Modeling Hardware Systems with Complex Clock Synchronizations in the SGM Formal Verifier [*]

Wen-Shiu Liao and Pao-Ann Hsiung[†]
Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi
[†]hpa@computer.org

Abstract

Traditional verification techniques such as simulation and emulation can no longer exhaustively verify large scale hardware designs, hence many researchers and designers are trying to apply formal verification to hardware systems. However, many formal verification tools either have difficulties to model complex clock behaviors such as multiple bus clocks or gated clock, or do not support clock based synchronization behavior in their model. In this paper, we propose a novel mechanism to precisely model hardware systems with full clock based synchronization support using extended timed automata. The proposed mechanism has been integrated into the formal verifier SGM.

1 Introduction

With the ever-increasing capacity of integrating gates into chips, verification has become a serious problem in the VLSI design. The traditional verification methods suffer from huge numbers of test vectors and are becoming inefficient. Formal verification, in contrast to traditional verification methods, provides more confidences by exhaustively traversing complete system state spaces. When a system fails to satisfy a property, formal verification produces a counterexample that is very attractive and helped to system designers. Several efforts on formally verifying VLSI indicate that verifying a large system by using formal methods is feasible although formal verification has the state space explosion problem [1].

A study of recent research on formal verification of hardware systems and on contemporary formal verification tools shows that there still exists some restrictions on modeling complex hardware systems in these formal verification tools. A major problem is the restriction on modeling complex clock behaviors of hardware systems. For example, SMV has its limits in modeling hardware systems with multiple clocks or a gated clock because in SMV there is only one implicit global clock. This forces a designer to perform some tricks in modeling such behaviors as in [2]. VIS does not support multiple clocks in their model either [3]. UPPAAL [4] provides a basic synchronization mechanism to model message passing between two components by declaring a channel. HyTech [5] also provides synchronization to ensure two transitions in different automata can trigger at the same time when an event is received. These two synchronization mechanisms neither model hardware systems with multiple bus clocks nor a gated clock. The model checker SPIN [6] is designed to verify asynchronous behaviors in software systems rather than synchronous behaviors in hardware systems. From above, it is thus concluded that there is a general lack of mechanism and methodology to model clock based synchronization in formal verification tools.

The article organization is as follows. Section 2 describes our system model and how we model hardware systems with complex clock synchronization. Section 3 will discuss in detail how we compose synchronization transition in SGM. Section 4 gives the final conclusions with future work.

2 System and Clock Models

To handle above issues, we choose State Graph Manipulators (SGM) [7] to implement our modeling methodology. SGM is a high-level compositional model checker with multiple state-space reduction techniques for the verification of real time systems. In SGM, a system is described by a set of communication extended timed automata (ETA) [8] and a property is specified by Timed Computation Tree Logic (TCTL) [7]. In SGM, the global system state-space is computed iteratively by composing one timed automaton at a time.

2.1 System Model

Our system model is a set of *Extended Timed Automata* (ETA), which have shared variables and synchronization labels and are formally defined as follows.

Definition 1 **Mode Predicate**
Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η over C and D is defined as: $\eta := \mathit{false} \mid x \sim c \mid x - y \sim c \mid d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$, $c \in \mathcal{N}$, the set of integers, $d \in D$, and η_1, η_2 are mode predicates. □

[*]This work was supported by project grant NSC 91-2215-E-194-008 from the National Science Council, Taiwan, ROC.

Let $B(C,D)$ be the set of all mode predicates over C and D.

Definition 2 **Extended Timed Automaton**
An *Extended Timed Automaton* (ETA) is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, L_i, \chi_i, E_i, \lambda_i, \tau_i, \rho_i)$ such that: M_i is a finite set of modes, $m_i^0 \in M$ is the initial mode, C_i is a set of clock variables, D_i is a set of discrete variables, L_i is a set of synchronization labels, $\chi_i : M_i \mapsto B(C_i, D_i)$ is an *invariance* function that labels each mode with a condition true in that mode, $E_i \subseteq M_i \times M_i$ is a set of transitions, $\lambda_i : E_i \mapsto L_i$ associates a synchronization label with a transition, $\tau_i : E_i \mapsto B(C_i, D_i)$ defines the transition triggering conditions, and $\rho_i : E_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such as resetting some clock variables and setting some discrete variables to specific integer values. □

Using the above ETA definition, our system model can be defined as follows.

Definition 3 **Hardware System**
A *Hardware System* is defined as a set of hardware and clock components. Each component is modeled by one or more timed automata. A system is modeled by a network of communicating timed automata. If a system $\mathcal{S}$ has a set of hardware components $\{H_1, H_2, \ldots, H_n\}$ and a set of clock components $\{C_1, C_2, \ldots, C_m\}$, then $\mathcal{S} = H_1 \| H_2 \| \ldots \| H_n \| C_1 \| C_2 \| \ldots \| C_m$, where $\|$ is a parallel composition operator resulting in the concurrent behavior of its two operands. If H_i is modeled by an ETA A_{H_i}, $1 \le i \le n$, and C_j is modeled by an ETA A_{C_j}, $1 \le j \le m$, then the ETA defined by $A_S = A_{H_1} \times \ldots \times A_{H_n} \times A_{C_1} \times \ldots \times A_{C_m}$ is an ETA model for system $\mathcal{S}$, where $\times$ is the state-graph merge operation in SGM and concurrency semantics is defined as follows:

- two concurrent transitions with the same synchronization label are represented by a single synchronized transition in the product automaton, and
- two concurrent transitions without any synchronization label are represented by interleaving them, resulting in possibly two different paths (computations). □

2.2 Clock Models

We show how we handle system models with multiple bus clocks and a gated clock. Figure 1 illustrates an example of modeling two bus clocks. We use clock variables X and Y to represent the progress of time in the two bus clocks, and register variables *CLK_A* and *CLK_B* to represent the signal outputs of the two bus clocks. In the timed automata A and B, *A0* (*B0*) and *A1* (*B1*) represent the states of the bus clocks driven low and high, respectively. The transitions from *A0* (*B0*) to *A1* (*B1*) represent the change of bus clock level from low to high, which is the positive edge of the bus clock signal. Similarly, the transitions from *A1* (*B1*) to *A0*

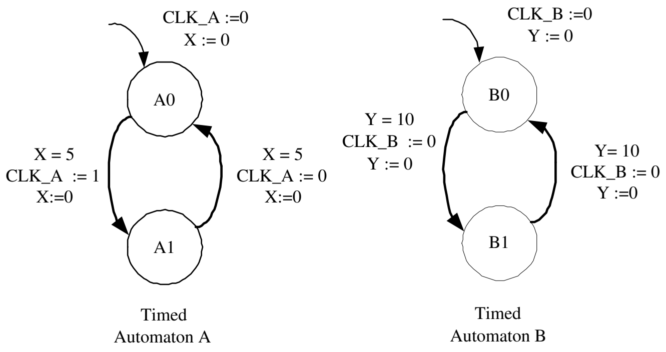

Figure 1: Two bus clocks model

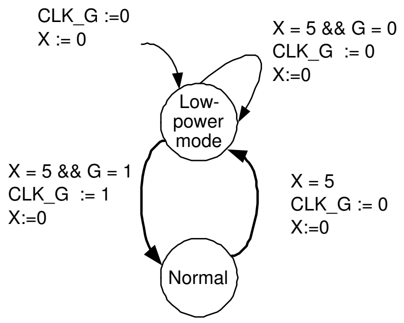

Figure 2: Gated clock model

(*B0*) represent the negative edge of the bus clock. The different rates of multiple bus clocks in hardware system can be modeled by setting different values to the clock variables X and Y on the transitions. For example, the rate of clock *CLK_B* is twice that of clock *CLK_A* in Figure 1. The initialization of bus clocks can be modeled by setting the initial state of the automata. If transitions in some hardware component models need to be triggered at the positive edge of *CLK_A*, then the user can synchronize these transitions with the clock transition from *A0* to *A1* in order to guarantee that these transitions will be triggered at the same time.

Figure 2 is a gated clock model. Gated clock is a common technique for low power design in hardware. Register variable G is a control signal that enables/disables the clock. When G is one, the clock operates normally. When G becomes zero, the timed automata will eventually stay in Low-power mode until G becomes one, which represents the gated condition of a clock.

Figure 3 illustrates transfer phase and termination phase in non-address pipelining of an IBM CoreConnect PLB [1] bus arbiter. We only illustrate time-out and *Sl_addrAck* of termination phase in Figure 3. A *PLB_PAValid* signal will be asserted high when the arbiter grants the bus to a master at the beginning of the transfer phase. If the *Sl_wait* is asserted by a PLB slave, the bus arbiter will continue to drive *PLB_PAValid* as well as the address and transfer qualifier signals until the slave asserts the *Sl_addrAck* signal. A register variable C counts the number of bus clock cycles

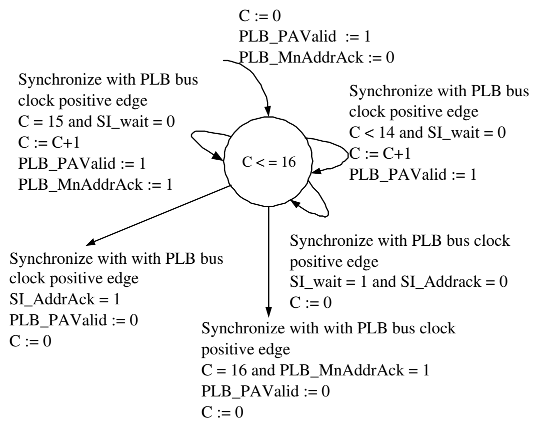
C := 0
PLB_PAValid := 1
PLB_MnAddrAck := 0
Synchronize with PLB bus clock positive edge
C = 15 and SI_wait = 0
C := C+1
PLB_PAValid := 1
PLB_MnAddrAck := 1
Synchronize with PLB bus clock positive edge
C < 14 and SI_wait = 0
C := C+1
PLB_PAValid := 1
C < = 16
Synchronize with with PLB bus clock positive edge
SI_AddrAck = 1
PLB_PAValid := 0
C := 0
Synchronize with PLB bus clock positive edge
SI_wait = 1 and SI_Addrack = 0
C := 0
Synchronize with with PLB bus clock positive edge
C = 16 and PLB_MnAddrAck = 1
PLB_PAValid := 0
C := 0

Figure 3: Transfer phase and termination phase in non-address pipelining of an IBM CoreConnect PLB bus arbiter

in Figure 3. *C* will start counting as soon as the bus arbiter grants the master's request by asserting *PLB_PAValid* to high. If the slave does not assert *SI_wait* and has not responded within 15 bus clock cycles, then the arbiter will assert PLB_mnAddrAck high to indicate time-out and the transfer will be terminated at the 16th bus clock cycle. If a slave acknowledges the bus arbiter by asserting *SI_addrAck*, the address cycle is terminated and *C* will be reset.

3 Modeling Synchronizations

To ensure each hardware component changes its state synchronously at the same clock triggering edge, the ETA models of these components must have the same synchronization label on the synchronizing transitions. By doing so, a hardware ETA transition will be taken only if there is a triggering transition on a clock component model. Thus, we can create multiple clock component models to represent multiple hardware clocks or create an enhanced clock model to represent a gated clock or a skewed clock.

We define four data attributes for synchronization transitions as follows.

1. Synchronization Label: A label which is associated with two or more transitions that are to be synchronized is call a synchronization label and the transitions are called synchronization transitions.

2. Type of Synchronization:

   (a) Base synchronization (abbreviated as sync_base) transition: A synchronization transition of clock component models is called a *base* synchronization transition. Our modeling methodology does not allow synchronization between two base synchronization transitions.

   (b) Reference synchronization (abbreviated as sync_ref) transition: A synchronization transition of hardware components is called a *reference* synchronization transition.

   (c) Synchronized synchronization (abbreviated as sync'ed) transition: A transition that results from synchronizing one or more synchronization transitions of hardware components with synchronization transition of a clock component model is called a synchronized synchronization transition.

3. Transition Synchronization Base: If the synchronous type of a synchronization transition is sync'ed, then the sync_base transition must be recorded for future compositions.

4. Transition Synchronization References: A list of sync_ref transitions that have been composed together into one transition. This is used in sync_ref and sync'ed transitions.

Table 1 shows the synchronization algorithm for composing two modes of two different automata. Details are left out due to page limits. The basic idea is to compose out-going synchronization transitions having the same synchronization labels. The main issues in synchronizing two transitions are: (1) which transition should be preserved, and (2) whether we should create two branching synchronization transitions when the composition fails. We need to handle only four composition cases as described in the following. Let tr1 represent a transition of a mode from a composed automaton and tr2 represent a transition of a mode from an automaton that is being composed.

1. tr1 is a sync_base transition and tr2 is a sync_ref transition: tr1 can be still be triggered even if the composition of the two transitions fails. (Steps 6, 7)

2. tr1 is a sync_ref transition and tr2 being a sync_base transition: Due to tr2 being a sync_base transition, we must compose all out-going sync_ref transitions having the same synchronization label with tr2. If the transition composition fails, tr2 will still be triggered. (Steps 8, 9)

3. tr1 is a sync_ref transition and tr2 is a sync_ref transition: If the composition of these two transitions succeeds, the sync_ref attribute in the newly composed transition is the union of the sync_ref attributes of tr1 and of tr2. If the composition fails, this means there are some contradictions between the triggering conditions of tr1 and tr2. Thus, a new branching transition is needed. So, we must first keep tr1. Then, we compose all non-conflicting transitions in the sync_ref attribute of tr1 with tr2. (Steps 10, 11)

4. tr1 is a sync'ed transition and tr2 is a sync_ref transition: We compose tr2 with the base transition of tr1 at the beginning. If the composition succeeds, we then

Table 1: Composition of synchronization transitions

```
Compose_Synchronous_Transitions(mode1, mode2)
mode1 = a mode of an already composed automaton;
mode2 = a mode of the automaton to be composed;      {
    for each out transition tr1 of mode1{                         (1)
        if tr1 is a synchronization transition {                  (2)
            ϕ = synchronization label of tr1                      (3)
            {Sync_list} = Get_Same_Sync_Lab_Trans(mode2, ϕ);      (4)
            for each transition tr2 ∈ {Sync_list} {               (5)
                if tr1 is sync_base and tr2 is sync_ref {         (6)
                    Compose_Sync_Base_Ref_Trans(tr1, tr2);        (7)
                }
                else if tr1 is sync_ref and tr2 is sync_base {    (8)
                    Compose_Sync_Ref_Base_Trans(tr1, tr2);        (9)
                }
                else if tr1 is sync_ref and tr2 is sync_ref {     (10)
                    Compose_Sync_Ref_Ref_trans(tr1, tr2);         (11)
                }
                else if tr1 is sync'ed and tr2 is sync_ref {      (12)
                    Compose_Sync'ed_Ref_Trans(tr1, tr2);          (13)
                }
            }
        }
    }
}
```

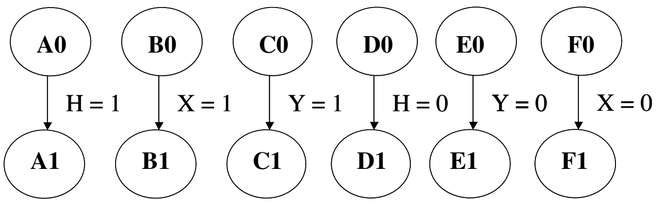

Figure 4: Reference synchronization transitions

compose the newly composed transition with tr1. If the new composition fails, this means there are contradictions between triggering conditions of tr1 and of tr2. Thus, a new branching transition is needed. So, we must first keep tr1. Then, we compose all non-conflicting transitions in the sync_ref attribute of tr1 with tr2. (Steps 12, 13)

Figure 4 is an example of composing synchronization transitions. Each transition in Figure 4 is a sync_ref transition and they all share the *same* synchronization label. There are six ETA to be composed. The triggering condition of the transition $A0 \rightarrow A1$ conflicts with that of the transition $D0 \rightarrow D1$. The triggering condition of the transition $B0 \rightarrow B1$ also conflicts with that of the transition $F0 \rightarrow F1$. After applying our proposed synchronization modeling techniques, the resulting eight transitions after composition are depicted in Figure 5. Each of the resulting transition is composed by three sync_ref transitions. For example, the transition $Z0 \rightarrow Z1$ is composed by transitions $A0 \rightarrow A1$, $B0 \rightarrow B1$, and $C0 \rightarrow C1$. If the transition $Z0 \rightarrow Z1$ is taken, it means the ETA models will change from modes $A0$, $B0$, and $C0$ to modes $A1$, $B1$, and $C1$, simultaneously.

We have applied the proposed clock and synchronization models to IBM CoreConnect and ARM AMBA bus architectures in our FVP formal verification platform [9].

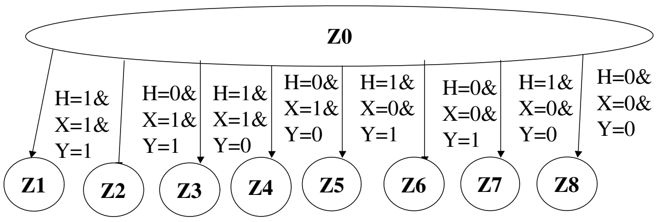

Figure 5: Composed state-graph

4 Conclusion

By proposing multiple clocks model, a gated clock model, and clock-based synchronization models, we have shown how formal verification tools for hardware systems can be extended to analyze multi-rate systems such as System-on-Chip (SoC). By applying our modeling techniques to common on-chip bus architectures such as IBM CoreConnect and ARM AMBA we have shown the feasibility and benefits of our approach. By implementing all the techniques and integrating them into the SGM model checker we have allowed verification engineers to actually verify SoCs formally.

References

[1] A. Goel and W. R. Lee. Formal verification of an IBM CoreConnect processor local bus arbiter core. In *Procs. of the 37th Design Automation Conference*, pages 196–200, 2000.

[2] H. Choi, B. Yun, Y. Lee, and H. Roh. Model checking of S3C2400X industrial embedded SoC product. In *Procs. of the 38th Design Automation Conference*, pages 611–616, June 2001.

[3] UT Austin. Examples of hardware verification using VIS. 1997. http://vlsi.colorado.edu/ vis/texas-97/texas97benchmarks.ps.

[4] J. Bengtsson, F. Larsen, K.and Larsson, P. Petterson, Y. Wang, and C. Weise. New generation of UPPAAL. In *Procs. of the Intl Workshop on Software Tools for Technology Transfer (STTT'98)*, July 1998.

[5] Thomas Henzinger. The theory of hybrid automata. In *Procs. of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[6] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[7] F. Wang and P.-A. Hsiung. Efficient and user-friendly verification. *IEEE Transactions on Computers*, 51(1):61–83, January 2002.

[8] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[9] W.-S. Liao and P.-A. Hsiung. FVP: A formal verification platform for SoC. In *Procs. of the 16th IEEE International SoC Conference*, September 2003.