Incremental ABV for Functional Validation of TL-to-RTL Design Refinement *

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

Abstract

Transaction-level modeling (TLM) has been proposed as the leading strategy to address the always increasing complexity of digital systems. However, its introduction arouses a new challenge for designers and verification engineers, since there are no mature tools to automatically synthesize an RTL implementation from a transaction-level (TL) design, thus manual refinements are mandatory. In this context, the paper presents an incremental assertion-based verification (ABV) methodology to check the correctness of the TL-to-RTL refinement. The methodology relies on reusing assertions and already checked code, and it is guided by an assertion coverage metrics.

1. Introduction

EDA researchers are proposing to modify the design and verification flow of embedded systems in the following directions, to deal with always increasing complexity [1]:

- *raising the abstraction level* to simplify system-level design, architecture exploration and functional verification;
- *joining static and dynamic verification* to provide an easier and more powerful way to verify complex systems;
- *exploiting a reuse-based methodology*, where both verification rules and IP-cores can be reused moving from an abstraction level to another.

According to these considerations, the emerging transaction level modeling [2] and assertion-based verification [3] are gaining consensus more and more.

In a TL model, the system is designed and verified in terms of functionalities 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 among the computational 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. Transaction level modeling is motivated by a number of practical advantages. These include:

- implementation details are abstracted while preserving the behavioral aspects of the system; this allows a faster simulation (up to 1,000x) than at RTL;
- IP components and buses can be modified and replaced in an easier way than at RTL, thus system level design exploration and verification are simplified;
- an early platform for SW development can be quickly developed;
- deterministic test generation and assertion checking are more effective and less tedious than at RTL, since tests and assertions are written without taking care of the communication protocol between components [2].

In this context, the OSCI TLM library [2] 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].

Functional verification based on assertions represents the main verification technique joining dynamic and static verification [3]. In ABV, assertions are the central focus of the verification process; they detect bugs and guide testbenches in the stimuli production. An assertion, sometimes called a checker or monitor, is a precise description of what behavior is expected when a given input is presented to the design. It raises the level of verification from RTL to TL where users can develop tests and debug their designs closer to design specifications. Consequently, design functions are exercised efficiently (with minimum required time) and monitored effectively by detecting hard-to-find bugs [3]. ABV supports two methods: dynamic verification using simulation, and formal or semi-formal verification using model checking. In this context, a *property* is defined as a boolean description built from Hardware Description Language (HDL) expressions, temporal operators and sequences while an *assertion* is defined as a directive to a tool to prove a property [8]. However, due to the thin difference, the two terms are often indistinctly used. Thus, in the following we generally adopt the term *assertion*.

Previous considerations motivate the recent trend of proposing design and verification methodologies based on TLM and ABV (see Figure 1). However, such a trend raises new challenges for both designers and verification engineers. In fact, it is evident the lack of widely accepted methodologies and reliable tools to automatically derive the RTL implementation, once the TL design has been carried out. The refinement process from TL to RTL is much more difficult than logic synthesis from RTL to gate level. A synthesizable RTL design contains all information required by the synthesis tool to generate the corresponding gate-level netlist. On the contrary, a TL description is very far from including the implementation details which must be added at RTL. Then, a fully automatized process to convert TL designs into RTL implementations is still an utopia. For this reason, it is mandatory that new design and verification methodologies are proposed in order to efficiently check the correctness of the TL-to-RTL manual conversion.

In this context, some approaches based on Transactor-based Verification (TBV) have been recently proposed from both EDA companies and academic researchers [4, 5, 9, 10]. Despite of technical details, all of them exploit the concept of *transactor* to allow a mixed TL-RTL co-verification (Triangle shape in Figure 1). A transactor works as a translator from a TL function call to an RTL sequence of statements, i.e., it provides the mapping between transaction-level requests, made by TL components, and detailed signal-level protocols on the interface of RTL IPs. Thus, testbenches and assertions, defined to check the TL design by ABV, can be directly reused, through the transactor, to apply ABV on the RTL implementation. This avoid time-consuming and error-prone manual conversion of testbenches and assertions moving from TL to RTL.

The effectiveness of reusing testbenches and assertions by TBV, with respect to their manual conversion, has been theoretically proven in [11]. However, the same considerations previously reported to motivate the lack of TL-to-RTL synthesis tools, make evident that assertion reuse is not enough to guarantee the correctness

*This work has been partially supported by the European project VERTIGO FP6-2005-IST-5-033709.

978-3-9810801-2-4/DATE07 © 2007 EDAA

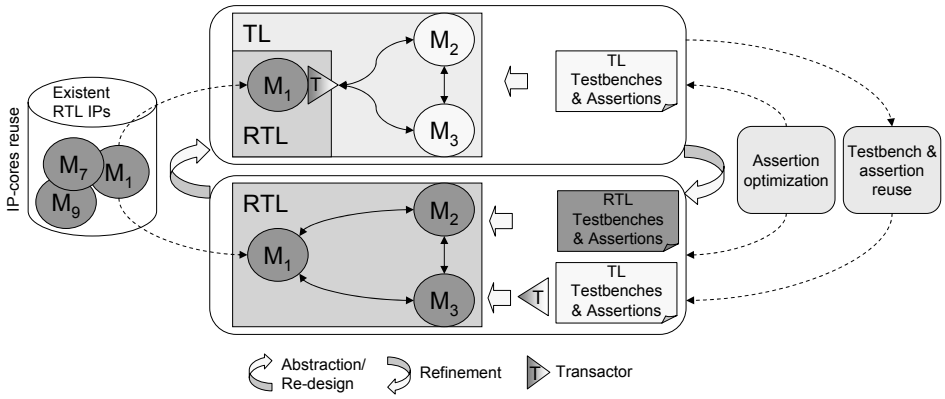

Figure 1: TLM/ABV design and verification flow.

of the refined RTL design. The implementation details added by the refinement process require to be accurately checked by adding new assertions. In this context, the paper presents an incremental ABV methodology to accurately verify the correctness of the TL-to-RTL refinement.

The paper is organized as follows. Section 2 presents an overview of the proposed methodology. Section 3 summarizes the concept of assertion coverage. Sections 4 and 5 describe how already defined assertions can be reused.Then, Section 6 addresses the problem of completing the verification of the TL-to-RTL refinement by adding new assertions. Section 7 highlights the effectiveness of the proposed verification methodology by comparing it with the standard RTL verification which requires to completely generate a new set of RTL assertions after the TL-to-RTL refinement. Finally, experimental confirmation is reported in Section 8 and concluding remarks in Section 9.

2. Verification Methodology

The ABV methodology proposed in this paper incrementally checks the correctness of the TL-to-RTL refinement as shown in Figure 2.

The starting point is represented by the set of TL assertions defined to functionally verify the TL implementation of the design under verification (DUV), when ABV is adopted. The definition of such assertions can be guided by using an assertion coverage metrics [12, 13, 14, 15, 16] that allows one to identify DUV areas not covered by the assertions. At transaction level, verification engineers focus only on the DUV functionality to map the informal specification into a set of formal properties. Temporal relationships between primary inputs (PIs) and primary outputs (POs) and communication protocols between different components are not considered. Thus, generally, TL assertions are expressed as a set of simple Hoare implications [17]. Some of them refer to the relationship between PIs and POs, while others are merged into the TL implementation to check the correctness of particular partial results on internal variables which may hide corner cases.

Assertion definition becomes harder moving from the untimed TL description to the clock-accurate RTL implementation[1]. In fact, during the refinement, timing synchronization and communication protocols between components are added for a more accurate simulation aiming at performance estimation. In this case, an incremental verification methodology, that reuses and refines the TL assertions into an extended set of RTL temporal properties, is preferable to completely defining ex-novo a set of assertions targeted for the RTL description. Thus, we propose to verify the TL-to-RTL refinement as follows:

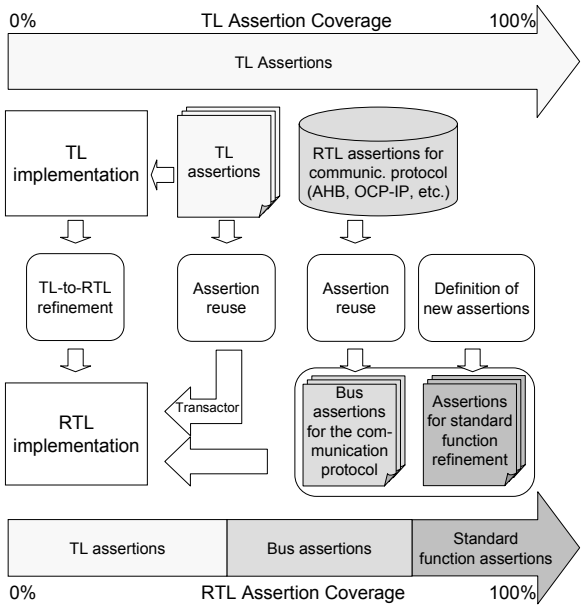

Figure 2: Proposed Incremental ABV methodology.

1. assertions defined at TL are reused to check the functional correctness of the RTL implementation of the DUV by means of a transactor (Section 4);
2. RTL assertions defined to check the correctness of the RTL bus implementation, which will be connected to the RTL DUV, are reused to check the correctness of the RTL DUV communication protocol (Section 5);
3. areas of the RTL implementation not covered by TL assertions or RTL bus assertions are identified by using an assertion coverage metrics (Section 3);
4. new RTL assertions are automatically defined to take care of the implementation details added by the TL-to-RTL refinement process, when TL standard function are replaced by RTL code (Section 6). Such assertions are defined by reusing the already verified TL code through *satellites* [18].

[1]The TL-to-RTL refinement is composed of several steps. In fact, the term transaction level refers to a group of three abstraction levels: TL3 (the highest), TL2, and TL1 (the lowest), each varying in the degree of expressible functional and temporal details [9]. However, without loss of generality, in this paper we consider TL as a single level, but the proposed methodology can be also applied for verifying the correctness of the refinement from TLi to TL$i-1$.

3. Assertion Coverage

The assertion coverage is used to evaluate if a sufficient set of assertions has been defined to be ensured about the correctness of the DUV. In fact, a design implementation that satisfies an incomplete set of assertions cannot be considered bug-free. Thus, in this paper, we use assertion coverage to monitor the effectiveness of the incremental ABV methodology previously summarized.

Different papers have been proposed to address the problem of assertion coverage. The majority of them propose formal method-based methodologies [12, 13, 14, 15] which statically analyze the effectiveness of assertions in covering all states of the DUV. The main limitation of such techniques is represented by the state explosion problem that may arise in case of medium-large DUVs. Thus, we adopt a different approach based on dynamic verification, where the assertion coverage is computed by analyzing the assertion capability of detecting DUV perturbations [16] that affect the behavior of the DUV. Nevertheless, the presented methodology does not strictly depend on such a metrics and can be applied by using any of the several existent coverage metrics.

The assertion coverage methodology is applied on a set of assertions which hold on the DUV. The presence of unsatisfied assertions requires a refinement process of the DUV and/or the assertions themselves. Then, the computation of assertion coverage can be summarized in the following steps:

1. a checker for each defined assertion is generated, by using for example the IBM tool FoCs [19];
2. the DUV is perturbed by using an high-level fault model to obtain a set of perturbed implementations whose behavior differs from the unperturbed one (i.e., perturbation derived by redundant faults are not considered [16]);
3. perturbed implementations are simulated and their behavior is monitored by the checkers. Fault *f* is *covered* by assertion φ if its checker fails during the simulation of the perturbed implementation corresponding to *f*.

If a checker fails in presence of a fault *f*, the corresponding assertion φ is able to distinguish between the perturbed and the unperturbed DUV. This means that φ covers the logic cone of the DUV that may be affected by *f*. Thus, according to the selected fault model, the assertion coverage C_Φ of a set of assertions Φ is defined as:

$$C_\Phi = \frac{\text{\# of faults covered by at least one assertion } \varphi \in \Phi}{\text{\# of generated faults}}$$

All perturbed implementations, whose behavior differs from the faulty-free one, must be covered by the assertion set, i.e. assertion coverage must achieve 100%. A lower assertion coverage is symptom that the assertion set is incomplete, and new assertions must be added to addresses the uncovered perturbations. The same assertion coverage is applied at both TL and RTL.

4. Reusing TL Assertions

Assertions defined to check the TL design should be manually converted into RTL assertions to be used during the verification of the RTL refined design. Such a tedious and error-prone conversion can be avoided by exploiting transactors. In this way, TL assertions can be directly reused at RTL. This represents the first step of the proposed incremental verification methodology.

Following the guideline proposed in [4, 5], a transactor is implemented as a translator from a TL function call to an RTL sequence of statements. It provides the mapping between transaction-level requests, made by TL components, and detailed signal-level protocols on the interface of RTL IPs. Thus, transactors are mainly adopted as interfaces to allow co-simulation of TL-RTL mixed designs. However, a transactor can be exploited to reuse TL assertions on an RTL design as reported in Figure 3.

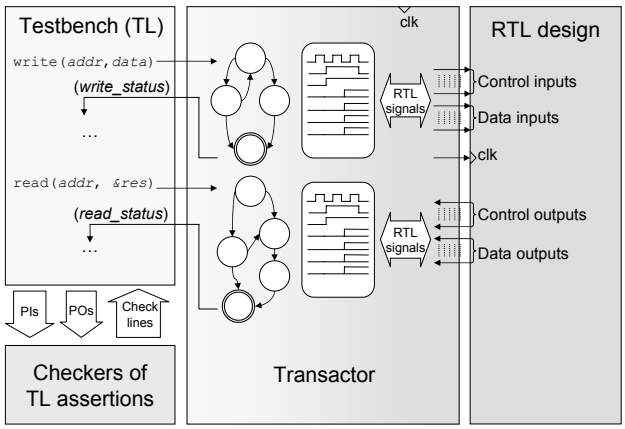

Figure 3: The role of the transactor in assertion reuse.

The testbench carries out one transaction at time, composed by two TL function calls (`write()` and `read()`). It is worth to note that, at TL, 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 TL is untimed, 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 a finite state machine with datapath (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. Thus, data are first 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 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. At this time, assertion checking is invoked. 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 checkers. Finally, the testbench drives the next transaction.

During simulation, the coverage achieved by reusing TL assertions can be computed as described in Section 3. Perturbations not covered by reusing TL properties are investigated to guide the addition of properties related to the communication protocol and implementation details introduced by the TL-to-RTL refinement.

5. Reusing Bus Assertions

It is extremely unlikely that the reuse of TL assertions allows to achieve 100% assertion coverage on the RTL implementation. In fact, further assertions are required to check the communication protocol, since moving from TL to RTL a refinement is performed on the data exchange mechanism between the DUV and the environment where it will be embedded. At TL, communication is generally implemented by message passing through function calls. For example, let us consider a write operation between two components *A* and *B*. At TL, a function call of the kind `write(B, data)` represents the preferable solution. Moreover, it is very likely that the best implementation for the `write`'s body consists of the assign-

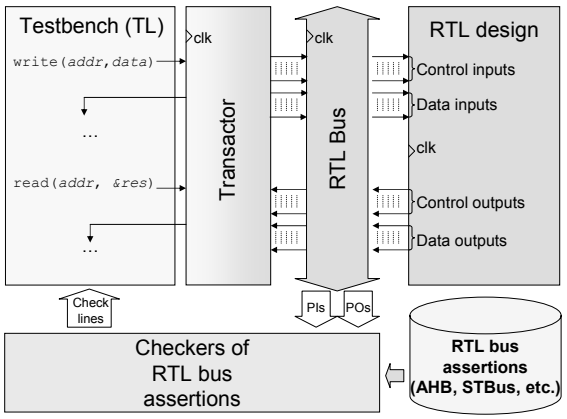

Figure 4: Reuse of RTL bus assertion for checking the DUV communication protocol.

ment `B->buff=data` only, without worrying about if *B* is ready or not to accept data from *A*. If it needs, a classical paradigm based on semaphores can be adopted, when synchronization is explicitly required. Finally, the write operation is instantaneously executed and *B* can use `data` just after `write` returns. On the contrary, at RTL, different components are generally connected through a bus which manages the communication. Thus, a more complex protocol must be implemented to be ensured that write and read operations on the bus are correctly accomplished.

In this context, standard and reliable bus protocols, such as ARM AHB, STBus, OCP-IP, etc., are generally adopted to guarantee a compatibility of the DUV with already existent RTL IP-cores. In such cases, the selected RTL bus implementation is very likely provided with a library of RTL assertions that has been used to check the correctness of the bus [20]. However, such assertions can be used also to verify the correctness of the communication interface of designs connected to the bus. Thus, after TL-to-RTL refinement, we connect the RTL DUV implementation to the desired RTL bus. Then, we check the bus assertions on the whole system composed of the DUV and the bus as shown in Figure 4. The transactor is still adopted to reuse the TL testbench. However, if an RTL testbench is available the transactor can be removed. If some assertion fails, it means that the communication protocol implemented in the RTL DUV is wrong, since, without loss of generality, the bus implementation is supposed to be correct. In this way, already existent RTL assertions can be reused, according to the proposed methodology, to check the correctness of the communication protocol defined during the TL-to-RTL refinement. Finally, as done for TL assertions, the effectiveness of reusing RTL bus assertions is measured by using the assertion coverage metrics presented in Section 3.

6. New Assertions for Standard Functions

In some cases, the reuse of TL assertions and RTL bus assertions is unable to guarantee 100% assertion coverage. This depends on the fact that further details are introduced during the TL-to-RTL refinement besides the communication protocol. At TL, designers can fully exploit the potentiality of the programming language adopted to implement the DUV. In the case of SystemC, for example, all the existing C++ libraries can be used to model the DUV (e.g., the huge amount of functions defined in the mathematical library or in the standard template library). These functions allow to implement complex functionalities in very few code lines. Moreover, the correctness of such functions is almost definitely guaranteed by the correctness of the adopted language libraries[2]. For example, a call to `sqrt(n)` is enough to compute the root of number `n`, and no assertion must be defined to check the correctness of the result. If it needs, TL assertions are defined to check if the parameter `n` assumes the correct value before calling `sqrt()`. On the contrary, during the TL-to-RTL refinement, C++ standard functions included in the TL description are substituted by synthesizable implementations whose correctness must be checked by defining new assertions. In this context, we propose a standard template that can be used to automatically define such assertions.

The main idea consists in providing a mechanism to reuse already checked TL code into RTL assertions. Thus, for example, checking if the RTL implementation of the `sqrt()` TL function is correct can be performed by writing an assertion which executes the TL function, and then compares its result with the one provided at RTL. In this way, assertion definition is straightforward and it can be automatized. On the contrary, it is very hard and time-consuming writing an RTL assertion which uses only operators provided by a temporal logic (e.g., CTL, LTL, etc.) to check the correctness of the RTL `sqrt()` implementation.

The Property Specification Language [18] (PSL) allows to define assertions including pieces of code note as satellites. Such assertions can be verified by using commercial tools (e.g., Magellan [21]) that provide a simulation-based assertion checking engine. However, only a subset of SystemVerilog constructs can be used to create satellite-based assertions, while SystemC needs a lot of work to be partially supported[3]. On the contrary, in this paper we propose a technique to fully exploits SystemC TL code.

To better clarify the proposed approach, we refer to the simple example of Figure 5 which shows how an assertion can be defined to verify the RTL implementation of `sqrt()` by reusing the TL code. The assertion definition consists of the following steps.

1. A set of checkpoints is identified into the RTL implementation to mark the RTL functionality which is the target of the verification. In particular, the designer has to define a checkpoint for each register which represents an input value of the RTL functionality to be verified, and a checkpoint for the register which represents the result. Each checkpoint is implemented by inserting a call to the SystemC `notify()` function. In our example, two checkpoints are inserted into the RTL design: `notify(event_n)` in state C_n, where the value of n (representing the input of `sqrt()`) is available, and `notify(event_sqrt)` in state C_{sqrt} where the `sqrt()` result is ready to be checked.
2. A checker is defined to compare the RTL functionality to be verified with the corresponding, already checked, TL code. The checker implements three methods: `set_value()` to store the value of registers involved in the verification, `check_assertion` to start the assertion verification, and `satellite_sqrt()` which includes the reference TL code. When `check_assertion()` is called by the testbench, the checker exploits the satellite to calculate the reference value to be compared with the RTL result. In our example, the satellite includes a call to the C++ `sqrt()` function However, more complex TLM code can be inserted into the satellite according to the RTL functionality to be verified.
3. Two checkpoint processes (`start_sqrt` and `end_sqrt`) are instantiated into the testbench. These processes are sensitive to a checkpoint event (respectively `event_n` and `event_sqrt`).

[2]Checking the correctness of the language libraries is out of the scope of this paper.

[3]Even if the last Language Reference Manual of PSL (IEEE-1850) reports the SystemC flavor, its Working Group is still active to solve several syntactical inconsistencies between PSL and C++/SystemC.

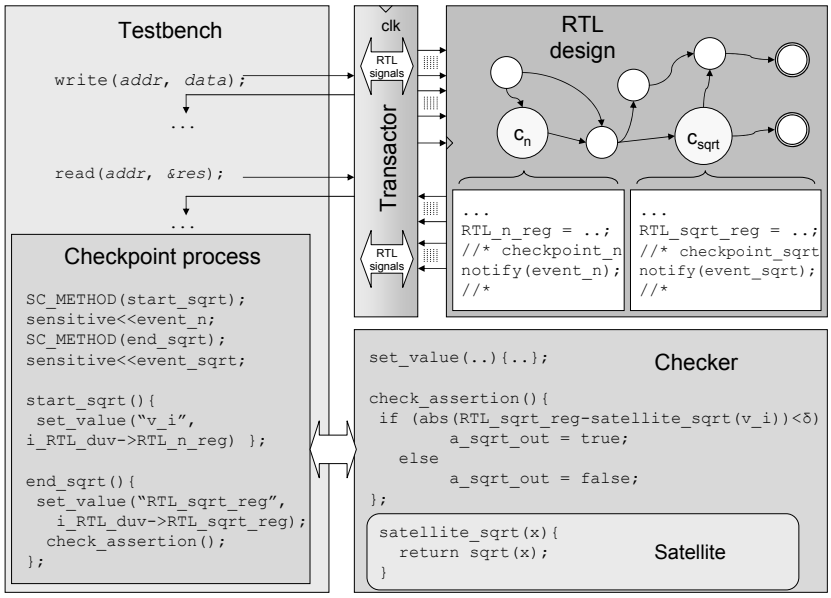

Figure 5: Example of a satellite-based assertion.

Thus, they wake up whenever their event happens. The transactor opportunely drives the RTL PIs to start the RTL simulation, after the testbench calls the `write()` function[4]. When the first checkpoint related to `n` is reached (state C_n), `notify(event_n)` alerts that the value of `n` has been computed and it is ready to be stored by the checkpoint process (by calling `set_value()`). In the same way, when the checkpoint in state C_{sqrt} is reached, the checkpoint process wakes up and it stores the result of the RTL computation. Finally, the checker's method `check_assertion()` is invoked to verify if the RTL implementation provides the same result computed by the corresponding TL code.

7. Comparison to a standard RTL verification

Designers that adopt a standard RTL verification flow must define an ex-novo set of assertions to verify the RTL DUV after the TL-to-RTL refinement. On the contrary, the incremental ABV methodology proposed in the previous sections allows verification engineers to avoid the time-consuming and error prone activity of defining ex-novo RTL properties. Thus, we claim that the proposed methodology reduces the total effort spent to verify the RTL DUV. In fact, it exploits the *reuse* concept without any additional effort due to the following main reasons:

- TL assertions related to the DUV functionality are reused by means of transactors. No additional cost is required for the transactor implementation, since it must be defined in any case. In fact, the transactor is an essential component during the TL-to-RTL refinement. Besides, its generation is going to be an automatic process [22].
- RTL assertions provided with the bus adopted to implement the communication protocol can be directly reused without any particular effort. Thus, verification engineer must written RTL assertions related to the communication protocol only when they are not already available. On the contrary, their definition is always mandatory when the standard RTL verification flow is adopted.

[4]Note that the use of the transactor is not essential for the definition of satellite-based assertions. In the case an ad-hoc RTL testbench is available, the transactor can be removed without any modification to the proposed approach.

- Pieces of TL code are reused, within satellites, to define RTL assertions that check parts of the RTL implementation corresponding to TL standard functions (that do not required to be checked at TL). The satellite template presented in Section 6 allows us to automatically define such new properties. The cost of inserting the checkpoints required by the satellite is negligible, and it can be done during the TL-to-RTL refinement. On the contrary, specifying the characteristics of a standard function (e.g., `sqrt()`) by means of RTL temporal operators may be very difficult.

8. Case Study

The effectiveness of the proposed verification methodology has been evaluated by using the STMicroelectronics *Face Recognition System* [23]. In particular, three modules (*ROOT*, *DIV* and *DISTANCE*) have been considered, since they were selected to become HW components. Thus, their TL descriptions have been refined into RTL models and connected to an AMBA AHB bus. The communication between the DUVs and the bus has been implemented by defining a transducer, i.e., a component that allows modules with different interface protocols to communicate [24]. The characteristics of RTL implementations are reported in Table 2. The Table shows, respectively, the number of gates and flip-flops, and the number of TL and RTL faulty implementations generated to compute the assertion coverage as reported in Section 3.

Columns *Cov.* of Table 1 show the assertion coverage percentage achieved for each module, respectively, at TL (*TL*), at RTL after the TL-to-RTL refinement by applying the incremental ABV methodology proposed in this paper (*RTL(incremental)*), and at RTL by defining new RTL assertions according to the standard RTL verification flow (*RTL(standard)*). The TL description of the considered modules has been verified by defining a total number of 19 assertions which achieve 99% assertion coverage. After TL-to-RTL refinement, the RTL implementation has been verified by using a total number of 34 assertions according to the proposed incremental verification methodology. Reused TL assertions (*TL Asser. Reuse*) are 19, RTL bus assertions reused to target the communication protocol implemented by the transducer (*Bus Assertions*) are 7 (the same assertions for all modules, since these have been connected to the same AMBA AHB bus), and satellite-based assertions (*Satellites*) are 8. In particular, satellites have been used to check the RTL cor-

		ROOT		DIV		DISTANCE		TOTAL	
		Asser.#	Cov.	Asser.#	Cov.	Asser.#	Cov.	Asser.#	Cov.
TL	TL Assertions	11	99%	5	99%	3	99%	19	99%
RTL(incremental)	TL Asser. Reuse	11	79%	5	79%	3	78%	19	79%
	TL Asser. Reuse + Bus Assertions	11+7	95%	5+7	87%	3+7	88%	19+7	90%
	TL Asser. Reuse + Bus Assertions + Satellites	11+7+1	99%	5+7+3	96%	3+7+4	96%	19+7+8	97%
RTL(standard)	RTL Assertions	11	99%	10	96%	12	96%	33	97%

Table 1: Assertion coverage results.

Module	Gates	FFs	TL faults	RTL faults
ROOT	7802	155	196	1955
DIV	11637	269	1017	2661
DISTANCE	40663	100	2327	3389

Table 2: Characteristic of the case study.

rectness of the following functionalities: for *ROOT*, the square root algorithm; for *DIV*, the computation of a normalization factor used to remove the blue component of a pixel and accordingly recompute the red and green ones; and for *DISTANCE*, the algorithm used to compute the distance of the red, blue and green components of a target pixel with respect to images stored in the face database. Finally, we have tried to define RTL properties ex-novo. The achieved coverage (*RTL(standard)*) is comparable with the one achieved by the incremental methodology, however, the ex-novo definition of RTL properties required one week of work, while the set up of the proposed incremental methodology required few hours.

9. Concluding Remarks

The paper addressed the problem of verifying the correctness of the TL-to-RTL refinement process. An incremental ABV methodology has been proposed which relies on four basic concepts: transactor, bus assertions, satellite and assertion coverage. Transactor has been introduced to reuse TL assertions avoiding tedious and error-prone manual conversions. Assertions defined to check standard RTL bus implementations can be reused to verify the correctness of the communication protocol introduced during the TL-to-RTL refinement. Satellites have been used to automatically define new assertions that exploit already checked TL code to verify implementation details typical of the RTL implementation. Finally, a fault model-based assertion coverage has been adopted to incrementally measure the capability of assertions in covering all the DUV functionalities. The proposed methodology allows to avoid the ex-novo manual definition of RTL assertions which represents a time-consuming and error-prone activity.

References

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

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

[3] Synopsys Inc. *Assertion-Based Verification*, 2003. White paper. www.synopsys.com.

[4] D. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. N. 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] Doulos and Mentor Graphics. *Verification Methodology in a Mixed Language Environment*. In *Solutions Workshop 3 at IEEE DATE*. 2006.

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

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

[11] N. Bombieri, F. Fummi, and G. Pravadelli. *On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL*. In *IEEE DATE*, vol. 1, pp. 1–6. 2006.

[12] Y. Hoskote, T. Kam, P. H. Ho, and X. Zao. *Coverage Estimation for Symbolic Model Checking*. In *Proc. of ACM/IEEE DAC*, pp. 300–305. 1999.

[13] S. Katz, O. Grumberg, and D. Geist. *Have I Written Enough Properties? - A Method of Comparison between Specification and Implementation*. In *Proc. of IFIP CHARME*, pp. 280–297. 1999.

[14] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. *A Practical Approach to Coverage in Model Checking*. In *Proc. of CAV*, pp. 66–78. 2001.

[15] N. Jayakumar, M. Purandare, and F. Somenzi. *Dos and Don'ts of CTL State Coverage Estimation*. In *Proc. of ACM/IEEE DAC*, pp. 292–295. 2003.

[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] C. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, vol. 12(10):pp. 576–585, 1969.

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

[19] 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 *LNCS*, pp. 538–542. Springer-Verlag, 2000.

[20] P. Wodey, G. Camarroque, F. Baray, R. Hersemeule, and J.-P. Cousin. *LOTOS code generation for model checking of STBus based SoC: the STBus interconnection*. In *ACM/IEEE MEMOCODE*, pp. 204–213. 2003.

[21] Synopsys. *Magellan - Hybrid RTL Formal Verification*, http://www.synopsys.com/products/magellan/, 2004.

[22] F. Balarin and R. Passerone. *Functional Verification Methodology Based on Formal Interface Specificaton and Transactor Generation*. In *IEEE DATE*. 2006.

[23] M. Borgatti, A. Capello, U. Rossi, G.L.Lambert, I. Moussa, F. Fummi, and G. Pravadelli. *An Integrated Design and Verification Methodology for Reconfigurable Multimedia System*. In *IEEE DATE*, pp. 266–271. 2005.

[24] H. Cho, S. Abdi, and D. Gajski. *Design and Implementation of Transducer for ARM-TMS Communication*. In *Proc. of IEEE ASP-DAC*, pp. 126–127. 2006.