

# Automatic Generation of Cycle Accurate and Cycle Count Accurate Transaction Level Bus Models from a Formal Model

Chen-Kang Lo, Ren-Song Tsay  
Department of Computer Science,  
National Tsing-Hua University, Taiwan

**Abstract**— This paper proposes the first automatic approach to simultaneously generate *Cycle Accurate* and *Cycle Count Accurate* transaction level bus models. Since TLM (Transaction Level Modeling) is proven as an effective design methodology for managing the ever-increasing complexity of system level designs, researchers often exploit various abstraction levels to gain either simulation speed or accuracy. Consequently, designers repeatedly perform the time-consuming task of re-writing and performing consistency checks for different abstraction level models of the same design. To ease the work, we propose a correct-by-construction method that automatically and simultaneously generates both fast and accurate transaction level bus models for system simulation. The proposed approach relieves designers from the tedious and error-prone process of refining models and checking for consistency.

## I. INTRODUCTION

As the number of components in SoC (System-On-a-Chip) increases rapidly, the multiplied data exchange rate makes system-level communication a critical design issue. Chip designers are constantly challenged to meet stringent requirements by quickly and extensively exploring design space.

Efficient design space exploration calls for fast system design, verification, and validation. TLM (Transaction Level Modeling), a modeling methodology, is proposed to support system design flow, while RTL (Register Transfer Level) model is not recommended for early-stage verification/validation due to its heavy modeling effort and slow simulation speed [1, 2].

The refinement-based TLM methodology not only provides a way for easy design simulation speed and accuracy trade-off but also opens up the possibility for early design space exploration. Fig.1 illustrates a transaction level model refinement flow of bus designs. The TL3 (Transaction Level 3) model (Fig. 1.a) is a high level abstraction model good for capturing design intention, where PEs (Processing Elements) communicate with each other by point-to-point *abstract channels*. The abstract channel is free from any specific bus protocol or topology and can model communication using high-level read/write function calls. Thus, it achieves high simulation speeds with little modeling effort. To capture more design details such as bus protocol and topology, a designer may refine a TL3 model to a bus-specific transaction level model (Fig. 1.b). It then can be further refined into to an

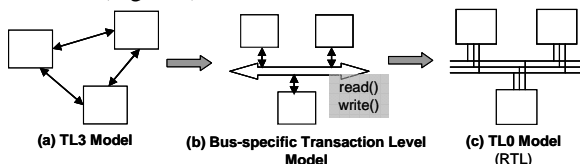


Fig. 1: A Transaction Level Modeling design flow.

RTL model (Fig. 1.c).

Despite its advantages, the selection of abstraction levels and modeling styles of a bus-specific TLM bus model was divisive in the literature since the model is sensitive to simulation speed and modeling effort [3-8]. Increasingly, more researchers [6-7] have begun to discuss how to choose appropriate abstraction levels at different design stages for different purposes and gradually refine between these proposed levels of abstraction. For instance, a designer may have a cycle accurate bus model for verification purposes and a more abstract bus model for better simulation speed and bus utilization (and contention) analysis. Consequently, designers often have to spend valuable time preparing models of different abstraction levels for the same design. Additionally, there is no good known (timing) consistency check methodology across levels. As a result, designers often spend more hours on resolving modeling issues than on real design issues.

To alleviate this problem, the authors propose a correct-by-construction approach that captures bus design in a formal model, then automatically and simultaneously generates cycle accurate *Transaction Level 1 (TL1)* and *Cycle Count Accurate TL2 (CCA-TL2)* bus models to gain both accuracy and simulation speed. The CCA-TL2 bus model gives the same cycle count timing as the cycle accurate TL1 bus model at the beginning and the end of every bus transaction; thus, it provides sufficient accuracy for bus analysis. As a result, the automatic generation of bus models relieves designers from tedious and error-prone modeling work and helps them focus on real design issues.

We will discuss related work in the next section and outline a proposed methodology in section three. Then, section four provides crucial observations and the problem formulation. These observations motivate the automatic generation algorithm proposed in the paper, which are then presented in section five. Finally, section six will give the experimental results and section seven concludes the paper.

## II. RELATED WORK

TLM is a refinement-based design methodology with a separation of communication and computation, while SystemC [1] is a modeling language specifically designed for TLM. It can model a design of various levels, from the specification level down to a detailed implementation level such as RTL. SpecC [2], yet another modeling language, has a proposed design flow with four abstraction levels for design refinement. For instance, the *Communication Model* in its methodology is a bus-specific, pin/cycle accurate model, which is refined from a high level abstract channel in the *Architecture Model*.

There is significant work on bus modeling. Some projects used C++ or SystemC to build higher abstraction level models rather than pin-accurate RTL [3-5]. For instance, Caldari et al. [5] demonstrated an AMBA model [16] in SystemC at the Bus Cycle Accurate (BCA) level, which corresponds to TL1. They used automata, or *program state machines* introduced in their paper, to model bus states in channels and applied the Interface Method Call (IMC) to model data transfer. The result was a cycle accurate bus model with invaluable losses in speed from a cycle count accurate bus model. However, all these manual bus modeling techniques referred in this paragraph attempted to gain either simulation speed or accuracy separately.

To aim at a smoother design refinement process, library-based approaches with primitives of various abstraction levels were also proposed. For instance, OSCI [9] provided primitives such as ports/interfaces for bus-independent transaction level model construction without direct support of the bus-specific model. GreenBus [7] proposed the concept of *atom* with library support to capture the basic concurrent and non-interruptible parts (or phases) of a bus protocol for TL2 and TL1 models. OCP (Open Core Protocol) [6] provides a library for modeling their own protocol at four proposed transaction levels (TL3~TL0). However, the above referred approaches are all manual refinement flows with library support, in which designers have to undergo tedious refinement work and error-prone consistency checks.

Pasricha et al. [8] proposed a manual modeling technique for the abstraction level, Cycle Count Accurate at Transaction Boundaries (CCATB), which eliminates the intermediate states of a bus transaction (intra-transaction visibility) to enhance simulation speed while maintaining accurate *cycle count information*. In comparison, our proposed automatic algorithm can precisely and systematically eliminate unnecessary modeling details while achieving the same performance and accuracy.

Shin et al. [10] and Vercauteren et al. [11] proposed an automatic refinement method which starts from abstract channel models described in message passing semantics. The algorithm preserves the semantics of abstract channels and generates pin/cycle accurate bus models from a given library. Nevertheless, there was no automatic transaction level bus model generation such as we propose in this paper. Coware [13] suggested an automatic TL1 bus model generation from a formal description and helped alleviate the time-consuming modeling work faced by designers. Nevertheless, they could not automatically generate the TL2 model and focused on integrating the generated bus model into MP-SoC (Multi-Processors SoC).

D'silva et al. [14] proposed *Synchronous Protocol Automata (SPA)* for verifying bus property, such as protocol compatibility between master and slave interfaces. We also adopt SPA to construct formal models. However, we have further devised an automatic algorithm for systematic transaction level bus model generation using SPAs for protocol description.

### III. METHODOLOGY

#### A. Overview of Transaction Levels

We follow the definitions from OCP [6] to introduce transaction levels. There are four transaction levels defined by OCP from TL3 to TL0; each one is suitable for a different purpose with different data granularity and protocol timing in system level refinement flow. Interfaces for data transfer in transaction

levels (TL3 to TL1) are different from RTL (TL0), since transaction level bus models are notable of transferring data by IMC (interface method call) instead of signals. Also, note that we add an additional level, *Cycle Count Accurate TL2 (CCA-TL2)*, which will be automatically generated by our algorithm. The CCA-TL2 level differs from TL2 by its cycle count accurate timing while the TL2 proposed by OCP only has approximate timing.

Abstraction Level	Purpose	Data Granularity	Protocol Timing	IF
TL3	algo.	abstract data type	none	IMC
TL2	arch.	burst size	approx.	IMC
CCA-TL2	arch.	burst size	CCA	IMC
TL1	protocol	bus size	CA	IMC
TL0 (RTL)	impl.	bus size	CA	signal

Tab. 1: Summary of Transaction Levels

#### B. Modeling Methodology

After capturing the preliminary design intent by using TL3 abstract channels, decisions about the bus topology and bus interface are two main factors in refining the communication part of the design (Fig. 2). Designers can describe the bus interfaces of each computation module using SPA formal models. Then, the CCA-TL2 and TL1 models are simultaneously generated in SystemC by applying our proposed automatic algorithm. As a result, designers can have both a lower abstraction level model (TL1) to verify design details in a *cycle accurate* manner and a higher abstraction level model (CCA-TL2) for fast system simulation in a *cycle count accurate* manner for architecture exploration.

Each generated TL1 model has the same bus states as the input *SPA-pair*, which stands for a pair of master and slave interfaces described in SPA. For a TL1 model, each clock tick advances one simulation time unit and each bus state corresponds to one clock cycle time. In contrast, a generated CCA-TL2 model has fewer states compressed from an input SPA-pair and it advances simulation time in a wait-for-time style (wait for a number of clock cycles) with transferring a pack of data; it thus achieves a higher simulation speed.

After (communication) architecture exploration, designers can further refine the design to the TL0 model [12]. In the following section, we will focus on the automatic generation of TL1 and CCA-TL2 models.

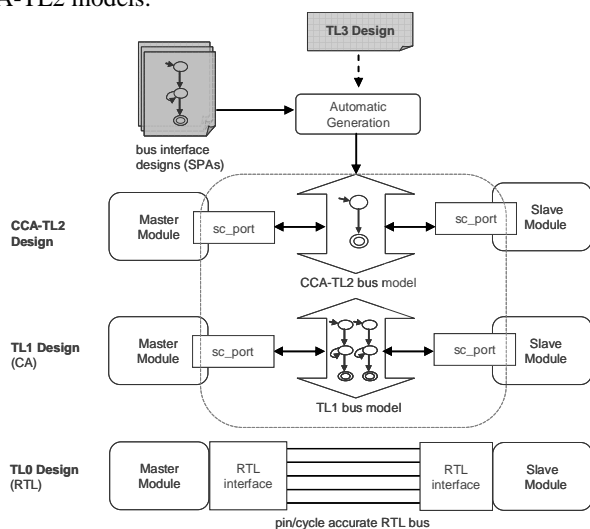


Fig. 2: The proposed design flow.

## IV. PROBLEM FORMULATION

### A. Bus Transaction Formal Modeling

In general, a bus transaction is a read/write data transfer between a master and a slave interface. Before discussing formal modeling, we first use a simple example to illustrate how a SPA-pair can describe a bus transaction (Fig. 3).

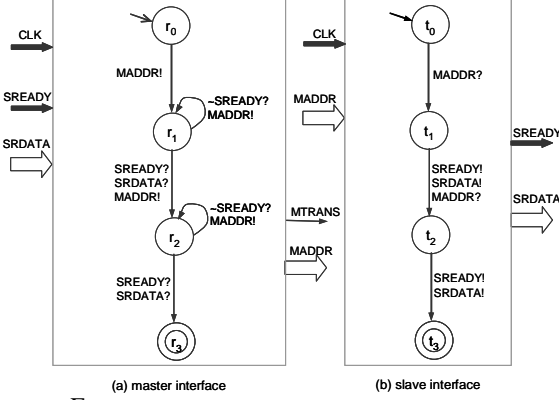


Fig. 3: An SPA-pair bus transaction example.

As an example in Fig. 3, both master and slave interfaces start from initial states,  $r_0$  and  $t_0$ , respectively. At the first clock tick, the master writes data onto a *data signal* MADDR (denoted MADDR!) and progresses from  $r_0$  to  $r_1$ . Simultaneously, the slave reads data from the MADDR (MADDR?) and progresses from  $t_0$  to  $t_1$ . At next tick, the master checks the *control signal* SREADY (SREADY?) to decide whether to stay looping at  $r_1$  or move on to  $r_2$ . In this case, the master will progress from  $r_1$  to  $r_2$  to read the data on SRDATA (SRDATA?) since the slave will assert SREADY (SREADY!) to declare readiness to transfer data on SRDATA (SRDATA!). Therefore, a transaction can be viewed as a sequence of state-pairs:  $(r_0, t_0)$ ,  $(r_1, t_1)$ ,  $(r_2, t_2)$ , and  $(r_3, t_3)$  with data read/write operations, where the state-pair  $(r_x, t_x)$  denotes that at the same time the master is in state  $r_x$  and the slave is in state  $t_x$ .

To further our discussion, we modify from [14] and define some terms and notations in the following.

**Definition 1:** A *Synchronous Protocol Automata (SPA)* is a tuple  $(Q, D, C, A, V, \rightarrow, \text{clk}, q_0, q_f)$ , where

1.  $Q$ : a finite set of control states.
2.  $D, C$ : a set of disjoint input or output *data* and *control* signals.
3.  $A$ : a set of actions.
4.  $V$ : a set of internal variables
5.  $\rightarrow \in Q \times Q \times A \times \text{clk}?$ : transition relations
6.  $q_0, q_f \in Q$ : initial state and final state

In other words, a SPA comprises a set of finite control states  $Q$ , which includes an initial state  $q_0$  and a final state  $q_f$  with a set of disjoint input or output data signals,  $D$ , and control signals,  $C$ . While data signals can be of any type, control signals are usually Boolean. Additionally, each signal has to be associated with a read or a write event, which is triggered by a corresponding read/write operation.

An action  $S \in A$  is of the form  $G:N$ , *guards* followed by *non-blocking operations*, where  $G$  is a set of guards (or blocking operations) that check for the assertion of a control signal  $c$  ( $c?$ ) or check for the output value of a logic expression  $b$  ( $b?$ ) comprising data signals and internal variables.  $N$  is a set of non-

blocking operations, which are *writes on a control signal*  $c$  ( $c!$ ) or *reads and writes on a data signal*  $d$  (denoted  $d?$  and  $d!$ , respectively).

A transition is written as  $r_x \xrightarrow{S} r_y$ , where  $r_x, r_y \in Q, S \in A$ . State  $r_x$  transits to state  $r_y$  when action  $S$  is *satisfied* and clock tick triggered. We use a Boolean function  $\text{canProgress}(S_1, S_2)$  to indicate whether both actions  $S_1$  and  $S_2$  are satisfied. It returns true when all guards of the two actions are evaluated true, otherwise false.

**Definition 2:**  $\text{CanProgress}(S_1, S_2)$  is true when the following conditions hold: (1) any  $c \in S_1, S_2$  such that  $c? \in S_1$  and  $c! \in S_2$ , where  $c \in C$ , and vice versa, (2) any  $b \in S_1, S_2$  such that  $b$  is evaluated true, where  $b$  is a logic expression.

Last but not least, a *compatible SPA-pair* is a pair of SPAs which has a legal (according to the definition of state progression) sequence of state-pairs. In addition, two compatible SPAs are required to start from initial states and end at final states simultaneously without non-deterministic transition in actual executions. For information about compatibility check, readers may refer to [14]. Then, a bus transaction can be defined in the following.

**Definition 3:** A *bus transaction* is a sequence of state-pairs, from the initial state-pair to the final state-pair, in a compatible SPA-pair describing legal data transfer.

### B. Observations

Following the design flow and formal model, there are three observations that motivate the idea for automatically raising abstraction level from a cycle accurate model (TL1) to a cycle count accurate model (CCA-TL2).

1. A bus transaction is repeatedly executed because of hardware nature. In other words, it often follows the same sequence of state-pairs although the data transferred varies.
2. A computation module concerns only the data content transferred, but not how data are transferred or the intermediate states of the bus transaction [8].
3. Most routine transitions of SPA-pair can be *pre-determined* (know which transition will be taken or not) at static time (when bus model is known).

These lead to an idea for speeding up system simulation using higher abstraction level models that *compress* intermediate transitions based on static-time bus transaction analysis.

### C. Problem Formulation

Therefore, the problem can be formulated as the following: *Given an SPA-pair, determine a Compressed Automaton (CA) that has the same "cycle count timing" as the two given interfaces. Subsequently, generate a TL1 bus model according to the SPA-pair and then a CCA-TL2 bus model according to the CA.* Here, a *Compressed Automaton* is an extended SPA, as shown in Fig. 4, where each transition edge has a weight number representing the corresponding clock tick count that the CA takes (or waits) before state progression. With this, the generation algorithm can be presented.

## V. TRANSACTION LEVEL BUS MODEL GENERATION

The transaction level bus model generation has two steps. First, the *Compression Algorithm* compresses an input SPA-pair and generates a Compressed Automaton (CA). Then, the TL1 and CCA-TL2 bus models can be created accordingly in SystemC.

The observations in section four motivate the Compression Algorithm. When simulating bus transactions at run time (actual execution), a simulator takes significant scheduling overhead managing event queue and sensitizing/trigging events for routine transitions of repeated transactions. Thus, the main idea of the Compression Algorithm is to *compress* the *pre-determined* sequence of consecutive transitions, which is *not of concern to computation modules but is repeatedly executed* by the simulator, into a new *simplified* transition in CA. The algorithm consolidates the data operations of the compressed transitions in SPA-pair with using a weight number to count the number of transitions compressed (see the example in Fig. 4.a). As a result, the simplified transitions in CA are then executed at run time while eliminating most scheduling overhead of the routine transitions at every clock tick.

However, not all transitions are pre-determined. Then, instead of compressing the non-predetermined transitions, the algorithm reserves the non-predetermined guards of these transitions (that cannot be evaluated at static time) into the simplified transitions in CA to maintain the cycle count accuracy.

Next, we will discuss the non-predetermined transitions by three cases, the Compression Algorithm, and the SystemC bus model generation.

### A. Three Bus Transaction Cases

According to the causes of the non-predetermined transitions, there are three bus transaction cases: the *constant case*, *data-dependent case*, and *control-dependent case*.

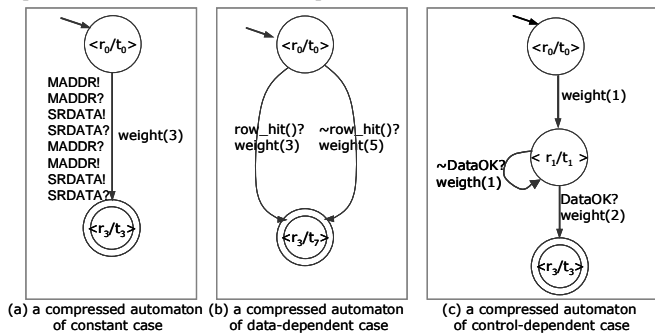


Fig. 4: Examples of output Compressed Automata<sup>A</sup>.

#### 1) The Constant Case

The whole transition sequence of the bus transaction in the constant case can be pre-determined, and thus the constant case bus transaction, after it is initiated by computation module, always takes a constant number of cycles (or transitions) to complete data transfer. For the example in Fig. 3, the number of cycles required to transfer two beats (words) of data from the initial state-pair to the final state-pair can be determined at static time. The generated CA is shown in Fig. 4.a. Note that we denote a generated state in CA as  $\langle r_x/t_y \rangle$  to indicate that the state is generated corresponding to a state  $r_x$  in a master interface and a state  $t_y$  in a slave interface. For the example,  $\langle r_0/t_0 \rangle$  is an initial state of the CA and is generated corresponding to a initial state  $r_0$  in a master interface and an initial state  $t_0$  in a slave interface.

#### 2) The Data-Dependent Case

Some transitions cannot be pre-determined at static time because their guards are evaluated depending on the values of some data operations, such as the address of the data fetch from a

processor. The transition sequence of the bus transaction in the data-dependent case can be compressed into branches (Fig 4.b), and each branch is annotated with the corresponding guards, which cannot be evaluated at static time, and with a cycle count number. Nevertheless, for the compressed branch model, the bus transaction cycle count is *instantly computable* once the branch taken is determined at run time.

For example, consider a SPA-pair comprising a typical (slave) RAM interface in Fig. 5.a and the master interface in Fig. 3. The cycle count of the bus transaction varies depending on the value of the data operation MADDR!. When a master interface initiates a bus transaction, the slave interface calls *row\_hit()* (a Boolean expression) to check whether the previous row address, stored in an internal variable, is of the same row address decoded from MADDR. If the address requested hits the same row, the slave interface will directly fetch from the buffer by following the left branch; otherwise, it follows the right branch and fetches the data requested. The generated CA is shown in Fig. 4.b.

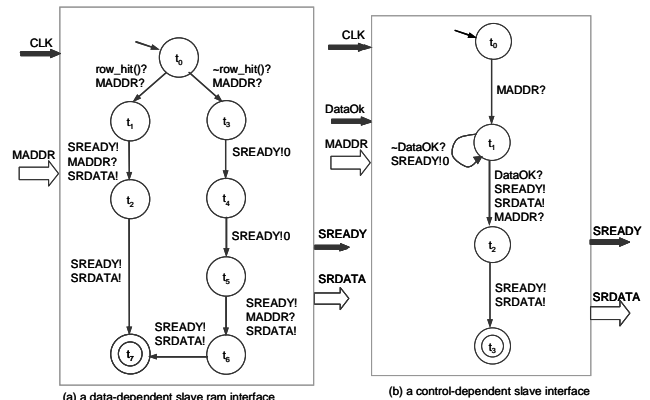


Fig. 5: Examples of data- and control-dependent slave interfaces.

#### 3) The Control-Dependent Case

For the control-dependent case, the bus transaction cycle count neither can be pre-determined at static time nor instantly at run time if there are guard evaluations depending on control operations from the computation modules to determine which transition to progress the states. Take the slave interface in Fig. 5.b as an example. The state after state  $t_1$  depends on an outside control signal *DataOK*.

Nevertheless, for this case, the segments not affected by the control signals still can be compressed. For instance, the generated CA shown in Fig. 4.c is a result of the SPA-pair of the master interface in Fig. 3 and the slave interface in Fig. 5.b.

After discussions of the three cases, we are ready to present the algorithm.

### B. The Compression Algorithm

#### 1) The Compression Algorithm

The pseudo code of the Compression Algorithm is listed below with a compatible SPA-pair, M and S, as input and a generated CA as output. The PendingTrans is a queue of  $\{r, t, S_1, S_2\}$ , where  $r \in M.Q$ ,  $t \in S.Q$ ,  $S_1: r \xrightarrow{S_1}$ ,  $S_2: t \xrightarrow{S_2}$ , to record current pending transitions. Also, the variable *count* is used to record the number of transitions compressed and the action S' the operations consolidated.

<sup>A</sup> data operations in Fig. 4.b and Fig. 4.c are eliminated for clarity.

The algorithm begins with setting the initial value of the PendingTrans (line 8), which is set by adding the initial state  $r_0$  and  $t_0$  along with all progress-able action-pairs, sourced from  $r_0$  and  $t_0$ , by following the definition of *CanProgress()* in section 4. Then, the *inner while loop* (line 13 to 25) traverses by following the transitions in PendingTrans one at a time, and compresses a pre-determined sequence of transitions of the input SPA-pair. At the end of the algorithm (line 26 to 35), three cases of conditional branches generate states or compressed transitions into the CA, and the outer while loop checks whether the queue is empty as a halting condition of the algorithm.

#### Algorithm - Compression(M,S)

```

1. M, S: input SPA;
2. CA : output Compressed Automaton with initial state  $\langle r_0 / t_0 \rangle$ ;
3. count : integer = 0;
4. progressing : Boolean = true;
5. S' : an action =  $\emptyset$ ;
6. PendingTrans: a queue of  $\{r, t, S_1, S_2\} = \emptyset$ ;
7. Begin
8. PendingTrans.init(); // setting initial value of PendingTrans
9. while PendingTrans  $\neq \emptyset$  do // main body of the algorithm
10.    $\{r, t, S_1, S_2\} = \text{PendingTrans.pop}()$ ;
11.   set  $r' = r, t' = t, S' = \emptyset$ , and count = 0;
12.   progressing = true;
13.   while progressing do
14.     progressing = false;
15.      $r' = \text{next\_state}(r', S_1)$ ;
16.      $t' = \text{next\_state}(t', S_2)$ ;
17.      $S' = S' \cup \text{compress\_act}(S_1, S_2)$ ;
18.     count++;
19.     for each  $S_1' : r' \xrightarrow{S_1}$ , and  $S_2' : t' \xrightarrow{S_2}$  do
20.       if canProgress( $S_1', S_2'$ )
21.         progressing = true;
22.         set  $S_1 = S_1'$  and  $S_2 = S_2'$ ;
23.       end if
24.     end for
25.   end while //end of inner while loop
26.   if state  $\langle r' / t' \rangle$  is in CA
27.     add transition  $\langle r / t \rangle \xrightarrow{S', \text{count}} \langle r' / t' \rangle$  into CA;
28.   else if  $\langle r' / t' \rangle = \langle r_i / t_i \rangle$ 
29.     add state  $\langle r' / t' \rangle$  into CA;
30.     add transition  $\langle r / t \rangle \xrightarrow{S', \text{count}} \langle r' / t' \rangle$  into CA;
31.   else
32.     add state  $\langle r' / t' \rangle$  into CA;
33.     add transition  $\langle r / t \rangle \xrightarrow{S', \text{count}} \langle r' / t' \rangle$  into CA;
34.     resolve( $r', t'$ );
35.   end if-else
36. end while //end of outer while loop
37. End

```

Fig. 6 shows the traces of the SPA-pair and the CA obtained by using the constant case example in Fig. 3 as an example to illustrate the algorithm. At the beginning, the two output transitions of the initial states are chosen and inserted into the queue PendingTrans since there are no guards in their actions (Fig. 6.a). Then the algorithm progresses the state-pair, consolidates the data operations of  $S_1$  and  $S_2$  into the action  $S'$  and increments the variable *count* (Fig. 6.b). Following this, another two transitions are taken to progress to a new state-pair. The transition in the master interface with the guard SREADY? is taken, since the operation SREADY! in the slave interface will make the guard evaluate true (Fig. 6.c). Then the algorithm keeps traversing the state-pairs until it reaches the final state-pair  $\langle r_3, t_3 \rangle$ . Subsequently, the generated state  $\langle r_3 / t_3 \rangle$  and transition, with the compressed action  $S'$  and *count*, are added into the CA (Fig. 6.d).

In contrast to the constant case examples, which traverse directly to the final state-pair, non-predetermined transitions of the

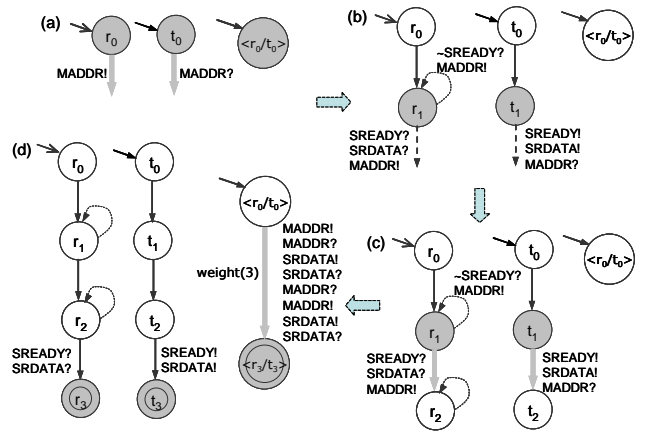


Fig. 6: Algorithm illustration with a constant case example.

data- and control-dependent cases may block the traverse. When the *inner while loop* encounters a non-predetermined transition, it ends and the algorithm then calls a subroutine *resolve(r,t)* to continue traversing each possible *branch*. The subroutine *resolve(r,t)* temporally disables those non-predetermined guards and continues searching pairs of progress-able transitions again. Each pair of transitions which can be progressed is inserted into PendingTrans and the algorithm continues traversing by following each of them. Note that the subroutine *compress\_act(S<sub>1</sub>, S<sub>2</sub>)* not only merges the data operations but also reserves the non-predetermined guards into CA.

### 2) Complexity

The complexity of Compression Algorithm is  $O(m*n)$ , where  $m$  and  $n$  are the number of states of master and slave interfaces, respectively. The worst case happens when all the combinations of  $m$  states in a master interface and  $n$  states in a slave interface (i.e.  $\langle r_i, t_j \rangle, \forall i, j$ ) are traversed by the algorithm. Nonetheless, the situation rarely happens and normally the algorithm runs in linear time, because most bus protocols are designed in the handshake style for data transfer. Moreover, note that what the algorithm does (compressing the SPA-pair and collecting data operations) is actually the same as what the bus simulation executes (progressing the SPA-pair and transferring data) at run time, although the compressing is done beforehand at static time.

### 3) Correctness

The generated CA has the same cycle count as the input SPA-pair for timing correctness, while with the same data operations for functional correctness, between the beginning and the end of the described transaction. For functional correctness, both CA and SPA-pair have the same data operations since the algorithm compresses the SPA-pair without eliminating any data operation. For timing correctness, both CA and SPA-pair have the same cycle count between the beginning and the end of the transaction. Considering state-pairs that have no (output) non-predetermined transitions, the cycle count of the CA is the same as that of the SPA-pair because both follow the same sequence of state-pairs, which is known at static time, to complete the transaction. Considering state-pairs which have non-predetermined transitions, the CA will choose the same sequence of state-pairs, while encountering the non-predetermined transitions at run time, to continue the transaction because all the non-predetermined guards are reserved from the SPA-pair by the algorithm.

### C. SystemC Model Generation

Finally, SystemC bus models, implemented in typical SystemC interfaces and channel patterns [1], are generated according to the SPA-pair and the resulted CA in order to be integrated into system verification tools. Additionally, the signals in SPA are translated into variables, according to the declarative type, and read/write events in SystemC with data operations implemented as IMC. Master and slave modules can sensitize the events to transfer data from IMC, while the events are triggered by corresponding operations when the transition is taken.

While the TL1 bus model is clock-driven, the CCA-TL2 bus model is of event-driven style. When each CA transition is taken in simulation, the CCA-TL2 bus model transfers packed data according to the action of the transition and then advances clock cycle time by the cycle count number calculated by the Compression Algorithm, through calling the SystemC wait-for-time method.

## VI. EXPERIMENTAL RESULTS

Experiments are run on a Linux 64-bit workstation using Intel 3.40 GHz Xeon CPU. We run three proprietary testcases (one for each of the three discussed cases); each of which comprises one master and one slave transferring data. For the experiment, the core protocol, *burst write with handshake*, from OCP-IP [15] is chosen, and burst sizes of 2, 4, 8, and 16 data beats are tested. The system configuration is given to highlight the speed comparison of bus models; however, we do not foresee any difficulty integrating a ISS (Instruction Set Simulator) or a hardware IP into our system.

The experiments compare the speed of the generated TL1 and TL2 models with manual-written TL1 models constructed by the OCP-provided library [6]. For accuracy, the cycle count timing of the TL2 model is verified against with its corresponding TL1 model at the beginning and end of every bus transaction. In Fig. 7, the CO-TL1 denotes Constant case OCP TL1, CDG-TL2 denotes Control-Dependent case Generated TL2, and so on. The y-axis is the throughput measured by number of bytes per second for data transfer on the host machine (MB/Sec.) and the x-axis is the burst size. In comparison with throughput, while the generated TL1 models performs almost the same with the manual-written TL1 models, the generated TL2 model becomes 3 to 14 times faster than the manual-written TL1 model as the beat number increases from 2 to 16.

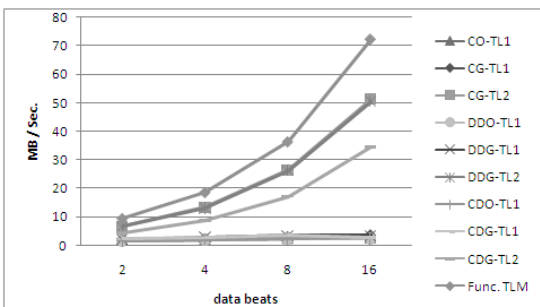


Fig. 7: The experimental results.

The speed improvement is from reduced run-time scheduling overhead (of the SystemC simulator) and decreased number of IMC. More transitions are compressed by the algorithm more the run-time overhead is reduced; hence the throughput of TL2

model (CDG-TL2) is increasing along with the increasing burst size. That also explains why control-dependent testcases, which have a state designed to wait an outside signal as in Fig. 5.b, possess lower throughput compared with their corresponding constant and data-dependent cases. Finally, note that we do not compare the generated CCA-TL2 models with OCP TL2 models because their models have no cycle count accurate timing. Instead, a functional TLM model (without any timing) is reported.

Our automatic generation approach takes almost no time (0.01 sec) to generate bus models, which alone already can save at least 30 hours, estimated by the Line of Code metric. Additionally, the TL2 model is correct-by-construction, saving designers from time-consuming model consistency checks.

## VII. CONCLUSION AND FUTURE WORK

This paper proposes the first automatic approach to generate both fast and accurate transaction level bus models of multiple abstraction levels simultaneously from a formal model, and it inherently eliminates the inconsistency problem. In addition, the paper is the first one that formally discusses what and how information between different bus-specific transaction levels can be simplified (or compressed). Finally, the experiments demonstrate that the automatic generation of the CCA-TL2 models gains invaluable simulation speed improvement while maintaining cycle count accuracy.

The multiple masters and multiple slaves without an arbiter can be considered as pairs of masters and slaves to apply the same approaches in the paper. An automatic bus model generation approach for cases of multiple masters and slaves with an arbiter will be the next topic in the future to be considered.

## REFERENCES

- [1] T. Grötker, S. Liao, G. Martin, S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.
- [2] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.
- [3] S. Malik, X. Zhu, "A hierarchical modeling framework for on-chip communication architectures", *Proc. Computer-Aided Design*, 2002, pp. 663-671.
- [4] O. Ogawa, et al., "A practical approach for bus architecture optimization at transaction level", *Design, Automation and Test in Europe Conf.*, 2003, pp. 176-181.
- [5] M. Caldari, et al., "Transaction-level models for AMBA bus architecture using SystemC 2.0", *Design, Automation and Test in Europe Conf.*, 2003, pp. 26-31.
- [6] A. Harverinen, M. Leclercq, N. Weyrich, D. Wingard, "A SystemC™ OCP Transaction Level Communication Channel", Technical Report'07.
- [7] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntsev, and M. Burton, "GreenBus: a generic interconnect fabric for transaction level modeling", *Proc. Design Automation Conf.*, 2006, pp. 905-910.
- [8] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration", *Proc. Design Automation Conf.*, 2004, pp. 113-118.
- [9] Open SystemC Initiative (OSCI). SystemC Version 2 Documentation. <http://www.systemc.org>.
- [10] D. Shin, A. Gerstlauer, J. Peng, R. Dömer, and D. Gajski, "Automatic generation of transaction level models for rapid design space exploration," *Proc. Hardware/Software Codesign and System Synthesis*, 2006, pp. 64-69.
- [11] S. Vercauteren, B. Lin, and H. D. Man, "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications," *Proc. Design Automation Conf.*, June 1996, pp. 521-526.
- [12] Jon. D. Kleinsmith and D. Gajski, "Communication Synthesis for Reuse", UC Irvine, ICS-TR-98-06, March 1998.
- [13] T. Michiels, "Generating TLM bus models from formal protocol specifications", *European SystemC Users Group Meeting*, 2004.
- [14] V. D'silva, S. Ramesh, and A. Sowmya, "Synchronous Protocol Automata: A Framework for Modeling and Verification of SoC Communication Architectures", *Proc. Design, Automation and Test in Europe*, 2004, pp. 20-27.
- [15] Open Core Protocol International Partnership (OCP-IP), [www.ocpip.org](http://www.ocpip.org).
- [16] D. Flynn. "AMBA: enabling reusable on-chip designs", *IEEE Micro*, 1997, pp. 20-27.