2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)

A Hierarchical Approach to Self-Timed Circuit Verification

Cuong Chau, Warren A. Hunt Jr., Matt Kaufmann
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
Email: {ckcuong,hunt,kaufmann}@cs.utexas.edu

Marly Roncken, Ivan Sutherland
Maseeh College of Engineering and Computer Science
Portland State University
Portland, OR, USA
Email: mroncken@pdx.edu, ivans@cecs.pdx.edu

***Abstract*—Self-timed circuits can be modeled in a link-joint style using a formally defined hardware description language. It has previously been shown how functional properties of these models can be formally verified with the ACL2 theorem prover using a scalable, hierarchical method. Here we extend that method to parameterized circuit families that may have loops and non-deterministic outputs. We illustrate this extension with iterative self-timed circuits that calculate the greatest common divisor of two natural numbers, with circuits that perform arbitrated merges non-deterministically, and with circuits that combine both of these.**

I. Introduction

Self-timed circuits and systems proceed at their own rate without reference to a global clock. We use ACL2 [1], [2], [3] to model self-timed circuits at the gate level and to verify that those models exhibit specified functional properties. We build on our previous work [4] that demonstrated a hierarchical approach to support efficient, scalable proof, including support for substitution of functionally equivalent submodules without the need to rework proofs. In this paper we extend that methodology to circuits that may be parameterized by data width, contain loops, and/or provide non-deterministic outputs. We illustrate this extension by modeling and verifying functional correctness of two families of self-timed circuits parameterized by data width: one that computes the *greatest common divisor* (GCD) and one that performs *arbitrated merges*.

We have chosen the *link-joint model* as the basis for our ACL2 formalization of self-timed circuits and systems. That model, originally proposed by Roncken et al. [5], [6], is a universal generalization for a number of self-timed circuit models (e.g., Click [7], Mousetrap [8], Micropipeline [9], and GasP [10]). The link-joint model of self-timing manages communication locally via links on an individual joint-to-joint basis. In circuit implementation terms, this means that instead of using a global-clock signal to indicate when all clocked storage elements should accept new data, self-timed storage elements (links) accept input only when they are ready and the input data are valid — and when a link accepts new input, it signals to its datum provider that the provider may proceed to calculate its next output value(s).

To capture the link-joint computational model, we reuse the DE hierarchical circuit verification approach [11]. Originally, this automated approach was used to verify the FM9001 microprocessor design [12], but we have generalized the semantics of this *hardware description language* (HDL) based circuit specification and verification approach to allow the analysis of self-timed circuits, where their implementations proceed at their own rate. We dispense with a universal clock to accommodate self-timed circuit models. We include a new DE-language link-control primitive that provides the control necessary to signal the storage in each link to accept data while also providing acknowledgment signaling that links have accepted their input data. Using this new primitive, we then abstract away implementation details of link and joint control circuitry. Note, a joint delivers data to output links only when the joint has a valid *go* signal. We use such *go* signals to model interleavings of circuit operations, and we prove the correctness of all self-timed circuit models without assumptions about the arrival time of *go* signals. These signals can also provide post-manufacture test control.

Unlike many efforts for validating timing properties of self-timed systems, we are interested in verifying functional properties. Specifically, we verify the functional correctness of self-timed systems in terms of relationships between their input and output sequences. Though correctness of the lower-level circuit implementations may depend on circuit-level timing constraints, we assume that such timing proofs can be provided separately, as suggested by Park et al. [13].

This paper makes the following contributions:

- extend our previous framework [4] to model and verify circuit generators with parameterized data sizes;
- demonstrate that the data-loop-free verification framework proposed by our previous work is applicable to circuits with loops as well. In particular, we show that, by applying our previous framework, we are able to specify and verify an iterative self-timed circuit model that computes the GCD of two natural numbers. Our modeling avoids imposing communication restrictions introduced by Chau et al. [14];
- formalize an arbitrated merge joint that provides mutually exclusive access to its output link from its two input links; and
- develop strategies for verifying the functional correctness of self-timed circuits performing arbitrated merges. This work includes library development for reasoning about

the membership relation and the interleaving operation.

The rest of the paper is organized as follows. Section II overviews some related work. An overview of the DE system is given in Section III. Section IV describes our methodology for modeling and verifying self-timed circuits. Sections V and VI demonstrate our methodology by describing our modeling and verification of a GCD circuit synthesizer and circuits involving arbitrated merges, respectively. Concluding remarks and future work appear in Section VII.

II. Related Work

Most verification efforts that use formal techniques for analyzing self-timed circuit implementations concern circuit-level timing properties. Timing verification has been investigated by several groups [15], [16], [17], [18], [13]. For instance, Park et al. [13] developed the ARCtimer framework for modeling, generating, and verifying timing constraints on individual handshake components. ARCtimer uses the NuSMV model checker for its analysis. The authors' goal was to ensure that the network of logic gates and wires, along with their associated delays, meets the component's protocol requirements. In contrast, our goal concerns proving that a self-timed circuit or system meets its functional specification, while ignoring circuit-level timing constraints that can be investigated by tools like ARCtimer.

Dill [19] developed a trace theory for hierarchical verification of communication sequences in speed-independent circuits. The author focused only on control circuits, while data circuits were not involved. His method checks circuit properties by simply searching through the state-transition graph that models the circuit behavior. Although this approach is automatic, it explicitly represents and stores all possible states. This is quite inefficient when dealing with circuits consisting of a large number of states. In addition, while the author proposed two theories for modeling and checking safety and liveness properties of speed-independent circuits, only the theory for dealing with safety properties was implemented.

The use of hierarchical verification methods in self-timed circuit contexts has also been explored by Clarke and Mishra [20], in their attempt to verify safety and liveness circuit properties automatically. Their analysis approach is based on model checking, and they investigated the correctness of a self-timed FIFO queue element. Their approach assumes a unit delay for each gate in a self-timed circuit, where our approach avoids imposing any restrictions on gate delays.

Previous applications of ACL2 to asynchronous circuit designs have focused on properties other than their functional correctness. Verbeek and Schmaltz [21] have formalized and verified *blocking* (failing to transmit data) and *idle* (failing to receive data) conditions about delay-insensitive primitives from the Click circuit library. By using ACL2, these conditions were translated into SAT/SMT instances to confirm deadlock freedom in the self-timed circuits investigated. Peng et al. [22] presented a framework for detecting glitches that occur in synthesized clock-domain-crossing netlists but are not apparent in the original RTL specifications. The authors' approach integrates ACL2 with a SAT solver for verifying, in synthesized netlists, the glitch-free property of each state-bit associated with a corresponding flip-flop output. They demonstrated their tool on commercial designs from Oracle Microelectronics.

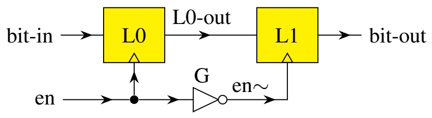


Fig. 1. A flip-flop composed of two latches

We previously developed a hierarchical reasoning method that avoids exploring interleavings within verified submodules [4]. We demonstrated that method on data-loop-free self-timed circuits with fixed data widths. We now extend that framework to circuits with loops and with parameterized data widths. We also develop strategies for reasoning about circuits with arbitrated merges that produce non-deterministic outputs.

III. Overview of the DE System

DE is a hierarchical circuit description language defined in the ACL2 logic. The DE system includes an ACL2 predicate that recognizes a syntactically well-formed netlist; this predicate enforces syntactic requirements on naming, arity, occurrence structure, and signal connectivity. The semantics of the DE language is given by a simulator whose *se* function computes the outputs and whose *de* function computes the next state for a module from the module's current inputs and current state. The *de* simulation function operates in two passes: it first propagates values from primary inputs and internal states throughout the netlist, calculating values for every internal wire. Once the values on all wires are known, it produces the outputs by accessing the appropriate wire values. To produce the next state, *de* makes a second pass over the entire netlist propagating previously-calculated wire values into storage elements.

A *well-formed DE netlist* is an ordered list of modules, where each module may include references to previously defined modules or to DE primitives. Each module definition consists of five ordered entries: a unique module name, input names, output names, internal-state names, and a list of occurrences that references previously defined submodules or DE primitives. Each occurrence in a module consists of four ordered entries: a module-unique occurrence name, a list of output names, a reference to a DE primitive or a submodule, and a list of input names. Below is a DE netlist for a flip-flop circuit that is built from two latches `L0` and `L1`, using a simplified version of the DE latch primitive and the DE `b-not` primitive. Figure 1 offers a schematic diagram of this circuit.

```
(defconst *netlist*
  '((flip-flop    ;; Module's name
     (en bit-in)  ;; Inputs
     (bit-out)    ;; Outputs
     (L0 L1)      ;; Internal states
     ;; Occurrences
     ((L0 (L0-out)  latch (en bit-in))
      (G  (en~)     b-not (en))
      (L1 (bit-out) latch (en~ L0-out))))))
```

Below is an example of evaluating the output and next state for the $flip\text{-}flop$ module using the se and de functions, respectively. These two functions require the same four ordered arguments: the name of the module to evaluate, its input values, its current-state value, and a well-formed DE netlist containing the definition of the module and submodules to be simulated. The semantics of `latch` is given as follows: when the *enable* signal is on, `latch` will propagate the input value to the output and update its internal state with the input value; otherwise it will report the current state to the output and its state remains unchanged. For the DE system, we use the ACL2 constants `t` and `nil` to represent Boolean true and false, respectively. The single quotation marks require the evaluator to use the inputs as given, thus the expression `'(nil nil)` provides a list of two Boolean values: false, false. In this example, we instantiate *en* := `nil`, *bit-in* := `nil`, $L0$:= `'(t)`, and $L1$:= `'(nil)`. Note that we use the ACL2 (prefix) syntax to describe the formulas in this section.

```
(se 'flip-flop '(nil nil) '((t) (nil)) *netlist*) =
'(t)
```

```
(de 'flip-flop '(nil nil) '((t) (nil)) *netlist*) =
'((t) (t))
```

The DE system provides a hierarchical approach to analyze DE circuit descriptions. In particular, we prove the following two lemmas in a hierarchical manner for every DE module: a *value lemma* characterizing a module's outputs and a *state lemma* characterizing a module's next state — and for other than the lowest-level modules, these two lemmas are proved by automatic application of the value and state lemmas of submodules, without referencing the internal details of the submodules. A purely combinational module requires only a value lemma. Here are the value and state lemmas for module $flip\text{-}flop$.

```
(se 'flip-flop inputs st *netlist*) =
(flip-flop$outputs inputs st)
```

```
(de 'flip-flop inputs st *netlist*) =
(flip-flop$step inputs st)
```

where the $outputs$ and $step$ functions $flip\text{-}flop\$outputs$ and $flip\text{-}flop\$step$ are defined as symbolic logical expressions characterizing the outputs and next state of $flip\text{-}flop$, respectively.

```
(flip-flop$outputs inputs st) :=
(let ((en      (first inputs))
      (L0.data (first (first st)))
      (L1.data (first (second st))))
  (list (if (not en) L0.data L1.data)))
```

```
(flip-flop$step inputs st) :=
(let ((en      (first inputs))
      (bit-in  (second inputs))
      (L0.data (first (first st)))
      (L1.data (first (second st))))
  (list ;; L0's next state
        (list (if en bit-in L0.data))
        ;; L1's next state
        (list (if (not en) L0.data L1.data))))
```

From the state lemma, we prove the following *multi-step state lemma* by induction,

```
(de-n 'flip-flop inputs-seq st *netlist* n) =
(flip-flop$run inputs-seq st n)
```

where $de\text{-}n$ and $flip\text{-}flop\$run$ are defined recursively in terms of de and $flip\text{-}flop\$step$, respectively. For instance, below is the definition of $flip\text{-}flop\$run$.

```
(flip-flop$run inputs-seq st n) :=
(if (<= n 0)
  st
  (flip-flop$run
   (rest inputs-seq)
   (flip-flop$step (first inputs-seq) st)
   (- n 1)))
```

Once the state lemma and the multi-step state lemma are proved, we use only the $step$ and run functions in reasoning about the module behavior.

IV. Methodology

We use the ACL2 logic to model self-timed systems and circuits based on the link-joint model. Implementations are written in the DE language. We use the ACL2 theorem prover to certify properties of these implementation models.

A. Modeling

We follow the modeling approach proposed by our previous work [4]. Here we briefly describe the link-joint model that we use to represent self-timed circuits with the DE language. Links are communication channels in which data are stored along with a validity signal. Joints perform data operations and implement flow control. Joints are the meeting points that coordinate links and share link data. A self-timed system can be viewed as a directed graph with links as edges and joints as nodes. The control logic of a joint is an AND function of the conditions necessary for it to act. A joint can have multiple such AND functions to guard different actions, which are usually mutually exclusive. To enable a joint action, all input and output links of that action must be full and empty, respectively, as illustrated by the AND gate in Figure 2. Enabled joints (that is, when at least one action is enabled) may fire in any order due to arbitrary delays in wires and gates. We model this 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. In case a joint has multiple such AND functions, they may share the same *go* signal as long as at most one function can fire at a time. When a joint acts, the following three tasks will execute in parallel:

- using data from full input links, compute results to transfer to empty output links;
- *fill* (possibly a subset of) the empty output links, leaving them full; and
- *drain* (possibly a subset of) the full input links, leaving them empty.

Our DE description of a self-timed module allows links and joints to appear in any order in the module's occurrence

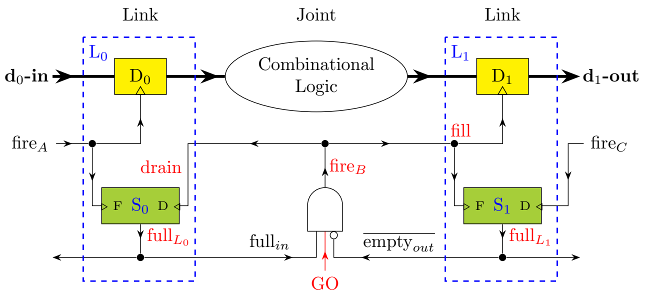


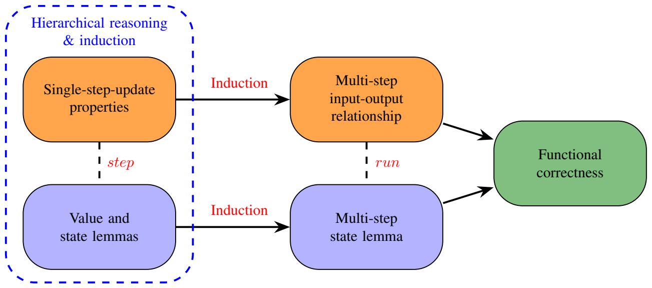
Fig. 2. A diagram of a link-joint circuit is shown. It has two links, $L0$ and $L1$, and three joints A, B, and C. Only joint B is shown in its entirety. The upper and lower boxes in each link represent link data and link full/empty status, respectively.

list, except that each link must be declared before its input and output joints so that when the module is being evaluated, the se function called in the first pass will extract the links' full/empty states and data and provide these values as inputs for the corresponding joints; the de function will make the second pass to update the link's full/empty states and data using the joints' output values calculated from the first pass. Below is a DE description of the self-timed module shown in Figure 2, where D_0 and D_1 are latches. The Combinational Logic oval represents the data computation of the joint which is a storage-free amplifier (`buffer` below) in this DE description. The diagram shows four state-holding devices that work together in pairs: data latch D_0 (`d0` below) and its associated full/empty flag S_0 (`s0` below), and data latch D_1 (`d1` below) and its associated full/empty flag S_1 (`s1` below).

```
'(link-joint                         ;; Module's name
  (fireA fireC d0-in go)             ;; Inputs
  (s0-status s1-status d1-out)       ;; Outputs
  (s0 d0 s1 d1)                      ;; Internal states
  ;; Occurrences
  ( ;; Link L0
   (s0 (s0-status) link-cntl (fireA fireB))
   (d0 (d0-out) latch (fireA d0-in))
   ;; Link L1
   (s1 (s1-status) link-cntl (fireB fireC))
   (d1 (d1-out) latch (fireB d1-in))
   ;; Joint B
   (jb-cntl (fireB)
            joint-cntl
            (s0-status s1-status go))
   (jb-op (d1-in) buffer (d0-out))))
```

As an example, we use the list `'((t) (nil) (nil) (t))` to represent the state of the above module where link $L0$ is full and its data value is `nil`, and link $L1$ is empty and its data value is `t`. Note that when a link is empty, its data are invalid.

Self-timed modules can be abstracted as "complex" links or "complex" joints [4]. A module is a *complex link* if only links appear at its input and output ports. Similarly, a module is a *complex joint* if only joints appear at its input and output ports. A complex link is limited to a single input and single output link to remain a point-to-point connection, while a complex joint can have many inputs and many outputs. It is typical that self-timed modules receive and send data via different links, using separate input and output communication signals.



Hierarchical reasoning & induction
Single-step-update properties
step
Value and state lemmas
Induction
Induction
Multi-step input-output relationship
run
Multi-step state lemma
Functional correctness

Fig. 3. Verification flow

B. Verification

Most existing work in formally verifying self-timed systems has concerned only flow control. Those efforts have mainly explored strategies for validating timing and communication properties. Our approach, on the other hand, considers self-timed circuits involving both data operations and flow control. We verify functional properties of a self-timed system as a whole. Our approach supports scalability via *hierarchical reasoning* and *induction*. We extend the methodology developed by our previous work [4] for verifying the functional correctness of self-timed circuits and systems in terms of the relationships between their input and output sequences. One of our extensions develops a proof strategy for reasoning about recursively defined circuit generators, which emerge from the context of parameterized data widths. In particular, our approach applies induction in proving the value and state lemmas for those parameterized circuit generators. Another extension in our reasoning method also applies induction to establish the *single-step-update* properties for self-timed circuits containing loops. This appears in the context of reasoning about the algorithmic specification functions, which are defined recursively, for those circuits. Figure 3 depicts our verification flow for a self-timed module. The value and state lemmas and the multi-step state lemma were already discussed in Section III. The single-step-update properties will be explained in more detail later in this section. The next two sections will discuss the multi-step input-output relationship through several self-timed circuit models. The functional correctness theorem is a direct corollary of the multi-step input-output relationship that is stated in terms of the $de\text{-}n$ function, while that relationship is formalized in terms of the run function.

Although we aim to verify the multi-step input-output relationship for each self-timed module, our hierarchical reasoning is indeed applied only at one-step updates. Once the one-step update on the output sequence is established, the multi-step input-output relationship can then be proved by induction. In order to specify the input-output relationship at one step, we introduce a set of *extraction functions* for each sequential module. An extraction function $extract(st)$ returns a sequence of values computed from valid data residing in state st. We use such a function to abstract away state transitions internal to its corresponding module; it will return the same sequence for any two input states if one of those states can reach the other through merely internal transitions. Applying $extract$ to the $step$ function, i.e. $extract(step(inputs, st))$, will compute the one-step update on the abstracted state given the current inputs $inputs$ and current state st. Recall that $step$ symbolically specifies the module's next state in one (de) step (see $flip\text{-}flop\$step$ in Section III for an example). To establish the multi-step input-output relationship by induction, we prove the following key lemma, which is called the single-step-update property [4],

$$extract(step(inputs, st)) = extracted\text{-}step(inputs, st) \quad (1)$$

where $extracted\text{-}step$ is the specification for the one-step update on the abstracted state. An important property of $extracted\text{-}step$ is that its definition avoids exploring the module's internal operations and their possible interleavings — $extracted\text{-}step(inputs, st)$ depends only on the values of $extract(st)$ and the communication signals at the module's input and output ports. In Sections V and VI, where we describe our experiments, we will discuss the use of $extract$ and $extracted\text{-}step$ in more detail [1]. The examples in these two sections demonstrate the scalability of our approach. It is critical that $step$ and $extract$ are defined hierarchically so that the single-step-update property (1) can be proved hierarchically. A naive approach that expands the definitions of the $step$ and $extract$ functions of submodules when proving (1) may lead to a computational explosion. Our proofs cover all possible interleavings of circuit operations by considering all combinations of *go* signals' values. Much of our proof process is stylized. We automate the proof process by introducing proof idioms via macros and by developing lemma libraries.

V. A GCD Circuit Model

In this section we demonstrate our methodology by modeling and verifying the functional correctness of an iterative self-timed circuit synthesizer that computes the greatest-common-divisor (GCD) of two natural numbers. In particular, we specify and verify the self-timed circuit model, called gcd, displayed in Figure 4. Notice that this is really a family of circuit models, parameterized by data width n for inputs a and b. This circuit family computes the GCD of two input operands as described by the following algorithm.

```
$gcd\text{-}alg(a, b) :=$
  if $(a = 0)$ then $b$
  else if $(b = 0)$ then $a$
  else if $(a = b)$ then $a$
  else if $(a < b)$ then $gcd\text{-}alg(b - a, a)$
  else $gcd\text{-}alg(a - b, b)$
```

$gcd\text{-}alg$ is formalized in ACL2 to serve as the functional specification for gcd. By proving the following properties (where d is any common divisor of a and b), we show that

[1]Our proofs for the circuits described later in Sections V and VI are available at https://github.com/acl2/acl2/tree/master/books/projects/async.

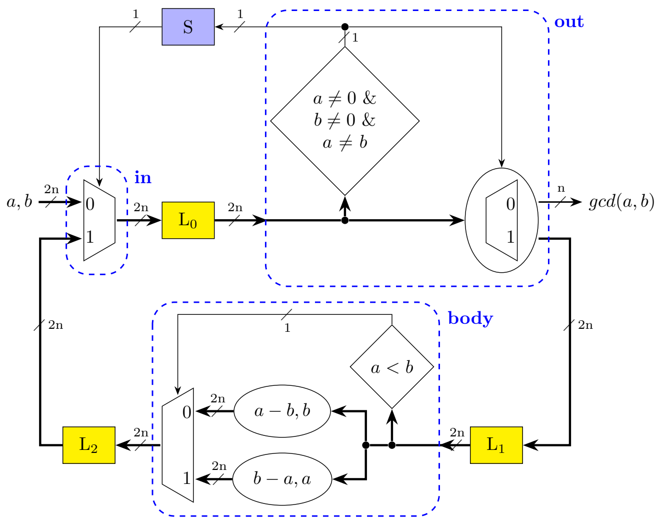


Fig. 4. Data flow of module *gcd*: a circuit that computes the GCD of two natural numbers. n is the number of bits in each operand. Solid rectangles represent links. Dashed, rounded-corner rectangles identify joints.

$gcd\text{-}alg$ correctly computes the GCD of two natural numbers a and b.

$$a \ \mathbf{mod}\ gcd\text{-}alg(a, b) = 0,$$
$$b \ \mathbf{mod}\ gcd\text{-}alg(a, b) = 0,$$
$$\big((a > 0) \lor (b > 0)\big) \land (a \ \mathbf{mod}\ d = 0) \land (b \ \mathbf{mod}\ d = 0) \Rightarrow 0 < gcd\text{-}alg(a, b) \land d \leq gcd\text{-}alg(a, b)$$

Our DE-based gcd circuit model, based on Sparso's design [23], consists of four links and three joints. Link S holds a binary value that acts as the mux *select* signal for joint **in**. The two operands stored in link L_0 will be passed to joint **out** to check if they are non-zero and do not have the same value — we call this condition the GCD condition. If the GCD condition is true, the two operands will enter the body of the loop and be stored in link L_1. Otherwise, the circuit will report the result and be ready to accept new inputs after the result has been accepted (because in this case 0 will be stored in link S). In the case the two operands have entered the loop and been stored in link L_1, the greater operand will be updated as described in the $gcd\text{-}alg$ algorithm. The updated operand and the remaining operand will then be stored in link L_2.

When joint **in** fires, it drains either link L_2 or the link providing the external input [2], according to the value of the mux select signal. Thus, joint **in** has two mutually exclusive actions. In a similar manner, joint **out** also has two mutually exclusive actions: when it fires, it fills link S and either link L_1 or the link accepting the final output [3], depending on the value

[2]The link that provides the external input is represented only by the small arrow at the left of the drawing labeled a, b in Figure 4.

[3]The link that accepts the final output is represented only by the small arrow at the right of the drawing labeled $gcd(a, b)$ in Figure 4.

of the demux select signal. Joint **body** updates the operands as described below.

```
$gcd\text{-}body\$op(a, b) :=$
  **if** $(a < b)$ **then** $swap(a, b)$
  **return** $(a - b, b)$
```

We model the three joints **in**, **out**, and **body** as DE modules and prove the value lemma for each of them. There are no state lemmas for these joints because they are purely combinational modules.

Let *in-act* denote the *fire* signal from the AND gate (not illustrated) associated with the input port (labeled a, b) in the control logic of joint **in**. Module gcd accepts a new input data item each time the *in-act* signal fires. We define *in-seq*, the *accepted input sequence*, as the sequence of input data items that have passed joint **in**. Similarly, let *out-act* denote the *fire* signal from the AND gate (not illustrated) associated with the output port (labeled $gcd(a, b)$) in the control logic of joint **out**. We define *out-seq*, the *valid output sequence*, as the sequence of data items that have passed through joint **out** while *out-act* fires. The functionality of gcd over a data sequence of natural-number pairs is specified recursively as follows,

```
$gcd\$op\text{-}map(seq) :=$
  **if** $(seq = NULL)$
  **then** $[]$  // an empty list
  **else**
    **let** $in := first(seq)$
    **return** $[gcd\text{-}alg(in.a, in.b)]$ ++ $gcd\$op\text{-}map(rest(seq))$
```

where ++ indicates concatenation. We then define an extraction function that extracts the future output sequence from the current state, st, as described below,

$$gcd\$extract(st) := gcd\$op\text{-}map(data([st.L_1, st.L_2, st.L_0]))$$

where the projection function $data(l)$ returns the list generated by mapping over the links in l, collecting the data item of each full link and ignoring each empty link. For example, suppose links L_0 and L_1 are full, and link L_2 is empty in state st; then $data([st.L_1, st.L_2, st.L_0])$ will return $[d_1, d_0]$, where d_i is the data item of link L_i. Note that the order of links L_i to be extracted does not affect our correctness proof for gcd, because we impose a condition that there be at most one valid data item, not including S, in the system at any time. In particular, we require the following condition, which we prove is an invariant,

```
$gcd\$inv(st) :=$
  **if** $full(st.S.s) \wedge (st.S.d = 0)$
  **then** $len(gcd\$extract(st)) = 0$
  **else** $len(gcd\$extract(st)) = 1$
```

where $len(l)$ counts the number of elements in list l. This invariant is necessary to maintain the first-in-first-out relationship between input and output sequences. We verify the functional correctness of gcd by formalizing the relationship between its input and output sequences after an n-step execution from its initial state, for any natural number n and any data size. We expect the output sequence is the sequence of GCDs of the natural-number pairs in the input sequence. Our formalization also considers two facts: the initial state may contain some valid data, and there can be some valid data remaining in the final state because the circuit operation over these data has not yet finished. Formally, we establish the following multi-step input-output relationship,

$$\begin{aligned}&gcd\$extract(gcd\$run(inputs\text{-}seq, st, n)) \mathbin{++} out\text{-}seq = \\ &gcd\$op\text{-}map(in\text{-}seq) \mathbin{++} gcd\$extract(st) \qquad (2)\end{aligned}$$

where $gcd\$run$ is recursively defined in terms of $gcd\$step$ (e.g., see $flip\text{-}flop\$run$ in Section III). It trivially follows that $out\text{-}seq = gcd\$op\text{-}map(in\text{-}seq)$ when the initial and final states of gcd contain no valid data. Note that the parameters $inputs$ and $inputs\text{-}seq$ we mention throughout our experiments consist of both input data and input control signals, including *go* signals for every joint. On the other hand, *in-seq* and *out-seq* contain only data and are devoid of control information. Moreover, *in-seq* is the sequence of inputs extracted from $inputs\text{-}seq$ that are accepted by gcd, specifically when: L_0 is empty, S is full with a value of 0, the link providing the input data is full, and the corresponding *go* signal is active. Our proof of (2) uses induction and the following single-step-update property, which is an instance of (1), as a supporting lemma,

$$\begin{aligned}&gcd\$extract(gcd\$step(inputs, st)) = \\ &gcd\$extracted\text{-}step(inputs, st) \qquad (3)\end{aligned}$$

where the extracted next-state function $gcd\$extracted\text{-}step$ extracts the future output sequence from the next state in terms of the $gcd\$extract$ function, as described in Figure 5. $gcd\$extracted\text{-}step$ avoids considering internal operations of gcd; it considers only the values of $gcd\$extract$ and the *in-act* and *out-act* signals at gcd's input and output ports respectively, thus reducing the complexity of extracting valid outputs from gcd's next state to four cases, regardless of how internal operations proceed. Equation (3) holds when $gcd\$inv(st)$ holds. Since we already proved that $gcd\$inv$ is an invariant, our induction proof for (2) still applies as long as the initial state of gcd satisfies $gcd\$inv$.

VI. Arbitrated Merge

The circuit discussed in the previous section has the first-in-first-out input-output property. This implies that the order in the output sequence is deterministic. Now we discuss another well-known type of self-timed circuit that produces non-deterministic output sequences, that is, circuits allocating mutually exclusive access to shared resources. A *mutual-exclusion circuit* or *arbiter* is commonly used in self-timed systems to provide mutually exclusive access to a shared resource on a first-come-first-served (FCFS) basis [24]. Since the arrival times of requests are variable, the grant outcomes are essentially non-deterministic.

$$gcd\$extracted\text{-}step(inputs, st) := \begin{cases} gcd\$extract(st), & \textit{if } in\text{-}act = nil \wedge out\text{-}act = nil \\ [gcd\text{-}alg(inputs.a, inputs.b)] \mathbin{++} gcd\$extract(st), & \textit{if } in\text{-}act = t \wedge out\text{-}act = nil \\ remove\text{-}last(gcd\$extract(st)), & \textit{if } in\text{-}act = nil \wedge out\text{-}act = t \\ [gcd\text{-}alg(inputs.a, inputs.b)] \mathbin{++} remove\text{-}last(gcd\$extract(st)), & \textit{otherwise} \end{cases}$$

where $remove\text{-}last(l)$ returns list l except for its last element.

Fig. 5. Definition of $gcd\$extracted\text{-}step$

A. Arbitrated Merge Joint

We model a simple arbitrated merge joint that *randomly* transfers data from one of the two input links to the output link if both input links are full "nearly" at the same time. When only one input link is currently full, the merge will transfer data from that link to the output link. In either case, the data transfer occurs only when the output link is empty. This implementation of an arbitrated merge joint was described in Roncken et al.'s paper [6]. In our modeling, we use an oracle signal called *select* to perform random selections when necessary. Although quite simple, this arbitrated merge model serves our purpose of illustrating the handling of non-determinism in our verification framework. In the future, we intend to model an arbitrated merge joint with an arbitration mechanism that supports *fairness* [6]: when two requests arrive at nearly the same time, they must be served before the merge serves new requests.

B. Experiments

We use the membership relation ($\in$) and the interleaving operation ($\otimes$) for establishing the multi-step input-output relationships for self-timed circuits performing arbitrated merge operations. For example, the output sequence from an arbitrated merge can be expressed as a member of all possible interleavings of the two input sequences as follows.

$out\text{-}seq \in (in_0\text{-}seq \otimes in_1\text{-}seq)$

The interleaving operation $\otimes$ computes all interleavings of its two input sequences, e.g. [4],

$$[1, 2] \otimes [a, b] = [[1, 2, a, b], [1, a, 2, b], [1, a, b, 2], [a, 1, 2, b], [a, 1, b, 2], [a, b, 1, 2]].$$

Verifying the multi-step input-output relationships for circuits performing arbitrated merges involves developing a library that supports reasoning about the membership relation and the interleaving operation. In this library, referred to as *mem-interl-lib*, we prove lemmas about the preservation of the membership under the concatenation operation with the presence of the interleaving operation. For example,

$$x \in (y \otimes z) \Rightarrow (x \mathbin{++} x1) \in ((y \mathbin{++} x1) \otimes z) \wedge (x \mathbin{++} x1) \in (y \otimes (z \mathbin{++} x1)). \quad (4)$$

We will present more key lemmas from *mem-interl-lib* when we discuss our experiments below. As we did for gcd, our strategy for proving the multi-step relationship is based on single-step-update properties. For circuits involving arbitrated merges, we introduce two extraction functions to extract two valid input streams for each arbitrated merge; and we prove a single-step-update property for each extraction function.

[4]The two input operands of $\otimes$ do not necessary have the same length.

1) Example 1

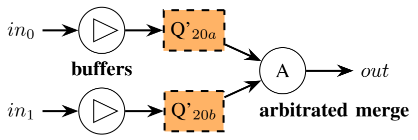


Fig. 6. Data flow of $interl$. Dashed rectangles represent complex links. Circles represent joints.

Our first example considers a circuit that connects two 20-link queues to the two input ports of an arbitrated merge (Figure 6). We model a 20-link queue as a complex link that concatenates two 10-link queues, Q'_{10} [4], which are also complex links, via a storage-free buffer joint. Let $in_0\text{-}seq$ and $in_1\text{-}seq$ represent two accepted input sequences connected to Q'_{20a} and Q'_{20b}, respectively. We prove that for any interleaving x of two data sequences remaining in the final state, the concatenation of x and the output sequence must be a member of $(seq_0 \otimes seq_1)$; where seq_0 is the concatenation of $in_0\text{-}seq$ and the valid data sequence in Q'_{20a} at the initial state, and seq_1 is the concatenation of $in_1\text{-}seq$ and the valid data sequence in Q'_{20b} at the initial state. Formally, we prove the following property,

$$\begin{aligned} &\textbf{let } st_f := interl\$run(inputs\text{-}seq, st, n), \\ &\forall x \in \big(interl\$extract_0(st_f) \otimes interl\$extract_1(st_f)\big). \\ &(x \mathbin{++} out\text{-}seq) \in \Big(\big(in_0\text{-}seq \mathbin{++} interl\$extract_0(st)\big) \otimes \\ &\qquad \big(in_1\text{-}seq \mathbin{++} interl\$extract_1(st)\big)\Big) \quad (5) \end{aligned}$$

where $interl\$extract_0$ and $interl\$extract_1$ extract valid data from Q'_{20a} and Q'_{20b}, respectively. When the initial and final states have no valid data, we obtain the corollary $out\text{-}seq \in (in_0\text{-}seq \otimes in_1\text{-}seq)$. We prove property (5) by induction after proving two single-step-update properties (one for each extraction function) and lemma (4). Note that the single-step-update properties for $interl$ are proved in a hierarchical manner. Specifically, these properties are proved by applying the single-step-update properties of submodules Q'_{20a} and Q'_{20b} accordingly, without exploring the operations internal to these submodules.

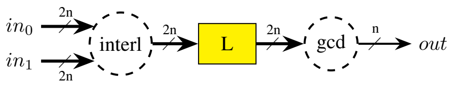


Fig. 7. Data flow of *igcd*. Dashed circles represent complex joints.

2) Example 2

The next example further illustrates hierarchical reasoning: the verification of a self-timed module that contains self-timed submodules with and without arbitration. We model a module, called $igcd$, that connects the output port of $interl$ to the input port of gcd via a link (Figure 7). Notice that each item in each input stream of $igcd$ carries a pair of operands for a GCD operation. We verify the correctness of $igcd$ by proving that this circuit produces a sequence of GCDs over any interleaving of two input sequences. Our approach uses three extraction functions that compute the GCDs of the valid data residing in $interl.Q'_{20a}$, $interl.Q'_{20b}$, and the concatenation of L and gcd, respectively.

$$igcd\$extract_0(st) := gcd\$op\text{-}map(interl\$extract_0(st.interl)),$$
$$igcd\$extract_1(st) := gcd\$op\text{-}map(interl\$extract_1(st.interl)),$$
$$igcd\$extract_2(st) :=$$
$$gcd\$op\text{-}map(data([st.L])) \mathbin{++} gcd\$extract(st.gcd)$$

We formalize the multi-step input-output relationship for $igcd$ in terms of function $prepend\text{-}rec(x, y)$ that prepends each list in x to y. For example,

$$prepend\text{-}rec([[1,2],[3,6,4]],[a,b]) = [[1,2,a,b],[3,6,4,a,b]].$$

Below is the multi-step input-output relationship for $igcd$ that we formalize.

$$\textbf{let } st_f := igcd\$run(inputs\text{-}seq, st, n),$$
$$\forall x \in \big(igcd\$extract_0(st_f) \otimes igcd\$extract_1(st_f)\big).$$
$$(x \mathbin{++} igcd\$extract_2(st_f) \mathbin{++} out\text{-}seq) \in$$
$$prepend\text{-}rec\Big(\big(gcd\$op\text{-}map(in_0\text{-}seq) \mathbin{++} igcd\$extract_0(st)\big) \otimes \big(gcd\$op\text{-}map(in_1\text{-}seq) \mathbin{++} igcd\$extract_1(st)\big), igcd\$extract_2(st)\Big)$$

Our proof applies the following lemma in *mem-interl-lib*.

$$e \in x \Rightarrow (e \mathbin{++} e1) \in prepend\text{-}rec(x, e1)$$

Again, we apply the hierarchical mechanism to prove the single-step-update properties for $igcd$ from the single-step-update properties of submodules $interl$ and gcd accordingly.

3) Example 3

We continue illustrating our hierarchical reasoning method via a circuit model, $comp\text{-}interl$, that composes three instances of $interl$ as displayed in Figure 8. Loosely speaking, we prove that the output sequence from $comp\text{-}interl$ is an interleaving of its four input sequences. That correctness theorem (omitted here) may be stated in terms of the nested interleaving operator, $\otimes_2$, where $x \otimes_2 y$ interleaves each list in x with each list in y. For example,

$$[l_1, l_2, l_3] \otimes_2 [l_4, l_5] = (l_1 \otimes l_4) \mathbin{++} (l_1 \otimes l_5) \mathbin{++} (l_2 \otimes l_4) \mathbin{++} (l_2 \otimes l_5) \mathbin{++} (l_3 \otimes l_4) \mathbin{++} (l_3 \otimes l_5).$$

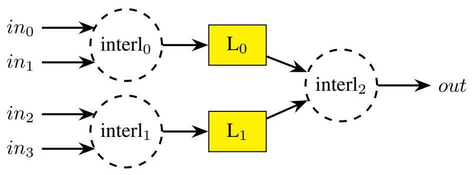


Fig. 8. Data flow of *comp-interl*

The proof relies on our library, *mem-interl-lib*, specifically the following lemma.

$$x \in (y \otimes_2 z) \Rightarrow (x \mathbin{++} x1) \in (prepend\text{-}rec(y, x1) \otimes_2 z) \wedge (x \mathbin{++} x1) \in (y \otimes_2 prepend\text{-}rec(z, x1))$$

Figure 9 reports the verification times of the self-timed circuits discussed in our experiments. All circuits were verified in seconds.

Circuit	Proof time
gcd	8s
Q'_{20}	3s
$interl$	4s

Circuit	Proof time
$igcd$	6s
$comp\text{-}interl$	19s

Fig. 9. Proof times for the self-timed circuits discussed in our experiments. All experiments used an Apple laptop with a 2.9 GHz Intel Core i7 processor, 4MB L3 cache, and 8GB memory. The proof time for a module excludes proof times for its submodules.

VII. Conclusion and Future Work

We have discussed the specification and verification of self-timed circuits represented with a formally-defined, hierarchical HDL. We extend our previous framework [4] to sequential circuits with loops and non-deterministic outputs (in particular, arbitrated merges) that may be parameterized by data size. Hierarchical verification is a key methodology supporting the efficient scalability of our correctness proofs.

We represent self-timed circuits, composed of links and joints, in the DE HDL. This hierarchical HDL provides combinational primitives and several latches suitable for modeling links and joints. To model the non-deterministic advance of signals, every primitive (combinational-logic-only) joint includes an additional *go* signal that, when disabled, prevents a joint from *firing*. A joint action will fire only when all of its input-output conditions and its externally-provided (from an oracle) *go* signal are valid. Thus, when we undertake the verification of a circuit composed of combinational-logic joints and state-holding links, we are modeling all of the possible interleavings of circuit activity.

Our present arbitrated merge model does not guarantee fairness. We plan to implement an arbitration mechanism that supports fairness for this merge operation. We also expect to consider the verification of mixed self-timed, synchronous circuits. For instance, we wish to verify the correctness of data exchange between two synchronous systems over an asynchronous interconnect fabric. Such an advance could contribute to the use of self-timed networks to reduce the use of inter-clock-domain synchronizers.

Acknowledgments

We thank the reviewers for useful comments. This effort has been supported by DARPA under Contract No. FA8650-17-1-7704.

References

[1] M. Kaufmann and J S. Moore. (2019) ACL2 Home Page. http://www.cs.utexas.edu/users/moore/acl2/.

[2] M. Kaufmann, P. Manolios, and J S. Moore, *Computer-Aided Reasoning: An Approach*. Boston, MA: Kluwer Academic Press, 2000.

[3] ——, *Computer-Aided Reasoning: ACL2 Case Studies*. Boston, MA: Kluwer Academic Press, 2000.

[4] C. Chau, W. A. Hunt Jr., M. Kaufmann, M. Roncken, and I. Sutherland, "Data-Loop-Free Self-Timed Circuit Verification," in *Proc of the Twenty Fourth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2018)*, 2018, pp. 51–58.

[5] M. Roncken, S. M. 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)*, 2015, pp. 77–84.

[6] M. Roncken, I. Sutherland, C. Chen, Y. Hei, W. Hunt Jr., C. Chau, S. M. Gilla, H. Park, X. Song, A. He, and H. Chen, "How to Think about Self-Timed Systems," in *Proc of the Fifty First IEEE Asilomar Conference on Signals, Systems, and Computers (Asilomar-2017)*, 2017, pp. 1597–1604.

[7] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, "Click Elements: An Implementation Style for Data-Driven Compilation," in *Proc of the Sixteenth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2010)*, 2010, pp. 3–14.

[8] M. Singh and S. M. Nowick, "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[9] I. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[10] I. Sutherland and S. Fairbanks, "GasP: A Minimal FIFO Control," in *Proc of the Seventh International Symposium on Asynchronous Circuits and Systems (ASYNC-2001)*, 2001, pp. 46–53.

[11] W. A. Hunt Jr., "The DE Language," in *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J S. Moore, Eds. Springer US, 2000, ch. 10, pp. 151–166.

[12] B. C. Brock and W. A. Hunt Jr., "The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor," in *Formal Methods in System Design*. Kluwer Academic Publishers, 1997, vol. 11, pp. 71–104.

[13] H. Park, A. He, M. Roncken, X. Song, and I. Sutherland, "Modular Timing Constraints for Delay-Insensitive Systems," *Computer Science and Technology*, vol. 31, no. 1, pp. 77–106, 2016.

[14] C. Chau, W. A. Hunt Jr., M. Roncken, and I. Sutherland, "A Framework for Asynchronous Circuit Modeling and Verification in ACL2," in *Proc of the Thirteenth Haifa Verification Conference (HVC-2017)*, 2017, pp. 3–18.

[15] M. Bozga, H. Jianmin, O. Maler, and S. Yovine, "Verification of Asynchronous Circuits using Timed Automata," in *Electronic Notes in Theoretical Computer Science*, 2002, vol. 65, pp. 47–59.

[16] K. Desai, K. S. Stevens, and J. O'Leary, "Symbolic Verification of Timed Asynchronous Hardware Protocols," in *Proc of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI-2013)*, 2013, pp. 147–152.

[17] P. Joshi, P. A. Beerel, M. Roncken, and I. Sutherland, "Timing Verification of GasP Asynchronous Circuits: Predicted Delay Variations Observed by Experiment," in *Lecture Notes in Computer Science*, D. Dams, U. Hannemann, and M. Steffen, Eds. Springer Berlin Heidelberg, 2010, ch. 17, pp. 260–276.

[18] H. Kim, P. A. 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)*, 2002, pp. 115–124.

[19] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT press, 1989.

[20] E. Clarke and B. Mishra, "Automatic Verification of Asynchronous Circuits," in *Proc of the Workshop on Logic of Programs*, 1983, pp. 101–115.

[21] 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)*, 2013, pp. 70–84.

[22] Y. Peng, I. W. Jones, and M. R. Greenstreet, "Finding Glitches Using Formal Methods," in *Proc of the Twenty Second IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2016)*, 2016, pp. 45–46.

[23] J. Sparso and S. Furber, *Principles of Asynchronous Circuit Design - A Systems Perspective*. Springer US, 2001.

[24] C. L. Seitz, "System Timing," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Addison-Wesley, 1980, ch. 7, pp. 218–262.