Noname manuscript No.
(will be inserted by the editor)

Reusing RTL assertion checkers for verification of SystemC TLM models

Nicola Bombieri · Franco Fummi · Valerio Guarnieri · Graziano Pravadelli · Francesco Stefanni · Tara Ghasempouri · Michele Lora · Giovanni Auditore · Mirella Negro Marcigaglia

Received: date / Accepted: date

Abstract The recent trend towards system-level design gives rise to new challenges for reusing existing register-transfer level (RTL) intellectual properties (IPs) and their verification environment in transaction-level modeling (TLM). 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 under-explored, particularly when assertion-based verification (ABV) is adopted. Some 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, except by using transactors to create a mixed simulation between the TLM design and the RTL checkers corresponding to the assertions. However, the use of transactors may lead to longer verification time due to the need of developing and verifying the transactors themselves. Moreover, the simulation time is negatively affected by the presence of transactors, which slow down the simulation at the speed of the slowest parts (i.e., RTL checkers). This article proposes an alternative methodology that does not require transactors for reusing assertions, originally defined for a given RTL IP, in order to verify the corresponding TLM model. Experimental results have been conducted on benchmarks with different characteristics and complexity to show the applicability and the efficacy of the proposed methodology.

Keywords Assertion-based verification, transaction-level modelling, RTL abstraction.

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

Nicola Bombieri · Franco Fummi · Graziano Pravadelli · Francesco Stefanni
EDALab s.r.l., Italy
E-mail: name.surname@edalab.it

Nicola Bombieri · Franco Fummi · Tara Ghasempouri · Valerio Guarnieri · Michele Lora · Graziano Pravadelli
University of Verona, Italy
E-mail: name.surname@univr.it

Giovanni Auditore · Mirella Negro Marcigaglia
STMicroelectronics s.r.l., Italy
E-mail: name.surname@st.com

1 Introduction

Several frameworks have been proposed in the past years to deal with 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 RTL [1]. On the other hand, SystemC and TLM [2] have gained a broad consensus for system-level design and verification, architectural exploration and HW/SW co-simulation [3].

Such a language and paradigm heterogeneity in the today's design flows has led to the fact that IP models could be available at different abstraction levels. This increases IP reuse and integration among different projects at RTL and TLM, ideally reducing design and verification time, particularly when an existing RTL model can be reused in a new TLM context. However, the actual degree of IP reusability heavily depends on the design and verification environment adopted by the designers, which could require a significant manual effort to plug the original model in the new abstraction level. The risk relies on the fact that an IP is 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

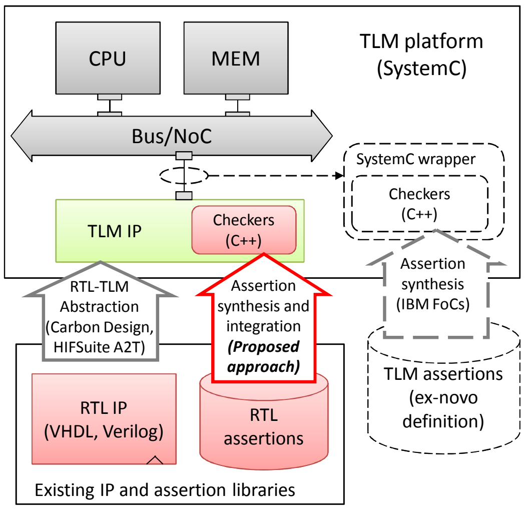

CPU
MEM
TLM platform (SystemC)
Bus/NoC
SystemC wrapper
Checkers (C++)
TLM IP
Checkers (C++)
Assertion synthesis (IBM FoCs)
RTL-TLM Abstraction (Carbon Design, HIFSuite A2T)
Assertion synthesis and integration (Proposed approach)
TLM assertions (ex-novo definition)
RTL IP (VHDL, Verilog)
RTL assertions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

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

it 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.

Methodologies and tools for the automatic generation of SystemC TLM models starting from existing RTL IPs have been recently proposed [4,5,6], and they 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 verification 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. For this reason, 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 [7,8,9]. In contrast, some simulation-based techniques [10, 11,12,13] and frameworks [14,15,16] have been proposed to allow designers to adopt 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 static or dynamic-based *decision procedure* to check the consistency between such assertions and the design under verification (DUV). Model checking represents the main static approach to verify the consistency between assertions and RTL designs [17]. However, due to its complexity, model checking is generally adopted for verification of small, safety-critical components, rather than for the whole DUV. Alternatively, a non-exhaustive but less expensive, semi-formal approach is represented by dynamic ABV. In this case, assertions are converted into *checkers* [11], i.e., components that monitor **observable signals** of the DUV during simulation and raise a failure signal when counterexamples are found for the corresponding assertions (see right-most side of Figure 1).

Dynamic ABV relying on checkers has been extensively applied to verify RTL models, where the trigger mechanism to evaluate checkers is guaranteed by the presence of a clock signal. In contrast, the application of dynamic 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 [15]. Indeed, TLM lacks a synchronous timing reference that precisely identifies evaluation points for checkers.

The techniques recently proposed to apply dynamic ABV at TLM can be classified into two categories: approaches that define a way to specify temporal assertions and that suppose the presence of an event-based triggering mechanism for checkers [18], and approaches that opportunely synchronize checker activation and DUV simulation [10]. Despite of their technical differences, all these works assume a top-down design and verification flow, where assertions definition is initially carried out at TLM level and then refined towards RTL implementations. This requires, in case of bottom-up flows based on RTL IP reuse, verification engineers to redefine an ex-novo set of TLM assertion 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 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. **As a first step towards the automatic abstraction of RTL assertions at TLM, this work focuses on TLM cycle-accurate models. In particular, this work well applies when automatic tools for RTL-to-TLM abstraction are used. An extension of this work to support approximately-timed, loosely-timed and untimed TLM models, which may be modelled by hand, is part of our current and future work.**

Experimental results have been conducted on different benchmarks and several RTL assertions have been synthesized into checkers and plugged in to the system platform. The results confirm 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 article is organized as follows. Section 2 presents a more accurate analysis of the related work. Section 3 introduces the most important concepts of ABV and RTL-to-TLM abstraction for a better understanding of the proposed methodology. Section 4 presents the methodology. Section 5 reports the experimental results, while Section 6 is devoted to conclusions and remarks.

2 Related works

The problem of applying ABV at TLM has been investigated first for cycle-accurate TLM models [3,19]. In [3] 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 [19], 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 [18,20], general concepts and requirements related to the use of dynamic ABV at TLM are defined for the specific case of TLM 1.0. Verification of TLM 1.0 models is proposed also in [21], which defines a library of assertions to allow self-checking of TLM channels, and in [22], where a set of assertion primitives to handle the temporal logic beyond the cycle accurate level is defined. The last approach offers interesting mechanisms to construct assertions synchronized with events. Nevertheless, callbacks should be inserted in the original SystemC code to report event occurrences.

Synchronization policies between assertion checkers and DUV have been proposed also in [23], where an event-based synchronization mechanism is assumed instead of the traditional clock-based approach adopted at RTL. This approach is supported by a specific assertion language that allows to define assertions independently from the abstraction level. A SystemC implementation of an ABV framework that relies on such a language is then described in [14]. Assertions written by using this language are then compiled and translated into SystemC modules. Transactions between two modules are detected by proxy monitors such that assertion modules receive events from these monitors and generate assertion results. Even if this approach is interesting, it requires to redefine assertions by using a new specific language, which could be unappealing for verification engineers that are used to adopt standard ABV languages like SystemVerilog Assertion (SVA) or Property Specification Language (PSL).

A different ABV framework for SystemC TLM verification is presented in [24]. The framework, implemented in C++, takes PSL assertions and supports all coding styles of standard TLM 2.0. Unfortunately, this approach requires to instrument the code of the DUV.

Few modifications in the original SystemC code are required also by the technique proposed in [25] and implemented in a tool called Horus. The authors use a model that allows to observe the transactional events in the system and to trigger the monitors at appropriate instants according to the *observer* design pattern. Evolution of this approach aiming at automatic generation of checkers suited to perform dynamic ABV at TLM are also presented in [26,12,27,28]. In [26] the authors present a methodology that enables the dynamic verification of temporal assertions for TLM specifications by checking the validity of PSL assertions that express properties on communications. In [12] a corresponding prototype tool, called ISIS, is described. The work in [27] is devoted to present a formal, operational semantics of PSL endowed with the modeling layer of PSL that has been implemented in ISIS. Finally, the contribution reported in [28] is intended to support reentrant assertions (i.e., different instances of a same assertion evaluated on overlapping evaluations cycles), through the use of multiple checker instances, with local variables.

Approaches that do not require modifications of the original SystemC code are presented in [29,30], where aspect-oriented mechanisms are exploited to write temporal assertions that fit TLM 2.0 requirements. Functional as well as performance assertions are addressed.

A different synchronization policy between PSL checkers generated by using IBM FoCs [31] and SystemC TLM designs is presented in [10], where checkers are evaluated at the starting of each SystemC transaction.

A methodology to check the functional consistency between TLM and RTL models is instead proposed in [32], where the reuse of TLM assertions at RTL is guarantee by ad-hoc refinement rules.

Finally, a formal tool for assertion checking of TLM SystemC descriptions is proposed in [15]. 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.

Reuse of ABV properties in TLM-based design flows has been addressed in [13,16,33]. In particular, [13,16] present techniques to reuse TLM properties at RTL through TLM/RTL transactors. Instead, [33] presents a methodology to check the functional consistency between TLM and RTL models by reusing TLM properties

at RTL through ad-hoc refinement rules. All these techniques assume a top-down design and verification flow, where properties are defined ex-novo at TLM level, and then reused at RTL.

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 intellectual property (IP)-cores are plugged into SystemC TLM system platforms. To the best of our knowledge, the only approach that partially supports the reuse of RTL assertions at TLM has been presented in [34]. [34] proposes to adopt *transactors* to allow a TLM-RTL mixed simulation. A transactor works as a translator from a TLM function call to an RTL sequence of statements and vice versa, 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. In this way, checkers corresponding to assertions defined at RTL can be connected to a TLM model through opportune transactors. However, the implementation of the transactor and its verification could be compared to rewriting assertions at TLM, from the point of view of the development time. Some semi-automatic approaches have been proposed for transactor generation [35], but a complete automatic tool has never been implemented. Anyway, simulation time is negatively affected by the presence of transactors that slow down the simulation at the speed of the slowest (i.e., RTL) parts (i.e., checkers).

3 Assertion-based verification in TLM

This section firstly summarizes the preliminary concepts related to PSL [36], one of the most widespread language for specification of temporal assertions. Then, the most important notions related to RTL-to-TLM abstraction are presented to better understand the assertion reuse technique proposed in the following sections.

3.1 PSL assertions and assertion checkers

PSL is nowadays one of the most prominent standards for formalizing specifications into assertions. Based on the *Sugar language* by IBM, PSL has been proposed by the Accellera consortium as a specification language to define assertions with a concise syntax and a clearly-defined formal semantics. PSL shows many similarities with respect to 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. Assertions written in PSL can be seen as an executable documentation for hardware and embedded software design.

PSL is an extension of the standard temporal logics Linear Time Temporal Logic (LTL) and Computation Tree Logic (CTL). PSL assertions are built upon four layers which cooperate to guarantee the expressiveness of the language:

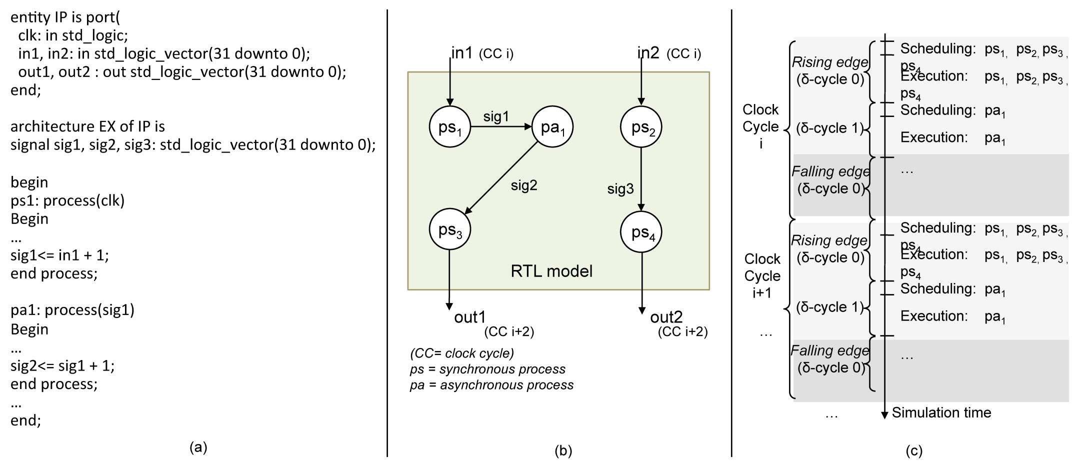
– *Boolean layer:* it is adopted to build basic expressions commonly used by the other layers.
– *Temporal layer:* it can be considered as the core of the language since it gives the possibility of describing temporal relations, which are verified over a set of evaluation cycles.
– *Verification layer:* it provides the directives for using assertions during a verification run.
– *Modeling layer:* it can be used to characterize the behavior of design inputs and to model auxiliary variables representing the environment where the DUV lives.

To use PSL assertions for (dynamic) simulation-based verification, it is necessary to map them into executable specifications, i.e., assertion checkers, or simply *checkers*, that must be connected to the DUV to monitor its behavior. Checkers can be generated only from assertions compliant with the *simple subset* of PSL, which conforms to the notion of monotonic advancement of time, and it is close to the concept of simulation itself.

In the past, checkers were manually written and embedded within the system description. However, this was a time-consuming and error-prone task. Thus, much effort has been recently spent to make the checker synthesis from formal specifications automatic. The most prominent technique to generate checkers from assertions is based on automata [31, 11,37]. The simple subset of PSL subsumes the LTL by introducing *regular expressions* and syntactic sugar. PSL expressiveness is equivalent to *omega-regular languages*, which are recognized by Büchi automata. Consequently, the internal implementation of a checker strongly resembles the structure of the automaton that recognizes the formula expressed in the assertion. To guarantee the evolution of the automaton during simulation, the checker is interfaced with the DUV. The actual implementation of the interface depends on the target language, which is generally VHDL, Verilog or SystemC/C++. From the user point of view, a checker can be considered as a function invoked periodically over the DUV I/O signals.

3.2 RTL-to-TLM abstraction

At the state of the art, *Carbon Studio* [4] and *HIFSuite A2T* [5] are the main important and widespread tools for automatic RTL-to-SystemC TLM abstraction. Despite technical differences, the abstraction process implemented by the



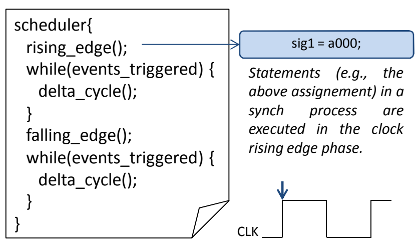
```
entity IP is port(
  clk: in std_logic;
  in1, in2: in std_logic_vector(31 downto 0);
  out1, out2 : out std_logic_vector(31 downto 0);
end;

architecture EX of IP is
signal sig1, sig2, sig3: std_logic_vector(31 downto 0);

begin
ps1: process(clk)
Begin
...
sig1<= in1 + 1;
end process;

pa1: process(sig1)
Begin
...
sig2<= sig1 + 1;
end process;
...
end;
```

Fig. 2 Dynamic scheduling overview: RTL model example (a), the corresponding graph of process syntchronization and communication (b), and the process scheduling over simulation time (c).

tools involves two main aspects, namely the abstraction of the I/O interface and the abstraction of the IP functionality. Concerning the abstraction of the communication protocol, a dedicated C++ data structure is created to store a field for every port of the original RTL model. Such a data structure is used to provide input data to the abstracted design and to retrieve output data from it. To properly model RTL port bindings at TLM, C++ pointers are exploited to introduce a data sharing mechanism, which mimics the port binding behavior typically featured in Hardware Description Languages (HDLs).

From the functionality point of view, the SystemC TLM code is obtained by translating hardware description language (HDL) statements into SystemC statements and by handling the RTL concurrency through a *dynamic scheduling* routine, which reproduces the behavior of the RTL process scheduler.

Consider, as a simple example, the IP block in Figure 2. Figure 2(a) shows the VHDL code of the IP block. Figure 2(b) represents the RTL IP model of such a block through a graph, each process being a vertex and each signal being an oriented edge. The graph represents the synchronization and communication net among processes. The RTL IP block consists of four synchronous processes (ps1-ps4), one asynchronous processes (pa1), two input ports (in1, in2), two output ports (out1, out2), and internal signals (sig1- sig3). Figure 2(c) represents the corresponding process execution order, by underlining update and evaluate steps, simulation cycles, and delta cycles. In dynamic scheduling, the RTL processes are activated whenever an event to which they are sensitive occurs. Simulated time granularity equals one clock period when the generated TLM model is cycle-accurate. Synchronous processes are firstly run on the rising event of the corresponding clock. Then, asynchronous processes, sensitive to events triggered by previously executed synchronous



```
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.
CLK

Fig. 3 Overview of the SystemC TLM scheduling activity.

processes, are executed. This last routine repeats until no further event is triggered. Each of these iterations corresponds to a *delta cycle*, which is a simulation cycle where simulated time does not advance [38]. The same procedure is then executed with respect to the falling edge of the clock. When delta cycles have finished, the simulation advances to the next clock cycle and the simulated time is updated.

According to such a simulation scheduling, Figure 3 provides a visual description of the scheduler activity for the cycle accurate TLM model generated starting 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 synchronized to the falling edge of the clock.

To properly manage the process synchronization, the RTL signals are converted into a pair of corresponding TLM variables having the same type as the signal. One variable stores the current value of the signal during the simulation of a delta cycle, while the other contains the new value that will be updated at the end of the delta cycle. The use of a pair of variables is required to properly implement, at TLM, the deferred assignment mechanism featured by RTL signals.

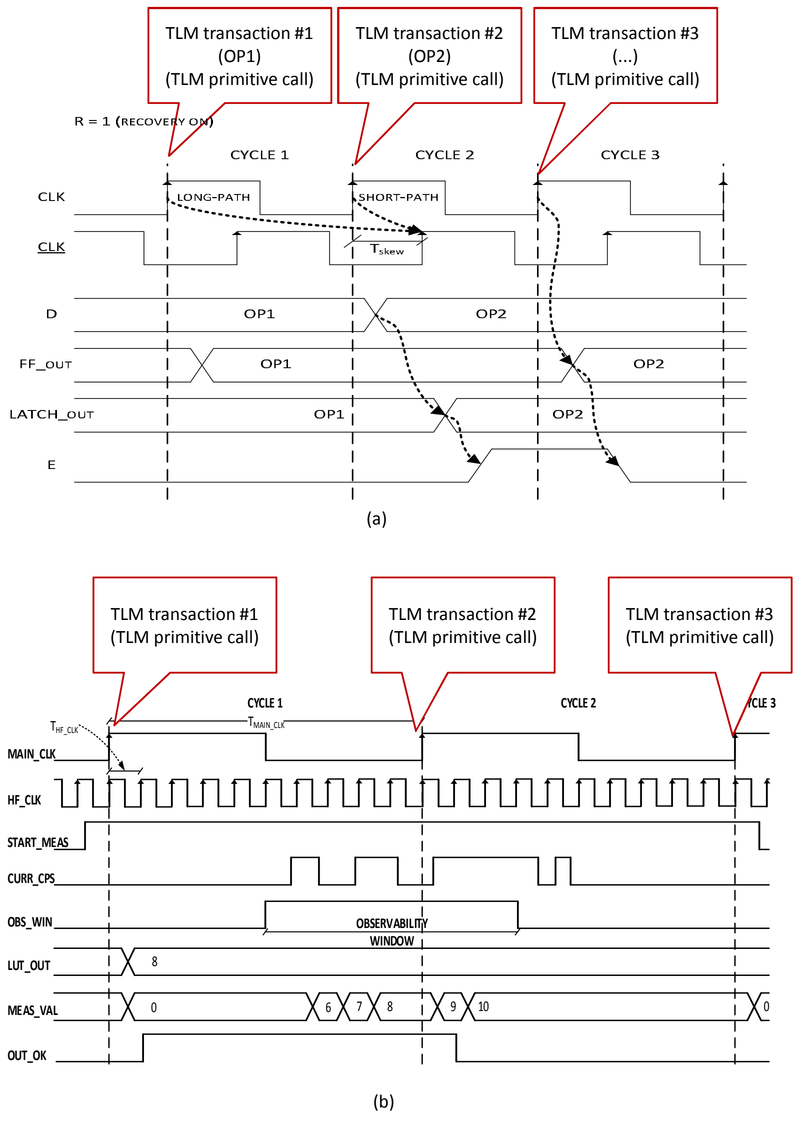

TLM transaction #1 (OP1) (TLM primitive call)
TLM transaction #2 (OP2) (TLM primitive call)
TLM transaction #3 (...) (TLM primitive call)
R = 1 (RECOVERY ON)
CYCLE 1
CYCLE 2
CYCLE 3
CLK
CLK
LONG-PATH
SHORT-PATH
T skew
D
FF_OUT
LATCH_OUT
E
OP1
OP2
(a)
TLM transaction #1 (TLM primitive call)
TLM transaction #2 (TLM primitive call)
TLM transaction #3 (TLM primitive call)
T HF_CLK
T MAIN_CLK
MAIN_CLK
HF_CLK
START_MEAS
CURR_CPS
OBS_WIN
OBSERVABILITY WINDOW
LUT_OUT
MEAS_VAL
OUT_OK
(b)

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

C++ assignments are in fact immediate, and do not support the typical delay mechanism of HDL concurrent processes. Since functions corresponding to processes executing in the same delta cycle are invoked sequentially by the main scheduler function, a separate variable is required to store the new value of the signal. The code of the process functions is modified so that every write operation on a signal has the new value variable on the left-hand side. This prevents the alteration of the current value being read by all other processes executing in the same delta cycle.

4 Methodology

Independently from the approach/tool adopted to abstract RTL IPs towards the TLM, we assume that the generated SytemC TLM models are accurate enough to guarantee the simulation of timing delays, i.e, only TLM cycle accurate models compliant with the scheduling policy described in Section 3.2.

Consider, for example, two cycle accurate RTL models that implement delay sensors, whose waveforms are shown in Figure 4. The first sensor (Figure 4(a)) relies on the Razor flip-flop (FF) [39]. The sensor aims at enhancing FFs of IP critical paths by introducing a *shadow latch* that samples the FF input data on the negative level of the delayed clock signal *$\underline{CLK}$*. Since *$\underline{CLK}$* is delayed by half *CLK* period ($\frac{T_{CLK}}{2}$), the Razor working time window is bounded by the rising and falling edge of CLK. If the values contained by FF and by the shadow latch differ, an error signal *E* is asserted to notify the timing failure. When the control signal *R* is high, the recovery mechanism is executed and the error in the faulty FF is corrected. The correction feature can be selectively ac-

tivated on each modified Razor FF acting on the corresponding signal *R*.

The second sensor relies on a simple counter to measure the propagation delay on IP critical paths. Compared to the Razor FF, it provides an absolute measure of delay rather than a timing failure detection. Using an additional clock (i.e., HF_CLK) with higher frequency multiple of the clock frequency of the IP (i.e., CLK), the monitor enumerates the amount of HF_CLK periods elapsed for the signal propagation from the path start point to the path end point. The measurement is performed during a predefined time window called *observability window* (i.e., OBS_WIN) where all signal transitions are captured. The position in time and width of OBS_WIN are chosen at design time according to the expected time interval where signal transitions may occur. Two registers store the counter value on the occurrence of both rising and falling transitions. The delay measure is then selected according to the last captured transition. A control block compares the obtained value with reference values determined at design time.

The two cycle-accurate examples differ for the number of clock signals (i.e., one clock signal for the Razor FF, two clock signals for the counter-based sensor). In this context, the methodology we propose to reuse RTL assertions at TLM applies to two different scenarios:

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

We explored two ways of reusing RTL assertions at TLM for both scenarios. These approaches rely on the capability of checker generators (e.g., IBM FoCs) of synthesizing an assertion into an automaton that can be modeled either by a set of RTL processes described through traditional HDL languages (e.g., VHDL, Verilog) or by a set of C++ functions.

1. *Abstraction of RTL HDL checkers.* In this case, PSL assertions defined for the RTL IP are first converted into RTL checkers (see Figure 5(a), step 1). Checker's behavior is implemented through HDL processes that describe a synchronous state machine modeling the semantics of the original PSL assertions. Such kind of checkers (which are called *monitors* in literature) can be directly integrated into the RTL IP model (step 2). The reuse of RTL assertions at TLM is then obtained by abstracting the RTL IP model extended with RTL checkers towards a SystemC TLM implementation, by using any automatic RTL-to-TLM tool (step 3).
2. *Generation of C++ checkers.* The second approach consists of generating C++ checkers (see Figure 5(b), step 1) that can be directly integrated into a SystemC TLM IP model. In this case, only the RTL IP is abstracted from RTL to SystemC TLM (step 2). On the contrary, C++ checkers do not required to be abstracted since they are opportunely invoked by the SystemC scheduler according with the TLM cycle accurate simulation semantics (step 3).

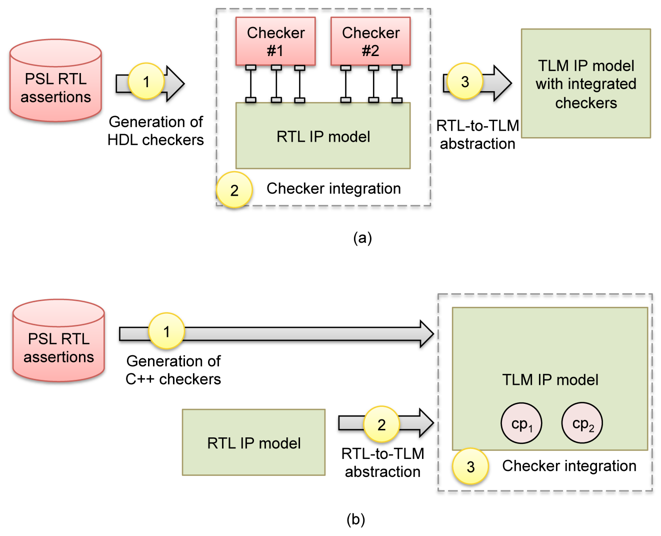


Fig. 5 Alternatives to reuse RTL assertions at TLM: The generation of HDL checkers and the integration phase before the RTL-to-TLM abstraction (a), and the generation of C++ checkers and the integration phase in the abstracted TLM IP model (b).

Both alternatives are automatic. The first one (Figure 5(a)) is straightforward, since it consists of applying existing tools in cascade (e.g., IBM FoCS for HDL monitor generation and HIFSuite A2T for RTL-TLM abstraction). The second one (Figure 5(b)) still consists of applying two existing tools but, since the monitors are generated in C++ rather than HDL, it requires manipulating the automatically generated SystemC TLM code to opportunely invoke C++ checkers. The methodology proposed in this paper adopts the second alternative, since it guarantees better simulation performance. To understand the reason, we have to better detail the three steps of the methodology, which are explained in Sections 4.1, 4.2 and 4.3. The comparison of the two alternatives will be discussed in Section 4.4.

4.1 Step 1: generation of C++ checkers

In the first phase, a checker generator is applied to automatically generate run-time C++ checkers from a starting set of assertions defined for the RTL model[1].

Figure 6.1 on the top depicts an example of a PSL RTL assertion, *P1*. It asserts, globally, whether *A* or *B* is always followed in the next clock cycle by *C*. The $@-clause$ **at the end of the assertion specifies the RTL events in which the property must be checked (in our example, the rising edge of the clock). Given *P1*, the checker generator produces a C++ code (`P1(){..}`, in the example) that implements the automaton corresponding to the property semantics. To verify property *P1* during simulation, the C++ checker must be called exactly in the events specified by the @-clause of the original PSL assertions.**

In general, the checker code is composed of two routines. The first must be invoked at every event specified in the @-clause of the original PSL assertion (e.g., the rising edge of the clock). This allows the checker to evolve through the automaton and to assess the assertion (true or false). The second routine must be called whenever the abort condition of the assertion occurs.

4.2 Step 2: IP abstraction

In the second phase, the RTL model is abstracted into SystemC TLM (see Figure 6.2). The RTL model consists of a number of synchronous and asynchronous processes, and its semantic relies on the scheduling of the RTL processes. During simulation, the processes are woke up only if necessary. Synchronous processes wake up at each clock cycle. Asynchronous processes wake up if an event to which they are sensitive has occurred. During abstraction, RTL processes are translated into C++ functions, which are scheduled exactly in the same way as at RTL by a *scheduler*, which is embedded in the SystemC TLM code (see Section 3.2).

The C++ checkers derived from step 1 are integrated among the functions implementing the IP functionality (Figure 6.3) and invoked by scheduler as explained in the following Section.

4.3 Step 3: integration of checkers at TLM

After the 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 6.3) that is responsible for carrying out the design functionality in the TLM description.

[1] It is worth noting that the proposed methodology is independent from the checker generator employed in this step.

Since the process scheduler in the abstracted description distinguishes between synchronous functions and asynchronous functions, the checkers invocations are inserted in the opportune 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 a 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.

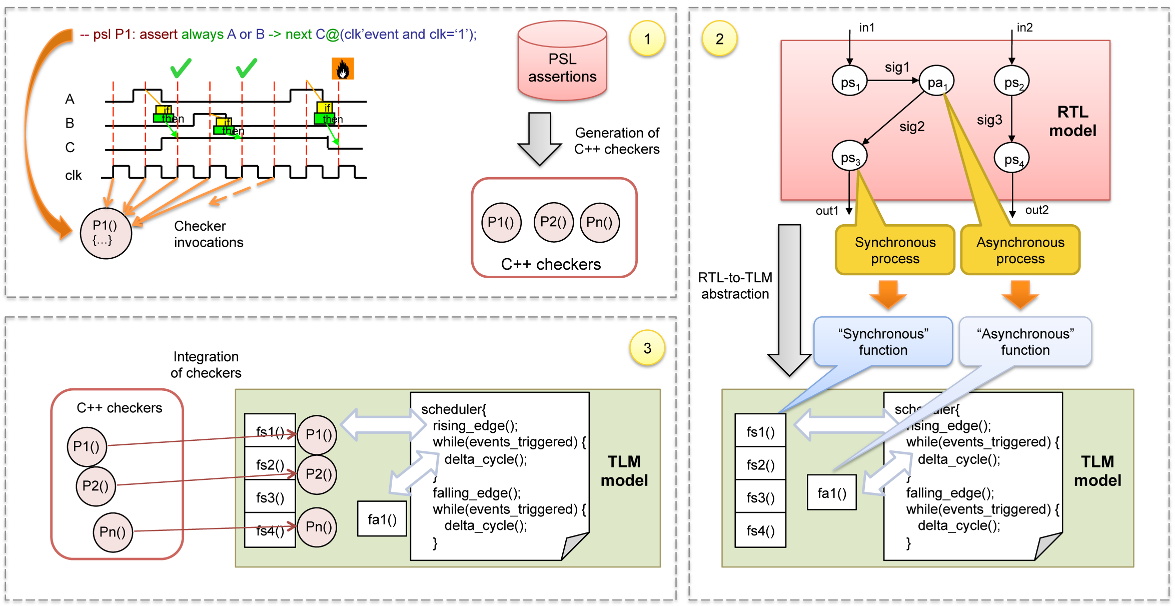


Fig. 6 The three steps of the methodology: the checker generation through synthesis with an example (a), the RTL-to-TLM abstraction with an example (b), and the checker integration in the SystemC TLM model

4.4 HDL vs C++ checkers integration

The two alternatives for reusing RTL assertions reported in Figure 5 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 TLM dynamic scheduler whenever the corresponding event specified in the @-clause occurs in both alternatives. 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 3.2).

However, alternatives number 2, which has been described in details in Sections 4.1, 4.2 and 4.3 offers the following advantages over alternative number 1, which relies on 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.

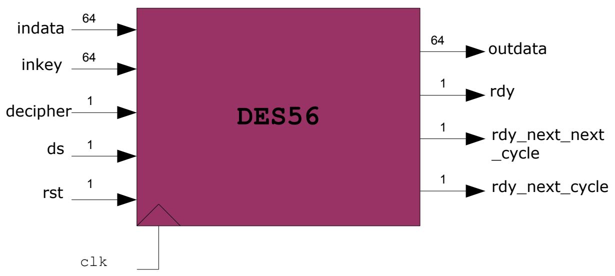
5 Experimental results

The methodology presented in this article has been applied to different VHDL IP blocks: a DES56 cryptographic module, a ColorConverter model, which transforms an image from the RGB format to the YCbCr601 format, 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 1 reports their main characteristics in terms of number of synchronous and asynchronous processes, number of lines of code (*loc*), number of pipeline stages, and latency in clock cycles.

Table 1 Characteristics of the RTL IPs.

Design	Processes		RTL	Pipeline	Latency
	Async.	Sync.	loc	stages	(cc)
DES56	20	6	2,022	–	17
ColorConverter	12	3	1,454	8	8
UART	416	77	5,866	–	16
Root	2	0	343	–	16
Div	1	5	1,283	–	16
QNR	7	17	518	16	16
RLE	14	17	678	9	9
FDCT	259	196	5,935	388	67
JPEG	281	231	18,381	80	80
Error Correction	6	11	1,666	–	130
Lambda Root	0	5	1,092	–	790
Omega Phy	17	4	1,595	–	294

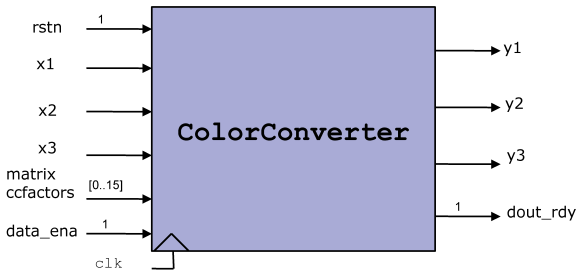
The RTL SystemC models have been obtained by using the HDL conversion tools provided by HIFSuite [5], while the **corresponding SystemC TLM cycle-accurate models have been generated by the HIFSuite's A^{2}T back end**.



- **indata**: the data that will be encrypted
- **inkey** : the key that will be used to encrypt the data
- **decipher**: the direction of the encryption , 0 => encrypt / 1 => decrypt
- **ds**: the data-valid flag, it is enabled if we want that the encryption starts
- **clk**: the clock signal
- **rst**: the reset flag, necessary for setup a known situation

- **outdata**: the results of the encryption
- **rdy**: flag that will be enabled when the module ends encryption
- **rdy_next_next_cycle**: flag that will be enabled when the module is near to end encryption , missing 2 cycles
- **rdy_next_cycle**: flag that will be enabled when the module is near to end encryption , missing 1 cycle

Fig. 7 The structure of the DES56 RTL model.



- **rstn:** reset the registers. It is necessary before start the conversion.
- **[x1..x3]** : the image matrix values that will be converted.
- **matrix ccfactors**: matrix coefficients needed to perform the conversion.
- **[y1..y3]**:the image conversion output values
- **data_ena:** to indicate that input data are ready
- **dout_rdy:** to indicate that output data are ready.

Fig. 8 The structure of the ColorConverter RTL model.

As a starting point, different PSL assertions have been provided for each RTL IP model. Some examples of PSL assertions applied for verifying the DES56 RTL model (see Figure 7), and then reused at TLM, are the following. **The *always* operator at the beginning of each assertion means that its right operand holds globally.**

1. $always\ (ds = 1 \wedge indata \neq 0 \wedge next(ds = 0)) \rightarrow next[17](rdy = 1 \wedge outdata \neq 0)@clk\_pos$;
   "if not null data ($indata \neq 0$) are ready ($ds = 1$ and at the next clock cycle $ds = 0$) on the input port, then a not null result ($outdata \neq 0$) should be ready ($rdy = 1$) on the output port after 17 clock cycle".
2. $always\ (ds \rightarrow next[17](!ds))\ until\ rdy@clk\_pos$;
   "until result is ready (rdy), if the ds flag denotes that data are ready on the inputs, then after 17 cycles the ds flag should be 0".
3. $always\ (ds \wedge indata = 0) \rightarrow next[17](out\,! = 0)@clk\_pos$;
   "if zero is provided to the input port ($indata = 0$), then the encrypted data provided after 17 clock cycles should be not null ($next[17](out\,! = 0)$)".

Some examples of PSL assertions applied for verifying the ColorConverter model (see Figure 8) are the following:

1. $always\quad next[8](dout\_rdy) until!(data\_ena)$;
   "the enabling flag for the output data $dout\_rdy$ will be high in 8 clock cycles, and will stack high until the enabling flag $data\_ena$ for the input data is high".
2. $always\quad data\_ena \rightarrow next[8]dout\_rdy$;
   "if the enabling flag of the input data $data\_ena$ is high, then the enabling flag for output data will be hight (result is ready) after 8 clock cycle".

The PSL assertions have been synthesized into cycle accurate C++ checkers through IBM FoCs [40]. These checkers can be integrated into both SystemC RTL and SystemC TLM cycle accurate models of IPs under verification. This is possible because both the TLM cycle accurate model and the RTL model of an IP evolve according to the same timing reference (i.e., both are cycle accurate). Thus, the C++ automata implemented in the checkers synthesized by FoCs work properly at RTL as well as at TLM cycle accurate. The only difference relies on the way C++ checkers are invoked during RTL and TLM simulation. At RTL, FoCs checkers have been wrapped inside *sc_methods* which run concurrently with respect to RTL processes. At TLM, the mechanism described in Section 4.3 has been adopted.

To evaluate the simulation overhead caused by the introduction of the checkers, three different contexts have been tested for both the RTL and TLM IP models. The first consists of the IP models without any checker. This allows us to estimate the reference speed-up due to the RTL-to-TLM abstraction. The second and third contexts represent the IP models with a few and many checkers, respectively (in particular, two and forty checkers) to evaluate the overhead caused by a different amount of inserted checkers on traditional RTL simulation and when they are reused at TLM as proposed in this article. The set of checkers integrated in both the RTL and TLM models is the same. Table 2 reports the obtained results. For each design, column *Checkers* identifies the context (*i.e.*, with 0, 2 or 40 checkers), columns RTL and TLM report the execution time (in seconds) employed by the simulation. Columns *Overhead* report the overhead on the simulation time caused by the integration of the checkers, in percentage with respect to the version without checkers. Fi-

Table 2 Experimental results.

Design	Checkers (#)	RTL (s)	Overhead (%)	TLM (s)	Overhead (%)	Speed-up (x)
	0	42.48	–	20.18	–	2.11
DES56	2	100.28	136.06	23.50	16.45	4.27
	40	989.34	2,228.95	312.35	1,447.82	3.17
	0	112.12	–	18.76	–	5.98
ColorConverter	2	209.61	86.95	23.31	24.25	8.99
	40	1,072.11	856.22	378.01	1,914.93	2.84
	0	24.19	–	11.61	–	2.08
UART	2	54.69	126.13	23.71	104.22	2.31
	40	588.71	2,334.08	458.94	3,852.97	1.28
	0	22.75	–	19.94	–	1.14
Root	2	97.44	328.32	37.00	85.55	2.63
	40	1,422.16	6,151.53	1,203.10	5,933.60	1.18
	0	46.03	–	20.79	–	2.21
Div	2	125.43	172.51	23.05	10.87	5.44
	40	1,528.48	3,220.83	665.41	3,101.40	2.30
	0	105.59	–	18.57	–	5.69
FDCT	2	209.65	98.55	34.58	86.24	6.06
	40	2,250.88	2,031.78	1,054.86	5,581.66	2.13
	0	94.54	–	12.09	–	7.82
QNR	2	202.46	114.15	25.45	110.57	7.95
	40	2,110.55	2,132.37	950.84	7,766.67	2.22
	0	96.12	–	12.99	–	7.40
RLE	2	219.80	128.66	28.19	117.12	7.80
	40	2,207.52	2,196.53	985.70	7,491.11	2.24
	0	307.83	–	42.02	–	7.33
JPEG	2	622.94	102.37	82.96	97.42	7.51
	40	6,257.01	1,932.61	3,084.88	7,241.11	2.03
	0	197.56	–	34.77	–	5.68
Error-correction	2	386.73	95.76	70.02	101.38	5.52
	40	3,971.00	1,910.05	2,603.49	7,388.39	1.53
	0	487.54	–	69.13	–	7.05
Lambda	2	791.15	62.27	121.79	76.18	6.50
	40	7,406.19	1,419.08	3,568.10	5,061.51	2.08
	0	487.54	–	80.94	–	6.02
Omega-phy	2	935.10	91.80	144.80	78.91	6.46
	40	8,945.88	1,734.89	3,978.45	4,815.49	2.25

nally column *speed-up* reports the simulation speed-up between the RTL and TLM implementations.

In general, we observed that the checker integration affects the RTL and TLM execution times in a different way, even if the checkers are the same. In particular, the presence of checkers affects linearly both the RTL and TLM execution time (i.e., the overhead introduced by the checkers increases linearly with the number of inserted checkers). However, the overhead over the number of inserted assertions increases more rapidly at TLM than RTL. In the RTL IP simulation, with few checkers, the event-driven execution of the IP core dominates the event-driven execution of the checkers. On the other hand, by considering the RTL IP code and the checker code, the ratio between the scheduling events raised and the code complexity of IP core and checkers is comparable. As a consequence, the event-driven execution of the checkers becomes dominant over the IP when the amount of code implementing checkers is larger than the IP code. In the TLM IP simulation, with few checkers, the execution of the IP core (which relies on fewer scheduling events than at RTL) is also dominant over the checkers. Nevertheless, by increasing the number of checkers, the event-driven execution of the checkers becomes dominant over the IP execution sooner (with less checkers) than at RTL.

This behavior has a direct impact on the RTL vs. TLM simulation speedup. By comparing the simulation times between RTL and TLM implementations, starting with no checkers and integrating an increasing number of checkers, we observed that the RTL-TLM speedup initially improves (i.e., by moving from no checkers to two checkers), while it linearly decreases as the number of checkers increases. In general, the RTL-TLM speedup is preserved to a minimum of 2x when the number of checkers is significantly high (i.e., around forty). In two cases (UART, Root) we observed the speedup being canceled (i.e., around 1x). This is due to the fact that, in those

cases, also the initial speedup (IPs with no checkers) is very low.

We expect that, in each context, 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 re-writing 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.

6 Conclusions

This article presented a methodology to reuse assertions, originally defined for an 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 with different characteristics and complexity to show the applicability of the proposed methodology. The results show that the methodology finds the best applications whenever the number of assertions reused at TLM are limited (10-15) per IP, which, in our opinion, could be enough in several cases. The results also underline 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.

The methodology can be applied only for cycle-accurate TLM descriptions automatically generated according to the scheduling policy described in Section 3.2. On the contrary, approximately timed, loosely timed and untimed TLM models are currently not supported.

References

1. D. J. Smith. *VHDL & Verilog compared & contrasted - plus modeled example written in VHDL, Verilog and C*. In *Proc. of ACM/IEEE DAC*, pp. 771–776. 1996.
2. L. Cai and D. Gajski. *Transaction Level Modeling: An Overview*. In *IEEE/ACM CODES+ISSS*, pp. 19–24. 2003.
3. A. Habibi and S. Tahar. *Design and verification of SystemC transaction-level models*. IEEE Trans. on VLSI Systems, vol. 14(1):pp. 57–67, 2006.
4. Carbon Design Systems. *Carbon Model Studio*. http://carbondesignsystems.com/.
5. EDALab. *HIFSuite*. "http://www.hifsuite.com/".
6. N. Bombieri, F. Fummi, and G. Pravadelli. *Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions*. IEEE Trans. on Computers, vol. 60(12):pp. 1730–1743, 2011.
7. A. Mathur and V. Krishnaswamy. *Design for Verification in System-level Models and RTL*. In *Proc. of IEEE/ACM DAC*, pp. 193–198. 2007.
8. N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. *Towards Equivalence Checking Between TLM and RTL Models*. In *Proc. of ACM/IEEE MEMOCODE*, pp. 113–122. 2007.
9. M. Fujita. *Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths*. ACM TODAES, vol. 10(4):pp. 610–626, 2005.
10. 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*, pp. 85–90. 2006.
11. M. Boulé and Z. Zilic. *Generating hardware assertion checkers: for hardware verification, emulation, post-fabrication debugging and on-line monitoring*. Springer, 2008.
12. L. Ferro and Laurence. *ISIS: runtime verification of TLM platforms*. In *Proc. of FDL*, pp. 1–6. 2009.
13. N. Bombieri, F. Fummi, and G. Pravadelli. *Incremental ABV for Functional Validation of TL-to-RTL Design Refinement*. In *Proc. of ACM/IEEE DATE*, pp. 882–887. 2007.
14. W. Ecker, V. Esen, and M. Hull. *Implementation of a transaction level assertion framework in SystemC*. In *Proc. of IEEE/ACM DATE*, pp. 894–899. 2007.
15. D. Grosse, H. Le, and R. Drechsler. *Proving transaction and system-level properties of untimed SystemC TLM designs*. In *Proc. of IEEE/ACM MEMOCODE*, pp. 113–122. 2010.
16. N. Bombieri, F. Fummi, G. Pravadelli, and A. Fedeli. *Hybrid, Incremental Assertion-Based Verification for TLM Design Flows*. IEEE Design and Test, vol. 24(2):pp. 140–152, 2007.
17. E. M. Clarke, E. A. Emerson, and J. Sifakis. *Model Checking: Algorithmic Verification and Debugging*. Commun. ACM, vol. 52(11), 2009.
18. W. Ecker, V. Esen, and M. Hull. *Execution semantics and formalism for multi-abstraction TLM assertions*. In *Proc. of IEEE/ACM MEMOCODE*, pp. 93–102. 2006.
19. 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(4-5):pp. 282–302, 2005.
20. W. Ecker, V. Esen, and M. Hull. *Requirements and concepts for transaction level assertions*. In *Proc. of IEEE ICCD*, pp. 286–293. 2006.
21. A. Ghofrani, F. Javahery, and Z. Navabi. *Assertion based verification in TLM*. In *Proc. of IEEE EWDTS*, pp. 509–513. 2010.
22. A. Kasuya and T. Tesfaye. *Verification methodologies in a TLM-to-RTL design flow*. In *Proc. of ACM/IEEE DAC*, pp. 199–204. 2007.
23. W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. *Specification Language for Transaction Level Assertions*. In *Proc of IEEE HLDVT*, pp. 77–84. 2006.
24. Z. Xiong, J. Bian, and Y. Zhao. *An assertion-based verification method for SystemC TLM*. In *Proc of IEEE ICCCAS*, pp. 842–846. 2010.
25. L. Pierre and L. Ferro. *A Tractable and Fast Method for Monitoring SystemC TLM Specifications*. IEEE Trans. Computers, vol. 57(10):pp. 1346–1356, 2008.
26. 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*, pp. 237–242. 2008.
27. L. Ferro and L. Pierre. *Formal semantics for PSL modeling layer and application to the verification of transactional models*. In *Proc. of ACM/IEEE DATE*, pp. 1207–1212. 2010.

28. L. Pierre and L. Ferro. *Enhancing the assertion-based verification of TLM designs with reentrancy*. In *Proc. of ACM/IEEE MEMOCODE*, pp. 103–112. 2010.
29. M. Kallel, Y. Lahbib, R. Tourki, and B. A. *Aspect-based ABV for SystemC transaction level models*. In *Proc. of IEEE ICM*, pp. 304–307. 2009.
30. M. Kallel, Y. Lahbib, R. Tourki, and B. A. *Verification of SystemC transaction level models using an aspect-oriented and generic approach*. In *Proc. of IEEE DTIS*, pp. 1–6. 2010.
31. Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. *FoCs: Automatic Generation of Simulation Checkers from Formal Specifications*. In *Proc. of CAV*, pp. 538–542. 2000.
32. M. Chen and P. Mishra. *Assertion-based functional consistency checking between TLM and RTL models*. In *Proc. of IEEE VLSID*, pp. 320–325. 2013.
33. L. Pierre and Z. B. H. Amor. *Automatic refinement of requirements for verification throughout the SoC design flow*. In *Proc. of ACM/IEEE CODES+ISSS*, pp. 1–10. 2013.
34. N. Bombieri, F. Fummi, and G. Pravadelli. *On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL*. In *Proc. of ACM/IEEE DATE*, pp. 1–6. 2006.
35. N. Bombieri, N. Deganello, and F. Fummi. *Integrating RTL IPs into TLM designs through automatic transactor generation*. In *Proc. of ACM/IEEE DATE*, pp. 15–20. 2008.
36. Accellera. *Property Specification Language Reference Manual*. http://www.accellera.org, 2004.
37. A. Pnueli and A. Zaks. *PSL model checking and run-time verification via testers*. In *FM 2006: Formal Methods*, pp. 573–586. Springer, 2006.
38. *IEEE Standard SystemC Language Reference Manual*. http:///ieeexplore.ieee.org, 2006.
39. S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. *A self-tuning DVS processor using delay-error detection and correction*. vol. 41(4):pp. 792–804, 2006.
40. IBM. *FoCs Property Checkers Generator*. https://www.research.ibm.com/haifa/projects/verification/focs/start.html.