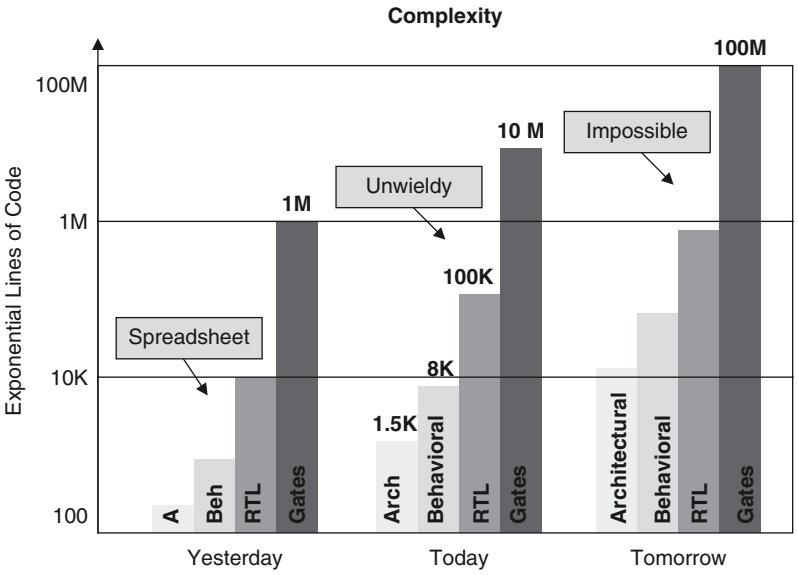

Fig. 1.1 Code complexity for four levels of abstraction

Today's integrated circuits often exceed 10 million gates, which conservatively translates to one hundred thousand lines of RTL code. Today's designs are practical because of the methodologies that apply RTL synthesis for automated generation of gates. Tomorrow's integrated circuits, which are being designed today, will exceed one hundred million gates. This size equates to roughly one million lines of RTL code, if written using today's methodologies.

Notice that Figure 1.1 considers only a single integrated circuit. It does not reflect the greater complexity of a system with several large chips (integrated circuits or FPGAs) and gigabytes of application software. Many stop-gap approaches are being applied, but the requirement for a fundamentally new approach is clear.

1.2.2 Shortened Design Cycle = Need For Concurrent Design

Anyone who has been part of a system design realizes that typical design cycles are experiencing more and more schedule pressure. Part of the reason for the drive for a shortened design cycle is the perceived need for a very fast product cycle in the marketplace. Anyone who has attempted to find a cell phone or a laptop "just like my last one", even just nine months after buying the "latest and greatest" model, will find themselves looking long and hard throughout the web for a replacement[2].

[2] This scenario describes a recent experience by one of the authors.

Many are under the misguided assumption that shorter design cycles imply reduced development expenses. If the scope of the new system is not reduced and the schedule is reduced, then additional resources are required. In this scenario, a shorter design cycle actually requires more resources (expenses). The project requires more communication between development groups (because of concurrent design), more tools, more people, and more of everything. ESL and TLM are an approach to reduce the cost of development through more efficient communication and through reduced rework.

1.2.2.1 Traditional System Design Approach

In the past, when many systems were a more manageable size, a system could be grasped by one person. This person was known by a variety of titles such as system architect, chief engineer, lead engineer, or project engineer. This guru may have been a software engineer, hardware engineer, or algorithm expert depending on the primary technology leveraged for the system. The complexity was such that this person could keep most or all of the details in his or her head. This technical leader was able to use spreadsheets and paper-based methods to communicate thoughts and concepts to the rest of the team.

The guru's background usually dictated his or her success in communicating requirements to each of the communities involved in the design of the system. The guru's past experiences also controlled the quality of the multi-disciplinary trade-offs such as hardware implementation versus software implementation versus algorithm improvements.

In most cases, these trade-offs resulted in conceptual disconnects among the three groups. For example, cellular telephone systems consist of very complex algorithms, software, and hardware. Teams working on them have traditionally leveraged more rigorous but still ad-hoc methods.

The ad-hoc methods usually consist of a software-based model. This model is sometimes called a system architectural model (SAM), written in C, Java, or a similar language. The SAM is a communication vehicle between algorithm, hardware, and software groups. The model can be used for algorithmic refinement or used as basis for deriving hardware and software subsystem specifications. The exact parameters modeled are specific to the system type and application, but the model is typically un-timed (more on this topic in the following section). Typically, each team then uses a different language to refine the design for their portion of the system. The teams leave behind the original multi-discipline system model and in many cases, any informal communication among the groups.

The traditional approach often resulted in each design group working serially with a series of paper specifications being tossed "over the wall" to the other organization. This approach also resulted in a fairly serial process that is many times described as a "waterfall schedule" or "transom engineering" by many program managers and is illustrated in Fig. 1.2.

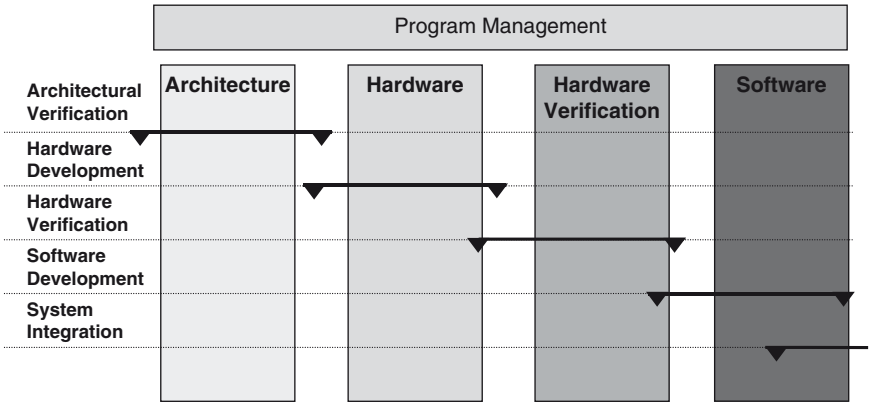

Fig. 1.2 The traditional approach of waterfall scheduling

To minimize the serialization of the design process, many techniques have been used to create some design concurrency. These techniques include processor development boards using a single-chip implementation of the processor. These implementations were used on the SoC or embedded system, FPGA prototypes of algorithms, and hardware emulation systems, just to name a few. These techniques were focused on early development of software, usually the last thing completed before a system is deployed.

The ESL approach uses these existing techniques. ESL also leverages a virtual system prototype or a TLM model of the system to enable all the system design disciplines to work in parallel. This virtual system prototype is the common specification among the groups. The resulting Gantt chart is illustrated next in Fig. 1.3.

Even though all of the electronic system design organizations will finish their tasks earlier, the primary reason for ESL is earlier development of software. Even getting a product to market a month earlier can mean tens of millions of dollars of business to a company.

Not using ESL methods will likely result in the under-design or over-design of the system. Both of these results are not good. Under-design is obviously not good. The product may be bug-free, but it doesn't necessarily meet the customer's requirements. The product may not operate fast enough, may not have long enough battery life, or just may not have the features required by the customer.

Over-design is not as obvious, but it is not good either. Over-design takes significantly more resources and time to achieve, and it adds a heavy cost to an organization. In addition, over-designed products usually are more complex, more expensive to manufacture, and are not as reliable.

The authors have significant anecdotal stories of under-design and over-design of systems. One company built an ASIC with multi-processors that made "timing closure" and paired those processors with software that made the "timing budget."

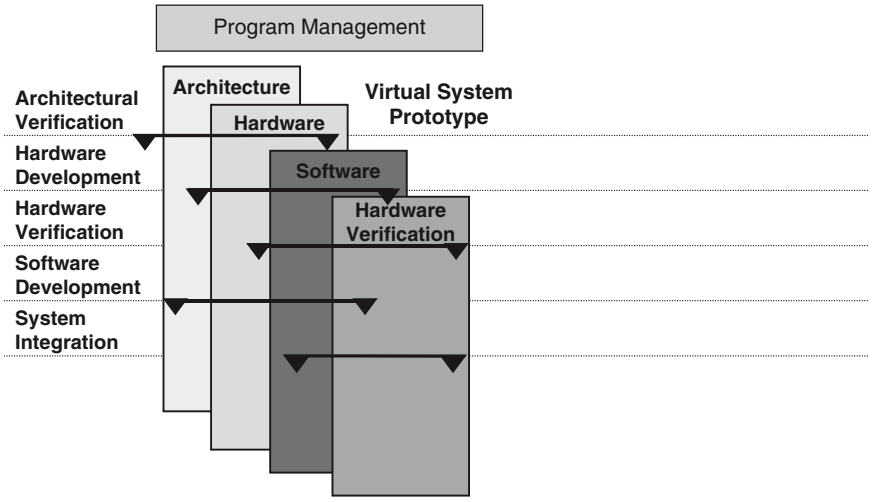

Fig. 1.3 The ESL approach of parallel schedule

Unfortunately, the ASIC didn't meet the customers requirements because of "on chip" bottlenecks. Another company related how a significant function on a chip caused weeks of schedule slip for design and verification. However, the function was later found not to be used by the software.

Things become even more interesting if a system, say a cell phone, are really a subsystem for the customer, who is a mobile phone and infrastructure provider. Now, the customer needs models very early in their design process and will be making system design trade-offs based on a model provided by the subsystem provider (previously system). In addition, the subsystem provider likely relies on third-party intellectual property. The subsystem cell phone supplier will then need a model of the intellectual property used in their subsystem very early in the development cycle to enable their design trade-offs. Customers up and down the value chain may now be making business decisions based on the type and quality of the model provided by their supplier. This hierarchy of reliance is fundamentally a different way of doing business.

The virtual system prototype may have different blocks (or components or subsystems) at different levels of abstraction for a particular task to be performed by one of the system disciplines. Initially, most of the system may be very abstract for software development until the software team is reasonably sure of the functionality. At this point, a more detailed model of the blocks that closely interact with the software can be introduced into the model.

The technique that allows this "mixing and matching" of blocks at different levels of abstraction is called Transaction-Level Modeling or TLM. We will discuss TLM in much greater detail in the following section.

1.3 Transaction-Level Modeling

TLM and the associated methodologies are the basic techniques that enable ESL and make it practical. To understand TLM, one must first have a terminology for describing abstraction levels. Secondly, one must understand the ways the models will probably be used (the use cases).

1.3.1 Abstraction Models

Several years ago, Professor Gajski from UC Irvine proposed a taxonomy for describing abstraction levels of models. The basic concept states that refinement of the interface or communication of a logical block can be refined independently of functionality or algorithm of the logical block [3]. We have significantly modified his concept and presented the result in the figure below.

- Un-Timed (UT)
- Loosely Timed (LT)
- Approximately Timed (AT)
- Register Transfer Logic (RTL)
- Pin and Cycle Accurate (PCA)

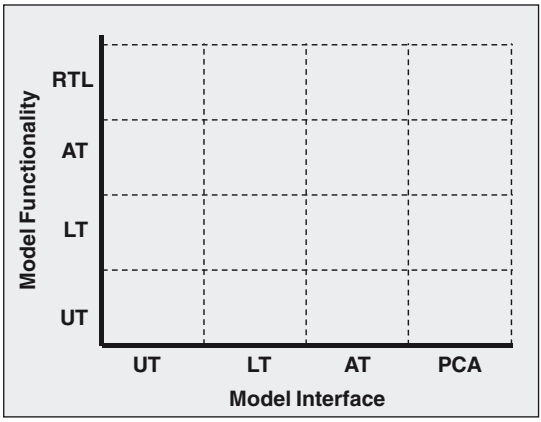

Fig. 1.4 Decoupling of abstraction refinement

[3] Gajski and L. Cai, "Transaction Level Modeling," First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2003), Newport Beach, CA, October 1, 2003

The y-axis of this graph is the abstraction level of the model functionality. The x-axis is the abstraction level of the model or logical block interface or communication. The key thing to notice is that the functionality can be refined independent of the Interface and the reverse is also true. A logical block may be approximately timed (AT) for the model functionality. For instance, it may be a basic "c call" to a graphics algorithm with a "lump" delay for the algorithm, possibly the specified number of pipeline delays. The corresponding interface for the block may be pin and cycle accurate (PCA). The interface replicates all the "pin wiggles" of the communication between blocks. This type of block is many times referred to as a bus functional model for functional verification.

When we map some common ESL use cases to this graph, the model interface can be loosely timed (LT) or approximately timed (AT) with an occasional bus cycle accurate (BCA) representation. When communication with RTL represented blocks is required, a transactor will be used to bridge to PCA. Model functionality can be un-timed (UT), LT, or AT abstraction level. This modeling use cases are graphically represented in the following figure:

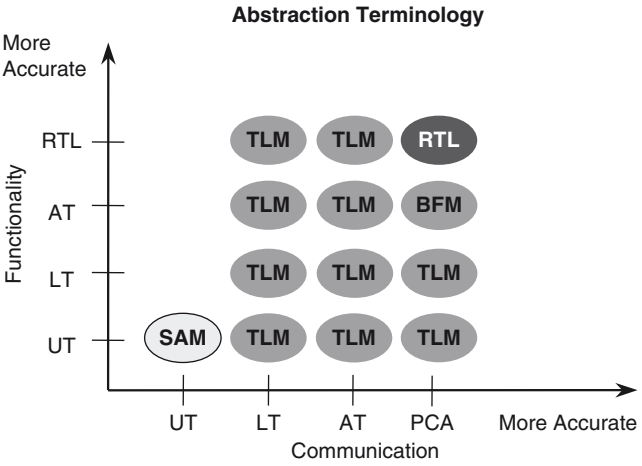

Fig. 1.5 TLM model mapping

For those from a software background, one can think of TLM style modeling as calling an application programming interface (API) for block-level model communication.

1.3.2 An Informal Look at TLM

In this section, we ease the reader into an understanding of TLM by presenting a less rigorous and more example-based discussion of TLM. We will assume a

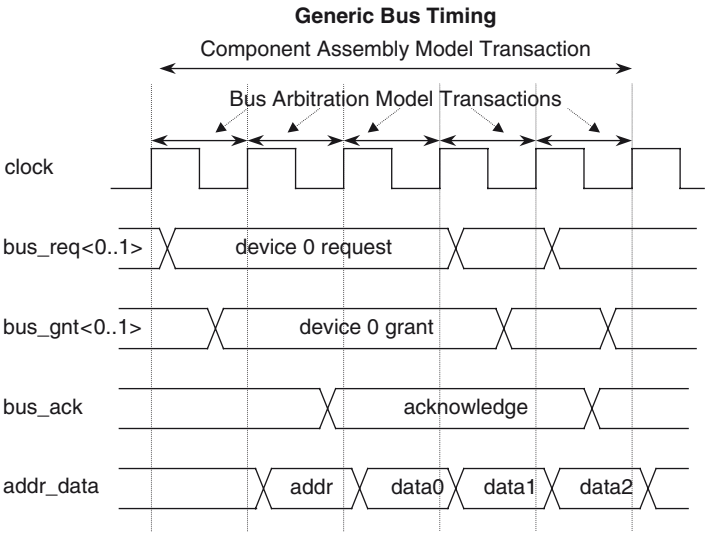

Fig. 1.6 Timing for generic bus

generic system containing a microprocessor, a few devices, and memory connected by a bus.

The timing diagram in Figure 1.6 illustrates one possible design outcome of a bus implementation. When first defining and modeling the system application, the exact bus-timing details do not affect the design decisions. All of the important information contained within the illustration is transferred between the bus devices as one event or transaction.

Further into the development cycle, the number of bus cycles may become important (to define bus cycle-time requirements, etc.). The information for each clock cycle of the bus is transferred as one transaction or event (bus-arbitration or cycle-accurate computation models).

When the bus specification is fully chosen and defined, the bus is modeled with a transaction or event per signal transition (bus functional or RTL model). Of course, as more details are added, more events occur and the speed of the model execution decreases.

In this diagram, the higher abstraction level model takes 1 "event" and the bus arbitration model takes approximately 5 "events." The RTL model takes roughly 75 "events" (the exact number depends on the number of transitioning signals and the exact simulator algorithm). This simple example illustrates the magnitude of computation required and why more system design teams are employing a TLM-based methodology.

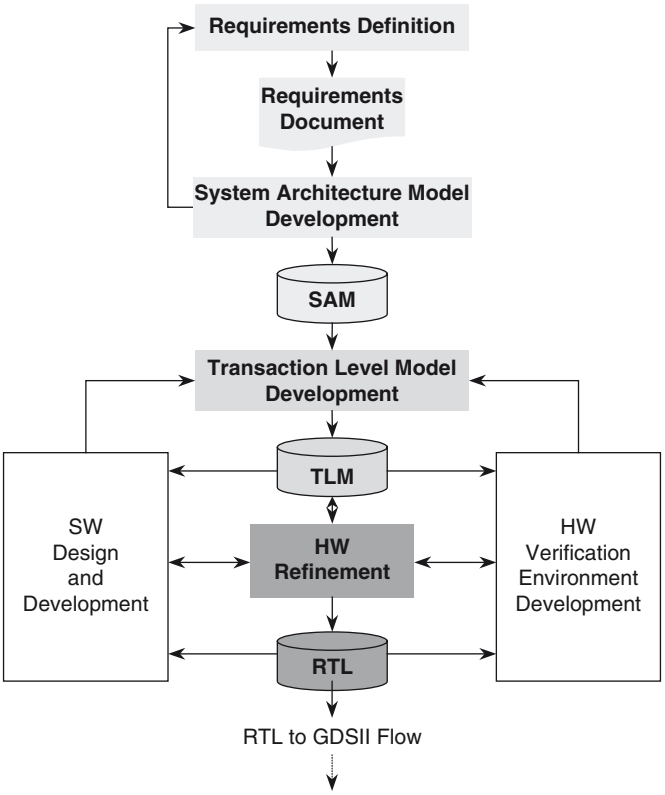

Fig. 1.7 TLM methodology

1.3.3 TLM Methodology

Now that we have discussed some of the TLM concepts, we can look more closely at a TLM-based methodology as illustrated in Figure 1.7.

In this methodology, we still start with the traditional methods used to capture the customer requirements: a paper-based Product Requirements Document (PRD). Sometimes, the product requirements are obtained directly from a customer. More likely, the requirements are captured through the research of a marketing group.

From the PRD, a high-level TLM model is developed. The TLM model development effort may cause changes or refinement of the PRD. The TLM model is usually written by an architect or architecture group. This model captures the product specification or system-critical parameters. In an algorithmic intensive system, the TLM model will be used to refine the system algorithms.

The TLM model is refined further as software design and development and hardware verification environment development progresses.

If the proper design language and techniques are used consistently throughout the flow, then the TLM can be reused and refined. The TLM has several use cases:

1. Architectural modeling
2. Algorithmic modeling
3. Virtual software development platform
4. Functional verification
5. Hardware refinement

We will further discuss these use cases in the next few sections.

At first, looked at from a particular development group's point of view, the development of the TLM appears to be a task with low return on investment (ROI). However, the TLM creates high value benefits including:

- Earlier software development, which is usually the schedule limiting task for deployment of a system
- Earlier and better hardware functional verification testbench
- A clear and unbroken path from customer requirements to detailed hardware and software specifications

After reading this book, you and your team should have the knowledge to implement TLM models quickly and effectively. The following section discusses in more detail the benefits that TLM modeling will bring to your organization.

1.3.3.1 Algorithmic Modeling

Algorithmic modeling is about the definition of application-specific algorithms such as those for cell phone receivers, encryption, video, and many forms of digital signal processing. Refining in a software environment provides a much friendlier environment for debug compared to RTL debug. In addition, a software model within the context of an ESL methodology provides the architect a platform for asking and answering a series of questions such as:

- How do we want to specify this for the hardware or software implementers?
- How sloppy can we get with our arithmetic and have the algorithm still work?
- Can I get away with a software-only solution?
- Can I use a lower clock rate for my processor (with the resulting power savings) if I implement this algorithm with a hardware co-processor?

After completing the design of the actual algorithm (getting it to work), the algorithm architect usually refines the algorithm from a floating-point implementation to a fixed-point (supported by SystemC) implementation (for hardware implementation). This architect also partitions the algorithm between hardware and software. An architect with broad experience will understand the different tradeoffs between software and hardware and will shape the algorithm implementation to be very implementable in the chosen space (for instance minimizing memory use for hardware implementation).

This activity is usually performed in SystemC, C/C++ or in MATLAB, or other commercial tools. This work is usually augmented with libraries for a particular application space such as a fixed-point library, a digital signal processing library, and others.

1.3.3.2 Architectural Modeling

Architectural modeling is about hardware and software partitioning, subsystem partitioning. It is also about bus performance analysis, and other initial refinements based on a power management concept. This modeling also includes existing intellectual property and designs as well as other product-based technical and management critical parameters and trade-offs. Some of the questions asked during this activity are:

- Is there enough performance in the bus architecture to implement our algorithm in hardware and realize performance gains?
- Is the bus performance adequate? Will we require multiple busses or a multi-tiered bus concept?
- Is the arbitration scheme sufficient?
- Is the estimated size and cost within our product goals?
- Would a different processor help performance? How would a lower clock rate or less capable processor affect performance?

This activity can be performed using SystemC or another language that supports modeling or simulation concurrency. In the past, some teams have used C or C++. These teams have been forced to develop a custom simulation kernel to support concurrency.

Many times for this activity, non-critical portions of the system are modeled using "traffic generators". These generators are based on an estimate of the bus traffic generated by the non-critical elements. As the model is refined, the blocks are refined to add the details required to start running software. There are several vendors of EDA tools that have offerings that can help accelerate this activity.

1.3.3.3 Virtual Software Development Platform

The Virtual software development platform allows very early development of system software. During this activity, the following questions are sometimes asked and answered:

- Does the hardware have the advertised features?
- Are the control and status registers supplied to the software sufficient? Does the software have the necessary control and observably to meet the customer requirements?
- Can the software meet the software timing budget with the current architecture?

SystemC is the language of choice for this set of activities. SystemC provides the necessary simulation features (simulation concurrency), ability to easily integrate with C and C++, and simulation performance.

The ability to easily integrate with C and C++ lets engineers wrap the instruction set simulators (ISS) within the model. In addition, early in the development process, C or C++ code can be wrapped and executed on the host modeling processor (versus the ISS). We call this technique direct execution. Even as the software is refined and OS calls are required, additional techniques can be used to enable direct execution.

1.3.3.4 Hardware Refinement

Depending on the nature of the system, the hardware refinement activities can be focused on the creation of an executable specification to unambiguously specify the hardware. This specification accelerates hardware development. Alternatively, activities can be focused on refinement for behavioral synthesis with even greater schedule and productivity improvements. Some of the questions asked during this set of activities are:

- What is the required gate count or size for this function?
- What is the expected latency and clock speed?
- What additional control and status is easy and inexpensive to provide to software?
- What is the estimated power consumption of this block?

Refining the specification for hardware designers eliminates potential overdesign. Many times during the development process, an RTL designer will ask an architect specific questions where the answer will significantly (or even slightly) affect performance. When the architect is asked about option A or option B, he or she will answer yes or both because of the absence of information, thus adding complexity. With a model that can be refined, the architect can model the affect of A and B and possibly come back with the answer that neither option helps. The refined model saves design time, verification time, and silicon area.

1.3.3.5 Functional and Architectural Verification

Traditional functional verification of hardware involves verifying the hardware to a functional specification. Unfortunately, as systems become more complex, the functional specs can be measured in feet (not inches) of paper. Obviously, any process requiring a natural language like English (or German, Chinese, French, etc.) is very susceptible to error (just check out the errata list for the first edition of this book).

We have added a new term (at least for us): architectural verification. The goal of this activity is to answer a few big questions:

- Does this system architecture meet the customer requirements?
- Does this system architecture work for all customer use cases?

We have purposely used the term, system architecture, because this involves integrating at least the lowest level software and a hardware model. The authors have a background in computer system design and have previously lived by the maxim "it isn't verified until the software runs." Since almost all of today's complex electronic systems have a significant component of "computer system," we have reverted to this maxim from our "youth" by emphasizing architectural verification.

Functional verification is also enhanced through the early availability of a target for testbench development and a "reference model" for newer functional verification methodologies such the Open Verification Methodology (OVM) and the methodology originally defined in the Verification Methodology Manual for SystemVerilog (VMM).

Verification teams are always complaining that they are not allowed to start until too late. Some of the reason for this trend is that management is reluctant to assign resources to project activities that will not show results or progress until much later in the project. The TLM modeling activity allows a target for early development. TLM has the added benefit of reuse of some portions of the "testbench" for the system-level TLM model and for later functional verification with the RTL.

Many of the newer verification methodologies such as OVM and VMM make heavy use of constrained-random test stimulus generation. These methodologies require the development of a "reference model" to check the results of the "randomized inputs." Given a well thought out methodology, the TLM model or a portion of the TLM model can be used by the functional testbench to check the RTL results.

1.4 A Language for ESL and TLM: SystemC

ESL and TLM impose a set of requirements upon a language. Some of these requirements are:

- Abstraction spans several levels
- Standard and open language
- Common skill set
- Proper simulation performance and features
- Productivity tool support
- Supports TLM concepts

We hope that by the end of this section we will have made the case that SystemC is the best modeling language available today, and that it is the language to launch the adoption of ESL and TLM modeling.

1.4.1 Language Comparisons and Levels of Abstraction

Strictly speaking, SystemC is not a language. SystemC is a class library within a well-established language, C++. SystemC is not a panacea that will solve every design productivity issue. However, when SystemC is coupled with the SystemC Verification Library, it does provide in one language many of the characteristics relevant to system design and modeling tasks that are missing or scattered among the other languages. In addition, SystemC provides a common language for software and hardware, C++.

Several languages have emerged to address the various aspects of system design. Although Java has proven its value, C/C++ is predominately used today for embedded system software (at least the lower software levels). The hardware description languages, VHDL and Verilog, are used for simulating and synthesizing digital circuits. Vera, *e*, and recently SystemVerilog are the languages of choice for functional verification of complex application-specific integrated circuits (ASICs). SystemVerilog is a new language that evolved from the Verilog language to address many hardware-oriented system design issues. MATLAB and several other tools and languages such as Signal Processing Workbench (SPW[4]) and CoCentric® System Studio[5] are widely used for capturing system requirements and developing signal processing algorithms.

Figure 1.8 highlights the application of these and other system design languages. Each language occasionally finds use outside its primary domain, as the overlaps in the figure illustrate.

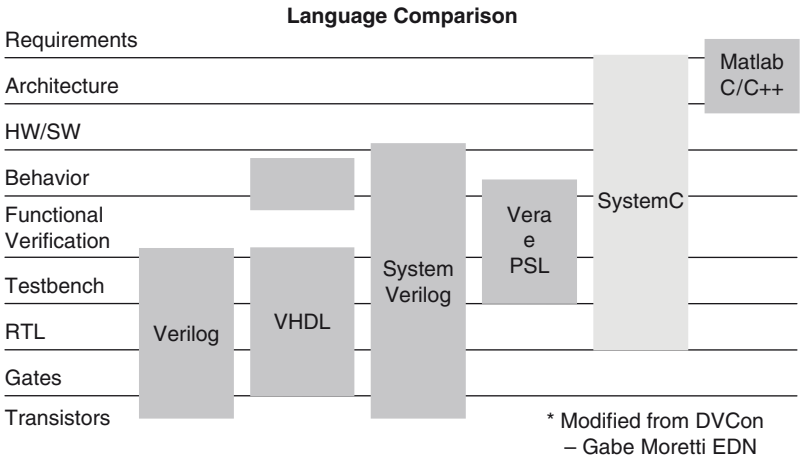

Fig. 1.8 Use of languages

[4]SPW is available from CoWare, Inc. (www.coware.com).

[5]CoCentric System Studio is available from Synopsys, Incorporated (www.synopsys.com).

1.4.2 SystemC: IEEE 1666

When a language is standard and open to the industry, then a wide array of benefits are available. These benefits include access to a set of commercial and freeware-based tools and support. SystemC was honored as a standard in December of 2006 when it was approved as IEEE 16666.

The Open SystemC Initiative (OSCI) provided much pre-standardization work to refine the SystemC language. Today, the OSCI continues this effort. The most notable effort is the standardization of a set of TLM interfaces.

OSCI also provides a proof of concept implementation of the IEEE 1666 standard. It is provided freely from the OSCI website at www.systemc.org.

1.4.3 Common Skill Set

SystemC is not yet a common skill set among all system engineers whether they be of a hardware or software orientation. However, SystemC is based on C++ and object-oriented techniques, which are a common skill set to all recent graduates of leading engineering schools. Many older engineers have also upgraded themselves by learning C++ or Java and the corresponding object-oriented techniques and thinking. These are the skills that enable a great SystemC modeler.

1.4.4 Proper Simulation Performance and Features

Obviously models need to execute swiftly. Not so obviously to some, the language needs to support concurrent simulation (we will talk about this extensively later in the book). The language must also support a series of additional productivity features. These features include the ability to manage the model complexity through hierarchy, debug features, and a list of other features discussed throughout this book.

Any model needs to execute relatively swiftly. A model is never fast enough, so what is fast enough. One can look at the requirements for model speed when used for common ESL purposes like early software development. To keep software developers from extreme frustration, the model needs to boot its operating system in just a couple of minutes or close to “real time”.

No modeling language can make up for inefficient modeling practices. The compiled nature of SystemC coupled with the availability of very good C++ compilers complemented with good modeling practices can produce a model that meets the speed requirements for all but the very largest systems.

As the reader may have guessed by now, developing a model is different than developing an application program. The difference is that in an ESL model, the

code modules need to appear as if they are executing in parallel or concurrently. When a digital system is running, then many, many computations are running at the same time or are running concurrently. SystemC provides a simulation kernel to give this illusion of concurrency. This feature, as well as several others, helps manage the complexity of the model. These features are discussed in detail in subsequent chapters.

1.4.5 Productivity Tool Support

Since SystemC is based on C++, there are a wealth of productivity tools and application-specific libraries that are available. In addition, SystemC is supported by a large and growing ecosystem of commercial suppliers.

Many of the popular C++ tools are freely available under various open source licenses. Some of these tools include integrated development environments (IDEs), performance analysis tools, and lint tools to illustrate just a few. There are many more tools available from leading software tool vendors for a small fee. These tools come with varying degrees of support.

Many of the groups using SystemC are large organizations that require ESL-specific productivity tools and technology as well as significant support. The three largest electronic design automation (EDA) vendors and an extensive list of smaller EDA companies now support SystemC. The last time we counted, the list was over 40 companies and growing.

The tools range from co-simulation with their HDL simulators to behavioral synthesis tools.

Possibly the biggest productivity boost is the availability of application libraries on the web in C or C++. The availability of graphic algorithms, DSP algorithms, and a plethora of other application libraries make the writing of initial models very easy. An extreme example is the use of an H.264 algorithm freely available on the web that is matched with about 20 lines of SystemC code to produce a graphics model in a matter of hours for an initial system model.

1.4.6 TLM Concept Support

Lastly, a language for ESL model development needs to support TLM concepts. It must support the easy substitution of one communication implementation with another without changing the interface to that implementation. As we progress through this book, we will show that C++ and the concept of interfaces implemented through a class of pure virtual functions coupled with a few coding styles enables TLM concepts efficiently.

1.5 Conclusion

We hope that we have motivated you to not only read our book, but also to study it and apply it to the examples provided at our web site. You will then be equipped to charge into the brave new world of ESL and TLM modeling and to bring about significant changes in your organization, company, and the industry.