A Mutation Model for the SystemC TLM 2.0 Communication Interfaces [*]

Nicola Bombieri Franco Fummi Graziano Pravadelli
Dipartimento di Informatica, Università di Verona, Italy
{nicola.bombieri, franco.fummi, graziano.pravadelli}@univr.it

Abstract

Mutation analysis is a widely-adopted strategy in software testing with two main purposes: measuring the quality of test suites, and identifying redundant code in programs. Similar approaches are applied in hardware verification and testing too, especially at RTL or gate level, where mutants are generally referred as faults, and mutation analysis is performed by means of fault modeling and fault simulation. However, in modern embedded systems there is a close integration between HW and SW parts, and verification strategies should be applied early in the design flow. This requires the definition of new mutation analysis-based strategies that work at system level, where HW and SW functionalities are not partitioned yet. In this context, the paper proposes a mutation model for perturbing transaction level modeling (TLM) SystemC descriptions. In particular, the main constructs provided by the SystemC TLM 2.0 library have been analyzed, and a set of mutants is proposed to perturb the primitives related to the TLM communication interfaces.

1 Introduction

Mutation analysis and mutation testing have definitely gained consensus during the last decades as being important techniques for software testing [1]. Mutation analysis is presented as an approach to validate the effectiveness of a test suite with respect to its ability in discovering defects in software programs [2], while mutation testing is the process of generating new test suites to improve the mutation analysis score [3]. Such 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 behavior of the program to see if 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 [4], and test pattern generation to improve fault coverage, thus, providing more effective test suites for the design under verification (DUV). In this case, mutations introduced in the HW descriptions are referred as faults [4].

Nowadays, (i) the close integration between HW and SW parts in modern embedded systems, (ii) the development of high-level languages suited for modeling both HW and SW (like SystemC with the TLM library), (iii) the need of developing verification strategies to be applied early in the design flow, require the definition of mutation analysis-based strategies that work at system level, where HW and SW functionalities are not partitioned yet. In this context, the paper proposes a mutation model targeting the communication primitives of the new SystemC TLM 2.0 library [5]. In particular, the paper analyzes the main primitives provided with such a library, and it presents the following innovative contributions (Figure 1):

- a way for formalizing the internal behavior of the TLM 2.0 communication primitives by using the extended finite state machine (EFSM) model [6];
- a set of mutations for such EFSMs based on an extension of the well-known transition fault model for FSMs [7];
- identification of relations between the proposed mutations and typical design errors.

Based on these EFSM mutations we have implemented mutated versions of the TLM 2.0 primitives that can be used for measuring, via mutation analysis, the quality of test suites defined for verifying SystemC descriptions.

The paper is organized as follows. Section 2 is devoted to related works. Section 3 introduces the main concepts of the SystemC TLM 2.0 library and formalizes the TLM primitives by means of the EFSM model. Section 4 presents the proposed mutation model and its relation with design errors. Section 5 shows the effectiveness of the proposed mutations in measuring the quality of test suites. Finally, conclusions are discussed in Section 6.

2 Related works

Several approaches [8–13], empirical studies [14] and frameworks [15, 16] have been presented in the literature for

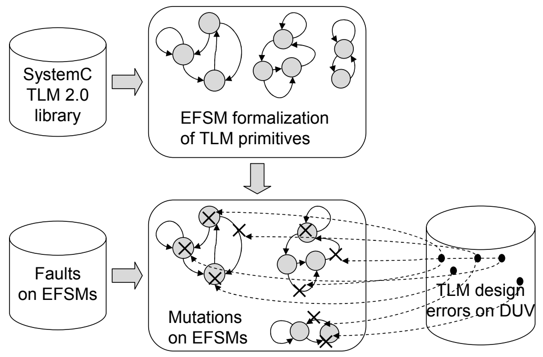


Figure 1: Mutations on TLM 2.0 primitives vs. design errors.

[*]This work has been partially supported by the European project VERTIGO FP6-2005-IST-5-033709. The authors would like to thank Mark Hampton from Certess, and Andrea Fedeli and Umberto Rossi from STMicroelectronics for their valuable contribution in discussing the relation between mutants and design errors.

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.

In [8], a technique for performing mutation analysis on object-oriented programs is presented. Faults are injected into objects (e.g., I/O streams) that instantiate items from common Java libraries and user defined classes. In [9] an extensive set of mutation operators are defined by means of formal specifications to target basic constructs of traditional programming languages, like conditions, mathematical operators, data types, etc.. In [10], very specific mutants are proposed for testing classic synchronization constructs like monitors, semaphores, etc. implemented in Java. However, all these approaches are suited to target basic constructs and low-level synchronization primitives rather than high-level primitives typically used for modeling TLM communication protocols.

Approaches described in [11–13, 17] present mutation operators targeting formal abstract models, independently from specific programming languages. In [11], the great variety of operators used to introduce mutation in graph-based models is gathered into two basic operations (i.e., insertion and omission) and their combinations. In [12], mutation analysis is applied to integration and system level testing in addition to unit level testing. The application of the mutation analysis criterion in the context of specification based on finite state machine is proposed in [13]. Finally, a mutation model for representing a specific class of timing faults is presented in [17] by exploiting the EFSM model. These approaches are valuable to be applied at TLM levels. However, the authors do not show a strict relation between the modeled mutants and the typical design errors introduced during modeling steps.

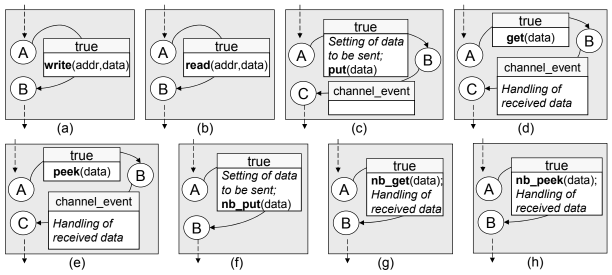
To the best of our knowledge, no paper in the literature addresses the problem of defining mutations for the SystemC TLM 2.0 communication primitives.

3 EFSM models of SystemC TLM 2.0 communication interfaces

Recently, the OSCI committee has proposed a TLM library [18, 19] composed of a set of SystemC primitives that allow designers to implement several TLM communication protocols with different degrees of accuracy. Thus, for each TLM abstraction level, i.e., programmer view (PV), programmer view with time (PVT) and cycle accurate (CA) [20], the best-suited communication protocols can be implemented depending on the purpose of the modeling/verification activity that has to be performed (e.g., architectural exploration, functional verification, performance evaluation, etc.). Such different protocols can be classified according to the following aspects, that characterize the underlying TLM primitives:

- type of the communication channel (i.e., bidirectional or unidirectional primitives);
- communication schema (i.e., blocking or non-blocking primitives);
- timing (i.e., timed or untimed primitives).

In this context, we summarize, hereafter, the features of the main SystemC TLM 2.0 primitives, and for each primitive we



A
B
C
true
write(addr,data)
read(addr,data)
Setting of data to be sent; **put**(data)
channel_event
get(data)
Handling of received data
peek(data)
Setting of data to be sent; **nb_put**(data)
nb_get(data); *Handling of received data*
nb_peek(data); *Handling of received data*
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)

Figure 2: EFSM models of SystemC TLM 2.0 primitives.

propose a formalization by means of the EFSM model. Such a formalization allow us to (i) precisely define the behavior of each primitive, and (ii) define a mutation model (Section 4) to perturb the communication interface of TLM designs according to typical design errors. For sake of simplicity and lack of space, we use graphical descriptions of such EFSMs in the following sections, but a more detailed textual explanation is available in [21].

3.1 The EFSM model

An EFSM is a transition system which allows a more compact and intuitive representation of the state space with respect to the traditional finite state machines. In an EFSM, transitions are 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 is composed of a set of conditions on data, while the update functions is composed of a set of statements performing operations on data. Given a transition out-going from a state, the transition is fired, bringing the machine from the current state to the next state and performing the operation included in the update function, once the conditions involved in the enabling function are all satisfied. The EFSM model is widely used for modeling complex systems like reactive systems [22], communication protocols [23], buses [24] and controllers driving data-path [25].

3.2 Bidirectional/unidirectional primitives

In TLM, communication is generally accomplished by exchanging, through a channel, *request* and *response* packets, containing data and control values, between an initiator (master) module and a target (slave) module. In this context, the choice between a bidirectional interface and an unidirectional interface depends on the degree of detail desired for modeling the design description.

SystemC *Bidirectional primitives* (i.e., `read()` and `write()`[1]) are used to implement a bidirectional interface each time a tight one-to-one binding between a request and the corresponding response is required without taking care of details related to the communication protocol.

Figures 2(a) and 2(b) show the EFSM models corresponding, respectively, to the `write()` and `read()` bidirectional primitives. In both cases, the EFSM is composed of two states. The enabling function of the connecting transition is always

[1]Such primitives rely on the SystemC TLM `transport()` function, which opportunely calls the implementations of `read()` and `write()` defined by the designer.

true, to indicate that as soon as a module calls a `write()` or a `read()`, the corresponding operations are executed and a return value is provided to the caller.

On the other hand, when a more detailed description of the communication protocol is desired, it is more reasonable to break down the protocol operations into a sequence of transfers by exploiting the SystemC *unidirectional primitives* (i.e., `put()`, `peek()` and `get()`). The behavior of `put()` and `get()` is intuitive, they are used to, respectively, send and receive packets on channels. The `peek()` function is called to understand if the channel is ready to provide packet to a subsequent `get()`. By composing such primitives at PVT level, communication protocols can be more accurately described to simulate, for example, complex phases of handshaking between the initiator and the target modules.

Figures 2(c-h) show the EFSM models corresponding to `put()`, `get()` and `peek()`. There exist both blocking and non-blocking (`nb` prefix) unidirectional primitives. The explanation of the corresponding EFSMs is demanded to the next subsection, where concepts related to blocking and non-blocking aspects are detailed[2].

3.3 Blocking/non-blocking primitives

SystemC *blocking primitives* might implicitly call a `wait()` statement to suspend their execution until a synchronization event occurs. In this way, for example, it is possible to model a process that starts a transaction by calling the blocking primitive `put()` to send a request (or response) packet through a communication channel. If the channel is not ready (e.g., since it is full), the `put()` implicitly calls a `wait()` to suspend the process execution until the channel become ready.

On the other hand, processes cannot be suspend if *non-blocking primitives* are used. When a process starts a transaction, by calling a SystemC non-blocking primitive (e.g., `nb_put()`) to send a request (or response) packet, it immediately receives a response concerning if the access to the channel succeeded, and then it continues the execution. Thus, synchronization between processes adopting non-blocking primitives generally relies on polling mechanisms explicitly defined by the designers.

The EFSM models of unidirectional blocking and non-blocking primitives are shown in Figure 2(c-h). For example, the EFSM of the blocking primitive `get()` is composed of three states. Once the `get()` is called, it immediately moves from state *A* (initial state) to state *B* and it asks the channel to provide a packet (either a request or a response). Then, `get()` suspends in state *B* waiting for an event from the channel (*channel event*) indicating that the packet is ready to be retrieved. Finally, the retrieved data are handled by executing the operations included in the update function moving from *B* to the final state *C*. The behavior of `peek()` and `put()` is almost analogous to `get()`, except that `put()` sets the data to be sent moving from *A* to *B*, and it does not perform operations moving from *B*

[2]It is worth noting that the bidirectional `write()` and `read()` primitives exist only in the blocking form.

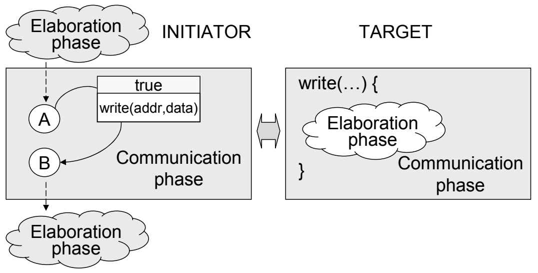


Figure 3: EFSM model of Bidirectional Blocking Untimed TLM protocol.

to *C* once the channel event has been triggered to unblock the execution.

EFSM models of non-blocking primitives are composed of two states only. Removing the wait on the channel, primitives `nb_get()`, `nb_peek()` and `nb_put()` perform the required operation as soon as they are called, and they immediately reach the final state in the corresponding EFSMs. The caller process is informed if the non-blocking primitive succeeded by looking at its return value.

3.4 Untimed/timed primitives

A modeling style in which there is no explicit mention of time or clock cycles, like for example TLM PV, is called untimed. In this case, the sequence of operations performed by TLM PV processes are synchronized by using events and the SystemC *untimed primitives* described in the previous subsections (independently from their bidirectional/unidirectional or blocking/non-blocking characteristics).

On the contrary, TLM primitives are provided with timing annotations to describe timed-approximated models at TLM PVT level. In this way, timed behaviors such as system delays and response latencies can be easily verified. The SystemC *timed primitives* differ from the corresponding untimed versions, since they include a parameter more for expressing time delays. The EFSM models of timed primitives resemble the corresponding untimed EFSMs, but their final state is reached only when the desired delay is elapsed.

3.5 EFSM models of TLM communication protocols

Several TLM communication protocols can be modeled by using the TLM primitives previously described, and their EFSM models can be represented by sequentially composing the EFSMs of the involved primitives. According with the OSCI TLM 2.0 proposal [18, 19], the most relevant protocols are the following.

1) ***Bidirectional Blocking Untimed.*** This protocol is implemented by calling a single untimed bidirectional primitive. Thus, the EFSM model of the protocol is composed only of the EFSM of the used primitive. For example, Figure 3 shows the EFSM model of a bidirectional blocking untimed communication protocol, where the initiator enters the communication phase by calling the `write()` primitive, which is atomically executed by the target. Since the primitive is untimed, the write is executed instantaneously, and the communication phase ends

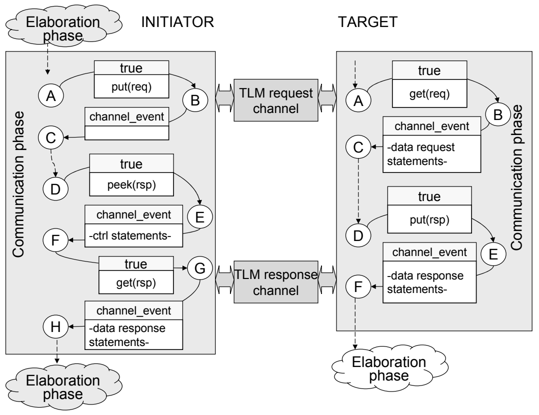


Figure 4: EFSM model of Unidirectional Blocking Untimed TLM protocol.

by providing the return value to the initiator, which can proceed with the subsequent elaboration phase.

2) ***Bidirectional Blocking Timed.*** The EFSM model of such a kind of protocol resembles the corresponding untimed EFSM model. The target, in performing the required operation, takes care of time delays expressed by the parameter passed to the adopted bidirectional primitive.

3) ***Unidirectional Blocking Untimed.*** This protocol is implemented by splitting write and read transactions into a sequence of unidirectional blocking untimed primitives. Figure 4 shows the EFSM models of initiator and target for performing the same operation of Figure 3, by using the unidirectional blocking untimed protocol instead of the bidirectional untimed protocol. The initiator, first, requests a transaction by means of `put(req)` (states *A*, *B*, *C* on the left). Then, once the request succeeds, the initiator peeks the channel for a response by calling `peek(rsp)` (states *D*, *E*, *F* on the left). Finally, it gets the response from the channel by means of `get(rsp)` (states *F*, *G*, *H* on the left) and it proceeds with the subsequent elaboration phase. Analogously, the target, first asks for a request by means of `get(req)` (states *A*, *B*, *C* on the right), and then, after the elaboration phase (which corresponds to the elaboration phase implemented in the `write()` function of Figure 3), it provides a response by calling `put(rsp)` (states D, E, F on the right).

4) ***Unidirectional Non-blocking Untimed.*** In the non-blocking interface, TLM primitives are not allowed to call `wait()` functions. Thus, every call to non-blocking primitives returns a boolean value to indicate whether the access succeeded, and polling mechanisms should be adopted. By considering the example of Figure 4, the EFSMs modeling `put()`, `peek()` and `get()` must be substituted with the EFSM models of, respectively, `nb_put()`, `nb_get()` and `nb_peek()` to represent the same behavior through the implementation of an unidirectional non-blocking untimed protocol.

5) ***Unidirectional Non-blocking Timed.*** The EFSM model of such a kind of protocol resembles the corresponding untimed EFSM model. Timing information is encoded in the enabling and update functions of the EFSM modeling the timed annotated version of `nb_put()`, `nb_get()` and `nb_peek()`.

4 TLM mutation model

In this section we present a mutation model to perturb the communication protocol of TLM 2.0 SystemC-based designs. The strategy we followed for defining such a mutation model consists of the following steps:

1. Identify a set of design errors typically introduced during the design of TLM communication protocols.
2. Identify a fault model to introduce faults (i.e., mutations) in the EFSM representations of the TLM 2.0 primitives.
3. Identify the subset of faults corresponding to the design errors identified at step 1.
4. Define mutant versions of the TLM 2.0 communication primitives implementing the faults identified at step 3.

4.1 Design errors

Based on the expertise we have gained about typical errors made by designers during the creation of a TLM description, we have identified the following classes of design errors:

1. deadlock in the communication phase;
2. forgetting to use communication primitives (e.g., missing to call `put()` for sending a request, before calling get() for retrieving the response);
3. misapplication of TLM operations (e.g., calling `write()` for reading data instead of `read()`);
4. misapplication of blocking/non-blocking primitives;
5. misapplication of timed/untimed primitives;
6. erroneous handling of request/response packets (e.g., failing to set or read the packet fields);
7. erroneous polling mechanism (e.g., infine loop).

Other design errors could be added to the previous list to expand the proposed mutation model without altering the methodology we have proposed to define it. Each of the previous error classes has been associated with at least a mutation of the EFSM models representing TLM communication primitives, as described in the next subsection. It is worth noting that we have not considered errors leading to problems during the code compilation.

4.2 Design errors vs. EFSM mutations

According to the classification of errors that may affect the specification of finite state machine, proposed by Chow [26], different fault models have been defined for perturbing FSMs [7,13]. They target, generally, boolean functions labeling the transitions, and/or transition's destination states. Mutated versions of an EFSM can be generated in a similar way, by modifying the behavior of enabling and update functions and/or changing the destination state of transitions.

Hereafter, we present how EFSMs of Figure 2 can be perturbed to generate mutant versions of the TLM primitives according to the design errors summarized in Section 4.1. We differentiate among mutations that (i) affect the destination state of transitions, (ii) change the truth value of the enabling functions, (iii) modified the operations performed in the update functions. Figure 5 shows how such kinds of mutations are used to affect the behavior of a representative for each class of primitives (respectively, a `write()`, a `put()` and a

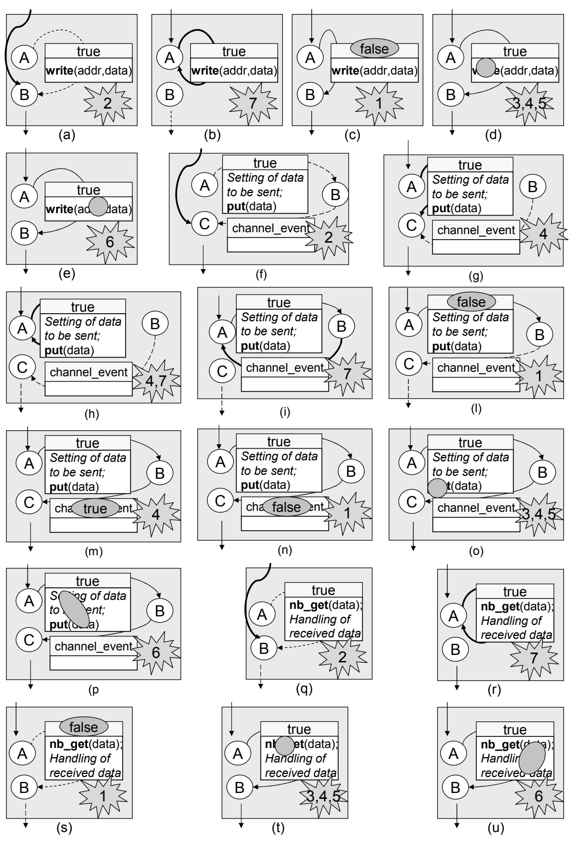


Figure 5: Mutations on EFSMs representing TLM 2.0 communication primitives.

nb_get() for bidirectional, unidirectional blocking and unidirectional non-blocking primitives). Numbers reported in the bottom-right part of each EFSM identify the kind of design errors modeled by the mutation w.r.t the classification of Section 4.1.

4.2.1 Mutations on destination states

Changing the destination state of a transition allows us to model design errors of type 2, 4 and 7 w.r.t the classification of Section 4.1. For example, let us consider Figure 5. Cases (a,b) show mutated versions of the EFSM corresponding to the bidirectional write() primitive that affect the destination state of transitions. Mutation (a) models the fact that the designer forgets to call write() (design error #2), while (b) models an erroneous polling mechanism (design error #7). Similar mutations have been defined for unidirectional blocking (cases (f,h)) and non-blocking primitives (cases (q,r)). However, in the case of unidirectional blocking primitives there exist two more mutations, i.e., cases (g,i) which model, respectively, the misapplication of a non-blocking primitive instead of a blocking one, since the wait on channel event is bypassed (design error #4), and a different way for representing an incorrect use of the polling mechanism w.r.t case (h) (design error #7).

4.2.2 Mutations on enabling functions

Changing the truth value of enabling functions allows us to model design errors of type 1 and 4 w.r.t the classification of Section 4.1. For example, let us consider Figure 5. Case (c) shows a mutated version of the EFSM corresponding to the write() primitive, where the transition from *A* to *B* is never fired and *B* is never reached. Such a mutation corresponds to a deadlock in the communication protocol (design error #1), due for example to a wrong implementation of the write() in the target module. Similar mutations have been defined for unidirectional blocking (cases (l,n)) and non-blocking primitives (case (s)). In these cases, the deadlock may be caused by an incorrect synchronization between get and put operations performed by communicating modules, such that the intermediate channel is always full (block on put operation) or always empty (block on get operation). Unidirectional blocking primitives can also be mutated as shown in case (m), which corresponds to using a non blocking primitive instead of a blocking one, since the wait in B for the channel event is prevented by an always-true enabling function (design error #4).

4.2.3 Mutations on update functions

Changing the operations performed in the update functions allows us to model design errors of type 3, 4, 5 and 6 w.r.t the classification of Section 4.1.

Two kinds of mutations are defined for the update functions: modification of the operation, and perturbation of data included in request or response packets. In the first case, shown in cases (d,o,t), respectively, for bidirectional, unidirectional blocking and unidirectional non-blocking primitives, the mutation correspond to a misapplication of the communication primitives, like, for example, calling a write() instead of a read() (design error #3), a put() instead of an nb_put() (design error #4), a timed primitive instead of an untimed one (design error #5). On the contrary, mutations to EFSMs for perturbing the exchanged data are shown in cases (e,p,u). They allow us to model design errors corresponding to an erroneous handling of request/response packets (design error #6).

5 Experimental results

Effectiveness of the mutation analysis based on the proposed mutation model has been evaluated by verifying the design examples released with the OSCI TLM 2.0 library [5]. In particular, six examples have been considered where different TLM communication protocols are implemented, i.e., bidirectional blocking untimed for *pv_example*, *byte_enable_single* and *byte_enable_block*, unidirectional blocking untimed for *example_3.3*, and unidirectional unblocking timed for *p2p_pipe_thread* and *bus_1m_3s*.

Mutated versions of SystemC TLM primitives have been implemented and included in a library linked to the considered benchmarks, instead of the original OSCI SystemC TLM 2.0 library. Then, a SystemC framework have been developed to perform the mutation analysis by using the test benches provided with the benchmarks (note that, such test benches have been modeled by people of the SystemC TLM working group unaware of our mutation model). The framework works according to the single mutation assumption, i.e., each test bench is simulated by activating a single mutation during each simulation run and the result is compared with the one obtained by running a reference simulation without activating mutants.

Design	P.	Mut. on dest. states (#)		Mut. on en. fun.(#)		Mut. on up. fun.(#)		Total mut. (#)		Coverage (%)							
										Design errors							Total
	(#)	app	det	app	det	app	det	app	det	1	2	3	4	5	6	7	
pv_example	7	14	14	7	7	49	34	70	55	100	100	100	na	na	58.3	100	78.6
example_3_3	2	8	7	6	5	21	15	35	27	100	100	100	50.0	na	44.4	100	77.1
p2p_pipe_thread	3	6	6	3	3	66	27	75	36	100	100	84.2	66.7	100	28.2	100	48.0
bus_1m_3s	9	18	18	9	9	198	84	225	111	100	100	78.9	33.3	50.0	33.3	100	49.3
byte_enab_single	5	10	10	5	5	155	38	168	53	100	100	100	na	na	20.0	100	31.5
byte_enab_block	4	8	8	4	4	124	48	136	60	100	100	100	na	na	37.9	100	44.1

Table 1: Experimental results.

The framework computes the mutation coverage and provides feedback to the designers by showing which mutations are covered/not covered by each test bench.

Table 1 shows the obtained results. Column *P. (#)* shows the number of instances of communication primitives used in the designs. Each primitive has been mutated according to the mutation model presented in Section 4.2. The number of applied (*app*) and detected (*det*) mutations on destination states, enabling functions and update functions is reported, respectively, in Columns *Mut. on dest. states (#)*, *Mut. on en. fun. (#)* and *Mut. on up. fun. (#)*, while Column *Total mut.(#)* shows the total. Columns *Coverage (%)* report the percentage of covered mutations corresponding to the seven categories of design errors classified in Section 4.1, and the total coverage. Note that, mutants corresponding to the design error category number 4 (i.e., misapplication of blocking/non-blocking primitives) are not applicable (*na*) for benchmarks using bidirectional primitives, since there are not non-blocking versions for such primitives. Moreover, mutants corresponding to the design error category number 5 (i.e., misapplication of timed/untimed primitives) are not applicable for benchmarks *pv_example*, *example_3_3*, *byte_enab_single* and *byte_enab_block* because they involve primitives that are not provided for considering time.

The achieved coverage proves that test benches released with the examples are not accurate enough to detect some possible design errors in the communication protocol. Most of the undetected mutations are related to design errors belonging to category number 6, i.e., erroneous handling of packets. This is due to the fact that the test benches do not set/read some fields inside the packet (e.g., priority of the request, transaction id, master thread id, etc.). Test benches that do not consider such fields may lead to a false sense of security, since they fail to check the correctness of the synchronization mechanism between masters and slaves. Moreover, for some benchmark the considered test benches fail to cover mutations corresponding to design errors in category 3, 4 and 5 too. Even if such errors are not often recurring, high-quality test benches should detect them to avoid error conditions in the communication phase.

6 Concluding remarks

In this paper, we have presented a mutation model for the new SystemC TLM 2.0 library. First, TLM communication primitives have been formalized by using the EFSM model. Then, mutations for perturbing the behavior of the resulting EFSMs have been defined. Such mutations have been put in relation to actual design errors, and they have been implemented in mutant versions of the TLM primitives to be used during mutation analysis of TLM SystemC descriptions. Experimental results have confirmed the effectiveness of such mutants, highlighting, for example, the inability of test benches provided with the reference examples of the SystemC TLM library in covering all aspects of the adopted communication protocols.

Future works will deal with the definition of mutants to address design errors affecting the delay parameter of TLM timed primitives.

References

[1] D. Hyunsook and G. Rothermel. *On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques*. IEEE Transaction on Software Engineering, pp. vol. 32, Issue 9, 733–752, 2006.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer*. IEEE Computer, pp. vol. 11, Issue 4, 34–41, 1978.

[3] R. A. DeMillo and A. J. Offutt. *Constraint-Based Automatic Test Data Generation*. IEEE Trans. Softw. Eng., vol. 17(9):pp. 900–910, 1991.

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

[5] *Open SystemC Initiative*. www.systemc.org.

[6] D. Lee and M. Yannakakis. *Online minimization of transition systems*. In *Proc. of ACM Symposium on the Theory of Computing*, pp. 264–274. 1992.

[7] K.-T. Cheng and J.-Y. Jou. *A Single-State-Transition Fault Model for Sequential Machines*. In *Proc. of IEEE ICCAD*, pp. 226–229. 1990.

[8] R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia. *Mutation of Java objects*. In *Proc. of IEEE ISSRE*, pp. 341–351. 2002.

[9] P. E. Black, V. Okun, and Y. Yesha. *Mutation operators for specifications*. In *Proc. of IEEE ASE*, pp. 81–88. 2000.

[10] J. S. Bradbury, J. R. Cordy, and J. Dingel. *Mutation Operators for Concurrent Java (J2SE 5.0)*. In *Proc. of IEEE ISSRE*, pp. 11–11. 2006.

[11] F. Belli, C. J. Budnik, and W. E. Wong. *Basic Operations for Generating Behavioral Mutants*. In *Proc. of IEEE ISSRE*, pp. 10–18. 2006.

[12] T. Olsson and P. Runeson. *System level mutation analysis applied to a state-based language*. In *Proc. of IEEE ECBS*, pp. 222–228. 2001.

[13] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. *Mutation analysis testing for finite state machines*. In *Proc. of IEEE ISSRE*, pp. 220–229. 1994.

[14] 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 ISSRE*, pp. 119–130. 2003.

[15] J. S. Bradbury, J. R. Cordy, and J. Dingel. *ExMan: A Generic and Customizable Framework for Experimental Mutation Analysis*. In *Proc. of IEEE ISSRE*, pp. 4–9. 2006.

[16] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. O. H. Pan, and E. H. Spafford. *The Mothra tool set (software testing)*. In *Proc. of IEEE HICSS*, pp. vol. 2, 275–284. 1989.

[17] S. S. Batth, E. R. Vieira, A. Cavalli, and M. Ü. Uyar. *Specification of Timed EFSM Fault Models in SDL*. In *Proc. of FORTE*, pp. 50–65. 2007.

[18] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. *Transaction Level Modeling in SystemC*, 2004. White paper. www.systemc.org.

[19] T. Kogel, T. Wieman, H. Keding, O. Fathy, and M. Burton. *OSCI TLM 2.0. PVT Modeling Examples*, 2007. www.systemc.org.

[20] J. A. Colgan and P. Hardee. *Advancing Transaction Level Modeling (TLM): Linking the OSCI and OCP-IP Worlds at Transaction Level*. In *White paper. http://www.opensystems-publishing.com/whitepapers*.

[21] N. Bombieri, F. Fummi, and G. Pravadelli. *EFSMs and Fault Modelling*. Deliverable D2.5 of VERTIGO European Project, University of Verona, Italy, 2007.

[22] 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. of IEEE CACSD*, pp. 522–527. 1999.

[23] 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. of ACM/IEEE DAC*, pp. 762–767. 2000.

[24] A. Zitouni, S. Badrouchi, and R. Tourki. *Communication Architecture Synthesis for Multi-bus SoC*. Journal of Computer Science, vol. 2(1):pp. 63–71, 2006.

[25] A. Guerrouat and H. Richter. *A component-based specification approach for embedded systems using FDTs*. ACM SIGSOFT Softw. Eng. Notes, vol. 31(2):pp. 14–18, 2006.

[26] T. Chow. *Testing software design modeled by finite state machines*. IEEE Trans. on Software Engineering, vol. 4(3):pp. 178–187, 1978.