On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL*

Nicola Bombieri Franco Fummi Graziano Pravadelli
Dipartimento di Informatica - Università di Verona
{bombieri, fummi, pravadelli}@sci.univr.it

Abstract

Transaction level modeling (TLM) is becoming an usual practice for simplifying system-level design and architecture exploration. It allows the designers to focus on the functionality of the design, while abstracting away implementation details that will be added at lower abstraction levels. However, moving from transaction level to RTL requires to redefine TLM testbenches and assertions. Such a wasteful and error prone conversion can be avoided by adopting transactor-based verification (TBV). Many recent works adopt this strategy to propose verification methodologies that allow (1) mixing TLM and RTL components, and (2) reusing TLM assertions and testbenches at RTL. Even if practical advantages of such an approach are evident, there are no papers in the literature that evaluate the effectiveness of the TBV compared to a more traditional RTL verification strategy. This paper is intended to fill in the gap. It theoretically compares the quality of the TBV towards the rewriting of assertions and testbenches at RTL with respect to both fault coverage and assertion coverage.

1. Introduction

The design of SoCs, based on the integration of RTL components, is more and more complex, due to deployment of objects like multi-processor platforms with embedded memories and third party Intellectual Property (IP) blocks [1]. RTL design is simply not suitable for designing a multi-processor system with complex protocols. Its relatively low level of abstraction makes tedious, time consuming, and error prone its use in high-level system activities such as IP integration and architecture evaluation. For this reason, the emerging transaction level modeling is gaining consensus more and more [2].

In a transaction level model, the system is designed and verified in terms of functionality characterized by high-level I/O events and data transfers between computational blocks. The communication, which is separated from the computation, is modeled by channels that provide high-level communication primitives towards the computation components. On the contrary, implementation details related to timing, algorithm optimization, communication protocol, etc., are hidden and may be added at lower levels of abstraction. Such a kind of transaction level modeling is motivated by a number of practical advantages. These include:

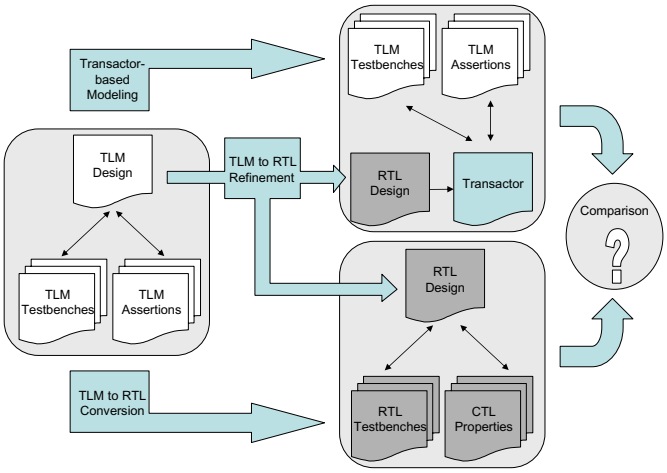

Figure 1. Transactor-based modeling vs. RTL modeling.

- implementation details are abstracted while preserving the behavioral aspects of the system; this allows a faster simulation (up to 1,000x) than at RTL;
- system level design exploration and verification are simplified, since IP components and buses can be modified and replaced in an easier way than at RTL;
- an early platform for SW development can be quickly developed;
- deterministic test generation is more effective and less tedious than at RTL, since tests are written without taking care of the communication protocol between components.

In this context, the OSCI TLM library [3] based on SystemC represents a valuable set of templates and implementation rules aiming at standardizing the different TLM methodologies that have been recently proposed [4, 5, 6, 7]. In fact, in spite of its name, transaction level does not denote a single level of description. Rather, it refers to a group of three abstraction levels: level 1 (the lowest), level 2, and level 3 (the highest), each varying in the degree of expressible functional and temporal details [8].

However, modeling a complex system completely at TLM could be inconvenient when already existent IP cores are reused. In fact, many vendors provide them at RTL. Thus, the concept of transactor has been proposed to allow a TLM-RTL mixed simulation [4, 5, 8, 9]. Whatever the adopted transaction level is (1, 2 or 3), a transactor works as a translator from a TLM function call, to an RTL se-

*The authors would like to thank the STMicroelectronics Verification Group of Agrate for the valuable support provided in assertion/property definition.

3-9810801-0-6/DATE06 © 2006 EDAA

quence of statements, i.e., it provides the mapping between transaction-level requests, made by TLM components, and detailed signal-level protocols on the interface of RTL IPs.

Besides the capability of simulating TLM-RTL mixed design, the use of transactors provides also valuable advantages from the verification point of view. Functional verification based on testbench generation [10, 11] and assertion-based verification (ABV) [6, 7] (using, for example, a property specification language like PSL [12]) represent the main verification techniques at TLM. The adoption of a transactor-based design methodology allows an easy reuse of TLM testbenches [4, 5] and TLM assertions [8] at lower levels of abstraction. This avoids a wasteful and error-prone conversion of both TLM assertions into RTL properties, and TLM testbenches into RTL ones.

Even if transactor-based verification is increasingly used, to the best of our knowledge, there are no works in the literature which evaluate the effectiveness of the TBV with respect to a fully RTL verification. In this paper, we provide a theoretical and practical comparison showing that TBV (upper side of Figure 1) is at least as efficient as a fully RTL verification methodology which requires to convert TLM assertions into RTL properties and to create new RTL testbenches (lower side of Figure 1). Such a comparison is twofold and it relies on the use of a high-level fault model as a common metrics to estimated coverage of both testbenches and assertions/properties (see Section 2). Given an RTL description, we show that:

- in case of fault simulation, the reuse of TLM testbenches through TBV allows to detect the same set of faults detectable by RTL testbenches (see Section 3);
- in case of ABV, checking TLM assertion through TBV allows to verify the same set of behaviors covered by model checking the RTL design, once the assertions have been refined into temporal properties to reflect the interface and timing differences between the TLM and the RTL models (see Section 4).

The theoretical results are confirmed by experimental results reported in Section 5.

2. Evaluation Methodology

Figure 2 shows the proposed evaluation methodology. The TLM design is refined in an equivalent RTL module by following a standard semi-automatic TLM design flow. Then, in the upper side, the RTL module is embedded in the transactor-based verification architecture, where it interacts with TLM testbenches and TLM assertions through the transactors. In this way, both the simulation engine and the ABV infrastructure are unchanged moving from TLM to RTL. On the contrary, in the lower side of the Figure, the RTL module communicates directly with new RTL testbenches. Moreover, a model checker is used to verify the CTL properties derived from the TLM assertions. The proposed evaluation methodology compares the two alternatives by measuring the effectiveness of fault simulation and assertion/property checking.

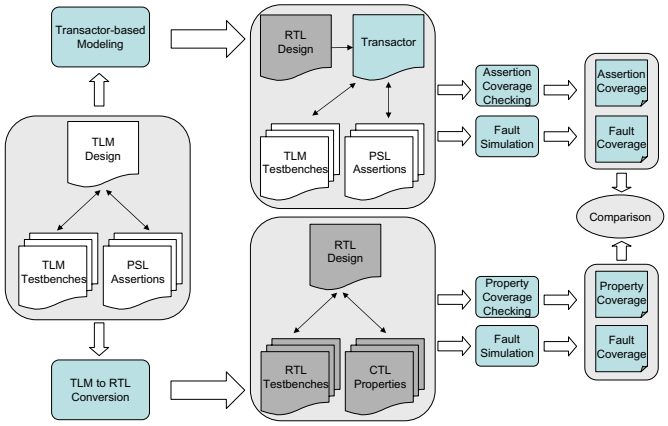

Figure 2. Evaluation methodology flow.

Testbench generation, based on (fault) coverage metrics, is the core of dynamic verification, the main verification technique at RTL [13]. Given an high-level fault model, testbench effectiveness is measured as fault coverage, i.e., the percentage of faults detected by the testbenches with respect to the total number of modeled faults. This allows first to measure the quality of testbenches, and secondly to identify possible design errors by analyzing the nature of undetectable faults [14]. In the following we show that, independently from the adopted high-level fault model, the fault coverage achieved by a set of RTL testbenches is comparable with the one achieved by reusing the TLM testbenches through the transactor.

Static verification based on model checking [15] represents, whenever applicable, a valuable alternative to dynamic verification. It formally verifies the correctness of the design by checking if it satisfies temporal properties (generally expressed by using CTL or LTL) derived from the specification. Unfortunately, model checking cannot be applied on TLM descriptions, since model checkers require a synthesizable model to represent the Kripke structure they use to accomplish the property verification. Thus, at TLM, ABV represents the natural alternative to model checking. Assertions are defined, instead of CTL or LTL properties, by using, for example, the property specification language proposed by Accellera. Assertions are monitored during the simulation of the design providing an immediate check of failures.

In the following we show that, at RTL, the set of design behaviors monitored by model checking CTL properties, note as property coverage, is comparable with the assertion coverage, i.e., the set of behaviors monitored by reusing the corresponding TLM assertions through the transactors. Such a comparison between property and assertion coverage is performed by exploiting a fault simulation based technique that relies on the same high-level fault model adopted to compare the fault coverage [16].

The proposed twofold evaluation, based on fault coverage and assertion/property coverage, shows that the transactor-based verification is not only valuable for time savings and conversion error avoidance, but also because it is at least as effective as RTL verification.

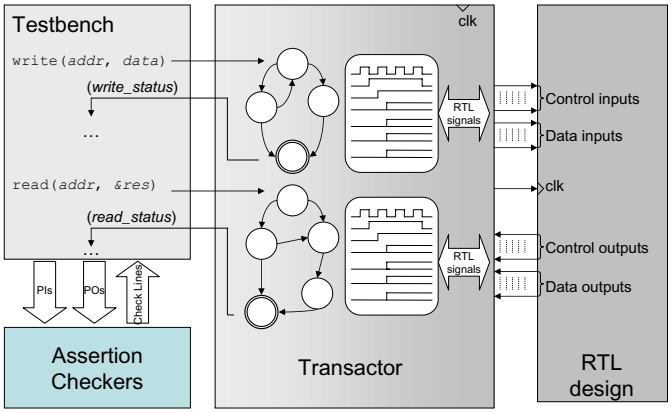

Figure 3. The role of the transactor.

2.1 Transactor Definition

Figure 3 shows how the transactor is exploited to reuse TLM testbenches and assertions on the RTL design [4, 5]. The testbench carries out one transaction at time, composed by two TLM function calls (`write()` and `read()`). First, data are provided to the RTL design by means of `write(addr, data)`. The transactor converts the `write()` call to the RTL protocol-dependent sequence of signals required to drive control and data inputs of the design under verification (DUV). Moreover, the write status is reported to the testbench to notify about successes or errors. Then, the testbench asks for the DUV result by calling `read(addr, &res)`. The transactor waits until the DUV result is ready by monitoring the output control ports, and, finally, it gets the output data. Then, testbench can carry on with the next transaction. If assertion checking is desired, the parameter of the function calls (`addr, data, write_status, &res, read_status`), which represent inputs and outputs of the RTL computation, are provided to the assertion checkers. The testbench is modeled at transaction level, thus, assertions are checked when `write()` and/or `read()` return according to the aim of assertions.

2.2 High-Level Fault Model

Before describing the evaluation methodology in details, it is necessary to explain what motivations lie under the adoption of an high-level fault model to compare TBV vs. RTL verification.

Both ABV and functional verification based on testbench simulation are necessarily incomplete, since it is not computationally feasible to exhaustively simulate sequential designs. It is important, therefore, to quantitatively measure the quality of testbenches used during the verification. Traditional coverage metrics derived from SW testing (e.g., statement, branch, path coverage) represent a low cost popular solution [17]. However, they are based on controllability information, i.e., the activation of statements, branches or sequences of statements, and they do not address observability requirements, i.e., to see whether effects of possible errors activated by tests can be observed at the DUV outputs. The fact that a statement with a bug has been activated by input stimuli does not mean that the observed outputs will be incorrect.

An alternative approach is represented by the use of high-level fault models [18], which include the characteristics of both coverage metrics and logic-level fault models [19]. Independently from its typical implementation (perturbed assignment, operator substitution, mutants, saboteurs, etc.), an high-level fault provides an abstraction of a possible design error, since it produces perturbed DUV behaviors. Thus, the analysis of its nature allows an effective verification of the expected and unexpected behavior of the DUV, particularly when faults are directly injected into RTL code, which is very familiar to the designer. In such a way, the fault coverage is used as a metrics to evaluate the quality of testbenches as well as to reveal design errors. Achieving 100% fault coverage is generally harder than 100% statement or branch coverage. Then, stimuli generators targeted to fault coverage allow a wider exploration of the DUV state space. Thus, they provide better test cases with respect to ones obtained by using traditional coverage metrics. For this reason, given an high-level fault model, the fault coverage represents a good parameter to measure the effectiveness of TBV with respect to RTL test generation.

Moreover, according to the methodology proposed in [16], an high-level fault model can be used also to evaluate the quality of the model checking process by computing the property coverage. This measures the capability of properties to identify high-level faults that perturb the original functionality of the design implementation. The presence of a detectable high-level fault implies that the behavior of the perturbed implementation differs from the behavior of the unperturbed implementation. Thus, while the set of properties is satisfied by the original unperturbed implementation, at least one of them should be refuted if checked on the perturbed implementation. The property coverage is measured as the percentage of high-level faults that causes at least a property failure. Extending this strategy to assertions, as reported in Section 4, allows us to compare TBV assertion coverage to RTL property coverage.

Independently from the adopted high-level fault model, not all faults must be considered during test generation or assertion/property coverage evaluation, but only those detectable under the constrains which model the environment where the DUV will be embedded in. Environment constraints model restrictions which must be fulfilled by input sequences. In fact, when the design is embedded in a real application, its input signals can be driven by others modules which produce only a subset of all possible input sequences. Thus, the environment is modeled as a deterministic FSM which interacts with the FSM of the DUV.

Under this assumption, a classification of faults is needed. It is based on well known testing concepts reported, for example, in [20].

Definition 1 *Given the implementation, $\mathcal{I}$, of the DUV, a set of faults, $\mathcal{F} = \{f_1, \ldots, f_n\}$, a set of perturbed implementations, $\mathcal{I}_{\mathcal{F}} = \{\mathcal{I}_f | f \in \mathcal{F}\}$, the environment, $\mathcal{E}$, where $\mathcal{I}$ is embedded, and the set of FSM retroactive networks*

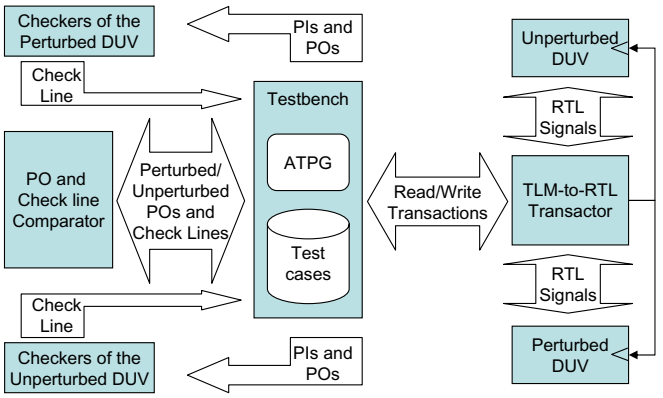

Figure 4. Simulation engine.

originated by $\mathcal{E}$, $\mathcal{N}_{\mathcal{IE}} \cup \mathcal{N}_{\mathcal{I}_\mathcal{F}\mathcal{E}}$ [1], where $\mathcal{N}_{\mathcal{I}_\mathcal{F}\mathcal{E}} = \{\mathcal{N}_{\mathcal{I}_f\mathcal{E}} | f \in \mathcal{I}_\mathcal{F}\}$, a fault $f \in \mathcal{F}$ is:

Detectable *if there is at least an input sequence, $\iota = (i_1, \ldots, i_n)$, such that at least one output of $\mathcal{I}$ differs from the respective output of $\mathcal{I}_f$ when ι is simultaneously applied to $\mathcal{I}$ and $\mathcal{I}_f$. We say that ι is a test sequence for f on $\mathcal{I}$.*

$\mathcal{E}$-detectable *if there is at least an input sequence $\iota = (i_1, \ldots, i_n)$, such that at least one output of $\mathcal{N}_{\mathcal{IE}}$ differs from the respective output of $\mathcal{N}_{\mathcal{I}_f\mathcal{E}}$ when ι is simultaneously applied to $\mathcal{N}_{\mathcal{IE}}$ and $\mathcal{N}_{\mathcal{I}_f\mathcal{E}}$. We say that ι is a test sequence for f on $\mathcal{N}_{\mathcal{IE}}$ [2]. We call $\mathcal{E}$-det the set of $\mathcal{E}$-detectable faults.*

The proposed evaluation methodology is independent from the adopted high-level fault model, which is only responsible to generate perturbed implementations according to the previous definition. However, different high-level fault models can provide different coverages. Thus, more the high-level fault model is suitable to detect design errors, more the application of the methodology is accurate. We use the *bit coverage* high-level fault model since it has been proved to be related to design errors [21].

2.3 Simulation Engine

Figure 4 shows the architecture of the simulation engine that has been implemented to accomplish the evaluation methodology previously summarized. The Figure refers to the TBV case, while the RTL implementation directly connects the testbench to the perturbed and unperturbed RTL DUVs without passing through the transactor.

The testbench either acts as a random automatic test pattern generator (ATPG) or it simply loads test cases from file. Then, it controls the TLM-to-RTL transactor (if it is present) which drives test sequences into the perturbed and the unperturbed RTL implementations of the DUV. Assertions/properties are converted in checkers by using FoCs [22]. It takes an assertion/property and it generates a state machine that monitors a set of signals to notify assertion/property failures. The comparison of primary outputs (POs) and check lines are used for computing respectively the fault coverage and the assertion/property coverage.

[1] An FSM retroactive network $\mathcal{N}_{\mathcal{IE}}$ is composed of two FSMs: $\mathcal{I}$, which describes the DUV, and $\mathcal{E}$, which models the environment where $\mathcal{I}$ is embedded. Some output lines of $\mathcal{I}$ are connected to the input lines of $\mathcal{E}$, and some output lines of $\mathcal{E}$ are connected to input lines of $\mathcal{I}$.

[2] Note that ι is also a test sequence for f on $\mathcal{I}$.

3. Testbench Reuse

We consider a general high-level fault model to theoretically show that reusing TLM testbenches by means of a transactor allows to detect the same set of faults detectable by applying a testbench directly to the RTL model. This conjecture relies on the following definitions and theorem.

Definition 2 *Under the same conditions of Def. 1 and assuming that the implementation $\mathcal{I}$ is modeled at transaction level, a fault is* ***TLM-detectable*** *if there is a test vector such that the outputs of the unperturbed and perturbed DUVs differ when the test vector is simultaneously applied to both the designs. The fault is* ***TLM-undetectable*** *if such a test vector does not exist.*

Definition 3 *Under the same conditions of Def. 1 and assuming that the implementation $\mathcal{I}$ is modeled at RTL, a fault is* ***RTL-detectable*** *if there is a test sequence such that the outputs of the unperturbed and perturbed DUVs differ when the test sequence is simultaneously applied to both the designs. The fault is* ***RTL-undetectable*** *if such a test sequence does not exist.*

It is worth to note that, at TLM, testbenches are composed of test vectors, while, at RTL, we need test sequences generally composed of more than one test vector. This is due to the fact that TLM is untimed (eventually a clock can be introduced at level 1), thus the result of a transaction is instantaneously available once a single test vector is applied. On the contrary, at RTL the design is generally modeled as an FSMD where the result is available after a number of clock cycles and it may depends on values provided to the primary inputs at different times. When a TLM testbench is applied to an RTL design, the transactor converts test vectors in the corresponding test sequences modeling the communication protocol needed by the RTL design. From this observation the following definition derives.

Definition 4 *Under the same conditions of Def. 1 and assuming that the implementation $\mathcal{I}$ is modeled at RTL and wrapped by a TLM-to-RTL transactor as defined in Section 2.1*[3]*, a fault is:*

Functionally T-detectable*, if there is a test vector such that the outputs of the transactor connected to the unperturbed and perturbed DUVs are available at the same time and they differ, once the transactor has simultaneously applied the test sequence derived from the test vector to both the designs.*

Timing T-detectable*, if there is a test vector such that the outputs of the transactor connected to the unperturbed and perturbed DUVs are available at different times, but they are equal, once the transactor has simultaneously applied the test sequence derived from the test vector to both the designs.*

Functionally and timing T-detectable*, if there is a test vector such that the outputs of the transactor connected to the unperturbed and perturbed DUVs are available at different*

[3] Please, note that in this case the environment constraints are directly modeled by the transactor.

times, and they differ, once the transactor has simultaneously applied the test sequence derived from the test vector to both the designs.

***T-detectable**, if it is functionally T-detectable and/or timing T-detectable.*

***T-undetectable**, if for each test vector the outputs of the transactor connected to the unperturbed and perturbed designs are available at the same time and they are equal, once the transactor has simultaneously applied the test sequence derived from the test vector to both the designs.*

Theorem 1 *An RTL-detectable fault is also T-detectable.*

Proof: If a fault is RTL-detectable there exist a test sequence such that it propagates the effect of the fault to at least one output of the perturbed RTL DUV (Def. 3). Note that, such a test sequence respects the protocol imposed by the environment constraints that must be connected to the RTL DUV as required by Def. 1.

Let us consider that the fault is propagated to a data output. Accordingly to the transactor implementation described in Section 2.1 and Def. 4, such a fault becomes functionally T-detectable (then, T-detectable) when the RTL design is connected to a TLM testbench through the transactor. In fact, the same test sequence generated by the RTL testbench can be obtained by applying an opportune test vector to the transactor (which acts like the environment constraints).

On the contrary, let us consider that the fault is propagated to a control output. In this case, the communication protocol between the transactor and the perturbed RTL DUV is necessarily changed. This causes that: the result of the perturbed DUV is provided to the transactor with a timing discrepancy with respect to the unperturbed DUV. Thus, according to Def. 4, the fault is timing T-detectable (then, T-detectable). □

Theorem 1 shows that the TBV is at least as effective as the RTL verification from the fault coverage point of view. The Theorem assumes that the TLM testbenches are able to produce a set of test vectors that can be converted from the transactor into a set of test sequences which includes the test sequences directly generated by the RTL testbenches. However, such an assumption is reasonable, since it is much more difficult to create efficient RTL testbenches than TLM ones [4].

4. Assertion Reuse

Let us compare now assertion coverage and property coverage to show the effectiveness of reusing TLM assertions at RTL through TBV, instead of converting assertions into CTL properties specifically tailored to the RTL design. The desired goal is to show that TLM assertions cover the same set of behaviors covered by the corresponding CTL properties.

The assertion/property coverage measures the quality of assertions/properties to detect design errors in all parts of the DUV description [16]. It is computed by analyzing the capability of assertions/properties to highlight differences between the unperturbed and the perturbed implementations of the same design. If an assertion/property, which holds on the unperturbed design, fails in presence of a fault, then the behavior perturbed by the fault is covered by the property/assertion.

Definition 5 *Under the same assumption of Definition 1, the property coverage, C_P, and the assertion coverage, C_A, are defined as:*

$$C_P = \frac{\#\ \textit{of faults that causes a property failure}}{\#\ \textit{of RTL-detectable faults}} \quad (1)$$

$$C_A = \frac{\#\ \textit{of faults that causes an assertion failure}}{\#\ \textit{of T-detectable faults}} \quad (2)$$

Given the previous Definition and Theorem 1, the simulation engine of Figure 4 is used also to compare TLM assertion coverage and RTL property coverage. In fact, the methodology presented in [16] for property coverage can be reused for assertion coverage, provided that, the definition of RTL-detectable faults is substituted with the definition of T-detectable faults.

For sake of completeness, it is worth to note that a set of assertions, achieving 100% assertion coverage on the TLM design, may not achieve 100% on the RTL design. This is due to the fact that at TLM no assertions can be defined related to communication protocols and timing between events, since such details are not modeled at transaction level. However, this observation does not affect the effectiveness of reusing TLM assertions at RTL, since they allows to check the functionality of the RTL design. Then, new RTL properties must be added only to verify timing and communication protocols.

5. Experimental Results

Experimental results have been conducted by using three components (*Root*, *Div* and *Dist*) of a real industrial SoC implementing a face recognition system provided by STMicroelectronics. The modules are composed of, respectively, 7802, 11637, 40663 gates, and 155, 269, 100 flip-flops.

5.1 Fault Coverage Comparison

Table 1 reports the results related to the fault coverage comparison. Column $\#Faults$ reports the total number of modeled high-level faults. Columns $FC\%$, $\#TV$ and $Time$ show, respectively, the achieved fault coverage, the number of test vectors generated by the testbench, and the time required to generate/simulate such vectors, for the TBV and the RTL verification flow. In particular, *TBV (reuse)* is related to the reuse of TLM testbenches at RTL via transactor, while *TBV (reuse+ATPG)* is related to the integration of the TLM testbenches by adding new test vectors generated by applying an ATPG to the RTL module via transactor. As expected from theoretical results, the TBV fault coverage is greater than or equal to the RTL one for all the modules. Moreover, it is interesting to note that

		TBV (reuse)			TBV (reuse+ATPG)			RTL		
Design	#Faults	FC%	#TV	Time (s.)	FC%	#TV	Time (s.)	FC%	#TV	Time (s.)
Root	1627	94.5	23	23	98.5	25	145	97.8	741	2050
Div	2333	84.3	20	31	97.0	191	112	86.6	1073	1774
Dist	3061	90.7	8	57	98.9	15	316	94.2	254	11540

Table 1. Fault coverage comparison.

the number of test vectors and the time required by TBV is lower than the corresponding RTL quantities. In particular, the time spent by TBV is extremely lower than the one needed at RTL. This derives from the fact that, at RTL, test vectors must be completely generated ex-novo. On the contrary, TBV can reuse the ones generated during the verification of the TLM descriptions. Thus, fault simulation is required instead of ex-novo test generation when TBV is applied. Indeed, the reuse of high-quality TLM test vectors could be insufficient to achieve an high-fault coverage also on the RTL design (*TBV (reuse)*). For this reason, the TLM testbenches have been integrated by generating new test vectors achieving the fault coverage reported in *TBV (reuse+ATPG)*. Thus, the TBV time is composed by summing the time required for fault simulation and the time required for testbench integration.

5.2 Property/Assertion Coverage Comparison

Regarding the comparison between assertion coverage and property coverage, experimental confirmation is provided only for the *Root* module, since no assertions were available for other designs at the moment of paper submission.

Five assertions have been defined to check the functionality of the TLM description *Root*. They achieved 100% assertion coverage, showing that they are enough to verify the correctness of the TLM description. After RTL refinement, the same assertions have been checked on the RTL implementation by using TBV. Then, they have been converted into CTL properties and verified by using the SMV model checker. Finally, assertion coverage via TBV, and property coverage have been computed and compared on the RTL implementation. The first achieves 95.1%, while the second 95.9% confirming the effectiveness of TBV.

As observed in Section 4, assertion/property coverage on the RTL design does not achieve 100%, even if the same assertions do it at TLM. This emphasizes the fact that TLM assertions, but also the corresponding RTL properties, are not able to identify perturbations that affect behaviors depending on timing synchronization or communication protocols. This has been confirmed by analyzing the nature of high-level faults not covered by the assertions/properties.

6. Concluding Remarks

The paper presented a theoretically-based methodology to evaluate the quality of TBV with respect to a more traditional RTL verification flow. The evaluation relies on comparing both fault coverage and assertion/property coverage by using and not using TBV to verify the correctness of an RTL design. In this way, we showed that TBV is effective for reusing testbenches as well as assertions when a TLM description is refined into an RTL implementation. The reported experimental analysis confirmed the expected results.

Future works will be related to the analysis of which typology of assertions cannot be expressed at TLM, but must be included during RTL verification to achieve 100% coverage. Moreover, we are working to compare fault coverage between TLM and RTL.

References

[1] J. Krasner. *Embedded Software Development Issues and Challenges*. Embedded Market Forecaster, 2003.

[2] L. Cai and D. Gajski. *Transaction Level Modeling: An Overview*. In *IEEE CODES + ISSS*, pp. 19–24. 2003.

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

[4] D. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. Norris Ip, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting. *The Transaction-Based Verification Methodology*. Tech. Rep. CDNL-TR-2000-0825, Cadence Berkeley Labs, 2000.

[5] C. Norris Ip and S. Swan. *A Tutorial Introduction on the New SystemC Verification Standard*, 2003. White paper. www.systemc.org.

[6] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib. *Combining System Level Modeling with Assertion Based Verification*. In *IEEE ISQED*, pp. 310–315. 2005.

[7] A. Habibi and S. Tahar. *Design for Verification of SystemC Transaction Level Models*. In *IEEE DATE*, pp. 560–565. 2005.

[8] N. Bombieri, A. Fedeli, and F. Fummi. *On PSL Properties Re-use in SoC Design Flow Based on Transactional Level Modeling*. In *IEEE MTV*. 2005.

[9] R. Jindal and K. Jain. *Verification of Transaction-Level SystemC Models Using RTL Testbenches*. In *ACM/IEEE MEMOCODE*, pp. 199–203. 2003.

[10] K. Ara and K. Suzuki. *A Proposal for Transaction-Level Verification with Component Wrapper Language*. In *IEEE DATE*, pp. 82–87. 2003.

[11] Z.-H. Wang and Y.-Z. Ye. *The Improvement for Transaction Level Verification Functional Coverage*. In *IEEE ISCAS*, pp. 5850–5853. 2005.

[12] Accellera. *Property Specification Language Reference Manual*, 2004.

[13] *The Medea+ Design Automation Roadmap*, 2002.

[14] F. Ferrandi, F. Fummi, G. Pravadelli, and D. Sciuto. *Identification of Design Erros through Functional Testing*. IEEE Trans. on Reliability, vol. 52(4):pp. 400–412, 2003.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[16] F. Fummi, G. Pravadelli, and F. Toto. *Coverage of Formal Properties based on a High-Level Fault Model and Functional ATPG*. In *IEEE ETS*, pp. 162–167. 2005.

[17] G. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.

[18] S. Ghosh and T. Chakraborty. *On Behavior Fault Modeling for Digital Designs*. International Journal of Electronic Testing: Theory and Applications, vol. 2(2):pp. 135–151, 1991.

[19] M. Breuer, M. Abramovici, and A. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[20] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[21] M. B. Santos, F. M. Gonalves, I. C. Teixeira, and J. P. Teixeira. *RTL-based Functional Test Generation for High Defects Coverage in Digital SoCs*. In *Proceedings of ETW 2000*, pp. 99–104. 2000.

[22] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. *FoCs - Automatic Generation of Simulation Checkers from Formal Specifications*. In *CAV)*, vol. 1855 of *Lecture Notes in Computer Science*, pp. 538–542. Springer-Verlag, 2000.