A Mixed Language Fault Simulation of VHDL and SystemC

Silvio Misera, Heinrich Theodor Vierhaus, Lars Breitenfeld, André Sieber
Brandenburg University of Technology Cottbus, Computer Engineering Department
<sm, htv, lars.breitenfeld, asieber@informatik.tu-cottbus.de>

Abstract

Fault simulation technology is essential key not only to the validation of test patterns for ICs and SoCs, but also to the analysis of system behavior under fault transient and intermittent faults. For this purpose, we developed a hierarchical fault simulation environment that uses structural VHDL models at the gate level, but is able to model embedded blocks in C++. With SystemC becoming a de-facto standard in high-level modeling, a simulation approach had to be developed which makes effective use of SystemC technology by encapsulating such "threads" into the fault simulation environment. Furthermore, it can be shown that SystemC allows the modeling of complex transistor-level structures, for which equivalent gate-level representations are not adequate.

1. Introduction

Systems on a Chip (SoCs) have been a challenge to IC test technology for the last decade because of their inherently high complexity and the limited test access to internal circuit nodes [13]. More recently, SoCs implemented in deep sub-micron technologies need to be designed for handling transient fault effects, introduced by "single event upsets" (SEUs) from particle radiation, but also by electromagnetic interference. In either case, a fault simulation tool is needed either for the validation of test patterns, but also for the validation of fault-tolerant designs and architectures.

Fast and efficient gate-level fault simulators as parts of automatic test pattern generation (ATPG) environments have been developed throughout the 1990s [2] and have become a standard technology. Unfortunately, such simulators are typically not able to simulate the propagation of fault effects over functional blocks that are not available as gate level models. In such cases, behavioral simulators are used instead, which typically have only a limited capability to handle effective and realistic fault models [7, 10]. This is the reason why we implemented FIT [1] as a hierarchical fault simulation environment, which uses a mixture of structural VHDL model at the gate level and C++-models for large-size macros. Furthermore, FIT supports non-trivial fault models such as single-event upsets (SEUs).

While C++ is relatively effective in some cases, it cannot handle typical properties of hardware such as concurrency. SystemC [3] has recently emerged as a C++-compatible extension, which is able to model such effects. For a real system-level fault simulation, concurrency becomes a must. Therefore the compatibility of SystemC concepts and structures with the basic approach followed with FIT had to be analyzed. Unlike C++, which is just a means of description, SystemC provides its own simulation environment. Combining FIT with SystemC therefore meant either a transfer of FIT into a SystemC environment or a coupling of two independent simulators. Coupling technologies have become popular using either the PLI/VPI- or the VHPI interface standard [8]. However, recent developments [4, 9] seem to favor a single simulation kernel for performance reasons. They remove the SystemC simulation kernel and transfer SystemC code into internal data formats.

None of these two approaches is satisfactory. In simulator coupling, a certain freedom in modeling is paid by longer simulator run times. The second alternatives is faster, but it depends on vendor-specific support of SystemC interfaces and libraries.

The new approach presented in this paper uses the simulator facility of the SystemC kernel for enhanced speed, but incorporates SystemC simulation tasks as "thread" processes into the FIT simulation processes. In this combination, we can combine effective modeling of fault conditions with reasonable performance of the simulator.

Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD'06)
0-7695-2609-8/06 $20.00 © 2006 IEEE

2. Modifications of the SystemC-Library

In a simulation environment that links two independent simulators, the mutual switch between simulator cores typically means a context switch to be performed by the operating system of the host computer. Typically, a context switch will take up to ten thousands of CPU cycles. On the other hand, only a change from one thread of execution to another one within the same run time environment, which is typical for an organisation based on light-weight processes or "threads", is much less expensive. We would expect a relation of at least ten or more.

For the simulation environment, we have to remove the original SystemC simulation process and organize at as a thread. The critical piece of code is shown in Figure 1.

```
void *t_test(void *param)
{
    int status = 0;
    try
    {
        //pln();
/* old main commands
* follows
* here
*
*/
   // return status;
        return NULL;
}


int create_SystemC_Thread(){
  pthread_t thread;
  int *param = 0;
  int make_Thread = pthread_create(&thread,
      NULL, t_test, param);
  if(make_Thread != 0){
    printf("Thread was not created!\\n");
  }
  pthread_detach(thread);
  return(0);
}
```

Figure 1. Modifications in sc_main.cpp

The original `main()`-call is substituted by a function `t_test()` and slightly modified. The SystemC-based simulation is now started by the function `create_SystemC_Thread()`. The thread itself is executed as a Posix-thread [11] in order to have a platform-independent implementation. This version can be run in a Windows or a LINUX environment, which is typically used for parallel processing.

Some simulation results that were taken from the example of a ripple carry adder are shown in Table 1.

Table 1: Simulation results for a ripple carry adder

model	M1 (counter)	M2 (random)	M3 (bit change)
a16_sc_b	5,2 s	7,2 s	5,2 s
a16_dll_b	4,3 s	4,9 s	4,2 s
a16_sc_r	5,9 s	16,7 s	5,3 s
a16_dll_r	6,0 s	16,4 s	5,5 s

The adder was simulated with different sets of input patterns. Example M1 is based on incrementing of input values, M2 is made from a random pattern sequence of the same length that results in massive input changes between patterns, and M3 is a series of alternating two input bits between steps. The experiment was performed twice, since the actual run time is massively influenced by the actual status and the work load of the host computer.

Apparently, the DLL-based approach (mixed language simulation) is significantly faster in the first case, when the host CPU has a higher load.

3. Coupling a VHDL Fault Simulator and the SystemC model

Simulator coupling is done by interface descriptions. Every call of the SystemC environment contains a function call that installs the SystemC testbench. No modifications to the SystemC model itself are necessary.

The structure is depicted in Figure 2 . The FIT fault simulator executes the overall control function. It reads three files that contain the patterns to be applied to the circuit inputs, the fault list and a configuration file that controls the simulation process. For a single fault simulation, the fault is first injected and then simulated with all available input data. This process generates another file containing the simulation results. The interface has two essential tasks to perform.

On the one hand, it uses the five function calls of the fault simulator to a High Level Description. With the function calls the interaction is ensured between FIT and a High Level Description (HLD).

Besides the initialization, reset and destructor function the two calculation functions are of special significance for the simulation. With the first calculation function (figure 3) a HLD is asked to execute a calculation. The second calculation function provides the calculation results.

On the other hand, the interface contains the testbench of the SystemC project which is otherwise usually contained in the `main.cpp` of a SystemC project.

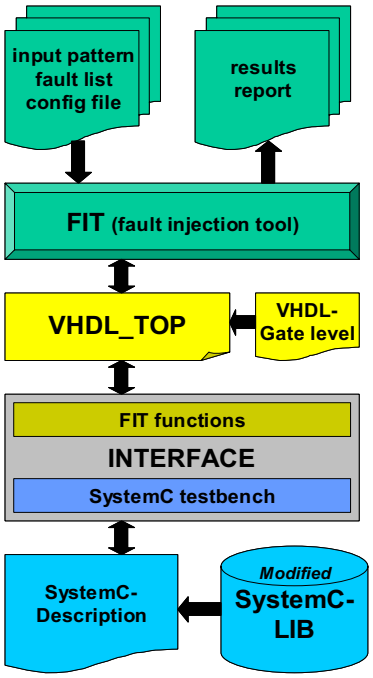

Figure 2. Mixed-Language-Project

Figure 3 shows a few lines of the interface description, indicating how FIT can start and synchronize a SystemC- simulation process and synchronize it.

```
//first calculate the component & store the
//result temporary for one step

extern "C"  char* calculate_1(int identify,
              int NrOfInputs, int* Inputs){
        int argc = 0;
        char** argv = NULL;
        if(NrOfInputs != NoGI) {
           return strdup( "Inputs are not
                    correct" );
        }

        globalInputs = Inputs;
        if(PassFlag == 0){
                GoOn =1;
                create_SystemC_Thread();
                while(GoOn == 1 ){
      Sleep(0); }
       }else{
                GoOn = 1;
    while(GoOn == 1 ){ Sleep(0); }
    PassFlag = 1;
    return 0;
}
```

Figure 3: FIT procedural call in MyInterface.cpp

When the simulation run is started for the first time, the PassFlag is at the initial value of 0, and the function `create_SystemC_Thread()` from the modified SystemC.lib is called. Any further call does not initialize new threads, as the flag bit has been set to 1. The `GoOn` variable does the synchronization between the SystemC-Simulation and FIT. `GoOn == 1` means the end of a simulation step. Then the `Sleep(0)`- instruction of the operating system scheduler can release the thread for further use.

The testbench for the SystemC-Model is depicted in Figure 4.

```
// testbench for SystemC model

SC_MODULE (testbench){
 sc_out<sc_logic>A0_p,A1_p,B0_p,B1_p,CIN_p;
 sc_in<sc_logic> SUM0_p,SUM1_p,COUT_p;

 SC_CTOR (testbench){  SC_THREAD (process);
 }
 ~testbench(){ }

// Define the functionality of the
// "process" thread.
void process(){
  while(1){
    if( globalInputs[0] == LOGIC_0)
        {A0_p = SC_LOGIC_0;}
    . . .
    if( globalInputs[4] == LOGIC_1)
        {CIN_p = SC_LOGIC_1;}
    wait (10, SC_NS);
    if(SUM0_p == SC_LOGIC_0)
        {globalOutputs[0] = LOGIC_0;}
    . . .
    if(COUT_p == SC_LOGIC_1)
        {globalOutputs[2] = LOGIC_1;}
    GoOn = 0;
    while(GoOn == 0){ Sleep(0); }
  }
}


int sc_main(){
  sc_signal<sc_logic> A0_s,A1_s,B0_s,
        B1_s,CIN_s,SUM0_s,SUM1_s,COUT_s;

  addtop add1 ("Addtop");
  testbench test1("TestBench1");

  add1 << A0_s << A1_s << B0_s << B1_s
    << CIN_s << SUM0_s << SUM1_s <<
  COUT_s;
  test1 << A0_s << A1_s << B0_s << B1_s
    << CIN_s << SUM0_s << SUM1_s <<
  COUT_s;

  sc_start(-1,SC_NS);
  return(0);
}
```

Figure 4. Testbench for the SystemC-Modell in MyInterface.cpp

In the `SC_MODULE (testbench)` inputs and outputs are fixed first, followed by the constructor and the destructor. The `process()` method performs the adaptation and the transfer of FIT-data to SystemC and vice versa. The GoOn variable is used to tell the OS scheduler that the `testbench::process()` has come to an end.

The `sc_main()` routine, is, as usual, taken for the declaration of signals, the instantiation of the testbench and the creation of links before the SystemC-simulator is started.

So far, FIT is essentially a fault simulator that works on structural VHDL netlists with comfortable fault injectjob mechanisms, which can interface either a C++-based behavioral model or a SystemC simulation process for hardware macros.

4. Fault Simulation Refinements

First investigations on fault simulation in System C have been published. However, all these approaches end up at the register transfer level. Authors point out that for a more low-level and fine-grained resolution SystemC is less effective than gate-level simulators. However, this approach does not consider the shortcomings of gate-level models on one hand the potential of SystemC-models on the other hand.

Typically, design flows that use SystemC for simulation and validation at the micro-architectural and the RT-level subsequently switch to standard RTLs, not using SystemC to it's full potential. In particular, a fine-grained hardware synthesis at the gate level directly from SystemC descriptions [12] is not considered, although is may have advantages over other languages that are hardly explored.

Gate level simulators (and ATPG tools) tend to know only "gates" as the lowest abstraction. Complex gates that combine multiple gate-level functions have been used heavily in the past, and pass-transistor logic as favoured by at least one school of low-power design. Gate level models are not quite equivalent, at least not under fault conditions. This is the main reason why, while a functional simulator may get along with a reasonably equivalent model at the gate level, a fault simulator may get problems if real fault effects need to be simulated.

Some gate level fault simulators allow for the declaration of "macros", whose internal structure may be derived from a functional description or can be stored in a macro-library. A fault list generator will typically try to split such macros into gates for fault injection. In a more refined environment, such an activity must be controlled by constraints.

SystemC allows for a direct description of a complex gate without a virtual resolution into equivalent gates. Several gates that work concurrently can also be modeled as concurrent units in SystemC. Then FIT, which does not know other units than gates, flip-flops and macros, needs to contact such units. The concept developed for this purpose is a "saboteur" [7]. Based on such a concept, the process of fault injection into a SystemC model is shown in Figure 5.

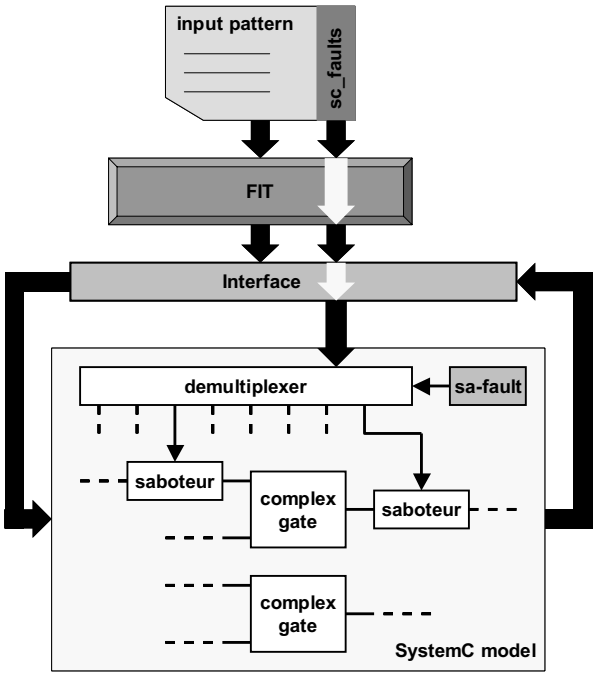

Figure 5: Fault injection into a SystemC-model based on "saboteurs"

A saboteur is a circuit modification, representing the fault case, which is inserted at every prospective fault location. This saboteur is virtually connected to a central virtual de-multiplexer. The de-multiplexer gets the respective saboteur selection for the fault case from an additional "fault bus", which is added to the normal inputs and outputs. This fault bus may gets it's signals from an extension of the normal input pattern file. In case of a single fault we obtain a 1 to 2^n selection of the saboteur, where n is the width of the fault bus. A direct connection of bus bit lines to the inputs of saboteurs would mean a simpler circuit but a much wider bus width and longer simulation runs. As a saboteur has to react to a fault activation, the respective line for fault activation must be part of the sensitivity list associated with the method "saboteur". An execution is then done only when a specific signal in the sensitivity list is triggered, resulting in the activation of just the saboteur to be addressed.

On the other hand, including the whole bus into the sensitivity list used for activating all saboteurs upon a call would result bin much longer execution times.

The implementation of saboteurs into the net list means a substantial increase in (virtual) circuit size, but the fault model is simple, and the rise in simulation efforts seems to be acceptable. The code for the implementation of a saboteur is depicted in Figure 6.

```
if (fault_active.read()==1) {
        out.write(sc_logic_1);
        }
else {
        out.write(in.read());
}
```

Figure 6: Code of „saboteurs“ for a stuck_at_1 fault

With these methods, we can insert a simple and straightforward fault injection also into the SystemC code, which is controlled by the FIT fault simulator core.

5. Results and Forthcoming Work

It has been shown that there are methods to implement a mixed-level fault simulation process based on SystemC not only at the RT-level, but in a more general sense.

Hierarchical fault simulation experiments as reported in [1] could also be done by coupling VHDL and SystemC in a hierarchical design.

We have first results on simulation times that will be presented at the conference.

It was found that using the data type “bool” results in relatively long simulation times, most likely due to internal conversions of data types.

Further work is under way in the identification of performance bottlenecks. We also intend to operate the simulation environment on a multi-processor environment that used the MPI [14] middleware platform.

6. References

[1] Misera, Vierhaus : FIT – a Parallel Hierarchical Fault Simulation Environment, Parelec 2004, Dresden.

[2] H. K. Lee and D. S. Ha, „HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits“, Proc. ACM-IEEE Design Automation Conference (DAC) 1992, pp. 336-340.

[3] SystemC: www.systemc.org, 1.03.2006.

[4] Modelsim Designer: www.model.com, 1.02.2006.

[5] F. Bruschi, F. Ferrandi and D. Sciuto: A Framework for Functional Verification of SystemC Models, International Journal of Parallel Programming, Vol. 33, No 6, December 2005.

[6] F. Bruschi, M. Chiamenti, F. Ferrandi, D. Sciuto: Error Simulation Based on the SystemC Design Description Language,Automation and Test in Europe Conference and Exhibition (DATE'02), 2002, pp. 1135.

[7] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil: Comparison and Application of Different VHDL-Based Fault Injection Techniques, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'01), October 2001, pp. 0233.

[8] Interfacing SystemC to HDL using PLI/VPI & FLI http://www.anslab.co.kr/psce/tutorial/counter32/counter32.htm, 1.10.2005.

[9] Aldec Inc.: Active-HDL und Riveria, www.aldec.com, 1.11.2005.

[10] B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante: New Techniques for Accelerating Fault Injection in VHDL Descriptions, July 2000, pp.61.

[11] Posix-Threads: http://sources.redhat.com/pthreads-win32/, 09.03.2006.

[12] Synopsis, Inc.: Describing Synthesizable RTL in SystemC, Jan. 2002.

[13] Galke, Misera, Vierhaus : Parallele Hierarchische Fehlersimulation zur Validierung des Fehlerverhaltens für SoCs, GI/ITG/GMM Workshop (Modellierung und Verifikation) 2005, München.

[14] LAM/MPI parallel computing: http://www.lam-mpi.org, 10.03.2006.