Communication Protocol Analysis of Transaction-Level Models using Satisfiability Modulo Theories

Che-Wei Chang

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
e-mail: cheweic@uci.edu

Rainer Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
e-mail: doemer@uci.edu

Abstract—A critical aspect in SoC design is the correctness of communication between system blocks. In this work, we present a novel approach to formally verify various aspects of communication models, including timing constraints and liveness. Our approach automatically extracts timing relations and constraints from the design and builds a Satisfiability Modulo Theories (SMT) model whose assertions are then formally verified along with properties of interest input by the designer. Our method also addresses the complexity growth with a hierarchical approach. We demonstrate our approach on models communicating over industry standard bus protocol AMBA AHB and CAN bus. Our results show that the generated assertions can be solved within resonable time.

I. Introduction

In system-level design, a transaction level model (TLM) describes the system components, their abstract computation behavior, and in particular the system communication over busses at an abstract functional level. Typically, the functionality and timing of a TLM is validated through simulation. In this paper, we formally verify the model and propose a method using Satisfiability Modulo Theories (SMT) [4] to statically analyze the TLM and verify features of interest. In particular, our main focus here is on the timing constraints in the communication protocols. As illustrated in Fig 1, we perform multiple rounds of verification using SMT, following a designer-in-the-loop methodology.

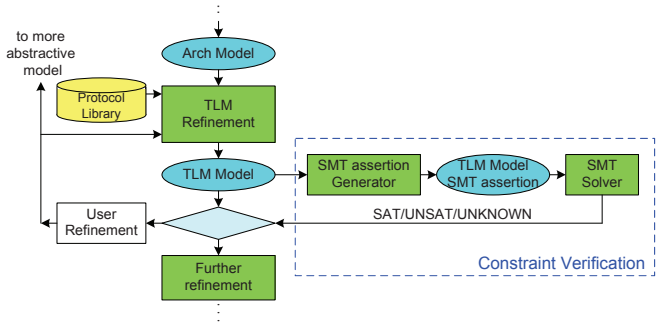


Fig. 1. Refinement methodology with static constraint analysis

Based on the given execution semantics of Discrete Event Simulation (DES), our proposed approach extracts the timing relations specified in the design model and converts them into assertions as input for the SMT solver[1]. The SMT solver checks the satisfiability of the assertions and reports the result to the system designer. If the assertions are satisfiable, the SMT solver can provide a detailed report of the symbol assignments which make the assertions true. On the other hand, if the model is found unsatisfiable, the SMT solver reports the conflicting assertions leading to the unsatisfiability. Based on the result, the system designer can determine whether or not the TLM satisfies the desired design requirements, as well as where the design fails the requirements, if so.

A. Designer Augmented Assertions

Our proposed methodology in this paper is to automatically extract timing relations and constraints from a given design and build a corresponding SMT model as verification framework. Then the designer can verify the properties of interest on the framework by augmenting the SMT model with assertions reflecting his points of interest. For example, to verify that the execution time of the application is always less than 100 time units, the designer can augment the SMT model with an assertion asking "$Can\ the\ execution\ time\ be\ more\ than$ 100 $time\ units$?". If this is found unsatisfiable, the application *will* execute in 100 time units or less, taking all conditions into account. In other words, the execution time is proven to meet the timing constraint. On the other hand, if it is found satisfiable, the tool will also lists the conditions satisfying the assertions so that the designer can examine the situation.

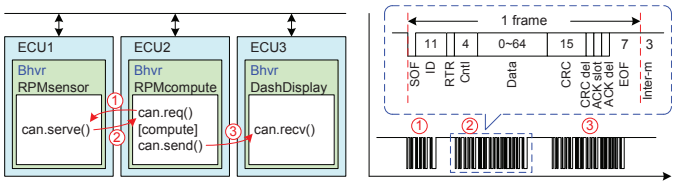
B. TLM with Communication Timing

In a top-down system design flow, the system architecture model is further refined into a TLM. The main objective of TLM refinement is to choose and parameterize a bus protocol to implement the communication between the processing elements in the system. The communication protocol is specified by the inserted transaction level bus model which specifies the detailed communication timing, including synchronization and delays compliant to the chosen protocol. Compared with the previous model, TLM with communication timing better represents the real-world design in which communication does take time. The communication timing also has great influence on the execution time and even liveness of the design. Fig 2 shows two TLM examples and their corresponding timing diagrams. In both designs, the corresponding timing information for AMBA Advanced High-performance Bus (AHB) and Controller Area Network (CAN) bus protocol is specified in the TLM channel[10]. By formally extracting the timing rela-

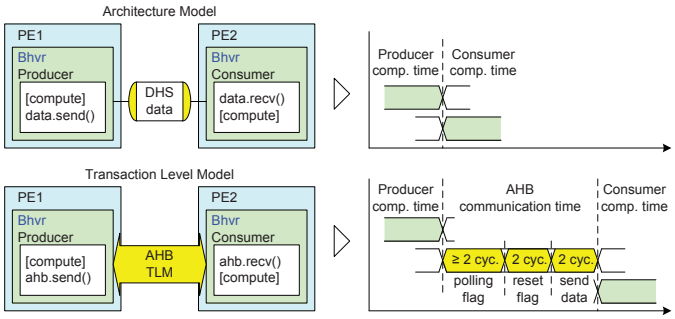
[1] We use Z3 theorem prover [3] developed by Microsoft Research.

978-1-4799-7792-5/15/$31.00 ©2015 IEEE

tions from the TLM and checking these with the SMT solver, our proposed method can *verify* the meeting of the timing constraints with the selected architecture and bus protocol.



(A) Timing diagram for CAN bus TLM of a RPM Display example



(B) Timing diagrams for architecture model and TLM of a Producer-Consumer example

Fig. 2. TLMs with detailed communication timing

C. Related Work

There is significant work in the realm of formal verification of system-level design, and one research method in this area is to convert the semantics of a behavioral model into another well-defined representation and make use of existing tools to validate the properties of interest. In [8] designs in SystemC are transformed into UPPAAL time automata and verified by UPPAAL model checker; in [7] and [6] a method to convert SystemC into state machines for verification is proposed; [9] proposed to translate SystemC models into a Petri-net based representation for embedded systems (PRES+) for model checking; [2] proposes a multi-layer modeling to represent SystemC design in a predictive synchronization dependency graph (PSDG) and extended Petri net is proposed for formal deadlock checking. [1] translates SystemC to Kripke structure and applies symbolic model checking for verification. In contrast, Our approach acts as an interactive property checking tool which brings uncertainties about critical properties and corner cases to the attention of the designer. In our method, the designer verifies points of interest by adding corresponding assertions to the extracted SMT model. The satisfiability report obtained through our method highlights often special situations, such as missed acknowledge signals or unsatisfied condition, and assists the designer in verification of the model for all cases. In this paper, we use SpecC[2] language [11] to create the system level model.

II. Timing Relations in Models

A. Time Interval Model

In system design, functionality is not the only concern. Timing constraints are critical as well, especially for real-time systems and communication protocols. Therefore, the notion of time is an important aspect of the model. In this paper, we use a time interval $\langle T_{start}, T_{end} \rangle$ [5] to represent the start and end time of the execution of a statement s in the model. To properly reflect the discrete event semantics with delta cycles, we make every time stamp a 3-tuple $\langle Time(t), Delta(d), Order(o) \rangle$. Note that we use the third member, called *order*, to distinguish statements that otherwise happen at the same time and delta cycle. The ordering is determined based on the timing relation between statements and assigned automatically by the solver. For such time stamps, we define a set of operations as listed in Table I, describing the relations equality and greater-than, as well as time advance by wait-for-time.

TABLE I
Operations of Time Stamp

$Operation$	$Definition$
$T_A = T_B$	$T_{A.t} = T_{B.t}, T_{A.d} = T_{B.d}, T_{A.o} = T_{B.o}$
$T_A > T_B$	$T_{A.t} > T_{B.t}$ or $T_{A.t} = T_{B.t}, T_{A.d} > T_{B.d}$ or $T_{A.t} = T_{B.t}, T_{A.d} = T_{B.d}, T_{A.o} > T_{B.o}$
T_A waitfor N	$T_{A.t} = T_{A.t} + \text{N}, \quad T_{A.d} = 0$

Exact timing, such as delay or execution time of computation and communication, can be specified by using wait-for-time statements that carry a time argument of integral constant type. When a wait-for-time statement is executed, the current behavior is suspended from execution for the specified time. Fig 3(A) shows an example with `waitfor`, `wait`, and `notify` statements. Here, statement A is executed val time units before statement B. Formally "$T_{start.t}(stmnt\_B) = T_{end.t}(stmnt\_A) + val, T_{start.d}(stmnt\_B) = 0$" will be generated in the SMT model.

B. Timing Constraints

Minimum or maximum bounds on the time between two statements in the model are called timing constraints. To meet real-time constraints imposed on the application by the environment, e.g. for communication, such constraints need to be specified with the design model so that it can be implemented accordingly.

In the SpecC language [11], timing constraints can be specified in the model with a special **do-timing** construct, with which the timing constraints can be checked during simulation or, in our case, be extracted to assertions for formal verification. The syntax of timing constraints contains two parts: the `do` block specifies a set of labeled statements, whereas the `timing` block contains the actual constraints. In the `do` block, the statements whose timing the designer wants to check are given a unique label and in the following `timing` block the labels are used to set the constraints. Constraints are specified with the `range` construct, which takes four arguments. The first two arguments specify the labels and the last two the lower and upper bounds of the timing constraint, respectively. A **do-timing** example is shown in Fig 3(B). There are three labels in the `do` block, and two constraints are specified with `range` constructs in the `timing` block. Note that label *L2* is attached to a compound statement which contains two child behavior calls. The following condition must hold for the constraints in this example:

$$0 \leq T_{start.t}(L2) - T_{start.t}(L1) \leq 100$$
$$0 \leq T_{start.t}(L3) - T_{start.t}(L2) \leq 300$$

[2]Due to its similarity, our results are equally applicable to SystemC

Since $T_{start}(L1) = T_{start}(i\_A)$ and $T_{start}(L2) = T_{end}(i\_A)$, the first condition limits the execution time of *i_A*, and the second sets the constraint for the total execution time of *i_B* and *i_C*.

III. Timing Relation Extraction

A system model is composed of multiple computation blocks (modules, behaviors) with communication (channel) between them. We distinguish two types of behaviors: 1)**Leaf** and 2)**Hierarchical** behavior, which implements the computation and specifies the composition of instances respectively.

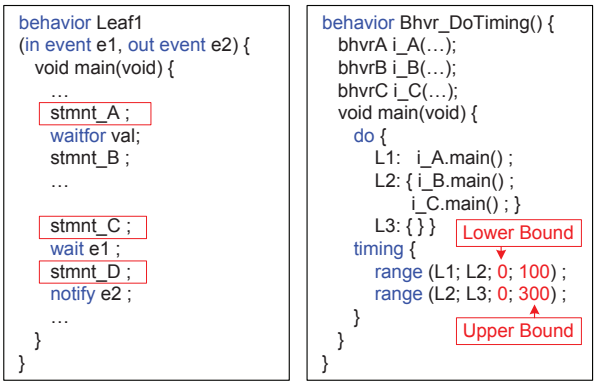


Fig. 3. Two types of timing specification in SpecC language[11]

In our proposed method, we utilize the logic of uninterpreted functions with linear arithmetic (QF_LIA) which incorporates the `Core` and `Ints` theories to generate the assertions. The `Core` theory contains the basic types and operations for Boolean logic, and the `Ints` theory defines the integer type and basic functions for integer arithmetic and comparison. We use a function symbol (in SMT-LIB2 language) to represent each time stamp in the model and convert the timing relations between those stamps into assertions. For a newly introduced function symbol, the user can define the number of arguments, and the data type of the argument and the return value. In our method, the return value of an uninterpreted function is seen as the value of a time stamp, and the arguments of the function are used to specify the number of iterations a block is executed (if in a pipelined or loop structure). Take the `waitfor` statement in Fig 3(A) as an example. In the example, stmnt_A and stmnt_B will be executed once only and no argument is needed in the function symbol declaration for these two statements, and there is a delay of *val* time units between the execution of stmnt_A and stmnt_B inserted by the `waitfor val` statement. Thus, the assertions below are generated for the timing relation above. Our tool will name the symbol with the full hierarchy path to ensure the uniqueness of each function symbol.

```
(declare − fun Tend.t_stmntA () Int)
(declare − fun Tstart.t_stmntB () Int)
(assert (= Tstart.t_stmntB (+ Tend.t_stmntA val)))
```

A. Timing Relation for Hierarchical Behaviors

In SpecC, the child behavior instantiation implies a function call to the child behavior. For a behavior S consisting of a set of child behavior instances $\langle s_1, s_2, s_3, ...s_n \rangle$, the following condition holds:

$$\forall i \in \{1,2,3,...n\}, T_{start}(S) \leq T_{start}(s_i),\ T_{end}(S) \geq T_{end}(s_i)$$

The timing relation between the child behaviors is dependent on the execution type specified in the parent behavior. In this paper, we support *sequential*, *parallel*, *pipelined*, and *loop* behaviors.

1) *Sequential Execution* of statements is defined by ordered time intervals that do not overlap. Formally, for a statement S consisting of a sequence of sub-statements $\langle s_1, s_2, ...s_n \rangle$, the following conditions hold:

$$\begin{aligned} \forall i \in \{1,2,...,n\}, & T_{start}(S) \leq T_{start}(s_i), \\ & T_{end}(s_i) \leq T_{end}(S) \\ & T_{start}(s_i) < T_{end}(s_i) \\ \forall i \in \{1,2,...,n-1\}, & T_{end}(s_i) \leq T_{start}(s_{i+1}) \end{aligned}$$

2) *Parallel Execution* can be specified by *par* or *pipe* statements. Formally, for a `par` statement S consisting of concurrent child statements $\langle s_1, s_2, ...s_n \rangle$, the following conditions hold:

$$\begin{aligned} \forall i \in \{1,2,...,n\}, & T_{start}(S) \leq T_{start}(s_i), \\ & T_{end}(S) \geq T_{end}(s_i) \\ & T_{start}(s_i) < T_{end}(s_i) \end{aligned}$$

3) *Pipelined Execution* of statements is a special form of concurrent execution. Formally, for a `pipe` statement S executed for m iterations, let $s_{i \cdot j}$ represents the j-th iteration of the execution of statement s_i. Then, the following conditions hold:

$$\begin{aligned} & \forall i, x \in \{1,2,...,n\},\ j, y \in \{1,2,...,m\} : \\ & T_{start}(s_{i \cdot j}) < T_{end}(s_{i \cdot j}), \\ & T_{start}(s_{i \cdot j}) = T_{start}(s_{x \cdot y}), \quad if \quad i+j = x+y \\ & T_{end}(s_{i \cdot j}) = T_{end}(s_{x \cdot y}), \quad if \quad i+j = x+y \\ & T_{end}(s_{i \cdot j}) \leq T_{start}(s_{x \cdot y}), \quad if \quad i+j < x+y \end{aligned}$$

A limitation of our approach is that the number of iterations m has to be a *known* integer. If it is statically unknown (i.e. a variable), our tool will prompt the designer to input an upper bound for the loop.

4) *Loop Execution* can be regarded as a special case of *pipelined* execution with only one stage. As above, we assume that the number of iterations is a finite constant.

B. Timing Relation Extraction for Leaf Behaviors

We pay significant attention in this paper to analyze the timing information specified in leaf behaviors and channels, which is critical in order to capture communication timing in TLMs. Fig. 4(A) highlights the statements which are analyzed in the source code as well as the rules to extract the corresponding timing relations for the static analysis. The rule for the `waitfor` statement has been introduced already. We now describe the others.

1) *Conditional Execution*: When conditional execution, such as a `if` statement or `if-else` statement, is used in the model, we create a time interval $\langle T_{if\_start}, T_{if\_end} \rangle$ and a logic stamp C_{if} which represents the logic condition (for if-else, we also create a tuple $\langle T_{else\_start}, T_{else\_end} \rangle$). Fig. 5 illustrates the timing relations for the conditional execution. Here, T_{prev} and

T_{next} represent the time stamps before and after the conditional execution. As shown with the selection structure in Fig. 5, the value of T_{next} is dependent on the binary value of C_{if}. Note that T_{never} is a time stamp with a very large value representing infinity. We represent the situation that a statement will never be executed by giving the corresponding time stamp this large value (there is no way to represent infinity in the SMT-LIB language). Any time stamp greater or equal to T_{never} means that the corresponding situation will never happen. Note that our tool will not analyze the specified condition in an if-statement, but only create the conditional assertions as listed in the illustration. It is the SMT solver's job to find an assignment for the condition and time stamps that satisfy the assertions.

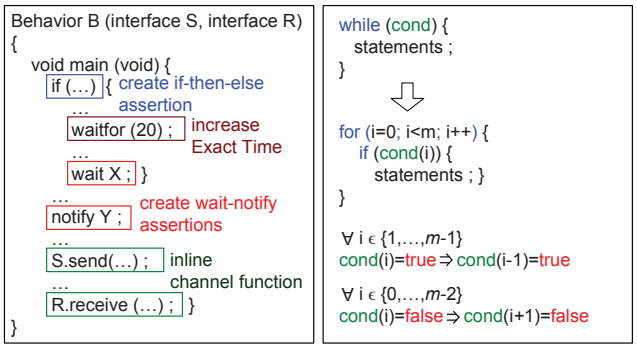


Fig. 4. Timing relation extraction for a leaf behavior

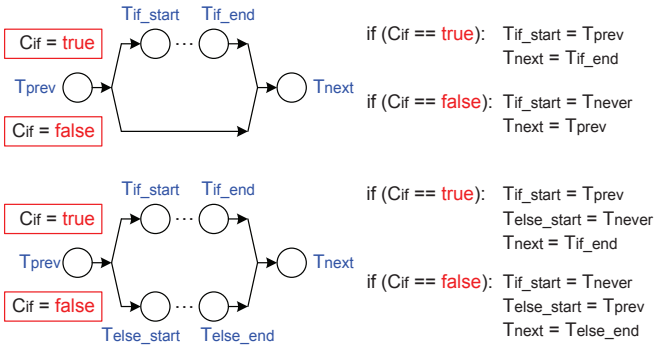


Fig. 5. Timing relation extraction for conditional execution

2) *Loop Unrolling*: To limit the verification space and the execution time of the solver, for each loop with undefined iteration count (i.e., the condition is variable), our tool will prompt the designer to provide an the upper bound for the loop, and then unroll the loop to multiple `if` statements. Fig. 4(B) illustrates the loop unrolling performed by our tool. It also shows that the tool creates implication assertions for the conditions generated by loop unrolling.

3) *wait-notify synchronization*: In order to analyze a TLM with synchronization among multiple concurrent behaviors, we support events and the corresponding `wait-notify` synchronization. When a `wait` statement is executed, it suspends the current thread from execution until the event is triggered by a `notify`. A time interval $\langle T_{start}, T_{end} \rangle$ is generated as for other statements. For a `wait` statement W triggered by a `notify` statement N, the following conditions hold:

$$T_{start}(W) \leq T_{start}(N),$$
$$T_{end.t}(W) = T_{end.t}(N),$$
$$T_{end.d}(W) = T_{end.d}(N) + 1$$

Note that T_{start} equals T_{end} for a `notify` statement. Also, to analyze the satisfiability of the specified timing constraints, we have to determine the mapping between `wait` and `notify` statements, i.e. which `notify` wakes up which `wait`. Our proposed method to generate the assertions for the wait-notify mapping is illustrated in Fig 6.

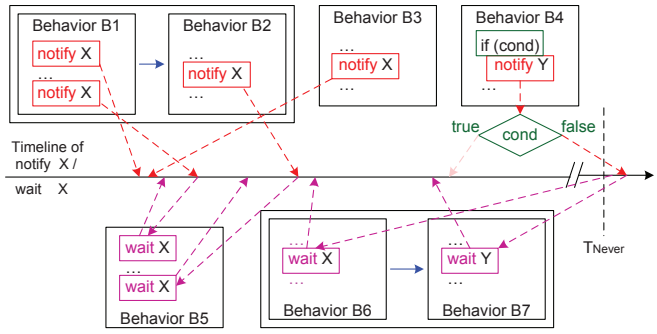


Fig. 6. Timing relation extraction for the **wait-notify** statement

In this example, all behaviors are executed in parallel except for the two behavior pairs ⟨B1, B2⟩ and ⟨B6, B7⟩ executed sequentially. Our method consists of two steps:
(1) for every event in the model, our approach generates the assertions to sort the time stamps of the `notify` statements which trigger the event. This step is illustrated in the upper part of Fig 6. Note that if the `notify` statement is inside a conditional statement, the value of its time stamp is dependent on the condition. For example, $T_{start.t}$ for the `notify` statement in behavior B4 in Fig. 6 will be greater than T_{never} if the logic condition is false.
(2) for every `wait` in the model, we generate the assertions to "search" the sorted time stamps of the `notify` statements and find one that is greater than and the closest to T_{start} of the `wait`, and set the time and delta cycle of the `wait` using the condition we listed above. This step is illustrated in the lower part of Fig. 6.

4) *Channel Interface Function Call*: In a TLM, the timing information of the target bus protocol and the synchronization mechanism between communicating parties are specified in the interface methods defined in the channel. The communication between the behaviors takes place by calling those interface functions. To generate assertions for the SMT solver, our approach traverses down to the interface method in the channel when it is called. Consequently, the timing information specified in the channel model is taken into consideration during the timing analysis of the behavior.

C. Liveness and Deadlock

For a multi-PEs system model, improper execution order or communication may lead to problems, including deadlock. In our method, a deadlock caused by circular waiting in the model

will be reported to the designer in the form of unsatisfiable assertions since there are conflicts in the timing relations. Another potential deadlock would be a `wait` statement missing the wake-up signal. Fig. 6 also shows examples for two cases. Behavior B6 shows one case in which `wait X` misses all notification for `X` therefore it will never be waken up. Behavior B7 illustrates another case. `wait Y` can not wake up if the condition for `notify Y` in behavior B4 is not true. Both situations are covered by our tool and reported to the designer.

D. Hierarchical Timing Analysis

The number of assertions generated by our method increases with the complexity of the model. To keep the number of assertions manageable and limit the run time of the SMT solver, our method addresses the complexity growth by analyzing the timing constraints in a hierarchical manner. Timing constraints verified at a lower hierarchy level are regarded as the prerequisite conditions for the verification of the higher level. Verified timing constraints can be specified by use of the do-timing construct in the model. When our method finds a do-timing construct during the design traversal, it will take the constraints as they listed and not traverse further down the hierarchy. Thus, the assertions needed for model verification at the higher hierarchical level are greatly reduced.

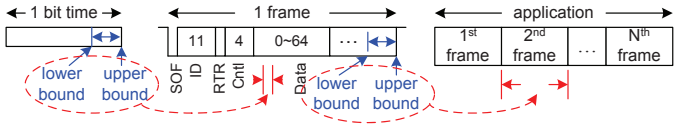


Fig. 7. Hierarchical timing analysis of CAN bus protocol

Take the CAN bus protocol as an example. The bit time generated by the bit time logic for each engine control unit (ECU) can vary due to different local operating frequencies. Thus, the time needed for transmission can differ from one frame to another. To verify the timing constraint of the frame transmission, we use the pre-verified lower and upper bound of the bit time as prerequisite conditions. Fig. 7 illustrates the hierarchical timing analysis of CAN protocol from the bit time via frame time up to the application.

IV. Experiments

We use two standard bus protocols widely used in industry to demonstrate our approach. As shown in Table *II*, both models are of reasonable size with practical analysis times. The first example is a three-ECU communication over a CAN bus protocol[10]. In this automotive example, the RPMcompute ECU issues a request to an RPMsensor using a remote frame. Upon the reception of the request, the sensor initiates an operation to read revolutions per minute (RPM) from the engine and sends it back to RPMcompute using a data frame. After receiving the raw RPM from the sensor, the RPMcompute ECU calculates the average RPM and sends that to Dashboard ECU for displaying. The procedure is illustrated in Fig. 2(A), and the detailed bus TLM is shown in Fig 8. Note that the bit time units required for each communication step in CAN bus protocol are specified with do-timing construct as prerequisite. In this example, the timing is analyzed on assumptions that reading RPM value from engine takes 40 bit time units and computing average RPM takes 10 to 20 bit time units.

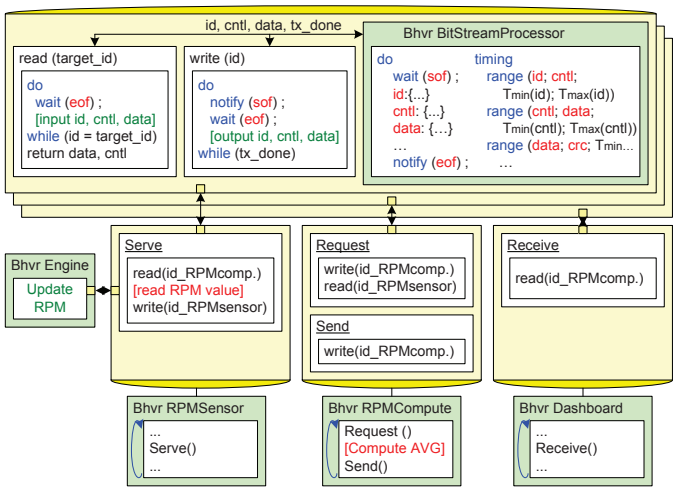


Fig. 8. TLM of automotive example using CAN bus

Our second example is a producer-consumer model communicating over an AMBA AHB protocol. Here, the producer and consumer call interface functions `send` and `receive`, respectively, to transfer data through an AMBA AHB channel specified at TLM abstraction [10]. Fig. 9 illustrates the TLM with the detailed bus model. Note that a parallel behavior PollFlag is created to respond for the slave (Consumer) to all polling requests from the master (Producer). The procedure contains three steps as the timing diagram shown in Fig. 2(B): master reads the flag in PollFlag, master resets the flag, and then sends the data to slave. In this model, the delays compliant to the AHB reference are specified by `waitfor` statements.

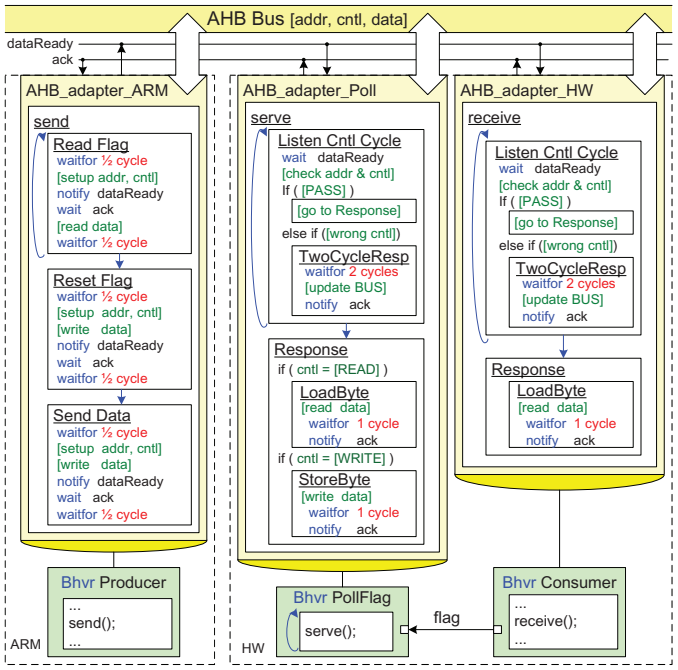


Fig. 9. TLM of Producer-Consumer example using AMBA AHB

The statistics of TLM timing analysis for both bus protocols

TABLE II
STATIC SMT ANALYSIS OF TLM EXAMPLES USING AMBA AHB AND CAN BUS PROTOCOLS

exp	$Constraint$	$Condition$	$\#ofassertions$	LOC	$Time$	$Result$
Liveness and timing analysis for CAN TLM						
1	None	No Circular Waiting	382 (3 augmented)	24963	$>$ 2hr	UNKNOWN
2	T_{end}(DashDisp)$<T_{never}$	No Circular Waiting	383 (4 augmented)	24972	189s	SAT
3	T_{end}(RPMcmp)$\leq$ 200 units	Minimum execution time	384 (5 augmented)	24975	50s	UNSAT
4	T_{end}(RPMcmp)$\leq$ 500 units	Write always fails in the 1st attempt	387 (8 augmented)	24984	284s	SAT
5	T_{end}(RPMcmp)$\geq$ 300 units	Write succeeds in the 1st attempt	387 (8 augmented)	24984	5s	UNSAT
6	T_{end}(RPMcmp)$\leq$ 500 units	Bus utilization $\leq$ 60%	385 (6 augmented)	25054	135s	SAT
Liveness and timing analysis for AHB TLM						
7	None	No Circular Waiting	240 (6 augmented)	19389	332s	SAT
8	T_{end}(Prod)$<T_{never}$	No Circular Waiting	241 (7 augmented)	19392	313s	SAT
9	T_{end}(Prod) $<$ 6 cycles	Minimum execution time	242 (8 augmented)	19395	4s	UNSAT
10	T_{end}(Prod) $\leq$ 10 cycles	Polling succeeds in the first 2 attemps	243 (9 augmented)	19401	333s	SAT
11	T_{end}(Prod) $\geq$ 10 cycles	Polling succeeds in the first 2 attemps	243 (9 augmented)	19401	114s	UNSAT

are listed in Table II. In the experiments, we verify the satisfiability of liveness and timing constraints. Exp.1 and Exp.7 check if there is any conflict caused by circular waiting in the model, and the others verify the liveness and timing constraints in various scenarios specified with user augmented assertions. Exp.6 in the table shows a scenario where we allow the model to utilize the bus up to 60% maximum, that is, on average over 5 slots only 3 may be used. The number of assertions, lines of code (LOC) for those assertions and the run time of the solver are also listed in the table. According to the measured time, the satisfiability searching for these two models and the constraints we added is reasonably fast, and as is often expected, unsatisfiable solutions are faster obtained than satisfiable ones. Note that for Exp.1 the solver gave no answer after two hours of calculation. To reduce the search time, we added an assumption that the entire transaction finishes in finite time (test case 2). All experiments have been performed on a host PC with a 4-core CPU (Intel$^{(R)}$ Core$^{(TM)}$2 Quad) at 3.0 GHz with Microsoft Z3 solver (version 4.1). Both experiments contain initial designer augmented assertions (3 for the CAN TLM and 6 for the AHB TLM) which are inserted to specify the real use case. Taking the CAN bus as an example, the three user augmented assertions reflect the timing relations shown in Fig. 2(A): the end time of remote frame transmission in RPMcompute equals the end time of the remote frame reception in RPMsensor; the end of frame transmission in RPMsensor equals the end of the frame reception in RPMcompute; and the end of the frame transmission in RPMcompute equals the end of the frame reception in Dashboard. The initial 6 user assertions in the AHB example reflect a similar situation.

V. CONCLUSION

In this paper, we have proposed an approach to verify liveness and timing constraints by extracting timing relations from a TLM design model and using a SMT solver to verify the satisfiability of the corresponding assertions. We verify the timing information specified in computation as well as in communication. Also, we introduce a hierarchical method to cope with the complexity growth of the model. We demonstrated our approach with two standard bus protocols AMBA AHB and CAN bus. Our approach utilizes the designer's augmented assertion reflecting the properties of interest. In future work, we plan to improve the interaction between the designer and the SMT assertion generator.

REFERENCES

[1] C. Chou, Y. Ho, C. Hsieh, C. Huang. *"Symbolic model checking on systemc designs,"* in DAC '12, pages 327–333. ACM.

[2] C. Chou, C. Hsu, Y. Chao, and C. Huang, *"Formal deadlock checking on high-level systemc designs,"* in ICCAD '10, pages 794–799. IEEE Press.

[3] L. De Moura and N. Bjørner, *"Z3: An efficient smt solver,"* in TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag.

[4] L. De Moura and N. Bjørner, *"Satisfiability modulo theories: Introduction and applications,"* in Commun. ACM, 54(9):69–77, Sept. 2011.

[5] M. Fujita and H. Nakamura, *"The standard specc language,"* in *ISSS*, pages 81–86. ACM Press, 2001.

[6] A. Habibi, H. Moinudeen, and S. Tahar, *"Generating finite state machines from systemc,"* in DATE '06, pages 76–81. European Design and Automation Association.

[7] A. Habibi S. Tahar, *"An approach for the verification of systemc designs using asml,"* in *ATVA*, pages 69–83. Springer, '05.

[8] P. Herber, J. Fellmuth, and S. Glesner, *"Model checking systemc designs using timed automata,"* in CODES+ISSS '08, pages 131–136. ACM.

[9] D. Karlsson, P. Eles, and Z. Peng, *"Formal verification of systemc designs using a petri-net based representation,"* in DATE '06, pages 1228–1233.

[10] G. Schirner and R. Dömer, *"Quantitative analysis of the speed/accuracy trade-off in transaction level modeling,"* in *ACM Trans. Embed. Comput. Syst.*, 8:4:1–4:29, Jan. 2009.

[11] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *"SpecC: Specification Language and Methodology,"* in *Kluwer Academic Publishers*, Boston, March 2000.