Facilitating Testability of TLM FIFO: SystemC Implementations

Homa Alemzadeh[1], Marco Cimei[2], Paolo Prinetto[2], Zainalabedin Navabi[1]

[1]*CAD Research Group*
ECE Department, University of Tehran
Tehran 14399, Iran
{homa, navabi}@cad.ut.ac.ir

[2]*Dipartimento di Automatica e Informatica*
Politecnico di Torino, I-10129 Torino, Italy
s133891@studenti.polito.it
paolo.prinetto@polito.it

Abstract

TLM is a high-level approach to modeling digital systems with an emphasis on separating computations from communications within a system. With the evolution of design methodologies to transaction level the need for definition of DFT (Design for Test) techniques at this very high level of abstraction arises. This paper focuses on the implementation of three different high-level testing strategies for TLM FIFO as the basic TLM communication channel. These strategies are implemented by adding Built-in Functional Self Test (BIFST) utilities to the channels and computation units. We present SystemC implementations of the utilities that we have developed in the form of new SystemC classes and methods.

1. Introduction

With the increasing complexity of digital systems, and shrinking time to market, Electronic System Level (ESL) design has emerged as the main design methodology for implementing large digital systems. The evolution of ESL design methodologies has introduced *Transaction Level Modeling (TLM)*. TLM is a transaction-based modeling approach, originally based on high-level programming languages such as C++ and SystemC, which emphasizes on separating communication from computation within a system. In the TLM notion, communication mechanisms are modeled as abstract channels accessed resorting to interface functions [1].

Contrary to the migration of design methodologies from gate and register transfer levels to higher abstraction levels such as TLM, testing and testability techniques are still mostly performed at lower abstraction levels. It is thus gaining importance for system level designers the introduction of DFT techniques to insert testability features directly at TLM level in a completely automatic way, without concerning themselves with the intricacies of lower level implementations.

In [2] we introduced a testing methodology applicable at the TLM level during the system level design phase. We proposed our test techniques that can be applied to the design even before hardware/software partitioning, and we focused on TLM communication channels. In the present paper we present the SystemC implementations of different TLM Testing Strategies introduced in [2].

The paper is organized as follows: Section 2 has an overview on the TLM Testing Methodology presented in [2]. Section 3 provides the SystemC Implementation for three proposed TLM testing strategies, and finally Section 4 concludes the paper.

2. TLM Testing Methodology

This section presents an overview of the proposed high-level TLM testing methodology proposed in [2].

In [2] we took the preliminary steps toward definition of DFT at TLM abstraction level by developing a testing methodology during the system level design phase, before hardware/software partitioning with a focus on TLM communication channels. Since the only possible and reasonable testing strategy at a very high abstraction level like TLM is functional testing, we provided a design methodology capable of defining functional tests at TLM abstraction level, to be later on automatically translated into Built-in Functional Self Test (BIFST) facilities in the final product. These BIFST facilities are added into each computation unit and the communication channel in the TLM design and can be later synthesized either into hardware or software according to the designer's choices and needs. Figure 1 shows the proposed test architecture of [2] in which each of the computation units (Writer and Reader) and the communication channel are modified to include the required BIFST facilities.

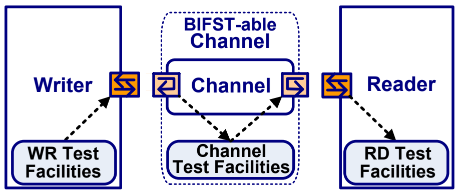

Figure 1 TLM Test Architecture [2]

Three different testing strategies are considered in this architecture. These are: Transaction Testing, Self Testing, and Integration Testing. Each of these testing strategies can be implemented by definition of the involved blocks in test procedure and implementing the required facilities in them. The BIFST units added inside computation units and the communication channels include Test Data Generators (TDGs), Test Response Evaluators (TREs), Test Controllers, and Interfaces [2].

The proposed architecture is a general architecture for implementing different testing strategies and a designer can have his/her own test approach and implementations for the added facilities. In the next section we present the SystemC implementations of BIFST facilities and the TLM definition of BIFST-able *tlm_fifo*, as the most basic TLM primitive channel in TLM Library, based on the FSM model test approach presented in [2] and [3].

3. SystemC Implementations

In the test architecture of Figure 1 the implementation of TDG and TRE units correspond to the definition of both test cases and test oracles for all the TLM methods of the communication channel under test. Test Oracles are the set of operations needed to check the correct execution of test cases. These operations include the comparisons to check the state of the channel and values returned to outside, as well as methods to be performed during the test procedure in order to put the channel in the required working states and prepare it for test execution [3].

Our overall idea is to define, for each method of *tlm_fifo* primitive channel, based on its FSM model, a suitable functional test sequence. In particular, we generate the test cases for each *tlm_fifo* method by trying to stress it in different operational states, satisfying the transition coverage criterion [3]. Table 1 shows the test sequences for three methods of FIFO (*put(), nb_put(),* and *nb_can_put*) implementing WRITE functionalities. Test cases for all other methods of *tlm_fifo* have been generated in a similar way.

The SystemC implementation of these test sequences will be different according to the chosen testing strategy. In the following subsections we see SystemC implementations of the proposed BIFST facilities for three different strategies introduced in Section 2.

Table 1 - Test Sequences for *tlm_fifo* Methods (Write Functionalities)

#	*put()*	#	*nb_put()*	#	*nb_can_put()*
1	✓ *put()*	1	*nb_put()*	1	*nb_can_put()*
n-1	✓ *put()*	*n*-1	*nb_put()*	1	*put()*
1	✓ *put()*	1	*nb_put()*	1	*nb_can_put()*
1	*get()*	1	*get()*	*n*-1	*put()*
1	*get()*	*n*-2	*get()*	1	*nb_can_put()*
n-2	*get()*	1	*get()*	*n+1*	*get()*
1	*get()*	1	*get()*	1	*nb_can_put()*
1	*get()*	1	*nb_put()*	1	*put()*
1	✓ *put()*	1	*peek()*	1	*peek()*
1	*peek()*	1	*nb_put()*	1	*nb_can_put()*
1	✓ *put()*	1	*get()*	1	*put()*
1	*get()*			1	*get()*

3.1. Transaction Testing Implementation

Transaction testing strategy includes testing the transactions between channel and computation units independently. This can be done for testing Write Transactions or Read Transactions by checking the correct functionality of the interconnection between the Writer/Reader and Channel without using the other side Reader/Writer. Figure 2.a and b show the implementation of Write and Read Transaction tests in the architecture of Figure 1.

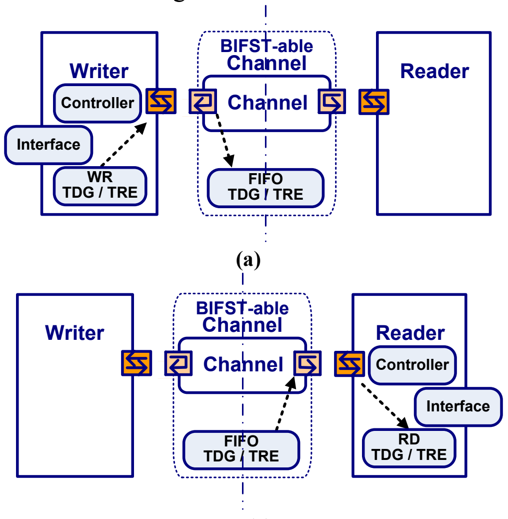

Figure 2 Transaction Testing: (a) Write Transaction (b) Read Transaction

This strategy has several advantages. First of all the "writing" and "reading" functionalities of the channel can be easily tested autonomously (e.g., testing the writing functionalities does not require the reader to be involved). The reader and writer are not concurrently involved in the test, and their related activities do not need any ad-hoc timing. Each of them has the responsibility of testing the methods of the channel actually used. The main drawback of this approach is that the communication channel and the computation units have to be modified to include the test facilities which lead to an overhead.

In this test strategy, each single block receives external requests concerning the beginning of the test phase. Then the corresponding computation unit (Writer/Reader) waits until the communication channel becomes ready for performing test. After this, the Transaction Test mode is activated for the communication channel and the test sequence starts by applying generated test cases into the channel. Since only the Writer/Reader and that part of the channel that is responsible for Writing/Reading are participating in performing a Transaction test, the communication channel should emulate the other side's functionality by calling the methods internally. For example the *put()* functions check-marked in the first column of Table 1 are test transactions issued by the Writer. But the *get()* and *peek()*s should be emulated inside the FIFO. As shown in Figure 2, for testing Write Transactions the TRE unit of channel performs test oracles by responding to the test transactions issued by the Writer; and for testing Read functionalities the TDG unit of Reader generates test cases, $\text{TRE}_{\text{Reader}}$ plays the role of Test response evaluator, and TRE_{FIFO} performs operations for moving FIFO between the required working states.

The generated test cases and Oracles are added into SystemC description of Writer, Reader and the Channel. We implement the new SystemC class of *testable_tlm_fifo* which inherits from *tlm_fifo* and overrides all the FIFO methods to include BIFST facilities inside [2]. The functions which are going to be emulated internally by FIFO are defined as new TLM methods in the new BIFST-able version of *tlm_fifo*. These methods are started with "*t_"* indicating the "Test" version of each method and include: *t_nb_peek(), t_compare(),t_get(), t_put(),* and *t_peek().* The first two functions are used in the implementation of comparison oracles for checking the values and states. The *t_nb_peek()* method is a non-blocking test method which always *nb_peek()* the most recently written element to the FIFO. Its difference with *nb_peek()* is that it is called internally from the FIFO and reads the most recently element instead of the first one. The *t_compare()* method is for comparing the test response (value and state of FIFO) with the fault-free reference model of FIFO. The *t_get(), t_peek()*, and *t_put()* methods are implemented by calling the original *get()*, *peek()* and *put()* methods [2], respectively.

```
//put()
template <typename T>
inline
void
testable_tlm_fifo<T>::put(const T& val)
{
 //Test Mode
 if (m_N_Tmode == true){
    //Full: Unblock  Writer Thread
    if (!tlm_fifo<T>::nb_can_put())
        fifo_full_event.notify(
                          SC_ZERO_TIME);
 }
//Normal Operation
tlm_fifo<T>::put(val_);
wait(0,SC_NS);
//Test Mode
if (m_N_Tmode == true){
    //Before FIFO becomes Full
    if (m_test_num <= m_size)
    {
       in >> TestData;
       t_compare(t_nb_peek(),TestData);
       m_test_num++;
    }
    //Before FIFO becomes Empty
    else if (m_test_num == m_size+1)
    {
       //Peek and Compare the MRW
       in >> TestData;
       t_compare(t_nb_peek(),TestData);

       //Get until FIFO Empty
       while(tlm_fifo<T>::nb_can_get()
       {
           x = t_get();
       }
       //EMPTY: Unblock Reader Thread
       fifo_empty_event.notify(
                          SC_ZERO_TIME);
       wait(1,SC_NS);
    }
    //FIFO is Empty
    else if (m_test_num == m_size+2)
    {
       in >> TestData;
       t_compare(x, TestData);
       //Back to Unblock Reader Thread
       wait(1,SC_NS);
    }
    else if (m_test_num == m_size+3)
    {
       in >> TestData;
       t_compare(x, TestData);
       x = t_get();
    }
} }
```

Figure 3 SystemC Description of Overridden *put*() Method in *testable_tlm_fifo*

The class definition of *testable_tlm_fifo* including the header of the new overridden methods, threads, new test methods, and data types is presented in [2]. In Figure 3 the SystemC descriptions for overriding the *put()* method as an example of implementing the test sequence of Table 1 is shown. The Writer issues the test transactions to the FIFO by calling this overridden *put()* method and FIFO responds to them by running the function of Figure 3. The deterministic test data are fed from an input file which is shared between TDG and TRE units (e.g. *in >> TestData* in Line 14). The *m_N_Tmode* (Lines 1 and 10) indicates the Test mode and the *m_test_num* variable is for counting the steps of the test sequence. Two threads, called *unblock_writer* and *unblock_reader*, are defined for *testable_tlm_fifo* to provide parallelism to FIFO [2]. They enable continuing the sequence of actions when the FIFO is blocked to perform a blocking TLM function and cannot continue the sequential code of its *put()* method. These threads are activated by notifying their corresponding events from *put()* method (Lines 4, 30, 39).

3.2. Channel Self-Testing Implementation

Channel Self-Testing Strategy tests a channel as an isolated component, without consideration of its connection to the Writer and Reader. In this strategy only the FIFO needs to be modified. The reader and writer are not involved in the test and do not have any information about the topology of the system. The drawback of this approach is the overhead of modifying the communication channel to include the proposed BIFST facilities.

The implementation of this strategy is done by the definition of TDG and TRE units and the test controller inside *tlm_fifo*. A new *SC_THREAD* called *t_self_test* is described inside the *testable_tlm_fifo* SystemC description and is initiated by activating the Self Test mode from outside the FIFO. This thread performs the test sequences for each method of *tlm_fifo*. The *tlm_fifo* methods generated in test sequences of Table 1 are replaced by their "*t_*" versions and are called internally with the same ordering inside the FIFO.

3.3. Integration Testing Implementation

The last strategy is for testing the integration between Writer, FIFO and Reader. In the Integration Testing strategy, FIFO always runs in its normal mode and is not involved in the test procedure. The reader and writer are concurrently responsible for performing tests by some external synchronization and timing considerations. Both the reader and writer should be aware of the topology of the communication channel to which they are connected and need a protocol for being synchronized with each other.

This strategy is implemented by definition of TDG_{Writer} and TRE_{Reader} units for testing Write functionalities and TDG_{Reader} and TRE_{Writer} units for Read methods. Also we need to define the synchronization protocol and the test controller.

The generated test cases and test oracles are added into the SystemC description of Writer and Reader and the test control is implemented by using *ok_to_put()*, *ok_to_get()*, *ok_to_peek()* methods of FIFO and *wait()* statements which enable the synchronization between Writer and Reader. Figure 4 shows part of a SystemC code implementing the TRE_{Reader} and Reader Controller for the *put()* method test procedure. Before performing any response, the Reader waits for FIFO to complete the execution of *put()* transaction issued by the Writer. This is done by using *ok_to_get()/ok_to_peek()* methods which notify an event whenever a write is done inside the FIFO. In other words, the SystemC codes of the body of the overridden methods in Transaction Testing implementation, here in Integration Testing are used as portions of codes between *wait(ok_to_get())/wait(ok_to_peek())*s. The implementation of the other side unit (TDG_{Writer}) is the same as Transaction Testing strategy and consists of a sequence of *put()* invocations.

```
//Reader TRE
. . .
while (nb_can_put()){
   wait(ok_to_peek());
   nb_peek(val_, m_num_readable-1);
   tr_compare(val_, TestData[i]);
   wait(SC_ZERO_TIME, NS);
}
. . .
```

Figure 4 SystemC Description for TRE in Write Transaction Test Strategy

4. Conclusions

In this paper we focused on SystemC implementations of *testable_tlm_fifo* for three different TLM Testing Strategies introduced in [2]. The proposed SystemC implementations can be used as DFT rules in the process of TLM design.

5. References

[1] T. Grötker, S. Liao, G. Martin, S. Swan, System Design with SystemC. Springer, 2002, Chapter 8, pp. 131.

[2] H. Alemzadeh, S. D. Carlo, F. Refan, P. Prinetto, Z. Navabi, "Plug & Test at System Level via Testable TLM Primitives," To appear in Proc. of *International Test Conference (ITC'08),* Pre-prints Available at: http://orion.polito.it/~dicarlo/plist/ITC08.pdf

[3] H. Alemzadeh, S. D. Carlo, A. Scionti, P. Prinetto, Z. Navabi, "Functional Testing Approaches for "BIFST-able" *tlm_fifo*," To appear in Proc. of *IEEE International High-Level Design Validation and Test Workshop 2008.*