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: 213-740-9079
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 lead to a system model comprised of communicating finite state machines 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.

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 lead 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 40% faster on average than any comparable synchronous design [14], and efficient pausible clocking strategies for interfacing different clock domains [15]. 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, 5, 8, 2, 12], verifying asynchronous circuits with timing assumptions has been rather limited. [4, 10, 13] have addressed verifying gate-level timed circuits, but their techniques have been limited to small circuits due to the state explosion problem associated with their underlying explicit state techniques. The authors in [6] have addressed using implicit state techniques for gate-level timed circuits and we apply some of their modeling techniques to the register-transfer level (RTL). In addition, these authors address verifying only hazard-freedom and not properties specific to RTL designs.

The paper begins by describing our *timed asynchronous system* design methodology. It describes typical components of such asynchronous and heterogeneous systems, including data path components with data-dependent delays (i.e., adders/multipliers), distributed asynchronous control circuits (i.e., XBM burst-mode controllers [17]), and interface glue-logic (i.e., mutual-exclusion elements). The paper then describes the verification problems specific to RTL design 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 bounded delay assumptions sometimes necessary to ensure their correctness.

We then describe our RTL verification methodology. First, we abstract each system component to a synchronous finite state machine (FSMs) 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.

Finally, the paper presents a prototype tool to support this methodology and describes two real-life case studies in which the verification and design proceeded concurrently. 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. Using this tool, we quickly discovered and corrected many design errors that would have been difficult to find using simulation.

The remainder of this paper is organized as follows. Section II describes the different timed data path and control components that may make up a timed asynchronous system. Section III describes the RTL verification tasks relevant to such systems. Section IV describes modeling techniques in SMV for the different system components, and CTL specification for the needed verification tasks. Section V describes the verification methodology tool flow and section VI describes two case studies: a pausible clocking interface module and a differential equation solver. Finally, section VII gives our conclusions.

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 conform 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 consist of 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

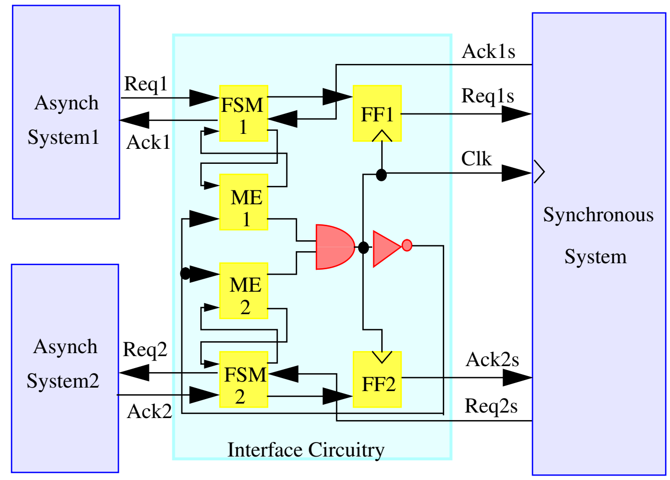

Fig. 1. A pausible clocking interface, an example of a timed asynchronous system.

or static gates [10, 17]. 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 set-up and hold times are satisfied at the synchronous latches, 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 Systems at the RT Level

Whenever 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.

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 of 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 [11, 10] are straight-forward.

B. Extended Burst Mode specification formalism

An XBM controller is formally specified by a tuple *g:<In, In0, Out, Out0, S, Trans (S-Src, S-Dest, In-Burst, Out-Burst), 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. Each state transition is labelled with an input-burst, *In-Burst*, and a possible output-burst, *Out-Burst*. A burst is a non-empty set of signal transitions. *s0* is the start state of the machine. An XBM specification can automatically be realized into an asynchronous circuit implementation [17]. An example of a burst-mode specification is shown in Fig. 2.

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*. A state transition occurs only if all the conditions are met and all the terminating edges have occurred. 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.

There are a number of communication constraints that the environment of the circuit must satisfy. First, each edge signals 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 signal that are not mentioned are not allowed to change.

An extended burst mode asynchronous finite state machine is specified by a state diagram which consists of a finite number of states, a set of labeled state transitions connecting pairs of states, and a start state, *s0*. Each state transition 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

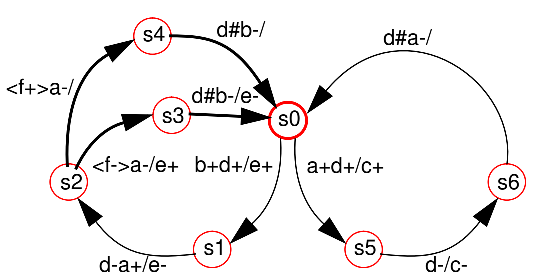

Fig. 2. Example of a burst-mode specification.

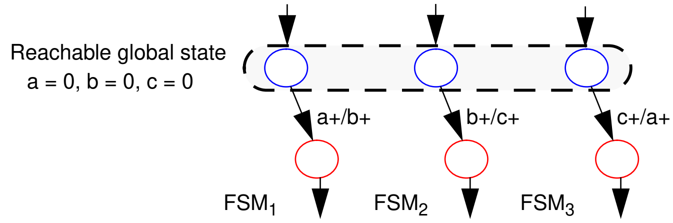

Fig 3. An example of deadlock; In the highlighted state none of the FSMs can ever fire.

(terminating or directed don't care), at least one of which 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 a new 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.

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 violations. In the following, the above properties are illustrated given 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.

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 the following values for the signals $a = b = c = 0$, 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 the two following sequence of global state changes: Seq_1 = *000, 100, 110, 111, 121* and 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*, *c* 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 the two following sequence of states: Seq_1 = *000, 100, 110, 111* and 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 more strict 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 if no compulsory transitions occur until the FSM has been in the corresponding next 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.

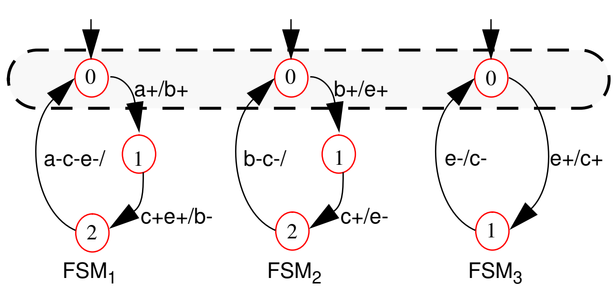

Fig. 4. Example of potential monotonicity violation dependent on timing assumptions

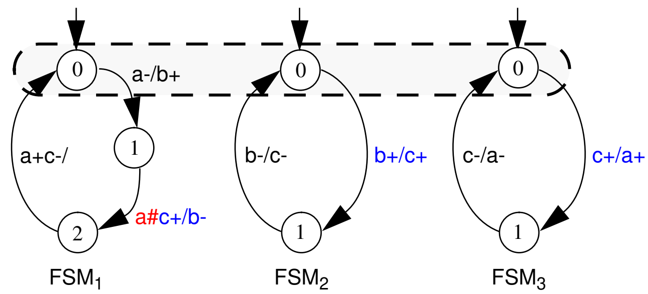

Fig. 5. An example where a DDC is needed if timing assumptions don't hold.

D. Set-up and hold times

Currently, we consider set-up and hold-times for only memory elements like latches or flip-flops. The data input of the element should be stable, before and after the clock/enable is asserted, for the set-up time and hold time, respectively. This ensures that the correct value of the data is latched. Set-up 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. Modeling in SMV

This section describes how we use SMV to verify RTL timed asynchronous systems. We will first briefly introduce SMV and then describe our modeling techniques in detail.

A. The SMV system

The SMV system is a tool for checking whether finite state systems satisfy specifications described in the computational tree logic (CTL). The input language of SMV allows the description of finite state systems that range from completely synchronous to completely asynchronous. The logic CTL allows a rich class of temporal properties, including safety, liveness, and fairness to be specified in a concise syntax. SMV uses OBDD-based symbolic model-checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied [9].

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 all 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.

In modeling timed systems and trying to facilitate modeling bounded delays, we abstract each system component as a synchronous finite state machine (FSMs) for which time is discretized. We encode passage of time using explicit timers for any 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. We thus assume that the shortest event in the system takes one unit of time and specify other delays accordingly. To make all timers progress at the same rate, we verify the synchronous composition of the components using the simultaneous semantics of SMV. We use either fixed delays or delay bounds for system components, which are discretized estimates of the real delays. This approximation of the system has not yet been proved to guarantee absence of false positive verification results; that is it may not be a conservative approximation. Nevertheless, we have been able to find many design bugs in early design stages, where exact delay bounds of the components were not yet known.

B. Modeling XBM controllers

We will use the XBM controller described 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 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, but since the different output delays in our controllers were about the same we have adopted this model. 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, in state *s2*, when *a* falls, depending on the value of the conditional signal *f*, the machine will have a transition to *s3* or *s4*.

```
MODULE XBM_Controller (a, b, d, f)
VAR
state : {s0, s1, s2, s3, s4, s5, s6};
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 (state = s6 -> d)
SPEC    (!state = s1 -> AX (state = s1 -> !a))
```

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

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 following CTL formulas:

AG EF (state = x),   *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 the deadlock-freedom verification for this example.

Safety and monotonicity of controller inputs are also verified by proper CTL formulas in the SPEC section of the module. 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 transitions from *s6* to *s0*. Thus if at any time in state *s6*, *d* has acquired the value of its terminating edges, it should not change value again. This is verified by the fourth SPEC,

(state = s6 & d -> AX (state = s6 -> d)).

As the final example consider state *s0*. Here, any of the edges *a*+, *b*+, or *d*+ may occur, but once in state *s1*, *a*+ should not have occurred, and once in state *s5*, *b*+ should not have occurred. Here, *a*+ is a compulsory edge on the transition from *s1* to *s2*. We notice that *a* should be low at the entry point of state *s1*, while it can later rise. Thus we should have the last

```
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.

indicated SPEC,

(!state = s1 -> AX (state = s1 -> !a)).

We may have similar formulas for all other similar monotonicity and safety constraints. All safety and monotonicity SPECs are automatically generated while the XBM specification are being translated to SMV code.

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.

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. This variable is a down-counter that is loaded as soon as a new state is entered and which 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. Whenever 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 an adder is shown in Fig. 8. The *adder* interacts with its controller through a four phase communicating protocol via the input signal *Prech* and the output signal *Done*. The *adder* 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 *adder* 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 *adder* in SMV closely follows that of the previous controller. *Done* is declared as a SMV variable since 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 [6] proposed a similar approach to model gates with bounded delays.

Assuming that the communication behavior of the overall systems depends only on the *Carry* output of the *adder*, and not the value of the actual output result, we can abstract away the result, and also model *Carry* 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 temporal behavior of *Carry*.

The following CTL formulas can verify the correct behavior of the *adder*. They can verify that the four phase protocol of

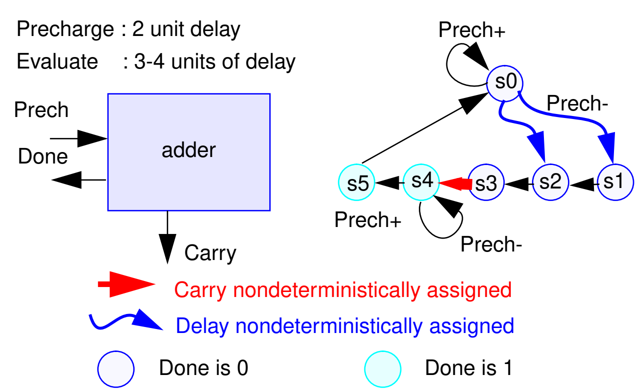

Fig 8. FSM model of a precharged adder with bounded delays.

```
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 set-up and hold times.

communication is followed by the *adder* and its controller; that is, each request on *Prech* line will not be removed until it is acknowledged. They also verified that any request to the unit is eventually acknowledged.

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 specifications, 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 set-up and hold times may also need to be verified. Fig. 9 shows the SMV code for a flip-flop with set-up and hold times equal to two 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 right after *clock* rises. 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 we created and the SMV system developed by McMillan [9]. The SMV translator reads a structural VHDL description of the system. For each structural component, it attempts to read its burst-mode specification, in XBM format. 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 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 specifications is satisfied or not and provides a counter-example for each specification that is not satisfied. To speed up the run time of SMV, variable ordering heuristics of [1] are implemented into SMV.

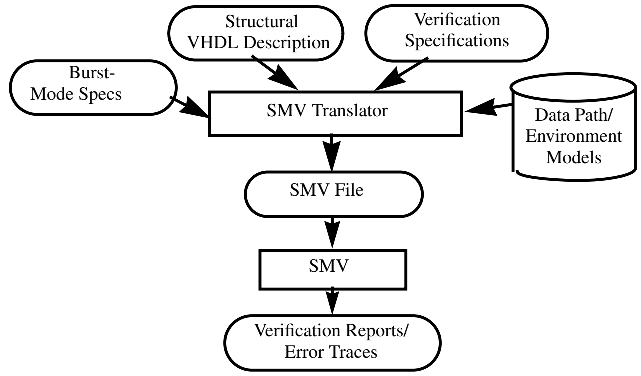

Fig. 10: The tool flow for our verification methodology

VI. Case Studies

We have applied our verification methodology to two real-designs. The first is a pausible clocking interface design to support efficient communication between different clock domains. The design, illustrated, in Fig. 1, is an enhanced version of a four-phase pausible clocking interface designed by Yun et. al [15]. 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. 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 monotonicity, and setup-violations were discovered until this safe design was finally achieved.

TABLE I: Verification Run-Time Results

Example	Deadlocks	No of Specs	UserTimes Sec	Total BDD Nodes	Transition- Table BDD Nodes	Reachable State Size	State Space Size
Diffeq		58	21.5	12494	2758	17212	2.65e+8
	✓	58	1	10017	2748	256	2.65e+8
PCI		55	159	13725	829	3344	1.13e+8

The second example was an asynchronous differential equation solver that uses timing assumptions to reduce the control overhead to less than 12% making the design approximately 40% faster on average than any comparable synchronous design [14]. The design implements a well-known numerical algorithm to solve a given differential equation. The implementation consisted of two self-timed adders, with precharge delay between 1ns and 2ns and evaluate delay between 2ns and 4ns, two self-timed multipliers, with precharge delay of 2ns and evaluate delay between 5ns and 6ns, 4 associated extended burst-mode controllers, with delay 1ns and between 4 and 12 states, one 1ns C-element for glue-logic, and one component modeling the environment whose minimum response time is 2ns and whose maximum response time is unbounded. As a sanity check we also verified an altered version of the design in which we had intentionally added deadlock.

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 may have been easily missed by simulation. In total the verifier discovered over 10 errors that we had not initially considered.

We used SMV version 2.4 on a SPARCStation 5 with 32 Megabytes of memory. We manually applied a variable ordering heuristic described by Aziz et al. [1] in order to find a good variable ordering. Also, we used the option to compute the reachable state space before checking the specifications which made dramatic improvements in run-times. The overall run-times, depicted in Table I, demonstrate that our verification methodology is computationally quite practical for interesting designs.

VII. Conclusion

This paper describes a tool-supported verification methodology for RTL asynchronous and heterogeneous hardware systems whose correctness depend 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 is practical for two real-life, non-regular systems of significant size, having state spaces of over 10^8 states.

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 for his comments and feedback.

References

[1] 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.

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

[3] 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.

[4] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. Ph.D. Thesis, Carnegie Mellon University, Pittsburg, PA, August 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] K. Hamaguchi, H. Hiraishi, and S. Yajima, “Formal Verification of Speed-Dependent Asynchronous Circuits Using Symbolic Model Checking of Branching Time Regular Temporal Logic,” *Computer Aided Verification, Proc. of the 3rd Intl. Workshop, CAV ‘91*, Aalborg, Denmark, July 1991; LNCS Vol. 575, pp. 410-420.

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

[8] 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.

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

[10] C. J. Myers, T. G. Rokicki, T. H.-Y. Meng, “Automatic Synthesis of Gate-Level Timed Circuits with Choice,” in *Chappel Hill Conf. on Advanced Research in VLSI*, ARVLSI ‘95, March 1995.

[11] S. M. Nowick and B. Coates, “UCLOCK: Automated Design of High-Performance Unclocked State Machines,” In *Proc. of the ICCD: VLSI in Computers and Processors*, Oct. 1994.

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

[13] 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.

[14] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo, "A Low-Control-Overhead Asynchronous Differential Equation Solver,” *Proc. of the 22nd European Solid-State Circuits Conference*, 1996. Neuchatel, Switzerland, September 17-19.

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

[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*, pages 576-580, IEEE Computer Society, Nov. 1992, Santa Clara, California.

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