

Design for Verification in System-level Models and RTL

Anmol Mathur Venkat Krishnaswamy

Calypto Design Systems, Inc

Abstract

It has long been the practice to create models in C or C++ for architectural studies, software prototyping and RTL verification in the design of Systems-on-Chip (SoC). It is often the case that by the end of a design project, multiple C models exist for different uses. Clearly, this leads to wasted effort on the part of model developers, and creates risk of functional divergence across models. In this paper we present some guidelines for system-level modeling and RTL design to allow for efficiently leveraging the system-level model for RTL verification via simulation based techniques, as well as via sequential equivalence checking. The paper presents the challenges of keeping system-level models and RTL synchronized from a functional perspective and presents some techniques for overcoming these challenges.

Categories and Subject Descriptors

B.5.2 [RTL Implementation] Design Aids

General Terms

Reliability, Verification.

Keywords

System-level models, equivalence checking, simulation, RTL models

1 Introduction

C/C++ and SystemC models are created by SoC design teams to serve different purposes during design including architectural exploration, software prototyping, supplying customers with executable prototypes, and verification of RTL models [1]. In this paper we will refer to these models as system-level models (SLM). Since these tasks to which the models are put differ widely in purpose, the SLMs created for each of these can vary in abstraction level across the axes of functional accuracy and detail in communication.

SLMs used for architectural exploration vary in functional accuracy depending on the application domain. In networking applications, architects are interested in sizing resources to sustain peak and average network traffic while obeying hardware constraints. In such situations, it is common to use abstract mathematical or stochastic models such as queueing systems. Such models cannot be considered “functionally accurate”, and have no utility beyond the specific task for which they are designed.

Architectural models written for signal processing or image processing applications tend to be functionally accurate in the sense that they accept an accurate input set and simulate the behavior of the device on that input. The model is used to gain confidence in the algorithm being implemented, and possibly to decide on the optimal word widths to support the desired bit error rates. While these models are necessarily functionally accurate, they need not model the timing relations to match hardware.

Models written for the purpose of enabling early software development or for shipment as an executable prototype are by definition functionally accurate. Such models must present the same register level interface to the software as the actual device would in hardware. However, the model need not be “cycle accurate” with the hardware, since it only needs ensure that operations initiated in software occur in the same order as they would in hardware. In order to achieve simulation speeds sufficient to be productive software development platforms, communication is usually abstracted. RTL verification requires accurate functional representation. From a timing perspective, the models must at least be cycle approximate. Thus, the verification models can largely be derived from other functionally accurate models such as those described in the preceding paragraphs, but with added timing detail.

Traditionally, verification of RTL models against “golden” SLMs has been done through the use of co-simulation. Recent advances in sequential equivalence checking technology [3][4], have made it possible to formally verify whether a SLM is functionally equivalent to the RTL at the block level. In this paper we propose that the most effective way to leverage SLMs for RTL verification is to keep the verification goals in mind while designing both the system-level models and RTL. We identify the key attributes of system-level models that are needed in order to ensure that they can be effectively used for RTL verification. We show how to support writing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...\$5.00.

SLMs usable for sequential equivalence checking while maximizing code reuse across the different models developed throughout the design process. Reuse of SLMs not only limits wasted effort, but also significantly improves the quality of verification. Code reused over several models is likely to be of high quality in terms of functional fidelity. This is particularly true because the workloads run on models designed for architectural exploration or software prototyping are likely to be both realistic and as comprehensive as possible.

2 Verification Methodology Between System-level Models and RTL

Most design teams use simulation or co-simulation based techniques for using SLMs to verify RTL. The SLM is treated as the “golden” functional reference model that generates the expected outputs for a given input stimulus. Since the SLM is more abstract than the RTL and does not model a lot of the micro-architectural details, specially if it is untimed or partially timed, the SLM simulates several orders of magnitude faster (typically 10x to 1000x) than the RTL model. As such the typical process many design teams use for RTL verification using SLMs is the following:

1. Validate the SLM by running actual applications on it. In the case of a SLM for a graphics chip for example, the SLM can be plugged into a simulation environment that stimulates it with actual images and the output of the SLM is actually rendered and checked for issues with the quality of the generated images. This is possible because the SLM simulates fast enough that an almost real-time simulation using it can be done in an actual system. Such validation gives a high level of confidence in the functional correctness of the SLM.
2. If the SLM is not cycle accurate with respect to the RTL, the stimulus used for exercising the SLM cannot be directly used as RTL stimulus. Typically, design/verification teams write adapters that convert the SLM stimulus to RTL stimulus. Once such stimulus adapters have been written, the actual RTL can be instantiated in another top-level hierarchy that places transactors at the RTL inputs and outputs so that the SLM input stimulus can be used for RTL simulation. The RTL with transactors is called the “wrapped-RTL”. There are two possible strategies for using the SLM as a reference model for RTL verification:
 - a. Independently simulate the SLM and wrapped-RTL using the same stimulus. The SLM generates the expected

outputs. These are compared against the outputs of the wrapped-RTL. Temporal differences between when the SLM and wrapped-RTL produce outputs means that the procedure that compares the SLM outputs with RTL outputs needs to account for the timing differences. We will discuss the nature and causes of timing differences between the SLM and RTL in Section 3.

- b. Replace a block of the SLM with an wrapped-RTL corresponding to that SLM block and co-simulate the wrapped-RTL and the remaining SLM blocks. The results of the co-simulation can be compared to those of simulating the full SLM.

In recent years, sequential equivalence checking has come up as a viable alternative to compare a SLM block against the corresponding RTL block. Sequential equivalence checking requires the specification of how the inputs map between the SLM and RTL and specification of when to check the outputs. Typically, this requires specifying a repeating computational transaction in the SLM and the corresponding transaction in the RTL model and the sequential equivalence checker then ensures that from the given initial states, the SLM and RTL produce matching outputs when stimulated in accordance with the input mappings specified in the SLM and RTL transactions. Sequential equivalence checking is very effective at quickly finding discrepancies between SLM and RTL models and getting them in a functionally consistent state, without having to write testbenches at the block level. This is a very effective way to transfer the high level of confidence in the functional correctness of the SLM to the RTL blocks.

3 Challenges in Keeping SLMs Consistent with RTL

In order to effectively use SLMs as a functional reference model for RTL verification, either using simulation or using sequential equivalence checking, it is important to follow certain guidelines in the design of the SLM and RTL models. We propose that keeping these “design for verification guidelines” in mind upfront in the design of the models is very effective in creating models that can be used for both simulation-based and sequential equivalence checking based verification. A similar theme was discussed in many papers in [5].

There are two key aspects of keeping SLMs and RTL consistent and these will be separately addressed in this paper:

- Computational accuracy between SLM and RTL
- Communication/Interface timing accuracy between SLM and RTL

Each of the above axes expose their own challenges in the design of the SLM and will be discussed in this section.

3.1 Sources of Computational Inconsistency

The fundamental requirement on a SLM for it to be a useful functional reference model is that it should be bit-accurate with respect to the RTL model. A SLM is said to be bit-accurate with respect to an RTL if for the same values of inputs they produce exactly the same output values. It does not necessarily imply that the bit-widths of the inputs and outputs in the SLM and RTL be exactly the same for the corresponding inputs and outputs.

3.1.1 SLM and RTL Model Fixed-point/integer Computations

The main source of computational discrepancy between SLM and RTL is the difference in data types used in SLM and RTL, both at the interfaces of the models and internally in the computation. RTL designers attempt to reduce the widths of arithmetic operators in datapaths in order to optimize the area, power and delay of the final implementation. As such, RTL models always use bit-vectors of varying widths to represent values both at the interfaces and internally in the computation. On the other hand, C/C++ models are restricted by the language to only use the standard data types available in the language. As such C/C++-based SLMs typically use data types such as `int`, `short int` and `long long` (along with their signed and unsigned versions). These data types have fixed widths based on the C/C++ semantics and the machine platform being used (typically 8-bits for `char`, 32 bit for `int` and 64 bit for `long long`).

Ensuring that the values computed in such C/C++ models using such `int`-based data types are the same as those computed in RTL models with custom-sized bit-vectors is challenging for the following reasons:

- Finite bit-vector arithmetic has many non-intuitive rules: One instance of this is that addition is not associative

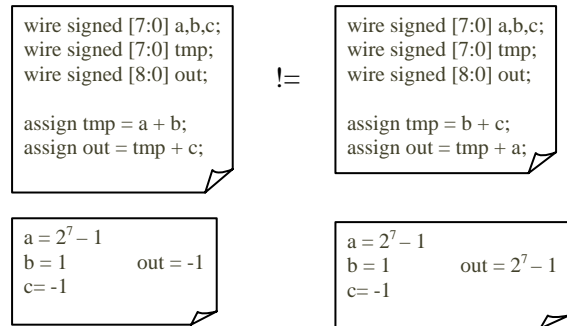


Fig 1. Addition is non-associative in finite precision arithmetic.

Since the reason for the non-associativity is the overflow due to the variable “`tmp`” not being wide enough, such effects can be easily masked in a C/C++ model where all the values are represented as `int` and the overflow does not happen.

- Large bit-vectors are not natively supported in C/C++ : this results in SLM designers having to create their own libraries for bit-vector operations beyond 64-bit values. This can often be a source of discrepancy, since the semantics of sign extension and arithmetic operators in standard HDLs such as Verilog and VHDL need to be faithfully captured in the new bit-vector library or the differences between the HDL semantics and the new library need to be clearly understood by the creators of the SLM.
- Lack of bit-vector operations in C/C++ such as selecting a few bits from a bit-vector or concatenating two bit vectors, means that SLM models written in C/C++ often have to use mask and shift to implement such operations:

```
int x, y, mask;
mask = 0x00ff0000;
y = x & mask;
```

to select the bits [23:16] from the `int x`.

SystemC supports bit-vectors of custom widths via the `sc_bv`, `sc_lv`, `sc_int` and `sc_bigint` types. It also defines the standard arithmetic operators on `sc_int` and `sc_bigint` types and provides for range extraction and concatenation. As such, SystemC has a richer set of data types and allows for SLMs that can be computationally closer to the RTL. Another point that should be noted is that even if a design team does not use SystemC to model concurrency, clocks and time in their SLM, they can still use the SystemC data types for representing computations

and reduce the possibility of creating a SLM that is not bit-accurate.

3.1.2 Floating Point Computations

Typically, floating point computations in a SLM use the standard `float` or `double` data types in C/C++ and SystemC. RTL models on the other hand need to implement these operations explicitly via manipulation of the mantissa and exponent of a floating point number represented using bit-vectors. While the system-level languages use the IEEE floating point standard to manipulate their native floating point data types, RTL designers often do not implement the full IEEE standard. The reason is that implementing the full IEEE standard and handling all its corner case combinations (normalized numbers, denormalized numbers, NaN (not-a-number) and infinity) can be prohibitively costly in hardware. Further, often RTL designers know that constraints on the input values in their design will result in such corner cases never being hit.

Such differences between the SLM and RTL can result in the SLM not being bit-accurate. If the SLM and RTL are not bit-accurate then, the comparison of output values needs to account for the difference and cannot be exact. While this can be achieved in simulation-based comparison by allowing for some error between the output values of the SLM and RTL, it is harder for sequential equivalence checking to account for such differences. The most effective technique to apply sequential equivalence checking to a (SLM, RTL) design pair with such differences, is to constrain the input space under which the sequential equivalence checker compares the models such that the differences do not show up.

3.2 Sources of Interface Timing Inconsistency

How much timing needs to be modeled in the SLM depends on its intended use. A purely algorithmic model is often untimed. It is a pure C/C++ function that takes in a set of inputs and computes outputs. Such models are very fast to simulate since they have no concurrent processes or events to be dealt with. However, these models do not model any micro-architectural aspects of the hardware being designed and hence cannot be used for making tradeoffs between different micro-architectural alternatives. Even a SLM model used for software/firmware development requires a model that captures the programmer-visible state of the hardware and often needs to model the interface timing. Design teams often use a custom simulation kernel to model timing and events in such models. Recently, SystemC has become

the language of choice for such models since it provides notions like clocks, clocked threads, events and hierarchy for modeling hardware micro-architecture.

Even SLMs that capture timing, often have discrepancies with respect to the final RTL due to the following reasons:

- The SLM does not fully capture the micro-architecture implemented in the RTL. This is often intentional since the SLM model is intended to be more abstract than the RTL and would not simulate fast enough if it has all the details of the RTL. For example, the SLM may model a memory simply as a static array in C (accessed and written without any delay), while the RTL implements a real memory that has a delay of one clock cycle for memory reads. The RTL may even have a hierarchical memory with a cache, where the latency of a memory read is a function of the state of the cache. Such cases make it very hard to get the cycle timing to match exactly between the SLM and the RTL.
- At the block level, the SLM and RTL often have different interfaces. Typically, SLMs have very parallel interfaces while RTL has serial interfaces due to resource constraints. For example, the SLM of an image processing block may read in the entire image as a single array of pixels while the RTL reads it as a stream of pixels. Again, the SLM has parallel interfaces to speed up the simulation speed of the SLM. Such intentional interface differences between the SLM and RTL need to be accounted for appropriately either in the interface transactors in simulation or via appropriate interface mappings in sequential equivalence checking.
- RTL models often have variability in input to output latency due to the presence of external or internal stall conditions under which operations in the design are stopped. Such conditions are typically not modeled in the SLM. These can result in the interface timing between the SLM and RTL not being cycle accurate. Sometimes such behavior in the RTL can mean that the order in which the RTL produces outputs may be different than the order in which SLM produces the corresponding outputs. Out-of-order output generation can result in complicated transactors being needed to compare the outputs of the SLM and RTL in both simulation as well as sequential equivalence checking.

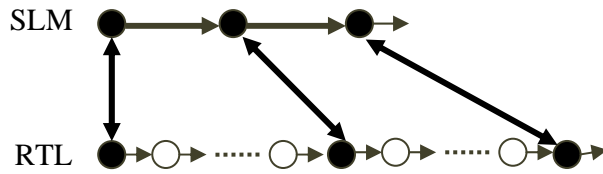


Fig 2. Timing alignment between SLM and RTL can be non-trivial.

4 Some Recommendations for Design-for-Verification

In order to facilitate the use of the SLM as a reference model for functional verification of RTL, the design teams responsible for the SLM and RTL models need to keep the causes of computational and temporal/interface inconsistency raised in Section 3 in mind and attempt to eliminate as many of them as possible. Apart from these, there are some general recommendations that enable design teams to better keep their SLM and RTL models in synch and these will be discussed in this section. Also, if certain guidelines are followed in the design of the SLM, then using it for sequential equivalence checking of RTL or for automated generation of RTL via behavioral synthesis tools becomes easier. We will also discuss these guidelines in this section.

4.1 Coordination between SLM and RTL Teams

In order to leverage SLMs for RTL verification, the teams designing the SLM and RTL need to work together to ensure consistency between the models. Further, both the models need to have clear ownership in the team and need to be constantly maintained and “alive” in the course of their use. If the SLM design team “throws a model over” to the RTL design team, then it is hard to effectively use the SLM. This is specially the case since veryoften simple changes to the SLM can make it much easier to check the consistency of the SLM and RTL models. Also, often bugs are found in the SLM when using simulation or sequential equivalence checking to compare it to the RTL and these need to be fixed to make further progress in the verification.

Further, it often makes sense to start keeping the SLM and RTL consistent early in the process of development of the two models rather than getting to this task only after both the models are ready and have been independently verified. This can start weeding out sources of inconsistency in the computational or interface aspects of the models early in the design process. Also, incremental runs of sequential equivalence checking between SLM and RTL are much more effective in terms

of run time and can help localize the source of any difference between the models quickly. If these runs are started late in the development of the SLM and RTL models, then inconsistencies between the models may have caused large temporal or computational divergence between the models requiring complicated transactors or long debug cycles to figure out the source of the inconsistency : error in SLM, error in RTL, error in transactors or missing constraints while performing the verification.

4.2 Design Partitioning

In order to constrain designs easily for verification using sequential equivalence checking, it is necessary to ensure that correspondences exist between algorithms (contained within C functions or C++ objects) and hardware module hierarchies. There should be clear functional boundaries both in the SLM and the RTL at blocks that will be equivalence checked or co-simulated. If the SLM is done using SystemC, then `SC_MODULE` construct provides a clean way to encapsulate the interface and functionality of a portion of the SLM model into a hierarchy. Partitioning the SLM and RTL consistently is very beneficial since the individual blocks of the SLM and RTL then have a one-to-one correspondence and cleanly defined interfaces where the timing alignment is well understood.

Clean and consistent design partitioning provides an opportunity to use sequential equivalence checking at the level of individual SLM/RTL blocks. It also allows for easier plug-and-play between SLM and RTL blocks. This can be leveraged to co-simulate SLM with a few RTL blocks plugged in (with appropriate transactors) for verification of the RTL models in the context of the rest of the system.

4.3 C/C++ Model Conditioning

In order to perform SLM to RTL equivalence checking using a sequential equivalence checker, the SLM must be written such that a hardware-like model can be inferred statically from the source by the tool. This requires the team creating the SLM to follow certain coding guidelines that allow for static analysis of the SLM.

4.3.1 Creation of algorithmic code with hardware intent

The use of certain programming constructs makes it impossible to statically determine the size of hardware structures that implement the functionality. Since sequential equivalence checking and behavioral synthesis tools need to be able to statically analyze the SLM, it is necessary to avoid such practices in order to leverage

these techniques. The following are some recommendations that can allow the SLM to be amenable to tools requiring static analysis of the SLM:

- Use statically sized arrays rather than pointers that are assigned memory allocated dynamically using `new` or `malloc`. This is typically a simple design guideline and typically has no impact on the simulation speed or expressiveness of the model.
- Explicit use of memories rather than using pointer aliasing. Pointer aliasing is often used to have multiple variables point to the same physical memory in software. Using an explicit memory construct instead of such aliasing can make the micro-architectural intent in the design more explicit. This also makes the design more amenable to static analysis.
- Using static loop bounds with conditional exits. Loops in C/C++/SystemC models need to be unrolled for static analysis. Thus, data-dependent or variable loop bounds are a problem for tools performing static analysis of the SLM. Converting such loop bounds to be static (upper bound of the possible values for the loop bound) with a conditional exit from the loop allows the SLM to be amenable to static analysis.

These recommendations are described in more detail in [2].

In addition, the SLM should follow the following guidelines if it represents an untimed algorithm:

- The algorithm should be coded in an untimed, single threaded manner
- The algorithm should have a single point of entry i.e. one well defined top level function or method call that may in turn invoke other functions
- The algorithm should be self contained within the source files supplied for compilation

These rules make it easy to write wrappers that impose a hardware-like structure around the algorithm, as we shall describe below.

4.4 Orthogonal Communication and Computation

Typically, the system being modeled is assumed to be composed of multiple functional (computational) blocks that communicate with one another. In this methodology, correspondences are maintained between these functional blocks and hierarchical boundaries in the RTL. The level of detail in communication across functional blocks is a first order determinant of the simulation speed achieved by the system model. The level in communication detail

increases from architectural models to software prototyping models to verification models. However, all of these models are functionally accurate. It follows that a possible method of leveraging the functional or computational core or kernel across models would be to split out computational and communication functionality.

Timed communication is often performed through the use of an event based or clocked synchronous simulation kernel. There is a wide variety of simulation kernel API's in use ranging from home grown varieties to standardized and publicly available kernels such as SystemC (reference to SystemC). Clear separation between the computational and communication aspects of the SLM allows for easier refinement of the communication protocol if needed to make the interface timing more closely aligned with that of an RTL model. This is also the primary recommendation of transaction-based modeling that is becoming common in many design teams [1].

5 Conclusions

Keeping the potential sources of computational and communication discrepancies in mind while designing the SLM and RTL can go a long way in ensuring that the SLM can be effectively used to verify the RTL model. In addition, by following some simple guidelines for the design of the SLM in C/C++ or SystemC, design teams can ensure that their SLMs are useful beyond just simulation. This can result in significant reduction in the time to attain a high quality RTL model that is crucial for functionally correct first silicon.

6 References

- [1] Frank Ghenassia (ed), "Transaction Level Modeling with SystemC TLM Concepts and Applications for Embedded Systems", Springer, 2005.
- [2] P. Georgelin and V. Krishnaswamy, "Towards a C++-based Design Methodology Facilitating Sequential Equivalence Checking", pp 93-96, Proceeding of DAC, 2006.
- [3] Calypto Design Systems, Inc., <http://www.calypto.com>
- [4] Z. Khasidashvili, M. Skaba, D. Kaiss, Z. Hann, "Theoretical Framework for Compositional Sequential Hardware Equivalence Verification in Presence of Design Constraints", pp. 58-65, Proceedings of ICCAD, 2004.
- [5] Design to Enable Verification, Special Session, ICCAD 2006.