RTL property abstraction for TLM assertion-based verification

Nicola Bombieri, Riccardo Filippozzi, Graziano Pravadelli and Francesco Stefanni
Department of Computer Science - University of Verona
{firstname.lastname}@univr.it

***Abstract*—Different techniques and commercial tools are at the state of the art to reuse existing RTL IP implementations to generate more abstract (i.e., TLM) IP models for system-level design. In contrast, reusing, at TLM, an assertion-based verification (ABV) environment originally developed for an RTL IP is still an open problem. The lack of an effective and efficient solution forces verification engineers to shoulder a time consuming and error-prone manual re-definition, at TLM, of existing assertion libraries. This paper is intended to fill in the gap by presenting a technique to automatically abstract properties defined for RTL IPs with the aim of creating dynamic ABV environments for the corresponding TLM models.**

I. INTRODUCTION

Electronic system-level (ESL) design, assertion-based verification (ABV), and reuse of intellectual-property (IP) models are three key approaches often combined to address the increasing complexity of todays System-on-chip (SoC) design.

The trend of ESL design and verification has led both industry designers and third-party vendors to extend the library of register-transfer level (RTL) IP implementations with the corresponding transaction-level modeling (TLM) descriptions. Even if such higher-level models are still mainly developed by hand, both methodologies [1], [2] and commercial tools [3], [4] for reusing the existing RTL IPs and automatically abstracting them into TLM models are spreading.

On the other hand, several techniques and frameworks have been developed to apply ABV to ESL design, particularly at TLM. First, approaches have been proposed for both static and dynamic ABV of cycle-accurate TLM models [5], [6]. Then, general concepts [7], requirements [8], and frameworks [9], [10] have been presented to adopt dynamic ABV in more abstract TLM models. Alternative ABV frameworks based on Property Specification Language (PSL) have been also presented to support TLM 2.0 coding styles [11], [12]. Finally, a methodology [13] and the corresponding tool [14] have been developed to enable dynamic verification of temporal properties for TLM specifications, where PSL is used to express communication behaviours.

Reuse of ABV properties in TLM-based design flows has been addressed in [15], [16], [17]. In particular, [15] and [17] present two different methodologies to check the functional consistency between TLM and RTL models by reusing TLM properties at RTL through ad-hoc refinement rules. Instead, [16] presents a technique to reuse TLM properties at RTL through TLM/RTL transactors. 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, reusing existing properties in an RTL-to-TLM bottom-up design flow to check the consistency of TLM models w.r.t. the corresponding RTL models is still an open problem. An attempt has been proposed to reuse RTL checkers at TLM [18]. Nevertheless, this approach suffers from applicability, since it is suited for cycle-accurate TLM models only.

This work was supported by the EC co-funded SMAC (SMArt Systems Co-Design) project Grant Agreement FP7–ICT–288827.

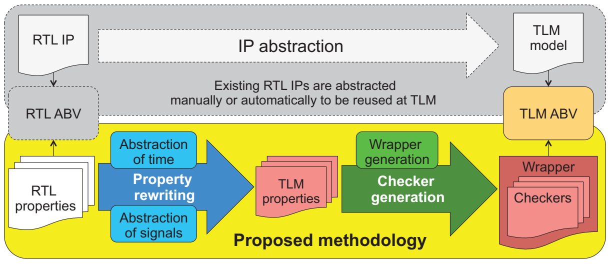


Fig. 1. Methodology overview.

The main intent of this work is to automatically build a dynamic ABV environment for a TLM model, with no restriction on the abstraction level, by starting from a set of properties initially defined for a corresponding RTL implementation[1], as depicted in Fig. 1. To achieve this goal, the proposed methodology acts in two directions. First, we automatically rewrite cycle-accurate RTL properties into a set of properties suited to be checked on an event-based TLM model. This is done by applying a set of transformation rules that reflect, on the properties, the effect of abstraction (i) on the timing reference (abstraction of time) and (ii) on the communication protocol (abstraction of I/O signals) of the design under verification (DUV). Secondly, we define an approach to synthesize TLM properties into checkers to be adopted for dynamic ABV of the TLM model. The approach is independent from the methods applied to generate checkers thanks to the definition of an opportune wrapper.

The advantages of the proposed solution are the following:

- avoiding the time-consuming and error-prone manual effort for re-defining TLM properties;
- reusing existing tools, which are at the state of the art for RTL ABV, to synthesize TLM properties into checkers; and
- minimizing the overhead introduced by the generated checkers in the TLM simulation.

The paper is organized as follows. Section II summarizes the background related to linear temporal logic (LTL) logic and PSL. Section III describes how RTL properties are rewritten to be compliant with TLM verification. Section IV deals with checker generation. Section V discusses experimental results. Finally, conclusions are summarized in Section VI.

II. BACKGROUND

The methodology proposed in this work is intended for abstracting LTL properties compliant with the simple subset of PSL. This section summarizes basics concepts on LTL logics and PSL language.

By using the PSL syntax for temporal operators, LTL is defined as reported in Def. II.1.

[1]To keep the notation simple and intuitive, in the following, we use the terms *RTL properties* and *TLM properties* to indicate properties defined for RTL and TLM models, respectively.

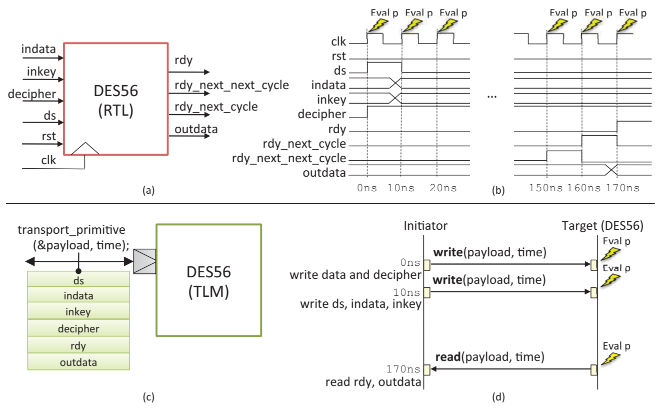


Fig. 2. (a) A RTL DES56 encryption/decryption model, (b) a snippet of the I/O waveforms, (c) a TLM timing-equivalent model, and (d) the sequence of TLM transactions implementing the I/O data exchange corresponding to the time frame of (b).

$$\text{RTL properties}\begin{cases} p_1 : always\ \big(!(ds \wedge indata = 0) \vee (next[17](out \neq 0))\big)@clk\_pos \\ p_2 : always\ \big(!ds \vee (next(!ds\ until\ next(rdy)))\big)@clk\_pos \\ p_3 : always\ \big(!ds \vee (next[15](rdy\_next\_next\_cycle) \wedge \\ \qquad next[16](rdy\_next\_cycle) \wedge next[17](rdy))\big)@clk\_pos \end{cases}$$

$$\text{TLM properties}\begin{cases} q_1 : always\ \big(!(ds \wedge indata = 0) \vee (next^{1}_{170}(out \neq 0))\big)@T_b \\ q_2 : always\ \big(!ds \vee (next^{1}_{10}(!ds)\ until\ next^{2}_{20}(rdy)))\big)@T_b \\ q_3 : always\ \big(!ds \vee next^{1}_{170}(rdy)\big)@T_b \end{cases}$$

Fig. 3. RTL properties for the DES56 models of Fig. 2 and the corresponding TLM properties generated with the proposed methodology.

Definition II.1. Given a finite set of atomic propositions AP, the set of LTL properties over AP can be defined, in negation normal form, as follows:

- $a \in AP$ and $\neg a$ are LTL properties;
- if p_1 and p_2 are LTL properties then $p_1 \vee p_2$, $p_1 \wedge p_2$, $next\ p_1$, $p_1\ until\ p_2$ and $p_1\ release\ p_2$ are LTL properties.

Intuitively, the semantics of temporal operators *next*, *until* and *release* is:

- $next\ p_1$ holds at time t if p_1 holds at time $t+1$;
- $p_1\ until\ p_2$ holds at time t if p_1 holds for all instants $t' \geq t$ until p_2 holds;
- $p_1\ release\ p_2$ holds at time t if p_2 holds for all instants $t' \geq t$ until and including the instants where p_1 first becomes true; if p_1 never becomes true, p_2 holds forever.

In the rest of the paper, a composition of n *next* operators $next(next(\ldots next(a)\ldots))$ is abbreviated in $next[n](a)$, according to the PSL syntax.

PSL is a property specification language that extends LTL and CTL for enabling designers to capture their intent in a verifiable form and verification engineer to validate that the implementation satisfies its specification through dynamic (i.e., simulation) or static (i.e., formal) verification techniques. Dynamic verification is performed by synthesizing PSL properties into checkers, i.e, components that monitor the evolution of the DUV during simulation and raise a failure when a property violation is observed. Checker generation is easy when the simple subset of PSL is adopted, which restricts the composition of temporal properties to ensure that time moves forward from left to right through a property, as it does in a timing diagram [19]. Even if PSL has been originally intended for RTL verification, several works have extended its application to TLM [20], [11], [14].

III. Property rewriting

The abstraction of a property from RTL to TLM must reflect changes implemented during the abstraction of an RTL DUV towards an equivalent TLM model. In particular, two main aspects must be analyzed to map RTL properties towards TLM properties: (i) the change of temporal reference for temporal operators, caused by moving from an RTL cycle-accurate simulation towards a TLM transaction-based simulation (Section III-A), and (ii) the removal of some primary inputs/outputs, caused by the abstraction of the I/O communication protocol (Section III-B). Designers and abstraction tools can address such aspects in various ways leading to the definition of different versions of TLM models corresponding to the same RTL implementation. This diversity makes the definition of an automatic procedure for abstracting properties particularly challenging. The solution we propose to overcome this problem relies on the assumption that the RTL implementation and the corresponding TLM model are *timing equivalent* according to the following definition.

Definition III.1. An RTL model M_{RTL} and a TLM model M_{TLM} are timing equivalent if and only if for all signals s belonging to the I/O interface of both models, when s is assigned to a value v on M_{RTL} at time t, the same assignment happens on M_{TLM} at the same time, and vice versa.

The previous definition guarantees that, whatever procedure is applied to abstract the RTL implementation, the final TLM model preserves both the IP functionality and the IP timing.

A. Abstraction of time

At RTL, the DUV simulation behaves according to cycle-accurate events, generally synchronized with respect to the rising and/or falling edge of one or more clocks, when input signals are assigned and outputs signals are observed. In contrast, at TLM, the simulation proceeds along with asynchronous events corresponding to transactions. According to the selected coding style (e.g., loosely-timed, approximately-timed, cycle-accurate), *write* transactions ask for a task elaboration, while *read* transactions get the result back. This difference is depicted in Fig. 2, where, as running example, the snippet of the RTL I/O waveforms and the corresponding sequence of TLM transactions of a DES56 encryption/decryption model are reported.

Temporal logics, like LTL, and property specification languages, like PSL, are suited to capture behaviours of cycle-accurate RTL models, where temporal operators are naturally referred to clock events. In PSL, for example, every property has an associated *clock context* that specifies when the property must be evaluated. The base clock context is *true*, which means the time granularity adopted to check the consistency between DUV and properties is defined by the verification tool. More frequently, an explicit clock context is specified during the definition of the property through the @ operator. In this case, the property is evaluated when the Boolean expression following the @ operator holds. Often, for RTL properties, the @ expression simply specifies clock events (i.e., falling/rising edges). For instance, the RTL property p_1 in Fig. 3 specifies that $out \neq 0$ is true 17 events later than $ds \wedge indata = 0$ has become true, each time $ds \wedge indata = 0$ holds. In this case, according to the @ expression, the property is evaluated at every positive edge of the clock as reported in Fig. 2(b).

At TLM, the clock is abstracted away, consequently the evaluation points of a property can solely be expressed in terms of a *transaction context* referring to starting and/or ending points of transactions, as shown, for example, in Fig. 2(d). Therefore, as a first step towards RTL-to-TLM property abstraction, we map the RTL clock context in a TLM transaction context according to the following definition.

Definition III.2. Given an RTL property p with clock context

C, a transaction context T for the corresponding TLM property q is defined as follows:

- if C is either the basic clock context (i.e., *true*) or it belongs to the set $\{@clk, @clk\_pos, @clk\_neg\}$, then C is mapped on the basic transaction context T_b, which evaluates q at the end of every TLM transaction;
- if C is expressed as $clock\_expr \wedge var\_expr$, where $clock\_expr$ is one of $\{@clk, @clk\_pos, @clk\_neg\}$ and var_expr is a Boolean expression involving other variables of the DUV different from a clock, then C is mapped on $T_b \wedge var\_expr$, where T_b is the basic transaction context.

Mapping the clock context to the transaction context is not enough to guarantee a correct RTL-to-TLM property abstraction. In fact, as a result of the RTL-to-TLM abstraction of the DUV, a transaction generally merges several RTL events in a single *read* or *write* operation, and thus it embraces an imprecise number of clock cycles. While this does not affect the evaluation of properties that include the $until$ and $release$ operators (and all the other LTL operators that are derived from them, i.e., $always$ and $eventually$), it represents a problem for properties that explicitly count the passing of time through the *next* operator, as clarified hereafter.

The $until$ and $release$ operators do not refer to a precise time instant, but to some event which must occur in an imprecise point in the future. Given that the functionality of the design is preserved during the RTL-to-TLM abstraction, an until or release-based property that holds on the RTL DUV is also true, without the need of being transformed, when evaluated on the corresponding TLM model. This is proven by the next Theorem.

Theorem III.1. Given an RTL implementation M_{RTL}, a timing-quivalent TLM model M_{TLM}, and a property p with clock context C involving, among temporal operators, only *until* and *release*, if $M_{RTL} \models p\ @\ C$ then $M_{TLM} \models p\ @\ T$, where the transaction context T is generated according to Def. III.2.

Proof: [2] Since M_{RTL} and M_{TLM} are timing equivalent, a transaction is executed by M_{TLM} in all instants corresponding to RTL clock cycles where at least one I/O signal of M_{RTL} changes its value, to reflect the same I/O modification at TLM. Otherwise, the two models would not be timing equivalent. The only possibility for having that $p\ @\ T$ fails on M_{TLM} is represented by the fact that the transaction context T excludes from verification one or more of such instants missing the observation of something relevant for determining the truth value of p. However, this is impossible by construction of the transaction context (Def III.2). ∎

Differently from $until$ and $release$, the $next$ operator explicitly counts events. Since RTL events (based on clock cycles) differ from TLM events (based on transactions), properties including $next$ operators that hold on an RTL DUV cannot be re-used to check a corresponding abstracted TLM model by simply replacing the clock context with a transaction context. For example, for property p_1 in Fig. 3 the checking procedure (independently from its static or dynamic nature) needs to count 17 clock cycles before evaluating the consequence of the implication, after the antecedent has been fired[3]. If the same property was evaluated at TLM by simply substituting the clock context with a transaction context based on Def. III.2, the checking procedure would wait for 17 transactions, definitely invalidating the property.

A naive solution to such a problem would be scaling RTL clock cycles to TLM transactions, i.e., mapping the n clock cycles analysed by a $next[n]$ operator included in a RTL property p on a corresponding number of m transactions, such that, at TLM, we can substitute $next[n]$ with $next[m]$ inside p. However, this solution is not generally applicable because it requires to precisely know the number of clock cycles corresponding to each transaction and the exact sequence of transactions that will be executed by the DUV in the time window monitored by each property. On the contrary, the arrival of an overlapping (unexpected) transaction affecting a part of the design not monitored by a property could introduce an extra evaluation point for that property causing its inopportune failure [7], [17].

[2]For lack of space, only an informal idea of the proof is reported.

[3]Note that $\neg a \vee b \equiv a \rightarrow b$, thus p can be read as an implication.

To effectively address RTL-to-TLM property abstraction without the limitations of the previous naive solution, we propose the definition of a new operator, $next^{\tau}_{\epsilon}$, where $\tau \in \mathbb{N}$ and $\epsilon \in \mathbb{N}^{+}$, which is specifically intended for dynamic (i.e., simulation-based) ABV at TLM. The apex τ represents the position of $next^{\tau}_{\epsilon}$ in the property with respect to other occurrences of the same operator; it is introduced for a correct generation of checkers as reported in Section IV. The subscript ϵ represents the required evaluation time, i.e., the exact simulation time when the operand must be evaluated with respect to the firing of the property. More formally, we define the semantics of $next^{\tau}_{\epsilon}$ as follows.

Definition III.3. Given a model M, an LTL property p, and a dynamic ABV environment E, $M \models next^{\tau}_{\epsilon}(p)$ if p is true after ϵ simulation instants (expressed in nanoseconds). If the verification environment E is unable to evaluate p at time ϵ, $next^{\tau}_{\epsilon}(p)$ is false; this happens when no event is observable by E at time ϵ.

In a simulation-based context, a $next$ operator can be replaced by $next^{\tau}_{\epsilon}$, with opportune values for τ and ϵ, without changing the semantics of the property. For example, assuming a clock period of 10ns, p_1 in Fig. 3 can be equivalently expressed as the following p_1' property:

$$always(!(ds \wedge indata = 0) \vee (next^{1}_{170}(out \neq 0)))@clk\_pos.$$

When evaluated at RTL with clock context clk_pos, p and p_1' are equivalent. On the contrary, during TLM verification, the clock context of p' is substituted with a transaction context, leading to the property q_1 reported in Fig. 3. In this way, q_1 preserves the original intent of p, by checking, at TLM, $out \neq 0$ after 170ns from the firing of $ds \wedge indata = 0$.

According to the previous observations, we propose a property abstraction methodology based on the following sequence of automatic steps.

Methodology III.1. Given an RTL implementation M_{RTL} and a timing-equivalent TLM model M_{TLM}, the following steps are executed to transform an RTL property p with clock context C into a TLM property q with transaction context T, such that if $M_{RTL} \models p\ @\ C$ then $M_{TLM} \models q\ @\ T$, in the context of dynamic ABV:

1) transform p such that it is expressed in the negation normal form according to Def. II.1;
2) leave unchanged $until$ and $release$ operators and remap the $next$ operators of p with a sequence of $next^{\tau}_{\epsilon}$ operators in q;
3) remap the RTL clock context C with a TLM transaction context T according to Def. III.2.

Steps 1 and 3 are straightforward. Step 1 follows from well-know transformation rules, while step 3 is implemented according to Def. III.2. Step 2 requires, instead, a two-phase elaboration. The first phase, called *push_ahead_procedure*, pushes ahead the *next* operators in p such that their operands can be exclusively atomic propositions, negation of atomic propositions, or $next$ operators. This is obtained by applying the following transformation rules:

- $next(a \vee b) \equiv next(a) \vee next(b)$;
- $next(a \wedge b) \equiv next(a) \wedge next(b)$;
- $next(a\ until\ b) \equiv next(a)\ until\ next(b)$;
- $next(a\ release\ b) \equiv next(a)\ release\ next(b)$.

Pushing ahead the $next$ operators is preparatory to the second phase, where a composition of $next$ operators is substituted with a single $next_{\epsilon}^{\tau}$, by setting τ and ϵ according to Algorithm III.1. Given a property p obtained from the application of the *push_ahead_next* procedure, Algorithm III.1 gets, as inputs, the clock period c of the original RTL DUV where p holds, and the sequence of sub-formulas $S = s_1(a_1), \ldots, s_m(a_m)$ of p, where each $s_i(a_i)$ is the composition of an arbitrary number n_i of $next$ operators applied to the atomic proposition a_i (or to its negation), i.e., in PSL notation $next[n_i](a_i)$.

Algorithm III.1 Substitution of $next[n_i]$ with $next_{\epsilon}^{\tau}$

```
1: procedure NEXT_SUBSTITUTION($c$, $S$)
2:     for all $s_i(a_i) \equiv next[n_i](a_i) \in S$ do
3:         $\epsilon = n\_i * c$
4:         $\tau = i$
5:         substitute $s_i(a_i)$ with $next_{\epsilon}^{\tau}(a_i)$
6:     end for
7: end procedure
```

According to the semantics of the $next_{\epsilon}^{\tau}$ operator, Algorithm III.1 guarantees that the resulting property can be verified on a TLM DUV by using a verification environment based on a transaction context, without the risk of failures due to evaluation of the property at incorrect time instants. In other words, if a failure occurs by evaluating the property at TLM, it is only due to a wrong abstraction of the TLM design with respect to the original RTL implementation. This is formally proven by Theorem III.2.

Theorem III.2. Given an RTL implementation M_{RTL}, a timing-equivalent TLM model M_{TLM}, a property p with clock context C and a property q with transaction context T, derived from p by following Methodology III.1, if $M_{RTL} \models p\ @\ C$ then $M_{TLM} \models q\ @\ T$.

Proof: [4] When $p = q$ we reduce to the case of Theorem III.1. When $p \neq q$ it means p involves at least one instance of the operator $next$. We observe, first, that $M_{RTL} \models p\ @\ C \Rightarrow M_{RTL} \models q\ @\ C$. In fact, Steps 1 and the *push_ahead_procedure* of step 2 of Methodology III.1 are only syntactic transformations that do not affect the semantics of p. Algorithm III.1 replaces occurrence of $next[n_i](a_i)$ with $next_{\epsilon}^{\tau}(a_i)$ where $\epsilon = n_i * c$ and c is the clock period of M_{RTL}. But, according to the semantics of $next[n_i](a_i)$, a_i must be true after n_i clock cycles, i.e., after $n_i * c$ nanoseconds, which exactly corresponds to the semantics of $next_{\epsilon}^{\tau}(a_i)$ reported in Def. III.3. This proves $M_{RTL} \models p\ @\ C \Rightarrow M_{RTL} \models q\ @\ C$. At this point, with considerations similar to the proof of Theorem III.1 and on the basis of the timing equivalence between M_{RTL} and M_{TLM} we can derive that $M_{RTL} \models q\ @\ C \Rightarrow M_{TLM} \models q\ @\ T$, which finally leads, by transitivity, to $M_{RTL} \models p\ @\ C \Rightarrow M_{TLM} \models q\ @\ T$. ■

To clarify the proposed methodology on the DES56 example, let us consider property p_2 in Fig. 3. Being already in negation normal form, step 1 is skipped. Then, by applying the *push_ahead_procedure* included in step 2, we obtain:

$$always(!ds \vee (next(!ds)\ until\ next[2](rdy)))@clk\_pos.$$

Step 2 is completed by applying Algorithm III.1 that, supposing an RTL clock period of 10ns, produces:

$$always(!ds \vee (next_{10}^{1}(!ds)\ until\ next_{20}^{2}(rdy)))@clk\_pos.$$

Finally, the substitution of the clock context by following Def. III.3 provides the final TLM property q_2 showed in Fig. 3.

[4]For lack of space, only an informal idea of the proof is reported.

a_s	$\rightsquigarrow$	$\emptyset$	$next(a_s)$	$\rightsquigarrow$	$\emptyset$
$p \vee \emptyset$	$\rightsquigarrow$	p	$\emptyset \vee p$	$\rightsquigarrow$	p
$p \wedge \emptyset$	$\rightsquigarrow$	p	$\emptyset \wedge p$	$\rightsquigarrow$	p
$p\ until\ \emptyset$	$\rightsquigarrow$	p	$\emptyset\ until\ p$	$\rightsquigarrow$	$\emptyset$
$p\ release\ \emptyset$	$\rightsquigarrow$	$\emptyset$	$\emptyset\ until\ p$	$\rightsquigarrow$	p

Fig. 4. Transformation rules for signal abstraction.

B. Abstraction of signals

At RTL, the I/O protocol is accurately described. Control signals, in addition to data signals, are used to implement the handshaking mechanism between components. At TLM, when coding styles higher than cycle-accurate TLM are adopted, the I/O protocol may be abstracted by removing control signals to focus on the pure functionality and to speed-up the simulation. This means that RTL properties including abstracted signals need to be reformulated to exclude such signals at TLM.

The removal of a signal implies that its role is no longer expressed in the TLM model. Hence, subformulas involving the abstracted signals become irrelevant and cannot be evaluated at TLM. For this reason, such subformulas must be removed as well. The impact of their removal on the semantics of the remaining formula must be accurately analysed. According to the timing equivalence definition (Def. III.1), we consider only the case in which the timing of TLM events on the preserved I/O signals is equivalent to the timing of corresponding I/O signals at RTL. On the base of this assumption, the solution we propose consists in defining a set of transformation rules that delete subformulas including abstracted signals and preserve, when possible, the intent of the original property. The proposed transformation rules are reported in Fig. 4, where a_s represents an atomic proposition involving abstracted signals to be deleted, p is a generic LTL formula, $\emptyset$ is used to represent that the subformula has been removed as effect of the application of a transformation rule.

In some cases, the application of rules in Fig. 4 leads to the deletion of the whole property. This happens when the semantics of the property is completely dependent on the RTL handshaking mechanism rather than on the pure IP functionality, and thus the property becomes meaningless on a model that definitely abstracts the protocol. On the other cases, different considerations apply. Let us consider that p and p' represent, respectively, an RTL property including subformulas that operate on abstracted signals, and the corresponding property after the application of the rules in Fig.4. On the assumption that p holds on the RTL implementation, transformation rules may lead p' to be a logical consequence of p or not. On the first case, p' is still true at RTL, and thus, after the application of Methodology III.1, it must holds also on the TLM model. On the contrary, the TLM model would not be timing-equivalent to the RTL implementation. When p' is not a logical consequence of p, human investigation is required to analyse the result of checking p' on the TLM model. A failure at TLM could depend either on a wrong implementation of the TLM model or on a modification of the semantics of the property that, due to the application of the transformation rules, does not reflect the change occurred in the RTL-to-TLM abstraction of the communication protocol. In this second case, p' requires to be manually refined for restoring the compliance with the designer intent. A completely automatic procedure would be applicable only in presence of strict and well-defined rules the designer should apply for abstracting the communication protocol, which is generally not the case.

For example, property q_3 in Fig. 3 has been obtained from property p_3 by applying Methodology III.1 and rules in Fig. 4, on the assumption that signals $ready\_next\_cycle$ and $ready\_next\_next\_cycle$ have been removed during the RTL-to-TLM abstraction of DES56 (Fig. 2).

IV. Checker generation

Properties abstracted according to the methodology proposed in the previous section are intended to be synthesized into checkers to set up the dynamic ABV environment for TLM models depicted in Fig. 1. The approach is independent from the way checkers are generated; for example, techniques and tools described in [21], [22], [23] for the PSL language can be adopted. To take care of the presence of $next_{\epsilon}^{\tau}$ operators, for each property we defined a *wrapper* that executes checkers at the correct simulation instants. From the point of view of the checker generator, $next_{\epsilon}^{\tau}(a)$ is synthesized as it was $next[\tau](a)$, i.e., without the wrapper, a would be evaluated after τ events according to the defined transaction context. The wrapper restricts the set of events where a can be evaluated according to the value of ϵ. In practice, when $\tau - 1$ events have been consumed by the verification environment, a is evaluated only when and if a transaction at time ϵ occurs, which finally is identified as event τ. If a transaction arrives at time $t < \epsilon$, it is not considered for the evaluation of $next_{\epsilon}^{\tau}(a)$ and the last consumed event remains $\tau - 1$ waiting for the next transaction. If a transaction arrives at time $t > \epsilon$ and the event τ has not been processed yet, a failure is raised.

To clarify the approach and describe the structure of the wrapper, let us consider the properties p_3 and q_3 in Fig. 3, where q_3 has been generated from p_3 according to the abstraction methodology presented in Section III and by assuming, at RTL, a clock period of 10ns.

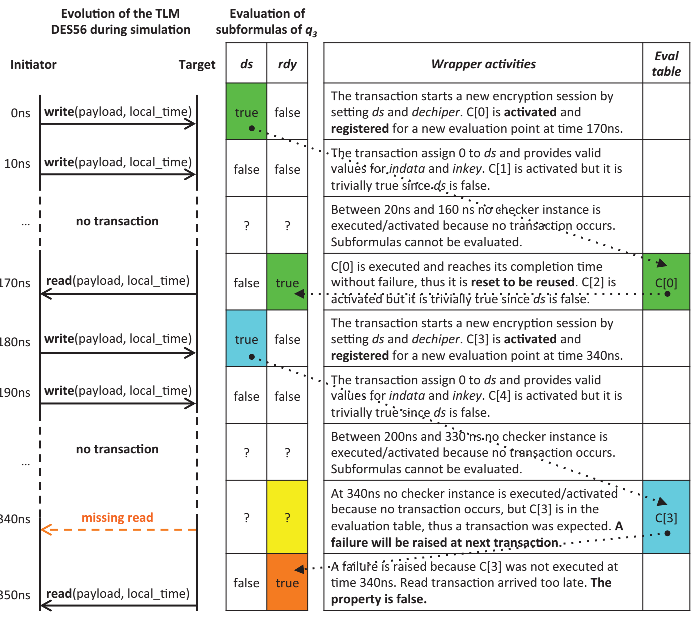
At RTL, the checker of p_3 is called at each rising edge of the clock. If ds becomes true, for example, at clock cycle c_i, the checker monitors clock cycle c_{i+170} to see if rdy becomes $true$. In case of a violation, it raises a failure signal. According to the presence of the $always$ operator, the checker repeats this sequence of evaluations by starting a new verification session at each clock cycle with fresh values for the involved variables.

At TLM, the wrapper for the checker of the property q_3 takes role. It generally behaves as follows.

1- Allocation of checker instances. At the beginning, the wrapper allocates in memory an array C of checker instances for the corresponding property. The size of the array depends on the *lifetime* of a checker instance. The lifetime is the maximum number of instants where transactions can occur in the interval $(t_{fire}, t_{end}]$, where t_{fire} and t_{end} are, respectively, the *firing time* and the *completion time* of the property. The firing time corresponds to the instant where the first subformula of the property is evaluated. The completion time corresponds to the expected verification instant for the last subformula. For example, concerning q_3, t_{fire} and t_{end} correspond, respectively, to the instants in which ds and rdy become true. Then, the size of the array for q_3 is 17, because, being the reference RTL clock period 10ns, we have at maximum 17 instants where transactions can occurs in $(t_{fire}, t_{end}]$ (i.e., $t_{fire}+10ns$, $t_{fire}+20ns$, $\ldots$, $t_{fire}+170ns = t_{end}$). It was not possible having a transaction at a different time from those, because, in that case, it would mean that the RTL implementation and the TLM model would not be equivalent with respect to Def. III.1.

2- Evaluation of active checker instances. To execute checker instances (see Fig. 5), the wrapper maintains and consults an *evaluation table* where the next evaluation points of each checker instance is annotated (see the following point 4). On the occurrence of a transaction T at time t, the wrapper extracts from the table and calls all checker instances whose next evaluation point is expected at time t, if any. Instances still pending on subformulas that were supposed to be evaluated at time $t' < t$, if any, raise a failure. For example, in Fig. 5, the wrapper raises a failure at time 350ns because checker instances $C[3]$ was not executed when expected at time 340ns.

3- Reset and reuse of checker instances. When a checker instance arrives at its completion time t_{end}, the wrapper resets the checker instance such that it can be reused for a new verification session going on with the simulation. For example,


Evolution of the TLM DES56 during simulation
Evaluation of subformulas of q_3
Initiator
Target
missing read
no transaction

		ds	*rdy*	*Wrapper activities*	*Eval table*
0ns	**write**(payload, local_time)	true	false	The transaction starts a new encryption session by setting *ds* and *dechiper*. C[0] is **activated** and **registered** for a new evaluation point at time 170ns.	
10ns	**write**(payload, local_time)	false	false	The transaction assign 0 to *ds* and provides valid values for *indata* and *inkey*. C[1] is activated but it is trivially true since *ds* is false.	
…	**no transaction**	?	?	Between 20ns and 160 ns no checker instance is executed/activated because no transaction occurs. Subformulas cannot be evaluated.	
170ns	**read**(payload, local_time)	false	true	C[0] is executed and reaches its completion time without failure, thus it is **reset to be reused**. C[2] is activated but it is trivially true since *ds* is false.	C[0]
180ns	**write**(payload, local_time)	true	false	The transaction starts a new encryption session by setting *ds* and *dechiper*. C[3] is **activated** and **registered** for a new evaluation point at time 340ns.	
190ns	**write**(payload, local_time)	false	false	The transaction assign 0 to *ds* and provides valid values for *indata* and *inkey*. C[4] is activated but it is trivially true since *ds* is false.	
…	**no transaction**	?	?	Between 200ns and 330 ns no checker instance is executed/activated because no transaction occurs. Subformulas cannot be evaluated.	
340ns	missing read	?	?	At 340ns no checker instance is executed/activated because no transaction occurs, but C[3] is in the evaluation table, thus a transaction was expected. **A failure will be raised at next transaction.**	C[3]
350ns	**read**(payload, local_time)	false	true	A failure is raised because C[3] was not executed at time 340ns. Read transaction arrived too late. **The property is false.**	

Fig. 5. Evolution of the wrapper for property q.

in Fig. 5, $C[0]$ is reset at 170ns. It will be reused with fresh values after instance $C[16]$.

4- Activation of a new instance. A new instance $C[i]$ of the checker is activated at each transaction that respects the transaction context modelled by the @ expression of the property. Then, the wrapper registers $C[i]$ on the evaluation table according to the evaluation points required by the property after its firing, except when the it is trivially true. In Fig. 5, a new checker instance is activated at each transaction, according to the basic transaction context T_b. When ds is *false*, q_3 is trivially *true* and no further evaluation is necessary for that instance. On the other side, when ds is *true*, next the evaluation point of q_3 is registered in the evaluation table 170ns later than t_{fire} to remember at what instants rdy should be evaluated.

V. Experimental Results

The proposed methodology has been applied to two test cases: DES56 and ColorConv, whose VHDL (RTL) descriptions and the corresponding set of PSL properties (9, and 12, respectively) have been provided as a starting point. DES56 implements a reconfigurable (encrypt/decrypt) 64-bit cryptographic algorithm with a latency of 17 clock cycles. ColorConv is a is pipelined IP (8 stages) with a latency of 8 clock cycles. All properties were preserved during the abstraction process. IBM FoCs [21] has been applied to generate the checkers.

To measure the approach applicability and correctness, each test case has been implemented in three different SystemC models (i.e., at three levels of abstraction): at RTL, TLM cycle-accurate (TLM-CA) and TLM approximately-timed (TLM-AT). The TLM-CA model simulation allows us to evaluate the impact of checkers synthesized from the RTL properties without abstraction (i.e., without applying the proposed methodology). The TLM-AT model simulation allows us to evaluate the actual effect of the property abstraction. SystemC RTL has been automatically generated from the VHDL models, and then abstracted towards TLM-CA by using $HIFSuite$ [4] preserving the I/O communication protocol. The TLM-AT versions have been manually implemented by abstracting the I/O interfaces and implementing the IP algorithms with only one *write* transaction (for receiving input data) and one *read* transaction (to return results).

TABLE I. SIMULATION RESULTS

Abstr.level	DES56			ColorConv		
	Sim. time (s)		Overhead (%)	Sim. time (s)		Overhead (%)
	w/out c.	with c.		w/out c.	with c.	
RTL 1 C	4.25	6.93	63.0	11.36	12.46	9.7
RTL 5 C	4.25	9.99	135.1	11.36	13.18	16.0
RTL All C	4.25	16.83	296.2	11.36	15.74	38.6
TLM-CA 1 C	2.03	4.4	116.7	5.38	6.76	25.7
TLM-CA 5 C	2.03	7.77	282.8	5.38	6.99	29.9
TLM-CA All C	2.03	12.58	519.7	5.38	7.11	32.2
TLM-AT 1 C	2.01	2.09	4.0	1.86	1.90	2.3
TLM-AT 5 C	2.01	2.33	15.9	1.86	1.97	5.9
TLM-AT All C	2.01	2.95	46.7	1.86	2.01	8.1

The efficiency of the proposed methodology has been evaluated in terms of overhead introduced by the checkers on the overall model simulation. Table I reports the simulation results. For each abstraction level, the test cases have been simulated without checkers (w/out c.) and with different amounts of checkers (with c.: 1 C, 5 C, All C). Fig. 6 shows the average speedup of the different TLM implementations w.r.t. RTL, both with checkers and without checkers.

In general, the overhead of checkers synthesized from properties abstracted with the proposed methodology and applied to the TLM implementations is one order of magnitude lower than the overhead caused by the checkers of the original properties at RTL (RTL vs. TLM-AT rows in Table I). The overhead of checkers synthesized from the original RTL properties without abstraction and applied to the TLM-CA implementations doubles w.r.t. the overhead of the same checkers in the RTL implementations (RTL vs. TLM-CA rows in Table I). This is due to the fact that the event-driven simulation of the cycle-accurate checkers, which is comparable more to the RTL than to the cycle-accurate TLM simulation, influences most the latter. The number of activated checkers linearly affects the overhead in the overall simulation, in both testcases and at each abstraction level.

We observed the main advantage of the proposed methodology in the RTL vs. TLM speedup before and after the checker integration (see Fig. 6). The original speedup of the two testcases (i.e., without checkers) over the abstraction levels is different. This is due to the different characteristics of the RTL implementations. Without property abstraction, the reuse of RTL properties is possible in the TLM-CA implementations only. In these cases, the checker simulation leads to a decrease of the (even low) speedup between RTL and TLM-CA models (TLM-CA in Fig. 6). In contrast, the property abstraction prosed in this work is intended for TLM-AT models, leading to an increase of the speedup, up to double in the DES56 test case (TLM-AT in Fig. 6). This is due to the fact that the TLM-AT checkers marginally affect the overall event-driven simulation while, in the cycle accurate models, they sensibly increase the number of simulation events at each clock cycle.

We expect that the speedup may be even better by applying checkers synthesized from manually defined TLM properties. However, the results obtained with the proposed methodology have been achieved by automatically reusing the already existing verification environment, without relying on any time-consuming manual transformation.

VI. CONCLUSIONS

This paper presented a methodology to reuse properties, originally defined for an RTL IP model, to verify the corresponding abstracted TLM implementation. The methodology consists of transformation rules that reflect, on properties, the effect of the RTL-to-TLM abstraction and on an approach to synthesize TLM properties into checkers for dynamic simulation of the TLM model. The experimental results, which have been conducted on two representative test cases with different characteristics and complexity, show the applicability and the efficiency of the proposed methodology.

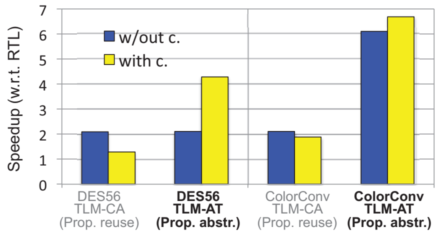


Fig. 6. RTL/TLM simulation average speedup.

REFERENCES

[1] 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.

[2] S. Syed, M. Jenihhin, and J. Raik, "Extensible open-source framework for translating RTL VHDL IP cores to SystemC," in *Proc. of IEEE DDECS*, 2013, pp. 112–115.

[3] Carbon Design Systems. Carbon Model Studio. http://carbondesignsystems.com/.

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

[5] 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.

[6] 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.

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

[8] ——, "Requirements and concepts for transaction level assertions," in *Proc. of IEEE ICCD*, 2006, pp. 286–293.

[9] ——, "Implementation of a transaction level assertion framework in SystemC," in *Proc. of IEEE/ACM DATE*, 2007, pp. 894–899.

[10] 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.

[11] Z. Xiong, J. Bian, and Y. Zhao, "An assertion-based verification method for SystemC TLM," in *Proc of IEEE ICCCAS*, 2010, pp. 842–846.

[12] L. Pierre and L. Ferro, "A tractable and fast method for monitoring SystemC TLM specifications," *IEEE Trans. Computers*, vol. 57, no. 10, pp. 1346–1356, 2008.

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

[14] ——, "Formal semantics for PSL modeling layer and application to the verification of transactional models," in *Proc. of ACM/IEEE DATE*, 2010, pp. 1207–1212.

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

[16] 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.

[17] 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*, 2013, pp. 1–10.

[18] N. Bombieri, F. Fummi, V. Guarnieri, G. Pravadelli, F. Stefanni, T. Ghasempouri, M. Lora, G. Auditore, and M. Marcigaglia, "On the reuse of RTL assertions in SystemC TLM verification," in *Proc. of LATW*, 2014, pp. 1–6.

[19] "Standard for property specification language (PSL)," *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pp. 1–184, 2012.

[20] 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.

[21] 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.

[22] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of PSL properties," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 1, pp. 4:1–4:21, 2008.

[23] G. Di Guglielmo, L. Di Guglielmo, A. Foltinek, M. Fujita, F. Fummi, C. Marconcini, and G. Pravadelli, "On the integration of model-driven design and dynamic assertion-based verification for embedded software," *J. Syst. Softw.*, vol. 86, no. 8, pp. 2013–2033, 2013.