On the reuse of RTL assertions in SystemC TLM verification

Nicola Bombieri[1,2], Franco Fummi[1,2], Valerio Guarnieri[1]
Graziano Pravadelli[1,2], Francesco Stefanni[1], Tara Ghasempouri[2],
Michele Lora[2], Giovanni Auditore[3], Mirella Negro Marcigaglia[3]
[1]EDALab s.r.l. - Verona, Italy {name.surname}@edalab.it
[2]Department of Computer Science - University of Verona, Italy[2] {name.surname}@univr.it
[3]STMicroelectronics s.r.l. - Catania, Italy {name.surname}@st.com

***Abstract*—Reuse of existing and already verified intellectual property (IP) models is a key strategy to cope with the complexity of designing modern system-on-chips (SoC)s under ever stringent time-to-market requirements. In particular, the recent trend towards system-level design and transaction level modeling (TLM) gives rise to new challenges for reusing existing RTL IPs and their verification environment in TLM-based design flows. While techniques and tools to abstract RTL IPs into TLM models have begun to appear, the problem of reusing, at TLM, a verification environment originally developed for an RTL IP is still underexplored, particularly when assertion-based verification (ABV) is adopted. Some techniques and frameworks have been proposed to deal with ABV at TLM, but they assume a top-down design and verification flow, where assertions are defined ex-novo at TLM level. In contrast, the reuse of existing assertions in an RTL-to-TLM bottom-up design flow has not been analyzed yet. This paper proposes a methodology to reuse assertions originally defined for a given RTL IP, to verify the corresponding TLM model. Experimental results have been conducted on benchmarks of different characteristics and complexity to show the applicability and the efficacy of the proposed methodology.**

I. Introduction

In the past years, there has been a consolidation of description languages, methodologies and tools for the design and verification of digital systems at different abstraction levels. VHDL and Verilog have been recognized to be the de-facto standard modeling languages for design and verification at register transfer level (RTL). SystemC and transaction level model (TLM) [1] have gained a broad consensus for system-level design and verification, architectural exploration and HW/SW co-simulation [2].

An important consequence of such a language and paradigm heterogeneity in the today's design flows is that an IP model is often implemented and optimized twice, at RTL and TLM. At the state of the art, the two implementations are developed by hands, independently, and, often, by different people. This makes difficult to maintain consistency between the two models. While one of the two evolves for any reason (i.e., customization, update, etc.), the other one needs to be manually adapted. This approach is, from an industrial point of view, expensive and not always convenient.

This work has been partially supported by the EU large-scale integrating project SMAC (SMArt systems Co-design) FP7-ICT-2011-7-288827.

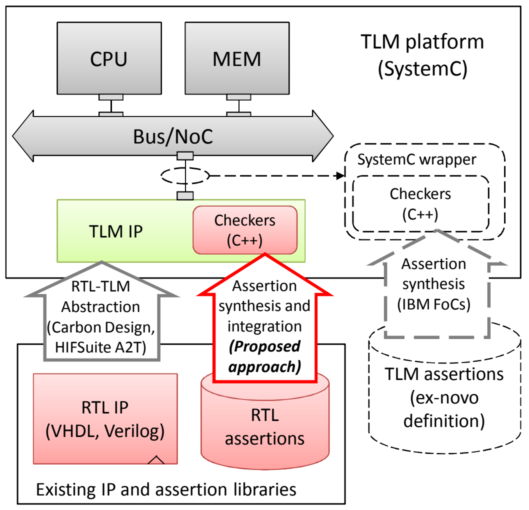

CPU
MEM
TLM platform (SystemC)
Bus/NoC
SystemC wrapper
Checkers (C++)
TLM IP
Checkers (C++)
Assertion synthesis (IBM FoCs)
RTL-TLM Abstraction (Carbon Design, HIFSuite A2T)
Assertion synthesis and integration (Proposed approach)
TLM assertions (ex-novo definition)
RTL IP (VHDL, Verilog)
RTL assertions
Existing IP and assertion libraries

Fig. 1. Reuse of existing RTL IPs and assertions in SystemC TLM design flows

In this context, methodologies and tools for the automatic generation of SystemC TLM models starting from existing RTL IPs have been recently proposed [3], [4], [5] and represent a valuable support for the design of modern complex systems (see left-most side of Figure 1).

On the other hand, the introduction of an automated RTL-to-TLM abstraction flow requires validation strategies to guarantee that the abstracted model is correct with respect to the starting RTL IP and that it behaves correctly once plugged into the TLM system model.

Different strategies have been proposed to adapt RTL verification techniques at TLM. Formal equivalence checking cannot be often applied being the process of abstraction intrinsically disruptive from a pure equivalence point of view [6], [7], [8]. In contrast, some simulation-based techniques [9], [10], [11], [12] and frameworks [13], [14], [15] have been proposed to allow designers adopting assertion-based verification (ABV) at transaction level.

ABV approaches require the definition of a set of (temporal) assertions that formally represent the intent of the designers (*specification*), and a *decision procedure* to check the consistency between such assertions and the design under verification

(DUV). ABV has been extensively applied to verify RTL models, where the trigger mechanism is guaranteed by the presence of a clock signal. In contrast, the application of ABV to more abstracted models like, for instance, TLM designs, is not straightforward. TLM models are represented with a set of event-based, non-clocked, untimed or timed-annotated descriptions that cannot easily fit with the concept of explicit discrete time passing that underlies the semantics of temporal assertions [14].

All the techniques recently proposed to apply ABV at TLM can be classified into two categories: techniques that define a way to specify temporal assertions and that suppose the presence of an event-based triggering mechanism [16], or, techniques that correctly synchronize checker activation and DUV simulation [9]. In both cases, the assertions are synthesized into *checkers*, which monitor inputs and outputs of the DUV during simulation (see right-most side of Figure 1) by searching for behaviors that are not consistent with respect to the corresponding assertions. All these works assume a top-down design and verification flow where assertions are defined ex-novo at TLM level. In case of bottom-up flows (i.e., RTL IP reuse), verification engineers define, for the second time, the set of assertions to check the correctness of the abstracted TLM models, even when RTL assertions are already available for the original RTL implementations.

Up to now, no paper exists in the literature that proposes a strategy for reusing, at TLM, assertions that have been originally defined at RTL. This work is intended to fill in the gap by proposing an automatic methodology to reuse RTL assertions into SystemC TLM models (see central part of Figure 1). In this way, error-prone and time consuming manual re-definition is avoided. Thus, verification engineers can focus their attention on the definition of assertions for checking the functionality of new components and the correct integration of the whole TLM system composed of new and abstracted components.

Experimental results have been conducted on different benchmarks and several RTL assertions have been synthesized into checkers to be plugged in the system platform. The results show the applicability of the methodology in reusing almost all the existing RTL assertions at TLM. They also show that the overhead introduced by such checkers in the TLM simulation platform is acceptable considering the advantages of the automatic process.

The rest of this paper is organized as follows. Section II presents a more accurate analysis of the state of the art. Section III gives an overview of the most important concepts of ABV and RTL-to-TLM abstraction for a better understanding of the proposed methodology. Section IV presents the methodology. Section V reports the experimental results, while Section VI is devoted to conclusions and remarks.

II. Related works

The problem of applying ABV at TLM has been investigated first for cycle-accurate TLM models [2], [17]. In [2] the assertions and the DUV are modelled by using abstract state machines and an approach is presented to perform static verification. In contrast, dynamic ABV is considered in [17], where a way to wrap C++ checkers into SystemC cycle-accurate descriptions is presented. However, these solutions are not suited for higher (asynchronous, untimed or timed-annotated) TLM levels whose semantics is not based over discrete time steps.

ABV at higher levels is mainly addressed by defining new synchronization mechanisms that replace, at TLM, the traditional RTL synchronization based on clock events. In [16], [18], general concepts and requirements related to the use of dynamic ABV at TLM are defined for the specific case of TLM 1.0. A specific assertion language is proposed to define assertions independently from the abstraction level, by assuming an event-based synchronization mechanism instead of the traditional clock-based approach adopted at RTL. A SystemC implementation of an ABV framework that relies on such a language is then described in [13]. Verification of TLM 1.0 models is proposed also in [19], which defines a library of assertions to allow self-checking of TLM channels.

Synchronization policies between assertion checkers and DUV have been also proposed in several works. In [9], checkers generated by using FoCs [20] are considered and evaluated at the starting of each transaction of SystemC TLM designs. Automatic generation of checkers suited to perform dynamic ABV at TLM are also presented in [21], [11].

A formal tool for assertion checking of TLM SystemC descriptions is proposed in [14]. The description is first converted into C code, then monitor logic is implemented by means of C asserts and finite state machines. Bounded model checking is finally employed to complete the verification process.

Finally, a methodology to check the functional consistency between TLM and RTL models is proposed in [22], where the reuse of TLM assertions at RTL is guarantee by ad-hoc refinement rules.

All previous approaches assume a top-down design and verification flow where assertions are originally defined at TLM, and, possibly, they are reused at lower abstraction levels, like, for instance, in [23]. In contrast, our approach addresses a different problem that fits bottom-up flows, i.e., how to reuse assertions defined at RTL so that they can be used to verify a TLM design where abstracted versions of existing RTL IP-cores are plugged into SystemC TLM system platforms.

III. Assertion-based verification in TLM

This section firstly summarizes the preliminary concepts related to Property Specification Language (PSL) [24], since it is one of the most widespread language for specification of temporal assertions. Then, the most important concepts related to RTL-to-TLM are presented to better understanding the assertion integration presented in the following sections.

A. PSL assertions

PSL is nowadays one of the most prominent standards for defining assertion specification. It defines a concise syntax with clearly defined formal semantics. PSL has been proposed

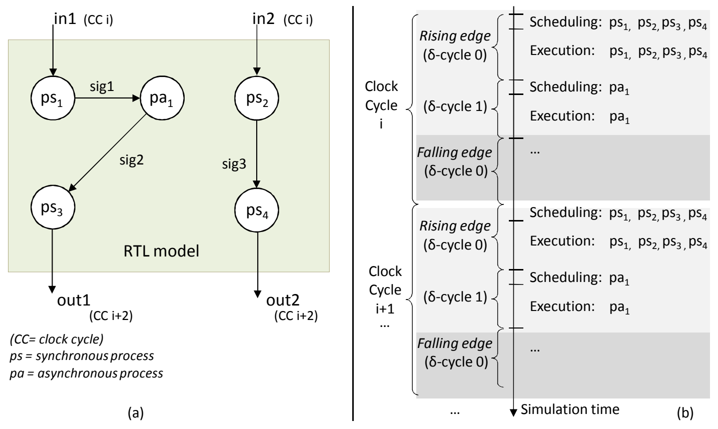


Fig. 2. Dynamic scheduling overview: RTL model example (a) and the corresponding process scheduling over simulation time (b)

by the Accellera consortium, it is based on the Sugar language from IBM and it shows many similarities with respect to SystemVerilog Assertion (SVA), the assertion sub-language of SystemVerilog. However, while SVA is strictly connected to SystemVerilog, PSL is a multipurpose, multilevel, multiflavor language. It is intended to be used for both functional verification and functional specification. Thus, it can be seen as an executable documentation for hardware and embedded software design.

PSL assertions are built upon four layers which cooperate to guarantee the expressiveness of the language.

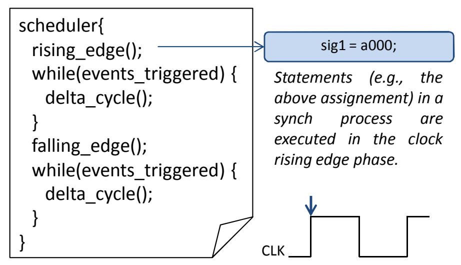
- The *Boolean layer* is adopted to build basic expressions commonly used by the other layers;
- The *Temporal layer* can be considered as the core of the language since it gives the possibility of describing temporal relations, evaluated over a set of evaluation cycles;
- The *Verification layer* provides the directives for using assertions during a verification run;
- The *Modeling layer* can be used to characterize the behavior of design inputs and to model auxiliary variables.

Since the approach proposed in this paper is based on simulation, we consider the "simple subset" of PSL, which conforms to the notion of monotonic advancement of time, and it is close to the notion of simulation itself.

The semantics of PSL assertions is defined with respect to *finite* or *infinite* traces. In dynamic verification, however, only behaviors that are finite in length are considered. For this reason, the standard defines four levels of satisfaction of an assertion: *holds strongly*, *holds*, *pending* (*i.e.*, no bad states have been encountered but future obligations have not been met), and *fails*.

B. RTL-to-TLM abstraction

Despite technical differences, the tools for automatic RTL-to-TLM abstraction [3], [4] generate SystemC TLM code by translating hardware description language (HDL) statements into SystemC statements and by handling the RTL concurrency through dynamic scheduling. In dynamic scheduling (see



```
scheduler{
  rising_edge();
  while(events_triggered) {
    delta_cycle();
  }
  falling_edge();
  while(events_triggered) {
    delta_cycle();
  }
}
```

sig1 = a000;

Statements (e.g., the above assignement) in a synch process are executed in the clock rising edge phase.

Fig. 3. Overview of the SystemC TLM scheduling activity

Figure 2), the RTL processes (i.e., concurrent statements) are woke up if and only if there has been an event to which they are sensitive. The simulated time has a finest granularity equal to one clock period when the generated TLM model is cycle accurate. On the clock rising event, all synchronous processes are firstly run. Then, if any event has been triggered (e.g., write on a signal), the asynchronous processes sensitive to that event are woke up. The routine iteratively goes on until there is not any further event. At each of these iterations corresponds a *delta cycle*, which is a simulation cycle in which the simulated time does not advance [25].

The SystemC TLM code is generated by translating RTL processes into C++ functions, and by implementing the dynamic scheduling through a C++ routine (i.e., the scheduler of functions), which reproduces the behavior of the RTL scheduler.

Figure 3 gives a high-level example of the scheduler activity of the cycle accurate TLM model generated from a synchronous RTL model. At each clock event, the scheduler first invokes the synchronous functions sensitive to the rising edge of the clock (`rising_edge()` in Figure 3 represents these invocations). Then, the scheduler iteratively invokes the asynchronous functions (`delta_cycle()` invocation) and moves on to the falling edge phase (`falling_edge()`) to invoke any process synchronous to the falling edge of the clock.

Existing tools generate SystemC TLM models that are accurate enough to guarantee simulation of timing delays. In this context, the proposed methodology applies to two different scenarios:

1) The generated SystemC TLM model is cycle accurate. In the SystemC simulation, a TLM transaction is run for each RTL clock cycle. The digital IP presented in Figure 4(a) is an example of this scenario.
2) The generated TLM model derives from an RTL implementation with two clock signals. The SystemC TLM model is cycle accurate for one of them only. The second clock signal is abstracted, i.e., a number of this clock cycles are included into one TLM transaction. The digital IP presented in Figure 4(b) is an example.

For both scenarios, the proposed methodology synthesizes the existing assertions into checkers and integrates such checkers into the SystemC TLM code, as detailed in the following section.

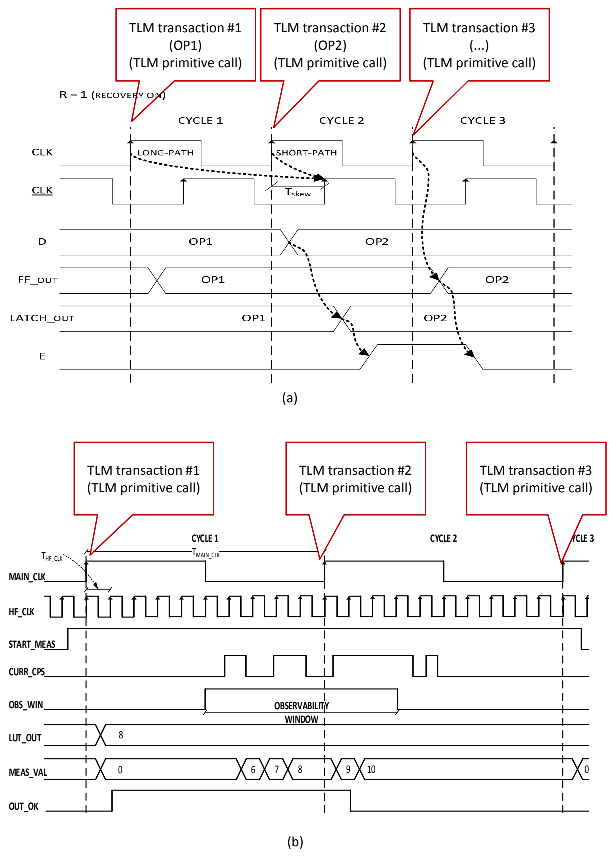

TLM transaction #1 (OP1) (TLM primitive call)
TLM transaction #2 (OP2) (TLM primitive call)
TLM transaction #3 (...) (TLM primitive call)
R = 1 (RECOVERY ON)
CYCLE 1
CYCLE 2
CYCLE 3
CLK
LONG-PATH
SHORT-PATH
CLK
Tskew
D
OP1
OP2
FF_OUT
OP1
OP2
LATCH_OUT
OP1
OP2
E
(a)
TLM transaction #1 (TLM primitive call)
TLM transaction #2 (TLM primitive call)
TLM transaction #3 (TLM primitive call)
THF_CLK
TMAIN_CLK
CYCLE 1
CYCLE 2
CYCLE 3
MAIN_CLK
HF_CLK
START_MEAS
CURR_CPS
OBS_WIN
OBSERVABILITY WINDOW
LUT_OUT
8
MEAS_VAL
0
6
7
8
9
10
0
OUT_OK
(b)

Fig. 4. Mapping of RTL waveforms to TLM transaction sequences: example of scenario 1 (a) and scenario 2 (b)

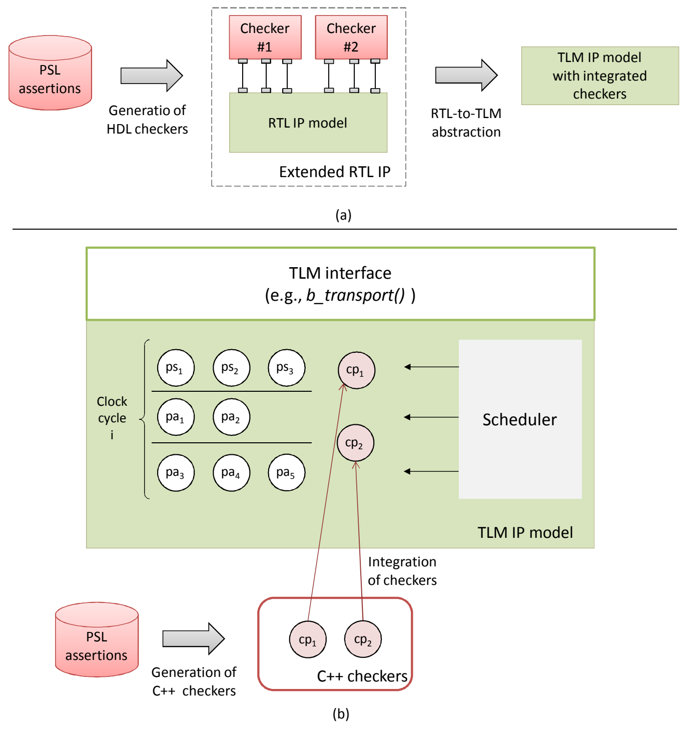


Fig. 5. The generation and integration phases in the two alternatives: generation of HDL checkers and RTL-to-TLM abstraction (a), generation of C++ checkers and integration in the abstracted TLM IP model (b)

IV. Methodology

The methodology consists of two main phases:

1) Given a set of assertions defined for ABV at RTL, applying an automatic checker generator to obtain the corresponding *checkers*, as explained in Section IV-A.
2) Checker integration into the abstracted TLM description, with the aims of exploiting the timing information from the scheduling activity of the SystemC TLM model to drive the checkers, as explained in Section IV-B.

A. Generation of checkers

In the first phase, a checker generator (e.g., IBM FoCs [26]) is applied to automatically generate run-time checkers from a starting set of generic assertions. It is worth noting that the proposed methodology is independent from the checker generator employed in this step. The run-time checkers can be automatically generated in two different ways:

1) *Generation of HDL checkers*. The first alternative consists of generating checkers in HDL language (which are usually called *monitors* in literature). The checker behavior is implemented thorough HDL processes and represents the state machine modeling the assertion semantics. The checkers are connected to the RTL IP (see Figure 5(a)) and the *extended* RTL IP can be then abstracted into SystemC TLM through any automatic RTL-to-TLM tool.
2) *Generation of C++ checkers*. The second alternative consists of generating checkers in C++ (see Figure 5(b)). In this case, the checker generator generates software routines that implement the automaton corresponding to the assertion semantics. In general, the checker generator generates two routines. The first one has to be invoked at every event (which is defined by the @-clause in PSL) of the given assertion (e.g., the rising edge of the clock). This allows the checker to evolve through the state machine and to assess the assertion (true/false). The second routine has to be called whenever the abort condition of the assertion occurs. The generated routines are then integrated into the scheduler of the SystemC TLM model, as explained in the following section.

In the proposed methodology, we adopt the second alternative. The choice is driven by portability and performance aspects. A comparison between the two different approaches is presented in the following.

B. Integration of checkers in the TLM description

After checkers have been generated, the main focus of the proposed methodology lies on where to integrate them within the abstracted TLM description. A strategy is devised to insert calls to the C++ routines that implement checkers by the process scheduler (see Figure 5(b)) that is responsible for carrying out the design functionality in the TLM description.

Since the process scheduler in the abstracted description distinguishes between synchronous functions and asynchronous functions, the routine calls are inserted in the proper scheduling function i.e., `rising_edge()`, `falling_edge()` or `delta_cycle()` (see Figure 3). In order to do so, the @-clause of the corresponding PSL assertion is examined, since it regulates how timesteps are determined during the evaluation of the assertion carried out by the generated checker. Furthermore, if the assertion features an `abort` clause, it must be also taken into account, as such a clause is asynchronous with respect to the @-clause.

If the @-clause refers to the rising edge (or the falling edge) of the clock signal, then the invocation to the C++ routine that implements the evolution of the checker is added at the end of the `rising_edge()` (or the `falling_edge()` scheduling functions). Otherwise, if the @-clause refers to a non-clock signal, then an if-condition checking whether the specified event occurred is added at the end of the `delta_cycle()` scheduling function. If such condition evaluates to true, the checker routine is invoked (once in the whole scheduling cycle) to allow a proper evolution of the state machine within the checker. Additionally, if the PSL assertion contains an `abort` clause, then an if-condition checking whether the abort condition occurred is added at the end of the `delta_cycle()` scheduling function. If such condition evaluates to true, the corresponding abort routine of the checker is invoked.

For example, let us consider the following RTL assertion written in PSL:

```
assert always ({[*1]; stable(stx)[*16]})
abort preset='0' @rising_edge(pclk)
```

Since the @-clause refers to the rising edge of the `pclk` clock signal, an invocation to the C++ routine that evolves the state machine is added at the end of the `rising_edge()` scheduling function. In order to properly take into account the `abort` clause, an if-condition checking whether the `preset` reset signal is low is added at the end of the `delta_cycle()` scheduling function. An invocation to the abort routine of the checker is then added to this if-branch.

The two alternatives for generating checkers (i.e., HDL and C++) are equivalent from the functionality point of view. As such, an invocation to the routine implementing the evolution of the checker is performed by the scheduler whenever the corresponding event specified in the @-clause occurs. The correspondence between original RTL events and TLM events is guaranteed whenever the clock accuracy is preserved and annotated in the TLM description (i.e., in the two scenarios presented in Section III-B).

The proposed integration methodology offers the following advantages over the abstraction of the RTL description together with RTL checkers:

- It is less time-consuming since the integration of C++ checker routines into the TLM scheduling functions is more immediate than the integration of RTL checkers within the starting RTL IP model.
- It relies on a higher-level implementation of the checkers, thus reducing the overhead caused by their introduction. In fact, directly generated C++ checker routines are bound to have better performance in simulation than their abstracted RTL counterparts.

TABLE I
CHARACTERISTICS OF THE RTL IPS.

Design	Processes (#)		RTL	Pipeline	Latency
	Asynch.	Synch.	loc	stages (#)	(cc)
UART	416	77	5866	–	16
Root	2	0	343	–	16
Div	1	5	1283	–	16
QNR	7	17	518	16	16
RLE	14	17	678	9	9
FDCT	259	196	5935	388	67
JPEG	281	231	18381	80	80
Error Correction	6	11	1666	–	130
Lambda Root	0	5	1092	–	790
Omega Phy	17	4	1595	–	294

V. EXPERIMENTAL RESULTS

The proposed methodology has been applied to different VHDL IP descriptions: a *UART* module, two sub-components of a face-recognition system (*i.e.*, *Root* and *Div*), a JPEG encoder and its sub-components (*i.e.*, *QNR*, *RLE*, *FDCT*) and some components of a Reed-Solomon decoder (*i.e.*, *Error Correction*, *Lambda Root*, *Omega Phy*). Table I reports their main characteristics in terms of number of synchronous and asynchronous processes, number of lines of code (*loc*), number of pipeline stages, clock cycle latency, and throughput. The RTL SystemC descriptions have been obtained by using the HDL conversion tools provided by HIFSuite [4], while the SystemC TLM descriptions have been generated by HIFSuite A^{2}T.

To evaluate the simulation overhead caused by the insertion of run-time checkers three different contexts have been tested for both the RTL and TLM description. The first one consists of the IP description at RTL and TLM without any checker. This allows us to estimate the speed-up due to the RTL-to-TLM abstraction. The second and third versions represent the IP model with a few and many C++ checkers, respectively (in particular, two and forty C++ checkers) to evaluate the overhead caused by a different amount of inserted checkers. The set of C++ checkers plugged in both the RTL and TLM models is the same.

Table II reports the simulation time for each version of both RTL and TLM, in order to observe the overhead involved by the plugged checkers at different abstraction levels, as well as the simulation speed-up between RTL and TLM simulations. For every design, columns *Checkers* identifies the version (*i.e.*, with 0, 2 or 40 checkers), column RTL and TLM report the execution time (in seconds) employed by the simulation. The columns *Overhead* report the overhead for the checker executions in percentage, with respect to the equivalent version without checkers. Finally the *speed-up* column reports the simulation speed-up between the RTL and TLM implementations.

The experimental results highlight the impact of the checker execution over the whole simulation. The highest speed-up is

TABLE II
EXPERIMENTAL RESULTS.

Design	Checkers (#)	RTL (s)	Overhead (%)	TLM (s)	Overhead (%)	Speed-up (x)
UART	0	24.186	–	11.614	–	2.083
	2	54.691	55.78	23.718	51.033	2.306
	40	588.706	95.892	458.940	97.469	1.283
Root	0	22.749	–	19.941	–	1.14
	2	97.438	76.653	36.998	46.105	2.63
	40	1422.161	98.4	1203.099	98.343	1.18
Div	0	46.027	–	20.785	–	2.21
	2	125.428	63.304	23.045	9.807	5.44
	40	1528.480	96.989	665.410	96.876	2.30
FDCT	0	105.587	–	18.566	–	5.69
	2	209.647	49.636	34.577	46.305	6.03
	40	2250.879	95.309	1054.857	98.240	2.14
QNR	0	94.543	–	12.087	–	7.82
	2	202.464	53.30	25.452	52.51	7.96
	40	2110.551	95.52	950.844	98.73	2.22
RLE	0	96.124	–	12.985	–	7.40
	2	219.795	56.267	28.193	53.942	7.55
	40	2207.519	95.646	985.698	98.68266	2.23
JPEG	0	307.831	–	42.022	–	7.36
	2	622.943	50.584	82.961	49.347	7.51
	40	6257.009	95.080	3084.881	98.638	2.03
Error-correction	0	197.557	–	34.767	–	5.68
	2	386.734	48.917	70.015	49.348	5.53
	40	3971.001	95.025	2603.490	98.386	1.52
Lambda	0	421.784	–	69.129	–	6.10
	2	791.150	46.687	121.791	43.240	6.49
	40	7406.187	94.305	3568.100	98.063	2.08
Omega-phy	0	487.543	–	80.937	–	6.02
	2	935.102	47.862	144.801	44.105	6.45
	40	8945.877	94.550	3978.451	97.97	2.25

achieved in the version with two checkers. This is due to the fact that the overhead of the checker invocation is less significant in the TLM descriptions than in the RTL descriptions. On the other hand, the insertion of the two checkers involves a significant simulation overhead with respect to the original version without checkers. The minimum speed-up has been observed in the version with 40 checkers. In this case, the checker overhead becomes largely dominant with respect to the IP functionality itself. As such, the simulation speed-up obtained by abstracting the RTL IP model into SystemC TLM is reduced proportionally to the number of checkers plugged into the model.

We expect that the achieved speed-up ends up being lower than the one that can be obtained by *manually* implementing a "higher-level" TLM description and consequently manually rewriting the assertions to be used with the new model. However, this double manual process would be time-consuming and error-prone. Conversely, the results obtained by using the proposed methodology have been achieved automatically reusing the already existing verification environment, without relying on any time-consuming manual transformation.

VI. CONCLUSIONS

This paper presented a methodology to reuse assertions originally defined for a given RTL IP, to verify the corresponding TLM model. The methodology applies to SystemC TLM models automatically generated from existing RTL IPs through any of the abstraction tools available in the commerce. The methodology consists of two automatic steps, in which assertions are firstly synthesized into C++ routines and then inserted in the SystemC TLM model. The experimental results have been conducted on benchmarks of different characteristics and complexity to show the applicability of the proposed methodology. The results show the simulation overhead caused by the automatic aspect of the methodology, which, in our opinion, is acceptable considering, as the alternative, the manual effort required to re-implement both the TLM model and the TLM assertions.

REFERENCES

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

[2] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Trans. on VLSI Systems*, vol. 14, no. 1, pp. 57–67, 2006.

[3] Carbon Design Systems. Carbon Model Studio. [Online]. Available: {http://carbondesignsystems.com/}

[4] EDALab. HIFSuite. [Online]. Available: "http://www.hifsuite.com/"

[5] N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic abstraction of RTL IPs into equivalent TLM descriptions," *IEEE Trans. on Computers*, vol. 60, no. 12, pp. 1730–1743, 2011.

[6] A. Mathur and V. Krishnaswamy, "Design for verification in system-level models and RTL," in *Proc. of IEEE/ACM DAC*, 2007, pp. 193–198.

[7] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva, "Towards equivalence checking between TLM and RTL models," in *Proc. of ACM/IEEE MEMOCODE*, 2007, pp. 113–122.

[8] M. Fujita, "Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths," *ACM TODAES*, vol. 10, no. 4, pp. 610–626, 2005.

[9] Y.Lahbib, M.-A. Ghrab, M. Hechkel, F. Ghenassia, and R. Tourki, "A new synchronization policy between PSL checkers and SystemC designs at transaction level," in *Proc. of IEEE DTIS*, 2006, pp. 85–90.

[10] M. Boulé and Z. Zilic, *Generating hardware assertion checkers: for hardware verification, emulation, post-fabrication debugging and on-line monitoring*. Springer, 2008.

[11] L. Ferro and Laurence, "ISIS: runtime verification of TLM platforms," in *Proc. of FDL*, 2009, pp. 1–6.

[12] N. Bombieri, F. Fummi, and G. Pravadelli, "Incremental ABV for Functional Validation of TL-to-RTL Design Refinement," in *Proc. of ACM/IEEE DATE*, 2007, pp. 882–887.

[13] W. Ecker, V. Esen, and M. Hull, "Implementation of a transaction level assertion framework in SystemC," in *Proc. of IEEE/ACM DATE*, 2007, pp. 894–899.

[14] D. Grosse, H. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *Proc. of IEEE/ACM MEMOCODE*, 2010, pp. 113–122.

[15] N. Bombieri, F. Fummi, G. Pravadelli, and A. Fedeli, "Hybrid, Incremental Assertion-Based Verification for TLM Design Flows," *IEEE Design and Test*, vol. 24, no. 2, pp. 140–152, 2007.

[16] W. Ecker, V. Esen, and M. Hull, "Execution semantics and formalism for multi-abstraction TLM assertions," in *Proc. of IEEE/ACM MEMOCODE*, 2006, pp. 93–102.

[17] Y. Lahbib, R. Kamdem, M.-l. Benalycherif, and R. Tourki, "An automatic ABV methodology enabling PSL assertions across SLD flow for SOCs modeled in SystemC," *Comput. Electr. Eng.*, vol. 31, no. 4-5, pp. 282–302, 2005.

[18] W. Ecker, V. Esen, and M. Hull, "Requirements and concepts for transaction level assertions," in *Proc. of IEEE ICCD*, 2006, pp. 286–293.

[19] A. Ghofrani, F. Javahery, and Z. Navabi, "Assertion based verification in TLM," in *Proc. of IEEE EWDTS*, 2010, pp. 509–513.

[20] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *Proc. of CAV*, 2000, pp. 538–542.

[21] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet, "Generation of test programs for the assertion-based verification of TLM models," in *Proc. of IEEE IDT*, 2008, pp. 237–242.

[22] M. Chen and P. Mishra, "Assertion-based functional consistency checking between TLM and RTL models," in *Proc. of IEEE VLSID*, 2013, pp. 320–325.

[23] A. Kasuya and T. Tesfaye, "Verification methodologies in a TLM-to-RTL design flow," in *Proc. of ACM/IEEE DAC*, 2007, pp. 199–204.

[24] Accellera, "Property Specification Language Reference Manual," http://www.accellera.org, 2004.

[25] "IEEE Standard SystemC Language Reference Manual," http:///ieeexplore.ieee.org, 2006.

[26] IBM. FoCs Property Checkers Generator. [Online]. Available: https://www.research.ibm.com/haifa/projects/verification/focs/start.html