The Diagnostic Channel: Increasing Visibility and Control in SystemC Models

Joseph Chapman – The MITRE Corporation

Abstract

Today's complex datapath architectures for System-on-a-Chip (SoC) and FPGA-based systems demand more effective verification techniques to reduce cost and schedule risks. Typical transaction-level-modeling techniques, when used for verification, lack in observability and controllability of internal states and signals. We introduced a novel approach to the functional verification of datapath systems by embedding diagnostic channel testbench components throughout the design. Diagnostic channels differ from traditional SystemC channels because they can verify and generate data as well as bind to multiple input and output ports. A brief case study using the diagnostic channels is presented along with plans for future work.

Table of contents

Table of Figures

1 Introduction

The growing complexity, size and performance demands of today's electronic applications is driving the industry towards using FPGA-based reconfigurable compute platforms. Even as FPGA vendors continue to offer products with higher densities and more embedded features, designs that are considered modest often out pace the latest single FPGA devices. Given these trends, designers are facing the challenge of designing and verifying architectures that are partitioned over a number of FPGAs communicating over platform-specific board-level interconnect standards such as VME, PCI-e and rapidIO to name a few. Even with platform support libraries, using conventional techniques, we have found it difficult to truly gauge system behavior before synthesis, place and route, and deployment. Additionally, using conventional techniques, bugs related to interconnect are not detected until the application is deployed which quickly leads to schedule overrun, especially as the size of the system increases.

To address the challenges of modeling and verifying complex reconfigurable systems, we adopted SystemC. Given that SystemC is a C++ class library, it seemed especially well suited for abstract modeling and rapid design exploration. In addition, VCS allows SystemC and RTL co-simulation which enables models to be used to verify hardware in a self-checking configuration. However, with all these benefits, we found that SystemC lagged behind the built in features of SystemVerilog, specifically with assertions, constrained randomization, and functional coverage metrics. Therefore, to make SystemC a more complete solution for our needs, we decided to add some robustness to its verification flow.

Our goal was to develop simple yet powerful verification components that were abstracted away from functional components, thus providing a reusable verification suite for SystemC designs. Since most of our designs are datapath centric, we decided that the communication points between functional blocks were good places to investigate and inject data. This naturally led us to focus on enhancing SystemC channels. The following sections contain a SystemC channel primer and describe our design in detail.

2 SystemC Communication Primer

In a general sense, SystemC channels are classes that encapsulate the behavior of the transport between design entities, cleanly separating implementations from interfaces. They are a powerful tool for modeling because they allow transport behavior to change without requiring functional blocks to change. For example, a channel implementing a single interface could represent something as simple as a wire, or something as complex as a bus fabric.

In SystemC, there are two types of channels: primitive channels and hierarchical channels. Primitive channels inherit from sc_prim_channel and have limited functionality, but higher performance than their hierarchical counterparts. Hierarchical channels inherit from sc_channel and can have internal hierarchy and processes. Hierarchical channels are more suitable for modeling complex interconnect at the expense of performance.

Strictly speaking, a given class is a SystemC channel if it inherits from one of the two channel base classes; however, to be of any practical use to a design, it also needs to implement one or more interfaces derived from sc_interface. SystemC interfaces can be written at a variety of abstraction levels. For example, the sc_signal_in_if is intended to provide access to a "signal" which is conceptually similar to a VHDL signal. SystemC signals could be used to communicate with a model of a FIFO by toggling them according to the FIFO's protocol, however the sc_fifo_in_if and sc_fifo_out_if interfaces provide convenient read() and write() functions instead. Designers can further abstract interfaces to provide methods that would take hundreds of clock cycles to complete in hardware.

Although SystemC can be written at a variety of abstraction levels, models are typically developed at the "transaction level". Transaction level modeling is commonly defined as using function-based communication between models. It is a widely used technique because transaction level models don't take as much time to develop as their more accurate RTL counterparts, yet can still be fairly accurate with regards to the timing and performance of the final system.

The final component involved in basic SystemC communication is the sc_port class. Ports are what modules use to communicate with each other. Channels are bound to ports

during the elaboration phase of the simulation and provide the communication path from one module to the next. Modules communicate with their ports, which in turn, communicate with bound channels. For example: in the system shown in Figure 1 below, let's assume that Module A calls write(x) on its output port. The port would, in turn, forward the function call to Channel B. Channel B's implementation would then store the data until read() was called through the input port of module C.

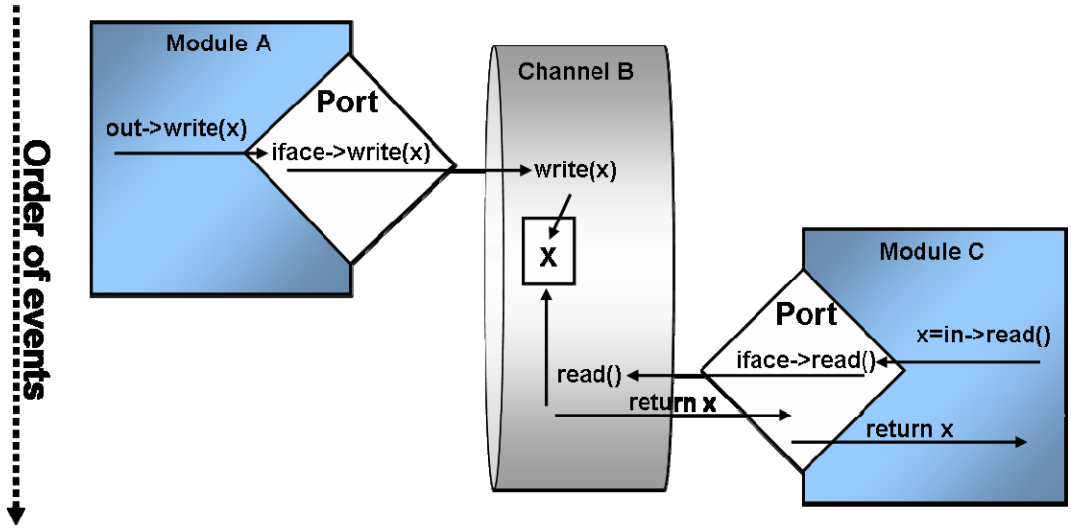

Figure 1: SystemC Communication Example

Although the channel implementation in this example behaved like a fifo, other implementations could disruptively change the data, delay its availability, selectively discard it, or otherwise behave in any user-defined way. Furthermore, because channels are abstracted away from modules through ports and interfaces, any changes to channel implementations do not require changes to module implementations. As described in the following section, our design exploits this concept to seamlessly insert verification facilities into channels implementing FIFO interfaces.

3 The diagnostic channel

The diagnostic channel was developed to both model interconnect at the transaction level and to provide significant verification facilities. The block diagram below shows the architecture of the diagnostic channel

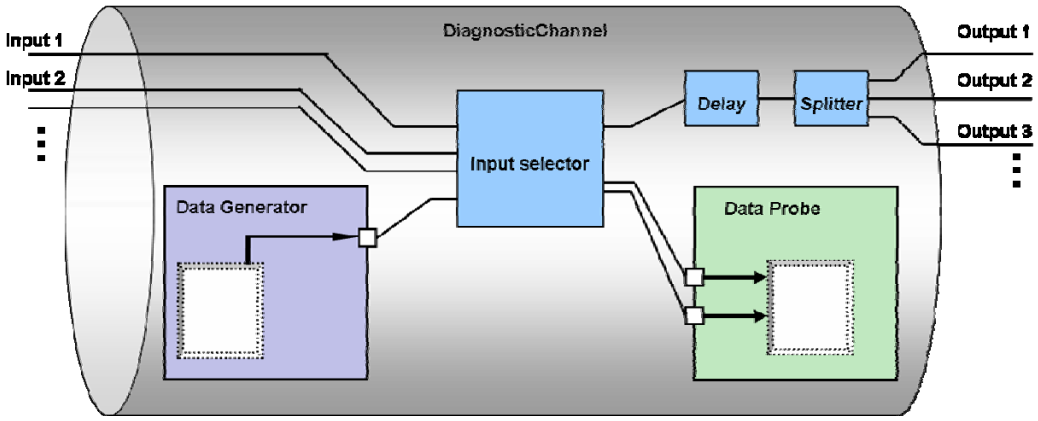

Figure 2: Diagnostic Channel Block Diagram

As can be seen in Figure 2, the diagnostic channel contains functional blocks to implement the following features:

- Multiple input/output sources
- Data generation facilities
- Data inspection facilities
- Delay insertion facilities

The following sections detail the functionality of the diagnostic channel.

3.1 Data types and interfaces

In order to increase the reusability of the diagnostic channels, we developed a data interface that is similar to the sc_extensions_if. This interface contains functions to access and modify data without compile-time knowledge of the internal type being accessed. Although the interface can be expanded, a current limitation of the implementation is that it only supports types that can be expressed as doubles, strings, or combinations thereof. A subset of the interface definition is listed below:

```
class DataInterface {
 public:
  typedef numericT double;

  // Get Number of numeric values
  virtual int getNumNumeric() const = 0;

  // Get numeric argument by index
  virtual int getNumeric(int index, numericT & value) const = 0;
```

```
    // Set numeric argument by index
    virtual int setNumeric(int index, const numericT & value) = 0;
}
```

Similar functions are declared for string values. Through this interface, structures composed of numeric types that can be converted to/from doubles and strings can be read and written without specific knowledge of the internal type used.

Though there are many interfaces provided with SystemC, FIFO interfaces are the most appropriate for modeling communication in data-path systems with point-to-point, flow-controlled streaming data. Therefore, we wrote transaction level interfaces that extended sc_fifo_in_if and sc_fifo_out_if to add some extra functions for additional high level control.

3.2 Channel Data Routing

SystemC FIFO channels do not allow connections to multiple input or output ports. This behavior is appropriate for modeling purposes as most hardware FIFOs are only connected to one data source and one sink. However, the ability to connect to multiple inputs and outputs can be advantageous for co-simulating models of the same block at different levels of abstraction as shown in Figure 3.

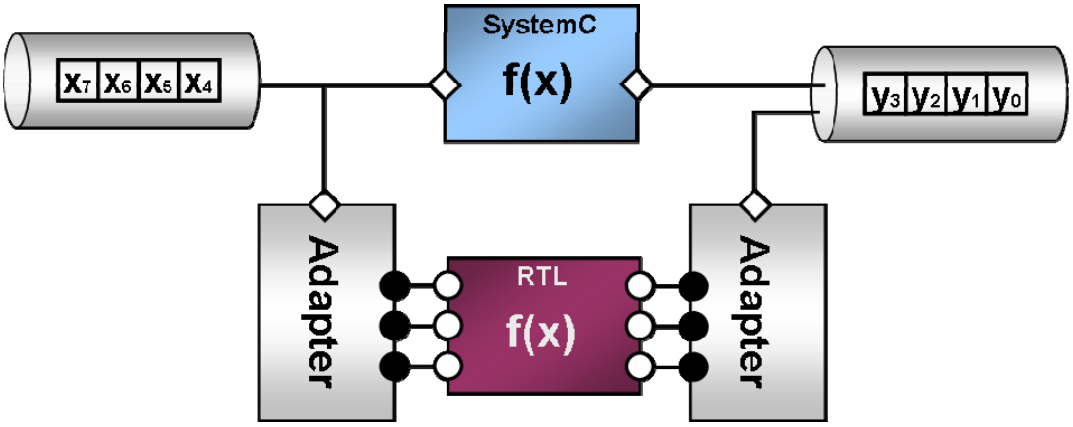

Figure 3: Co-simulating different versions of the same block

As shown above, a SystemC version of a model is used to verify an RTL equivalent in a self-checking configuration. The sc_fifo class would not allow connections to multiple ports, and would require four fifo channels overall, along with two more modules to write data at the input to the system and check it at the output of the system.

However, since the diagnostic channel allows for connections to multiple input and output ports, verification facilities can be embedded inside the channel. This also allows for a simpler top-level design file that requires fewer changes when switching between low-performance verification runs and high-performance characterization runs.

The function of the routing block is to select one of the many input sources and propagate data to each output source. In order to support multiple inputs, we developed the DataflowSelector, a class that contains a vector of fifos and provides methods to create and access them. On the output side of the channel is a DataflowSplitter, a class that has one input FIFO and copies that data to each of the output FIFOs contained within. A block diagram of the internal routing can be seen below in Figure 4.

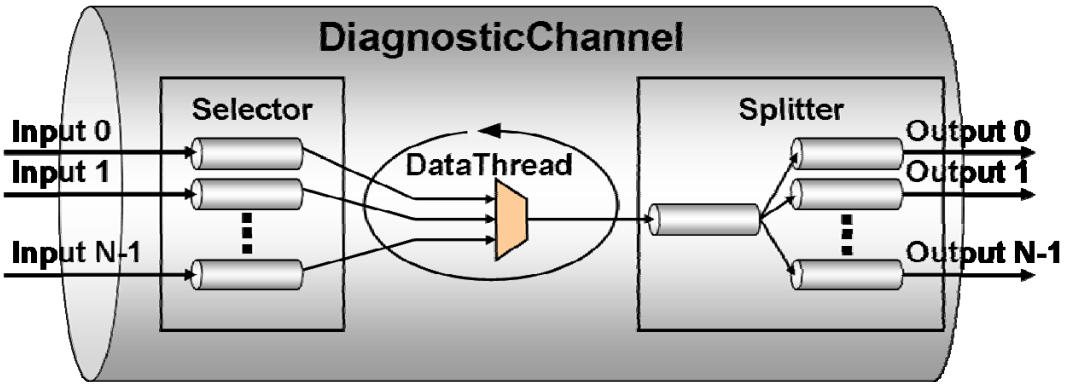

Figure 4: DiagnosticChannel routing block diagram

Data flows into the selector, and is multiplexed to the splitter by the DataThread. There is no limit on the number of inputs or outputs connected to the channel. The routing for each DiagnosticChannel is configured during runtime by a simple command script.

3.3 Timing

The DiagnosticChannel was designed to have configurable timing. Though adjusting timing does not necessarily affect the verification of flow-controlled data-path blocks, it can help provide insight into system performance. Currently, the implementation is limited to configurable latency, though it could easily be extended to add more features such as constrained random delay, bandwidth constraints, or other abstract parameters to model on and off-chip communication. Like routing, the latency is configured during runtime from a script.

3.4 Data generation

One of the key features of the DiagnosticChannel is the ability to inject data into the simulation from arbitrary data sources. We developed a module, the DataGenerator, that periodically generates data and a simple interface, DataPopulatorInterface, for data sources to implement. Both are described in more detail in the following sections.

3.4.1 DataPopulatorInterface

The DataPopulatorInterface is an interface used to populate data. Its definition is listed below:

```
class DataPopulatorInterface {
 public:
  virtual bool populate(DataInterface *data) = 0;
  virtual void reset() = 0;
}
```

The populate method takes a DataInterface pointer and returns a Boolean value indicating successful data population. The reset method is used to reset the populator to an initial state.

We have implemented some populators for internal use, which can generate data from any of the following sources:

- Files
- TCP/IP sockets
- The Standard C Library rand() function
- Custom LFSR implementation

Any class that implements the DataPopulateInterface can be used as a data source, providing a high degree of flexibility for end users.

3.4.2 DataGenerator

The DataGenerator is a simple module that periodically generates data from a class implementing the DataPopulatorInterface. It contains a sc_thread which is functionally similar to the following code:

```
// T must implement DataInterface
template <typename T>
void DataGenerator<T>::workerThread() {
```

```
    T data;
    while (1) {
      if (!stopped) {
        if (myDataPopInterface->populate(&data)) {
          outputPort->write(data);
          wait(dataPeriod, stoppedEvent);
        } else {
          stop();
        }
      } else {
        wait(startEvent);
      }
    }
  }
```

As can be seen in the example code, the thread creates a local copy of the data, attempts to populate it, writes it if that population succeeds, or stops itself if the population fails. If the DataGenerator is stopped, it waits for its startEvent to be notified, which results from the start() method being called.

3.5 Data inspection

In addition to injecting data, the DiagnosticChannel can be used to inspect data. The mechanism by which this happens is the ProbeInterface, which is listed below:

```
class ProbeInterface {
 public:
  virtual bool handleData(const std::vector<DataInterface *> &data) = 0;
}
```

The interface declares a single function that takes a vector of DataInterface pointers. Each element in the vector contains one datum from a single source. If a probe is configured to receive data from input 0 and input 3, the vector passed to handleData will be of length 2 and handleData and will be called for every pair of data arriving on the selector channels 0 and 3. While the current implementation of the DiagnosticChannel only contains a single DataGenerator, it can contain an unlimited number of data probes.

As with any interface in C++, all implementations that realize the ProbeInterface can be used interchangeably. This way, users can develop customized implementations as needed. We have implemented classes that perform the following common tasks:

- Write data to a file
- Compare values and log results to a file
- Calculate the bit error rate
- Write data out to a TCP/IP socket
- Plot data in Matlab

As mentioned earlier, any combination of these probes may be used as there is no limit to the number of probes contained in the DiagnosticChannel.

3.6 Example usage

This section illustrates some general examples of potential configurations of systems using the DiagnosticChannel. These represent the most common configurations we used while applying the channels to our designs. The example configurations are listed here:

- SystemC unit test
- SystemC system simulation
- SystemC/RTL co-simulation unit test
- SystemC/RTL substitution
- RTL/SystemC system simulation

Details for each configuration are in the following sections.

3.6.1 SystemC unit test

This configuration is used to initially verify SystemC blocks. Typically, there are some test vectors which are used to verify functionality. A block diagram is shown below:

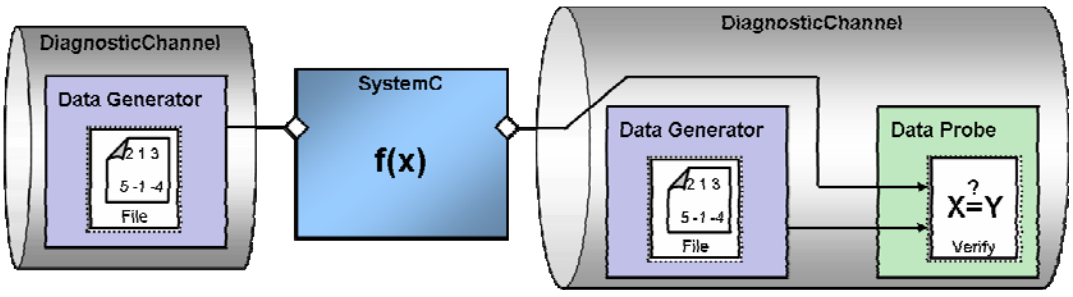

Figure 5: Example SystemC unit test configuration

In this configuration, the input channel generates data from an input file and the output of the functional block is compared against an expect file in the output channel.

3.6.2 SystemC system test

Once all of the SystemC models in the system are verified, it may be desirable to run a system test. The goals of the system test may be to get an accurate evaluation of system performance, or to identify behaviors previously undetected by unit tests. An example system test scenario is shown below:

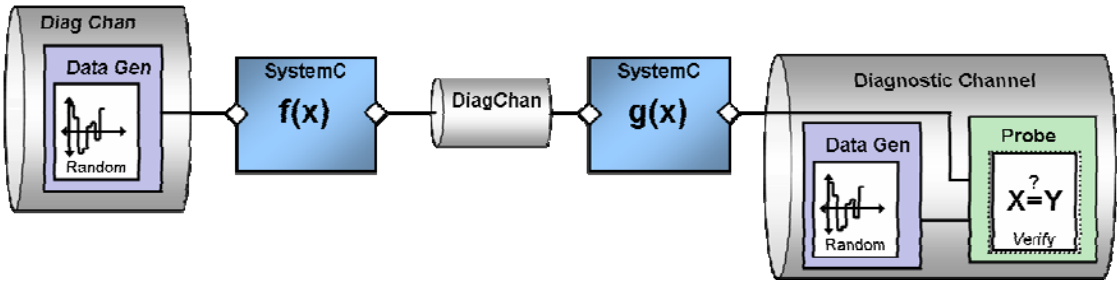

Figure 6: Example SystemC system test configuration

As can be seen, if f(x) and g(x) are inverse functions, random data can be used as the originating source and as the expect vectors, assuming that both output the same sequence of pseudo-random data.

3.6.3 SystemC/RTL Cosimulation Unit Test

In this scenario, an RTL block has been developed and is ready to be verified against its SystemC model. Assuming that there is a verified adapter to translate from the RTL signal interface to a transaction-level SystemC interface, verifying the RTL becomes a simple task. An example configuration is shown below:

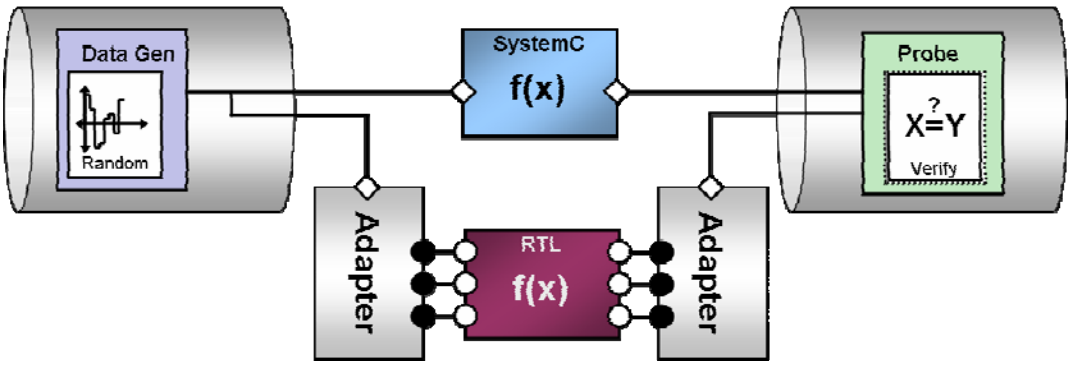

Figure 7: Example SystemC/RTL co-simulation unit test

In this configuration, it is unnecessary to have a reference test vector generator in the output channel because the SystemC serves as a golden reference model.

3.6.4 SystemC/RTL substitution

As RTL models are developed, they can be substituted for their SystemC counterparts in the system simulation. The advantages are twofold: verification is more robust than a unit test and the performance of the system simulation should greatly exceed the performance of an equivalent event-driven RTL simulation. Figure 8 shows an example:

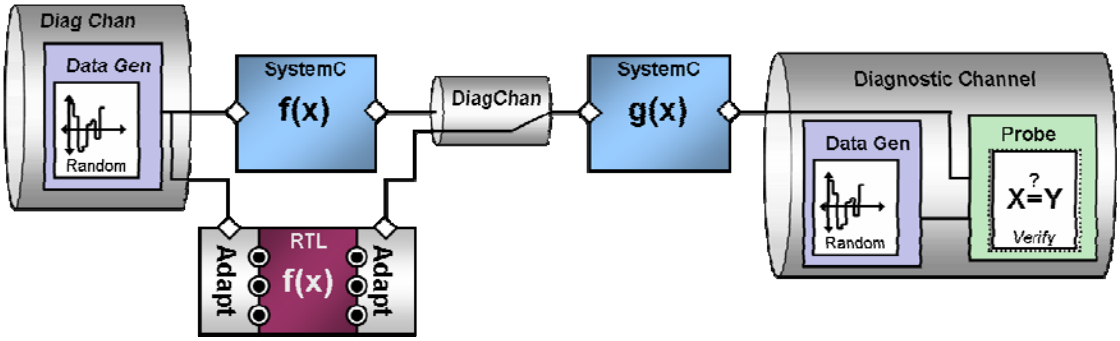

Figure 8: Example SystemC/RTL co-simulation substitution

Even though the diagram above doesn't explicitly depict it, there would most likely be a data probe inside the channel connecting f(x) to g(x) comparing the output of the SystemC version of f(x) to the output of the RTL.

3.6.5 RTL/SystemC system co-simulation

The RTL/SystemC system co-simulation use case is similar to the substitution case above, but instead of substituting a single RTL component for its SystemC counterpart, an entire RTL top-level is inserted as shown in Figure 9.

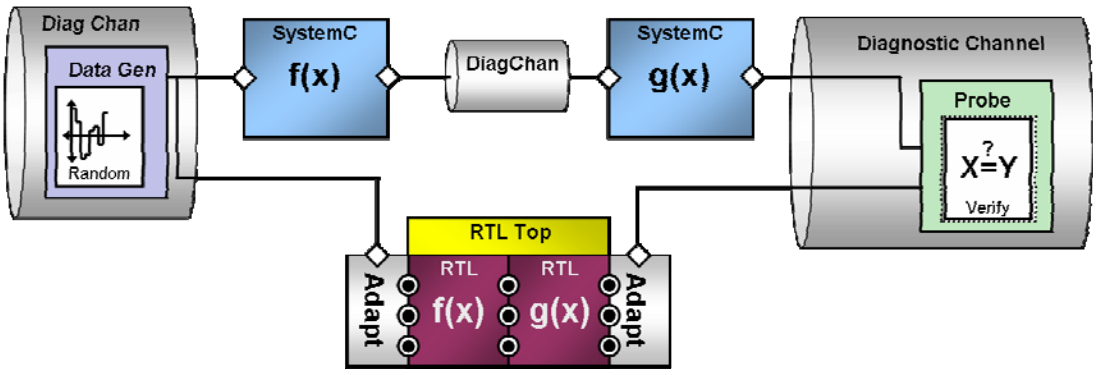

Figure 9: Example SystemC/RTL system co-simulation

Typically at this stage, each RTL block has been verified in a unit test, so the only new feature to verify is the connectivity of the RTL blocks inside the top level. Commercial

RTL simulators like VCS and visualization tools like Discovery Visualization Environment (DVE) can be used to trace signals internal to the RTL top-level block.

3.7 Summary

The DiagnosticChannel was designed to enable verification and modeling from within a single class in a SystemC environment. Unlike sc_fifo, the DiagnosticChannel can be bound to multiple ports templated on sc_fifo_in_if and sc_fifo_out_if. It can generate and inspect data and has configurable timing parameters. Though the DiagnosticChannel has an extensive feature set, users can implement the existing interfaces with new implementations that are tailored to their needs.

4 Results

This section contains initial results from the application of the DiagnosticChannel to a communication system. Impacts on the verification schedule are investigated as well as simulation performance overhead.

4.1 Verification of a high data rate wireless transceiver

As mentioned above, our application was a high data rate wireless transceiver. The transceiver had the following specifications:

- > 100Mbps sustained data rate
- Multi-mode, multi data rate operation
- Low SNR – sophisticated FEC required
- Portability – needs to be ported to other FPGA platforms

The implementation of the design spanned six Virtex2 pro p70's and two Virtex4 pros. The FEC alone spanned three FPGAs. The design had over 20,000 source lines of code.

We spent approximately 3 staff months developing the diagnostic channel and another 3 staff months developing other infrastructure and verification components for the new verification environment. The environment is being used internally across three projects and has been welcomed by senior staff as a valuable tool in significantly reducing ramp-up time for testbench development. Also, the ability to run a system simulation of the RTL with SystemC self-checking before deploying to the platform has been considered valuable. Although we tracked total hours spent integrating, verifying, and debugging each component for the pilot project, we do not have control data to

compare the results with. Regardless, we have had ample positive feedback to justify the continued use and development of the diagnostic channel and associated verification environment.

4.2 Performance impacts

The DiagnosticChannel adds verification and modeling features over conventional channels, but those features are at the expense of simulation performance. This section reviews the performance implications of the DiagnosticChannel.

4.2.1 Test Setup

The test system used to measure the overhead of the DiagnosticChannel was chosen to reflect the most common application of the DiagnosticChannel, component level verification. The component chosen was a fixed-point AWGN block based on the box-mueller algorithm and central limit theorem. This component was chosen because it is approximately average amongst SystemC blocks we have developed in terms of computational requirements.

A block diagram of the test setup can be seen below in Figure 10.

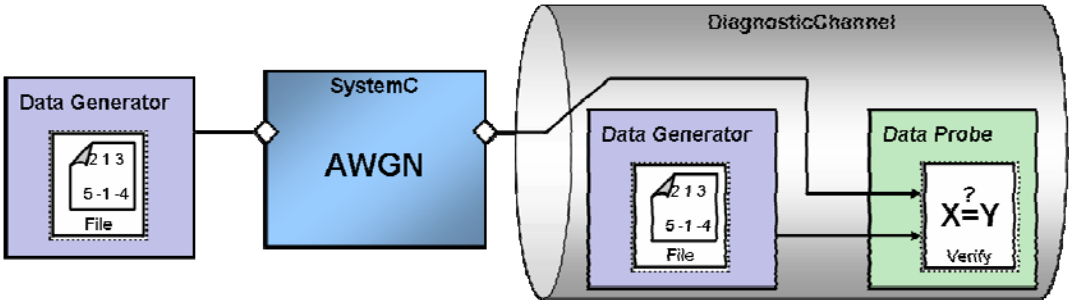

Figure 10: Performance test setup

The test system was composed of a testbench that read test vectors from a file, wrote them to the AWGN through a primitive channel and a DiagnosticChannel that verified the results against a second file. In the ordinary example, a primitive channel was used instead of a DiagnosticChannel and was bound to a module that consumed data by reading it.

The performance tests were compiled using optimization compiler flags and were run on a low-load server. The results displayed below show the averages of five runs per column. The unix command *time* was used to time the executable. *time* breaks down the

overall time into two categories, user and system time. User time is the amount of time the process spent executing instructions in user space. System time is the aggregate amount of time spent with the execution stack inside kernel-level functions. Elapsed time is the amount of time that elapsed between invocation and termination of the process. Elapsed time may not necessarily be equal to the sums of the user and system time as the process may either have been running multiple threads or child processes simultaneously, or may have spent some time blocked.

4.2.2 Test results

As can be seen below, the introduction of the DiagnosticChannel with data generation and data verification introduced approximately a 20% increase in elapsed simulation execution time.

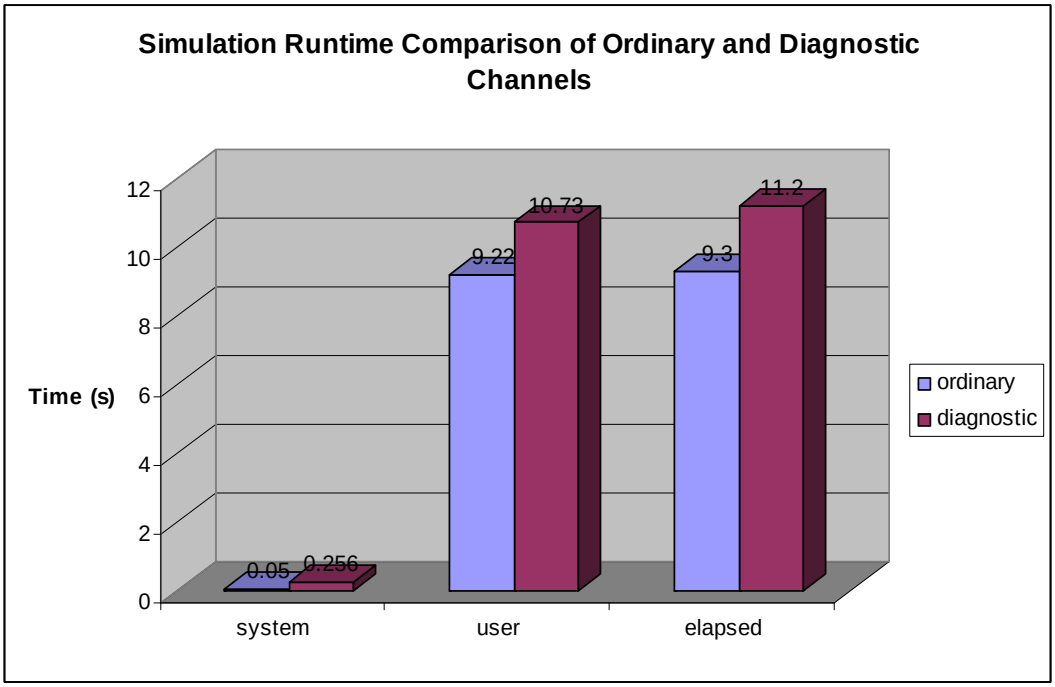

Figure 11: Simulation performance impacts

Because the test setup included an example module, the performance impact should not change significantly in simulations with more modules similar in complexity to the AWGN.

5 Future work

The DiagnosticChannel stands as a complete design, but it could be further enhanced. The next two sections describe possible enhancements.

- **Standardized timing control, constraints**
  The timing control for the DiagnosticChannel is the key feature for supporting approximate timed transaction level modeling. Though the current implementation allows for a simple latency, it could be made more configurable to allow for user-defined behavior. Such behavior could include constrained random latencies, latency spikes, bandwidth limitations, etc. This would allow for more sophisticated and accurate channel models.
- **Performance enhancements**
  Though the code for the DiagnosticChannel has been profiled and optimized accordingly, it could still benefit from performance enhancements. Specifically, the design of the selector and splitter could be revisited, eliminating the input channel from the splitter and using STL containers directly for storage instead of channels in both classes.

6 Conclusion

This paper discussed the DiagnosticChannel and its application to SystemC and RTL designs. The DiagnosticChannel is a class that provides sophisticated, convenient verification facilities to transaction-level SystemC models. As the results show, it contributes to shortened verification cycles at a reasonable performance cost. The DiagnosticChannel can be used as the primary verification tool in a design cycle, or as part of a larger verification solution.

As today's complex datapath architectures for System-on-a-Chip (SoC) and FPGA-based systems continue to grow in complexity, verification techniques will need to advance to meet the challenges. We feel that the DiagnosticChannel is a step in the right direction as it provides convenient verification and modeling capabilities without seriously reducing simulation performance.