Maintaining Consistency Between SystemC and RTL System Designs

Alistair Bruce
ARM
152 Rockingham Street
Sheffield, UK S1 4EB
alistair.bruce@arm.com

Andrew Nightingale
ARM
110 Fulbourn Road
Cambridge, UK CB1 9NJ
andrew.nightingale@arm.com

Nizar Romdhane
ARM
110 Fulbourn Road
Cambridge, UK CB1 9NJ
nizar.romdhane@arm.com

M M Kamal Hashmi
Spiratech Ltd
Carrington Business Park
Manchester, UK M31 4ZU
kamal@spiratech.com

Steve Beavis
Spiratech Ltd
Carrington Business Park
Manchester, UK M31 4ZU
steve.beavis@spiratech.com

Christopher Lennard
ARM
110 Fulbourn Road
Cambridge, UK CB1 9NJ
chris.lennard@arm.com

ABSTRACT

We describe how system design consistency can be maintained across multiple levels of design abstraction using a modular verification IP strategy. This strategy involves delivery of verification IP in an environment independent manner, utilizing a standard system verification architecture that leverages re-usable component verification drivers, transaction-based interfaces, and synchronization through a system-verification master. This enables a single test-bench to be applied for systems modeled both in SystemC, as well as at the RT level. The configuration of the verification testbench is kept consistent with the design by using system-design meta-data described using the specifications of The SPIRIT Consortium.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems) – *interfaces, topology.*

B5.2 [**Register Transfer Level Implementation**]: Design Aids - *verification*

General Terms

Design, Standardization, Languages, Verification.

Keywords

Verification, VIP, Testbench, SystemC, TLM, Transactor, RTL, SPIRIT

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

1. INTRODUCTION

Testbench re-use is key to bringing system-level design into standard system-on-chip (SoC) engineering practice. The benefits of optimizing at the Electronic System Level (ESL) are only realized if that flexibility can be quickly and verifiably reflected into the downstream design process. A unified testbench guarantees consistency across design refinements. This paper introduces several fundamental concepts needed for the construction of a re-usable testbench, and delivery of verification IP (VIP) compatible with such a multi-abstraction and vendor-neutral environment.

We describe the verification re-use strategy being adopted by ARM in collaboration with Partners. Section 2 covers the basic architecture of a re-usable system test-bench; and Section 3 describes how it interfaces to multiple abstractions of the device-under-test (DUT). Section 4 covers the issue of testbench configuration consistency, and Section 5 presents results as applied to an ARM® AMBA® AHB™ peripheral. The experiments use the ARM RealView® SoC Designer SystemC environment, the transactor technology from SpiraTech, and the fast register-transfer level (RTL) model compilation tools from Tenison Design Automation.

2. A MODULAR SYSTEM TESTBENCH ARCHITECTURE

The key to verification re-use is adherence to a common testbench architecture that applies across all levels of design abstraction. There are two elements to this: the architecture for test bench assembly and control, and the method to deliver verification stimulus and constraint information in an abstraction neutral way.

2.1 Structuring a VIP for Re-Use

XVCs (eXtensible Verification Components)[1] provide a foundation for authoring VIPs in a modular and scalable way, reusable across system-level verification environments with minimizal test set-up overhead. XVCs can be used to drive block interconnect infrastructures or external interfaces. They support

other XVC components by monitoring system state and providing notification information, and they also enable the delivery of test-stimulus in a language-neutral way.

An XVC is a container for verification IP divided into two elements. The top layer is a user-extensible library of test actions with a defined action interface for integration into a verification environment. The bottom layer integrates individual transactors for implementing the actions on a physical- or transaction-level interface.

The action interface of the top layer allows coordination of test stimulus. The top layer verification environment interface of an XVC can connect directly to an XVC manager, as shown in Figure 1. An XVC manager can support any number of XVCs at one time and uses this interface to schedule execution of individual actions within any given XVC. XVCs can also pass data and status back up to the test control infrastructure to communicate with other XVCs in the same environment.

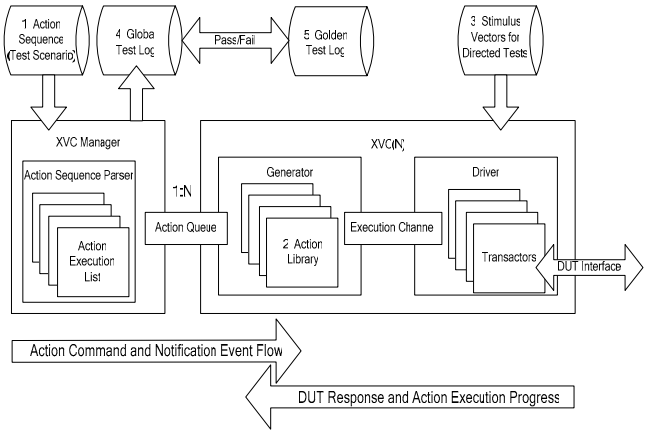

Figure 1. User test scenario being driven into XVC environment via XVC Manager

2.2 Delivering a VIP Environment

Delivering a VIP environment requires three major infrastructural elements to be supported:

(i) A predefined XVC manager that uses test scenario descriptions (Item 1 in Figure 1) written as external configuration files in a plain text format. These files can be reused or modified to new requirements with relative ease. The ability to direct the test via simple text input files enables the IP vendor to 'program' an XVC manager, and the user to obtain effective test results without a detailed understanding of the internals of the VIP environment. A system-level test program can contain one or more test scenarios. Test scenarios are designed to meet one or more verification requirements. A test program might consist of a number of test scenarios which, when taken together, fulfil several test requirements for a DUT. Test scenarios can also be used to encapsulate common sequences of actions (Item 2 in Figure 1) so they can be reused by different test programs. For example, the operations required of a number of XVCs to configure a system could be contained in a scenario.

(ii) A set of XVCs with pre-defined action libraries and transactors is also supplied. One example of an action library function is the ability to generate random data streams. XVCs may also contain action library functions to drive pre-defined test vectors (Item 3 in Figure 1). The pre-defined vectors are designed to contribute towards specific DUT functional coverage points. A test scenario as described above may direct a number of XVCs in concert, to hit further coverage goals. As mentioned above, action library functions operate at a higher abstraction level than their 'protocol-specific' transactor counterparts. This allows XVCs themselves to be used for testing DUTs at different abstraction levels.

(iii) Golden test logs (Item 5 in Figure 1) are supplied that should match the generated test logs (Item 4 in Figure 1). The pre-defined golden test logs represent output obtained from a complete functional coverage run.

With a verification environment set-up as described in above, test scenarios and stimulus vectors can be specified as environment-neutral external files. This eliminates the need for recompilation of testbench components or DUT to run different test scenarios. This achieves design-environment neutral delivery of VIP.

3. MULTI-ABSTRACTION TESTBENCH

For the system testbench to apply throughout the design refinement process without alteration, the generated test stimulus must be passed to the DUT through a driver that can handle multiple abstractions. These drivers are known as transactors, which we introduce here along with their application to standard transaction-level model (TLM) interfaces.

3.1 Driving a DUT through Transactors

If an interface protocol is specified as a set of abstraction levels and a mapping between these levels is defined for data and communication refinement, it is possible to write multi-level transactors to automatically bridge between abstractions. These transactors receive stimulus and translate the activity to the data-structure, call-structure and temporal mechanisms used by design models at other abstractions.

Traditionally, transactors were written as two unidirectional components or pieces of code – a "driver" from high to low levels and a "monitor" from low to high level. These drivers allow test library functions written in an abstract, untimed way (i.e., time just expressed as a constraint rather than an explicit synchronizing function) to exercise and check DUTs at different abstraction levels. This is depicted in Figure 2a. The transactors needed to fit the re-usable XVC verification architecture, however, need to function bi-directionally as an XVC may be able to both drive and monitor the DUT, as shown in Figure 2b.

As transactors are complex to design, those created by hand often only connect two levels of abstraction. However many design simulations today have several abstractions integrated together, including untimed or Programmer's View (PV) models, cycle approximate or Programmer's View with Timing (PVT) models, cycle-accurate, and intra-cycle timing accurate models. Consequently, handling only two abstraction levels with a transactor is not sufficient. Complex transactors that can handle interconnection of multiple abstraction levels to need to be automatically generated, and this requires capture of the interface abstractions in a single formal description [2].

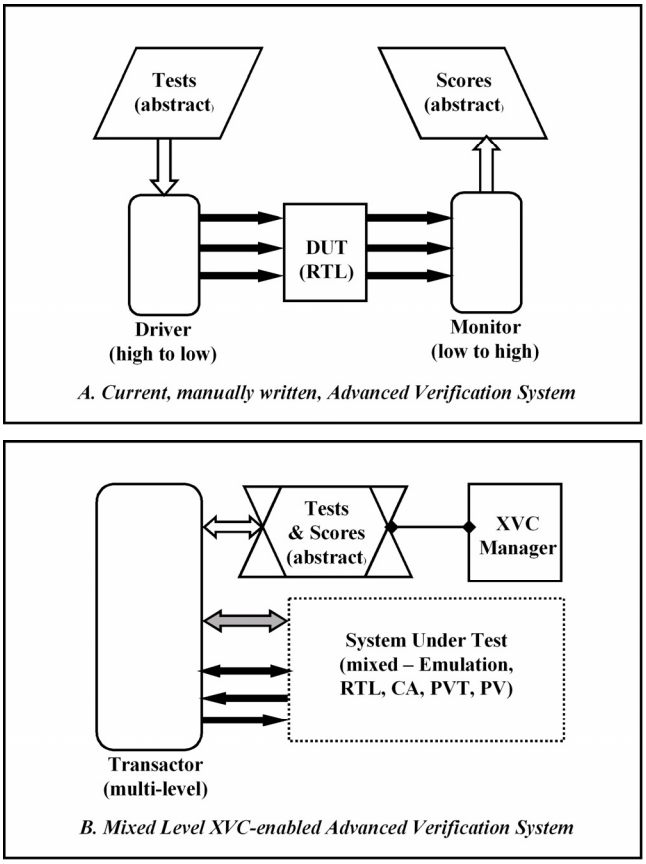

Figure 2: Transactors enable use of a single testbench across multiple levels of abstraction

A formal description of an interface abstraction hierarchy must capture the temporal, data and behavioral aspects of an interface protocol at multiple levels of abstraction along with the mapping between the levels. A transactor generated from these descriptions can simultaneously drive and monitor communication at any level of abstraction. This enables each component in a system to communicate using the interface semantics native to its design abstraction, while the transactor attached to the component interface translates the communicated information into an abstraction appropriate for the recipient.

3.2 Transactors for Standard TLM Interfaces

For SystemC modeling, the transactors are generated with a method-calling interface for each level of abstraction. To achieve plug-and-play compatibility, defacto-standard TLM interfaces that represent common modeling styles need to be supported. As an example, transactors can help connect components seamlessly to ARM SystemC models by implementing the ARM RealView ESL APIs [3]. The ARM RealView ESL APIs are a coordinated bundle of SystemC and C++ modelling interfaces for use in cycle-accurate simulation, debugging and profiling of SoC virtual platforms. The set is composed of three APIs: the Cycle-Accurate Simulation Interface (CASI), the Cycle-Accurate Debug Interface (CADI) and the Cycle-Accurate Profile Interface (CAPI). For the purposes of this paper, we only describe the simulation interface here.

The Cycle-Accurate Simulation Interface (CASI) defines a set of SystemC interfaces supporting a generic implementation of models (components), ports (master ports) and channels (slave ports), designed to be used in a cycle-based simulation environment. CASI models must implement two functions that are called by the scheduler: communicate and update. In the communicate function the communication with other models is performed. In the update phase the internal resources are updated. As an example of how these functions can be synchronized to a simulation, the user may choose to map them respectively to the positive and negative edges of a system clock.

The CASI simulation interface is a TLM transport layer that supports any bus protocol. The bus protocol support is built on top of the TLM transport layer. As an example, to support the AMBA 3 AXI™ protocol, the CASI multi cycle transaction interface captures all the AMBA 3 AXI signals within a single generic transaction information container populated and manipulated during the life span of the transaction. The data and control information as well as the channel handshakes of an entire transaction can be found in a single data structure which for a single transaction is shared by all channels. Each handshake sequence also referred to as transfer is mapped to a transaction step. The multi-level transactor for AMBA 3 AXI is able to unpack all the data in these transactions, and map that to a signal sequence at the RT level, and vice-versa, thereby creating an automatic bridge between the SystemC TLM model, an RT implementation, or the test stimulus generated by an XVC.

4. DUT & TESTBENCH CONFIGURATION

The multi-abstraction design flow requires automated alignment of the verification testbench with the design configuration. For example, integration tests require knowledge of register access profiles and these may be altered in the course of designing the SoC. The automated alignment is achieved by using design meta-data [4] to describe the design configuration, and associating this with automated design and verification set-up tasks. To achieve this consistency in a multi-vendor design-flow, a common exchange format for design meta-data is required. The SPIRIT Consortium is providing specifications to help standardize across the industry on a single representation design meta-data.

4.1 The SPIRIT Consortium Specifications

The SPIRIT Consortium [5] specification provides an XML schema [6][7] and semantics to enable the unified authoring, exchange and processing of design meta-data and it also provides a complete API for meta-data exchange and data-base querying. The user of a SPIRIT-enabled design environment is able to gather meta-data component descriptions together into a library, along with SPIRIT-enabled definitions for any bus interfaces referenced within the components. The SPIRIT-enabled meta-data for a component includes: top-level I/O and bus interface declarations, memory map definition, IP views identification linked to the files representing the views, and implementation and integration constraints for the IP to properly function within a subsystem. A design environment can then automatically instantiate, configure and connect SPIRIT-enabled components to form a system design, which is itself represented as an exchangeable meta-data file expressing component instantiation and connectivity.

To enable automated configuration of a design and its test-bench, The SPIRIT Consortium defines the concept of plug-in generators. Generators can be connected using a meta-data dumping mechanism, Loose Generator Interface (LGI), or the Tight

Generator Interface (TGI) which is a full API based on SOAP [8] to guarantee language and environment-independent generator integration.

4.2 Applying SPIRIT to the System Testbench

The current version of The SPIRIT Consortium specification, v1.2, enables automated composition, integration and configuration of an RTL verification environment. Figure 3 shows a system testbench annotated with the various SPIRIT-enabled features used to describe it. The SPIRIT design file contains a complete specification of the testbench as a set of component instances and the connections between them. Configurable components have had values for their parameters captured on the instances, where they are different from the defaults. Two types of verification interface are illustrated: *monitor interfaces* that allow verification components (e.g. XVC monitors) to be added or deleted without changing design connectivity, and *whitebox interfaces* that specify internal elements accessible from the design environment (especially by verification components such as XVC peripheral test blocks). The DUT design configuration, instance views, monitor insertion, connection points of the system testbench to the DUT, register configuration, and so on are all captured in SPIRIT-enabled meta-data.

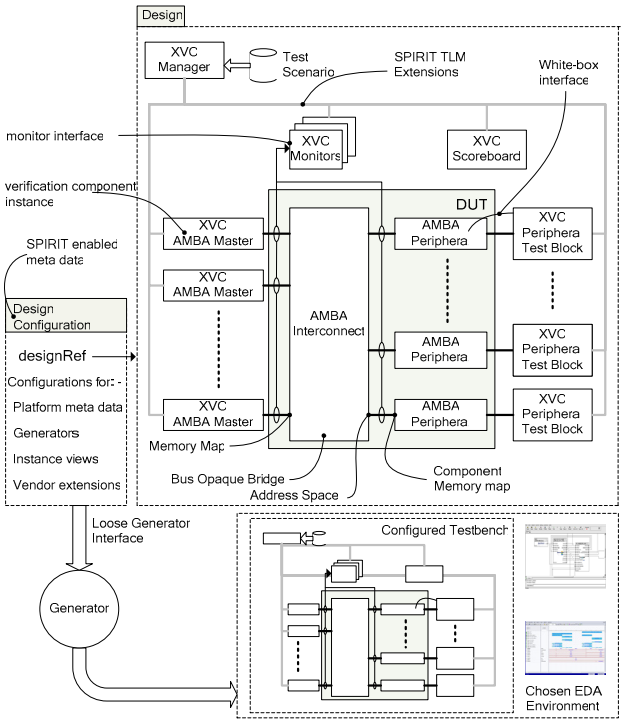

Figure 3. SPIRIT meta-data applied to a block interconnect infrastructure test environment

Combining XVC methodology with The SPIRIT Consortium specifications allows a single testbench description to be used on a wide variety of platforms. The SPIRIT-enabled meta-data provides a language independent description of the testbench configuration and its connection to the DUT, while the XVC methodology provides platform independence for the tests themselves.

To convert the testbench to a form suitable for a particular tool environment, generators that configure the XVCs are captured in a SPIRIT-enabled way and executed using the LGI (Loose Generator Interface). This takes the information held in a design configuration and produces a fully configured testbench. The design configuration can also specify a chain of generator calls that compiles the testbench and then runs the tests on it.

The internal architecture of the system testbench, including the XVC manager and the XVC generator layer, are implemented in a high level verification language (for example, e, SystemC or SystemVerilog) with high-level language transactional interfaces between them. The structure of these transactional connections between the XVC manager and the XVCs cannot be described using The SPIRIT Consortium v1.2 specification. However, this internal testbench architecture will be able to be described using the TLM extensions currently under development in The SPIRIT Consortium.

5. UNIFIED TESTBENCH IN PRACTICE

We apply the general techniques described above to a specific case of an AMBA AHB peripheral, the ARM PrimeCell® PL190 Vectored Interrupt Controller. We are modeling the AMBA bus matrix at the PVT level using the RealView ESP API SystemC interfaces, and connecting this to the PrimeCell PL190 Vectored Interrupt Controller at the RT level using transactor technology.

5.1 Building the System and Testbench

The SPIRIT-enabled meta-data describes the system to be tested. A meta-data design file specifies the required component and VIP connectivity. Parameters of design and verification component instances, such as bus_width or buffer_depth, are able to be set using the SPIRIT-enabled meta-data. A design configuration file captures the ancillary information about the tool environment. For example, which view of a component instance to use or which level its interface is modelled at. The test scenario file can also be configured here. To set-up a regression run, a design can have multiple SPIRIT-enabled design configuration files to specify a range of tests and processes on that design.

5.2 Creating and Inserting the Transactors

The generic SpiraTech AHB transactor defines the mapping between many levels of abstraction of the AMBA AHB protocol from a simple time-unaware read or write transaction down to cycle accurate messages like slave_control and right down to the data on the actual wire-level ports. The transactor is parametrized by the size of the data bus and the number of slaves and masters on the bus. The SPIRIT meta-data description of the design enables automated selection, generation, configuration and instantiation of a transactor into the system model.

In Figure 4, we see the result of a transactor instantiation driven through SPIRIT-enabled meta-data. The screenshot shows part of a testbench similar to that described in Section 4.2 loaded into the ARM AMBA Designer tool, an IP optimization, creation and verification tool built on RealView SoC Designer technology for TLM modeling. The AMBA bus matrix is modelled at the PVT level but the PrimeCell PL190 Vectored Interrupt Controller is at RTL. A transactor has therefore been generated to convert between the PVT transactions and RTL signals.

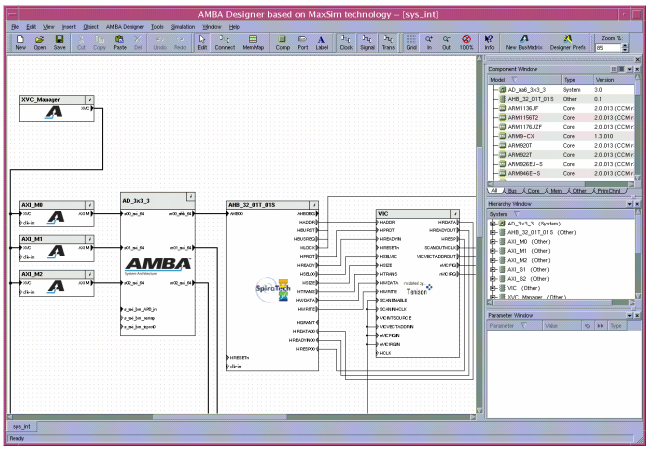

Figure 4. Fragment of system testbench in AMBA Designer showing automated transactor insertion

5.3 Executing a Simulation

Figure 5 shows some results from a run of an integration test, in particular an AHB read and write transaction and the corresponding low-level activity. Here the transactions to the PL190 are displayed at both the signal and transaction levels. The same tests can be run with a testbench generated from an all PVT configuration. The transactor dynamically generates high level transactions from low-level activity and vice versa – depending on which way information is flowing. Comparison of the results from the tests at each level demonstrates the consistency of the two levels.

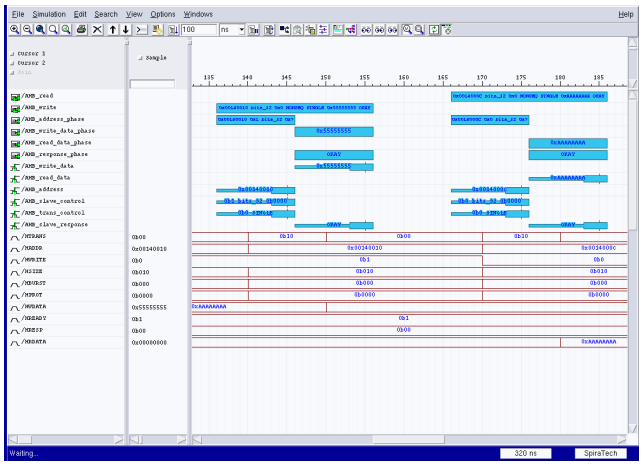

Figure 5. Multi-Abstraction Transactor display: transactions – top, wires - bottom

The simulation speed of any mixed model where RTL components are mixed with cycle-accurate and abstract models and test benches is always dominated by the RTL parts – usually over 95% of simulation time is spent in the RTL components. This is not surprising if the amount of work the RTL does is considered – all the hardware activity is modelled on every clock even if the outcome is not relevant. Even if the transactor is used to switch off RTL components that are not being addressed by the bus, the performance slowdown in running an RTL model with the highly performance optimised RealView components and engine is still very noticeable. To minimize the impact on simulation speed, the RTL has been converted from Verilog to a model running under RealView SoC Designer by the Tenison VTOC Generate 3.0.7 tool [9]. This tool abstracts RTL to cycle-accurate C++ [10], thereby achieving a speed-up of an order of magnitude while maintaining full cycle-timing functionality. An alternative approach is to place the RTL on an emulation platform. In this case, part of the transactor is emulated to create a co-simulation / emulation bridge.

6. Conclusions and Futures

We have introduced three key concepts that enable the unified system testbench. These are: (1) the XVC system testbench methodology which is an architecture for building testbenches to allow VIP to be delivered in an environment and model-abstraction independent manner, (2) transactor technology, which provides a constraint-based way of specifying the relationship between interface abstractions to allow bi-directional abstraction bridges to be automatically created, and (3) The SPIRIT Consortium meta-data specifications that enable design and test-bench configuration to be maintained in an environment and abstraction-independent manner.

We have shown that these concepts can be used today to link SystemC PVT models and RT level components under a unified system testbench. The forthcoming industry standardization around SystemC PV-level interfaces, and the extension of The SPIRIT Consortium specifications to handle abstract design specification, will enable this methodology to be automated across the entire system-level modeling and verification flow.

7. REFERENCES

[1] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, “Verification Methodology Manual for SystemVerilog”, September 28, 2005, Springer ISBN: 0387255389

[2] M M K. Hashmi, C. Jones, "Curing Schizophrenic Tendencies in Multi-Level System Design", Design & Verification Conference 2003

[3] ARM RealView ESL APIs. www.arm.com/products/DevTools/ESLmodelinterfaces.html

[4] C. Lennard, E. Granata, “The Meta-Methods: Managing design risk during IP selection and integration”. European IP 99 Conference, November 1999

[5] The SPIRIT Consortium, “SPIRIT 1.2 Specification”, www.spiritconsortium.org, April 2006

[6] World Wide Web Consortium, “Extensible Markup Language (XML) 1.0” Third Edition, 2004

[7] World Wide Web Consortium, “XML Schema Part 1: Structures”, Second Edition; “XML Schema Part 2: Datatypes” Second Edition, 2004

[8] SOAP Specifications: www.w3.org/TR/soap

[9] VTOC Generate: www.tenison.com

[10] D. Greaves. "A Verilog to C Compiler. International Workshop on Rapid System Prototyping", Paris, 21-23 June, 2000. Published in the IEEE Transactions on Software Engineering, 28 (9), September 2002.