# USING MATLAB AND SIMULINK IN A SYSTEMC VERIFICATION ENVIRONMENT

*Jean-François Boland, McGill University, QC, Canada, jfboland@macs.ece.mcgill.ca*
*Claude Thibeault, École de Technologie Supérieure, QC, Canada, thibeault@ele.etsmtl.ca*
*Zeljko Zilic, McGill University, QC, Canada, zeljko@macs.ece.mcgill.ca*

***Abstract*- Functional verification is a major bottleneck in today's design flow. Expensive and time consuming, the process of verifying complex designs is actually under serious reconsideration. We propose a new verification framework based on SystemC verification standard that uses MATLAB and Simulink to accelerate testbench development. Our major contributions are first a cosimulation interface between SystemC and MATLAB and Simulink, and next to enable functional verification of multi-abstraction levels designs. This paper presents the verification framework proposed and the cosimulation interface. An example shows how we used this verification framework in one of our project.**

## 1. Introduction

The verification task of today's multi-million gates designs has become the primary bottleneck in the design flow. Industry estimates are that functional verification takes approximately 70% of the total effort on a project. Rising gate count combine with greater design complexity has lead to much longer verification times. Time-to-market schedules are much harder to meet while project costs increase. According to a survey conducted by Collett International Research Inc. in 2002 [9], 60% of all tapeouts, that requires silicon re-spin, contained logic or functional flaws. Among those faulty integrated circuits, 82% had design errors. Incorrect or incomplete specifications, corner cases simply not covered during verification or changes in design specifications are a few causes of these flaws.

New verification techniques and methodologies are required to cut verification time and improve the quality of verification. Hopefully, hardware verification languages (HVL) come to the rescue, raising the testbench at a higher abstraction level. With specific verification syntax and faster simulation speed, HVLs improve performance and quality compared to RTL testbenches, thus reducing the time spent in verification.

In this work we focus our efforts toward the verification of digital signal processing (DSP) applications. Most signal processing designs begin with algorithmic modeling in the MATLAB and Simulink environment. Therefore, we believe that hardware verification could be significantly improved and accelerated by reusing these high level golden references models.

This paper presents a verification framework based on a novel cosimulation interface between SystemC and the MATLAB and Simulink environment to improve the hardware verification bottleneck. Our contribution tends to help hardware verification of DSP designs.

The paper is organized as follows. Section 2 discusses related work and section 3 present the context of this research. The proposed verification framework is presented in section 4. Section 5 presents the cosimulation interface. An example will highlight the verification framework in Section 6. Concluding remarks complete the paper.

## 2. Related Work

The MATLAB environment is a high-level technical computing language for algorithm development, data visualization, data analysis and numerical computing. One of the key features of this tool is the integration ability with other languages and third-party applications. MATLAB also included the Simulink graphical environment used for multi-domain simulation and model-based design. Signal processing designers take advantage of Simulink as it offers a good platform for preliminary algorithmic exploration and optimization.

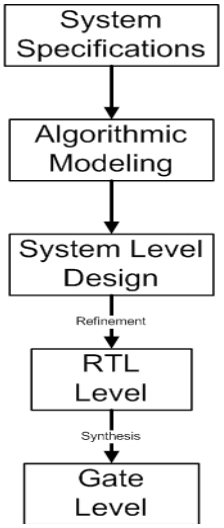


**Figure 1.** Simplify DSP design flow [8]

The next step in the design flow, after algorithmic development in MATLAB and Simulink, is the implementation phase. Typical DSP design flows, like the one shown in Figure 1, suggest that intermediate system level design take place to bridge the gap between high level modeling and low level RTL implementation. For this research work, SystemC language has been chosen for system level modeling. The Open SystemC Initiative (OSCI) is dedicated to supporting and advancing SystemC as an open source industry standard for system-level design. After several years of improvements, SystemC has become a concrete system level design and verification language.

To facilitate the transition between algorithmic modeling and system level design, an efficient interface is required between modeling tools. For the DSP design flow depict in Figure 1, we need a cosimulation interface

between the MATLAB and Simulink environment and SystemC simulation kernel.

Authors in [1] propose a solution to integrate SystemC models in Simulink. A wrapper is created using S-Functions to combine SystemC modules with Simulink. This wrapper initializes the SystemC kernel and converts Simulink data type to SystemC signals and vice versa. Simulation control is entirely handled by Simulink. Some extensions of the SystemC kernel are required for initialization and simulation tasks.

In [2], SystemC calls MATLAB using the engine library. MATLAB provides interfaces to external routines written in other programming languages. Using the C engine library, it is possible to share data between SystemC models and MATLAB. This simple working demo shows how to use the library to send and retrieve data from MATLAB workspace and plot some results. The main difference with [1] is with the simulation control: SystemC is now the master of the simulation and MATLAB operates as a slave process. Also, Simulink is not supported is this example.

In a similar way, MathWorks provide a commercial solution to close the gap between algorithmic domain and hardware design. Link for ModelSim [3] is a cosimulation interface that integrates MATLAB and Simulink into the hardware design flow. It provides a link between MATLAB and Simulink and Model Technology's HDL simulator, ModelSim. This interface makes possible the verification and cosimulation of RTL-level models from within MATLAB and Simulink. As oppose to the two previous techniques, there is no support for system level languages like SystemC.

These approaches [1, 3] all try to reduce the barrier that exist between higher level modeling and existing hardware design flow. While [3] is a fully functional commercial tool for RTL verification, [1, 2] suffer from their embryonic stage (i.e. incomplete solution for hardware design and verification).

Our project tries to push the idea a step further than just a cosimulation interface; it is a complete verification platform. This one uses MATLAB external interfaces, similar to the example describe in [2], to exchange data between SystemC and Simulink. Once this link is established, it opens up a wide range of additional capability to SystemC, like stimulus generation and data visualization.

The first advantage of our technique is to use the right tool for the right task. Complex stimulus generation and signal processing visualization are carried out with MATLAB and Simulink while hardware verification is performed with SystemC verification standard. The second advantage is to have a SystemC centric approach allowing greater flexibility and configurability.

The main contribution of this work is to propose a cosimulation interface for SystemC and the MATLAB and Simulink environment. The following sections will show how this interface can speed up DSP hardware verification while preserving verification quality.

## 3. Context of Work

The verification framework proposed integrates seamlessly in any DSP design flow, which uses MATLAB/Simulink and SystemC, like the one shown in Figure 2. This flow is actually used by our design team for one of our software defined radio project. We will go quickly through this flow to clearly identify the verification requirements and how we can build an efficient verification platform from this.

The design under consideration targets an FPGA and CPU based target. First, system specifications are written with the unified modeling language (UML) and text-based documents. Next, the designers begin the algorithmic modeling phase with MATLAB and Simulink to explore and optimize a variety of algorithms for the DSP design. The result of this design step is a golden reference model of the system that will be used later by the verification team. Then, system level modeling begins with SystemC and C++ languages. The golden reference is used at this stage to validate that the system level model still meets initial specifications (did we build the right product?). This validated model is taken through the partitioning process to generate hardware and software specifications. After that, regular hardware and software design flows are used to produce the final FPGA bit stream and CPU assembly program.

To ensure that the system has been correctly implemented into hardware (did we build the product right?) functional verification will be performed throughout the hardware design flow. Using the SystemC verification standard (presented in 4.1), a verification platform has been created. The main novelty with this platform is to use MATLAB and Simulink to assist the testbench for the verification of the design at different levels of abstraction. In other words, the golden reference model, previously created with Simulink, is used to produce a data generator and data analysis sub-modules. Then, through a cosimulation interface, these Simulink models will be used by the SystemC tesbench to verify the hardware design. It is also important to note that as the design is refined down to bit-true representation, the golden reference in Simulink will also be updated to reflect fixed bit depth quantisation. The following section presents the proposed verification framework.

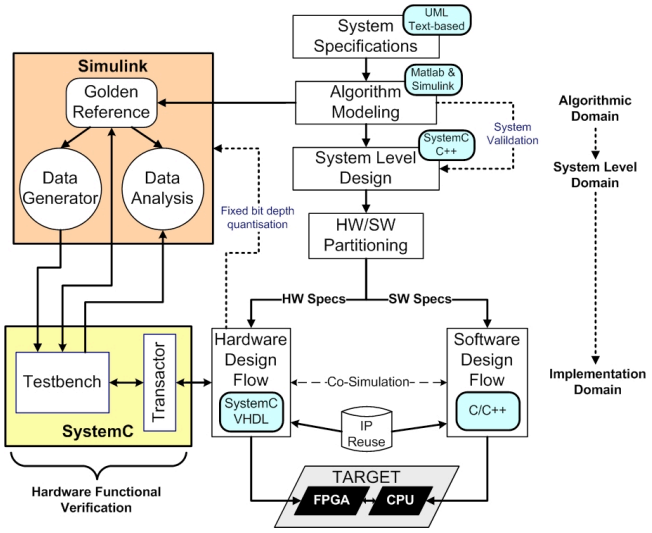


**Figure 2.** Complete DSP Design Flow

## 4. Proposed Verification Framework

DSP applications frequently requires complex environment to efficiently simulate and verify the design. For example, to completely verify our multi-equalizer architecture, real world telecommunication stimuli are needed to truly exercise the design. In addition, the criteria used to evaluate performances are quantitative measures such as least-mean-square, bit-error rate, and others. It is very cumbersome and time consuming to implement theses criteria directly into HDL or C++ testbenches. The verification framework showed in Figure 3 answers those issues. There are three main components:

- SystemC verification standard
- Transaction based verification
- MATLAB and Simulink

The core element of the verification platform is SystemC. It is important to understand at this point that SystemC language can be use for both design and verification. Test code is written with SystemC (and the verification library SCV) to produce scalable and intelligent testbenches. Since the design under verification (DUV) can be represented at multiple levels of abstraction, transactors (TR) are used to bridge the abstraction gap. Creating and maintaining testbenches in a higher abstraction level is inherently faster than HDL simulator (SystemC is compiled and run much faster than interpreted HDL) and can be reuse across abstraction levels.

Next we have MATLAB and Simulink in the verification flow. The primary strength of MATLAB and Simulink is not hardware verification. As mentioned, this tool is intended for algorithm development, numerical computing and data visualization. The verification framework takes this into consideration and uses Mathworks' tool to assist the SystemC testbench. This way each tools and languages are used for their intended purpose.

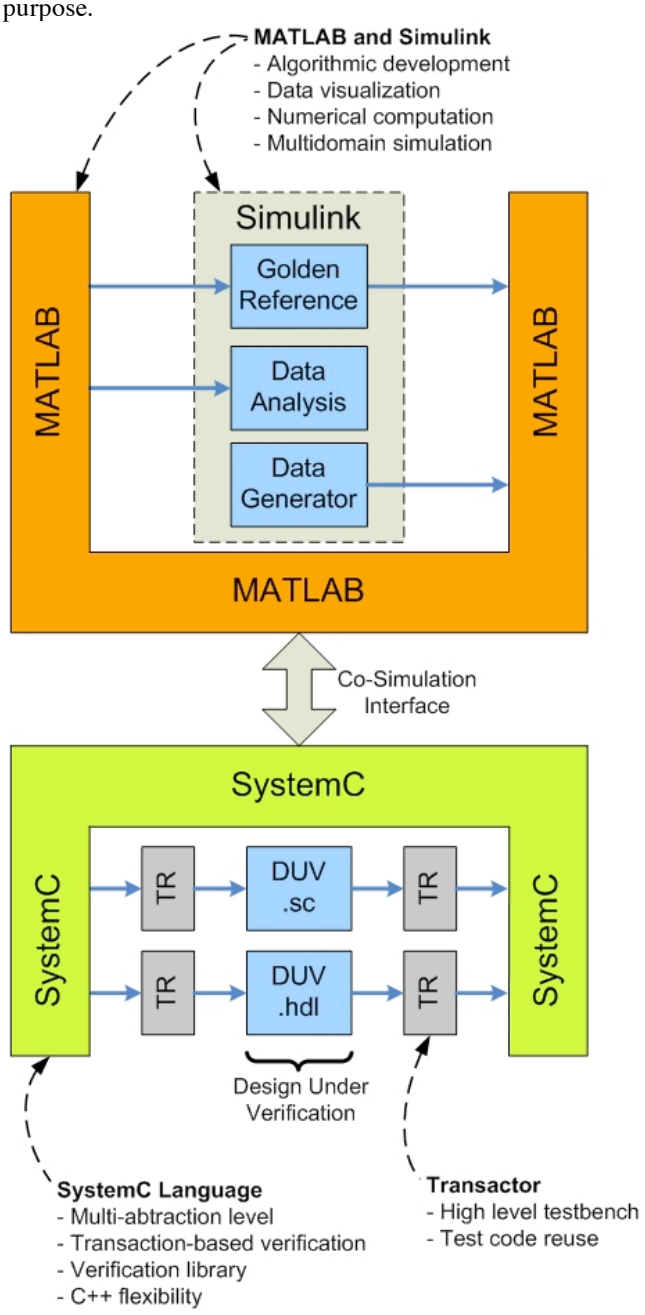


**Figure 3.** Verification Framework

### *4.1. SystemC Verification Standard*

A working group within the OSCI, which consist of several EDA companies, semiconductors developers, and academic institutions, recently launched the SystemC Verification standard. The goal was to define a set of classes within SystemC that would provide a basis for developing various verification methodologies. The result is a verification library called SCV [6] freely available from OSCI web site.

The verification library is composed of the following features:

- **Data introspection:** manipulation of arbitrary data types.
- **Randomization:** generation of random values through the 'scv_random' class that support advanced seed management and generation algorithm selection.
- **Constraints for randomization:** creation of constraint expressions, with the 'scv_constraint_base' class, to specify the range of legal values.
- **Weight for randomization:** possibility to bias the random values generation process, with the 'scv_bag' class, so that some values are generated more often than others.
- **Transaction-based verification:** modeling style for test bench with transactors and transaction recording through 'scv_tr_db', 'scv_tr_stream', and 'scv_tr_stream'.

Using SystemC for hardware verification allow the verification of designs written at any abstraction level. Transactors and transaction based verification make this possible.

### *4.2. Transaction-based Verification*

Transaction based verification [7] raises the level of abstraction from signals to transactions, thus easing the development of reusable test benches. Figure 4 shows typical transaction based verification architecture. The testbench is separated into two layers: the test code and the transactor. The test code is written at a higher level of abstraction than the DUV and the transactor is the mechanism that translates the test from transaction to signals activity.

The verification framework proposed in Figure 3 takes advantage of transaction based verification. Transactors are implemented with SystemC to verify the DUV. The test code is thus easier to reuse and run faster than traditional HDL testbenches.

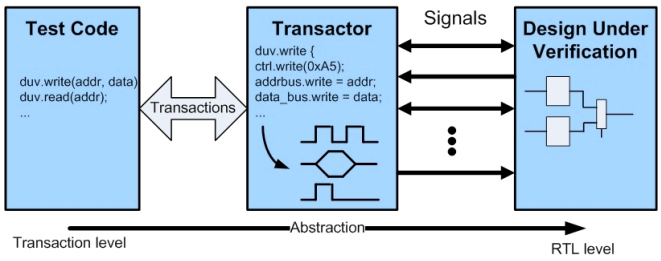


**Figure 4.** Transaction based verification

### *4.3. MATLAB and Simulink for Verification*

Since MATLAB and Simulink are used early in a DSP design flow, it makes sense to reuse as much as possible some components of these high level models to improve lower level hardware verification. The first objective here is to simulate at lower abstraction level only those portions of the system that are actually going to be synthesized into hardware. This results in faster simulation execution because the rest of the system can run at higher abstraction level. The second objective is to provide additional flexibility and robustness to SystemC testbench with pre-validates data generator and data analysis modules. Real life data can be quickly generated with the Simulink models using the various Blocksets available in the Simulink environment. Moreover, output data that came from the DUV can be forward by the testbench to Simulink. The verification engineer can now uses the graphical tools, like scopes, X-Y graphs or other mathematical operations of Simulink to further analyze the response of his system. One last benefit of using Matlab and Simulink in the verification flow is for golden reference. A Simulink golden model can be used as a reference model by the verification system to compare the expected behaviours.

## 5. Cosimulation Interface

Using MATLAB and Simulink to assist the SystemC verification framework relies on cosimulating the two environments. The cosimulation interface must provide adequate capabilities and reasonable simulation speeds. Our solution is based on the MATLAB engine external interface. Data is exchanged directly through share memory to obtain the optimal speed while keeping the interface and the protocol as simple as possible. This section presents the implementation details of this interface.

### *5.1. SystemC calls MATLAB*

The foundation of our interface is the data transfer between SystemC and MATLAB. For that purpose, we use the 'engine' library available with MATLAB. The work described in [2] uses a similar interface. The difference is that we employ this interface in a verification context. In addition, we will see in section 5.2 how we significantly improved the interface to communicate with Simulink.

The 'engine' library contains nine routines for controlling MATLAB computation engine from a C program. On Microsoft Windows, the engine library communicates with MATLAB using a Component Object Model (COM) interface (UNIX uses pipes). Table 1 summarizes these routines.

**Table 1.** MATLAB 'engine' library

| C Routines | Description |
|---|---|
| eng{open\|close} | Start/Close MATLAB engine |
| eng{get\|put}Variable | Get/Put a MATLAB array from/to the engine |
| engEvalString | Execute a MATLAB command |
| engOutputBuffer | Create a buffer to store MATLAB text output |
| engOpenSingleUse | Start a MATLAB engine session |
| eng{get\|set}Visible | Show or hide MATLAB engine session |

A SystemC module employs these routines to remotely control MATLAB and exchange data back and forth between SystemC and MATLAB workspace. Figure 5 shows a code snippet of the *matlab sc_module* that we uses for the cosimulation interface.

```
1 // MATLAB/Simulink Interface
2
3 #include "systemc.h"
4 #include <engine.h>   // Matlab engine header
5
6 // SystemC Module that create the interface to MATLA
7 SC_MODULE(matlab){
8   ...
9
10  Engine *ep;       // Pointer to MATLAB engine
11
12  // Module Constructor
13  SC_CTOR(matlab)
14  {
15    SC_THREAD(mat_sync);  // Thread function to sync
16    sensitive_pos << CLK; // Call at each rising clo
17
18    // Start MATLAB engine
19    if(!(ep = engOpen(""))){
20      cout << "\nCan't start MATLAB engine" << endl;
21      exit(-1); // Exit on error
22    }
23    engSetVisible(ep,1);  // Makes Matlab session wi
24
25    wd_path = get_wdpath();     // Get the working d
26    engEvalString(ep, wd_path); // Change MATLAB wor
```

**Figure 5.** Code fragment of the SystemC module that calls MATLAB

### *5.2. SystemC calls Simulink*

To exchange data between a Simulink model and a SystemC module, the cosimulation interface must integrate a bridge between MATLAB and Simulink. This bridge is build with two Simulink S-Functions. An S-Function is a computer language description of a Simulink block. It uses a special calling syntax so we can interact with Simulink solvers. For the needs of our bridge, we created two C++ S-Function. Figure 6 gives an overview of how a Simulink model and S-Functions are connected together.

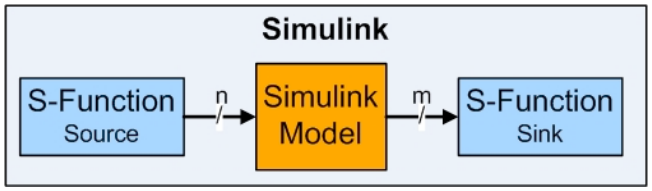


**Figure 6.** Simulink model with n-input source and m-output sink S-Functions blocks

The 'source' S-Function read data from MATLAB workspace (previously written by SystemC) and drives the corresponding signals in Simulink. On the other side, the 'sink' S-Function reads its inputs and update the corresponding variables in MATLAB workspace so it can be read back by SystemC. Both S-Functions include a configurable burst mode option to support variable width data burst transfers.

### *5.3. Simulator Synchronization*

The representation of simulation time differs significantly between SystemC and Simulink. SystemC is a cycle-based simulator and simulation occurs at multiples of the SystemC resolution limit. The default time resolution is 1 picoseconds; this can be changed with function *sc_set_time_resolution*. Simulink maintains simulation time as a double-precision value scaled to seconds. This time representation accommodates continuous and discrete models. Our cosimulation interface uses a one-to-one correspondence between simulation time in Simulink and SystemC. The Simulink solver is set to discrete fixed-step type, so one time step in Simulink correspond to one tick in SystemC.

As mention previously, SystemC is master of the simulation. Simulink is controlled from SystemC through the cosimulation interface. Using MATLAB commands *set_param* and *get_param* (with the appropriate argument) it is possible to have a complete external control over Simulink. SystemC uses *set_param* to start, stop, and continue Simulink execution. Simulation is suspended at each time step by the S-Function. For that purpose, the same command (*set_param)* is used at the end of the S-Function, but with the 'pause' argument. On the other hand, SystemC request Simulink status with the command *get_param* to synchronize both simulators. Figure 7 provides the SystemC code snippet of the command *get_param*.

```
27  // Check if simulation is running
28  engEvalString(ep, "get_param('fir16b','SimulationStatus')");
29  mxB = engGetVariable(ep, "ans");
30  string = mxArrayToString(mxB);
31  cout << "\nMatlab: simulation is " << string << endl;
```

**Figure 7.** SystemC check if Simulink simulation is running

### *5.4. Data Type Converstion*

The MATLAB language works with only a single object type: MATLAB array. These arrays are manipulated in SystemC using the 'mx' prefixed application programming interface (API) routines included in the MATLAB engine. This API consists of over 60 routines to create, access, manipulate, and destroy *mxArrays*.

## 6. Experimental Results

The verification framework is currently used in one of our project for the verification of a multi-equalizer architecture; one of the key component in a software defined radio receiver [11]. Simulink is used to model the external environment required to exercise the multi-equalizer design. Figure 8 shows the Simulink model that drives a SystemC representation of this multi-equalizer design. Instead of creating time consuming stimuli in SystemC, the data generator, the transmitter and fives different telecommunication channels are modeled with Simulink using the Communications Blockset. Via existing communication channel models and Matlab build-in mathematical functions, we were able to set up a complete real-word stimulus generator in less than a day; while it can take over a week to manually program the same models in C++. In addition, Matlab/Simulink Blocksets have already been validated making data generation less error prone. As a result, the overall quality of our functional verification gets improved.

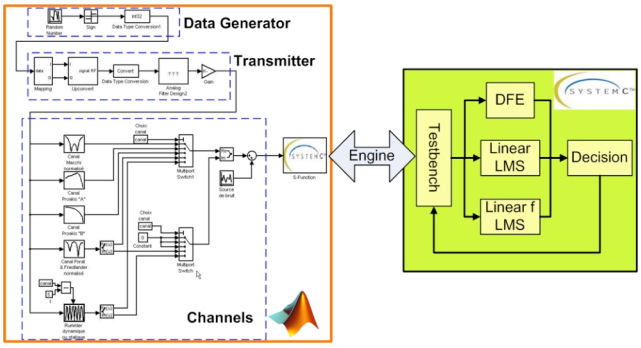


**Figure 8.** Example of Simulink used as a data generator for a SystemC design

Figure 9 shows the development environment used to create the cosimulation interface. SystemC and S-Function programming is done with Microsoft Visual C++. With the built-in debugger it is possible to attach the MATLAB process to the current SystemC simulation. This allows breakpoints to be easily inserted in the S-Function and SystemC programs to trace the execution. The same environment is employed to create the SystemC testbench and transactors.

## 7. Conclusion

In this paper a verification framework based on a novel cosimulation interface between SystemC and the MATLAB and Simulink environment has been presented. This platform is intended to help hardware verification of DSP designs. Using MALTAB and Simulink to assist a SystemC verification environment we are able to significantly improve the hardware verification bottleneck. A more complete testbench can be build up in a shorter period of time than with traditional HDL. Also, by means of transactors, the verification environment can be connected to the design at multiple levels of abstraction. This way verification does not start anymore at the end of the development cycle where bugs found at this late stage are extremely difficult and expensive to resolve.

## Acknowledgements

The authors wish to thank PROMPT Québec and the Regroupement Stratégique en Microélectronique du Québec (ReSMiQ) for their financial support. Special thanks go to the MAME project team for testing the cosimulation interface.

## References

1. F. Czerner, J. Zellmann, "Modeling Cycle-Accurate Hardware with Matlab/Simulink using SystemC", 6th European SystemC Users Group Meeting (ESCUG), October 2002
2. C. Warwick, "SystemC calls MATLAB", MATLAB Central, March 2003, http://www.mathworks.com/matlabcentral/
3. "Link for ModelSim 1.2; Cosimulate and verify VHDL and Verilog using ModelSim", The MathWorks, 2003
4. J.F. Boland, C. Thibeault, Z. Zilic, "Efficient Multi-Abstraction Level Functional Verification Methodology for DSP Applications", in Proc. of Global Signal Processing Expo and Conference, Santa Clara, California, September 2004
5. Arun Mulpur, "System-Level Verification of Signal Processing Applications on ASICs and FPGAs", in Proc. of Global Signal Processing Expo and Conference, Santa Clara, California, September 2004
6. "SystemC Verification Standard Specification version 1.0e", May 2003, http://www.systemc.org
7. Dhananjay S. Brahme, Steven Cox, Jim Gallo, Mark Glasser, William Grundmann, C. Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea, Karl Whiting, "The Transaction-Based Verification Methodology", Cadence Berkeley Labs, Technical Report #CDNL-TR-2000-0825, Cadence Design Systems, August 2000
8. J.F. Boland, A. Chureau, C. Thibeault, Y. Savaria, F. Gagnon, Z. Zilic, "An Efficient Methodology for Design and Verification of an Equalizer for a Software Defined Radio", in Proc. of 2nd Northeast Workshop on

Circuits and Systems, Montreal, Quebec, June 2004, pp. 73-76.
9. Collett International Research Inc., "2002 IC/ASIC Functional Verification Study", 2002.
10. The Open SystemC Initiative (OSCI) http://www.systemc.org
11. J. Mitola, "The Software Radio Architecture", IEEE Communications Magazine, Vol. 44, No. 5, pp. 26-38, May 1995
12. Grant Martin, "SystemC's Role in a Multilingual World", 8th European SystemC Users Group, Stuttgart, November 2003

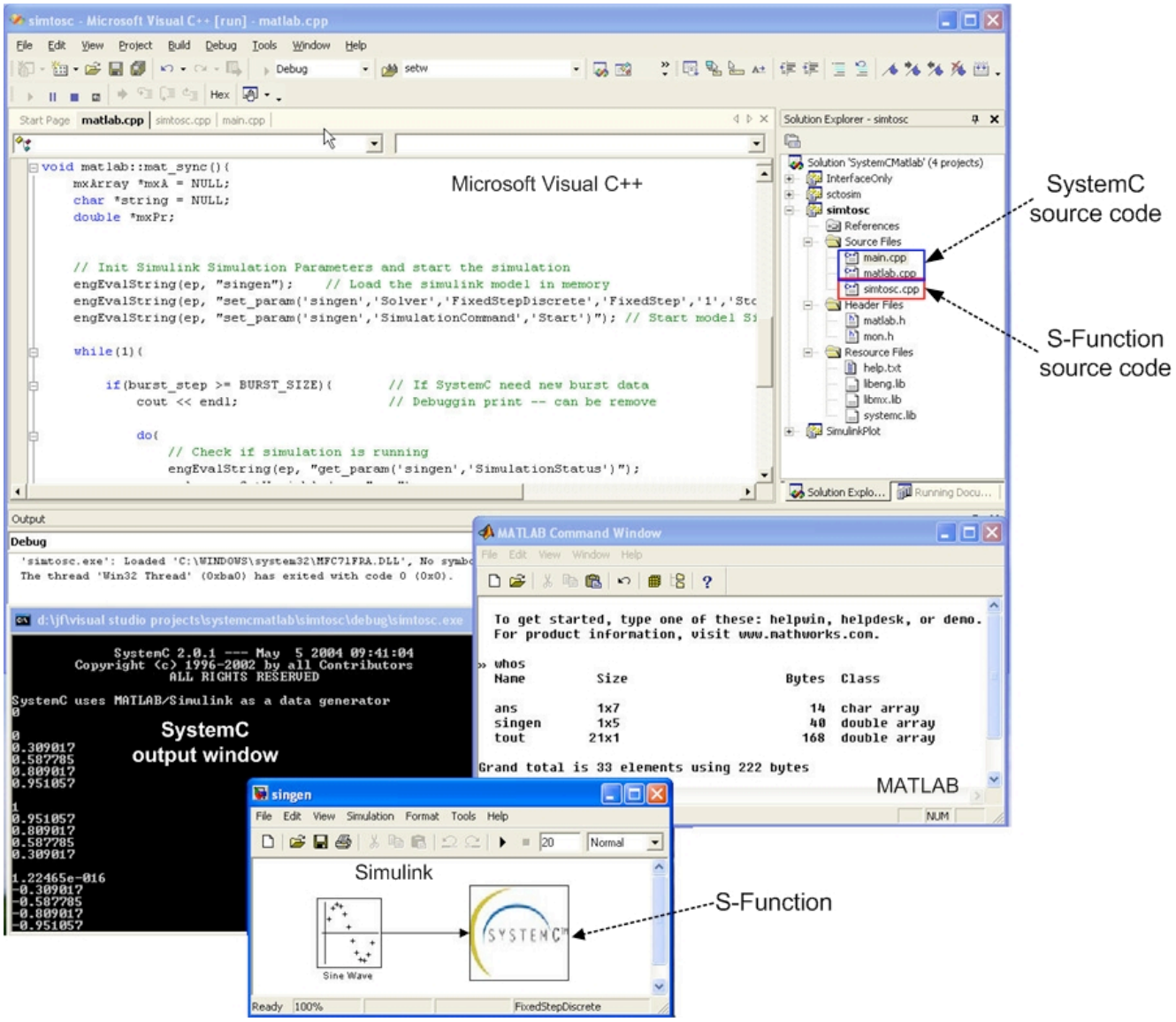


**Figure 9.** Development environment used to create the cosimulation interface