Towards Equivalence Checking Between TLM and RTL Models*

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

Joao Marques-Silva
School of Electronics and Computer Science
University of Southampton, UK
jpms@ecs.soton.ac.uk

Abstract

The always increasing complexity of digital system is overcome in design flows based on Transaction Level Modeling (TLM) by designing and verifying the system at different abstraction levels. The design implementation starts from a TLM high-level description and, following a top-down approach, it is refined towards a corresponding RTL model. However, the bottom-up approach is also adopted in the design flow when already existing RTL IPs are abstracted to be reused into the TLM system. In this context, proving the equivalence between a model and its refined or abstracted version is still an open problem. In fact, traditional equivalence definitions and formal equivalence checking methodologies presented in the literature cannot be applied due to the very different internal characteristics of the models, including structure organization and timing. Targeting this topic, the paper presents a formal definition of equivalence based on events, and then, it shows how such a definition can be used for proving the equivalence in the RTL vs. TLM context, without requiring timing or structural similarities between the modules to be compared. Finally, the paper presents a practical use of the proposed theory, by proving the correctness of a methodology that automatically abstracts RTL IPs towards TLM implementations.

1 Introduction

TLM is nowadays the reference modeling style for HW/SW design and verification of digital systems. TLM greatly speeds up the verification process by providing designers with different abstraction levels whereby digital systems are modeled and verified. Thus, the complexity of the modern systems can be handled by designing and verifying them through successive refinement steps [1].

In a TLM-based design flow, a system is first modeled at high-level in order to check the pure functionality, disregarding details related to the target architecture. Thus, motivated by the lack of implementation details, the simulation speed is a few orders of magnitude faster than at RTL. Then, step by step, designers refine and verify the system description more accurately, towards the final RTL implementation.

Reuse of previously-developed Intellectual Propertie (IP) modules is another key strategy that guarantees considerable savings of time in transaction level modeling. In fact, modeling a complex system completely at transaction level could be inconvenient when IP cores are already available on the market, usually modeled at RTL. In this context, modeling and verification methodologies are based on transactors for converting TLM function calls to sequences of RTL signals and vice-versa [2], thus allowing the integration between TLM and RTL components. Nevertheless, since the integration of RTL IPs into a TLM design involves slowing down the whole system simulation, a methodology has been recently proposed to automatically abstract RTL IPs towards TLM descriptions [3].

Whether in the top-down or in the bottom-up flow, it is mandatory to verify the equivalence between implementations at different abstraction levels after each refinement or abstraction step (see Figure 1). Some techniques have been proposed in the past to check the correctness of the top-down refinement flow [4, 5]. In [4], properties expressed at TLM are reused at RTL by means of transactors. Thus, the behavior expressed by the set of reusable properties can be checked also in the refined version of the design. In [5], an incremental verification based on assertions is proposed to validate the TLM-to-RTL design refinement. The concept of equivalence proposed in [4, 5] is based on properties, i.e., two implementations are equivalent if they satisfy the same set of properties. Thus, the effectiveness of such a kind of equivalence checking depends on the quality of the defined properties, since the equivalence is guaranteed only for behaviors for which a property has been defined.

The property-based approach differs from the traditional

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

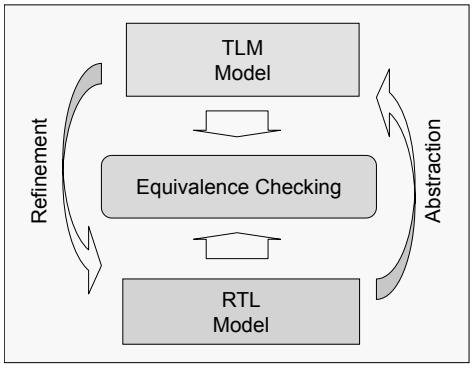

Figure 1. Equivalence Checking in TLM Design Flows.

equivalence checking (EC) strategies presented in literature. *Combinational EC* and *sequential EC* have achieved considerable success in the context of hardware verification, to prove the equivalence between two RTL implementations or between an RTL and a gate-level implementation [6, 7]. In these cases, the proof of equivalence relies on tight similarities in terms of interfaces, temporal and functional behaviors. Combinational EC checks two acyclic and gate-level circuits. Combinational equivalence checkers can also be used to check equivalence of two sequential designs (and thus called sequential EC) once the state encoding of the two designs are the same. Thus, the real challenge of sequential verification is in verifying two designs with different state encodings. Considerable research has been done to find compare-points for latch mapping [8, 9]. However, these techniques operate at gate-level, where they reason in the Boolean domain.

Equivalence checking between system-level (i.e., TLM) and RTL implementations is still an open problem, and, to the best of our knowledge, research work in this topic is still in its early stages [10, 11]. The main problem in checking the equivalence between RTL and TLM models is due to the fact that, generally, there are neither temporal nor structural similarities between TLM and RTL descriptions. Thus, equivalence checking by using the traditional RTL/gate-level techniques can be considered inapplicable. In [11], a technique to apply sequential EC to system level vs. RTL designs is presented. It proposes a methodology to alleviate the complexity of the latch mapping by comparing variables of interest (observables) in the design descriptions. Nevertheless, even if variable mapping is more intuitive and easier w.r.t. standard techniques, the problem becomes unmanageable when the compared circuits are significantly different.

Because of these substantial differences among TLM and RTL abstraction levels, the first main issue concerns what functional equivalence means when TLM and RTL descriptions are compared. In [12], a definition of functional equivalence is given but it is strictly related to C functions and the corresponding Verilog implementation. Several notions of equivalence have been formulated in [10]: combinational equivalence, cycle-accurate equivalence, pipelined equivalence, etc. All of them aim at comparing the model functionality by strictly considering timing. Hence, they are well suitable to be applied for checking "similar" descriptions, while they present strong limitations when applied for comparing models with no matchable timing behaviors.

In this paper, we first give a general formal definition of functional equivalence, that we call *event-based equivalence*, since it is based on the concepts of event and sequence of events. In particular, our definition differs from the others since it has the following key characteristics:

1. The models to be compared are considered as black-boxes and only the I/O behaviors are matched. Thus, no similarities on the internal structures of the models (e.g., common registers, or common finite state machine templates) are required to check the equivalence.
2. The functional equivalence can be checked even if there is no timing correlation between the models, since the definition proposed in this paper relies on the concepts of sequence of events disregarding the real instant when such events occur. That is, the two models executing the same functionality are allowed to perform a possibly lengthy computation completely independent from each other.

A second outcome of the paper is to show how the proposed definition of equivalence applies in the TLM-RTL context by settling the meaning of *event* and *sequence of events* at different abstraction levels. The concept of event plays a key role, as it characterizes the I/O operations whereby the equivalence of the models can be checked.

Finally, we give a more formal definition of the elementary steps of the RTL-to-TLM abstraction methodology presented in [3]. Then, we prove that modules abstracted according to such a methodology are correct by construction with respect to event-based equivalence. The proof relies on the fact that the abstraction steps preserve the event-based equivalence between the original RTL implementation and the TLM abstracted description.

The paper is organized as follows. Section 2 generally introduces the definitions of event, sequence of events, and event-based equivalence. Section 3 introduces the concept of event and event-based equivalence for TLM and RTL abstraction levels. Then, Section 4 presents an RTL-to-TLM abstraction methodology that preserves the event-based equivalence. Finally, Section 5 draws the conclusions.

2 Event-based Equivalence

The event is the base concept to evaluate the behavior of a model through its execution since both functional and performance analysis rely on events. Functional analysis checks the correctness of sequences of events, in terms of values of primary inputs (PIs), primary outputs (POs) and internal registers of the model. Model performance is evaluated by checking the distribution of events in terms of delay over time [13].

2.1 Events and Ordering of Events

In general, an *event* corresponds to *something happening* at a certain time during the evolution of the system model. According to the desired granularity, and the considered abstraction level, an event can be associated to the circumstance in which the clock ticks, an output changes its value, an instruction is executed, a function call returns, etc. Thus, the execution of a set of processes modeling the system under development can be considered as a sequence of events, provided that the granularity of events has been defined.

The main idea of the proposed event-based equivalence consists of providing a way for proving the equivalence of two implementations by comparing their sequence of events. However, the event-based equivalence must be independent from the concept of time, since we want to apply the definition for proving the equivalence between models with different time scales (e.g., RTL vs. TLM). A way for ordering sequence of events, and then comparing them without considering real time, has been proposed by Lamport [14]. Lamport defined relation "happens before" to order a sequence of events related to several processes running on a distributed system. A similar definition can be applied, in the context of this paper, for ordering the sequence of events generated by a model representing a digital system. Without loss of generality, we can assume that the model is composed of a single process. In fact, we consider the model as a black box whose events are observed only at PIs and POs, disregarding the number of processes that act on such PIs and POs. Thus, we define the "happens before" relation, denoted by "$\rightarrow$", as follows:

- event a happens before event b, if a is executed before b;
- if a, b and c are events such that $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$;
- finally, we say that two events, a and b, are concurrent if neither $a \rightarrow b$ nor $b \rightarrow a$. We write $a||b$ if a and b are concurrent.

We assume that $a \nrightarrow a$ for any event a, as systems in which an event can happen before itself do not seem to be physically meaningful. This implies that "$\rightarrow$" is an irreflexive partial ordering on the set of all events in the system.

2.2 Snapshots and Sequences of Events

As better clarified in Section 3, a model is described with a more or less degree of accuracy depending on the adopted abstraction level. In particular, each level is intended to model and verify either the pure untimed functionality or the approximately timed behavior or the cycle accurate behavior. Furthermore, two models, that implement the same functionality at different abstraction levels, very often present completely different internal structures. As a consequence, no relations on the internal structure of the models can be assumed. For this reason, the equivalence of two models should be proved only in terms of sequences of events.

Thus, in this section, we give a definition of equivalence checking which relies only on the assumption that the I/O of the models to be compared can be put in correspondence. Hence, instead of comparing, for example, the states of two finite state machines, we propose to consider the model like black boxes and to check only the I/O behavior that is externally visible at each event. Informally, assuming that the same values are given to the PIs of two models, we require that the outputs give the same results.

It is important to note that, in this paper, we do not introduce a new meaning of equivalence between design descriptions. All the simulation-based techniques, for example, rely on the same concepts by which two designs are equivalent if, having the same test patterns on input, they produce the same output results. The contribution of our definition of equivalence relies on the fact that it aims at proving the correctness of abstraction or refinement methodologies. In other words, considering a methodology that transforms a design into a more abstract or more refined description, the definition of equivalence can be applied to formally prove that the generated description is *correct-by-construction*. An application example is given in Section 4.

We start by defining the *model domains*. Given a model M where $< I_1, .., I_n >$ and $< O_1, ..., O_m >$ are, respectively, the set of PIs and the set of POs of M, we denote with $D(I_i)$ and $D(O_j)$ respectively, the domain of input I_i and output O_j. Furthermore, we denote the tuple of the input and output domains respectively as

$$D(I_1) \times D(I_2) \times .. \times D(I_n)$$

and

$$D(O_1) \times D(O_2) \times .. \times D(O_m).$$

Then, we call *snapshot* an instance of the tuple containing the values of PIs and POs when a particular event occurs. In this context, we distinguish an *input snapshot* (σ^I) and a *model snapshot* (σ^M) as follows:

$$\sigma^I \ \in \ D(I_1) \times D(I_2) \times .. \times D(I_n)$$

$$\sigma^M \ \in \ D(I_1) \times .. \times D(I_n) \times D(O_1) \times .. \times D(O_m)$$

where the snapshot is related only to PIs in the former case, while it is related to both PIs and POs in the latter case. Informally, a snapshot is the configuration of the model externally observed on the I/O when a particular event occurs during the model simulation.

Then, we define the *sequence of events*:

$$\Sigma^I = \ < \sigma_0^I \ , \ \sigma_1^I \ , \ ... >, \ \ where \, \forall i \ \sigma_i^I \rightarrow \sigma_{i+1}^I$$

and

$$\Sigma^M = \ < \sigma_0^M \ , \ \sigma_1^M \ , \ ... >, \ \ where \, \forall i \ \sigma_i^M \rightarrow \sigma_{i+1}^M$$

as the sequence of snapshots observed on the PIs for the former case (*event sequence on inputs*), and the the sequence of snapshots observed in both PIs and POs for the

latter case (*event sequence on the model*). The set of all the event sequences forms the universe of sequences:

$$\bigcup_{\Sigma^I} : \textit{universe of event sequences on inputs;}$$

and

$$\bigcup_{\Sigma^M} : \textit{universe of event sequences on the model.}$$

Finally, we formally define *behavior* $\digamma$ of the model with regard to event sequences on I/O as follows:

$$\digamma_M : \bigcup_{\Sigma^I} \longrightarrow \bigcup_{\Sigma^M}$$

Function $\digamma$ represents the behavior of the model considered as a black box where for any *event sequence on inputs* there is an *event sequence on the model*. The range of $\digamma$ considers both PIs and POs for preserving causality. In fact, it is easier to relate POs events with triggering PIs events if the sequence of events includes both. For example, assuming a system that might not respect causality (i.e., output events could occur before the associated input events), then we could wrongly conclude that such a system would be equivalent to a correct system.

Function $\digamma$ abstracts all the details concerning computation timing as well as internal structures. Hence, we rely on the definition of $\digamma$ to express the concept of equivalence between two models, as follows:

Definition 1 *Given two models M_1 and M_2 and the functions $\digamma_{M_1}$ and $\digamma_{M_2}$ representing their behaviors, M_1 and M_2 are event-based equivalent iff* $\forall\, \Sigma^I \in \bigcup_{\Sigma^I}$,

$$\digamma_{M_1}(\Sigma^I) = \digamma_{M_2}(\Sigma^I)$$

The next section uses event-based equivalence for comparing RTL vs. TLM models.

3 RTL-TLM Event-based Equivalence

TLM levels have been defined in literature in different ways and by proposing different interfaces [15, 16]. However, the key characteristic, which aggregates all of such definitions, concerns the classification of the TLM abstraction levels in terms of timing and communication mechanism. Thus, the following TLM abstraction levels can be defined by using the OSCI terminology [15]:

- TLM Programmer's View (PV), which is transaction-based and untimed;
- TLM Programmer's View with Time (PVT), which is transaction-based and approximately timed;
- TLM Cycle Accurate (CA), which is cycle-based and timed.

At the highest level (PV) a functional specification is created to provide a proof of concept. At this level, it is not determined yet which modules will be implemented in HW and which in SW. Both communication and computational parts of the system are untimed. Data transfers between modules rely on abstract types and point-to-point communications implemented by means of function calls.

At PVT, the model simulates in non-zero simulation time and, thus, a first estimation of timing performance can be performed. Furthermore, HW/SW partitioning is performed according to several constraints (e.g., performance, cost, component availability, etc.), and the abstract architecture is mapped into a set of interconnected resource-constrained blocks. Data transfers still relies on function calls, but they are characterized in terms of bit width and message size to estimate bus bursts. Pipelined structures are introduced by splitting complex operations into a timed sequence of simpler operations.

At the lowest TLM level (CA), the model is cycle accurate. SW components are implemented by means of standard SW engineering techniques, HW components are very similar to behavioral RTL descriptions, and interfaces between components are defined (even if pins can be still hidden). A bus model is introduced and clock-accurate protocols are mapped to the chosen HW interfaces and bus structure. Transactions are mapped directly to bus cycles. Therefore, a CA TLM model is very close to the corresponding RTL model, particularly with respect to the notion of time.

Thus, depending on the TLM abstraction level, a model is described with some degree of accuracy, and the communication protocol performing read or write actions can be syntactically and semantically fairly different. For example, at CA TLM, data is exchanged between the master and the slave ports by means of signals. Thus, a read operation for a slave corresponds to read from PIs. On the contrary, at PV or PVT TLM, the same operation consists of one or more transactions implemented by means of function calls, whose parameters contains the data to be exchanged.

For this reason, it is reasonable to relate the concept of event with respect to the abstraction level, to apply the definition of event-based equivalence formulated in Section 2. In particular, according to the different communication mechanisms implemented at TLM and RTL levels, we consider the following kinds of models:

- *Cycle Accurate* (CA) models, which includes both CA TLM and RTL descriptions;
- *Transaction-based* (TB) models, which includes both PV and PVT TLM descriptions.

Thus, in the following of the paper, we focus on CA models and TB models rather than generically speaking of TLM and RTL models.

3.1 Transaction-Based Events vs. Cycle Accurate Events

In Section 2, the concept of event has been generally defined as "something happening during the evolution of the system". The following definition specifies the concept of event for TB and CA models.

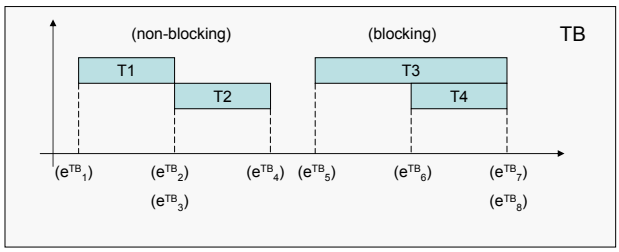

Figure 2. Events in a TB model.

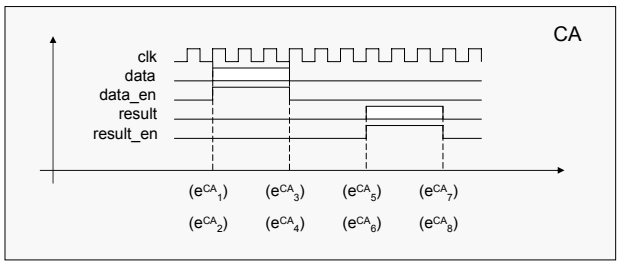

Figure 3. Events in a CA model.

Definition 2 *In a TB model, an event occurs each time a transaction starts or a transaction finishes. In a CA model, an event occurs each time a read on a PI or a write on a PO is performed.*

The "happen before" relation defined in Section 2 is suited for the concept of CA and TB event proposed in Def. 2. In fact, both a sequence of CA events and a sequence of TB events can be partially-ordered according to the "happen before" relation.

Let us consider Figure2 and Figure 3, respectively, for an example of TB sequence and CA sequence.

In the case of a TB model, we distinguish two events per transaction since the function calls characterizing a transaction can be either blocking or non-blocking. In particular, a transaction *T* is *blocking*, if the process triggering *T* is suspended until *T* terminates. In the meantime, other transactions from different processes can be executed (e.g., T3 and T4 in Figure 2). On the contrary, a transaction is *non-blocking* if it must terminate (possibly, with an error condition) before other transactions can be triggered (e.g., T1 and T2 in Figure 2). In case of blocking calls, the system could have overlapping transactions. However, distinguishing between the starting and the ending point of a transaction provide us with a more granular view of the system behavior [13]. As a consequence, we can partially order events in a TB model disregarding the difference between blocking or non-blocking transactions. For example, in Figure 2, the following relations among TB events hold:
$e_1^{TB} \rightarrow e_2^{TB} \rightarrow e_4^{TB} \rightarrow e_5^{TB} \rightarrow e_6^{TB} \rightarrow e_7^{TB}$,
$e_2^{TB} || e_3^{TB}$, $e_7^{TB} || e_8^{TB}$.

Similarly, the sequence of events occurring during the evolution of a CA model can be partially ordered according to the "happen before" relation. Figure 3 shows an example. Signals *data* and *data_en* represent, respectively, the data to be read and the flag for indicating that such data are available to be read, while *result* and *result_en* represent, respectively, the result of the computation and its enabling flag. In this example, the following relations among CA events hold:
$e_1^{CA} \rightarrow e_3^{CA} \rightarrow e_5^{CA} \rightarrow e_7^{CA}$,
$e_1^{CA} || e_2^{CA}$, $e_3^{CA} || e_4^{CA}$, $e_5^{CA} || e_6^{CA}$, $e_7^{CA} || e_8^{CA}$.

Given the definition of event proposed in Def. 2, we can also introduce the concept of snapshot, generally expressed in Section 2.2 as an instance of the tuple containing the values of PIs and POs when a particular event occurs. In a TB model, a snapshot captures the I/O values contained in the data structure exchanged by the transaction corresponding to the observed event. On the other hand, in a CA model, a snapshot captures the values read or written respectively on PIs and POs.

3.2 TB-CA Event-based Equivalence Checking

According to the definition of CA and TB events proposed in Def. 2, this section introduces the concept of event-based equivalence between a CA model and a TB model.

The general definition of event-based equivalence presented in Def. 1 assumes that the I/O operation performed by the models to be compared can be put in correspondence. In fact, assuming that the same input stimuli are provided to both a CA and a TB implementations, the event-based equivalence holds if and only if both implementations produce the same results independently of timing. However, the strategy and the data structure required for reading input stimuli or writing results is generally different, when a CA and a TB models are compared. Moreover, such I/O operations can possibly require a different number of events on CA models with respect to TB models.

Thus, the following correlations between CA and TB models are necessary conditions for proving the event-based equivalence:

- *The tuple of the model domains* (defined in Section 2.2) *of the TB and CA models must correspond.* However, PIs and POs in TB and CA models may differ in type and in number (without affecting the computation result) depending on the communication protocol. Thus, a set of "relevant" I/O objects have to be selected for representing the tuple of the model domains. Any object of this set corresponds to a PI or a PO that is present on both the TB model and the CA model. We assume that the designers provide the set of "relevant" I/O objects and the correspondence of PIs/POs between the CA and TB models.

- *The sequences of events observed during the evolution of the models to be compared must correspond.* This means that a bijection must exist between the CA and the TB sequences of events. Such a correspondence

is automatically achieved by means of two abstraction functions (one for CA models and one for TB models) that, given the CA and the TB sequences of events and the set of "relevant" I/O objects, generate two new sequences of events for which a bijective correspondence exists.

Before presenting the abstraction functions, let us consider an example to better clarify these aspects. Figure 4 shows the evolutions of the CA and TB modules implementing a simple sequential adder. In such an evolution, the adder calculates the sum of two data (*data1* and *data2*). The I/O operations performed by both CA and TB models are conceptually the following:

`read` *data1*; `read` *data2*; `write` *result*.

The CA model (Figure 4a) has 3 PIs (*clock*, *data*, and *data_en*) and 2 POs (*result* and *result_en*). Reading *data1* corresponds to read ports *data* and *data_en*, for getting the *data1* value only if the *data_en* flag is set. The same happens for reading *data2*. Thus, according to Def. 2, each read operation is associated with two CA events[1]: one for reading *data_en*, and the other for reading *data*. In the same way, writing the result corresponds to two events associated to a write on *result* and a write on *result_en*. Thus, for the CA model, a total number of 8 events are generated, where some of them are concurrent.

It is reasonable that a corresponding TB untimed implementation (i.e., PV TLM) of the same system preserves only the PI *data* and the PO *result*, since flags for enabling read and write operations are useless in the case of a transaction-based communication, and clock is not considered at all.

Furthermore, the read operation is performed by two transactions exchanged between the adder module and a TLM communication channel. The first transaction (*get(data1)*) gets the data value from the channel, while the second (*put(OK)*) returns the acknowledgement to the channel. Such a read operation can be considered accomplished only when the second transaction ends. The same happens for getting *data2*. Thus, each read operation is associated with 4 TB events. On the other hand, writing the result of the computation corresponds to 4 events: two for the transaction that puts *result* on the channel and two for the transactions that gets the corresponding acknowledgment. Thus, for the TB model, a total number of 12 events are generated, where some of them are concurrent.

The CA and TB sequences of events obtained so far cannot be matchable, since their lengths differ (8 vs. 12), and some events are not "relevant" to be compared, e.g., CA events on flags do not have corresponding TB events, and TB events on acknowledgment do not have corresponding CA events.

[1]Without loss of generality, in this example we do not consider the clock port. However, in case it was considered, all the events triggered by the read operation on this port would be dropped by the abstraction function presented in the following.

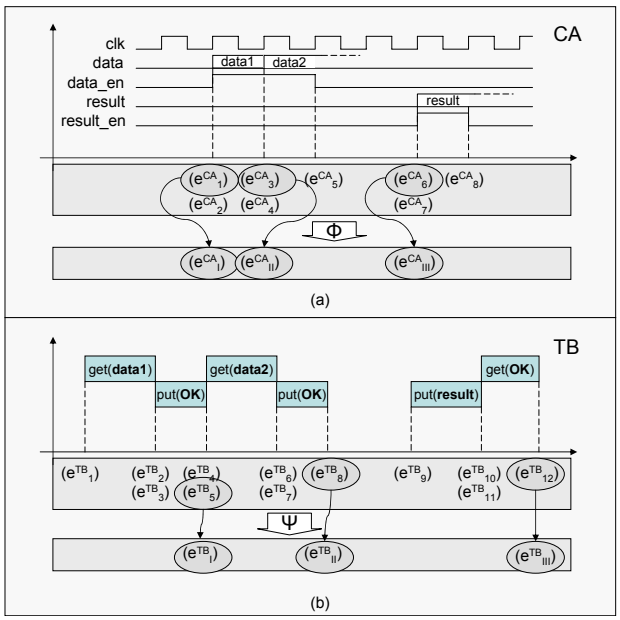

Figure 4. Example of abstraction and matching of CA and TB events.

The following operations must be performed for solving this undesired situation, that prevents the possibility of using the event-based equivalence for comparing TB and CA models:

1. "relevant" I/O objects must be defined by the designers;
2. concurrent events must be collapsed (to avoid multiple captures of the same I/O values in the same instant, since this would be redundant for the analysis of the model equivalence);
3. events specifically generated for compliance with the communication protocol must be removed.

Provided that the first point is accomplished by the designers, the second and the third points are implemented by two *abstraction functions*. Let R be the set of PIs and POs included in the set of the "relevant" I/O objects . We define the abstraction functions Φ and Ψ that work, respectively, on sequences of CA events and sequences of TB events as follows:

$$\Phi(\Sigma^{CA}, R) = \Sigma^{CA'}$$

where an event $e \in \Sigma^{CA}$ is preserved in the new sequence $\Sigma^{CA'}$ only if:

- e is generated by a *read* or a *write* operation performed on an element of R;
- $\Sigma^{CA'}$ does not already contain an event $e_1 \in \Sigma^{CA}$ such that $e||e_1$.

$$\Psi(\Sigma^{TB}, R) = \Sigma^{TB'}$$

where an event $e \in \Sigma^{TB}$ is preserved in the new sequence $\Sigma^{TB'}$ only if:

- e is generated by a *put* or a *get* transaction carrying the acknowledgement of a previous *get* or *put* transaction performed on an element of R;
- $\Sigma^{TB'}$ does not already contain an event $e_1 \in \Sigma^{TB}$ such that $e||e_1$.

Considering the example of Figure 4, Φ and Ψ work as follows, provided that the set of "relevant" I/O object, i.e., R, is composed of *data1*, *data2* and *result*.

$$\Phi(e_1^{CA}, .., e_8^{CA}) = (e_1^{CA}, e_3^{CA}, e_6^{CA}) = (e_I^{CA}, e_{II}^{CA}, e_{III}^{CA})$$

$$\Psi(e_1^{TB}, .., e_{12}^{TB}) = (e_5^{TB}, e_8^{TB}, e_{12}^{TB}) = (e_I^{TB}, e_{II}^{TB}, e_{III}^{TB})$$

In this way, the sequences generated by Φ and Ψ contain all and only events related to the I/O operations that are externally visible by considering the module implementations like black boxes (e.g., read *data1*, read *data2*, and write *result*). Moreover, a bijection exists between the *abstracted* sequences, thus, they can be used for analyzing the event-based equivalence.

From Def. 1 and from the definition of Φ and Ψ we derive the following definition of event-based equivalence that can be used for comparing CA vs. TB models.

Definition 3 *Given a CA model M_{CA}, a TB model M_{TB}, the corresponding universes of event sequences on inputs $\bigcup_{\Sigma_{CA}^I}$ and $\bigcup_{\Sigma_{TB}^I}$, the functions $F_{M_{CA}}$ and $F_{M_{TB}}$ representing the behaviors of M_{CA} and M_{TB}, the set R of "relevant" I/O objects provided by the designers, and the abstraction functions Φ and Ψ, M_{CA} and M_{TB} are event-based equivalent iff $\forall$ pairs $\langle \Sigma_{CA}^I, \Sigma_{TB}^I \rangle \in \bigcup_{\Sigma_{CA}^I} \times \bigcup_{\Sigma_{TB}^I}$, such that $\Phi(\Sigma_{CA}^I) = \Psi(\Sigma_{TB}^I)$,*

$$F_{M_{CA}}(\Phi(\Sigma_{CA}^I)) = F_{M_{TB}}(\Psi(\Sigma_{TB}^I)).$$

The next Section shows how the previous definition can be used to prove that the RTL-to-TLM abstraction methodology proposed in [3] is correct by construction, i.e., the TLM abstracted model is event-based equivalent to the RTL model.

4 RTL-TLM Abstraction Proof of Correctness

In this Section, we formalize the main concepts of the methodology presented in [3], that automatically abstracts RTL IPs towards TLM implementations. We focus on applying the concepts of event-based equivalence introduced in Section 3 to prove the correctness of the process that abstracts an RTL model (which is CA) towards a TLM PVT model (which is TB). A similar proof can be drawn to prove the correctness of the PVT to PV step.

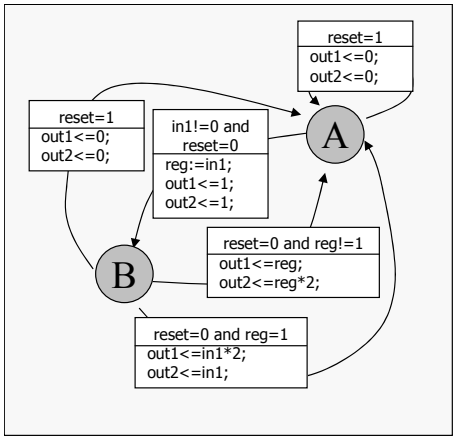

Figure 5. Example of EFSM.

To achieve such a goal, first we need to introduce a formal model that allows us to represent designs at different abstraction levels. Among different alternatives, we select the Extended Finite State Machine (EFSM) [17] since it captures the main characteristics of the state-oriented, activity oriented and structure-oriented model [18].

Definition 4 *An EFSM is defined as a 5-tuple $M = \langle S, I, O, D, T \rangle$ where: S is a set of states, I is a set of input symbols, O is a set of output symbols, D is a n-dimensional linear space $D_1 \times \ldots \times D_n$, T is a transition relation such that $T : S \times D \times I \rightarrow S \times D \times O$. A generic point in D is described by a n-tuple $x = (x_1, ..., x_n)$; it models the values of the registers internal to the design.*

A pair $\langle s, x \rangle \in S \times D$ is called *configuration* of M. An operation on M is defined in this way: if M is in a configuration $\langle s, x \rangle$ and it receives an input $i \in I$, it moves to the configuration $\langle t, y \rangle$ iff $((s, x, i), (t, y, o)) \in T$ for $o \in O$.

The EFSM differs from the traditional FSM, since each transition is not labeled in the classical form i/o, but it takes care of the register values too, as reported below. Transitions are labeled with an *enabling* function e, which represents the guard of the transition, and an *update* function u, which specifies how the values of registers and outputs evolve when the transition is traversed.

Given an EFSM $M = \langle S, I, O, D, T \rangle$, $s \in S, t \in S, i \in I, o \in O$ and the sets $X = \{x | ((s, x, i), (t, y, o)) \in T$ for $y \in D\}$ and $Y = \{y | ((s, x, i), (t, y, o)) \in T$ for $x \in X\}$, the *enabling* and *update* functions are defined respectively as:

$$e(x, i) = \begin{cases} 1 & \text{if } x \in X; \\ 0 & \text{otherwise.} \end{cases}$$

$$u(x, i) = \begin{cases} (y, o) & \text{if } e(x, i) = 1 \text{ and} \\ & ((s, x, i), (t, y, o)) \in T; \\ undef. & \text{otherwise.} \end{cases}$$

Figure 5 exemplifies the state transition graph (STG) of an EFSM representing a simple RTL IP-core. According to

the conditions reported in the enabling functions, the EFSM moves from a state to another at each clock cycle executing the HDL code included in the update function of the traversed transition. Note that, an EFSM can be automatically extracted from an HDL description as reported in [19, 17].

4.1 Cycle-Accurate RTL Model

Let $M_{CA} = \langle S_{CA}, I_{CA}, O_{CA}, D_{CA}, T_{CA} \rangle$ be the EFSM representing the cycle-accurate RTL model we want to abstract towards TLM.

We define a *computational phase* as a sequence of EFSM states that must be consistently traversed to get the input data (*input sub-phase*), elaborate them (*elaboration sub-phase*), and finally provide the related output result (*output sub-phase*). During the input sub-phase, the update functions of the traversed transitions read input data and control lines without performing any further elaboration. Then, data is manipulated in the elaboration sub-phase without reading new values from inputs neither writing on outputs. Finally, in the output sub-phase, the update functions do not modify the computation result anymore, while control and data output lines are written according to the communication protocol selected for the interaction between the module and the environment where it is embedded.

The identification of input, elaboration and output sub-phases on an EFSM is automatic. In fact, each computation phased is composed of three different sets of adjacent states, that can be recognized by parsing the enabling and the update functions of the EFSM transitions.

- **Input states.** A sequence of input states is recognizable since: (1) it starts with the initial state, (2) the enabling functions of the connecting transitions involve conditions on control PIs, and (3) the related update functions contain only assignments from PIs to internal registers.
- **Elaboration states.** A sequence of elaboration states is recognizable since: (1) it starts after a sequence of input states, (2) the enabling functions of the connecting transitions involve conditions on internal registers, and (3) the related update functions manipulate such registers to compute the final result without reading PIs nor writing POs.
- **Output states.** A sequence of output states is recognizable since: (1) it starts after a sequence of elaboration states, (2) the enabling functions of the connecting transitions involves conditions on control PIs, (3) the related update functions write on POs without further manipulation of internal registers, (4) the sequence of output states finishes with a transition in-going in the initial state or in an input state.

We call S^I_{CA}, S^E_{CA} and S^O_{CA} respectively the set of inputs, elaboration and output states, so that

$$S_{CA} = S^I_{CA} \cup S^E_{CA} \cup S^O_{CA}$$

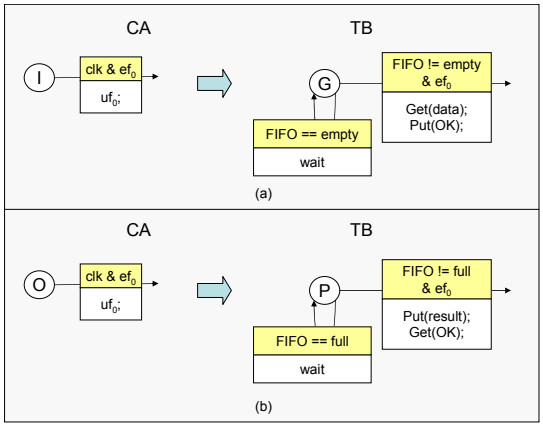

Figure 6. Abstraction of I/O states.

4.2 CA-to-TB Abstraction

The automatic abstraction from the cycle-accurate RTL design towards the transaction-based PVT TLM description is carried out by collapsing the RTL computational phases according to the three rules (input rule, elaboration rule, and output rule) formalized in the following subsections. The abstracted model is represented by a TB description modeled as an EFSM $M_{TB} = \langle S_{TB}, I_{TB}, O_{TB}, D_{TB}, T_{TB} \rangle$ where, $I_{TB} = I_{CA}$ and $O_{TB} = O_{CA}$.

4.2.1 Input Rule

For each input state $I \in S^I_{CA}$ of the CA model M_{CA}, one state G is generated for the TB model M_{TB} with two outgoing transitions (see Figure 6a). The enabling function ef_0 of the CA model is mapped into a system of transitions in the TB model, where data is got from the TLM channel (e.g., a FIFO) only if it is not empty (*FIFO!=empty*) and the enabling function ef_0 is satisfied. Otherwise, the module waits on the channel. The update function uf_0 of the CA model, which performs read operations on input ports, is mapped into a sequence of TLM function calls (*get()/put()*) that correspond to get input data from the TLM channel and to return the acknowledgment.

4.2.2 Elaboration Rule

Each sequence of states $(s_1, .., s_n)$ belonging to the same elaboration sub-phase (i.e., so that $s_i \in S^E_{CA}, i = 1, .., n$) of the CA model M_{CA}, is substituted by a single elaboration state E on the TB model M_{TB}. The state E and the corresponding in-coming and out-going transitions, are generated by recursively collapsing the CA states in accordance with the following rules (depicted in Figure 7):

1. If a state A in the CA model has a single outgoing transition to a state B (i.e., $A \rightarrow B$) whose enabling function is always true, then A and B are collapsed into a single state A', whose incoming transition has ef_0 as enabling function and the sequence of instructions included in uf_0 and uf_1 as update function (Figure 7a).

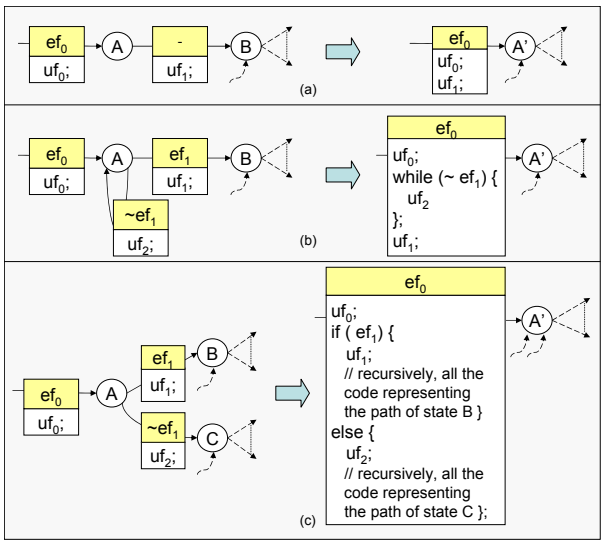

Figure 7. Basic steps for abstracting the elaboration sub-phases.

Further transitions incoming in B become incoming transition of A'.

2. If a state A in the CA model has an outgoing transition towards a state B and a transition incoming into the same state A, then A and B are still collapsed into a single state A'. However, in this case, the transition incoming into A' has a more complex form. The enabling function is ef_0, while the update function sequentializes uf_0, uf_2 and uf_1 provided that uf_2 is iteratively executed while ef_1 is false. In this way, the looping transition $A \rightarrow A$ in the CA model is implicitly represented by a while loop, as showed in Figure 7b. Further incoming transitions to B become incoming transition to A'.
3. If a state A in the CA model has two outgoing transitions towards states B and C, then A, B and C are collapsed in a single state A' in the TB model. The enabling function of the transition incoming into A' is ef_0, while the update function is composed of uf_0 followed by and *if-then-else* statement. The guard of such a statement is ef_1, while the *then* and *else* branches are obtained by recursively composing, respectively, uf_1 with the code that can be executed outgoing from B, and uf_2 with the code that can be executed outgoing from C, as showed in Figure 7c. Further incoming transitions to B and C become incoming transitions to A'.

4.2.3 Output Rule

CA output states are abstracted similarly to CA input states. Thus, for each output state $O \in S^O_{CA}$ of the CA model M_{CA}, one state P is generated for the TB model M_{TB} with two out-going transitions (see Figure 6b). The CA update function uf_0, which corresponds to write data on output ports, is mapped into a sequence of TB function calls for putting the result on the TLM channel, if it is not full, and getting the corresponding acknowledgment.

4.3 Proof of Equivalence

In this section, we prove that the abstraction methodology presented so far, is correct-by-construction with respect to the definition of event-based equivalence reported in Def. 3, i.e., we prove that whatever TB model obtained by applying the previous abstraction rules is event-based equivalent to the original CA model.

The proof relies on the fact that the abstraction rules change neither the order of events generated by the TB model with respect to the original CA model, nor the operations performed inside the update functions.

Let us consider first the elaboration rule. Such a rule cannot change the order of events since elaboration states do not generate events externally observable. In fact, the update functions involved in transition traversing the elaboration states neither perform read on input nor write on output. Furthermore, the elaboration rule does not change the order in which the code related to update function of transitions traversing elaboration states is executed. All the branch conditions of the original CA code are preserved into the collapsed TB representation. This guarantees that the functional part of the model is preserved, i.e., if input data are correctly read in the input sub-phase and output data are correctly written in the output sub-phase, then the TB model generates the same result of the corresponding CA model. Thus, the elaboration rule preserves the event-based equivalence, if the input and output rules do not wrongly affect the input and output sub-phases.

Considering the input rule, it modifies the way data are read by the TB model with respect the CA model. However, such a rule does not change the order of events generated by CA read operations when they are abstracted towards a sequence of TB *put/get* function calls. In fact, according to the definition of CA and TB events, and the definition of Φ and Ψ given in Section 3, for each CA event triggered by a read and preserved by the function Φ there is one and only one TB event triggered by the read operation of the same data on the abstracted model, preserved by Ψ. Moreover, the input rule does not affect the functional part of the model, since this is included in the elaboration states that are not considered by the input rule. Thus, the input rule preserves the event-based equivalence, if the elaboration phase does not wrongly affect the elaboration sub-phases.

The output rule is dual with respect to the input rules, and similar considerations can be applied. Thus, the output rule preserves the event-based equivalence, if the elaboration phase does not wrongly affect the elaboration sub-phases. This proves that the abstraction methodology is correct-by-construction with respect the event-based equivalence.

4.4 Implementation Details

The abstraction technique presented in [3] and summarized in Section 4.2 has been implemented in the A^2T

Design	RTL code rows#	PV code rows#	Abstr. time(s)
Root	204	391	3.55
Div	425	681	3.72
Dist	325	547	3.60
ADPCM	305	521	3.65
B01	266	456	3.51

Table 1. Details of Abstractor results.

(*Automatic Abstraction Tool*) prototype. It automatically abstracts RTL IPs towards TLM models guaranteeing the event-based equivalence presented in this paper. Experimental results have been conducted by applying the tool to several RTL IP designs:

- *Root*, *Div* and *Dist*: three industrial submodules of an STMicroelectronics SoC implementing a face recognition system [20].
- *ADPCM*: an Adaptive Differential Pulse Code Modulation (ADPCM) module of a voice over IP system.
- *B01*: a benchmark of the ITC-99 benchmark suite [21].

Table 1 shows some details on the automatic abstraction of these RTL IPs to their correspondent PV models by using A^2T. Column *RTL code lines #* reports the size of the RTL designs expressed in number of code lines. Column *PV code lines #* reports the number of code lines of the PV implementations. Finally, column *Abstr. time(s)* reports time in seconds spent by the tool for the automatic abstraction process.

5 Conclusions

The paper deals with the issue of equivalence checking between RTL and TLM models, and a formal definition of equivalence based on the concepts of event and sequence of events is given. Such a definition can be used for proving equivalence without requiring timing or structural similarities between the modules to be compared. Thus, the paper illustrates the use of the definition of equivalence in the RTL vs. TLM context. Finally, a practical use of the proposed theory is described, by proving the correctness of a methodology that automatically abstracts RTL IPs towards TLM implementations.

References

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

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

[3] N. Bombieri, F. Fummi, and G. Pravadelli. *A methodology for abstracting RTL designs into TL descriptions*. In *Proc. of ACM/IEEE MEMOCODE*, pp. 103–112. 2006.

[4] N. Bombieri, A. Fedeli, and F. Fummi. *On PSL Properties Re-use in SoC Design Flow based on Transaction Level Modeling*. In *Proc. of IEEE MTV*, pp. 127–132. 2005.

[5] 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.

[6] M. Fujita. *Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths*. ACM Trans. Des. Autom. Electron. Syst., vol. 10(4):pp. 610–626, 2005.

[7] R. Drechsler and D. Grosse. *System level validation using formal techniques*. IEE Proceedings-Computer and Digital Techniques, vol. 152(3):pp. 393–406, 2005.

[8] D. Anastasakis, R. Damiano, H.-K. Mah, and T. Stanion. *A practical and efficient method for compare-point matching*. In *Proc. of ACM/IEEE DAC*, pp. 305–310. 2002.

[9] J. R. Burch and V. Singhal. *Robust latch mapping for combinational equivalence checking*. In *Proc. of IEEE ICCAD*, pp. 563–569. 1998.

[10] A. Koelbl, Y. Lu, and A. Mathur. *Embedded tutorial: formal equivalence checking between system-level models and RTL*. In *Proc. of ACM/IEEE ICCAD*, pp. 965–971. 2005.

[11] S. Vasudevan, V. Viswanath, J. Abraham, and J. Tu. *Automatic decomposition for sequential equivalence checking of system level and RTL descriptions*. In *Proc. of ACM/IEEE MEMOCODE*, pp. 71–80. 2006.

[12] D. Kroening and E. Clarke. *Checking consistency of C and Verilog using predicate abstraction and induction*. In *Proc. of ACM/IEEE ICCAD*, pp. 66–72. 2004.

[13] W. Ecker, V. Esen, and M. Hull. *Execution semantics and formalisms for multi-abstraction TLM assertions*. In *Proc. of ACM/IEEE MEMOCODE*, pp. 93–79. 2006.

[14] L. Lamport. *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM, vol. 21(7):pp. 558–565, 1978.

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

[16] Open Core Protocol International Partnership (OCP-IP). http://www.ocpip.org.

[17] G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. *EFSM Manipulation to Increase High-Level ATPG Efficiency*. In *Proc. of ACM/IEEE ISQED*, pp. 57–62. 2006.

[18] D. Gajski, J. Zhu, and R. Domer. *Essential Issue in Codesign*. Technical report ICS-97-26, University of California, Irvine, 1997.

[19] K. Cheng and A. Krishnakumar. *Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model*. ACM Trans. on Design Automation of Electronic Systems, vol. 1(1):pp. 57–79, 1996.

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

[21] P.di Torino. *ITC-99 Benchmarks*, 1999. In http://www.cad.polito.it/tools/itc99.html.