Functional Testing Approaches for "BIFST-able" *tlm_fifo*

H. Alemzadeh, Z. Navabi

CAD Research Laboratory, Department of Electrical and Computer Engineering
School of Engineering, University of Tehran, Iran
Email: {homa, navabi}@cad.ut.ac.ir

S. Di Carlo, A. Scionti, P. Prinetto

Contro and Computer Engineering Department,
Politecnico di Torino, Italy.
Email: {stefano.dicarlo, alberto.scionti, paolo.prinetto}@polito.it

Abstract

Evolution of Electronic System Level design methodologies, allows a wider use of Transaction-Level Modeling (TLM). TLM is a high-level approach to modeling digital systems that emphasizes on separating communications among modules from the details of functional units. This paper explores different functional testing approaches for the implementation of Built-in Functional Self Test facilities in the TLM primitive channel tlm_fifo. In particular, it focuses on three different test approaches based on a finite state machine model of tlm_fifo, functional fault models, and march tests respectively.

Index Terms *— Design for Testability (DFT), System Test, Functional Testing, Transaction Level Modeling (TLM), System Level Design*

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 communications from computations within a system. In the TLM notion, communication mechanisms are modeled as abstract channels accessed resorting to interface functions. Transaction requests between modules take place by calling these functions that encapsulate low-level details of the information exchange. At the transaction level, the emphasis is more on data transfer functionalities rather than on their actual implementations.

Contrary to the migration of digital system design methodologies from gate and register transfer levels to higher abstraction levels such as TLM, testing and testability techniques are still mostly performed at the lower abstraction levels. It is thus gaining importance for system level designers the introduction of new tools to insert test and testability features directly at TLM level in a completely transparent and automatic way, without concerning themselves with the details and intricacies of lower level implementations. These tools will play, at system level, the same role that today EDA tools play at the gate and RT levels.

In [1] we presented a functional testing methodology applicable at the TLM abstraction level during the system level design phase, even before hardware/software partitioning. Functional testing is the only possible and reasonable testing strategy at this very high level of abstraction. The added testing capabilities can be later synthesized either into hardware or software according to the designer's choices and needs.

In the present paper we propose three different functional testing approaches for the implementation of TLM testing strategies introduced in [1].

This paper is organized as follows: Section 2 presents a background of the TLM testing methodology presented in [1]. Section 3 presents a FSM based testing approach, while Section 4 presents a pure functional fault based testing. Section 5 introduces a functional testing approach based on march tests and Section 6 concludes the paper.

2. TLM Testing Methodology

For the sake of clarity, this section briefly introduces the basis of the high-level TLM testing methodology proposed in [1]. The basic idea relies on introducing additional test functionalities to the blocks composing a TLM design to be translated later into *Built-in Functional Self Test* (BIFST) facilities available in the final product. In particular, the design methodology comprises the idea of enriching each computation unit of a design with predefined test facilities, and replacing each original communication channel with a corresponding new BIFST-able version. In addition, in order to evaluate the proposed testing

978-1-4244-2922-6/08/$25.00 © 2008 IEEE.

strategy, early quality evaluation metrics must be introduced during the design phase. These metrics should be easily measurable, available, and acceptable at a very high abstraction level.

The added BIFST capabilities can then be synthesized along with the whole TLM system either into hardware or software modules according to the designer's choices and needs. Also the evaluation metrics used during the design phase can be re-used on the final product with the same semantics and accuracy.

The preliminary test architecture for TLM designs presented in [1] is depicted in Figure 1. Each computation unit (Writer/Reader) as well as the communication channel is modified to include the required BIFST facilities. This architecture can fit any type of TLM communication channel; nevertheless, for the sake of simplicity and without any loss in generality, in this paper we will only focus on a specific channel: the basic TLM primitive *tlm_fifo*. *tlm_fifo* implements TLM unidirectional communications, and it is used in the implementation of all other TLM communication channels.

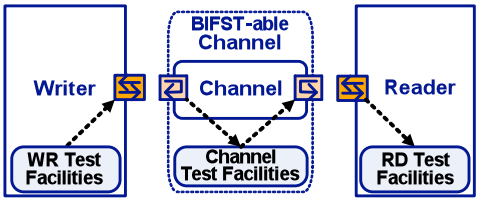

Figure 1: test architecture - added test facilities

The architecture proposed in Figure 1 can support three different testing strategies:

1. *Transaction Testing*: testing the transactions between the channel and Writer/Reader separately. It includes Write Transaction Testing and Read Transaction Testing;
2. *Channel Self-Testing*: testing the channel as an isolated component, without considering its connections with the Writer and the Reader;
3. *Integration Testing*: Testing the integration between Writer, Channel, and Reader [1].

Each of these testing strategies requires the implementation of a set of different functional blocks. Figure 2 shows a possible instantiation of the different blocks (BIFST units) required to implement the different test facilities. They include interfaces, Test Data Generators (TDGs), Test Response Evaluators (TREs), and controllers, each implemented according to the chosen functional test.

The proposed architecture is general enough to implement different test strategies. Moreover it can easily fit different types of user defined test strategies.

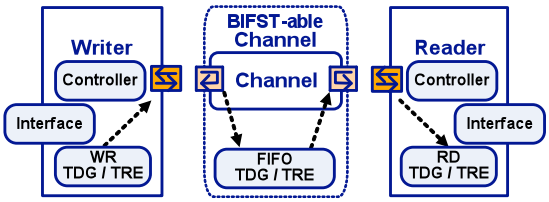

Figure 2 - Test facilities implementation

In the sequel of the paper we will propose three different functional testing approaches for the implementation of BIFST units. They include FSM model-based, functional fault model-based, and march-based test approaches.

3. FSM Model-Based Testing

Functional testing is the procedure of deriving test cases from functional specifications of the target implementation. One of the approaches to functional testing is the so-called *model-based testing* [2], which aims at deriving models of expected behaviors of the target system to produce test case specifications. Models can be expressed either in formal or semiformal ways.

Finite State Machines (FSM) are one of the common formal models used in the automatic generation of test cases. FSMs are often used to specify the sequences of interactions between a system and its environment, especially in control and reactive systems such as communication protocols [2].

In [1] an FSM model was defined to formally represent the *tlm_fifo* channel and to generate functional test cases for it. This state machine was extracted from the semi-formal specification of the *tlm_fifo* available in SystemC TLM Standard released by OSCI [3]. The test generation approach presented in [1] was rather intuitive, being mostly derived from test engineers' experiences. In this paper we present a more formal approach, based on software testing methodologies.

FSM models can be used both for generating test cases and for constructing *Test Oracles*. Oracles are units that inspect the test results and judge whether each observed behavior is correct or not [2]. In the architecture introduced in Figure 2, the implementation of TDG and TRE units correspond to the definition of both test cases and oracles for all methods of the *tlm_fifo* under test.

One of the most common ways of generating test cases from FSMs is checking state transitions. The *transition coverage* criterion, widely used within the software testing community, requires each transition of a FSM model to be traversed at least once [2]. We generated the test cases for each method of *tlm_fifo* by trying to stress this method in different operational

states in order to traverse all transitions it fires on the FSM *tlm_fifo* model. As an example, Table 1 shows a sequence of method calls which covers all transitions of the `put()` method in the FSM *tlm_fifo* model [1]. The first column shows the number of calls of each method in a *tlm_fifo* of size *n*. The rows indicated by check marks show the `put()` method in all possible states and the full coverage of transitions on the FSM *tlm_fifo* model. Test cases for other methods of *tlm_fifo* have been generated in a similar way.

Table 1 - Covering all transitions of put() method

#	TLM Method	Initial State	Final State
1	✓ `put()`	Empty	Semi-Full
n-1	✓ `put()`	Semi-Full	Full
1	✓ `put()`	Full	Blocked `put()`
1	`get()`	Blocked `put()`	Full
1	`get()`	Full	Semi-Full
n-2	`get()`	Semi-Full	Semi-Full
1	`get()`	Semi-Full	Empty
1	`get()`	Empty	Blocked `get()`
1	✓ `put()`	Blocked `get()`	Empty
1	`peek()`	Empty	Blocked `peek()`
1	✓ `put()`	Blocked `peek()`	Semi-Full
1	`get()`	Semi-Full	Empty

After generating the test cases and driving them into the *tlm_fifo*, we need to introduce test oracles. Test oracles are typically used in the software testing community [2], and they are here intended as the set of operations performed in response to test cases to check their correct execution. These operations include comparisons to check the correctness of the *tlm_fifo* state and the correctness of values returned by the execution of each called method, as well as methods called during the test procedure to put the *tlm_fifo* in the required working states, and to prepare it for test execution.

In other words, we need comparison-based oracles to verify the data written by each write operation (`put()`, `nb_put()`), each data read (`get()`, `nb_get()`, `peek()`, `nb_peek()`) from the *tlm_fifo*, and the return values of each non-blocking method. The `t_peek()` and `t_compare()` methods introduced in [1] are examples of oracles defined internally to the BIFST-able *tlm_fifo* for inspecting the last value written into its buffer.

In addition, we need some oracles implemented using TLM methods to move the *tlm_fifo* among its working states. As an example of these test oracles, see the `get()` transactions performed in the test sequence of Table 1 for testing the `put()` functionality. These `get()` calls are used for unblocking the $n+1^{th}$ `put()` which is called when the *tlm_fifo* is full and then bringing back the *tlm_fifo* into its Empty state. The last two `get()` and `peek()` calls in this table are required to check the functionality of `put()` when a blocking read is called on an Empty *tlm_fifo*.

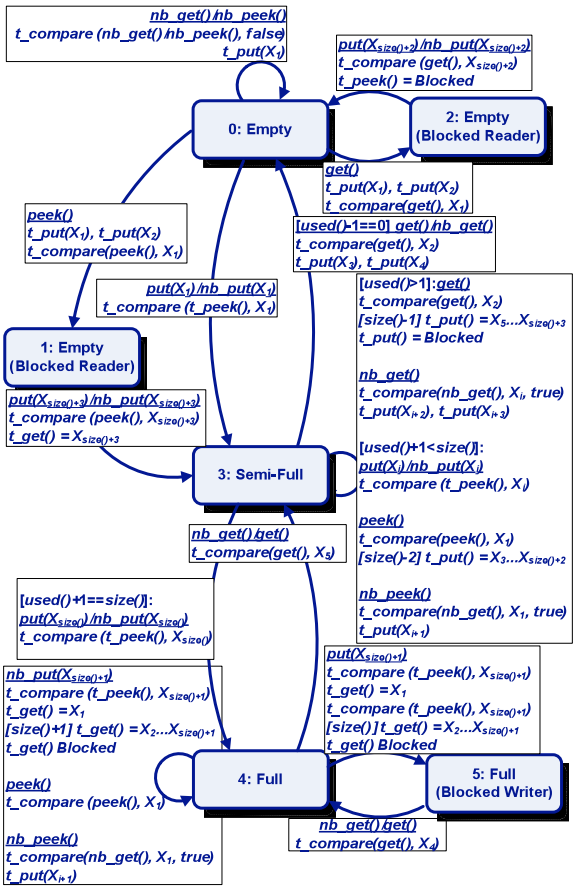

Figure 3: BIFST-able *tlm_fifo* FSM model - oracles added into state diagram

Based on the chosen test strategy, the involved test blocks, and the location of TDG and TRE units, the implementation of test cases and oracles would be different. For example, in a Integration Test Implementation of the `put()` method (see Section 2, Figure 2), the `put()` and `get()` calls of Table 1 are called by the Writer and the Reader respectively. Actually the sequence of `put()` methods plays the role of the TDG unit within the Writer. In a similar way, the TRE unit of the Reader is implemented by the sequence of `get()` call and by some comparison methods in response to each `put()`.

On the other hand, in a Transaction Test implementation, the Writer and *tlm_fifo* are responsible for generating the sequence of actions to perform the Write Transaction Test for `put()`. In this case new facilities must be added to the *tlm_fifo* to perform `get()` calls internally. These test oracles will be implemented as part of the TRE inside the *tlm_fifo* as new

TLM methods defined for BIFST-able version of the *tlm_fifo*.

Figure 3 shows the FSM model for the BIFST-able version of *tlm_fifo*. This diagram shows the functionality of BIFST-able *tlm_fifo* in a transaction testing implementation for testing write and read transactions. Test oracles discussed earlier are added on the arcs of the FSM as the responses to each issued transaction. As a concluding remark, while the FSM presented in [1] can be considered as modeling the *test specification* for the *tlm_fifo*, the modified FSM model in Figure 3 represents the *test implementation* by introducing the concept of oracles.

4. Functional Fault Model-Based Testing

This section exploits an alternative approach to functional testing widely used within the verification and validation community. According to this approach, test cases are generated by defining a set of high level fault models based on the behavioral description of the system. These fault models are defined based on either the textual behavioral descriptions or some other models of the system, like Control Data Flow Graphs (CDFGs), state machines, etc. [5]. In addition, several works present application specific functional fault models like those used in microprocessor [6, 7, 8], memory [9, 10] and NoC testing [11, 12].

The proposed functional fault model based testing here considers the definition of a set of high level functional faults based on the description of *tlm_fifo* as provided in OSCI SystemC TLM Library [3]. The set of functional faults is split into three subsets based on the functionalities of *tlm_fifo* as follows:

1. Faulty `put()/nb_put()/nb_can_put()`;
2. Faulty `get()/nb_get()/nb_can_get()` ;
3. Faulty `peek()/nb_peek()/nb_can_peek()`.

Table 2 lists the complete set of high level functional faults defined for these subsets and their possible low level causes. It is worth mentioning that these low level faults are manually derived from the *tlm_fifo* structure based on its description from SystemC TLM Library. Although these faults look similar to faults occurring in a hardware implementation, a software implementation may still encounter the same problems due to faults in its memory structure.

As one can see from the table, since the main functionalities of blocking and non-blocking versions of each method are the same and their differences lie in their blocking or non-blocking natures, the same low level faults will cause faulty operations in both cases. Also `get()` and `peek()` methods of *tlm_fifo* both perform a read operation on the *circular_buffer* embedded inside the fifo itself. The main difference is that the `get()` method updates the value of the read pointer and the `used()` variable of *circular_buffer*, while `peek()` just returns the data without any update. So the faulty operation of these two methods would also have the same causes. This also means that the good functionality of the `get()` method assures us about the healthy read operation of *circular_buffer* and the only remaining possible causes of faulty operation of the `peek()` method would be faults in the controller or input and output busses.

Based on the set of functional faults of Table 2, we defined the functional test sequence of Table 3. This sequence is generated by trying to force the *tlm_fifo* in different states, and test as many methods as possible in each state to detect the related faults. The first column of this table shows the number of repetitions in performing each method. The second and fourth columns show the test data and the expected response from the *tlm_fifo* respectively. The last column shows the set of faults of Table 2 detected by performing each test transaction.

The test sequence starts by issuing `put()` transactions to the *tlm_fifo* until it becomes full, and then performing `get()` operations to read the written data. The pseudorandom sequence $X_{1..N+1}$ is used as *test data*. This sequence will test the functionality of `put()` and `get()` in Empty, Semi-Full, and Full states of *tlm_fifo*. The sequence detects many faults of Table 2 related to the blocking behavior of `put()`, `get()`, and `peek()` (because of its similarity to `get()`), Data-in/Data-out Busses, and Controller and Empty/Full Flags. In order to optimize the test sequence and to avoid additional `put()` and `get()` calls required to make *tlm_fifo* full and empty, we also perform `nb_can_put()`, `nb_can_get()`, and `nb_peek()` calls in the middle of successive `put()` and `get()` calls. This gives us the added advantage of testing the functionality of these methods.

After this first sequence, the *tlm_fifo* has been checked for all possible faults in read and write operations. Additional tests should be added to inspect the functionality of the *tlm_fifo* controlling unit. This starts by doing a blocking `get()` and `peek()` calls in the Empty state followed by *nb_can_put()* and `put()` calls. This tests the blocking behavior of `get()` and `peek()` as well as the functionality of `nb_can_put()` and `put()` when *tlm_fifo* `get()` and `peek()` are blocked.

After this set of tests, the portions of the *tlm_fifo* control that is related to `nb_can_get()`, `nb_peek()`, `get()`, `nb_can_put()`, and `put()` methods are checked. The sequence is continued in a similar manner to check the control functionalities of

peek(), nb_can_peek(), and nb_put().The total number of test transactions performed in this sequence is *3n+35*. This means that all faults listed in Table 2 are detected with an order of O(*3n*).

The proposed test approach does not consider the possible faults of the memory inside *circular_buffer* but just tries to detect some of the memory faults by writing different orders of data in the process of writing to the *tlm_fifo*.

Table 2 - List of Functional Faults for *tlm_fifo*

Fault Subset	Functional Faults	Possible Low-level Causes
(Set A) **Faulty** **put()** **nb_put()** **nb_can_put()**	1. *tlm_fifo* is Full but: a. put() is done without blocking. b. nb_put() is done and returns true. c. nb_can_put() returns true.	Faulty Full Flag Faulty *tlm_fifo* Controller Faulty Return Data Faulty Increment of used() Faulty used() Register Faulty Comparator
	2. *tlm_fifo* Not Full but: a. put() is blocked. b. nb_put() is not done and returns false. c. nb_can_put() returns false.	
	3. put() is blocked for writing something but never returns.	Faulty Controller Faulty get()/nb_get()
	4. put()/nb_put() always write to the same place of *tlm_fifo*.	Faulty Write Pointer (m_wi) Faulty Inc. of Write Pointer
	5. put()/nb_put() always write the same data to the *tlm_fifo*.	Faulty Data in Bus
	6. put()/nb_put()/nb_can_put() called but other function is done.	Faulty Controller Other Faulty Function
(Set B) **Faulty** **get()** **nb_get()** **nb_can_get()**	1. *tlm_fifo* is Empty but: a. get() is done without blocking. b. nb_get() is done and returns true. c. nb_can_get() returns true.	Faulty Empty Flag Faulty *tlm_fifo* Controller Faulty Return Data Faulty Decrement of used() Faulty used() Register Faulty Comparator
	2. *tlm_fifo* Not Empty but: a. get() is blocked. b. nb_get() is not done & returns false. c. nb_can_get() returns false.	
	3. get() is blocked for reading something but never returns.	Faulty Controller Faulty put()/nb_put()
	4. get()/nb_get() always returns the same data from *tlm_fifo*.	Faulty Read Pointer (*m_ri*) Faulty Inc. of Read Pointer Faulty Data out Bus
	5. get()/nb_get() called but peek()/nb_peek() is done.	Faulty peek()/nb_peek() Faulty Controller Faulty Read Pointer Faulty Inc. of Read Pointer
	6. get()/nb_get()/nb_can_get() called but other function is done.	Faulty Controller Other Faulty Function
(Set C) **Faulty** **peek()** **nb_peek()** **nb_can_peek()**	1. *tlm_fifo* is Empty but: a. peek() is done without blocking. b. nb_peek() is done and returns true. c. nb_can_peek() returns true.	Faulty Empty Flag Faulty *tlm_fifo* Controller Faulty Return Data Faulty Decrement of used() Faulty used() Register Faulty Comparator
	2. *tlm_fifo* Not Empty but: a.peek() is blocked. b. nb_peek() isn't done & returns false. c. nb_can_peek() returns false.	
	3. peek() is blocked for reading something but never returns.	Faulty Controller Faulty put()/nb_put()
	4. peek()/nb_peek() always returns the same data from *tlm_fifo*.	Faulty Read Pointer (m_ri) Faulty Inc. of Read Pointer Faulty Data out Bus
	5. peek()/nb_peek()/nb_can_peek() called but other function is done.	Faulty Controller Other Faulty Function

Table 3 – Fault-Based Test Sequence for *tlm_fifo*

#	Test Data	tlm_fifo Status	Expected Response	Faults Detected
1	nb_can_put()	Empty	*true*	
1	put(X_1)	Empty	-----	Set A: 2.c
1	nb_can_put()	Semi-Full	*true*	
n-1	put($X_{2..N}$)	Semi-full	-----	Set A:2.a
1	nb_can_put()	Full	*false*	Set A: 1.c
1	put(X_{N+1})	Full	*Blocked*	Set A: 1.a
1	nb_can_get()	Full	*true*	
1	nb_peek()	Full	X_1, *true*	
1	get()	Full	X_1 ,*blocked* put(X_{N+1})	Set A:1.a,3
1	nb_can_get()	Full	*true*	
1	nb_peek()	Full	X_2, *true*	
1	get()	Full	X_2	Set A:3,4,5
1	nb_can_get()	Semi-full	*true*	Set B:2.a,2.c
1	nb_peek()	Semi-full	X_3, *true*	Set B:4,5
n-1	get()	Semi-full	$X_{3..N+1}$	Set C:2.b
1	nb_can_get()	Empty	*false*	Set B: 1.c
1	nb_peek()	Empty	*false*	Set C: 1.b
1	get()	Empty	*Blocked*	Set B: 1.a
1	nb_can_put()	Empty	*true*	Set A: 2.c
1	put(X_1)	Empty	*Blocked* get() = X_1	Set B:1.a,3,6
1	nb_can_peek()	Empty	*false*	Set C: 1.c
1	peek()	Empty	*Blocked*	Set C: 1.a
1	nb_can_put()	Empty	*true*	Set A: 2.c
1	put(X_2)	Empty	*Blocked* peek()=X_2	Set C: 1.a,3 Set A: 6
1	nb_can_peek()	Semi-full	*true*	Set C: 2.c
1	peek()	Semi-full	X_2	Set C: 2.a
1	nb_get()	Semi-full	X_2, *true*	Set B: 2.b
1	nb_get()	Empty	*false*	Set B: 1.b
1	get()	Empty	*Blocked*	Set B: 1.a
1	nb_put(X_{N+1})	Empty	*true blocked* get() = X_{N+1}	Set A: 2.b Set B: 1.a,3
1	peek()	Empty	*Blocked*	Set C: 1.a
1	nb_put(X_{N+1})	Empty	*true, blocked* peek():X_{N+1}	Set A: 2.b Set C: 1.a,3
n-1	nb_put($X_{N...2}$)	Semi-full	*true*	Set A: 2.b
1	nb_put(X_1)	Full	*false*	Set A: 1.b
1	put(X_1)	Full	*Blocked*	Set A: 1.a
1	nb_can_peek()	Full	*true*	Set C: 2.c
1	peek()	Full	X_{N+1}	Set C: 2.a
1	nb_get()	Full	X_{N+1}, *true* *Blocked* put(X_1)	Set B: 2.b Set A: 1.a, 3
1	nb_can_peek()	Full	*true*	Set C: 2.c
1	peek()	Full	X_N	Set C: 2.a, 5
1	nb_get()	Full	X_N, *true*	Set B: 2.b, 6

5. March-Based Testing

March-test based test tries to solve some of the problems of functional fault model based testing and in particular it tries to specifically address the problem of testing the memory elements composing the *tlm_fifo*.

March tests are a very established and efficient category of memory test algorithms with linear complexity [13]. A *march test* is a finite sequence of *march elements* delimited by a pair of braces. Each *march element* is composed of a sequence of memory operations applied to each element of a memory array delimited by parentheses. March elements are characterized by an *addressing order*, determining the order the memory elements are traversed during the test. March tests define two types of addressing orders: (i) *direct order* denoted by ⇑ (i.e., the scanning sequence goes from cell 0 to cell n-1), and (ii) *reverse order* denoted by ⇓ (i.e., the scanning sequence goes from cell n-1 to cell 0). In a single march element the possible memory operations are:

- *Write Operation (W_P):* a pattern P is written in the current memory cell;
- *Read & Verify (R_P)*: the content of the memory cell is read and verified whether it is equal to P.

To efficiently perform a march test, for each test pattern P, a complemented pattern P^* should be defined. For example, the march element *M2* showed Figure 4 uses a direct addressing order to apply the sequence of three operations R_P^*, W_p, W_p^* to each element of the cell array.

Besides their low complexity (linear in the number of memory cells), one of the main advantages of march tests is that they are built over a set of functional fault models that allow to design test algorithms independent of the current implementation of the memory under test [14, 15].

This section tries to extend this property at the TLM level. From a functional point of view the *tlm_fifo* is actually a memory array of abstract elements (*objects*), and it thus fulfills all requirements for the application of march tests. The question in this context is: are memory fault models designed for march based test meaningful when working with such high-level descriptions?

The answer to this question is for sure positive whenever the communication channel is designed to be mapped into a hardware component. In this case the testing scenario is exactly the one march tests are designed for. Nevertheless, even when considering a software implementation, the application of march tests could still provide very interesting functionalities.

Lets us consider a typical memory functional fault model addressed by march tests: the *stuck-at* fault, i.e., a memory cell is fixed at a certain value P. Considering a software implementation of the *tlm_fifo,* any software fault leading to the impossibility of changing the value of one of the fifo elements is equivalent to a

stuck-at fault and can be efficiently identified by a march test based test. The same is for other types of functional faults such as address faults, and read faults. We can thus conclude that march-based BIFTS is valuable for both hardware and software implementation of the *tlm_fifo*. Considering the possible functional faults that may appear in the *tlm_fifo*, a march test solution offers the same coverage as in the case of hardware implementation.

While Section 5.1 will provide details on how march tests can be implemented using TLM primitives, a few considerations concerning the concept of test pattern must first be introduced.

In a typical march test, a test pattern represents a value (i.e., a single bit equal to 0 or 1) to be written into a memory cell. In the *tlm_fifo* , each element of the array is instead an abstract object, such as, for instance, a JPG image. We exploit the concept of *object serialization* [16]. In computer science, serialization is the process of saving an object onto a storage medium (such as a file, or a memory buffer) or to transmit it across a network connection link in binary form. Applying this concept we can define a test pattern P, as the serialization of an object O to be stored in the *tlm_fifo*. With this definition the complemented pattern P^* can be defined as the serialization of an object O^* with all bits complemented with respect to the serialization of O.

The next subsection will detail how march tests can be efficiently implemented using *tlm_fifo* primitives.

5.1. March Test for *tlm_fifo*

The problem of applying march tests to fifo memories has already been addressed in the literature [10, 17]. It stems from the impossibility of applying the reverse addressing order due to the first-in-first-out access policy of the memory and from the limited possibility of performing multiple operations (e.g., multiple write operations) on a single cell.

Concerning the first limitation, in [18] the authors show how to build SAO (*Single Addressing Order*) march tests easy to be applied in all situations where a reverse addressing order is difficult to implement. In our march-based BIFST we will consider the use of SAO test algorithms.

Concerning the limitation on the sequence of operations, the TLM standard primitives represent a perfect support to implement any type of marching sequence. In order to detail how this implementation is possible we need to introduce a few assumptions on the internal behavior of the *tlm_fifo*. We consider a *tlm_fifo* implemented as a circular buffer of N elements as proposed in the OSCI SystemC TLM Library [3]. The next location to be written and the next location to be read are always identified by two pointers named `m_wi` and `m_ri`, respectively, and implementing the circular buffer. We also consider the following behavior of the main *tlm_fifo* methods that will be used to build the march test:

- `nb_get()`: returns the element pointed by `m_ri` and then increments the read pointer as *(`m_ri` +1) mod N*;
- `nb_put(P)`: inserts the object P in the element pointed by `m_wi` and increment the write pointer as *(`m_wi` +1) mod N*;
- `nb_peek()`: is equivalent to `nb_get()` but `m_ri` is not incremented after the read operation;
- `nb_poke(P)`: is equivalent to `nb_put(P)` but `m_wi` is not incremented after the write operation.

We consider non blocking primitives since the blocking poke primitive is not available in the *tlm_fifo*. We also resort to the additional `t_compare()` primitive introduced in [1] to check whether two values are equal. Considering the previous set of primitives, a generic march element can be applied to the *tlm_fifo* as follows:

- Translate each write operation except the last of the march element into a `nb_poke()` operation with the same pattern;
- Translate the last write operation of the march element into a `nb_put()` operation with the same pattern;
- Translate each read operation except the last of the march element into a `nb_peek()` followed by a test on the returned data;
- Translate the last write operation of the march element into a `nb_get()` followed by a test on the returned data;
- Repeat the sequence of operation N times.

Appling this rule, it is possible to apply any SAO march test to the *tlm_fifo* guaranteeing to maintain, by construction, exactly the same fault coverage and complexity of the original test. Table 4 shows the application of the first two march elements of SOA March B- (Figure 4) to a 4 elements *tlm_fifo*.

$$\{\Uparrow(W_P);\ \underbrace{\Uparrow(R_P,W_P^*,R_P^*,W_P,R_P,W_P^*)}_{M0};\ \underbrace{\Uparrow(R_P^*,W_P,W_P^*)}_{M1};\ \underbrace{\Uparrow(R_P^*,W_P,W_P^*,W_P)}_{M2};\ \underbrace{\Uparrow(R_P)}_{M3}\}$$

Figure 4: SOA-March B-

Table 4 - Application of *SOA-March B-*

	Content				m_ri	m_wi
Initial pointers position					0	0
nb_put(P)	P				1	0
nb_put(P)	P	P			2	0
nb_put(P)	P	P	P		3	0
nb_put(P)	P	P	P	P	0	0
compare(nb_peek(),P)	P	P	P	P	0	0
nb_poke(P*)	P*	P	P	P	0	0
compare(nb_peek(),P*)	P*	P	P	P	0	0
nb_poke(P)	P	P	P	P	0	0
compare(nb_get(),P)	P	P	P	P	0	1
nb_put(P*)	P*	P	P	P	1	1
compare(nb_peek(),P)	P*	P	P	P	1	1
nb_poke(P*)	P*	P*	P	P	1	1
compare(nb_peek(),P*)	P*	P*	P	P	1	1
nb_poke(P)	P*	P	P	P	1	1
compare(nb_get(),P)	P*	P	P	P	1	2
nb_put(P*)	P*	P*	P	P	2	2
compare(nb_peek(),P)	P*	P*	P	P	2	2
nb_poke(P*)	P*	P*	P*	P	2	2
compare(nb_peek(),P*)	P*	P*	P*	P	2	2
nb_poke(P)	P*	P*	P	P	2	2
compare(nb_get(),P)	P*	P*	P	P	2	3
nb_put(P*)	P*	P*	P*	P	3	3
compare(nb_peek(),P)	P*	P*	P*	P	3	3
nb_poke(P*)	P*	P*	P*	P*	3	3
compare(nb_peek(),P*)	P*	P*	P*	P*	3	3
nb_poke(P)	P*	P*	P*	P	3	3
compare(nb_get(),P)	P*	P*	P*	P	3	0
nb_put(P*)	P*	P*	P*	P*	0	0
…	…	…	…	…	…	…

6. Conclusions

In [1] we presented a testing methodology applicable at the TLM abstraction level, during the system level design phase, before hardware/software partitioning, with a particular emphasis on TLM communication channels. This paper explored three different functional testing approaches for testing *tlm_fifo* as the basic primitive in the library of TLM communication channels. These testing approaches were based on three common methods in functional testing: FSM model based, functional fault based, and march-based testing. All the proposed test approaches have been used in the implementation and definition of a BIFSTable *tlm_fifo*.

References

[1] H. Alemzadeh, S. D. Carlo, F. Refan, P. Prinetto, and 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.

[2] M. Pezzè and M. Young, Software Testing and Analysis: Process, Principles, and Techniques, WILEY, 2007.

[3] OSCI SystemC TLM 2.0 Standard, [Online Document], http://www.systemc.org/downloads/standards/tlm20/ (current Sep. 2008).

[4] A. Rose, S. Swan, J. Pierce, J.-M. Fernandez, Transaction Level Modelling in SystemC, OSCI white-paper, 2004.

[5] I.G. Harris, "Hardware-software co-validation: fault models and test generation," In Proceedings of *Sixth IEEE International High-Level Design Validation and Test Workshop,* pp 151 – 156, 2001.

[6] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation", In Proc. of *International Test Conference*, pp. 990–999, October 1998.

[7] D. Brahme and J. A. Abraham, "Functional testing of microprocessors", *IEEE Transactions on Computers*, pp. 475–485, June 1984.

[8] A. J. van de Goor and Th. J. W. Verhallen, "Functional testing of current microprocessors", In Proc. of *International Test Conference*, pp. 684–695, September 1992.

[9] A.J. Van de Goor, Y. Zorian, *"Fault models and tests specific for FIFO functionality"*, In Records of the *1993 IEEE International Workshop on Memory Testing*, 1993., 9-10 Aug. 1993 pp.72 – 76.

[10] Barbagallo, S., et al, "A Parametric Design of a Built-in Self-Test FIFO Embedded Memory," In Proceedings of *the 1996 Workshop on Defect and Fault-Tolerance in VLSI Systems,* pp 221, November 1996.

[11] Tomas Bengtsson, Shashi Kumar and Zebo Peng, "Application Area Specific System Level Fault Models: A Case Study with a Simple NoC Switch", International Design and Test Workshop (IDT), 2006.

[12] M. Sedghi, A. Alaghi, E. Koopahi, and Z. Navabi, "An HDL Based Platform for High Level NoC Switch Testing", In Proc. of *Asian Test Symposium (ATS),* Beijing, China, October 2007, pp. 453-458.

[13] A. J. van de Goor, "Testing Semiconductor Memories: theory and practice", *Wiley, Chichester (UK)*, 1991.

[14] Z. Al-Ars, Ad J. van de Goor, "Static and Dynamic Behavior of Memory Cell Array Opens and Shorts in Embedded DRAMs", In Proc. of *IEEE Design Automation and Test in Europe (DATE 2001)*, 2001, pp. 496-503.

[15] Z. Al-Ars and A.J. van de Goor, "Approximating Infinite Dynamic Behavior for DRAM Cell Defects", In Proc. of *20th IEEE VLSI Test Symposium*, 2002, pp.401-406.

[16] B. Carpenter, G. Fox, S. Ko, S. Syrac, "Object serialization for marshalling data in a Java interface to MPI", In Proceedings of the *ACM 1999 conference on Java Grande*, San Francisco, California, United States, pp.: 66 – 71.

[17] A.J. Van de Goor, Y. Zorian: "Functional tests for arbitration SRAM-type FIFOs", In Proc. of *First Test Symposium* (ATS '92), 26-27 Nov. 1992, pp. 96 – 101.

[18] A.J. Van de Goor, Y. Zorian "Effective march algorithms for testing single-order addressed memories", In Proc. of *4th European Conference Design Automation*, 22-25 Feb. 1993 Page(s):499 – 505.