Testbench Qualification of SystemC TLM Protocols through Mutation Analysis

Nicola Bombieri, *Member*, *IEEE*, Franco Fummi, *Member*, *IEEE*, Valerio Guarnieri, *Member*, *IEEE*, and Graziano Pravadelli, *Member*, *IEEE*

Abstract—Transaction-level modeling (TLM) has become the *de-facto* reference modeling style for system-level design and verification of embedded systems. It allows designers to implement high-level communication protocols for simulations up to $1000\times$ faster than at register-transfer level (RTL). To guarantee interoperability between TLM IP suppliers and users, designers implement the TLM communication protocols by relying on a reference standard, such as the standard OSCI for SystemC TLM. Functional correctness of such protocols as well as their compliance to the reference TLM standard are usually verified through user-defined testbenches, whose high quality and completeness play a key role for an efficient TLM design and verification flow. This article presents a methodology to apply mutation analysis, a technique applied in literature for SW testing, for measuring the testbench quality in verifying TLM protocols. In particular, the methodology aims at (i) qualifying the testbenches by considering both the TLM protocol correctness and their compliance to a defined standard (i.e., OSCI TLM), (ii) optimizing the simulation time during mutation analysis by avoiding mutation redundancies, and (iii) driving the designers in the testbench improvement. Experimental results on benchmarks of different complexity and architectural characteristics are reported to analyze the methodology applicability.

Index Terms—Transaction-level modeling (TLM), functional qualification, mutation analysis

1 Introduction

Transaction-level modeling (TLM) is nowadays the reference modeling style for embedded system design and verification at system-level [1]. It greatly speeds up the design process by allowing designers to model and verify the system at different abstraction levels [2], [3].

The Open SystemC Initiative (OSCI) [4] committee has been developing a reference standard for TLM in the last years for guaranteeing the maximum interoperability between suppliers and users, thus encouraging the use of virtual platforms for fast simulation prior to the availability of the RTL code. TLM-2.0 has eventually become the de-facto reference standard for SystemC TLM [5].

The OSCI TLM standard provides a library of primitives for implementing standard interfaces, such as blocking, non-blocking, direct memory interface, etc. Such interfaces are adopted for implementing different standard protocols (e.g., loosely-time, approximately-time, etc.) each one having more or less details according to the target use case (i.e., SW development, HW verification, architectural analysis, etc.) [5].

Correctness of such protocol implementations and their compliance with the OSCI standard are dynamically verified through TLM testbenches, which are also implemented by designers. As a consequence, issues like measure of quality of such testbenches as well as testbench improvement are fundamental for an efficient verification flow.

On the other hand, mutation analysis and mutation testing have definitely gained consensus during the last decades as being important techniques for software testing [6]. Mutation analysis is presented as an approach to validate the effectiveness of a test suite with respect to its ability to discover errors in software programs [7], while mutation testing is the process of generating new test suites to improve the mutation analysis score [8]. The testing approaches rely on the creation of several versions of the program to be tested, *mutated* by introducing syntactic changes. The purpose of such mutations consists of perturbing the program behavior to see whether the test suite is able to detect the difference between the original program and the mutated versions. The effectiveness of the test suite is then measured by computing the percentage of detected mutations. Similar concepts are applied also for HW testing, when verification engineers use high-level fault simulation to measure the quality of test benches [9], and test pattern generation to improve fault coverage, thus providing more effective test suites for the design under verification (DUV) [9].

Mutation analysis has never been applied for measuring how good and reliable are testbenches to verify SystemC TLM protocols, by also considering the protocol compliance to a defined standard. What is missing is:

- a reference model for representing the standard communication protocols, to capture the design errors related to the protocol;
- a model for representing design errors through mutants, strictly related to the communication protocol rather than the design functionality;
- a technique to inject mutants in code statements related to the protocol implementation rather than injecting mutants throughout the code.

• *The authors are with the Department of Computer Science, University of Verona, 37134 Verona, Italy. E-mail: nicola.bombieri@univr.it.*

Manuscript received 11 May 2012; revised 28 Oct. 2012; accepted 12 Dec. 2012; published online 30 Dec. 2012; date of current version 29 Apr. 2014.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2012.301

0018-9340 © 2012 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

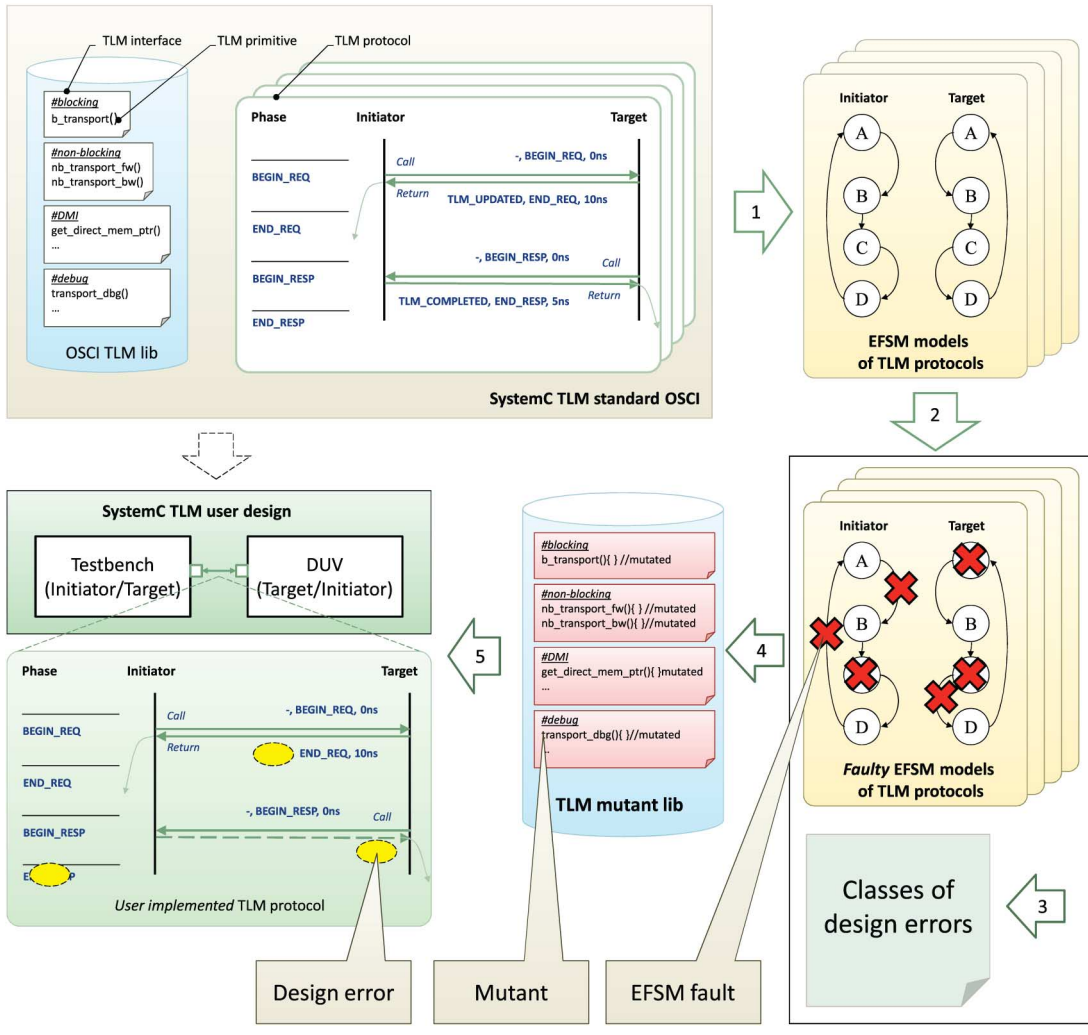


Fig. 1. Methodology overview.

It is important to note that designers may implement a TLM protocol compliant to the standard in different ways. In fact, OSCI provides a library of primitives for different standard interfaces and a set of specifications that define the standard TLM protocols (see upper left side of Fig. 1). By using the library and following the protocol specifications, designers implement their own OSCI compliant protocol (e.g., loosely-timed, 4-phase approximately-timed, etc.). Thus, the mutation analysis should give a quality measure of the testbench (i.e., *testbench qualification*) to verify the protocol correctness and compliance independently from their specific implementations.

In this context, this article proposes a methodology to apply mutation analysis for testbench qualification by considering the OSCI TLM-2.0 as the reference standard. The methodology, whose overview is shown in Fig. 1, consists of the following steps:

1. Modeling of the TLM standard primitives and the TLM protocols through the extended finite state machine (EFSM) formal model [10].
2. Definition of a fault model for EFSM and identification of *faults* into the EFSMs of TLM protocols. The proposed EFSM fault model is based on an extension of the well-known transition fault model for FSMs [11];
3. Identification of a set of classes of *design errors* that designers may commit during the TLM protocol implementation.
4. Implementation of a library of *mutants*, which are a mutated version of the standard TLM primitives and that are defined for implementing the EFSM faults. A correlation between *design errors* and *EFSM faults* is identified to define the minimal set of mutants that implements all possible design errors, in order to optimize the simulation phase of the mutation testing.
5. An automatic injection technique for injecting the mutants into the user design. The mutant injection is performed by replacing the TLM standard primitives instantiated and completed by the user with the mutants provided by the TLM mutant library.

It is important to note that, given a library of primitives and the specification of a reference protocol, the library of mutants is generated once for all and applies to the mutation analysis of all the SystemC TLM designs implemented with those primitives.

Moreover, the correlation between *design errors* and *EFSM faults*, and the corresponding generated *mutants* allow designers to exploit information on not-killed mutants for improving the testbench.

The rest of the article is organized as follows. Section 2 presents the related work. Section 3 introduces the background for the proposed methodology, including the TLM-2.0 standard, the notion of mutation analysis, and the EFSM model. The methodology is presented in Sections 3–7. Section 4 presents how TLM interfaces and protocols are represented by EFSMs. Section 5 presents the EFSM fault model for TLM protocols. Section 6 presents the classes of identified design errors. Section 7 shows how the minimal set of mutants is identified for implementing the TLM mutant library. Section 8 presents the technique for automatically injecting mutants into the user TLM design by adopting the TLM mutant library. Section 9 shows the experimental results. Finally, conclusions are drawn in Section 10.

2 Related Work

Mutation analysis has been actively researched for over 30 years in the SW testing community and it has been applied to many different programming languages (e.g., Fortran, C, Java, C#, SQL) [8], [12]–[14]. In particular, several approaches [15], [16], empirical studies [17] and frameworks [18] have been presented in the literature for mutation analysis. Different aspects concerning software implementation are analyzed in all these works, in which the approaches are mainly suited for Java or C constructs. However, all these approaches target basic constructs and low-level synchronization primitives rather than high-level constructs and primitives typically used for modeling SystemC TLM designs.

Only in recent years mutation analysis has been applied to languages for system-level design and verification such as SystemC [19]–[25]. [19], [20] propose mutation models for perturbing SystemC TLM descriptions. In particular, these works present different analysis of the main constructs provided by the first drafts of the TLM standard and a set of mutants to perturb the primitives related to the TLM communication interfaces.

[21] proposes a fault model by developing mutation operators for concurrent SystemC designs. In particular it aims at verifying SystemC descriptions by facing non-determinism and concurrency problems such as starvation, interference and deadlock typical of such language.

[22] introduces the concept of functional qualification for measuring the quality of functional verification of TLM models. In particular, it presents the results obtained by combining the mutation model of [19] with the testcases and assertions of an industrial functional verification framework.

[23] proposes to attack the verification quality problem for concurrent SystemC programs by developing mutation testing based coverage metrics. The approach involves a comprehensive set of mutation operators for concurrency constructs in SystemC and defines a concurrent coverage metric considering multiple execution schedules that a concurrent program can generate.

[24] introduces an automatic fault localization approach for TLM designs based on bounded model checking. Faults and possible fixes are first formalized in a fault model. Then simulation is used to produce an error trace showing the violation of the specification. Given this trace, bounded model checking is used to determine diagnoses among the possible diagnosis candidates.

[25] presents SCEMIT, a tool for the automated injection of errors into C/C++ SystemC models. A selection of mutation style errors is supported, and injection is performed though a plugin interface in the GCC compiler, which minimizes the impact of the proposed tool on existing simulation flows. The results show the value of high-level error injection as a coverage measure compared to conventional code coverage measures.

This article is based and extends the works presented in [19], [20]. Compared to [19], this article presents a new mutation model, which applies to the TLM primitives, interfaces and communication protocols of the final TLM 2.0 standard OSCI (which is structurally and semantically different from the first draft discussed in [19]). In particular, the new mutation model deals with the new protocols (i.e., approximately timed, loosely timed, etc.) and the new primitive semantics introduced by the TLM standard. Compared to [20], this article extends the analysis of the mutation model and its application from the single primitives to complete TLM protocols.

Finally, compared to all previous works, this article presents a comprehensive methodology for generating a library of mutants and the technique for automatically injecting the mutants into the user TLM design. The methodology aims at (i) generating the minimal set of mutants for reducing redundancy during the mutation analysis, and (ii) improving the feedback readability of the mutation analysis to locate the detected design errors in the user design. To do that, the work presents an analysis of correlation between design errors, EFSM faults, and mutants.

3 Background

This section introduces the key concepts of TLM and mutation analysis. Then it presents the formal model whereby TLM communication protocols are represented.

3.1 TLM: Use Cases, Interfaces, Primitives, and Protocols

The OSCI committee explicitly recognizes the existence of a variety of *use cases* in TLM, such as SW development, SW performance analysis, architectural analysis, and HW verification. However, rather than defining an abstraction level around each use case, the TLM-2.0 standard defines a set of *interfaces* (i.e., blocking, non-blocking, direct memory, and debug interfaces) and provides a library of *primitives* (i.e., `b_transport()`, `nb_transport()`, etc.) for implementing the communication side of transaction-level models. Then, the standard specifies a number of *protocols* that are appropriate for, but not locked to, the various use cases.

The different TLM protocols allow designers to describe and simulate a design with different levels of detail. The best-suited protocol is selected depending on the target use case and each protocol is implemented by using specific TLM primitives (as explained in Section 4.1). The most adopted TLM-2.0 protocols are loosely-timed and approximately-timed:

Loosely-timed (LT) is appropriate for software development, by using, for example, a virtual platform model of an MPSoC, where the software may include one or more

TABLE 1
Examples of Mutation Operator

Source code	Mutant #1	Mutant #2	Mutant #3

`if(a && b)`
`  c = 1;`
`else`
`  c = 0;`
	`if(a || b)`
	`  c = 1;`
	`else`
	`  c = 0;`
		`if(a && b)`
		`  c = 2;`
		`else`
		`  c = 0;`
			`if(a && b)`
			`  c = 1;`
			`else`
			~~`  c = 0;`~~

operating systems. Models implemented with this protocol have a loose dependency between timing and data. They do not depend on the advancement of time to be able to produce a response and, normally, resource contention and arbitration are not considered.

Approximately-timed (AT) is appropriate for architectural exploration and performance analysis. Models implemented with this protocol have a much stronger dependency between timing and data. Since these models must synchronize the transactions before processing them, they are forced to trigger multiple context switches in the simulation, eventually resulting in performance penalties. On the other hand, they easily model resource contention and arbitration.

Loosely-timed and approximately-timed are two examples of protocols proposed by OSCI but, in principle, users can extend the set of protocols by defining their own reference protocols. However, the proposed methodology and mutation model are independent from any reference protocol. For the sake of clarity, the article refers to the standard OSCI protocols as they are the most widespread in the SystemC community.

3.2 Mutation Analysis

Mutation analysis relies on the concept of creating several models of the design under verification (e.g., a SW program), each one *mutated* by introducing a syntactically correct functional change (*mutant*). The purpose of such mutations consists of perturbing the behavior of the model and to verify whether the test suite is able to detect the difference between the original model and the mutated versions [26]. A transformation rule that generates a mutant from the original program is called a *mutation operator*. Table 1 shows the generation of three mutants of a C++ code fragment, which replace *and* operator (&&) with *or* operator (||) and change two assignment statement. Other operators could be considered as well. Typical mutation operators are designed to modify variables, expressions and assignments by replacement, insertion or deletion operators.

Once a mutant has been generated, a test set is supplied to the system. The outputs of the model without the mutation are compared with those of the model with the mutation. If a difference can be observed on such outputs then the mutant is considered *killed*, otherwise it is said to be *survived* [27].

After all test sets have been executed, a few mutants may be still survived. Verification engineers can thus provide additional test inputs to kill such survived mutants and, thus, to improve the test set quality.

If a mutant cannot be killed by any possible sequence of inputs, such mutant is said to be *equivalent*. A model that has equivalent mutants is syntactically different but functionally equivalent to the model with no mutants. Automatically detecting equivalent mutants is impossible as such a model equivalence is undecidable [28].

Mutation analysis provides designers with an adequacy score, known as *mutation score*, which indicates the quality of the input test set. The mutation score is the ratio of the number of killed mutants over the total number of non-equivalent mutants. The goal of mutation analysis is to measure how far mutation score is from 100%, which indicates that the test set is sufficient to detect all the design errors represented by the mutants.

This work proposes to apply mutation analysis to TLM SystemC model of communication protocols.

3.3 The EFSM Model

An EFSM [29] is a transition system which allows a more compact and intuitive representation of the state space with respect to the traditional finite state machines (FSMs). The EFSM model is widely used for modeling complex systems like reactive systems [30], communication protocols [31], buses [32] and controllers driving data-path [33]. EFSM has been adopted in this work to easily represent communication protocols based on transactions as well as mutations on TLM code, as explained in Section 4. Among different alternatives, the EFSM formal model has been chosen also because it captures the main characteristics of the state-oriented, activity oriented and structure-oriented models [34].

Definition 1. *An EFSM is defined as a 5-tuple* $M = \langle S, I, O, D, T\rangle$ *where:* S *is a set of states,* I *is a set of input symbols,* O *is a set of output symbols,* D *is a n-dimensional linear space* $D_1 \times \ldots \times D_n$, T *is a transition relation such that* T*:* $S \times D \times I \rightarrow S \times D \times\ O$*. A generic point in* D *is described by a* n*-tuple* $x = (x_1, , x_n)$*; it models the values of the registers internal to the design.*

A pair $\langle s, x\rangle \in S \times D$ is called *configuration* of M, while an operation on an EFSM $M = \langle S, I, O, D, T\rangle$ is defined as follows:

Definition 2. *If* M *is in a configuration* $\langle s, x\rangle$ *and it receives an input* $i \in I$*, it moves to the configuration* $\langle t, y\rangle$ *iff* $((s, x, i), (t, y, o)) \in T$ *for* $o \in O$*.*

In an EFSM, each transition is associated with a couple of functions (i.e., an *enabling function* and an *update function*) acting on input, output and register data. The enabling function expresses a set of conditions on data, while the update function consists of a set of statements performing operations on data.

Definition 3. *Given an EFSM* $M = \langle S, I, O, D, T\rangle$, $s \in S, t \in T, i \in I, o \in O$ *and the sets* $X = \{x|((s, x, i), (t, y, o)) \in T$ *for* $y \in D\}$ *and* $Y = \{y|((s, x, i), (t, y, o)) \in T$ *for* $x \in X\}$*, the* enabling *and* update *functions are defined respectively as:*

$$e(x,i) = \begin{cases} 1 & \text{if } x \in X; \\ 0 & \text{otherwise.} \end{cases}$$

$$u(x,i) = \begin{cases} (y,o) & \text{if} e(x,i) = 1 \text{ and} \\ & ((s,x,i),(t,y,o)) \in T \\ undef. & \text{otherwise.} \end{cases}$$

Fig. 2 gives an example of the state transition graph (STG) of an EFSM. A transition is fired if all conditions in the enabling function are satisfied, bringing the machine from

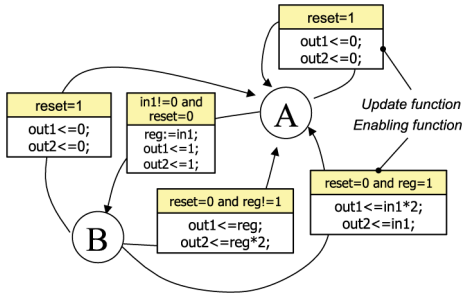


Fig. 2. Example of EFSM.

the current to the destination state and performing the operations included in the update function.

4 EFSM Models of TLM Interfaces and Communication Protocols

Representing TLM standard primitives and communication protocols with the EFSM model allows us (i) to focus on what can be altered in the code and (ii) to identify similarities that can be shared. More specifically, the use of EFSM models shifts the focus towards specific aspects of the communication protocol (e.g., payload fields, communication phases, etc.) that a generic model may ignore. Representing communication primitives with EFSM models enables the identification of design errors that can be implemented in the same way, i.e., by the same mutant. Hence, going through an EFSM representation before injecting mutants leads to a reduction of the number of mutants with respect to the number of possible design errors. Section 7.2 deals with this point more in detail.

The following sections summarize the features of the main SystemC TLM 2.0 primitives and protocols, and propose their formalization by means of EFSMs.

4.1 EFSM Models of TLM Primitives

TLM communication between an *initiator* and a *target* module relies on the exchange of packets containing data and control values, through a communication channel. The TLM library provides users with standard packets (*TLM payload*) which are passed as a primitive parameter through standard channels (e.g., a *TLM socket*).

Each TLM interface has a proper primitive (or set of primitives), packet and communication channel. The EFSM models of the TLM primitives are grouped into four main categories, corresponding to the standard interfaces: blocking, non-blocking, direct memory, and debug interfaces.

4.1.1 Blocking Interface

The blocking interface provides a simplified communication mechanism, suited for models which complete a transaction in a single primitive call. The TLM standard primitive is `b_transport ()` and the corresponding EFSM model is shown in Fig. 3(a). The EFSM model consists of three states (A, B, C) and two transitions. The initiator starts a transaction by firstly setting the payload and the local time to send to the target and, then, calling the primitive (transition $A \rightarrow B$). The enabling function is set to *true* as there are no explicit conditions on any event for starting a transaction in the TLM blocking interface (see Section 4.2).

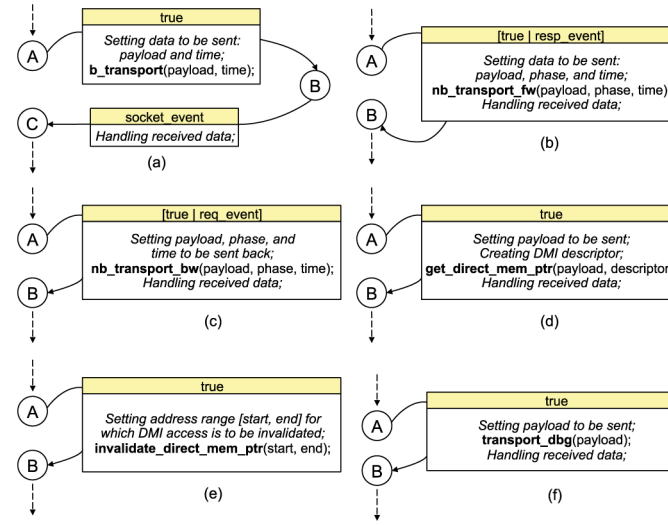


Fig. 3. EFSM models of the communication primitives for *blocking* (a), *non-blocking* (b, c), *DMI* (d, e), and *debug* (f) interfaces.

The EFSM is blocked in the intermediate state B waiting for an event from the channel (*socket_event*). State B represents the time in which the primitive body is executed, that is, the time in which the target executes the required functionality and sets the results into the returning payload. When the primitive returns, the socket event is triggered and the EFSM moves to the final state C, that is, the initiator can retrieve the payload and handle the returning data held in it.

4.1.2 Non-Blocking Interface

With the non-blocking interface mechanism, a transaction is decomposed into phases, aiming to implement a more accurate and detailed communication protocols. Such phases, as well as payload and timing information, are passed as parameters through the primitives `nb_transport_fw ()` and `nb_transport_bw ()`. The primitive `nb_transport_fw ()` is called by the initiator and implemented by the target, while vice-versa for the nb_transport_bw (). Aside from their name, the two primitives have similar semantics and are called in combined way for implementing detailed protocols, such as the approximately-timed one. The EFSM models of the two non-blocking primitives consist of two states A and B [see Fig. 3(b) and(c)] as they perform the requested operation as soon as they are called and return to the caller who can handle immediately the received data. Depending on the communication protocol, the enabling function can be always true or an explicit condition on a event (i.e., a response or a request event, as explained in Section 4.2).

4.1.3 Direct Memory Interface (DMI)

DMI is used by an initiator for directly accessing to a specified area of the target memory. It relies on the use of a direct pointer to the memory area, without having to resort to the transport interface. DMI guarantees an increment in simulation speed, as it skips the interconnect components located between the initiator and the target, thus speeding up simulation for the transactions that operate on memory in a loosely-timed model. The EFSMs of the non-blocking primitives for modeling DMIs (i.e., `get_direct_mem_ptr ()` and `invalidate_direct_mem_ptr ()`) are shown in Fig. 3(d) and (e).

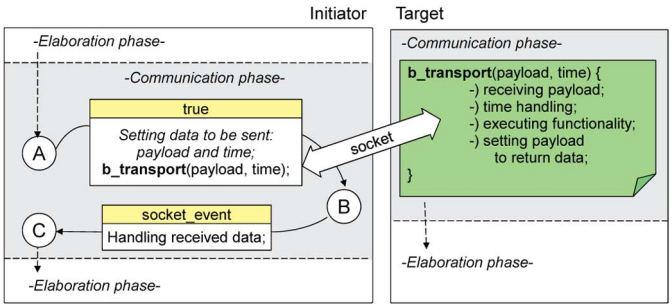


Fig. 4. EFSM model of the loosely-timed protocol.

4.1.4 Debug Interface

Debug interface aims at allowing designers to debug the communication protocol. Such a kind of interface allows the initiator to directly access a target memory disregarding delays information, temporization, event notifications or side effects that may occur in a transaction. It exploits the same channel used by the transport interfaces, but the access to the target is immediate. The EFSM model of the non-blocking debug primitive (i.e., `transport_dbg ()`) is shown in Fig. 3(f).

Hereafter, the article will focus on the loosely-timed and approximately-timed TLM protocols for the sake of clarity, even if the methodology applies to any other TLM protocol.

4.2 EFSM of TLM-2.0 Communication Protocols

Loosely-timed and approximately-timed TLM protocols are implemented with the blocking and non-blocking primitives, respectively.

Fig. 4 depicts the EFSM model of the loosely-timed communication protocol, in which the initiator and target behaviors have been divided into the elaboration (i.e., functional computation) and communication phases. The sequence of steps can be summarized as follows:

1. The initiator moves from an elaboration to a communication phase as soon as it starts a transaction, in which, (i) it sets the data to send to the target (i.e., payload and time), (ii) it calls the blocking primitive through the socket, and (iii) it waits for an event from the socket (see Section 4.1).
2. The primitive body is implemented into the communication side of the target and is executed in the target during the communication phase. The primitive body typically consists of (i) receiving the payload through the pointer passed as parameter, (ii) taking care of the time information for temporization, (iii) executing the target functionality, and (iv) modifying some payload fields to send the resulting data back to the initiator.
3. According to the behavior of the blocking primitives, a socket event is notified to the initiator as soon as the primitive returns, after which the initiator can receive the resulting data and conclude the transaction.

The EFSM model of the 4-phases approximately-timed protocol is depicted in Fig. 5. The transaction is composed of four phases: *begin request*, *end request*, *begin response*, *end response*. Each phase involves a primitive call. The sequence of steps can be summarized as follows:

1. The initiator starts the transaction in A by setting payload, time, and initializing the first protocol phase to *begin request* (transition $A \rightarrow B$). Then it calls *nb_transport_fw()* through the socket.
2. The target executes the primitive body by receiving the payload, handling the time, notifying a *request event* and returning a resulting status back to the initiator.
3. The initiator handles the received data and concludes the first communication phase in state B. Then it moves to the intermediate elaboration phase. In the meanwhile, the target process (woken up by the request event) carries on the second phase, *end request*, (transition $A \rightarrow B$) by calling `nb_transport_bw ()` through the socket.
4. The initiator executes the primitive by handling the received time information and returning a status to the target.
5. The target handles the received data and concludes the second communication phase in state B. Then it moves to the intermediate elaboration phase.
6. The target moves to a new communication phase, by starting the third phase, *begin response* ($C \rightarrow D$). It sets the payload fields to answer the starting request by the initiator, the time information, and calls again the non-blocking primitive along the backward path.
7. The initiator executes the primitive body, notifies the *response event* and returns its outcome to the target.
8. The target handles the received data and concludes the third communication phase in state D. Then it moves to a new elaboration phase. In the meanwhile, the initiator process (woken up by the response event) carries on the fourth and last phase, *end response* ($C \rightarrow D$) by calling the primitive `nb_transport_fw ()` through the socket.
9. The target executes the primitive by handling the received time information and returning a status to the initiator.
10. The initiator handles the received data and concludes the fourth communication phase in state D. Then it moves to a new elaboration phase.

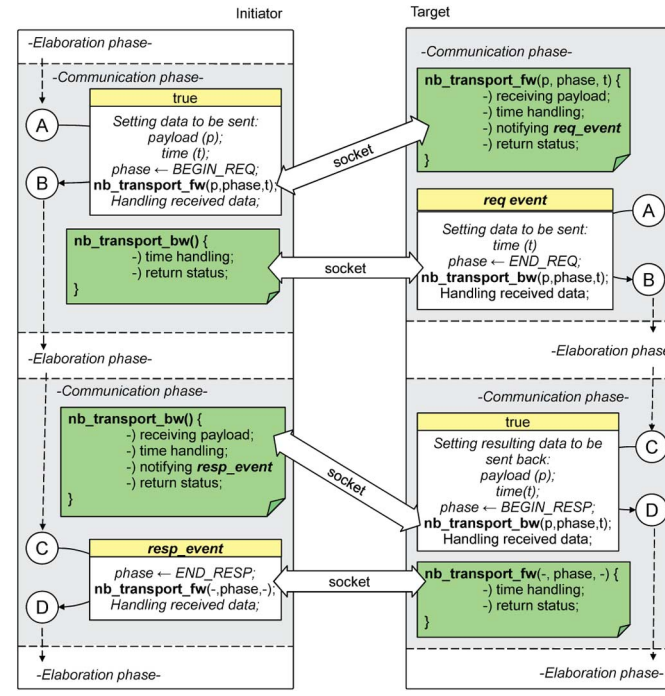


Fig. 5. EFSM model of the 4-phases AT protocol.

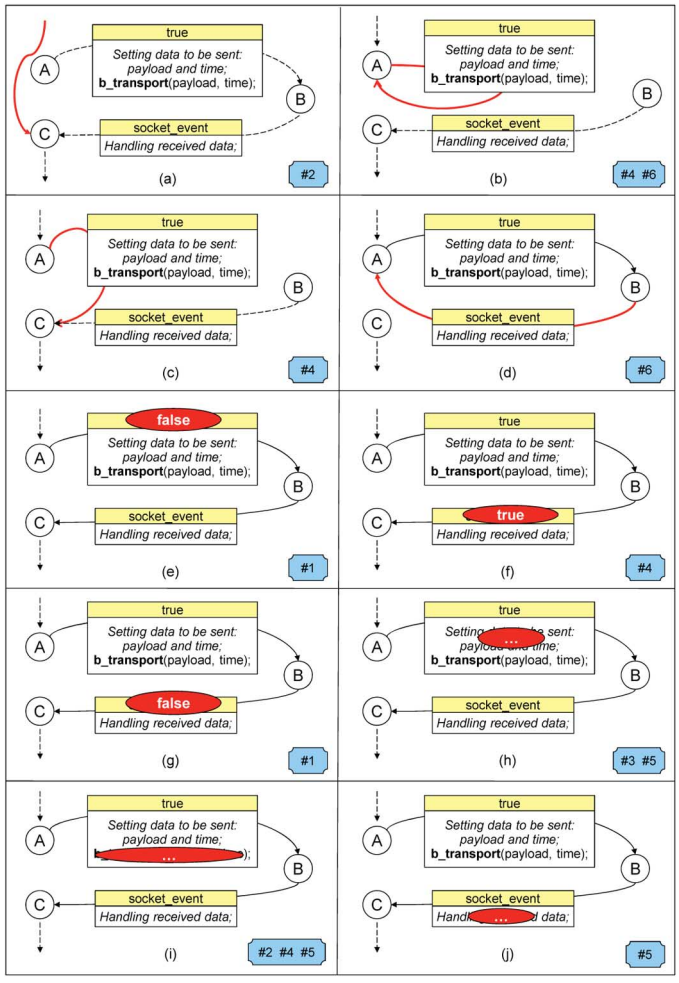


Fig. 6. Faults on the `b_transport ()` EFSM model.

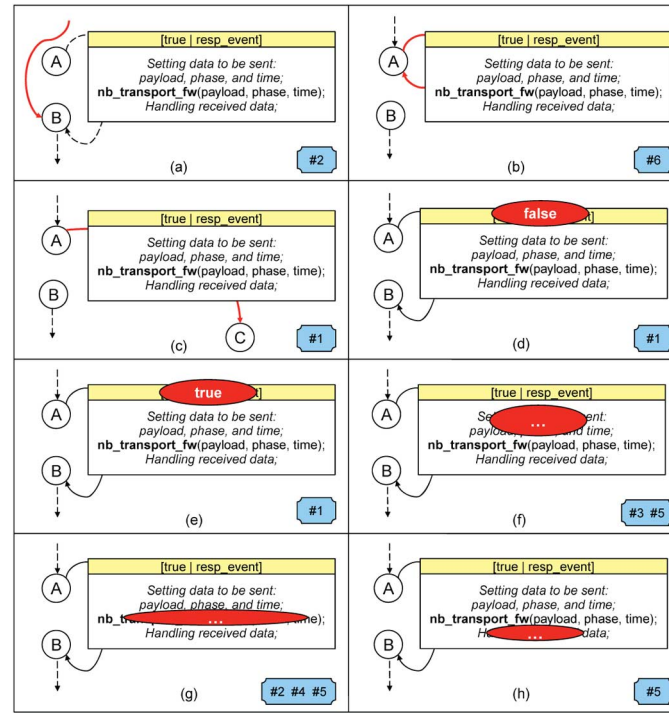


Fig. 7. Faults on the `nb_transport_fw ()` EFSM model.

The non-blocking interface is explicitly intended to support pipelined transactions and to model communication with a high degree of timing accuracy. Many other TLM communication protocols can be created by using the TLM interfaces previously described, and their EFSM models can be represented by sequentially composing the EFSMs of the involved primitives. The proposed methodology is independent from any specific protocol, and can be applied to any other TLM protocol in a similar way as shown for the chosen examples.

5 EFSM Fault Model for TLM Protocols

According to the classification of errors that may affect FSM proposed in [35], different fault models have been defined for perturbing FSMs [11], [36]. They generally target Boolean functions labeling the transitions and/or destination states of transitions. In this work, an EFSM is perturbed in a similar way by changing the destination state of transitions and/or modifying the behavior of enabling and update functions.

Hereafter, this section shows how the EFSMs of Fig. 3 are perturbed to generate mutated versions of the TLM primitives. The analysis focuses on the EFSM faults of primitives `b_transport ()` (Fig. 6) and `nb_transport_fw ()` (Fig. 7). The EFSM of the others primitives are similarly perturbed.

The EFSM faults are classified as follows:

- *Faults on destination states*. This class includes the faults that change the destination state of a transition. Examples are shown in Fig. 6(a)–(d) for primitive `b_transport ()` and Fig. 7(a)–(c) for primitive `nb_transport_fw ()`.
- *Faults on enabling function*. Faults of this class change the value of the enabling function associated to a transition. Examples are shown in Fig. 6(e)–(g) and Fig. 7(d) and (e) in which the enabling function value is switched to *false* or *true* for preventing or forcing the transition to fire.
- *Faults on update function*. All the faults that make changes to the code in the update functions belong to this class. In particular, they can be divided into three sub-classes:
  - Faults on the code *before* the primitive call. They involve the implementation code for setting the data to be sent (i.e., payload, time, and phase) [Figs. 6(h) and 7(f)].
  - Faults *in* the primitive call. They involve the primitive parameters and the whole code that implements the primitive [Figs. 6(i) and 7(g)].
  - Faults on the code *after* the primitive call. They involve the implementation code for handling the received data [Figs. 6(j) and 7(h)].

The numbers reported in the bottom-right part of each EFSM identify the design error classes defined in Section 6 that can be put in correspondence to the EFSM faults, as described in Section 7.1.

The proposed fault model is an extension of the well-known transition fault model [11], applied to generic descriptions. Given the complexity and the flexibility required for this purpose, EFSMs are better suited than traditional FSMs, since they allow for more expressive power. In this context, it is possible to extract a subset of transition faults which can be mapped to a corresponding design error, by tracking each

transformation taking place within the EFSM, one by one. To the best of our knowledge, without the use of the EFSM model it would not be feasible to identify a set of mutants corresponding to any possible design error.

6 Identification of Design Errors

Six classes of typical TLM design errors have been identified, by relying on the expertise we have acquired by working with designers of TLM platforms in different European projects [37], [38], by working with designers of TLM industrial platforms [39], and analyzing several TLM designs of literature [40]–[42][1]:

1. *Deadlock in the communication phase*. This class includes all the design errors due to the wrong use of primitive `wait()`. These errors leave the system in a deadlock state, in which one or more processes are left indefinitely waiting for an event which may be triggered only by a process belonging to the same set. Considering the EFSM model of `b_transport ()` in Fig. 3(a), a deadlock is represented by the transition $A \rightarrow B$ (or $B \rightarrow C$) not being fired, and by A (or B) becoming the final state. Similarly, a deadlock in the EFSM model of the non-blocking primitives in Fig. 3(b) and (c) is represented by the transition $A \rightarrow B$ not being fired, and by state A becoming the final state.
2. *Misapplication of primitive calls*. It includes errors on the primitive calls as well as the setting of the primitive parameters. Omitting a `b_transport()` call when expected is an example, as well as missing a `nb_transport_bw ()` call for advancing to the next transaction phase before initiating a new transaction, and, finally, setting a wrong phase to a multi-phase protocol (e.g., in a AT-based protocol). These errors can be represented by changing the destination state of the incoming transition from A to C, or by altering the update function of the transition $A \rightarrow B$ in the EFSM model of the `b_transport ()` primitive. Similarly, the destination state of the incoming transition can be changed from A to B, or the update function of the transition $A \rightarrow B$ can be modified to represent these errors in the EFSM models of the non-blocking primitives.
3. *Misapplication of TLM operations*. Using a wrong command with respect to the operation to be accomplished (e.g., setting a *write* command for reading data instead of *read*) leads to errors belonging to this class. The use of a wrong command is represented in the EFSM models of the primitives by a change in the update function of transition $A \rightarrow B$. The command field in the payload is set to `TLM_WRITE_COMMAND` instead of `TLM_READ_COMMAND`, or viceversa.
4. *Misapplication of blocking/non-blocking primitives*. Usually a wrong combination of the blocking and non-blocking interfaces is responsible for this error class. This wrong combination can be represented in the EFSM model of the blocking primitive by changing the enabling function of the transition $B \rightarrow C$ to be always true, or by changing the update function of the transition $A \rightarrow B$. The same effect can be achieved for the non-blocking primitives by editing the update function of the transition $A \rightarrow B$.
5. *Erroneous handling of the generic payload*. This class covers all the errors related to a misuse of the payload object, including write and read operations on the payload fields. These errors are represented by changes in the payload object, which is set and sent in the update function of the transition $A \rightarrow B$ in the EFSM models of the blocking and non-blocking primitives.
6. *Erroneous polling mechanism*. Errors caused by a misuse of polling belong to this class. They typically lead to an infinite loop during simulation. This loop can be represented by changing the destination state of transition $B \rightarrow C$ to B (or of transition $A \rightarrow B$ to A) for the blocking transport primitive, and the destination state of transition $A \rightarrow B$ to A for the non-blocking transport primitives, thus modeling an infinite loop.

Each of the previous error classes has been associated with at least one fault on the EFSM models of the TLM protocols, as described in Section 7.1. It is worth noting that errors leading to problems during the code compilation are not considered in this work, according to the theory of the mutation analysis.

1. Although to the best of our knowledge the six classes cover all design errors, any further design error can be added to the list without altering the methodology.

7 TLM Mutant Library

The mutant library consists of a set of TLM primitives, which are mutated versions of the standard TLM primitives and that are defined for implementing the EFSM faults. A correlation between *design errors* and *EFSM faults* is identified to define the minimal set of such TLM mutants in order to avoid redundant mutants and optimize the simulation phase of the mutation testing.

7.1 Design Errors vs. EFSM Faults

The design errors described in Section 6 can be put in correspondence with the EFSM faults defined in Section 5. In particular: (i) each EFSM fault allows us to represent at least one class of design errors, and (ii) each design error can be represented by one (and only one) EFSM fault. Such relationship analysis is reported in the following only for primitives `b_transport ()` and `nb_transport_fw ()`, but the same analysis approach applies to all other primitives.

7.1.1 Primitive `b_transport ()`

Faults on destination states of the EFSM allow us to represent classes #2, #4, and #6 of design errors. In particular:

- Fault of Fig. 6(a) makes the transition incoming to state A to be deviated to state C. As a consequence, transitions $A \rightarrow B$ and $B \rightarrow C$ are skipped. It models an omission of the primitive call by the designer (class #2).
- Fault of Fig. 6(b) makes the transition outcoming from state A to be closed in an infinite loop. It models a misuse of polling combined with the erroneous use of the non-blocking instead of blocking primitive (classes #4 and #6) (i.e., the primitive is called in an infinite loop and returns without waiting for any socket event).
- Fault of Fig. 6(c) models an erroneous use of the non-blocking instead of blocking primitive (class #4).
- Fault of Fig. 6(d) models a misuse of polling (class #6).

Concerning faults on the enabling functions:

- Fault of Fig. 6(e) makes the enabling function of transition $A \rightarrow B$ to be stuck at *true*. The transition is prevented from being fired and the machine actually gets stuck in the same state. It models a deadlock in the communication protocol, i.e., the fact that the primitive is never called by the initiator (class #1).
- Fault of Fig. 6(f) makes the enabling function of transition $B \rightarrow C$ to be stuck at *true*. The transition is fired, without waiting for any socket event. It models an erroneous use of the non-blocking instead of blocking primitive (class #4).
- Fault of Fig. 6(g) makes the enabling function of transition $B \rightarrow C$ to be stuck at *false*. The transition is prevented from being fired and the machine actually gets stuck in the same state. It models a deadlock in the communication protocol (class #1), but in this case, the deadlock is caused by a design error in the target implementation.

Considering faults on update functions:

- Fault of Fig. 6(h) models design errors in the initiator code before the primitive call, that is, wrong setting of the payload field (classes #3 and #5).
- Fault of Fig. 6(i) models general design errors in calling the primitive or setting the primitive parameters (classes #2, #4, and #5).
- Fault of Fig. 6(j) models design errors in the initiator code after the primitive call, that is, wrong handling of the received data (class #5)

7.1.2 Primitive `nb_transport_fw ()`

The non-blocking behavior of the primitive often leads to some different matchings between EFSM faults and classes of design errors w.r.t. the blocking primitive. In particular:

- Fault of Fig. 7(a) models an omission of the primitive call by the designer (class #2), as for the blocking primitive.
- Fault of Fig. 7(b) makes the primitive to be called in an infinite loop, modeling a misuse of polling (class #6).
- Fault of Fig. 7(c) makes the transition outgoing from state A to go into a new intermediate state C, from which the EFSM never comes out and, thus, never reaches the state B. This allow us to model a deadlock in the communication protocol (class #1) and, in particular, the fact that the non-blocking primitive is called but never returns to the caller (i.e., the deadlock is caused by a design error in the target implementation).

Considering faults on enabling functions:

- Fault of Fig. 7(d), similarly to the corresponding blocking version, prevents the transition from being fired and the machine actually gets stuck in state A. It models a deadlock in the communication protocol (class #1) caused by an implementation error in the initiator code.
- Fault of Fig. 7(e) makes the enabling function of transition $A \rightarrow B$ to be stuck at *true*. As a consequence, the EFSM misses any synchronization event [e.g., the response event of Fig. 5], leading to a possible deadlock (class #1).

Finally, faults on update functions [Fig. 7(f)–(h)] allow us to represent the same classes of design errors as in the case of blocking primitive.

On the other hand, it is possible to represent each design error by at least one EFSM fault. Each error may be committed in the initiator, in the target, or in both of them. For example, the error of forgetting to call the primitive will be certainly located in the initiator, while the error of wrongly setting the response status of the payload will be located in the target. Errors related to a deadlock may be located into both the initiator and the target, as the wait statement that leads to the deadlock may be used in any module involved in the communication protocol.

TABLE 2
Design Errors in the Initiator Code for the *b_transport()* Primitive and Correlation with EFSM Faults

Design error	Description	Class	EFSM fault	Cat.
1	Forgetting to call the primitive	#2	6a	DS
2	Calling *nb_transport_fw()* instead of *b_transport()*	#4	6c	
3	Calling the non-blocking primitive with a misuse of polling	#4,#6	6b	
4	Misuse of polling	#6	6d	
5	Forgetting to wait for call completion	#4	6f	EF
6	Forgetting to handle return from the primitive call	#5	6j	UFA
7-13	Wrong handling of the response status returned by the primitive call			
14	Primitive call in dead code	#1	6e	EF
15	Wrong command (*command* field)	#3	6h	UFB
16	Wrong address (*address* field)			
17	Forgetting to set the data pointer (*data pointer* field = NULL)			
18	Corrupted data pointer (*data pointer* field modified so that corrupted data are read when accessed)			
19	Wrong data length (*data length* field)			
20	Wrong streaming width (*streaming width* field)			
21	Forgetting to set the byte enable pointer (*byte enable pointer* field = NULL)			
22	Corrupted byte enable pointer (*byte enable pointer* field modified so that corrupted data are read when accessed)			
23	Wrong byte enable length (*byte enable length* field)			
24	Wrong primitive parameter (*payload pointer*)	#2	6i	UFI

Tables 2–5 report in detail the list of all possible design errors that may be located in the initiator and in the target, for primitives *b_transport()* and *nb_transport_fw()*, respectively. Column *Class* reports the class to which the design error belongs, while columns *EFSM fault* and *Cat.* report the EFSM fault that represents such design error and the corresponding category. The categories are indicated with the following abbreviations (see Section 5): *DS*: destination states; *EF*: enabling function; *UFB*: update function - before the primitive call; *UFI*: update function - in the primitive call; *UFA*: update function - after the primitive call.

A set of design errors with the same description (e.g., 7-13 in Table 2) represent all possible wrong combinations a designer may commit in setting an attribute (i.e., enumerable

TABLE 3
Design Errors in the Target for *b_transport()* Primitive and Correlation with EFSM Faults

Design error	Description	Class	EFSM fault	Cat.
1	Misuse of *wait()* in the primitive body	#1	6g	EF
2	Wrong command use (*command* field)	#3	6i	UFI
3	Wrong address use (*address* field)	#5		
4	Forgetting to set back the data pointer (*data pointer* field = NULL)			
5	Corrupted data pointer (*data pointer* field modified so that corrupted data are read when accessed)			
6	Wrong data length use (*data length* field)			
7	Wrong streaming width use (*streaming width* field)			
8	Forgetting to set back the byte enable pointer (*byte enable pointer* field = NULL)			
9	Corrupted byte enable pointer (*byte enable pointer* field modified so that corrupted data are read when accessed)			
10	Wrong byte enable length use (*byte enable length* field)			
11-18	Wrong response status (*response status* field)			

TABLE 4
Design Errors in the Initiator for *nb_transport_fw()* Primitive and Correlation with EFSM Faults

Design error	Description	Class	EFSM fault	Cat.
1	Forgetting to call the primitive	#2	7a	DS
2	Calling *b_transport()* instead of *nb_transport_fw()*	#4	7g	
3	Misuse of polling	#6	7b	
4	Forgetting to set the protocol phase (*phase* parameter)	#2	7f	UFI
5	Forgetting to handle return from the call to primitive		7e	UFA
6-8	Wrong return handling from the primitive call			
9	Primitive call in dead code	#1	7d	EF
10	Wrong command (*command* field)	#3	7f	UFB
11	Wrong address (*address* field)	#5		
12	Forgetting to set the data pointer (*data pointer* field = NULL)			
13	Corrupted data pointer (*data pointer field modified* so that corrupted data is read when accessed)			
14	Wrong data length (*data length* field)			
15	Wrong streaming width (*streaming width* field)			
16	Forgetting to set the byte enable pointer (*byte enable pointer* field = NULL)			
17	Corrupted byte enable pointer (*byte enable pointer* field modified so that corrupted data are read when accessed)			
18	Wrong byte enable length (*byte enable length* field)			
19-23	Wrong primitive parameter (*phase*)	#2	7g	UFI
24	Wrong primitive parameter (*payload pointer*)			

value) of a payload field. In the example, errors from 7 to 13 cover the seven possible attributes of the *response status* field of the payload.

Section 7.2 shows how the correlation between design errors and EFSM faults identified so far is exploited to implement the *minimal* set of mutants. It is important to note that in case a new design error was identified, the same approach applies to identify the correlation between such a new error and an EFSM fault. The identified correlation is then exploited to map the new design error to an already existing mutant or to implement a new mutant from scratch.

7.2 Identification of the Minimal Set of TLM Mutants

The correlations identified in the previous section can be grouped into three categories, each one involving the implementation of one or more mutants:

1. *1-1 to 1*. One design error is correlated to one EFSM fault, which is implemented by a single mutant.
2. *n-1 to n*. Multiple design errors are correlated to the same EFSM fault, which is implemented by multiple mutants.
3. *n-n to 1*. Different design errors are correlated to different EFSM faults, which are implemented by the same mutant.

The first category is the most straightforward, since there is a unique association between a design error, its corresponding EFSM fault and the mutant implementing such a fault. For example, design error 1 in Table 2 falls within this scenario. It is associated with EFSM fault 7(a), which is responsible for altering the destination state of the transition incoming to state *A*. This EFSM fault is implemented by one mutant that executes such a state alteration in the code.

The second category involves design errors that are correlated to the same fault, because they share the EFSM element they operate on (e.g., the same update or enabling function), but it is not possible to implement one mutant for all design errors. As an example, the design errors 2.7-2.13 correspond to an improper handling of the response status returned by the primitive call. They are all correlated to the EFSM fault 7(j), which operates on the update function that models the initiator code after the primitive call has completed. However, a different mutant for each design error is needed to properly implement every possible alteration to the response status.

The third category is the most interesting, since it allows a reduction of mutants. Design errors 2.2 and 2.5 are correlated with different EFSM faults, 7(c) and 7(f) respectively, and they are implemented by the same mutant. In fact, two different misconceptions in the design process end up producing the same code. In this case, this means invoking the *nb_transport_fw()* primitive instead of the *b_transport()* primitive.

Design errors 2.15-2.23 and 3.2-3.10 have the same behavior. They represent alterations to the correct values of payload fields. For example, errors 2.15 and 3.2 represent a wrong value in the *command* field. The first set of errors is correlated with the EFSM fault 7(h), while the second is correlated with

TABLE 5
Design Errors in the Target for the *nb_transport_fw()* Primitive and Correlation with EFSM Faults

Design error	Description	Class	EFSM fault	Cat.
1	Misuse of *wait() in* the primitive body	#1	7c	DS
2	Wrong command use (*command* field)	#3	7g	UFI
3	Wrong address use (*address* field)	#5		
4	Forgetting to set back the data pointer (*data pointer* field = NULL)			
5	Corrupted data pointer (*data pointer* field modified so that corrupted data are read when accessed)			
6	Wrong data length (*data length* field)			
7	Wrong streaming width (*streaming width* field)			
8	Forgetting to set back the byte enable pointer (*byte enable pointer* field = NULL)			
9	Corrupted byte enable pointer (*byte enable pointer* field modified so that corrupted data are read when accessed)			
10	Wrong byte enable length (*byte enable length* field)			
11-15	Wrong protocol phase (*phase* parameter)	#2		
16	Wrong response status (*response status* field)	#5		
17-20	Wrong return value	#2		

the EFSM fault 7(i). This is reasonable, since the two sets belong to two different communication sides, i.e. initiator and target. Nevertheless, in the code, the alteration of a payload field is achieved by the same mutant, regardless of whether the initiator or the target performed it. For this reason, each error in the first set and its corresponding error in the second set are implemented by the same mutant.

The use of the EFSM models plays a fundamental role in such a reduction, since it allows to group together design errors sharing the same target and to observe where their effects would be located within the communication protocol. This is extremely useful for the mutant implementation since, beside reducing their number, it ensures that all the mutants being implemented are tied to at least one possible design error. This prevents from injecting mutants that turn out to be unnecessary, since they do not map back to a design error and, thus, they do not reflect a real error that may be introduced during the development of TLM designs.

8 Mutant Injection into TLM Designs

The mutant injection into the the user design is automatically performed by exploiting the library implemented in the step before. Before simulation, by means of a code parser tool, each occurrence of the original primitives in the user design is renamed as *golden* primitive, and the corresponding mutated primitive provided by the TLM mutant library is inserted, as shown in Fig. 8.

Each mutated primitive implements a set of mutants, which represent the EFSM faults. The mutants represent the EFSM faults by perturbing the simulation execution [e.g., by replacing the primitive call by an empty code for implementing the fault of Fig. 6(a)] or changing the payload fields.

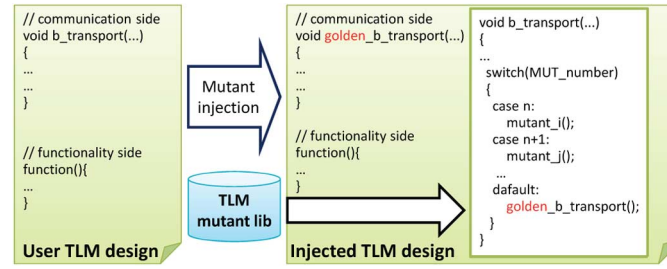


Fig. 8. Mutant injection of primitive `nb_transport_fw ()`.

All mutants are enumerated. At run time, one mutant at a time is activated through the `MUT_number` variable, thus employing a mechanism based on mutant schemata [43]. If the primitive contains the activated mutant, the primitive switches to the corresponding mutated behavior. Otherwise, the primitive simulates in the *fault free* mode through the golden primitive. Each mutant may simultaneously simulate more than one design error, according to the mapping identified in the previous sections. In this way, all possible design errors can be tested with a reduced set of mutants.

Finally, mutation analysis reports a coverage measure (i.e., mutation score) that expresses the quality of the TLM design testbench to find all possible design errors (as explained in Section 3.2). In particular, the analysis indicates which classes of design errors are not covered by the testbench and where such errors could be located in the TLM design (e.g., during the payload setting in the initiator, in the primitive call, etc.) thus allowing designers to better improve the testbench structure.

9 Experimental Results

The proposed methodology has been applied for qualifying the testbenches of the following benchmarks:

- the design examples released with the OSCI TLM-2.0 library [5];
- an Error Correction Code (ECC) module in both versions LT and AT [37], provided by STMicroelectronics;
- the Fast Fourier Transform (FFT) module of the Magali platform [38] in LT version, provided by STMicroelectronics and CEA-LETI.

The TLM mutant library has been implemented and includes the mutated version of all the primitives of the standard OSCI TLM2.0 library. The mutant library has been applied for the testbench qualification of all benchmarks. A SystemC framework has been developed to perform mutation analysis, by using the testbenches provided with the benchmarks. Each benchmark is stimulated through a single testbench. It is worth noting that such testbenches have been modeled by people unaware of our mutation model. The framework works according to the single mutation assumption. That is, during simulation, one single mutant is activated at a time. Then the output result is compared with the one obtained through a simulation with no mutants activated (i.e., through all the golden primitives, as explained in Section 8). If the output result differs, the framework reports that the activated mutant has been detected. The output result

TABLE 6
Experimental Results: Detected Mutants

Design	P	LoC	App	Det	Cov	T (s)
OSCI_LT	3	821	138	101	73.2	3.6
OSCI_LT_DMI	2	1058	92	58	63.0	3.5
OSCI_LT_ _temporal_decoupling	4	1065	184	114	62.0	5.6
OSCI_AT_1_phase	1	1514	49	26	53.1	2.3
OSCI_AT_2_phase	2	1572	98	49	50.0	4.7
OSCI_AT_4_phase	4	1727	196	96	49.0	12.9
ECC_LT	5	667	230	126	54.8	2.5
ECC_AT	2	727	98	36	36.7	1.1
FFT_LT	1	4885	46	28	60.9	3.4

depends on the benchmark. In the case of the TLM examples, the output result consists of the print statements on the standard output as a simulation trace. In the case of ECC and FFT, the output results consists of the output produced by the design functionality, in addition to the aforementioned print statements making up the simulation trace. We used the *MuffinTLM* tool of the *HIFSuite* framework [44] for the mutant injection phase and adopted the simulation kernel provided by the standard SystemC 2.2 release.

In this section, we refer to an *applied* design error as an error for which a corresponding mutant implementing it has been injected in the design. Then, we refer to a *detected* design error as an error for which a corresponding mutant implementing it has been injected in the design and killed by the testbench during simulation.

We also refer to *mutations* on enabling/update function or on destination state as mutants implementing EFSM faults belonging to the corresponding category (i.e., faults on destination state, on enabling function, on update function *before*, *in* and *after* the primitive call).

Tables 6 and 7 report the experimental results.

Table 6 focuses on mutants. Column P reports the number of TLM transport primitive invocations in the design. Column *LoC* indicates the size of the TLM description in terms of lines of code. Columns *App* and *Det* list the number of applied and detected mutants, respectively. Column *Cov* reports the coverage in terms of percentage of killed mutants. Column T reports the total simulation time in seconds.

Table 7 shows the results from the point of view of detected design errors. Columns A and D show respectively the number of applied and detected design errors for each category of the corresponding EFSM fault, according to the classification of Section 5 and to the correlation identified in Section 7.1. Column *Total* summarizes the total number of design errors considered and the coverage in terms of percentage of detected design errors. Column *Coverage (%)* reports the mutation score for each class of design errors according to the error classification of Section 6.

By comparing Tables 6 and 7, it is evident that coverage in terms of detected mutants is lower than coverage in terms of detected design errors for each design. The reason for this behavior is that the same mutant may implement more than one design error, as pointed out in Section 7.2.

The achieved mutation coverage proves that testbenches released with the examples are not accurate enough to detect some possible design errors in the communication protocol. The results show that mutations that alter the destination state or modify the enabling function of a transition are quite easy to detect. They lead to observable situations during simulation (i.e., infinite loops, deadlocks or abrupt terminations) especially for those implementing design errors belonging to classes #1 and #6 for protocols based on both blocking (i.e., LT) and non-blocking primitives (i.e., AT). Design errors belonging to classes #2 and #4 are more difficult to be detected in case of non-blocking (multiphase) and blocking primitives, respectively.

In contrast, mutations on update functions *before* the primitive calls are much more difficult to detect, in particular those that set some specific payload fields (classes #3 and #5). This is due to the fact that not every payload field is used in each transaction (e.g., the fields related to the *byte enable pointer* are employed in a limited number of circumstances). A modification to one of those values does not produce visible effects on the output if they are not used, and, as a consequence, the testbench is unable to detect the corresponding mutant. Designers are allowed to improve the testbench if necessary or to consider such errors as redundant after a more detailed analysis of the design.

On the other hand, the results show that mutation detection heavily relies on the testbench quality in case of: mutations on update function *in* and *after* the primitive calls (classes #2, #4, and #5), mutations that alter the primitive body (classes #2, #3, and #5) and mutations that handle the return values of the primitive calls (classes #2 and #5). If the testbench checks the returned values after each primitive call and is able to handle anomalies or error messages, the probability of detecting these mutants is drastically improved. For several benchmarks, the corresponding testbenches fail to cover such mutations. Even if many of such errors are not often recurring, high-quality testbenches should detect them to avoid error conditions in the communication phases. Even if the coverage

TABLE 7
Experimental Results: Detected Design Errors and Coverage on Design Error Classes

Design	Destination State		Enabling Function		Update Function						Total			Coverage (%)					
					Before		In		After					Design errors					
	A	D	A	D	A	D	A	D	A	D	A	D	%	1	2	3	4	5	6
OSCI_LT	24	22	15	13	54	18	57	53	48	48	198	154	77.7	100	100	100	77.8	72.2	100
OSCI_LT_DMI	16	16	10	10	38	18	38	28	32	14	134	86	64.2	100	100	100	100	51.0	100
OSCI_LT_ _temporal_decoup	32	32	20	20	72	32	76	52	64	30	264	166	62.9	100	100	100	100	49.0	100
OSCI_AT_1_phase	7	7	2	2	18	7	33	19	8	2	68	37	54.4	100	51.5	33.3	100	44.0	100
OSCI_AT_2_phase	14	14	4	4	36	12	66	36	16	5	136	75	52.2	100	53.0	66.7	100	42.0	100
OSCI_AT_4_phase	28	28	8	8	72	24	132	70	32	10	272	140	51.5	100	50.8	41.7	100	40.0	100
ECC_LT	40	36	25	21	90	35	95	77	80	34	330	203	61.5	100	100	100	73.3	50.4	100
ECC_AT	14	14	4	4	36	12	66	24	16	2	136	56	41.1	100	34.8	33.3	100	34.0	100
FFT_LT	8	8	5	5	18	7	19	17	16	8	66	45	68.2	100	100	100	100	56.3	100

is quite low by itself, it should be noted that the importance of an undetected mutant depends on the design itself, according to the TLM protocol being implemented for communication purposes. The reason for this lies in that the TLM-2.0 standard provides a number of payload fields which are not strictly required, i.e. they can be optionally set according to the communication protocol to be modeled in the TLM description. As such, an undetected mutant on a field not used in the communication protocol is definitely not as negative as an undetected mutant causing a missing primitive or a deadlock, for instance.

10 Concluding Remarks

This article presented a testbench qualification methodology for SystemC TLM protocols through mutation analysis. The article showed how the methodology has been defined for (i) qualifying the testbenches by considering both the TLM protocol correctness and their compliance to a defined standard, (ii) optimizing the simulation time during mutation analysis by avoiding mutation redundancies, and (iii) helping designers in the testbench improvement. The experimental results confirmed the methodology effectiveness by highlighting, for example, the inability of the testbenches provided with the benchmarks to cover all features of the adopted communication protocols.

References

[1] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proc. ACM/IEEE CODES + ISSS*, 2003, pp. 19–24.

[2] F. Ghenassia, A. Clouard, K. Jain, L. Maillet-Contoz, and J.-P. Strassen, *Using Transactional Level Models in a SoC Design Flow*. Norwell, MA: Kluwer, 2003.

[3] A. Donlin, "Transaction level modeling: Flows and use models," in *Proc. ACM/IEEE CODES + ISSS*, 2004, pp. 75–80.

[4] OSCI, (2009). *Accellera Systems Initiative* [Online]. Available: http://www.accellera.org, Accessed 2013.

[5] TLM-2.0, *OSCI TLM-2.0 Language Reference Manual*, Open SystemC Organization Initiative, 2009, Available: http://www.systemc.org

[6] D. Hyunsook and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Comput.*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[8] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[9] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. New York, NY: Computer Science Press, 1990.

[10] D. Lee and M. Yannakakis, "Online minimization of transition systems (extended abstract)," in *Proc. ACM Int. Symp. Theory Comput. (STOC'92)*, 1992, pp. 264–274.

[11] K.-T. Cheng and J.-Y. Jou, "A single-state-transition fault model for sequential machines," in *Proc. IEEE Int. Conf. Comput.-Aided Des. (ICCAD'90)*, 1990, pp. 226–229.

[12] M. E. Delamaro and J. C. Maldonado, "Proteum—A tool for the assessment of test adequacy for C programs," in *Proc. Conf. Performability Comput. Syst. (PCS'96)*, 1996, pp. 79–95.

[13] S. A. Irvine et al., "Jumble Java byte code to measure the effectiveness of unit tests," in *Proc. Mutat. Test. Workshop*, 2007, pp. 169–175.

[14] J. Tuya, M. J. Suarez-Cabal, and C. De La Riva, "SQLMutation: A tool to generate mutants of SQL database queries," in *Proc. Mutat. Test. Workshop*, 2006, pp. 39–43.

[15] R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia, "Mutation of Java objects," in *Proc. IEEE Int. Symp. Softw. Reliability Eng. (ISSRE'02)*, 2002, pp. 341–351.

[16] F. Belli, C.-J. Budnik, and W.-E. Wong, "Basic operations for generating behavioral mutants," in *Proc. IEEE Int. Symp. Softw. Reliability Eng. (ISSRE'06)*, 2006, pp. 10–18.

[17] M.-R. Lyu, H. Zubin, S.-K. S. Sze, and C. Xia, "An empirical study on testing and fault tolerance for software reliability engineering," in *Proc. of IEEE Int. Symp. Softw. Reliability Eng. (ISSRE'03)*, 2003, pp. 119–130.

[18] J. S. Bradbury, J. R. Cordy, and J. Dingel, "ExMan: A generic and customizable framework for experimental mutation analysis," in *Proc. IEEE Int. Symp. Softw. Reliability Eng. (ISSRE'06)*, 2006, pp. 4–9.

[19] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *Proc. ACM/IEEE Des., Autom. Test Eur. (DATE'08)*, 2008, pp. 396–401.

[20] N. Bombieri, F. Fummi, and G. Pravadelli, "On the mutation analysis of SystemC TLM-2.0 standard," in *Proc. IEEE Int. Workshop Microprocessor Test Verif. (MTV'09)*, 2009, pp. 32–37.

[21] A. Sen, "Mutation operators for concurrent SystemC designs," in *Proc. IEEE Int. Workshop Microprocessor Test Verif. (MTV'09)*, 2009, pp. 27–31.

[22] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe, "Functional qualification of TLM verification," in *Proc. ACM/IEEE Des., Autom. Test Eur. (DATE'09)*, 2009, pp. 190–195.

[23] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent SystemC designs using mutation testing," in *Proc. IEEE Int. High Level Des. Validation Test Workshop (HLDVT'10)*, 2010, pp. 75–81.

[24] H. Le, D. Grosse, and R. Drechsler, "Automatic TLM fault localization for SystemC," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 31, no. 8, pp. 1249–1262, 2012.

[25] P. Lisherness and K.-T. (Tim) Cheng, "SCEMIT: A SystemC error and mutation injection tool," in *Proc. ACM/IEEE Des. Autom. Conf. (DAC'10)*, 2010, pp. 228–233.

[26] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Proc. Mutat. Anal. Workshop*, 2001, pp. 34–44.

[27] R. Guderlei, R. Just, C. Schneckenburger, and F. Schweiggert, "Benchmarking testing strategies with tools from mutation analysis," in *Proc. IEEE Int. Conf. Softw. Test. Verif. Valid. Workshop (ICSTW'08)*, 2008, pp. 360–364.

[28] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verif. Rel.*, vol. 7, no. 3, pp. 165–192, 1997.

[29] K.-T. Cheng and A. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM TODAES*, vol. 1, no. 1, pp. 57–79, 1996.

[30] T. J. Koo, B. Sinopoli, A. Sangiovanni-Vincentelli, and S. Sastry, "A formal approach to reactive system design: Unmanned aerial vehicle flight management system design example," in *Proc. IEEE Comput.-Aided Control Syst. Des. (CACSD'99)*, 1999, pp. 522–527.

[31] H. Katagiri, K. Yasumoto, A. Kitajima, T. Higashino, and K. Taniguchi, "Hardware implementation of communication protocols modeled by concurrent EFSMs with multi-way synchronization," in *Proc. ACM/IEEE Des. Autom. Conf. (DAC'00)*, 2000, pp. 762–767.

[32] A. Zitouni, S. Badrouchi, and R. Tourki, "Communication architecture synthesis for multi-bus SoC," *J. Comput. Sci.*, vol. 2, no. 1, pp. 63–71, 2006.

[33] A. Guerrouat and H. Richter, "A component-based specification approach for embedded systems using FDTS," *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 2, pp. 14–18, 2006.

[34] D. Gajski, J. Zhu, and R. Domer, "Essential issue in codesign," Univ. California, Irvine, Tech. Rep. ICS-97–26, 1997.

[35] T. Chow, "Testing software design modeled by finite state machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178–187, 1978.

[36] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *Proc. IEEE 5th Int. Symp. Softw. Rel. Eng. (ISSRE'94)*, 1994, pp. 220–229.

[37] Vertigo Project. (2009). *Final Report* [Online]. http://vertigo-project.edalab.it//deliverables/08_D7.1_VERTIGO_R14.pdf

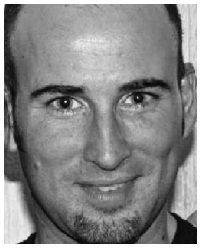
[38] Coconut Project. (2010), *Final Report* [Online]. http://www.coconut-project.eu/files/FP7-2007-IST-1-217069-Coconut-D7.1.pdf

[39] EDALAB partners. *Networked Embedded Systems* [Online]. Available: http://www.edalab.it/en/partner, Accessed 2013.

[40] TLMWG, *Transaction Level Modeling Working Group: OSCI TLM 2.0* [Online]. Available: http://www.accellera.org, Accessed 2013.

[41] ESCUG (European SystemC User's Group), [Online]. Available: www.ti.uni-tuebingen.de/escug

[42] NASCUG (North American SystemC User's Group), [Online]. Available: www.nascug.org, Accessed 2013.
[43] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. 1st ACM SIGSOFT Symp. Found. Softw. Eng. (SIGSOFT'93)*, 1993, pp. 139–148.
[44] HIFSuite [Online]. Available: http://www.hifsuite.com, Accessed 2013.



Nicola Bombieri received the Masters and PhD degrees in Computer Science from the University of Verona, Verona, Italy, in 2004 and 2008, respectively. Since 2008, he is a researcher and assistant professor at the Department of Computer Science, University of Verona. His research interests include TLM design and verification of embedded systems and automatic generation and optimization of embedded SW. He has been involved in several national and international research projects and has published more than 50 papers on conference proceedings and journals.

Franco Fummi received the Laurea and the PhD degrees in electronic engineering both from Politecnico di Milano, Milano, Italy, in 1990 and 1995, respectively. From 1995, he has been with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, with a position of assistant professor. In July 1998, he obtained the position of associate professor in computer architecture at the Dipartimento di Informatica, Università di Verona, Italy. He is a full professor at the Dipartimento di Informatica, Università di Verona since 2001. His research interests include electronic design automation methodologies for modeling, verification, testing, and optimization of embedded systems. He is a member of the IEEE test technology committee.

Valerio Guarnieri received the Masters degree in computer science from the University of Verona, Verona, Italy, in 2009. He has been a PhD student in the Department of Computer Science, University of Verona since 2010. His research interests include design and verification techniques for TLM-based design flows of embedded systems. He has been involved in international projects and has published 14 papers on conference proceedings and journals.

Graziano Pravadelli received the Masters degree in computer science and the PhD degree from the University of Verona, Italy, in 2001 and 2004, respectively. In January 2005, he obtained the position of assistant professor at the Department of Computer Science, University of Verona. Since January 2011, he is an associate professor at the same University. He is the co-founder of EDALab, a spin-off whose mission consists of giving support for innovation and technology adoption in traditional and emerging fields of networked embedded systems. His research interests include hardware description languages, methodologies to describe HW/SW systems, and functional verification of embedded systems. In this context, he collaborated in several national and international projects. He has published more than 90 papers in the electronic design automation field. He is an IEEE member.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.