Functional Qualification of TLM Verification

Nicola Bombieri Franco Fummi Graziano Pravadelli
Dipartimento di Informatica
Università di Verona, Italy
{firstname.lastname}@univr.it

Mark Hampton Florian Letombe
Certess
Moirans, France
{firstname.lastname}@certess.com

***Abstract*—The topic will cover the use of functional qualification for measuring the quality of functional verification of TLM models. Functional qualification is based on the theory of mutation analysis but considers a mutation to have been killed only if a testcase fails. A mutation model of TLM behaviors is proposed to qualify a verification environment based on both testcases and assertions. The presentation describes at first the theoretic aspects of this topic and then it focuses on its application to real cases by using actual EDA tools, thus showing advantages and limitations of the application of mutation analysis to TLM.**

I. Introduction

Mutation analysis and mutation testing [1] have definitely gained consensus during the last decades as being important techniques for software (SW) testing [2]. Such testing approaches rely on the creation of several versions of the program to be tested, "mutated" by introducing syntactically correct functional changes changes. The purpose of such mutations consists of perturbing the behaviour 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 hardware (HW) testing, when verification engineers (verifiers) use high-level fault simulation to measure the quality of test benches [3], 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 [3].

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 traditional mutation analysis the output of the DUV is compared with and without the mutation [4]. If there is a difference observed in the output then the mutant is considered to have been killed. Functional qualification (performed by the Certitude^{TM} tool [5]) introduced in this paper is different. It is based on the theory of mutation analysis but considers a mutation to have been killed only if a testcase fails. In the case where the verification environment models the expected behavior of the outputs then functional qualification can highlight missing checks. The checks could include the comparison of expected output behavior and assertions monitoring the program's internal or external behavior. This is a fundamentally different perspective because the ability of the verification to detect potential bugs is being measured whereas in traditional mutation analysis only the ability of the input sequences to propagate potential bugs to outputs is measured. Coverage metrics do not consider the checking of output behavior, therefore it was possible for these metrics to give high scores even if the output behavior of the DUV was not checked. Thus the term functional qualification has been introduced to capture this concept of measuring the bug detection ability.

Verification is required to ensure the quality of the design code and this activity often consumes around 70% [6] of the total design resources. A large amount of code must also be created to implement the verification environment. Errors in the verification environment can result in one of three situations:

- The testcase fails, in this case the error in the verification can be found.
- The testcase passes, in this case the testcase gives a false positive and may hide a real design bug.
- The testcase is missing, typically due to a mistake in the testplan.

Functional qualification is the first technology to indicate that a passing testcase is giving a false positive and also can identify a wide range of missing testcases that previous techniques can not detect, for example complex temporal sequences may be missing so potential bugs cannot propagate to outputs.

In this context, the paper presents the concept of functional qualification applied to TLM verification. Firstly, the paper extends the TLM mutation model presented in [7] in which a mutation model was proposed for the first draft of SystemC TLM (i.e., TLM-2.0 draft 1 - November 2006) to the last standard proposal (i.e., TLM-2.0 standard - June 2008). Then, the paper enhances the TLM fault model to be used for functional qualification in the context of a commercial tool like Certitude^{TM}. In particular, the paper presents the following innovative contributions:

- a way for formalizing the internal behavior of the TLM 2.0 communication primitives by using the extended finite

This work has been partially supported by the European project COCONUT FP7-2007-IST-1-217069.

978-3-9810801-5-5/DATE09 © 2009 EDAA

state machine (EFSM) model [8];
- a set of mutations for such EFSMs based on an extension of the well-known transition fault model for FSMs [9];
- identification of relations between the proposed mutations and typical design errors;
- a way to inject mutants into the C++ functionalities by using the Certitude^{TM} tool;
- an introduction to the notion of functional qualification and an experimental evaluation of it.

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

The paper is organized as follows. Section II presents the TLM mutation model for both the communication and functionality side of TLM designs. Section III presents Certitude^{TM}, a functional qualification framework. Section IV shows the effectiveness of the proposed mutations in measuring the quality of test suites. Finally, conclusions are discussed in Section V.

II. TLM Mutation Model

In this section we present a mutation model to perturb firstly the communication protocol and then the functionality side of TLM SystemC-based designs.

A. Mutation model for TLM communication protocols

The OSCI TLM-2.0 working group defines a set of interfaces (i.e., blocking, non-blocking, direct memory, and debug interfaces) for implementing transaction-level models and, then, describes a number of coding styles (i.e., loosely-timed, approximately-timed) that are appropriate for, but not locked to, various use cases such as SW performance analysis, architectural analysis, and HW verification [10]. The definition of the standard TLM-2.0 involves the TLM interfaces and the corresponding primitives, rather than the description of the coding styles.

We summarize, hereafter, the features of the main SystemC TLM-2.0 primitives, and for each primitive we 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 to perturb the communication interface of TLM designs according to typical design errors.

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, communication protocols, buses and controllers driving data-path [11], [12].

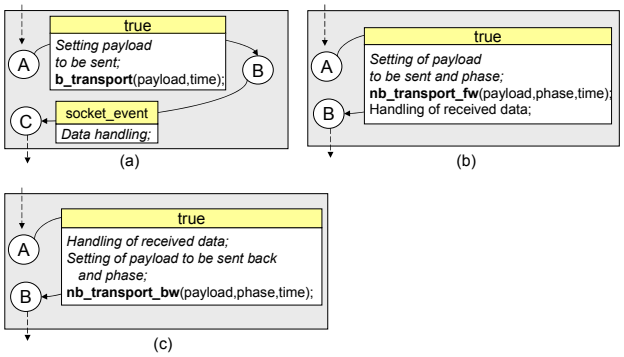

Fig. 1. EFSM models of some SystemC TLM-2.0 primitives.

2) Classification of TLM primitives: In TLM-2.0, communication is generally accomplished by exchanging packets containing data and control values, through a channel (e.g., a socket) between an initiator module (master) and a target module (slave). For the sake of simplicity and lack of space, we report in the follows the EFSMs representing the primitives and the proposed mutations of the only most relevant interfaces (i.e., blocking and non-blocking interfaces):

- *Blocking interface.* It allows a simplified coding style for models that complete a transaction in a single function call, by exploiting the blocking primitive `b_transport()`. The EFSM model of primitive `b_transport()` is composed of three states (see Figure 1.a). Once the initiator has called `b_transport()`, the EFSM moves from state *A* (initial state) to state *B* and it asks the socket channel to provide a payload packet to the target. Then, the primitive suspends in state *B* waiting for an event from the socket channel (*socket_event*) indicating that the packet can be retrieved. Finally, the retrieved data is handled by executing the operations included in the update function moving from *B* to the final state *C*. Timing annotation is performed by exploiting the *time* parameter in the primitives and managing the time information in the handling code of the received data for implementing, for example, the *loosely-timed* models.
- *Non-blocking interface.* Figures 1.b - 1.c show the EFSM models of the non-blocking primitives, which are composed of two states only. Primitives such as `nb_transport_fw()` and `nb_transport_bw()` perform the required operation as soon as they are called, and they immediately reach the final state in the corresponding EFSM. The caller process is informed if the non-blocking primitive succeeded by looking at its return value. Timing annotation is still performed by exploiting the *time* parameter in the primitives while parameter *phase* is exploited for implementing more accurate communication protocols, such as the four phases

approximately timed.

3) Mutation overview: 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. Starting from the EFSM models, we define the mutation model for the communication protocols, by following the strategy described in the follows:

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) 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 a `nb_transport_bw()` for completing transaction phases, before initiating a new transaction);
3) misapplication of TLM operations (e.g., setting a *write* command for reading data instead of *read*);
4) misapplication of blocking/non-blocking primitives;
5) misapplication of timed/untimed primitives;
6) erroneous handling of the generic payload (e.g., failing to set or read the packet fields);
7) erroneous polling mechanism (e.g., infinite loop).

Other design errors could be added to the previous list to expand the proposed mutation model without altering the methodology. Each of the previous error classes has been associated with at least a mutation of the EFSM models representing TLM primitives, as described in the next subsection.

5) Design errors vs. EFSM mutations: According to the classification of errors that may affect the specification of finite state machine, proposed by Chow [13], different fault models have been defined for perturbing FSMs [14], [9]. 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 an example of how the EFSM of Figure 1(a) can be perturbed to generate mutant versions of the TLM primitive according to the design errors summarized in Section II-A4. Figure 2 shows how such kinds of mutations are used to affect the behavior of primitive `b_transport()`. 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 II-A4.

Mutations on destination states. Changing the destination state of a transition allows us to model design errors of type 2,

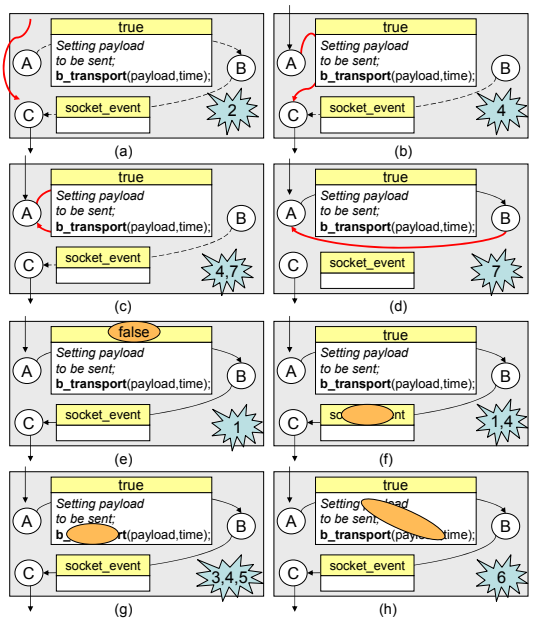

Fig. 2. Mutations on EFSM representing the TLM-2.0 primitive b_transport().

4 and 7 w.r.t the classification of Section II-A4. For example, let us consider Figure 2. Cases (a-d) show mutated versions of the EFSM that affect the destination state of the transition. Mutation (a) models the fact that the designer forgets to call `b_transport()` (design error #2), while (b) models the misapplication of a non-blocking primitive instead of a blocking one, since the wait on channel event is bypassed (design error #4). Cases (c) and (d) model two different incorrect uses of the polling mechanism (design error #7).

Mutations on enabling functions. Mutation on the truth value of enabling functions model design errors of type 1 and 4 w.r.t the classification of Section II-A4. For example, Figure 2(e) shows a mutated version of the EFSM corresponding to primitive `b_transport()`, 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 synchronization among modules with the socket channel. The primitive can also be mutated as shown in case (f), which corresponds to using a non-blocking instead of a blocking primitive, since the wait in `B` for the channel event is prevented by an always-true enabling function (design error #4).

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. Mutation on operations (shown in case (g)) corresponds to a misapplication of the communication primitives, like, for example, calling a transaction for writing instead of a transaction for reading (design error #3), a `b_transport()` instead of an `nb_transport()` (design error #4), setting the time parameter instead of don't setting it (design error #5). On the other hand, mutations on data included in the payload packets (shown in cases (h)) model

design errors corresponding to an erroneous handling of the payload packet (design error #6).

B. Mutation model for C++ functionalities

It is assumed that the functionality of the TLM model is a procedural style of code in one or more SystemC processes. Therefore, object oriented features of C++ have not yet been integrated into the mutation model for the functionality.

The mutation model is derived from work in [15] that defined mutation operators for the C language. Selective mutation (suggested by Mathur [16] and evaluated in [17]) is applied to ensure the number of mutations grows linearly with code size. A typical mutation density of about 1.4 mutants per line of code is achieved. Note that for the remaining of the paper, only valid mutants are introduced in designs:

Definition 1 (Valid mutant [18]): A mutant is considered valid if it is syntactically correct and inconsistent, such that a testcase can be extracted from it.

Moreover, those mutants are considered to be simples, i.e. not a composition of two or more other mutants.

For example, a mutation can be applied on a post-incrementation operator; this operator could for instance be mutated into a pre-incrementation operator as follows: `var1 = var2++;` would thus become `var1 = ++var2;`. This is a typical example of valid mutation. However, a mutation as `var1 = ;` is not valid: a compiler would throw a syntax error (e.g., gcc throws `expected expression before ';' token`).

The SystemC program is treated as a generic C++ program. The parsing of the C++ code is performed without macro expansion thus avoiding the introduction of mutants into individual instances of a macro. Currently the definition of each macro is not mutated.

III. Functional Qualification Framework

This Section describes the tool Certitude™, i.e. its approach w.r.t. functional qualification.

If there were a bug in your design,
could the verification environment find it?

Functional qualification is the first technology to provide an objective answer to this fundamental question. Functional qualification provides information about previously unknown verification weaknesses that when addressed can lead to significant improvements in verification quality. Large ICs require hierarchical verification [19] so improvements in the quality of lower level verification can result in significant productivity gains for the overall design cost. The core technology underlying functional qualification is mutation analysis [1]. Mutation analysis has been actively researched for over 30 years in the software testing community [20], [21] but Certess™ is the first company to provide a commercial tool (Certitude™ [5]) that uses this technology within the EDA space.

To be effective, verification must ensure that designs are shipped without critical bugs. To find a design bug, three

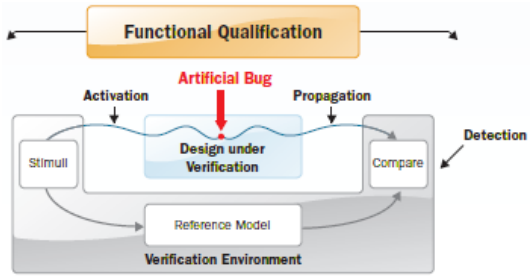

Fig. 3. Functional Qualification

things must occur during the execution of the verification environment:

1) The bug must be activated; i.e. the code containing the bug is exercised.
2) The bug must be propagated to an observable point; e.g. the outputs of the design.
3) The bug must be detected; i.e. behavior is checked and a failure indicated.

Traditional EDA technologies have focused on item 1, activating the bug. Techniques such as code coverage and functional coverage can help ensure that design code is well-activated, but they cannot guarantee that design bugs will be propagated. Nor can they guarantee that the bugs will be detected by the checkers, assertions or comparison against a reference model.

When integrating Certitude™ into a project environment, it is important to understand that it works on top of the simulation environment, and can make use of a batch interface into the environment. Certitude™ is a point tool that does not require changes to the project environment itself. However, slight modifications to some scripts may be necessary. To adapt Certitude™ to the project environment, it needs to have the following information and control:

- a list of all HDL files that make up the DUV,
- the ability to recompile the (instrumented) HDL source code,
- a list of testcase names,
- a script that can execute a testcase and return a pass or fail result.

Certitude™ automatically inserts mutants into the design and determines if the verification environment can detect these mutations (mutations can be thought of as artificial bugs, cf. Figure 3). A known mutant that cannot be detected points to a verification weakness. If a mutant cannot be detected, there is evidence that actual design bugs would also not be detected by the verification environment.

A functional qualification tool, such as Certitude™, helps the user understand the nature of these verification weaknesses. Functional qualification is able to provide new information to the verifier. For the first time, verifiers can measure the ability of their verification environments to propagate and check potential design bugs. Put more bluntly, Certitude™ is the first tool to measure the quality of their work comprehensively.

There are potentially a large number of live mutants so a principal concern is how much time it takes to analyze this information. Certitude™ can provide information to help in the analysis of live mutants for example graphical waveform diffs of signal behavior with and without a mutant. However the key to an efficient use is methodology, not technology.

If functional verification is seen as the measurement of the design's functionality then functional qualification can be seen as a calibration of the verification process. It then becomes clear that the verification activity should be driven from the verification plan - not from data resulting from it's calibration. From this perspective, the time spent in verification closure can be thought of as a direct measure of how good or bad the planning was.

When Certitude™ finds a live mutant the user is encouraged to find the root cause of this, then look for potentially related issues. For example a single live mutant may point out a missing checker, further analysis may point out a missing feature from the verification plan, still further analysis may point out a poorly written specification. With this in mind the user reviews similar sections of the specification checking if they resulted in an accurate verification plan. Once the root cause and related issues have been identified, from a single live mutant, there will often be considerable changes to the verification plan and then to the verification code. These changes can impact the status of many mutants. Therefore the user is encouraged not to analyze a large number of mutants (because many mutants may point to the same root cause) but to perform the verification improvement iteratively.

Because the user performs a root cause analysis, not all mutants need to be analyzed by the tool in each qualification iteration. This addresses the major concern of functional qualification - the runtime overhead. If only a subset of the faults need to be analyzed then the runtime overhead can be significantly reduced. Furthermore an ordering of the mutants is performed automatically in Certitude™ and maximize the efficiency of this methodology [22].

In the mutation analysis literature the coupling effect states that live mutants may be associated with more complex real design bugs [20]. Functional qualification expands on this relationship by introducing the intelligence and creativity of an engineer performing a root cause analysis of live mutants. Therefore the coupling between live mutants and design bugs during a Certitude™ qualification benefits from the multiplying effect of the user's insights into how to improve the verification plan and the overall verification process.

IV. Experimental Results

Effectiveness of the functional qualification based on the proposed mutation model has been evaluated by qualifying the design examples released with the OSCI TLM-2.0 library [23]. The first five examples have been released with the TLM-2.0 draft 1 library (2006), while the last two (i.e., lt and at) have been released with the TLM-2.0 final draft (2008).

Column *Communication protocol* of Table I shows the obtained results for the TLM communication protocol side. Column *P. (#)* shows the number of instances of TLM primitives used in the designs. Each primitive has been mutated according to the mutation model presented in Section II-A5. The number of applied (*app*) and detected (*det*) mutations on destination states, enabling functions and update functions is reported, respectively, in Columns *Mutations on dest. states (#)*, *Mutations on en. fun. (#)* and *Mutations on up. fun. (#)*. Columns *Killed (%)* report the percentage of killed mutations corresponding to the seven categories of design errors classified in Section II-A4. Columns *Total mutants(#)* and *FQ (%)* show, respectively, the total number of mutants applied to the designs and the functional qualification score, that is, the rate of mutants detected by the functional qualification process.

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., data length, byte enable, streaming width, 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 initiator and target. Moreover, for some benchmark the considered test benches fail to kill 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.

Column *Functionality* of Table I shows the obtained results for the TLM functionality side concerning the total number of mutants applied to the designs (column *Total mutants (#)*) and the functional qualification score (column *FQ (%)*).

The achieved results proves that test benches released with the examples are not accurate enough to detect many possible design errors in both the communication protocol and the functionality side of the designs.

Table II gives a comparison of Certitude™ with the code coverage tool *gcov* for the TLM functionality side of the designs. gcov is a tool to be used in conjunction with the gcc compiler to provide code coverage of a program. Column *Lines (#)* (resp. *Covered (%)*) is the number (resp. percentage) of statement lines occurring in (resp. covered by) the DUV provided by applying gcov to the designs. Regarding Certitude™, the number of mutants injected in the design and the functional qualification score are reported in columns *Mutants (#)* and *FQ (%)*, respectively.

Regarding Table II, these results clearly show that a good code coverage score does not necessarily imply a good functional qualification score (see for instance design *byte_enable*).

Finally, we highlight the FQ advantages w.r.t. mutation analysis (MA). MA compares the original design's outputs with a mutated design by considering `STDOUT`, `STDERR`, and generated cores. In the pvt examples, corresponding to rows p2p_pipe_thread and bus_1m_3s in Table II, the verification environment performs a diff of `STDOUT` with a golden reference file. However in these examples there is an assertion that prints to `STDERR`. In the case of MA, mutants causing the assertion to fire would be considered as killed but in FQ

Design	Communication protocol																	Functionality	
	P.	Mutants on dest. states (#)		Mutants on en. fun.(#)		Mutants on up. fun.(#)		Killed (%)							Total		Total	
								Design errors							Mutants	FQ	Mutants	FQ
	(#)	app	det	app	det	app	det	1	2	3	4	5	6	7	(#)	(%)	(#)	(%)
pv_example	7	14	14	7	7	49	34	100	100	100	na	na	58.3	100	70	78.61	345	40.29
p2p_pipe_thread	3	6	6	3	3	66	27	100	100	84.2	66.7	100	28.2	100	75	48.02	36	63.89
bus_1m_3s	9	18	18	9	9	198	84	100	100	78.9	33.3	50.0	33.3	100	225	49.36	36	55.56
byte_enab_single	5	10	10	5	5	155	38	100	100	100	na	na	20.0	100	168	31.56	174	29.31
byte_enab_block	4	8	8	4	4	124	48	100	100	100	na	na	37.9	100	136	44.11	174	36.21
lt	3	12	11	9	8	62	31	100	100	100	73.3	77.1	32.4	100	87	57.53	4	75
at	8	16	16	8	8	166	68	100	41.7	100	100	100	37.1	100	190	48.49	69	33.33

TABLE I
FUNCTIONAL QUALIFICATION OF TLM COMMUNICATION PROTOCOL AND FUNCTIONALITY

Design name	gcov		Certitude™	
	Lines (#)	Covered (%)	Mutants (#)	FQ (%)
pv_example	200	58	345	40.29
p2p_pipe_thread	41	92.68	36	63.89
bus_1m_3s	41	92.68	36	55.56
byte_enab_single	71	73.24	174	29.31
byte_enab_block	71	71.83	174	36.21
lt	17	100	4	75
at	99	54.55	69	33.33

TABLE II
CODE COVERAGE VS. FUNCTIONAL QUALIFICATION ON TLM FUNCTIONALITY

the mutants would not be killed. FQ highlights the problem in the verification environment where `STDERR` is not being checked.

V. CONCLUDING REMARKS

In this paper, we introduced the concept of functional qualification and we showed how it can be applied to the TLM context. We highlighted the advantages of functional qualification w.r.t. common verification techniques based on mutation analysis. We firstly presented a mutation model for the SystemC TLM-2.0 standard and, then, we proposed an enhancement of that model to be used for functional qualification in the context of a commercial tool like Certitude™. We reported a set of meaningful experimental results for showing the effectiveness of the proposed methodology.

Future work includes several activities. We are planing to integrate the SystemC TLM fault model into the Certitude™ product, and to extend the TLM mutation model for managing the temporal decoupling, typical characteristic of some standard primitives. Finally, an application of Certitude™ to embedded software is also envisioned.

REFERENCES

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

[2] D. Hyunsook and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transaction on Software Engineering*, vol. 32, no. 9, pp. 733–752, Sept. 2006.

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

[4] R. Guderlei, R. Just, C. Schneckenburger, and F. Schweiggert, "Benchmarking testing strategies with tools from mutation analysis," in *ICSTW'08*, 2008, pp. 360–364.

[5] "Certitude™ from Certess Inc.™," http://www.certess.com.

[6] J. Bergeron, *Writing Testbenchs: Functional Verification of HDL Models*. Kluwer Academic, 2000.

[7] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *Proc. of ACM/IEEE DATE*, 2008, pp. 396–401.

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

[9] K.-T. Cheng and J.-Y. Jou, "A single-state-transition fault model for sequential machines," in *IEEE ICCAD'90*, 1990, pp. 226–229.

[10] *OSCI TLM-2.0 User Manual*, http://www.systemc.org, June 2008.

[11] 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 *ACM/IEEE DAC'00*, 2000, pp. 762–767.

[12] A. Zitouni, S. Badrouchi, and R. Tourki, "Communication architecture synthesis for multi-bus soc," *Journal of Computer Science*, vol. 2, no. 1, pp. 63–71, 2006.

[13] T. Chow, "Testing software design modeled by finite state machines," *IEEE Trans. on Software Engineering*, vol. 4, no. 3, pp. 178–187, 1978.

[14] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *IEEE ISSRE'94*, 1994, pp. 220–229.

[15] M. E. Delamaro and J. C. Maldonado, "Proteum - A Tool for the Assessment of Test Adequacy for C Programs," in *PCS'96*, 1996, pp. 79–95.

[16] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *COMPSAC'91*, 1991, pp. 604 – 605.

[17] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *ICSE'93*, 1993, pp. 100–107.

[18] G. Fraser and F. Wotawa, "Using Model-Checkers for Mutation-Based Test-Case Generation, Coverage Analysis and Specification Analysis," in *ICSEA'06*, 2006, pp. 16–22.

[19] A. B. van der Wal, R. G. J. Arendsen, O. E. Herrmann, and A. C. Brombacher, "Hierarchical statistical verification of large full custom CMOS circuits," *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 443–446, 1994.

[20] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven CT, 1980.

[21] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[22] M. Hampton, "Procédé et système d'évaluation de tests d'un programme d'ordinateur par analyse de mutations," French Patent FR2873832 for Certess SARL, 2006.

[23] *OSCI TLM-2.0 standard*, http://www.systemc.org.