RTL Verification of Timed Asynchronous and Heterogeneous Systems using Symbolic Model Checking

Vida Vakilotojar and Peter A. Beerel

University of Southern California
Los Angeles, CA 90089, USA
Tel: 213-740-4481
Fax: 213-740-9803
vivakil,pabeerel@eiger.usc.edu

Abstract-- This paper describes a tool-supported methodology for the register-transfer-level formal verification of a growing hardware design paradigm--timed asynchronous systems. These systems are a network of communicating asynchronous and synchronous components and have correctness constraints that depend on specified bounded delays. This paper formalizes the verification problem and demonstrates how time-discretization, abstraction, and non-determinism can be used to obtain a system model comprised of communicating finite state machines that are composed synchronously. The paper then describes a translator that accepts structural VHDL system description along with controller specifications and generates the input to a symbolic model checker (SMV). Finally, we describe two case studies in which concurrent verification and design led to the correction of many errors not easily found using simulation.

Keywords: Register Transfer Level Verification, Symbolic Model Checking, Asynchronous Circuits, Timing Assumptions, Time Discretization.

I. Introduction

Asynchronous design techniques have long promised systems which have low-power, low average-case delay, and efficient communication between different clock domains. Traditional asynchronous design methodologies also claim seamless composition of components and an absence of timing assumptions, i.e., traditional designs are *speed-independent*. Not surprisingly, the circuitry required to implement the request/acknowledge communication protocol in speed-independent circuits often creates overhead, in terms of power, delay, and area, that limits their practicality. In order to reduce the communication overhead and speed up the circuits, many asynchronous researchers have abandoned speed-independent circuits in favor of designs that use timing assumptions at the transistor, gate, and register transfer (RT) level. This new design methodology has led to an x86 asynchronous instruction length decoder that is approximately three times faster than the current state of the art, a differential equation solver design that is almost 50% faster *on average* than any comparable synchronous design [1], and efficient pausible clocking strategies for interfacing different clock domains [2]. These recent successes among others are finally demonstrating the potential advantages of using asynchronous design techniques. However, the increased usage of timing assumptions, both local and global, makes ensuring correctness via simulation more difficult, motivating the need for formal verification.

While numerous tools and techniques have been developed for verifying speed-independent circuits [3, 4, 5, 6, 7, 8], verifying asynchronous circuits with timing assumptions has been rather limited. Burch [9, 10, 11] and Rokicki *et al*. [12] have verified gate-level timed circuits and Yoneda *et al.* [13] have verified time Petri nets, but all their approaches have been limited to small examples due to the state explosion problem associated with their underlying explicit state techniques. To handle larger gate-level circuits, Hamaguchi *et al.* [14] advocate implicit state techniques and we apply some of their modeling techniques to the register-transfer level (RTL). Hamaguchi *et al.* [14] focuses on verifying hazard-freedom. This paper, however, focuses on properties specific to RTL designs.

The paper starts by describing our *timed asynchronous system* design methodology, which encompasses asynchronous, synchronous, and heterogeneous architectures. The paper describes typical components of timed asynchronous systems, including data path components with data-dependent delays (e.g., adders/multipliers), distributed asynchronous control circuits (e.g., XBM burst-mode controllers [15]), and interface glue-logic (i.e., mutual-exclusion elements). The paper then describes the verification problems specific to RTL design that we have addressed. We focus on the *timed hardware protocol verification* particular to timed asynchronous systems and do not treat word-level verification issues common in synchronous RTL designs. The paper discusses the safety and liveness criteria associated with these hardware protocols as well as the delay assumptions sometimes necessary to ensure correctness.

We then describe our RTL verification methodology. First, we abstract each system component to a synchronous finite state machine (FSM) in which time is discretized and modeled as a sequence of states, and non-deterministic transitions are used to facilitate modeling bounded delays. Although we have not yet proved that such an approximation of a system is really conservative, it has helped us detect many design errors. We describe our correctness criteria in computational tree logic (CTL). We verify the synchronous composition of the components using a well-known symbolic model checker, SMV.

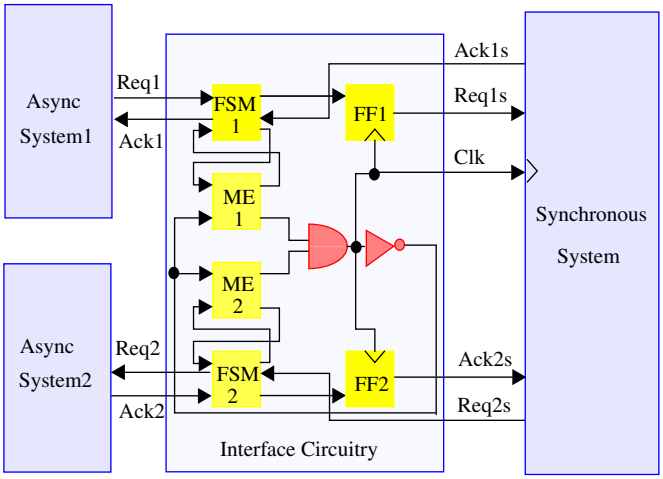

Fig. 1: A pausible clocking interface; an example of a timed asynchronous system.

The paper then presents a prototype tool to support this methodology. The tool accepts VHDL structural description of a timed system, a library of SMV-described components, and specifications of the XBM asynchronous controllers used. It then translates this input into SMV language and automatically generates many of the necessary CTL formula specifications. Next, we describe two real-life case studies, a differential equation solver and a pausible clocking interface, the verification and design of which proceeded concurrently. Using our tool, we quickly discovered and corrected many design errors that would have been difficult to find using simulation. Finally, the application of our tool on a scalable FIFO design with over 10^{13} reachable states is demonstrated.

II. Timed Asynchronous Systems

A timed asynchronous system is a network of timed control and data path components that synchronize data processing and communication using event-driven signalling. Controllers send requests to data-path components, e.g., functional units, latches, or flip-flops, signalling them to process data. Some data path elements, such as a self-timed adder, contain completion sensing circuitry that raises a done signal which is routed back to the controller as a form of acknowledgment. For other data path elements, such as edge-triggered flip-flops, the acknowledgment is inferred by timing assumptions. In addition, the communication circuity may contain glue logic, e.g., mutual exclusion elements and simple gates.

Each timed component has a specification which represents a contract between the component and its environment. It specifies the legal (possibly sequential) behavior of its inputs and the corresponding output behavior. The specification may include timing constraints restricting the legal input behavior, and timing assumptions describing temporal properties of the output behavior. Unlike untimed systems, these timing assumptions are often necessary to ensure that the network of communicating components will adhere to each of the individual component's communication convention.

The implementation of each timed component is itself a network of gates and/or transistors. Functional elements often are designed using static gates, precharged logic, dual-rail logic, domino logic, or a mixture of these styles. Traditional synchronous latches, registers, and routing components are often used with only slight modifications. Timed controllers are made up of specialized memory elements, such as generalized C-elements or static gates [12]. Typically, the representation of the implementation of a timed component is more complex than the representation of its specification.

As an example of a timed system, consider the pausible-clocking-interface (PCI), illustrated in Fig. 1, which allows two different asynchronous domains to efficiently communicate with one synchronous clock domain. The overall design contains one synchronous module, two XBM asynchronous controllers, and glue-logic. The principle verification goal is to ensure that setup and hold times at the synchronous latches are satisfied, thereby avoiding metastability errors. This requires careful analysis of the delays in the system. Notice that the synchronous module can be considered as one system component whose local clock controls the sampling and processing of all of its other inputs.

A. Modeling timed asynchronous systems at the RT level

Wherever possible, we have tried to hide internal details of a given component or its implementation. By modeling the specification of a component instead of the implementation, we simplify the model and its verification within the larger system. The fact that an implementation conforms to a specification can be separately verified.

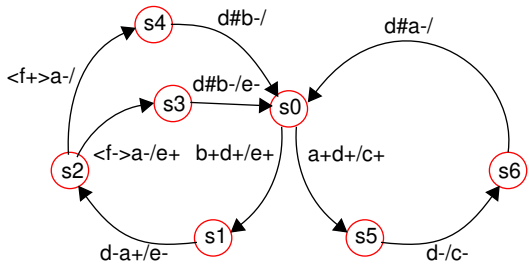

Fig. 2: Example of a burst-mode specification.

The environment surrounding the system is specified as a separate component, which interacts with the rest of the system by generating proper activation signals at proper times and being reset by receiving proper completion signals. For timed systems, the environment may need to have specific timing characteristics for the whole system to function correctly. Such characteristics are part of the constraints on the environment component.

Currently, we have focused on timed systems that use XBM burst-mode machines as the sole type of controllers, however extensions to other controller design styles (e.g., [12]) are straightforward.

B. Extended burst-mode specification formalism

An XBM controller is formally specified by a tuple *g:<In, In0, Out, Out0, S, Trans, s0>*. Here, *In* is the set of inputs, and *In0* the initial values for that set. *Out* and *Out0* are similarly defined. *S* is the set of states of the specification. *Trans* is the set of state transitions of the controller specification. *s0* is the start state of the machine. An XBM specification can automatically be realized into an asynchronous circuit implementation [15, 16].

An XBM specification, an example of which is illustrated in Fig. 2, describes the input/output behavior of a set of signals. Signals not enclosed in brackets and ending with + or - are *terminating* signals (edge signals). Those enclosed in brackets are *conditionals* (level signals). A signal ending with # is a *directed don't care (DDC)*.

Each state transition in an XBM specification is labeled with a set of conditional signal levels and two sets of signal edges; an input burst, and an output burst. An output burst is a set of output edges, and an input burst is a non-empty set of input edges (terminating or directed don't care). Every sequence of state transitions labeled with the same directed don't care signal must eventually end with a terminating edge of that signal. A terminating edge not immediately preceded by a directed don't care edge is a *compulsory* edge, since it must appear during the state transition it labels. At least one edge of each input burst must be compulsory.

In a given state, when all the specified conditional signals have correct values, and when all the specified terminating edges in the input burst have occurred, the machine generates the output burst and moves to the associated destination state. Specified edges of the input burst may appear in arbitrary temporal order. However, the conditional signals must stabilize to correct values before any compulsory edge in the input burst appears and must hold their values until after all of the terminating edges appear. However, they need not be stable out of the specified sampling periods. Outputs may be generated in any order, but the next set of compulsory edges from the next input burst may not appear until the machine has stabilized. This requirement is a variation of multiple-input change fundamental-mode environmental constraint.

There are a number of communication constraints that the environment of the circuit must satisfy. First, each edge signal must be *monotonic*; that is, 1) a compulsory edge must change value only once during a state transition, 2) a directed don't care may change at most once during a sequence of state transitions it labels, and if it does not change during this sequence, it must change during the state transition its terminating edge labels. While a level signal that is not mentioned in a particular state transition may change freely, edge signals that are not mentioned are not allowed to change.

III. RTL VERIFICATION

The systems that we have been considering are heterogeneous systems composed of both synchronous and asynchronous controllers, functional units, memory elements, and glue logic. RTL verification of such a system includes verification of the protocols of communication and some timing verification. Some of the properties to be verified are deadlock-freedom, input safety and monotonicity, hazard-freedom, and setup and hold time satisfaction. In the following, the above properties are illustrated through some examples containing XBM controllers. It is important to note that we focus on the communication aspects of the systems and, in particular, do not address word-level correctness properties.

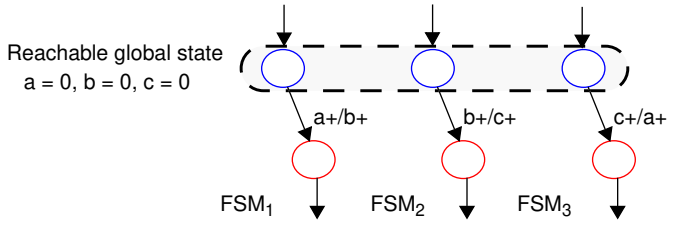

Fig. 3: A deadlock example; in the highlighted state none of the FSMs can ever fire.

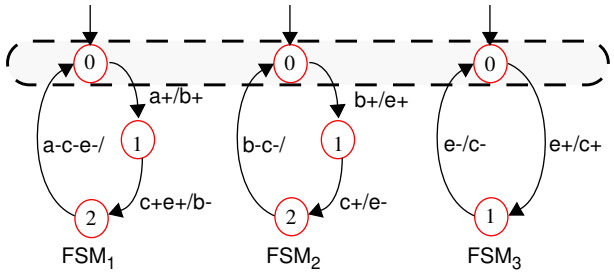

Fig. 4: An example of potential monotonicity violation dependent on timing assumptions.

A. Deadlock-freedom

In a system of communicating FSMs *deadlock* occurs when there is a subset of the FSMs that can not proceed to any next state from their current state. In the system shown in Fig. 3, assuming that the three shown FSMs are in the highlighted global state with signal values $a = b = c = 0$, none of the FSMs can proceed to its next state, because each of them is waiting for a signal that is generated by another one of them. This kind of deadlock may or may not be dependent on timing. For example, such a global state may only be reachable given certain relative delays of components.

B. Input safety and fundamental-mode constraints

An input-safe FSM is one in which no input changes when it is not supposed to change. In a XBM controller, monotonicity of inputs is actually an input safety constraint. In the system shown in Fig. 4 assume that the system starts from the highlighted global state *000,* where global state *ijk* indicates that FSM_1 is in its *i*th state, FSM_2 is in its *j*th state and FSM_3 is in its *k*th state. Depending on the timing properties of the system, we may have either of the two following sequence of global state changes: Seq_1 = *000, 100, 110, 111, 121* or Seq_2 = *000, 100, 110, 111, 211, 221*.

It can be seen that in the first case, *Seq1*, while FSM_1 is still in state *1*, *e* goes high and then low, violating monotonicity of inputs. Thus, under some timing assumptions, we may have monotonicity violation in this system.

As another example of monotonicity violation, consider the system depicted in Fig. 5. Again, depending on the timing of the system, we may have either of the two following sequence of states: Seq_1 = *000, 100, 110, 111* or Seq_2 = *000, 100, 110, 210, 211*. In the case of Seq_1, while FSM_1 is still in state *1*, *a* rises, violating input safety on the absence of the directed don't care *a#*. Thus again, under some timings of the system, we may have input safety violation. However, in this case this problem is removed by adding the directed don't care *a#* to the transition from state *1* to *2* in FSM_1.

Fundamental-mode constraints can be considered a stricter form of input safety constraints. Assume that it takes *x* time units for the circuit to settle after a particular state transition. Then the corresponding fundamental-mode constraint is satisfied iff no compulsory transition occurs until the FSM has been in the new state for at least *x* time units.

C. Hazard-freedom

A *hazard* occurs when an input signal of an enabled gate changes before the output has stabilized into its final value; that is, before the amount of time associated with the gate delay has passed. Hazard-freedom should be verified for all gates in the glue logic of the system.

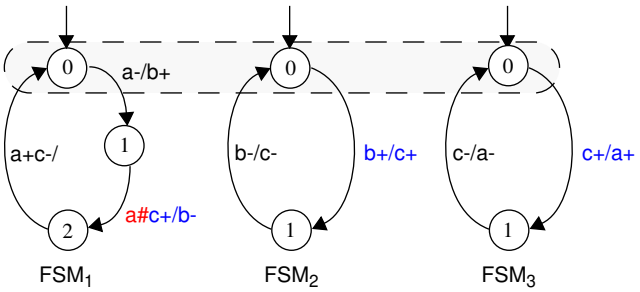

Fig. 5: An example where a DDC is needed if timing assumptions do not hold.

D. Setup and hold times

Currently, we consider setup and hold times for only memory elements like latches or flip-flops. The data input of an element should be stable, before and after the clock/enable is asserted, for the setup time and hold time, respectively. This ensures that the correct value of the data is latched. Setup and hold time violations should be verified for memory elements which are either within an asynchronous component, or while within a synchronous component, have non-synchronized input data.

IV. Discrete-Time Modeling in SMV

This section describes how we use SMV to verify RTL timed asynchronous systems. We first discuss our discrete time modeling approach and the application of SMV. Then, we describe our modeling techniques of different timed system components in detail.

A. Discretized time models and the SMV system

We adopt a discrete time system model for efficiency. Continuous time models, albeit more expressive, require the management of *timer regions*, which severely limits the size of systems that can be analyzed. Discrete time models have the disadvantage that discretization of timer regions leads to large state spaces [9], referred to as the state explosion problem. However, *finite* state spaces (even very large ones) resulting from discretization can often be symbolically represented efficiently using BDDs. In particular, symbolic model checking techniques using BDDs have been successfully applied to many practical designs, some of which having huge state spaces. This motivates the application of symbolic model checking to our discrete time models.

The SMV system is a well-known tool for symbolic model checking of finite state systems [17]. The input language of SMV allows the description of finite state systems that range from completely synchronous to completely asynchronous. Properties to be verified must be specified in computation tree logic (CTL). CTL facilitates the specification of a rich class of temporal properties, including safety, liveness, and fairness.

In SMV, system components are specified as modules connected to and interacting with each other. The behavior of each module, in terms of its response to its input stimuli, and its timing characteristics are specified in the definition of the module. Each module of the system, while interacting with other modules, can be checked for the satisfaction of any of its timing constraints, input safety, deadlock-freedom, etc., using CTL formulas as specifications to be verified for that module.

We are not checking whether our implementation *conforms* [10, 6] to a given specification which would guarantee safe replaceability of the specification by the implementation. Instead, we verify a set of properties that the implementation should have. There is no guarantee that the set of properties we verify is sufficient to ensure that the design will work or will do what it is intended to do. In this way, we are using the verification tool only as a debugging tool that supplements traditional simulation.

In modeling timed systems and trying to facilitate modeling bounded delays, we abstract each system component as a synchronous finite state machine (FSM) for which time is discretized. We encode passage of time using explicit timers for each component of the system which needs to sense time intervals greater than one unit of time. We assume that one unit of time is taken for any global state transition, where timer variables themselves define part of the global state. To make all timers progress at the same rate, we verify the synchronous composition of the components using the simultaneous semantics of SMV.

An important modeling issue is the choice of the value of the time-step. Using a smaller time step leads to a more accurate system model at the expense of a larger state space and longer run times. On the other hand, if the time-step is too large the model may be very inaccurate which can lead to false negative verification results. Fortunately, the timing assumptions typically have large safety margins (otherwise, the system would be too risky to build). Consequently, even relatively large time steps do not lead to many false negatives.

It is also important to note that the discretization of delays, if not done carefully, can introduce false positive verification results. Fortunately, other researchers have developed guidelines that ensure the absence of false positives in particular classes of systems [9, 12]. Based on these results, we are trying to prove whether or not this property is guaranteed by our system modeling approach. In any case, as suggested earlier, our primary goal is to find bugs that are difficult to detect by simulation alone, not to guarantee 100% correctness.

B. Modeling XBM controllers

We will use the XBM controller depicted in Fig. 2 to illustrate its model and verification in SMV. Its corresponding SMV code is partially given in Fig. 6.

The state of the controller is represented by a scalar variable *state* which can take on any of the symbolic values *s0..s6*. The inputs of the controller are defined as input parameters of the controller module. To simplify the model, the controller outputs (e.g., *c* and *e*) are modeled to be dependent only on the controller state; i.e., they are DEFINEd in terms of *state*. Thus, the controllers are

```
MODULE XBM_Controller (a, b, d)
VAR
state : {s0, s1, s2, s3, s4, s5, s6};
f : boolean;
ASSIGN
init (state) := s0;
next (state) := case
   ....
   state = s0 & a & d : s5;
   state = s0 & b & d : s1;
   state = s2 &!a & !f : s3;
   state = s2 &!a & f : s4;
   ...
   1 : state;
esac;

init (f) := 0;
next (f) := case
     state = s1 & next (state) = s2 : {0, 1};
     1 : f;
esac;

DEFINE
c := case
    state = s5 : 1;
   1 : 0;
esac;
....
SPEC     AG EF (state = s1)
SPEC     AG EF (state = s2)
SPEC     state = s1 -> b
SPEC     state = s6 & d -> AX  d
SPEC    (state = s0 -> AX (state = s1 -> !a))
```

Fig. 6: SMV code for the XBM controller illustrated in Fig. 2.

```
VAR wait : 0..3;

ASSIGN

init (wait) := 3;
next (wait) := case
   ...
   state = s0 & a & d & wait = 0 : 3;
   state = s0 & a & d & wait > 0 : wait-1;
   ...
   1 : wait;
esac;

init (state) := s0;
next (state) := case
   ....
   state = s0 & a & d & wait = 0 : s5;
   ...
   1 : state;
esac;
```

Fig. 7: SMV code modifications to model bounded delays.

modeled as Moore Style FSMs. This modeling style assumes that all output changes corresponding to a change of state, occur at the same time, right after the state changes. Although this is not guaranteed to be a conservative assumption, it is a reasonable initial model, since the different controller output delays are typically about the same. When possible the above mentioned simplification minimizes the number of SMV variables needed to model the controller, thus minimizing BDD sizes. The state transition graph of the XBM machine is coded into SMV by an assignment statement which assigns next values to the variable *state* depending on the current value of *state* and whether the input burst of a specific state transition has occurred. For example, when *a* falls in state *s2*,, depending on the value of the conditional signal *f*, the machine will have a transition to *s3* or *s4*.

To ensure that a network of controllers is deadlock-free, one can verify that none of the controllers may get stuck in any of its states. If there is deadlock, then at least two of the controllers must be involved. In any deadlocked controller, there is a state from which no state transition can fire. To verify the absence of such a state, one can select any two distinct states *x* and *y* in the controller and verify that these two states are reachable from any other reachable state. This can be verified by the two CTL formulas, *AG EF (state = x)* and *AG EF (state = y).*

The rationale of this result is as follows. If there exists a deadlock state *z*, then if *z* is not equal to either *x* or *y* then neither *x* nor *y* are reachable from *z* and none of the above CTL formulas are satisfied. If *z* is equal to *x* (*y*) then *y* (*x*) is not reachable from *z*; because *z* is a deadlock state and *x* and *y* are distinct states. Thus one of the above CTL formulas is not satisfied. Therefore, in the presence of deadlock, at least one of the above specifications is not satisfied, making the deadlock detectable. The above CTL formulas can also detect livelocks in a similar fashion. The first two SPECs in the SMV code of Fig. 6 implement deadlock-freedom verification for this example.

Safety and monotonicity of controller inputs are also verified by proper CTL formulas. Consider the machine in state *s1*. At this state *b* should never change value. This is verified by the third SPEC, *(state = s1 -> b)*.

As another case, *d* is a directed don't care on the transition from *s6* to *s0*. Thus if at any time in state *s6*, *d* has acquired the value of its terminating edge, it should not change value again. This is verified by the fourth SPEC, *(state = s6 & d -> AX d)*.

Fundamental-mode constraints can be modeled similarly. Assume the circuit has a settling-time of 1 time unit. Then, for example, for the compulsory edge *a+* in state *s1*, we should verify that *a* does not rise until the machine has been in state *s1* for at least 1 time unit. This can be verified with the SPEC *(state = s0 -> AX ((state = s1)->!a))*. Extensions to non-unit settling-times are also straight-forward.

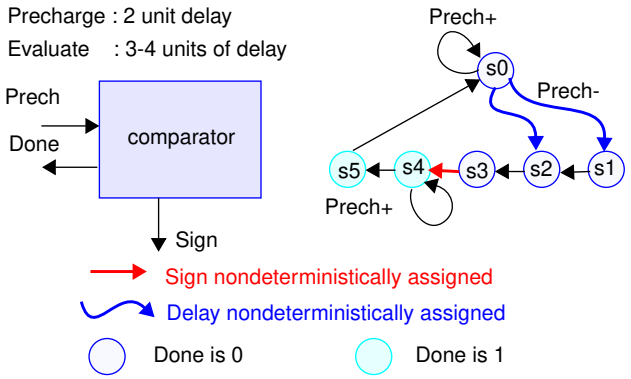

Fig 8: FSM model of a precharged comparator with bounded delays.

```
MODULE FlipFlop (clk, in1)
VAR
  out1 : boolean;
  d1 : delay (out1);
DEFINE
  out := d1.out;
ASSIGN
  init (out1) := in1;
  next (out1) := case
    !in1 = out1 & !clk & next (clk) : in1;
     1 : out1;
  esac;
SPEC AG (in1 & !clk -> AX (clk -> (in1 & AX (in1 & AX in1))))
SPEC AG (!in1 & !clk -> AX (clk -> (!in1 & AX (!in1 & AX !in1))))
SPEC AG (in1 & !clk -> AX (!in1 -> (!clk & AX (!clk & AX !clk))))
SPEC AG (!in1 & !clk -> AX (in1 -> (!clk & AX (!clk & AX !clk))))
```

Fig. 9: SMV code for a flip-flop with setup and hold times.

We have developed a CAD tool that automatically generates all safety, monotonicity, and fundamental-mode SPECs for all burst-mode controllers of a given system.

The SMV model in Fig. 6 assumes that the controller has a fixed delay of one; that is, after an input burst has occurred, the state will change in one time unit. To implement larger delays, a new SMV variable *wait* is introduced, which may take on any value in the specified delay range, minus 1 (see Fig. 7). This variable is a down-counter that is loaded as soon as a new state is entered and it starts to decrement as soon as all edges of the next input burst have occurred. A state transition happens whenever the input burst has occurred and *wait* has become zero; at the same time, *wait* is set to its maximum value minus 1. In this model, for those states which have only one state transition out of them, the transition can be fired only looking at *wait = 0*. Fig. 7 illustrates the necessary changes for the case of delays larger than 1. For a more accurate model, the outputs can be declared as variables which change value a specified delay after the arrival of the input burst. The *wait* variable can be used to keep track of the passage of time for all output variables.

C. Modeling gates and data-path elements in SMV

We model gates and data-path elements similarly in SMV. In our methodology, whenever the functionality of such elements can be hidden, the element is abstracted as a non-deterministic delay line with bounded delays. The delay range may also be made dependent on the current function of the element. For example, to model a precharged domino logic functional unit, we can consider different delay ranges for the precharge and evaluate phases. Wherever the output of a data-path element is going to determine the state of the whole system, it is modeled as a nondeterministic variable, so that all possible outcomes can be considered.

As an example, the abstract state transition diagram of a comparator is shown in Fig. 8. The *comparator* interacts with its controller through a four phase communicating protocol via the input signal *Prech* and the output signal *Done*. The *comparator* is precharged when the *Prech* input goes high, and it is assumed that the precharge takes two time units. The completion of precharge is indicated by the *Done* signal going low. The *comparator* starts evaluating when the *Prech* input goes low. The evaluation phase takes 3 or 4 time units, and its completion is indicated by the *Done* signal going high.

The modeling of the *comparator* in SMV closely follows that of controllers. *Done* is declared as a SMV variable because it is actually part of the state of the component. To model the delay of the unit, an extra variable *wait* is used as in the case of the controller example. To model the variable delay of the precharge and evaluate phases, at the end of each phase *wait* is non-deterministically assigned a number within the delay range of the next phase. After the next phase starts (when *Prech* changes value and becomes equal to *Done*) at each following step *wait* is decremented until it becomes zero, at which point the *Done* signal toggles. The authors in [14] proposed a similar approach to model gates with bounded delays.

Assuming that the communication behavior of the overall system depends only on the *Sign* output of the *comparator*, and not the value of the actual output result, we can abstract away the result, and also model *Sign* as a non-deterministic variable which may change value only on the indicated state transitions. This will, in effect, model all possible communication behaviors of the system which depend on the behavior of *Sign*.

The following CTL formulas can verify the correct behavior of the *comparator*. They can verify that the four phase protocol of communication is followed by the *comparator* and its controller; that is, each request on *Prech* line will not be removed until it is acknowledged. They also verify that any request to the unit is eventually acknowledged.

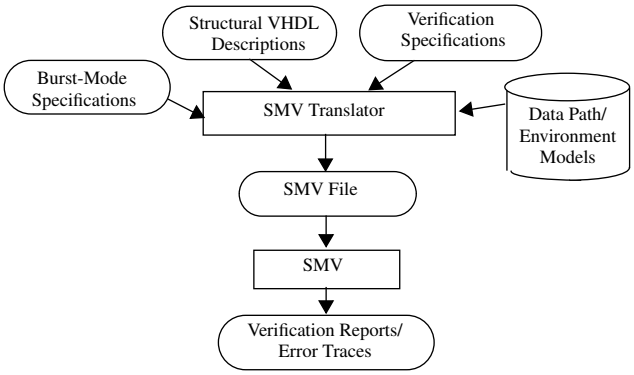

Fig. 10: The tool flow for our verification methodology

SPEC AG ((Done & Prech) -> A [Prech U !Done])
SPEC AG ((!Done & !Prech) -> A [!Prech U Done])

Hazard-freedom for gates is similarly checked by proper CTL formulas, making sure that no input changes before the output is set to its final value.

D. Modeling memory elements in SMV

For memory elements, e.g., latches or flip-flops, compliance to the setup and hold times may also need to be verified. Fig. 9 shows the SMV code for a flip-flop with setup and hold times equal to three units of time.

All the transitions of the flip-flop output, *out*, are synchronized with the rising edge of the clock input. The flip-flop output changes one unit of time after *clock* rises which is modeled using a delay element *delay* that has unit delay. The four given SPECs verify that the input of the flip-flop is stable long enough before and after the clock rises.

V. The Tool Flow

The tool flow of our verification methodology is depicted in Fig. 10. The two tools we use are a SMV translator that we developed and the SMV system developed by McMillan [17]. The SMV translator reads a structural VHDL description of the system. For each structural component, it attempts to read its burst-mode specification. If it exists, the program translates the specification into SMV code and automatically generates the associated CTL formula specifications. If a burst-mode specification does not exist, the program assumes that the module is either a data path unit or the environment. In this case, the program searches for its description in our library of predefined SMV modules for data path and environmental components. The program then combines all modules into a single file in which the user can add any additional verification tasks. This file is fed into the SMV system which reports whether each specification is satisfied or not and provides a counter-example for each specification that is not satisfied.

We used SMV version 2.4 in which we incorporated a variable ordering heuristic described by Aziz *et al.* [18] to automatically find a good variable ordering. We used the option to compute the reachable state space before checking the specifications which made dramatic improvements in run-times. It should be noted that our version of SMV only supports *monolithic transition relations* [5]. More advanced techniques that use *partitioned transition relations* (e.g., [5, 19]) may reduce run-times of the large examples and facilitate the verification of more complex systems.

VI. Case Studies

We have applied our verification methodology to three designs. The first design is a pausible clocking interface that supports efficient communication between different clock domains. One implementation, illustrated in Fig. 1, is an enhanced version of a four-phase pausible clocking interface designed by Yun et. al [2]. The basic idea of the design is to identify when asynchronous inputs to the synchronous component are changing and if necessary pause the clock of the synchronous system to prevent metastability. Variations on the design that use a two-phase communication protocol strategy were also verified. In addition to hazard-freedom of the gates and input safety of the burst-mode controllers FSM1 and FSM2, the principle verification task was to test the setup and hold time constraints on the synchronous flip-flops FF1 and FF2. Numerous earlier designs were tested and problems such as deadlock, input non-monotonicity, and setup violations were discovered until safe designs were finally achieved.

Table 1: Verification Run-Time Results

Example	Time Step of One State Transition	No of Specs	Total BDD Nodes	TR BDD Size	Reached States	State Space	Run-time (sec)
DIFFEQ Initial Model	1.0 ns	58	12,494	2,758	17,212	2.65e+08	21.5
DIFFEQ Back-annotated Models	0.5 ns	51	457,643	71,516	3.82e+7	2.94e+20	23,452
	0.75 ns	51	255,504	60,619	10.0e+7	6.45e+17	7,295
	1.0 ns	51	210,293	56,667	2.78e+6	6.41e+17	3,042
	1.25 ns	51	162,089	45,618	9.5e+5	7.67e+16	1,792
KYY-PCI-Old	1.0 ns	35	111,528	37,128	2,433	2.29e+10	400
KYY-PCI	1.0 ns	27	170,101	52,575	10,701	2.97e+13	437
USC-PCI 2-phase	1.0 ns	32	24,182	9,352	11,552	1.36e+9	14
USC-PCI 4-phase	1.0 ns	22	10,171	1,230	5,529	4.53e+8	4
T-FIFO-8	1.0 ns	52	13,728	2,229	136,862	1.80e+16	6
T-FIFO-16	1.0 ns	100	44,722	4,293	3.52e+09	5.07e+30	42
T-FIFO-24	1.0 ns	148	112,604	6,357	8.99e+13	1.43e+45	234
T-FIFO-32	1.0 ns	196	time-out	8,421	time-out	time-out	time-out

The second example is an asynchronous differential equation solver chip that uses timing assumptions to reduce the control overhead to less than 10%, making the design almost 50% faster on average than any comparable synchronous design [1]. The design implements a well-known numerical algorithm to solve a specific differential equation. Initially, a simplified model of the chip was used for verification. The chip consists of two self-timed ALUs, two self-timed multipliers, four extended burst-mode controllers, and one C-element. Bounds for ALU and multiplier precharge and evaluation delays were estimated. Specifically, the self-timed ALUs were modeled as having a precharge delay between 1 ns and 2 ns and an evaluation delay between 2 ns and 4 ns. The self-timed multiplier was modeled with a precharge delay of 2 ns and an evaluation delay between 5 ns and 6 ns. The controllers were non-trivial containing between 4 and 12 states. They were modeled with a fixed delay of one unit of time, with output signal transitions all occurring simultaneously with the state transitions. The chip's environment was modeled with a minimum response time of 2 ns and an unbounded maximum response time.

Verification and design proceeded in a ping-pong fashion. Once our verification discovered a bug, the design was altered and re-verified. Numerous fixes included missing DDCs, monotonicity violations under particular timing assumptions, deadlock, etc. Because many of the bugs were due to particular combinations of data-dependent delays in the data path, we concluded that the bugs might have been easily missed by simulation. In total, the verifier discovered over 10 errors that we had not initially considered. Additional details can be found in [1].

After laying out the design, more precise delay information was extracted from SPICE simulations. Using these delays, we created a more accurate model of the chip. First, buffers that drive control signals over long wires, which were not in the first model, were added. Then, for each component/signal transition, we conservatively modeled its delay by non-deterministically setting it to a value that lied within the tightest range of discrete time points that encompassed the delay bounds obtained by SPICE.

The more detailed design was modeled using a number of different time steps. For the resolution of 0.5 nanoseconds, the finest one used, no specifications were violated. Even when coarse resolution models were used, only three of the specifications yielded false negatives. In fact, a couple of these false negatives could be easily avoided by manually increasing the accuracy of the model (e.g., by enforcing a known ordering of output events).

Finally, to show the full potential of our method, we analyzed a simple scalable example, a timed asynchronous N-stage FIFO, illustrated in Fig. 11. Using a time-step of 1ns, a variety of N-stage FIFOs were verified. A 24-stage FIFO that has 1.23e+13 reachable states was successfully verified in under 4 minutes. A 32-stage FIFO, however, did not complete within 3 hours. This failure underscores an important point. While symbolic techniques often mitigate the state explosion problem, they are not a guarantee. Consequently, alternative techniques to handle the state explosion problem are an important area of research (e.g., zero-suppressed

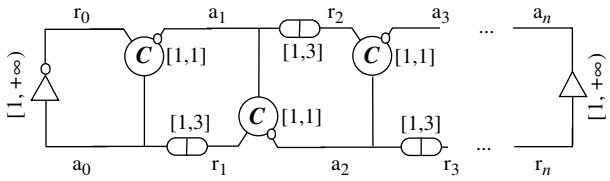

Fig. 11: An N-stage FIFO (3 stages are shown).

BDDs [20] and cube approximations [3]).

The experiments were run on a SPARCStation 5 with 32 megabytes of memory and are summarized in Table 1. The columns of the table show the time step used, the number of CTL specifications verified, the total number of BDD nodes allocated, the size of the monolithic transition relation BDD (used to find the reachable states), the number of reachable states found, and the size of the total state space. The run-time results demonstrate that our verification methodology is computationally practical for real designs.

VII. Conclusion

This paper describes a tool-supported verification methodology for RTL asynchronous and heterogeneous hardware systems whose correctness depends on timing assumptions. The paper illustrates that many desired system properties relate to correctness of the timed hardware protocols. The paper demonstrates that the application of symbolic model checking using discretized delays has been practical for two real-life non-regular systems of significant size, having state spaces of over 10^8 states. Moreover, a traditional scalable example that has over 10^{13} reachable states was verified.

Acknowledgments

This work was partially funded by a gift from Intel, a NSF CAREER Award MIP-9502386, and a research seed grant from the James H. Zumberge Faculty Research and Innovation Fund at USC. The authors would like to thank Youpyo Hong of USC for his comments and feedback.

References

[1] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo, "The Design and Verification of A Low-Control-Overhead Asynchronous Differential Equation Solver," *Proc. of the 3rd International Symposium on Advanced Research on Asynchronous Circuits and Systems,* Eindhoven, The Netherlands, April 1997.

[2] K. Y. Yun and R. P. Donohue, "Pausible clocking: A First Step Toward Heterogeneous Systems," *Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society, Oct. 1996, Austin, Texas.

[3] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng, "Efficient Verification of Determinate Speed-Independent Circuits," ICCAD-94, November 1994.

[4] J. R. Burch. E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation*, Vol. 98, No. 2, June 1992.

[5] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 4, April 1994, pp. 401-424.

[6] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertation, The MIT Press, 1988.

[7] K. L. McMillan, "A Technique of State Space Search Based on Unfolding," *Formal Methods in System Design*, Vol. 6, pp. 45-65, Kluwer Academic Publishers, Boston, 1995.

[8] O. Roig, J. Cortadella, and E. Pastor, "Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets," in *16th Intl. Conf. on Theory and Application of Petri-Nets*, Torino, Italy, June 1996.

[9] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. Ph.D. Thesis, Carnegie Mellon University, Pittsburg, PA, August 1992.

[10] J. R. Burch. "Combining CTL, Trace Theory and Timing Models," in *Proc. of Automatic Verification Methods for Finite State Systems*. pp. 334-348, Grenoble, France, 1989.

[11] J. R. Burch. "Modeling Timing Assumptions with Trace Theory," in *Proc. of IEEE International Conference on Computer Design*. pp. 208-211,1989.

[12] T. G. Rokicki and C. J. Myers, "Automatic Verification of Timed Circuits," in *Proc. of the Intl. Conf. on Computer Aided Verification*, pp. 468-480, Springer-Verlag, 1994.

[13] T. Yoneda, A. Shibayama, B.-H. Schlingloff, and E. M. Clark. "Efficient verification of parallel real-time systems," in *Proc. of the Intl. Conf. on Computer Aided Verification*, pp. 321-332, July 1993.

[14] K. Hamaguchi, H. Hiraishi, and S. Yajima, "Formal Verification of Speed-Dependent Asynchronous Circuits Using Symbolic Model Checking of Branching Time Regular Temporal Logic," *Proc. of the Intl. Workshop on Computer Aided Verification*, Aalborg, Denmark, July 1991; LNCS Vol. 575, pp. 410-420.

[15] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. Ph.D. Thesis, Stanford University, Stanford, California, 1994.

[16] K. Y. Yun and D. L. Dill, "Automatic Synthesis of 3D Asynchronous State Machines," In *Proceedings of the 1992 IEEE/ACM Intl. Conf. on Computer Aided Design*, pp. 576-580, IEEE Computer Society, Nov. 1992, Santa Clara, California.

[17] K. L. McMillan, *Symbolic Model Checking*. New York, Kluwer, 1993.

[18] A. Aziz, S. Tasiran, and R. Brayton, "BDD Variable Ordering for Interacting Finite State Machines," *Proc. of the 31st Design Automation Conference*, June 1994, pp. 283-288.

[19] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. Ph.D. Thesis, Stanford University, Stanford, CA, December 1995.

[20] T. Yoneda, H. Hatori, A. Takahara, S. Minato. "BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of Petri Nets," in *Proc. of the Intl. Conf. on Formal Methods in Computer-Aided Design*, Nov. 1996.

Vida Vakilotojar: Recieved her BS in Electrical Engineering from Sharif University of Technology and her MS in Computer Engineering from AmirKabir University of Technology, Tehran, in 1990 and 1995 respectively. She is currently a Ph.D. student at the University of Southern California, EE-Systems department. Her research interests are in formal verification of mixed synchronous and asynchronous VLSI systems, with an emphasis on timing verificatrion.

Peter A. Beerel: Received his B.S.E. degree in Electrical Engineering from Princeton University in 1989 and his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University in 1991 and 1994, respectively. He was a member of the technical program committee of the Second Working Conference on Asynchronous Design Methodologies, the 1995 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU'95), and the third International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '97). He is now the Program Co-chair for ASYNC'98. His research interests include computer-aided-design of asynchronous and mixed asynchronous/synchronous VLSI systems as well as formal verification of communication protocols.

Dr. Beerel was a co-winner of the Charles E. Molnar award for two papers published in ASYNC'97 that best bridged theory and practice in asynchronous design. He is a recipient of an NSF Career Award and a 1995 Zumberge Fellow. He has also been a primary consultant for the Intel Corporation on their Asynchronous Instruction Decoder Project.