[Proceeding] Functional Testing Approaches for "BIFST-able" tlm_fifo

Original Citation:

Availability:
This version is available at: http://porto.polito.it/1856181/ since: November 2008

Publisher:
IEEE

Published version:
DOI: 10.1109/HLDVT.2008.4695882

Terms of use:
This article is made available under terms and conditions applicable to Open Access Policy Article ("Public - All rights reserved") , as described at http://porto.polito.it/terms_and_conditions.html

Porto, the institutional repository of the Politecnico di Torino, is provided by the University Library and the IT-Services. The aim is to enable open access to all the world. Please share with us how this access benefits you. Your story matters.

(Article begins on next page)
Functional Testing Approaches for "BIFST-able" tlm_fifo

Authors: Alemzadeh H., Navabi Z., Di Carlo S., Scionti A., Prinetto P.,

Published in the Proceedings of the IEEE International High Level Design, Validation and Test Workshop (HLDVT), 19-21 Nov. 2008, Hyatt Regency Lake Tahoe Resort Incline Village (NV), USA.

N.B. This is a copy of the ACCEPTED version of the manuscript. The final PUBLISHED manuscript is available on IEEE Xplore®:

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4695882
DOI: 10.1109/HLDVT.2008.4695882

© 2000 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Functional Testing Approaches for “BIFST-able” *tlm_fifo*

Homa Alemzadeh¹, Stefano Di Carlo², Alberto Scionti², Paolo Prinetto², Zainalabedin Navabi¹

¹CAD Research Laboratory, ECE Department
School of Engineering, University of Tehran, Iran
Email: {homa, navabi}@cad.ut.ac.ir

²Dipartimento di Automatica e Informatica
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 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 strategy, early quality evaluation metrics must be introduced during the de-
sign 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](image)

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

<table>
<thead>
<tr>
<th>#</th>
<th>TLM Method</th>
<th>Initial State</th>
<th>Final State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>put()</td>
<td>Empty</td>
<td>Semi-Full</td>
</tr>
<tr>
<td>n-1</td>
<td>put()</td>
<td>Semi-Full</td>
<td>Full</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Full</td>
<td>Blocked put()</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Full</td>
<td>Semi-Full</td>
</tr>
<tr>
<td>n-2</td>
<td>get()</td>
<td>Semi-Full</td>
<td>Semi-Full</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Empty</td>
<td>Blocked get()</td>
</tr>
<tr>
<td>1</td>
<td>peek()</td>
<td>Empty</td>
<td>Blocked peek()</td>
</tr>
<tr>
<td>1</td>
<td>put()</td>
<td>Blockked get()</td>
<td>Empty</td>
</tr>
<tr>
<td>1</td>
<td>peek()</td>
<td>Blockked peek()</td>
<td>Semi-Full</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Semi-Full</td>
<td>Empty</td>
</tr>
</tbody>
</table>

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(), t_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+1th 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 by some comparison methods in response to each put() method.

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 \texttt{tlm_fifo}.

Figure 3 shows the FSM model for the BIFST-able version of \texttt{tlm fifo}. This diagram shows the functionality of BIFST-able \texttt{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 \texttt{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 \texttt{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 \texttt{tlm fifo} as follows:

1. Faulty \texttt{put()}/\texttt{nb_put()}/\texttt{nb_can_put()};
2. Faulty \texttt{get()}/\texttt{nb_get()}/\texttt{nb_can_get()};
3. Faulty \texttt{peek()}/\texttt{nb_peek()}/\texttt{nb_can_peek()}.

Table 2 lists the complete set of high level functional faults defined for these subsets and their possible low level ones. It is worth mentioning that these low level faults are manually derived from the \texttt{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 \texttt{get()} and \texttt{peek()} methods of \texttt{tlm fifo} both perform a read operation on the \texttt{circular_buffer} embedded inside the fifo itself. The main difference is that the \texttt{get()} method updates the value of the read pointer and the \texttt{used()} variable of \texttt{circular_buffer}, while \texttt{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 \texttt{get()} method assures us about the healthy read operation of \texttt{circular_buffer} and the only remaining possible causes of faulty operation of the \texttt{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 \texttt{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 \texttt{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 \texttt{put()} transactions to the \texttt{tlm fifo} until it becomes full, and then performing \texttt{get()} operations to read the written data. The pseudorandom sequence $X_{i}, k_{i}$ is used as test data. This sequence will test the functionality of \texttt{put()} and \texttt{get()} in Empty, Semi-Full, and Full states of \texttt{tlm fifo}. The sequence detects many faults of Table 2 related to the blocking behavior of \texttt{put()}, \texttt{get()}, and \texttt{peek()} (because of its similarity to \texttt{get()}).

In order to optimize the test sequence and avoid additional \texttt{put()} and \texttt{get()} calls required to make \texttt{tlm fifo} full and empty, we also perform \texttt{nb_can_put()}, \texttt{nb_can_get()}, and \texttt{nb_peek()} calls in the middle of successive \texttt{put()} and \texttt{get()} calls. This gives us the added advantage of testing the functionality of these methods.

After this first sequence, the \texttt{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 \texttt{tlm fifo} controlling unit. This starts by doing a blocking \texttt{get()} and \texttt{peek()} calls in the Empty state followed by \texttt{nb_can_put()} and \texttt{put()} calls. This tests the blocking behavior of \texttt{get()} and \texttt{peek()} as well as the functionality of \texttt{nb_can_put()} and \texttt{put()} when \texttt{tlm fifo get()} and \texttt{peek()} are blocked.

After this set of tests, the portions of the \texttt{tlm fifo} control that is related to \texttt{nb_can_get()}, \texttt{nb_peek()}, \texttt{get()}, \texttt{nb_can_put()}, and \texttt{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***

<table>
<thead>
<tr>
<th>Fault Subset</th>
<th>Functional Faults</th>
<th>Possible Low-level Causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Set A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Faulty put()          | *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 |
| nb_put()              | *tlm_fifo* Not Full but: a. put() is blocked.  
b. nb_put() is not done and returns false.  
c. nb_can_put() returns false. | Faulty Controller  
Faulty Increment of used()  
Faulty used() Register  
Faulty Comparator |
| nb_can_put()          | put() is blocked for writing something but never returns. | Faulty Controller  
Faulty get() / nb_get() |

| Faulty get()          | *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 |
| nb_get()              | *tlm_fifo* Not Empty but: a. get() is blocked.  
b. nb_get() is not done & returns false.  
c. nb_can_get() returns false. | Faulty Controller  
Faulty Decrement of used()  
Faulty used() Register  
Faulty Comparator |
| nb_can_get()          | get() is blocked for reading something but never returns. | Faulty Controller  
Faulty put() / nb_put() |

| Faulty peek()         | *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 |
| nb_peek()             | *tlm_fifo* Not Empty but: a. peek() is blocked.  
b. nb_peek() isn’t done & returns false.  
c. nb_can_peek() returns false. | Faulty Controller  
Faulty Decrement of used()  
Faulty used() Register  
Faulty Comparator |
| nb_can_peek()         | peek() is blocked for reading something but never returns. | Faulty Controller  
Faulty put() / nb_put() |

| Faulty read pointer   | *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 Inc. of Read Pointer | 
Faulty Data out Bus | |
| Faulty data out bus   | *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 Inc. of Read Pointer | 
Faulty Data out Bus | |
| Faulty read pointer   | *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 Inc. of Read Pointer | 
Faulty Data out Bus | |
Table 3 – Fault-Based Test Sequence for tlm_fifo

<table>
<thead>
<tr>
<th>#</th>
<th>Test Data</th>
<th>tlm_fifo Status</th>
<th>Expected Response</th>
<th>Faults Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nb_can_put()</td>
<td>Empty</td>
<td>false</td>
<td>Set A: 2.c</td>
</tr>
<tr>
<td>1</td>
<td>put(X)</td>
<td>Empty</td>
<td>false</td>
<td>Set A: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Full</td>
<td>true</td>
<td>Set A: 1.a</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Full</td>
<td>X₁, true</td>
<td>Set A: 1.a,3</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Full</td>
<td>true</td>
<td>Set A: 1.a,3</td>
</tr>
<tr>
<td>n-1</td>
<td>put(Xₙ₋₁)</td>
<td>Semi-full</td>
<td>Blocked</td>
<td>Set B: 2.a,2.c</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_put()</td>
<td>Full</td>
<td>false</td>
<td>Set A: 1.c</td>
</tr>
<tr>
<td>1</td>
<td>put(X)</td>
<td>Full</td>
<td>X₂, true</td>
<td>Set B: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Full</td>
<td>true</td>
<td>Set A: 1.a,3</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Full</td>
<td>X₂</td>
<td>Set B: 2.a,3,6</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Semi-full</td>
<td>true</td>
<td>Set C: 2.b</td>
</tr>
<tr>
<td>1</td>
<td>nb_peak()</td>
<td>Empty</td>
<td>false</td>
<td>Set B: 2.c</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set B: 2.a</td>
</tr>
<tr>
<td>n-1</td>
<td>put(X)</td>
<td>Semi-full</td>
<td>Blocked</td>
<td>Set B: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Empty</td>
<td>true</td>
<td>Set A: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set B: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_put()</td>
<td>Empty</td>
<td>true</td>
<td>Set A: 2.b</td>
</tr>
<tr>
<td>1</td>
<td>nb_peak()</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set C: 1.a,3</td>
</tr>
<tr>
<td>1</td>
<td>put(X)</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set B: 1.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_peak()</td>
<td>Empty</td>
<td>true</td>
<td>Set A: 2.b</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set B: 1.a,3</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set C: 1.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_put(Xₙ₋₁)</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set A: 2.b</td>
</tr>
<tr>
<td>1</td>
<td>peek()</td>
<td>Empty</td>
<td>Blocked</td>
<td>Set C: 1.a</td>
</tr>
<tr>
<td>n-1</td>
<td>nb_put(Xₙ₋₁)</td>
<td>Semi-full</td>
<td>true</td>
<td>Set A: 2.b</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_get()</td>
<td>Full</td>
<td>false</td>
<td>Set A: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>nb_peak()</td>
<td>Full</td>
<td>Blocked</td>
<td>Set B: 2.a</td>
</tr>
<tr>
<td>1</td>
<td>get()</td>
<td>Full</td>
<td>X₂, true</td>
<td>Set B: 2.a,3,6</td>
</tr>
<tr>
<td>1</td>
<td>nb_can_peak()</td>
<td>Full</td>
<td>true</td>
<td>Set C: 2.c</td>
</tr>
<tr>
<td>1</td>
<td>nb_get()</td>
<td>Full</td>
<td>X₇, true</td>
<td>Set B: 2.b,5</td>
</tr>
</tbody>
</table>

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 \( \uparrow \) (i.e., the scanning sequence goes from cell 0 to cell n-1), and (ii) reverse order denoted by \( \downarrow \) (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 \( W \)

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.

The 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 BIFSTs is valuable for both hardware and software implementation of the \textit{tlm_fifo}. Considering the possible functional faults that may appear in the \textit{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 \textit{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 \textit{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 \textit{tlm_fifo} primitives.

### 5.1. March Test for \textit{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 \textit{tlm_fifo}. We consider a \textit{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 $m_{\text{wi}}$ and $m_{\text{ri}}$, respectively, and implementing the circular buffer. We also consider the following behavior of the main \textit{tlm_fifo} methods that will be used to build the march test:

- \texttt{nb_get()} returns the element pointed by $m_{\text{ri}}$ and then increments the read pointer as $(m_{\text{ri}} + 1) \mod N$;
- \texttt{nb_put(P)}: inserts the object $P$ in the element pointed by $m_{\text{wi}}$ and increment the write pointer as $(m_{\text{wi}} + 1) \mod N$;
- \texttt{nb_peek()} is equivalent to \texttt{nb_get()} but $m_{\text{ri}}$ is not incremented after the read operation;
- \texttt{nb_poke(P)}: is equivalent to \texttt{nb_put(P)} but $m_{\text{wi}}$ is not incremented after the write operation.

We consider non blocking primitives since the blocking poke primitive is not available in the \textit{tlm_fifo}. We also resort to the additional \texttt{c_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 \textit{tlm_fifo} as follows:

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

Applying this rule, it is possible to apply any SAO march test to the \textit{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 SAO March B- (Figure 4) to a 4 elements \textit{tlm_fifo}.

\begin{equation}
\begin{align*}
\{ & \texttt{nb} (W_p) ; \texttt{nb} (R_p, W_p, W_p) ; M_0 \\
& \texttt{nb} (R_p, W_p, W_p) ; \texttt{nb} (R_p, W_p, W_p) ; \texttt{nb} (R_p) ; M_1 \\
& \texttt{nb} (R_p, W_p, W_p) ; \texttt{nb} (R_p, W_p, W_p) ; \texttt{nb} (R_p) \} \\
& M_2 \\
& M_3 \\
& M_4
\end{align*}
\end{equation}

\textbf{Figure 4: SOA-March B-}
Table 4 - Application of SOA-March B-

<table>
<thead>
<tr>
<th>Initial Pointers Position</th>
<th>Content</th>
<th>m Ri</th>
<th>m Wi</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb_put(P)</td>
<td>P</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nb_poke(P)</td>
<td>P</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>nb_put(P)</td>
<td>P</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>nb_poke(P)</td>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>compare(nb_peek(),P)</td>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nb_poke(P*)</td>
<td>p</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>compare(nb_peek(),P*)</td>
<td>p</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nb_poke(P)</td>
<td>P</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>compare(nb_get(),P)</td>
<td>P</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nb_poke(P)</td>
<td>P</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>compare(nb_peek(),P*)</td>
<td>p</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>nb_poke(P)</td>
<td>P</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>compare(nb_get(),P)</td>
<td>P</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>nb_poke(P)</td>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nb_put(P)</td>
<td>P</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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