# Seamless Refinement from Transaction Level to RTL Using SystemVerilog Interfaces

Jonathan Bromley

Doulos Ltd, Ringwood, England

jonathan.bromley@doulos.com

## ABSTRACT

The need for high-performance functional simulation of large system-on-chip designs, especially when the behavior of software is to be modeled in simulation, has given rise to a range of behavioral modeling styles that are often collectively known as *transaction level modeling* (TLM). A distinctive characteristic of TL models is the way in which they communicate with one another by means of subprogram calls. This form of module-to-module communication offers orders of magnitude performance improvement over the event-based communication used between modules in a traditional HDL simulation. Furthermore, because they operate at a much higher level of abstraction, TL models can be created and simulated long before synthesizable RTL code is available for testing.

TL modeling is often implemented in a general-purpose object-oriented programming language such as C++, and the existence of a widely accepted standard library (SystemC) has naturally increased the popularity of this approach. However, refinement from a SystemC design to synthesizable RTL code is unlikely to be automatic. Manual refinement is needed, followed by verification to check that the refinement was error-free. Unfortunately the resulting RTL design module is unlikely to operate successfully within the original TLM test environment, and so the successive steps in refinement cannot readily be verified within a common verification environment.

Our example exploits SystemVerilog *interface*s to bridge this gap between RTL and TL models, enabling seamless interoperation between them in any reasonable combination whilst preserving the best possible simulation performance at each step in the refinement process. The paper shows how interfaces can allow TL and RTL models to work together, with any necessary adapter functionality encapsulated in the interface. An interface's subprogram *import/export* mechanism provides a means for transaction-level models to work with the interface, while other connected modules can use a traditional Verilog signal-level connection.

Using the VCS$^{TM}$ simulator, we will show a fully TL model constructed entirely using SystemVerilog, with some of the TL components then progressively replaced by conventional RTL modules without affecting other TL components that continue to participate in the simulation. In this way, seamless refinement is achieved in the sense that refining one module in the system from TL to RTL does not entail rebuilding of other parts of the system model.

The paper also considers how to combine the same interface mechanism with SystemVerilog's Direct Programming Interface (DPI) to integrate SystemC models into an otherwise SystemVerilog-based environment.

## Table of Contents

## Table of Code Examples

## Table of Figures

## 1. Introduction

This paper discusses motivation and techniques for a modeling style that allows for the construction of simulation models in a mixture of transaction level (TLM) and register transfer level (RTL) styles. In particular, the approach outlined here makes it easy to change individual components of the system model from one style to the other without disrupting other parts of the model.

Sections 2 and 3 outline the problem of verification of refinement steps, and discuss some existing approaches to achieving verifiable refinement from behavioral to RTL modeling style.

Section 4 describes a small but non-trivial example system that will be used to exemplify the techniques described later.

Section 5 is a tutorial discussion of the fundamental SystemVerilog language feature (subprogram import/export through interface modports) that enables the proposed approach.

Sections 6 to 8 examine in detail the SystemVerilog implementation of a whole-system model. The system is first modeled at the transaction level (TL). Individual subsystems are then progressively refined to RTL without disturbing other parts of the model, permitting an unchanged verification environment to be used to perform functional tests at each stage.

Section 9 examines performance considerations, presenting empirical results from simulations using VCS as well as theoretical expectations

Section 10 briefly indicates how the same modeling and interfacing techniques could be used to integrate C or SystemC models with an otherwise SystemVerilog model.

Section 11 summarizes the conclusions of the paper.

## 2. The Unbroken Chain: Trustworthy Progressive Refinement

The process of creating a product – and, in particular, a chip design - can be thought of as a chain of steps or transformations. At each step, a description of the product is transformed into a different description that is better suited to some purpose. For example, a requirement specification agreed with a customer must be transformed into a functional specification in order to make it possible for engineers to implement it. Similarly, an RTL description of a digital design must be transformed into a netlist of technology primitives so that work can begin on device layout. In almost every case, such steps take a description and transform it to a lower level of abstraction: algorithmic to RTL, RTL to netlist, netlist to transistors, and so on.

Some refinement steps are largely manual, while other steps are highly automated by EDA tools. Automated steps such as synthesis of RTL to netlist are often considered inherently trustworthy, but even for these there is the possibility of tool bugs or incorrect use of the tool. Manual steps, such as the creation of an RTL design from a functional specification, are inevitably error-prone even with the best engineering teams. There is the ever-present risk of misinterpretation of a specification or other document, and of course there is the risk of human error in any manual process. Consequently every step in the refinement process must be checked in some way.

Some transformation steps are so widely used, with tools that have been so extensively debugged, that they can be considered inherently trustworthy. Conventional high-level language compilers typically fall into this category; few programmers (except those working in exceptionally safety-conscious industries such as aerospace and nuclear power) feel a need to check the machine-code output from their compilers for equivalence to their high-level language source code. However, most of the refinement steps commonly found in a digital design flow are much less likely to be free of errors, and must be verified.

The most obvious approach to verification is to take the original and the transformed design description and compare them for equivalence. Checking that two descriptions of a design are equivalent is of course much more tractable if both descriptions are machine-readable, or perhaps captured in some formal or mathematical notation. For example, an RTL design and a gate-level netlist synthesized from it can be compared by simulating both with the same input stimulus, or by the use of a formal equivalence-checking tool. In this way the RTL-to-netlist synthesis transformation – already one of the most trustworthy steps in a typical design flow – can be thoroughly audited by an independent check.

An alternative approach is to perform two independent transformations of a source description, creating two different output descriptions that can nevertheless be compared against one another. This technique is particularly valuable when the input description is a natural-language document, such as a specification, that is not directly amenable to automated processing. Typical examples of this approach might include the creation of both a behavioral model and an RTL model from a specification. Both models can be simulated in a testbench and the outputs checked for equivalence, or the behavioral model can be used as a reference in a self-checking verification environment. If the two transformed descriptions are created using different design styles and by separate teams, the risk of an identical error appearing in both is much reduced and comparison of the two descriptions should make it possible to detect and, ultimately, to locate any errors. For this approach to work successfully it is necessary for a single tool to be able to process and compare both descriptions. For example, a set of assertions written using a language such as PSL [1] or SVA [2] can capture some aspects of a design's expected behavior, and formal model-checking tools can be used to prove the conformance (or otherwise) of an RTL description to the expectations described by those assertions. Another example is provided by the use of simulation to compare the outputs of a behavioral model and an RTL design when simulated with identical input stimulus. Whenever functional verification is required, there is a need for an execution environment in which "before" and "after" models can be compared. The environment must be able to support both forms of model.

The main focus of this paper is the comparison of a high-level or abstract reference model against an RTL implementation, and verifying that the two models have equivalent behaviors. It is common for these two styles of model to be created not only in different programming languages but also using completely different modeling styles. In particular, high-level models are often written using a *transaction-level modeling* (TLM) style. TL models not only operate in a very different way than RTL; they also have traditionally been written in C++, taking advantage of the SystemC modeling environment. Consequently, although the two models can each be simulated alone in an appropriate environment, it is less easy to see how they can be executed together and compared in a common environment.

## 3. Alternative Approaches

Of course, the issues described in the preceding section are not novel and they have already been addressed in various ways. Some of the currently available techniques are indicated here. The remainder of the paper considers an alternative approach, entirely coded using SystemVerilog, which offers a different and (in the author's opinion) attractive compromise between ease of use and expressiveness.

### 3.1 Behavioral models written in a traditional HDL

It is entirely possible to write high-level behavioral models in Verilog or VHDL, and of course such models can easily be simulated in a common environment alongside RTL models. However, such HDLs present significant limitations when attempting to create flexible, transaction-level communication between modules. VHDL does not support procedural entry points to a module (i.e. cross-module subprogram call). Verilog does provide cross-module subprogram call, but has an insufficiently expressive data type system for serious TL modeling and, even more importantly, has no support for any kind of dynamically-allocated or dynamically-sized data objects. Neither language supports the object-oriented programming (OOP) facilities that are rightly expected by software engineers. These limitations make VHDL and Verilog ill-suited for serious efforts at writing TL models.

### 3.2 Converting RTL to SystemC

Some commercially available tools can take an RTL description, typically written in synthesizable Verilog, and automatically translate it to a SystemC model with equivalent functionality. This model can then be simulated in a SystemC modeling environment alongside the SystemC TL model with which it is to be compared [3].

### 3.3 TLM-to-RTL formal equivalence checking

The disparity in modeling style between typical RTL and TL models inevitably means that direct equivalence checking using formal techniques is unlikely to be tractable in general, although it may be applicable in some specific applications [4].

### 3.4 Synthesis of abstract models to RTL

A steady stream of tools intended to take abstract or behavioral descriptions and synthesize them to RTL or gate-level has been introduced to the market over the past few years. Such tools have met with patchy acceptance in the marketplace; for example the very interesting Behavioural Compiler tool [5] is no longer marketed. In some specialized areas, notably the synthesis of DSP designs, this approach has met with some success. A handful of "C-to-gates" tools is commercially available today and all of them find useful application in some areas, but today's mainstream digital design applications remain largely unaffected by this technology [6][7][8].

### 3.5 Mixed-language simulation

Digital simulation tools from several major vendors already offer the ability to integrate models in various languages, usually including at least Verilog, VHDL and C++/SystemC, into a common simulation. It is clear that this is a very powerful approach and tool vendors have made impressive progress in making such mixed-language simulation as straightforward as possible for the end user [9][10][11][12].

### 3.6 Separate execution

Probably the simplest solution for the problem of comparison of two disparate models is to simulate each in its own verification environment, and then to perform off-line comparison of the results using some comparison tool – either a conventional file-compare utility or some custom software. Whilst this approach has its place and is clearly a useful fallback, it seems clumsy and we do not consider it further here.

## 4. Example System Model

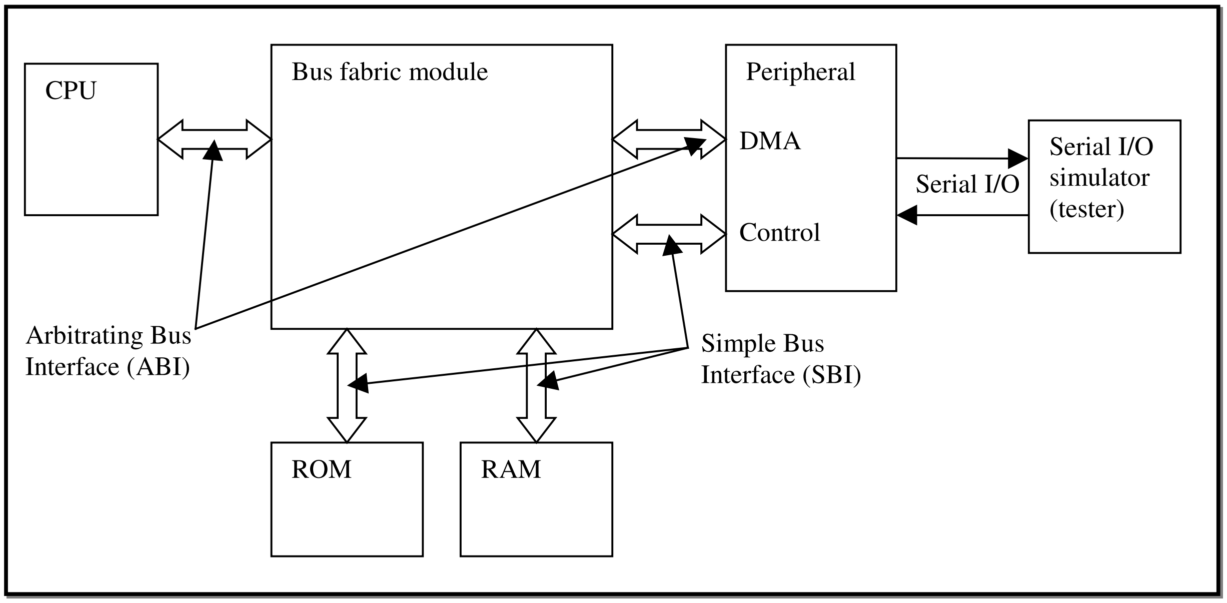
To provide a concrete embodiment of the ideas described in this paper we have chosen an example system that is very straightforward but has some interesting behaviors. A block diagram of the system is shown in Figure 1. It comprises five distinct functional blocks with various interfaces between them. These blocks are:

- a CPU model, which of course initiates transfers on its bus interface;
- two memory models, a ROM and a read/write memory (RAM), both of which act as slaves on their respective bus interfaces;
- a serial interface peripheral containing a slave-only interface, over which its control and status registers can be accessed, and a DMA controller which transfers data over a second bus interface;
- a bus fabric module, which arbitrates between the CPU and DMA controller for access to the slave devices, and directs such accesses to the appropriate slave device.

Components in the example system communicate using a parallel microprocessor-style bus, which appears in two forms in the system. The *simple bus interface* (SBI) is used to communicate between the bus fabric module and slave devices; it has a very simple fixed timing (slaves cannot introduce wait states into its transfers). The *arbitrating bus interface* (ABI) is used for communication between bus masters (CPU or DMA) and the bus fabric; it has a request/grant handshake, and also an additional control line used by the CPU to distinguish instruction fetches from other memory accesses.

The models and bus protocols in this system do not follow any industry standard arrangements, but are part of a system that has been developed as a tutorial example for use in training courses presented by the author's organization [13]. Although somewhat unrealistic in their detail, they reflect many of the concerns typically encountered in real microprocessor bus environments: multiple address spaces, and a bus fabric module that supports several point-to-point interconnects and arbitrates between CPU and DMA as they compete for bus bandwidth.

**Figure 1: Example system**


CPU
Bus fabric module
Peripheral
DMA
Control
Serial I/O
Serial I/O simulator (tester)
Arbitrating Bus Interface (ABI)
Simple Bus Interface (SBI)
ROM
RAM

## 5. Task Export/Import Using SystemVerilog Interfaces

Components in an RTL simulation communicate with one another using signals. An event (value-change) on a signal, caused by activity in one component, triggers the execution of activity in other components. This event-driven simulation offers accurate modeling of digital functionality, at the expense of a heavy performance cost. In a typical RTL simulation the processing of events dominates performance, often consuming much more CPU activity than the execution of behavioral code.

By contrast, transaction-level modeling (TLM) uses subprogram calls to communicate between components. A component initiating some activity calls a function; another component, the target of that activity, provides an implementation of the function. A single function call can give rise to a large amount of behavioral activity, dramatically reducing component-to-component communication overhead by comparison with an RTL simulation in which each value-change typically triggers only a small amount of behavioral activity.

The performance benefits of TLM are obviously increased if each function call can perform a large amount of activity. Consequently it is desirable for these functions to represent activity at the highest possible level of abstraction. Typical TLM function-call interfaces might represent the execution of a complete data transfer cycle on a microprocessor bus, or the movement of a data packet over a communications channel. Each such function call typically represents a block of activity that would span many clock cycles in an RTL simulation, and the TLM simulation therefore provides less accurate modeling of timing. At one extreme it is possible to write a model of the functionality of a system that takes no account of simulated time – an *untimed functional model* – whilst preserving the correct sequential ordering of different activities in the various components of the model.

Components in a transaction-level model, then, communicate by function call; the initiator calls a function in the target. However, concerns of modularity and component reuse dictate that this call should not be done directly. It is inappropriate for an initiator component to be hard-coded for details of the target. Instead, some kind of indirection is used. A typical mechanism, popularised by [14] and [15], uses the notion of *ports* and *exports*. An initiator is equipped with an instance of a suitable port object, and calls the appropriate method (function) in that port object. The target has an instance of a corresponding export object, and provides an implementation of the method specified in the export. The initiator is coded as if it calls a function in the port, and the target is coded as if the export calls its function. However, a *connection* mechanism allows ports and exports to be bound together so that the initiator's function call into its port is in fact redirected to the target's export, which in its turn redirects the function call to the real implementation in the target. Given appropriate support from the chosen programming language, this redirection can be achieved with negligible run-time overhead. In SystemVerilog the redirection is fundamentally achieved by putting the exported method in a class and instantiating that class within the provider module. Port-to-export binding then arranges that the client (caller) has a reference to the instance containing the method. The client can then easily call the method through this reference. Similar but more elaborate arrangements are used in SystemC, exploiting the C++ language's multiple-inheritance feature [16][17][18].

In the world of transaction-level modeling this port binding is done dynamically, but SystemVerilog also offers a *static* mechanism in which initiator-to-target binding is specified as part of the module instance hierarchy. Although this static mechanism is naturally less flexible than dynamic port binding, it fits better into a refinement flow that incorporates both RTL and TLM models, since RTL models are invariably instantiated and connected as part of the static (time-zero) elaboration of the module instance hierarchy. The mechanism in question is that of subprogram import and export via modports on SystemVerilog interfaces.

Code Example 1 illustrates the mechanism of subprogram import using an interface. It shows a SystemVerilog interface `Intf1` that contains the implementation of a subprogram (`task C`). The modport `Caller_MP` provides a suitable connection point for a module. As usual with SystemVerilog modports, directions in a modport are specified from the point of view of a connected module; consequently, task C is specified as an `import` because any module connected to the modport will *import* task C from the interface, making it available for use by the module. The module calls the function by means of the dotted name *port_name*.C, and is completely ignorant of the precise instance name or location of the interface in which task C is implemented. Binding of this topology-independent task call to the providing interface is achieved in the enclosing module `Top1`, which instantiates both module and interface and connects the module instance's port to the appropriate modport on the interface instance. This mechanism has been fairly widely used by SystemVerilog programmers and can even be used in synthesizable designs if appropriate coding styles are respected.

**Code Example 1: Task import through a modport of an interface**

```
interface Intf1();

  task C (input integer A, output logic B);
    ...
  endtask

  modport Caller_MP (import task C (input integer A, output logic B));

endinterface
```

```
module Caller_Client ( Intf1.Caller_MP caller_port );
  initial begin
    logic X;
    caller_port.C(42, X);
    ...
  end
endmodule
```

```
module Top();
  Intf1 intf1_instance();
  Caller_Client client_instance(intf1_instance.Caller_MP);
endmodule
```

Code Example 2 shows a SystemVerilog interface Intf2 that calls a subprogram (task C) that is implemented in some connected module. In this situation the connected module provides or *exports* task C to the interface, and so the modport Target_MP specifies task C as an *export*. Code in the body of the interface is now able to call task C without knowing the identity of the target module that provides this task. The providing module implements task *port_name*.C in the expectation that an interface connected to its port will in due course place calls to that task; again, this module remains ignorant of the precise location and instance name of that connected interface. An enclosing module instantiates both module and interface, and connects the module's port to the appropriate modport on the interface instance.

**Code Example 2: Task export through a modport of an interface**

```
interface Intf2();

  modport Target_MP (export task C (input integer A, output logic B));

  initial begin
    logic X;
    C(42, X);
    ...
  end

endinterface
```

```
module Target_Provider ( Intf2.Target_MP target_port );
  task target_port.C (input integer A, output logic B);
    ...
  endtask
endmodule
```

```
module Top2();
  Intf2 intf2_instance();
  Target_Provider provider_instance(intf2_instance.Target_MP);
endmodule
```

Code Example 3 indicates how the task import/export mechanism allows a caller (client) module to call a subprogram implemented in a provider (target) module without needing to know the location of that provider. The interface `Intf3` has two modports. `Target_MP` allows a target module to provide an implementation of task `C` so that it is available to the interface. The interface itself does not call that task, but instead exposes it to a client module through `Caller_MP`. Once again, binding of caller to target is achieved in the top level module `Top3`. This mechanism provides a static implementation of caller-to-target binding, facilitating the use of TL modeling styles in a static instance hierarchy that can subsequently be refined to RTL.

**Code Example 3: Interface allows one module to provide a task called by another module**

```
interface Intf3();
  modport Target_MP (export task C (input integer A, output logic B));
  modport Caller_MP (import task C (input integer A, output logic B));
endinterface
```

```
module Target_Provider ( Intf3.Target_MP target_port );
  task target_port.C (input integer A, output logic B);
    ...
  endtask
endmodule
```

```
module Caller_Client ( Intf3.Caller_MP caller_port );
  initial begin
    logic X;
    caller_port.C(42, X);     // This will ultimately call the task that is
    ...                       // implemented in module Target_Provider
  end
endmodule
```

```
module Top3();
  Intf3 intf3_instance();
  Target_Provider provider_instance(intf3_instance.Target_MP);
  Caller_Client client_instance(intf3_instance.Caller_MP);
endmodule
```

## 6. TL Model in SystemVerilog

Our example system (Figure 1) has been implemented in a TL modeling style, using SystemVerilog modules and interfaces. Readers familiar with TL modeling using OOP techniques should note that our modeling was accomplished without the use of SystemVerilog classes, instead using a completely static hierarchy of module and interface instances. It is this use of an entirely static connection mechanism for TLM that makes it possible to refine components to RTL with minimal disruption to other parts of the model. By contrast, if TLM is implemented in the traditional manner using OOP techniques then the connection mechanism between TL models is so very different from the connections between RTL models that it is very disruptive to change any one component from one modeling style to the other.

To illustrate the interface-based TL modeling technique we examine in detail the read/write memory model and its connection to the bus fabric over an interface that represents an SBI bus instance. In this part of the system it is clear that the memory acts as a target, and the bus fabric as an initiator.

Consequently the memory model must provide implementations of the communication functions. A model conforming to the TLM-1 standard [OSCI] would use a so-called *request-response* protocol. However, for the sake of simplicity of exposition we have chosen to implement the interface as simple read and write methods. The approach could readily be extended to implement the usual TLM-1 interface methods.

Code Example 4 shows pertinent code for the TLM implementation of an SBI interface; note that it has two modports, one for an initiator (fabric) and one for a target (slave). The interface has no internal connectivity or signals, and no functionality of its own. A bus fabric module connected to the `tlm_fabric` modport is able to call transaction tasks `Read()` and `Write()`, assuming that they exist in the interface itself. In fact these tasks are implemented in the slave module, and are exported to the interface through its `tlm_slave` modport. The code has been shortened for convenience and, in particular, the definition of data type `word_t` has been omitted; it is defined, in a package elsewhere, as a 16-bit vector to match the native word width of the system.

**Code Example 4: TLM implementation of SBI interface**

```
interface sbi_tlm();

  modport tlm_slave (
    export task Write(word_t _addr, word_t _data),
    export task Read (word_t _addr, output word_t _data)
  );

  modport tlm_fabric (
    import task Write(word_t _addr, word_t _data),
    import task Read (word_t _addr, output word_t _data)
  );

endinterface
```

Code Example 5 shows how the read-write memory module `ram_tlm` provides implementations of these `Read` and `Write` transaction methods for use by the interface. From the module's point of view, these methods are exported to the interface and are called from the interface. Note that the task names must include the interface port name as a prefix: for example, task `sbi.Write`. In the example offered below, the `Read` and `Write` tasks are untimed and simply perform the required operation directly on the memory array. However, since the tasks are written in conventional SystemVerilog it is easy to add timing behavior to them if required.

**Code Example 5: TLM implementation of read-write memory with SBI interface**

```
module ram_tlm (sbi_tlm sbi);

  word_t store [0:(1 << ram_aw)-1];   // memory array

  // tasks exported to the SBI interface via the modport

  task sbi.Write(word_t _addr, word_t _data);
    store[_addr] = _data;        // untimed model of write behavior
  endtask

  task sbi.Read(word_t _addr, output word_t _data);
    _data = store[_addr];
  endtask

endmodule: ram_tlm
```

Code Example 6, below, is a fragment of the bus fabric module showing how the bus fabric can act as an initiator by calling a method that appears to be imported through a modport from the SBI interface. The method in question is in fact implemented by, and exported from, the target memory module. Alternatively, a similar method in the target SIO module is called if the write address is within the appropriate range.

**Code Example 6: TLM implementation of bus fabric connecting to slave by SBI interface**

```
module bus_fabric_tlm
  ( ...
    sbi_tlm.tlm_fabric ram  // provides Read() and Write() methods
    sbi_tlm.tlm_fabric sio  // provides Read() and Write() methods
  );

  ...

  task do_write(word_t addr, word_t data);
    if (addr >= sio_base)           // Bus fabric performs address decoding
      sio.Write(addr, data);        // Call method in interface, but method
    else                            //   is implemented in target module
      ram.Write(addr, data);
  endtask

endmodule : bus_fabric_tlm
```

In a very similar manner the CPU, DMA and ROM blocks can be implemented as transaction-level behavioral models. In each case, an initiator module is written as though it were calling methods in the connected interface, and a target module provides methods that appear to be called from the connected interface. Ultimately a method in the target is called by the initiator, but the connection mechanism is completely invisible to both target and initiator modules.

## 7. Refining Part of the Model to RTL

Our TL system model can now be executed in simulation, with the CPU executing simulated instructions stored in the ROM model. Although there is no timing in this model, the relative ordering of memory accesses from CPU and DMA components is faithfully preserved and the overall flow of activity in the system can be observed and debugged. The individual component models are written in an abstract style, with no concern for timing or signal-level detail, and therefore are concise and easy to create. Models of this kind can usually be written much more quickly and reliably than RTL designs, facilitating early testing of the overall system in simulation.

As the design progresses, RTL implementations of individual modules become available. Thanks to the flexibility of SystemVerilog interfaces, it is relatively straightforward to integrate these RTL models into the otherwise TL system model. In this section we discuss in detail the integration of an RTL implementation of the memory component, while continuing to model the remainder of the system in a TLM style.

### 7.1 Using an interface to connect a TLM bus-fabric model to the RTL memory model

The original concept for the current paper was to illustrate the use of SystemVerilog interfaces as adaptors between TLM and RTL modules. A second interface can be implemented, having a `tlm_fabric` modport identical to that provided by the `sbi_tlm` interface but also presenting an RTL-style modport on the memory side, as shown in Code Example 7. TLM methods imported through the fabric modport are now actually implemented within the interface itself, instead of being exported into the interface from the TLM memory module. The methods in the interface are closely similar to traditional Verilog bus-functional model (BFM) tasks, converting the transaction-level `Read()` and `Write()` method calls into the appropriate sequence of signal-level activity. In Code Example 7 we show only the `Read()` method, for brevity.

**Code Example 7: TLM-to-RTL adaptor functionality in an interface**

```
interface sbiGasket_rtl_slave_tlm_fabric(input bit clock);

modport tlm_fabric (   // identical to that in the sbi_tlm interface
    import task Write(word_t _addr, word_t _data),
    import task Read (word_t _addr, output word_t _data)
  );

  // RTL signals
  word_t addr, dataw, datar;
  logic we, re;

  // RTL modport for connection to slave
  modport rtl_slave (
    output datar,
    input  clock, reset, dataw, addr, we, re
  );

  // BFM-style tasks
  task Read(word_t _addr, output word_t _data);
    @(posedge clock)
    addr <= #1 _addr;
    re   <= #1 1;
    @(posedge clock)
    addr <= #1 'x;
    re   <= #1 0;
    @(posedge clock)
    _data = datar;
  endtask
  ...
  // similar implementation of Write() task
  ...

endinterface : sbiGasket_rtl_slave_tlm_fabric
```

This approach does not meet all our objectives. We aimed to be able to refine any part of the model – in this case, only the memory – from TL to RTL without impacting any other part of the model. However, the interface that links the bus fabric and memory modules is now a `sbiGasket_rtl_slave_tlm_fabric` interface, replacing the original `sbi_tlm` interface that linked the two TLM modules. Consequently, the port list of the fabric module must be changed, as illustrated in Code Example 8, because a different type of interface is in use – even though it provides exactly the same modport for use by the fabric module.

**Code Example 8: Concrete interface ports**

```
interface sbi_tlm();
  modport tlm_fabric (...);
  ...
endinterface
```

```
module fabric_tlm (
  ...
  sbi_tlm.tlm_fabric sbi,
  ...
);
  ...
endmodule
```

```
interface sbiGasket_rtl_slave();
  modport tlm_fabric (...);
  ...
endinterface
```

```
module fabric_tlm (
  ...
  sbiGasket_rtl_slave.tlm_fabric sbi,
  ...
);
  ...
endmodule
```

The SystemVerilog language allows a module's port list to specify a port that will connect to an interface's modport using a generic interface style, as shown in Code Example 9. This feature would permit the refinement of the memory module (slave) from TL to RTL with no change to the remainder of the system except to replace the `sbi_tlm` interface instance with an instance of an `sbiGasket_rtl_slave` interface.

**Code Example 9: Generic interface ports**

```
interface sbi_tlm();
  modport tlm_fabric (...);
  ...
endinterface
```

```
interface sbiGasket_rtl_slave();
  modport tlm_fabric (...);
  ...
endinterface
```

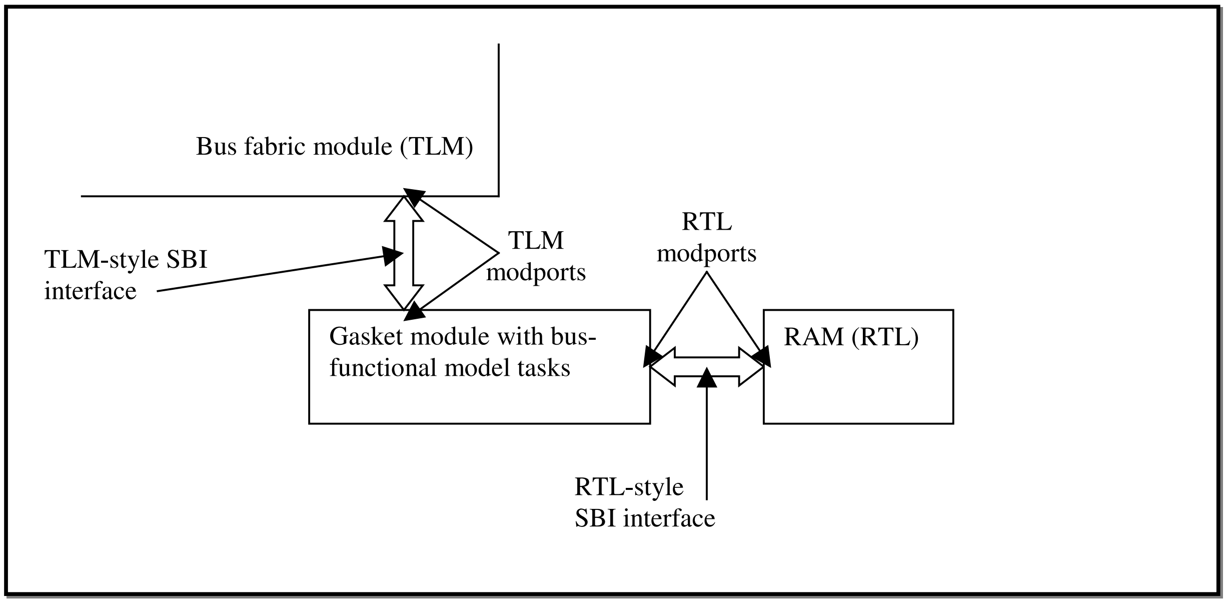
```
// Unchanged fabric module can use any
// interface, provided it has the
// correct kind of modport

module fabric_tlm (
  ...
  interface.tlm_fabric sbi,
  ...
);
  ...
endmodule
```

Unfortunately, current releases of Synopsys VCS offer the use of generic interface ports only as a limited customer availability (LCA) feature. In the interests of portability we therefore sought an alternative solution that would work with standard configurations of the tool.

## 7.2 Gasket module to bridge between interfaces

Because the desired approach of generic interface ports was unavailable, we chose instead to use a "gasket" module as a bridge between interfaces at different abstraction levels. This approach, illustrated in block diagram form in Figure 2, meets our original objectives (that the remainder of the system model should be undisturbed by the refinement step) at the cost of a somewhat more clumsy interconnection between TL and RTL models. The RTL memory model now connects to the same RTL-style interface that will be used in a pure-RTL system. The TL fabric model connects to the same TL-style interface that was used in the pure-TLM system. A new gasket module provides the bridge between the two interfaces. From the point of view of the TL interface it mimics the TL memory model; from the point of view of the RTL interface it mimics an RTL implementation of the fabric module.

**Figure 2: Use of a gasket module to adapt from one modeling style to another**



Implementation of the gasket module is straightforward and is given in Code Example 10 below.

**Code Example 10: TLM-to-RTL gasket module**

```
module sbi_tlm_gasket_rtl_slave(
  input logic clock,
  sbi_tlm.tlm_slave  tlm,  // Mimics TLM slave module
  sbi_rtl.rtl_fabric rtl); // Mimics RTL fabric module

  timeunit 1ns;
  timeprecision 1ns;

  import constants_pkg::*;  // provides definition of "word_t"

  assign rtl.clock = clock;

  // BFM tasks exported to the TLM fabric through port "tlm",
  // manipulating signals in the RTL interface through port "rtl"

  task tlm.Read(word_t _addr, output word_t _data);
    @(posedge clock)
    rtl.addr <= #1 _addr;
    rtl.re   <= #1 1;
    @(posedge clock)
    rtl.addr <= #1 'x;
    rtl.re   <= #1 0;
    @(posedge clock)
    _data = rtl.datar;
  endtask

  task tlm.Write(word_t _addr, word_t _data);
    @(posedge clock)
    rtl.addr  <= #1 _addr;
    rtl.dataw <= #1 _data;
    rtl.we    <= #1 1;
    @(posedge clock)
    rtl.addr  <= #1 'x;
    rtl.dataw <= #1 'x;
    rtl.we    <= #1 0;
  endtask

endmodule
```

## 8. Completing the migration to RTL

Given the gasket-module arrangements introduced in the previous section, it becomes very easy to migrate the model incrementally from TL to RTL modeling style. Use of SystemVerilog gasket modules means that each block in the design can connect to whatever sort of interface it expects to use. If a block at the other end of the interface is to be modeled in a different style, a gasket module and a second interface provides the necessary link; if that other block uses the same modeling style, no gasket and no second interface is needed. We have achieved our original aim of progressive module-by-module refinement of a model, with no disturbance to other parts of the model at each refinement step.

## 9. Performance Considerations

We have also implemented the entire system in a conventional RTL style. Even for this rather small system (fewer than 20,000 ASIC-equivalent gates) the simulation speed improvement of TLM over RTL was about a factor of 30 in our experiments using Synopsys's VCS$^{TM}$ simulator.

To make these speed measurements we created test programs for the modeled CPU that ran several million modeled CPU instructions, consuming many seconds of workstation CPU time. For simulations this long, we were able to observe simulation runtime increasing linearly with number of simulated instructions, to a repeatability of better than 5%. We therefore concluded that our simulations were long enough that the one-off time costs of loading the simulation, initialization and so forth were negligible and therefore our measurements were acceptably reliable. The measurements were made on a desktop Linux workstation with negligible CPU load other than the simulation being measured. Sample measurements are given in Figure 3.

For larger designs containing many more signals, we would expect to see even greater simulation performance benefits of TL modeling over a register-transfer style.

For our first performance tests we compared two versions of the system in which only the read-write memory module was changed from TL to RTL style, and the remainder of the system was all TLM. We were initially surprised to see only a factor of 2 slowdown in the RTL simulation, until we noted that our mix of instructions executed by the CPU performs a read/write memory access only about once for every 50 clock cycles. We then also migrated the ROM implementation from TL to RTL, and observed a much larger slowdown (the CPU performs many more accesses to ROM than to read/write memory). We conjecture that the total simulation time for such a model is dominated by, and is closely linear with, the amount of RTL activity. The transaction-level parts of the model represent only a very small part of the total simulation time when even a small part of the system is re-implemented as RTL. This observation leads us to believe that a hybrid TL/RTL system model of a system provides a useful and efficient vehicle for the testing of individual RTL models in an environment otherwise largely implemented using TL components.

**Figure 3: Sample performance measurements**

| Model configuration | Instructions executed on the modeled processor | CPU time consumed on simulation workstation | Modeled instructions per second of simulation CPU |
|---|---|---|---|
| All components transaction-level | 9,379,843 | 8.120 seconds | 1,155,153 |
| RTL model of RAM, all other components transaction-level | 9,379,843 | 14.440 seconds | 649,573 |
| ROM and RAM modeled at RTL, all other components TL | 9,379,843 | 48.270 seconds | 194,320 |
| All components RTL | 9,379,843 | 732.450 seconds | 12,806 |

## 10. Integration of C-based TL Models

It has been noted elsewhere [19] that SystemVerilog's Direct Programming Interface (DPI) features make it very straightforward to call C or C++ routines from a SystemVerilog model. Consequently, C models that have a TL-style interface can easily be called either from within a SystemVerilog interface or from a stub module that has an interface-type port. In both cases the C functions are then available, in the guise of SystemVerilog functions or tasks, for import from the SystemVerilog interface into a connected client module.

## 11. Conclusions

Using SV interfaces to implement TL-style communication and connection makes it possible to refine TL models to RTL and integrate them with a TL system model without disrupting other parts of the model. A SystemVerilog interface can b e modified to provide appropriate adapter functionality between modeling styles; alternatively, a gasket module can easily be introduced between different styles of interface, to meet the same need if tool support for generic interface ports is not available. Consequently each TL-to-RTL refinement step can be verified by execution in a common simulation environment, with minimal disruption to other components of the system model. Although the introduction of any RTL code into an otherwise TL system model is sure to impose a significant performance penalty, this penalty is no larger than would be expected with other techniques. Combined with the use of SV-DPI to link to external C/C++ models, this approach makes SystemVerilog interfaces attractive as "glue" for integrating heterogeneous modeling styles, permitting the same verification environment to be re-used whilst components of a system are progressively refined from behavioral to RTL modeling styles.

## 12. References

[1] IEEE Std.1850. The Property Specification Language. ISBN 0-7381-4780-X. IEEE 2005.

[2] Haque, F and Michelson, J. The art of verificaiton with SystemVerilog assertions. ISBN 0-9711994-1-8. Verification Central 2006.

[3] Carbon Design Systems Inc. *Carbon Model Studio* product. `http://carbondesignsystems.com/products.shtml`

[4] Calypto Design Systems. *SLEC* systemC to RTL equivalence checker. `http://www.calypto.com/slecsystem.php`

[5] Knapp D *et al*. Designing with Synopsys Behavioral Compiler. Springer 1995.

[6] Forte Design Systems Inc. *Cynthesizer* product. `http://www.forteds.com/products/cynthesizer.asp`

[7] Mentor Graphics Inc. *Catapult Synthesis* product. `http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm`

[8] ChipVision Design Systems AG. *PowerOpt* product. `http://www.chipvision.com/products/index.php`

[9] Aldec Inc. *Riviera* mixed-language simulator. `www.aldec.com`

[10] Cadence Design Systems Inc. *Incisive Enterprise Simulator*. `www.cadence.com`

[11] Mentor Graphics Inc. *Questa* simulator. `www.mentor.com`

[12] Synopsys Inc. *VCS* mixed-language simulator product. `http://synopsys.com/products/simulation/simulation.html`

[13] `www.doulos.com`

[14] Glasser M (ed). Advanced Verification Methodology Cookbook Version 2.0. Mentor Graphics Inc, 2006.

[15] Open Verification Methodology. See `www.ovmworld.org`.

[16] ISO/IEC Standard 14882: The C++ programming language. International Standards Organization, 2003.

[17] IEEE Std.1800. SystemVerilog – Unified Hardware Design, Specification and Verification Language. ISBN 0-7381-4811-3 (PDF document). IEEE 2005.

[18] IEEE Std.1666. SystemC. IEEE 2005.

[19] Sutherland S. Integrating SystemC models with Verilog and SystemVerilog models using the SystemVerilog Direct Programming Interface. SNUG Europe, Munich 2004.