A Framework for Asynchronous Circuit Modeling and Verification in ACL2

Cuong Chau[1], Warren A. Hunt, Jr.[1], Marly Roncken[2], and Ivan Sutherland[2]

[1] Department of Computer Science
The University of Texas at Austin
Austin, TX, USA

[2] Maseeh College of Engineering and Computer Science
Portland State University
Portland, OR, USA

{ckcuong,hunt}@cs.utexas.edu, marly.roncken@gmail.com, ivans@cecs.pdx.edu

Abstract. Formal verification of asynchronous circuits is known to be challenging due to highly non-deterministic behavior exhibited in these systems. One of the main challenges is that it is very difficult to come up with a systematic approach to establishing invariance properties, which are crucial in proving the correctness of circuit behavior. Non-determinism also results in asynchronous circuits having a complex state space, and hence makes the verification task much more difficult than in synchronous circuits. To ease the verification task by reducing non-determinism, and consequently reducing the complexity of the set of execution paths, we impose design restrictions to prevent communication between a module M and other modules while computations are still taking place that are internal to M. These restrictions enable our verification framework to verify loop invariants efficiently via induction and subsequently verify the functional correctness of asynchronous circuit designs. We apply a link-joint paradigm to model asynchronous circuits. Our framework applies a hierarchical verification approach to support scalability. We demonstrate our framework by modeling and verifying the functional correctness of a 32-bit asynchronous serial adder.

Keywords: asynchronous circuit modeling, asynchronous circuit verification, non-deterministic behavior, hierarchical verification, link-joint model, mechanical theorem proving, self-timed serial adder

1 Introduction

Asynchronous (or self-timed) circuits have shown their potential advantages over synchronous (or clock-driven) circuits for low power consumption, high operating speed, low electromagnetic interference, elimination of clock skew problems, better composability and modularity in large systems, etc [10,15]. Nonetheless, the asynchronous paradigm exposes great challenges in both design and verification that are not found in the clocked paradigm. It is still a daunting challenge

O. Strichman and R. Tzoref-Brill (Eds.): HVC 2017, LNCS 10629, pp. 3–18, 2017.
https://doi.org/10.1007/978-3-319-70389-3_1

to verify the correctness of asynchronous systems at large scale, mainly due to the high degree of non-determinism for event ordering inherent in such systems. Since verification is a critical component of any complex digital design, *scalable* methods for asynchronous system verification are highly desirable.

Our effort is complementary to the work introduced by Park et al. [11] to validate timing constraints for delay-insensitive handshake components. The authors used model-checking to perform timing verification on handshake components, to validate the correctness of local communication or handshake protocols with respect to delays in gates and wires. Our approach relies on such analysis to justify our abstraction of self-timed circuits to finite-state-machine representations of networks of communication channels, thus ignoring circuit-level timing constraints. Using the ACL2 theorem-proving system [7], we present a framework for specifying and verifying the functional correctness of those networks.

Our work focuses on developing scalable methods for reasoning about the functional correctness of self-timed systems. Our approach applies induction reasoning to establishing loop invariants of self-timed systems. We use the DE (Dual-Eval) system [3], which is built using the ACL2 theorem-proving system, to specify and verify self-timed circuit designs. DE is a formal hardware description language that permits the hierarchical definition of finite-state machines. It has shown its capabilities to specify and verify synchronous microprocessor designs [1,4]. A key feature of the DE system is that it supports *hierarchical verification*, which is critical in verifying the correctness of circuit behavior at large scale. It also provides a library of verified hardware circuit generators that can be used to build and analyze more complex hardware systems [1].

We use DE to model self-timed circuits as networks of communication and computation primitives that operate with each other locally via the *link-joint model* proposed by Roncken et al. [13], a universal model for various self-timed circuit families. To our knowledge, we are the first to model self-timed circuits using the link-joint model in a theorem-proving system. We also develop a method for verifying functional properties of self-timed circuits constructed via the link-joint model.

We model the non-determinism of self-timed circuit behavior by consulting an *oracle* field — an external field we inject into the circuit model. The challenge in reasoning about the correctness of non-deterministic systems is that their state space is not only large as compared to synchronous systems, but also ill-structured in such a way that computing invariants in those systems becomes highly complicated. Since invariants are crucial properties for proving the correctness of circuit behavior, we are interested in developing a method for computing invariants of self-timed circuits systematically, thus ultimately making the verification of these systems tractable. Our approach attempts to reduce non-determinism, consequently reducing the complexity of the set of execution paths, by imposing design restrictions to prevent communication between a module M and other modules while computations are still taking place that are internal to M. These design restrictions enable our verification approach to verify loop invariants efficiently via induction and subsequently verify the func-

tional correctness of self-timed circuit designs. We demonstrate our framework by modeling and verifying the functional correctness of a 32-bit self-timed serial adder [3]. This provides a significant first step towards the formal verification of arbitrary asynchronous designs.

The rest of the paper is organized as follows. Related work is given in Section 2. An overview of the DE system is presented in Section 3. Section 4 describes our self-timed circuit modeling and verification approach. Section 5 demonstrates our approach by describing our modeling and verification of a 32-bit self-timed serial adder. Possible future work and concluding remarks are given in Sections 6 and 7, respectively.

2 Related Work

Asynchronous circuit verification is an active research area in the hardware community. Many efforts in this area have focused on verifying properties of asynchronous circuits by applying timing verification techniques [6,8,9,11]. Park et al. [11] presented their framework, called ARCtimer, for modeling, generating, verifying, and enforcing timing constraints for individual delay-insensitive handshake components. ARCtimer uses the general-purpose model checker NuSMV to perform timing verification of handshake components. The authors' main goal was to verify that the network of logic gates and wires and their delays meet the component's communication protocol specification. Our goal is complementary: to verify that the network of handshake components and their protocols meets its functional specification, while ignoring circuit-level timing constraints that can be handled by tools like ARCtimer.

Verbeek and Schmaltz [17] formalized and verified with the ACL2 theorem prover *blocking* (not transmitting data) and *idle* (not receiving data) conditions over delay-insensitive primitives in the Click library. These conditions were then used to derive SAT/SMT instances from asynchronous circuits built out of these primitives for checking deadlock freedom in those circuits. While our approach also uses ACL2 to model and verify self-timed circuits, we verify the functional correctness of self-timed circuit models.

Clarke and Mishra [2] employed model checking to automatically verify some safety and liveness properties of a self-timed FIFO queue element specified in Computation Tree Logic (CTL). The authors also presented a hierarchical method for verifying large and complex circuits. Nevertheless, their approach imposed an unrealistic assumption on self-timed circuits that each gate has one unit delay. Our approach, on the other hand, does not restrict gate delays except that they are finite.

Other previous work on asynchronous circuit verification attempted to reduce non-determinism by adding restrictions to circuit designs, as presented by Srinivasan and Katti [16] and Wijayasekara et al. [18]. Srinivasan and Katti [16]

[3] The source code for this work is available at `https://github.com/acl2/acl2/tree/master/books/projects/async/serial-adder`

applied a refinement-based method for verifying safety properties of desynchronized pipelined circuits, while Wijayasekara et al. [18] applied the same method for verifying the functional equivalence of NULL Convention Logic (NCL) circuits against their synchronous counterparts. While their verification frameworks are highly automated by using decision procedures, both provided quite limited scalability and no liveness properties were verified. Although we also impose design restrictions to reduce non-determinism, our approach is capable of verifying liveness properties as described in Section 5 in our account of the verification of a 32-bit self-timed serial adder. Our approach exploits hierarchical verification and induction reasoning to support scalability.

3 The DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines [3]. It has been shown to be a valuable tool in formal specification and verification of hardware designs [14,5]. The operational semantics for the DE language is implemented as an output "wire" evaluator, `se`, and a state evaluator, `de`. The `se` function evaluates a module and returns its outputs as a function of its inputs and its current state. The `de` function evaluates a module and returns its next state; this state will be structurally identical to the module's current state, but with updated values. The interested reader may refer to Hunt's paper [3] for details about the `se` and `de` functions.

In synchronous circuits, storage elements update their values simultaneously at every global clock tick, where the clock rate is fixed. Hence the duration represented by two consecutive `de` evaluations of a synchronous module is fixed and exactly one clock cycle. In self-timed circuits, however, storage elements update their values whenever their local communication conditions are met; and hence the duration represented by two consecutive `de` evaluations of a self-timed module varies.

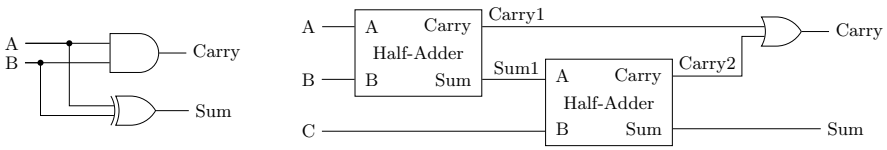


Fig. 1. Half-adder (left) and full-adder (right)

A DE description is an ACL2 constant containing an ordered list of modules, which we call a *netlist*. Each module consists of five elements in the following order: a netlist-unique module name, inputs, outputs, internal states represented by a list of occurrence names identifying those occurrences that contain state-holding devices, and occurrences. Each occurrence consists of four elements in

the following order: a module-unique occurrence name, outputs, a reference to a primitive or defined module, and inputs. For instance, the DE descriptions of the half-adder and full-adder netlists shown in Figure 1 are described below. Note that these adders are purely combinational-logic circuits; they do not contain any internal state.

```
(defconst *half-adder*
  '((half-adder
     (a b)
     (sum carry)
     ()   ;; No internal state
     ((g0 (sum)   xor (a b))
      (g1 (carry) and (a b))))))

(defconst *full-adder*
  (cons '(full-adder
          (c a b)
          (sum carry)
          ()   ;; No internal state
          ((t0 (sum1 carry1) half-adder (a b))
           (t1 (sum  carry2) half-adder (sum1 c))
           (t2 (carry)       or         (carry1 carry2))))
        *half-adder*))
```

A key feature of the DE system is that it supports hierarchical verification, which is critical in verifying the correctness of large circuit descriptions. The idea is to verify the correctness of a larger module by composing verified submodules without delving into details about the submodules. More specifically, each time a module is specified, we prove a *value lemma* specifying the module's outputs and a *state lemma* specifying the module's next state. If a module does not have an internal state (purely combinational), only the value lemma need be proven. These lemmas are used to prove the correctness of yet larger modules containing these submodules, without the need to dig into any details about the submodules. Such an approach can scale to very large systems, as has been shown on contemporary x86 designs at Centaur Technology [14]. We refer the interested reader to Hunt's paper [3] for an example of the value lemma of the full-adder mentioned above.

4 Modeling and Verification Approach

We model self-timed circuits by (1) adding local signaling to state-holding devices, (2) establishing local communication protocols, and (3) employing an oracle, which we call a collection of *go* signals, for modeling non-deterministic circuit behavior due to variable delays in wires and gates. The details of our modeling approach are described below.

In the clock-driven design paradigm, state-holding devices are all governed by a global clock signal such that their internal states are updated at the same time when the clock "ticks", which is simulated by a `de` evaluation in the DE system. There is no such global clock signal in the self-timed design paradigm.

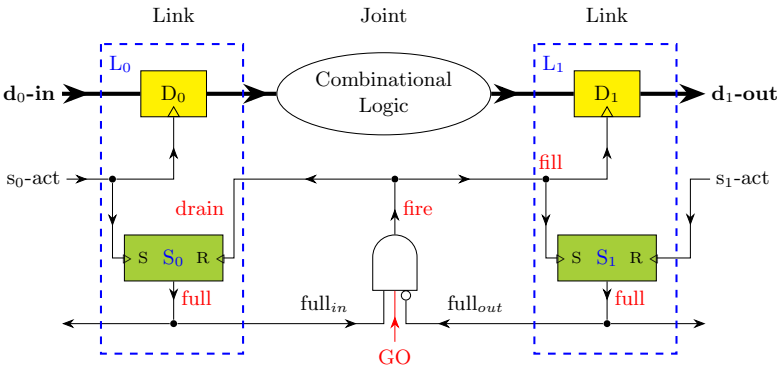


Fig. 2. Simple self-timed communication circuit using the link-joint model from Roncken et al. [13]

Thus, when a self-timed circuit is simulated by a `de` evaluation, its state-holding elements will update their states based on their inputs.

For establishing local communication protocols, we model the link-joint model introduced by Roncken et al. [13]. Our rationale for formalizing this model is the authors' demonstration that it is a universal communication model for various circuit families. In this model, links are communication channels in which data and full/empty states are stored, while joints are handshake components that implement flow control and data operations. Roughly speaking, joints are the meeting points for links to coordinate states and exchange data. Figure 2 shows an example of a simple self-timed communication circuit using the link-joint model. This circuit consists of a joint associated with an incoming link L_0 and an outgoing link L_1. In general, a joint can have several incoming and outgoing links connected to it, as depicted in Figure 3.

Links receive *fill* or *drain* commands from and report their full/empty states and data to their connected joints. A *full* link carries valid data, while an *empty* link carries data that are no longer or not yet valid. When a link receives a fill command, it changes its state to full. A link will change to the empty state if it receives a drain command. We use a set-reset (SR) latch to model the full/empty state of a link, as illustrated by the lower box in each link shown in Figure 2 [4]. The interested reader may refer to Roncken et al.'s paper [13] for other options of link control circuitry.

Joints receive the full/empty states of their links and issue the fill and drain commands when their communication conditions are satisfied. The control logic of a joint is an AND function of the conditions necessary for it to act. To enable a joint-action, all incoming links of a joint must be full and all outgoing links must

[4] Using SR latches in this manner requires an implementation to assure sufficient delay in the AND function to prevent overlap in the S and R inputs. This is handled at the circuit level by Park et al. [11].

be empty (see the AND gates in Figures 2 and 3). Due to arbitrary delays in wires and gates, enabled joints may fire in any order. We model the non-deterministic circuit behavior by associating each joint with a so-called *go* signal as an extra input to the AND function in the control logic of that joint. The value of the *go* signal will indicate whether the corresponding joint will fire when it is enabled. The idea of using *go* signals to model non-determinism was presented in a paper by Roncken et al. [12]. In our framework, when applying the `de` function that computes the next state of a self-timed circuit, only enabled joints with high values of the *go* signals will fire. When a joint fires, the following three actions will be executed in parallel [5]:

- transfer data computed from the incoming links to the outgoing links,
- fill the outgoing links, make them full,
- drain the incoming links, make them empty.

Below is our DE description of the self-timed module shown in Figure 2, where D_0 and D_1 are one-bit latches, and the combinational logic (Comb. Logic) representing the data operation of the joint is simply a one-bit buffer. This module contains state-holding devices S_0, D_0, S_1, and D_1.

```
'(link-joint
  (s0-act s1-act d0-in go)
  (d1-out)
  (s0 d0 s1 d1)   ;; Internal states
  (;; Link L0
   (s0 (s0-status)      sr    (s0-act fire))
   (d0 (d0-out d0-out-) latch (s0-act d0-in))
   ;; Link L1
   (s1 (s1-status)      sr    (fire s1-act))
   (d1 (d1-out d1-out-) latch (fire d1-in))
   ;; Joint
   (j (fire)  joint-cntl (s0-status s1-status go))
   (h (d1-in) buffer     (d0-out))))
```

We consider all possible interleavings of the *go* signals' values when reasoning about the correctness of circuit behavior. The only requirement is that when applying the `de` function to compute the next state of a module, the go signals are high for at least one enabled joint (if any such joint exists). We call this restriction the *single-step-progress* requirement.

Our framework exploits a hierarchical verification approach to formalizing single transitions of circuit behavior (simulated by `se` and `de` functions), as described in Section 3. The verification process at the module level requires us to show how several asynchronous blocks can be interconnected to provide provably correct, higher level functions. Our framework currently treats modules

[5] The work done by Park et al. [11] used ARCtimer to generate and validate timing constraints in joints. Their framework added sufficient delay to the control logic of each joint to guarantee that the clock pulse is wide enough for the three mentioned actions to be properly executed when the joint fires. Our work assumes that we have a valid circuit that satisfies necessary circuit-level timing constraints, as guaranteed by ARCtimer.

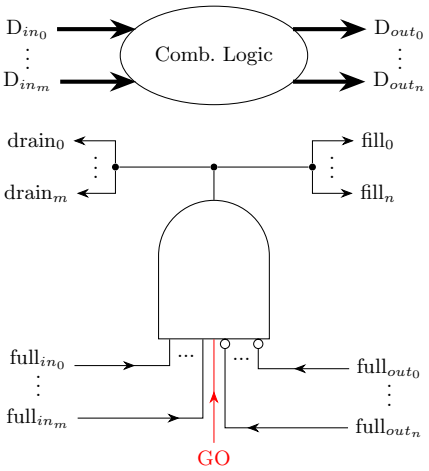


Fig. 3. Sketch of a joint with m incoming and n outgoing links [13]

as "complex" links that communicate with each other via local communication protocols. Hence, self-timed modules report both data and communication states to the joints connecting those modules. In the future, we plan to explore a notion of modules being treated as "complex" joints [6].

The communication state of a self-timed module is more complicated than that of a primitive link in the sense that it can be ready to send and receive data at the same time, or "not ready" to communicate with its connected modules. For this reason, self-timed modules use separate incoming and outgoing communication signals, whereas primitive links only need one full/empty signal for both incoming and outgoing communications. For example, the *ready-in-* (active low) and *ready-out* (active high) output signals of the module in Figure 4 are both active at the same time when the two links on the left side are empty and the three links on the right side are full. This module is in the not-ready state when the two links on the left side are full and the three links on the right side are empty.

In arbitrarily non-deterministic systems, the state space may not exhibit a clear structure for computing invariants effectively. Verification of such systems may require exploring the entire state space. To simplify the verification task by reducing non-determinism, and consequently reducing the complexity of the set of execution paths, we impose restrictions on circuit designs such that a module is ready to communicate with other modules only when it finishes all of its

[6] We choose to model modules as links for the purpose of storage-free connections between modules, since they are connected via storage-free joints in this setting. However, this modeling does not keep to the spirit of the link-joint paradigm in which computation is supposed to be done entirely in joints and links serve only to store data [13].

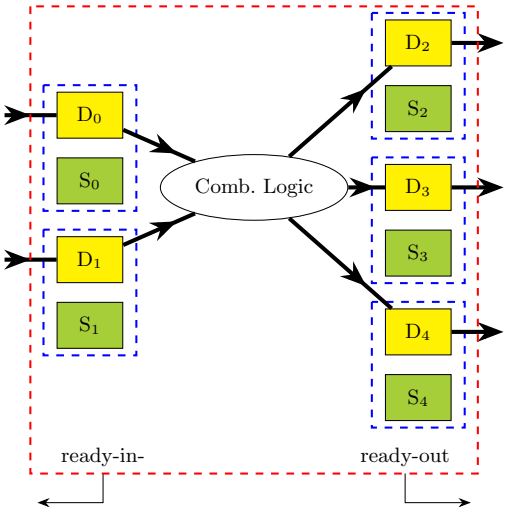


Fig. 4. Example of a self-timed module containing one joint with two incoming and three outgoing primitive links. The flow control of the joint is not shown in the figure for the sake of simplicity. Note that a self-timed module can contain several links and joints. We use this simple example for pedagogical purposes.

internal operations and becomes *quiescent*. By adding these restrictions, the state space is not only reduced but, more importantly, it also exhibits a structure for establishing loop invariants efficiently via induction. These restrictions guarantee that every module will reach a fixed point before it can communicate with other modules, and thus enable our framework to establish invariants and subsequently verify the functional correctness of circuit designs.

5 32-Bit Self-Timed Serial Adder Verification

In this section we demonstrate our framework by describing our modeling and verification of a 32-bit self-timed serial adder. This relatively simple example is sufficiently complex to demonstrate the generality of our approach. First let us introduce the *shift register* concept, which is used in constructing a serial adder. A shift register is a state-holding device that shifts in the data present at its input and shifts out the *least significant bit* (LSB) in the bit-vector whenever the register's advance ("clock") input transitions from low to high. Shift registers can have both parallel (bit-vector) and serial (single-bit) inputs and outputs. Figure 5 illustrates an example of a serial-in, serial-out, and parallel-out n-bit shift register. The figure shows that the shift register outputs both the LSB (serial-out) and the entire n-bit vector (parallel-out), but it only accepts single-bit inputs (serial-in). When the *write/shift* signal is high, the value of *Shift-Reg* will be shifted right by one position and the *bit-in* will be stored in *Shift-Reg* at the *most-significant-bit* (MSB) position.

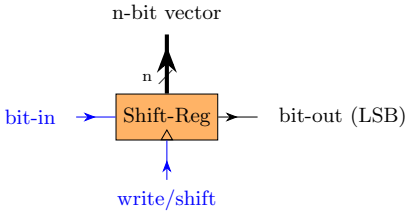


Fig. 5. Serial-in, serial-out, and parallel-out n-bit shift register

Table 1. 4-bit serial addition example. The bit-in is 0 for both shift registers A and B in this example.

A	**B**	**S**	$\mathbf{s}_i$	$\mathbf{c}_{i+1}$
1010	0011	1xxx	1	0
0101	0001	01xx	0	1
0010	0000	101x	1	0
0001	0000	1101	1	0

A serial adder is a digital circuit that performs binary addition via bit additions, from the LSB to MSB, one at a time. Bit additions are performed using a 1-bit full-adder to generate a sum bit and a carry bit, and two input operands and the accumulated sum are stored in the shift registers. Table 1 shows a 4-bit serial addition example.

We construct a 32-bit self-timed serial adder using the link-joint model, i.e., the communications between state-holding elements in the circuit are established via the link-joint model. Figure 6 shows the datapath of a 32-bit self-timed serial adder; the control path is elided for the sake of simplicity [7]. In terms of the link-joint model, the figure displays only the data operations of the joints (circles) and the link data (rectangles) [8]; it abstracts both the flow control of the joints and the link states. To model non-determinism, we associate each joint with a *go* signal. In Figure 6, we see that the *go* signals point to the data operations of their corresponding joints, and we also refer to joints by their *go* signals' names. But note that these *go* signals are indeed provided as inputs to the AND gates in the control logic of their corresponding joints, which are omitted from the figure. The roles of the storage elements (the rectangles in Figure 6) used in the serial adder are described below.

– Two 32-bit input operands are stored in *Shift-Reg*$_0$ and *Shift-Reg*$_1$, and the 32-bit sum is stored in *Shift-Reg*$_2$. The final 33-bit sum (including the carry-out) is stored in the regular register *Result*.

[7] The dotted lines emanating from the *Done-* latch represent the fact that the output of *Done-* is used in the control path, not in the datapath.

[8] Our approach currently declares shift registers as primitive state-holding devices and uses them to store link data. In the future, we plan to be faithful to the link-joint methodology by replacing the shift registers by links and joints.

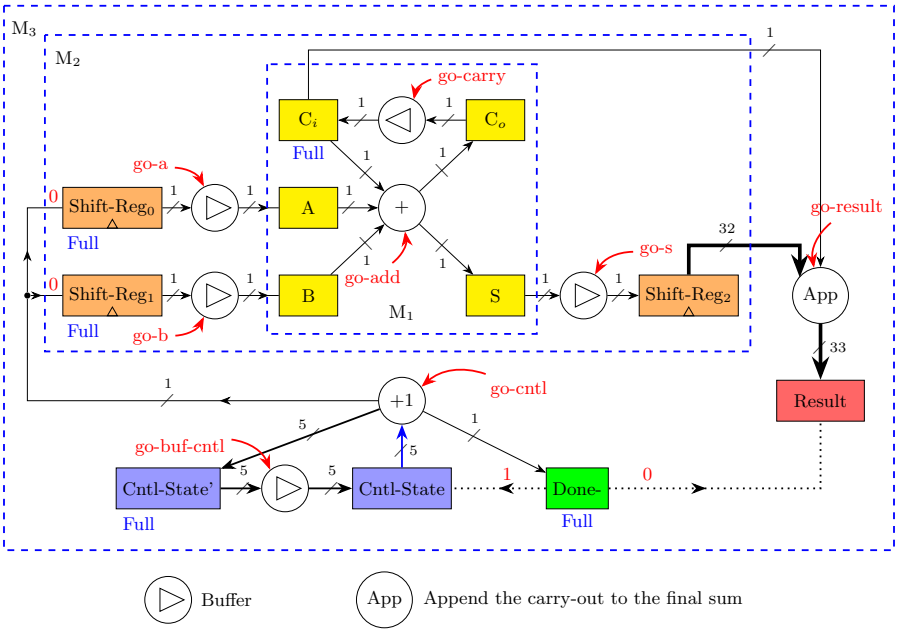


Fig. 6. Data flow of a 32-bit self-timed serial adder, M_3. Circles represent joints, rectangles represent links.

- *Cntl-State* and *Cntl-State'* are 5-bit registers that hold the current and next control states of the serial adder, respectively. The current control state acts as a counter that counts the number of times the bit addition has been executed. Since the adder performs 32-bit additions, the control states are 5 bits long.
- The output of the *Done-* latch indicates whether the circuit will write the final 33-bit result into the *Result* register when the corresponding communication is ready, or the circuit keeps updating the current control state *Cntl-State*.
- Latches A and B contain two 1-bit operands for the full-adder; latch C_i contains the carry-in. The 1-bit sum and the carry-out produced from the full-adder are stored in latches S and C_o, respectively.

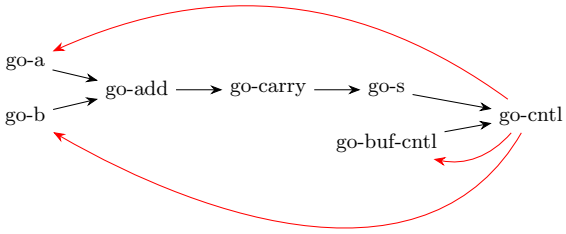
We prove that the self-timed serial adder indeed performs the addition under an appropriate initial condition. Initially, *Shift-Reg*$_0$, *Shift-Reg*$_1$, C_i, *Cntl-State'*, and *Done-* are full; other state-holding elements are empty. The initial values stored in *Shift-Reg*$_0$, *Shift-Reg*$_1$, and C_i represent two 32-bit input operands and the carry-in, respectively. The initial value of *Cntl-State'* is the zero vector, and the initial value of *Done-* is high (or 1). We prove that *Result* eventually becomes

full and its value at that point is the sum of the two 32-bit input operands and the carry-in.

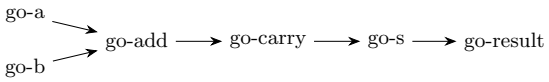
Our approach applies the hierarchical verification method as described in Section 3 to verifying the correctness of the self-timed serial adder. Specifically, we first construct module M_1 that performs bit additions using a 1-bit full-adder (the innermost dashed-line box in Figure 6). We place a constraint when constructing M_1 so that its *ready-out* signal is active (i.e., ready to send data) if the condition $\big(full(S) \wedge full(C_i)\big)$ is satisfied, and its *ready-in-* signal is active (i.e., ready to receive data) if the condition $\big((empty(A) \vee empty(B)) \wedge full(C_i)\big)$ is satisfied. This constraint guarantees that M_1 is ready to communicate with other modules only when it is quiescent. For example, consider the scenario when A and B and C_i are empty, S and C_o are full. Since the *ready-out* condition for M_1 is not satisfied, the joint associated with the *go-s* signal (henceforth, we refer to this joint simply as *go-s*) is not ready to act even if *Shift-Reg*$_2$ is empty. Likewise, neither *go-a* nor *go-b* is ready to act even if *Shift-Reg*$_0$ and *Shift-Reg*$_1$ are full, since the *ready-in-* condition for M_1 is not satisfied. Note that M_1 is still active in this case; it is not quiescent because *go-carry* is now ready to act.

After constructing module M_1, we prove an `se` value lemma and a `de` state lemma for this module. We then move on to construct module M_2 that performs serial additions without control states (the middle dashed-line box in Figure 6). We also place a constraint on M_2's design to guarantee that M_2 is ready to communicate with other modules only when it is quiescent: its *ready-out* signal is active when $full($*Shift-Reg*$_2)$ is satisfied, and its *ready-in-* signal is active when $\big(empty($*Shift-Reg*$_0) \wedge empty($*Shift-Reg*$_1) \wedge full($*Shift-Reg*$_2)\big)$ is satisfied. Since M_2 contains M_1 as a submodule, the two lemmas we already proved for M_1 are used in proving the value and state lemmas for M_2, without knowing any further details about M_1. These two lemmas about M_2 are then used in proving the value and state lemmas for circuit M_3, i.e., the serial adder with control states.

A key step in our verification of the self-timed serial adder is to establish the loop invariant of this circuit model via induction. Given the initial state of the circuit as mentioned earlier, we prove that *the full/empty state of every link in this circuit is preserved after each iteration of the circuit execution*, as long as the value of *Done-* before each iteration is 1. Each iteration performs one bit-addition and the orders of operations to be executed in one iteration are displayed by the dependency paths of the *go* signals in Figure 7. Each relation $go_i \rightarrow go_j$ shown in this figure indicates that go_j will not be ready if go_i is not executed. For instance, the two arrows from *go-a* and *go-b* to *go-add* indicate that *go-add* is ready only if *go-a* and *go-b* were executed. At the initial state, only *go-a*, *go-b*, and *go-buf-cntl* are ready to act: *go-a* is ready because *Shift-Reg*$_0$ is full, A is empty, and M_1 is ready to receive data (i.e., the *ready-in-* condition for M_1 is satisfied); *go-b* is ready because *Shift-Reg*$_1$ is full, B is empty, and M_1 is ready to receive data; *go-buf-cntl* is ready because *Cntl-State'* and *Done-* are full, *Cntl-State* is empty, and the value of *Done-* is 1. Each iteration except the last is finished when *go-cntl* is executed (Figure 7(a)). The last iteration (the value of *Done-* at the beginning of this iteration is 0) is finished when *go-result* is executed



(a) When the value of *Done-* is 1.



(b) When the value of *Done-* is 0.

Fig. 7. Dependency paths of the *go* signals displayed in Figure 6

(Figure 7(b)). Our correctness theorems include the interleaving specification hypotheses, which consider all possible interleavings of the *go* signals' values conforming to the single-step-progress requirement as well as the dependency paths in Figure 7. It is easy to check manually that these dependency paths cover all possible execution paths of M_3. We plan to formalize this claim in the future.

The following two theorems, both proved with ACL2, state correctness of our self-timed circuit. Theorem 1 (partial correctness) states that given the initial state *st* of the serial adder as mentioned previously (Hypothesis 2), and the input sequence *input_seq* satisfying the interleavings of the *go* signals specified by the dependency paths shown in Figure 7 (Hypothesis 4, note that the *go* signals are part of the inputs); if the *Result* register becomes full (Hypothesis 6) after running the serial adder n `de` steps from the initial state (Hypothesis 5), then the value of the *Result* register at that point is the sum of the two 32-bit input operands and the carry-in initially stored in two shift registers *Shift-Reg*$_0$ and *Shift-Reg*$_1$, and latch C_i, respectively. Theorem 2 (termination) states that the *Result* register will become full if n is large enough (Hypothesis 6'; Hypotheses 1-5 are the same as in Theorem 1).

In our verification effort, we automate the verification process for the serial adder by defining macros that help prove automatically the base case of the loop invariant for all possible execution paths. We prove about 230 supporting lemmas (not including other supporting lemmas imported from libraries from the ACL2 Community Books) that help discharge automatically proof obligations required to prove the two main theorems mentioned above. One of the main challenges in verifying the serial adder is to prove the loop invariant by induction: ACL2 fails to discover automatically the correct induction scheme to prove this loop invariant. We have to provide an induction scheme to ACL2. In spite of that,

our induction scheme is general enough to be applied to proving loop invariants of other self-timed circuits. The verification time of the 32-bit self-timed serial adder is about 80 seconds on a 2.9 GHz Intel Core i7 processor with 4MB L3 cache and 8GB memory. Since the loop invariant of the serial adder is established by induction, our proof technique can scale to any size of the adder.

Theorem 1 (Partial correctness).

$$
\begin{aligned}
& async\_serial\_adder(netlist) \wedge && (1)\\
& init\_state(st) \wedge && (2)\\
& (operand\_size = 32) \wedge && (3)\\
& interleavings\_spec(input\_seq, operand\_size) \wedge && (4)\\
& (st' = run(netlist, input\_seq, st, n)) \wedge && (5)\\
& \textbf{\textit{full(result\_status(st'))}} && (6)\\
\Rightarrow\ & (result\_value(st') = shift\_reg\_0\_value(st) + \\
& \qquad\qquad shift\_reg\_1\_value(st) + \\
& \qquad\qquad ci\_value(st))
\end{aligned}
$$

Theorem 2 (Termination).

$$
\begin{aligned}
& async\_serial\_adder(netlist) \wedge && (1)\\
& init\_state(st) \wedge && (2)\\
& (operand\_size = 32) \wedge && (3)\\
& interleavings\_spec(input\_seq, operand\_size) \wedge && (4)\\
& (st' = run(netlist, input\_seq, st, n)) \wedge && (5)\\
& \textbf{\textit{(n} $\geq$ \textit{num\_steps(input\_seq, operand\_size))}} && (6')\\
\Rightarrow\ & full(result\_status(st'))
\end{aligned}
$$

6 Future Work

In the future, we plan to prove the partial correctness of the self-timed serial adder without specifying the interleavings of the *go* signals' values. In other words, we aim to remove Hypothesis 4 from Theorem 1. A possible approach is that we can use Theorem 1 as a supporting lemma and also prove that for any interleaving i, there always exists a specified interleaving i' such that the orderings of operations when executing the circuit under i and i' are identical. From these two lemmas, we can derive the desired partial correctness theorem that does not specify the interleavings.

For the termination theorem (Theorem 2), simply removing Hypothesis 4 will make the theorem invalid. We need to add a constraint guaranteeing that delays are bounded in order to prove Theorem 2 without having Hypothesis 4.

We also plan to investigate a notion of modules with joints at the interfaces instead of links, where two modules are connected by one or more external links.

Another possibility for future work is to develop a better compositional reasoning method that improves scalability when the number of interleavings increases. The high-level idea is to verify the correctness of a larger module by composing verified submodules without delving into details about the submodules as well as the interleavings of their internal operations.

Our existing design restrictions may reduce the performance of self-timed implementations. Our purpose of imposing these restrictions in the design stage is to establish loop invariants for iterative circuits. For circuits that have no feedback loops, we are developing a method for verifying these systems without imposing the aforementioned restrictions.

7 Conclusion

This paper presents a framework for modeling and verifying self-timed circuits using the DE system. We model a self-timed system as a network of links communicating with each other locally via handshake components, which are called joints, using the link-joint model. To our knowledge, this is the first time self-timed circuits are modeled using the link-joint model in a theorem-proving system. We also model the non-determinism of event-ordering in self-timed circuits by associating each joint with an external *go* signal. In addition, presenting self-timed modules as complex links is also new in our paper. Another contribution of our work is our verification procedure to self-timed circuits as described. We show that the existing DE system already proven to be successful for synchronous circuits is adaptable for handling self-timed systems by reasoning with *go* signals as well as state-holding elements that have their own gating. Our verification approach is able to establish loop invariants using induction when the circuit behavior obeys the design restrictions we propose. Hierarchical verification is essential in our verification method and critical to circuit verification at large scale.

Acknowledgements

The authors would like to thank Matt Kaufmann for his encouragement, great discussions and feedback. We also thank Anna Slobodova for her useful comments and corrections on this paper. This material is based upon work supported by DARPA under Contract No. FA8650-17-1-7704.

References

1. C. Chau. Extended Abstract: Formal Specification and Verification of the FM9001 Microprocessor Using the DE System. In *Proc of the Fourteenth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, pages 112–114, 2017.
2. E. Clarke and B. Mishra. Automatic Verification of Asynchronous Circuits. In *Proc of the Workshop on Logic of Programs*, pages 101–115, 1983.

3. W. Hunt. The DE Language. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 10, pages 151–166. Springer US, 2000.
4. W. Hunt and E. Reeber. Applications of the DE2 Language. In *Proc of the Sixth International Workshop on Designing Correct Circuits (DCC-2006)*, 2006.
5. W. Hunt and S. Swords. Use of the E Language. In *Hardware Design and Functional Languages*, 2009.
6. P. Joshi, P. Beerel, M. Roncken, and I. Sutherland. Timing Verification of GasP Asynchronous Circuits: Predicted Delay Variations Observed by Experiment. In D. Dams, U. Hannemann, and M. Steffen, editors, *Lecture Notes in Computer Science*, chapter 17, pages 260–276. Springer Berlin Heidelberg, 2010.
7. M. Kaufmann and J. Moore. ACL2 Home Page. `http://www.cs.utexas.edu/users/moore/acl2/`, 2017.
8. H. Kim, P. Beerel, and K. Stevens. Relative Timing Based Verification of Timed Circuits and Systems. In *Proc of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC-2002)*, pages 115–124, 2002.
9. A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant. Checking Delay-Insensitivity: 10^4 Gates and Beyond. In *Proc of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC-2002)*, pages 149–157, 2002.
10. C. Myers. *Asynchronous Circuit Design*. Wiley, 2001.
11. H. Park, A. He, M. Roncken, X. Song, and I. Sutherland. Modular Timing Constraints for Delay-Insensitive Systems. *Journal of Computer Science and Technology*, 31(1):77–106, 2016.
12. M. Roncken, C. Cowan, B. Massey, S. Gilla, H. Park, R. Daasch, A. He, Y. Hei, W. Hunt, X. Song, and I. Sutherland. Beyond Carrying Coal To Newcastle: Dual Citizen Circuits. In A. Mokhov, editor, *This Asynchronous World Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pages 241–261. Newcastle University, 2016.
13. M. Roncken, S. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland. Naturalized Communication and Testing. In *Proc of the Twenty First IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2015)*, pages 77–84, 2015.
14. A. Slobodova, J. Davis, S. Swords, and W. Hunt. A Flexible Formal Verification Framework for Industrial Scale Validation. In *Proc of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE-2011)*, pages 89–97, 2011.
15. J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Springer US, 2001.
16. S. Srinivasan and R. Katti. Desynchronization: Design for Verification. In *Proc of the Eleventh International Conference on Formal Methods in Computer-Aided Design (FMCAD-2011)*, pages 215–222, 2011.
17. F. Verbeek and J. Schmaltz. Verification of Building Blocks for Asynchronous Circuits. In *Proc of the Eleventh International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2013)*, pages 70–84, 2013.
18. V. Wijayasekara, S. Srinivasan, and S. Smith. Equivalence Verification for NULL Convention Logic (NCL) Circuits. In *Proc of the Thirty Second IEEE International Conference on Computer Design (ICCD-2014)*, pages 195–201, 2014.