Data-Loop-Free Self-Timed Circuit Verification

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

{ckcuong,hunt,kaufmann}@cs.utexas.edu,

mroncken@pdx.edu, ivans@cecs.pdx.edu

[1] The University of Texas at Austin

[2] Portland State University

May 15, 2018

Motivation

Many efforts in verifying self-timed circuit implementations concern **circuit-level timing properties**.

- Electrical-level timing analysis is conducted to assure that signal propagation of ready signals is always slower than data propagation so that **data are valid when sampled**.

Motivation

Many efforts in verifying self-timed circuit implementations concern **circuit-level timing properties**.

> Electrical-level timing analysis is conducted to assure that signal propagation of ready signals is always slower than data propagation so that **data are valid when sampled**.

Most verification methods for self-timed circuits have concentrated on **small-size** circuits.

Motivation

Many efforts in verifying self-timed circuit implementations concern **circuit-level timing properties**.

- Electrical-level timing analysis is conducted to assure that signal propagation of ready signals is always slower than data propagation so that **data are valid when sampled**.

Most verification methods for self-timed circuits have concentrated on **small-size** circuits.

We are not aware of any scalable formal methods for validating functional properties of self-timed systems.

Motivation

Many efforts in verifying self-timed circuit implementations concern **circuit-level timing properties**.

Electrical-level timing analysis is conducted to assure that signal propagation of ready signals is always slower than data propagation so that **data are valid when sampled**.

Most verification methods for self-timed circuits have concentrated on **small-size** circuits.

We are not aware of any scalable formal methods for validating functional properties of self-timed systems.

Scalable methods for self-timed system verification are highly desirable.

Goals and Approach

Goals:

- Develop scalable methods for reasoning about the functional correctness of self-timed circuits and systems, while **abstracting away circuit-level timing constraints**.
- Implement those methods using the ACL2 theorem proving system, providing a useful automated framework with associated libraries to support the mechanical analysis of arbitrarily large, general-purpose, self-timed circuit designs.

Goals and Approach

Goals:

- Develop scalable methods for reasoning about the functional correctness of self-timed circuits and systems, while **abstracting away circuit-level timing constraints**.
- Implement those methods using the ACL2 theorem proving system, providing a useful automated framework with associated libraries to support the mechanical analysis of arbitrarily large, general-purpose, self-timed circuit designs.

Approach:

- Extend our DE-based, synchronous-style verification system to one that is capable of analyzing self-timed system models.
- Apply the link-joint model [Roncken et al.:2015] to modeling self-timed circuit designs.
- Develop a hierarchical (compositional) reasoning approach that is amenable to verifying correctness of large, non-deterministic systems.

Outline

1. DE System
2. Modeling and Verification Approach
3. Case Studies
4. Conclusions and Future Work

Outline

1. DE System
2. Modeling and Verification Approach
3. Case Studies
4. Conclusions and Future Work

DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing **Mealy machines** [Hunt:2000].

DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing **Mealy machines** [Hunt:2000].

The semantics of the DE language is given by a simulator that computes the **outputs** and **next state** for a module from the module's **current inputs** and **current state**.

DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing **Mealy machines** [Hunt:2000].

The semantics of the DE language is given by a simulator that computes the **outputs** and **next state** for a module from the module's **current inputs** and **current state**.

The DE system has previously been used to model and verify hierarchical synchronous circuits [Brock & Hunt:1997, Slobodova et al.:2011].

- The DE simulator is used repeatedly to evaluate a circuit netlist description each time the **clock input "ticks"** (changes).

DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing **Mealy machines** [Hunt:2000].

The semantics of the DE language is given by a simulator that computes the **outputs** and **next state** for a module from the module's **current inputs** and **current state**.

The DE system has previously been used to model and verify hierarchical synchronous circuits [Brock & Hunt:1997, Slobodova et al.:2011].

- The DE simulator is used repeatedly to evaluate a circuit netlist description each time the **clock input "ticks"** (changes).
- Prove the following two lemmas for each module: a value lemma specifying the module's outputs and a state lemma specifying the module's next state.

DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing **Mealy machines** [Hunt:2000].

The semantics of the DE language is given by a simulator that computes the **outputs** and **next state** for a module from the module's **current inputs** and **current state**.

The DE system has previously been used to model and verify hierarchical synchronous circuits [Brock & Hunt:1997, Slobodova et al.:2011].

- The DE simulator is used repeatedly to evaluate a circuit netlist description each time the **clock input "ticks"** (changes).
- Prove the following two lemmas for each module: a value lemma specifying the module's outputs and a state lemma specifying the module's next state.
- The value and state lemmas of a **composite module** are proved by application of the value and state lemmas of its submodules, **without exploring the internal structures of the submodules**.

DE System

In our self-timed modeling approach, we invoke the DE simulator whenever any primary input changes.

Allow the design to proceed at a rate moderated by **oracle** values — extra input values modeling non-determinacy — that can cause any part of the logic to **delay an arbitrary amount**.

DE System

In our self-timed modeling approach, we invoke the DE simulator whenever any primary input changes.

> Allow the design to proceed at a rate moderated by **oracle** values — extra input values modeling non-determinacy — that can cause any part of the logic to **delay an arbitrary amount**.

We extend the DE primitive database with a new primitive that models the **validity of stored data**.

Outline

1 DE System

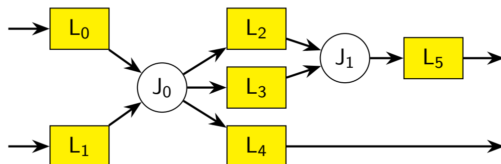
2 Modeling and Verification Approach

3 Case Studies

4 Conclusions and Future Work

Link-Joint Model

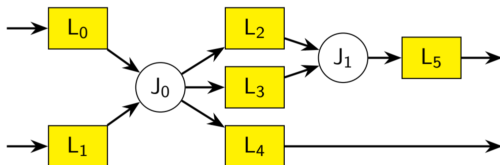
We model self-timed systems as **Mealy machines** representing networks of communication links and computation joints.



Links communicate with each other locally via joints using the **link-joint model** [Roncken et al.:2015].

Link-Joint Model

We model self-timed systems as **Mealy machines** representing networks of communication links and computation joints.

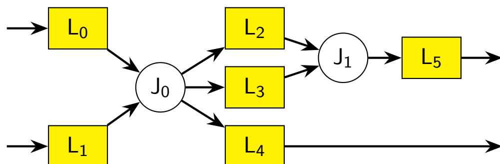


Links communicate with each other locally via joints using the **link-joint model** [Roncken et al.:2015].

- Links are communication channels in which **data** are stored along with **a full/empty signal**.
- Joints are handshake components that implement **data operations** and **flow control**.
- A link connects exactly to one input and one output joint.

Link-Joint Model

We model self-timed systems as **Mealy machines** representing networks of communication links and computation joints.

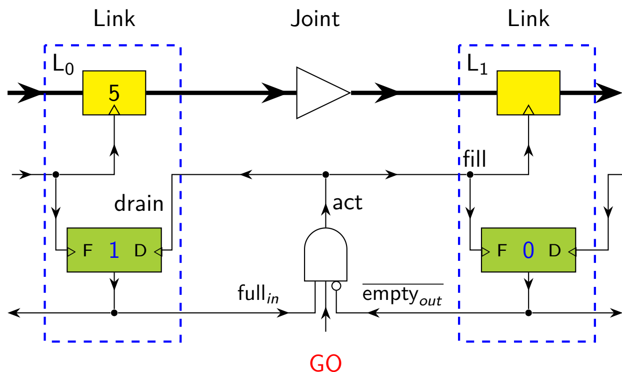


Links communicate with each other locally via joints using the **link-joint model** [Roncken et al.:2015].

- Links are communication channels in which **data** are stored along with **a full/empty signal**.
- Joints are handshake components that implement **data operations** and **flow control**.
- A link connects exactly to one input and one output joint.

Necessary conditions for a **joint-action** to fire: all input and output links of that action are **full** and **empty**, respectively.

Details of the Link-Joint Model

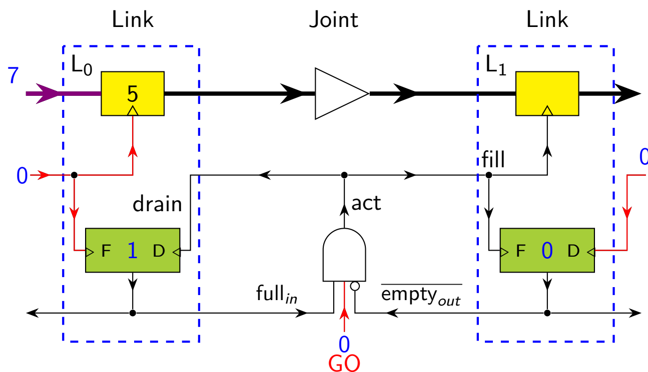


The green boxes represent instances of our new **DE** link-control primitive.

When a joint acts, three tasks will be executed in parallel:

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

Details of the Link-Joint Model (GO = 0)

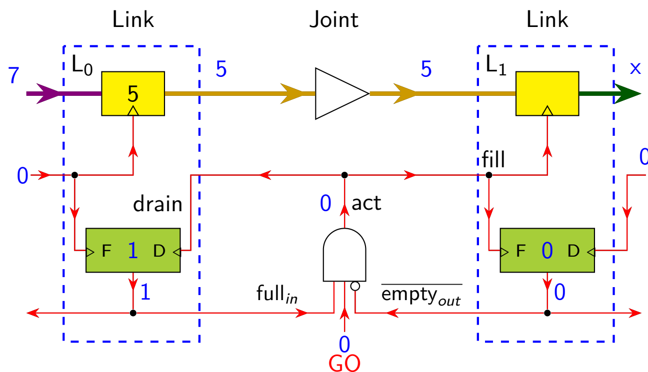


The green boxes represent instances of our new **DE** link-control primitive.

When a joint acts, three tasks will be executed in parallel:

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

Details of the Link-Joint Model (GO = 0)

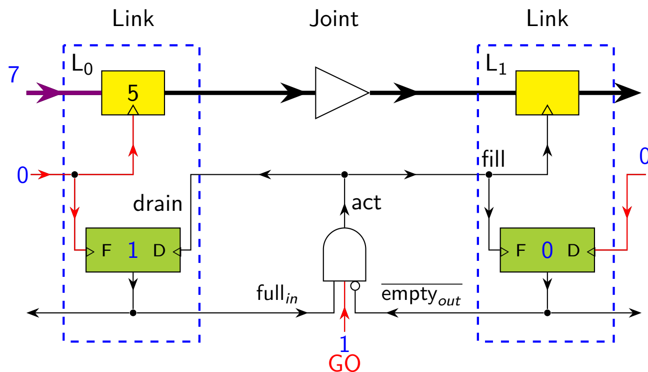


The green boxes represent instances of our new **DE** link-control primitive.

When a joint acts, three tasks will be executed in parallel:

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

Details of the Link-Joint Model (GO = 1)

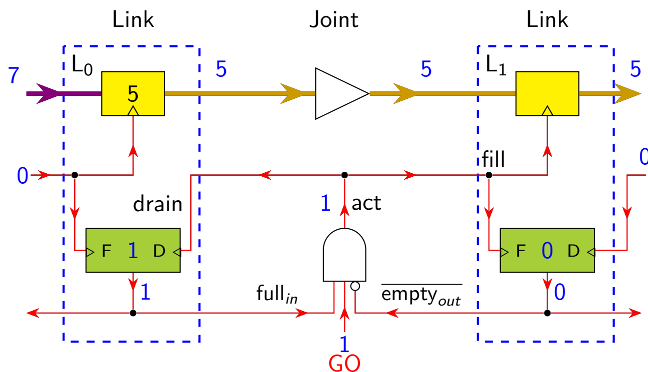


The green boxes represent instances of our new **DE** link-control primitive.

When a joint acts, three tasks will be executed in parallel:

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

Details of the Link-Joint Model (GO = 1)

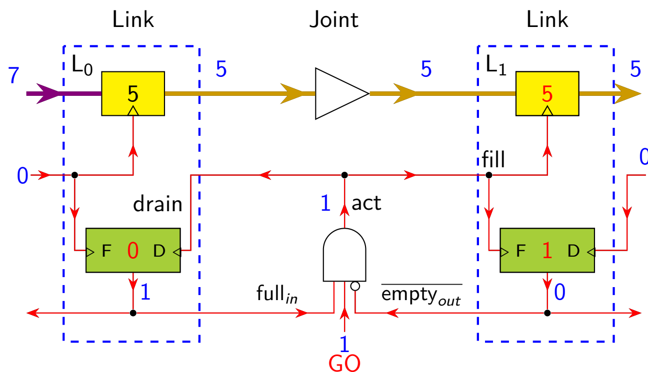


The green boxes represent instances of our new **DE** link-control primitive.

When a joint acts, three tasks will be executed in parallel:

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

Details of the Link-Joint Model $(GO = 1)$

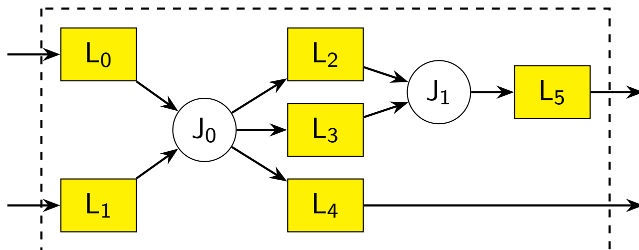


The green boxes represent instances of our new **DE** link-control primitive.

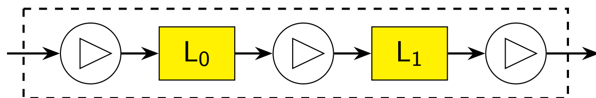
When a joint acts, three tasks will be executed in parallel:

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

Self-Timed Modules



Complex link



Complex joint: a queue of length two, $Q2$

Verification

Objective: Verify the **functional correctness** of self-timed circuit designs.

Approach:

- Develop a hierarchical reasoning approach that avoids exploring internal operations of submodules as well as their interleavings.

Verification

Objective: Verify the **functional correctness** of self-timed circuit designs.

Approach:

- Develop a hierarchical reasoning approach that avoids exploring internal operations of submodules as well as their interleavings.
  - Characterize the **one-step update** on the **output sequence** of a module given its current inputs and current state. We call this property the single-step-update property.

Verification

Objective: Verify the **functional correctness** of self-timed circuit designs.

Approach:

- Develop a hierarchical reasoning approach that avoids exploring internal operations of submodules as well as their interleavings.
  - Characterize the **one-step update** on the **output sequence** of a module given its current inputs and current state. We call this property the single-step-update property.
  - The single-step-update property of a module is established hierarchically using the single-step-update properties of its submodules, without exploring the internal structures of the submodules.

Verification

Objective: Verify the **functional correctness** of self-timed circuit designs.

Approach:

- Develop a hierarchical reasoning approach that avoids exploring internal operations of submodules as well as their interleavings.
  - Characterize the **one-step update** on the **output sequence** of a module given its current inputs and current state. We call this property the single-step-update property.
  - The single-step-update property of a module is established hierarchically using the single-step-update properties of its submodules, without exploring the internal structures of the submodules.
  - The **multi-step input-output relationship** is then proved by **induction** with the single-step-update property.

Verification

Objective: Verify the **functional correctness** of self-timed circuit designs.

Approach:

- Develop a hierarchical reasoning approach that avoids exploring internal operations of submodules as well as their interleavings.
  - Characterize the **one-step update** on the **output sequence** of a module given its current inputs and current state. We call this property the single-step-update property.
  - The single-step-update property of a module is established hierarchically using the single-step-update properties of its submodules, without exploring the internal structures of the submodules.
  - The **multi-step input-output relationship** is then proved by **induction** with the single-step-update property.
- Result: relationship between input and output sequences formalized.

Outline

1. DE System
2. Modeling and Verification Approach
3. Case Studies
4. Conclusions and Future Work

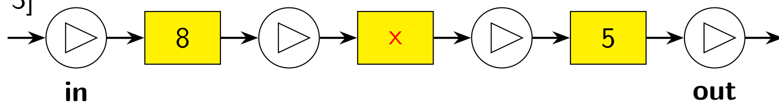
Case Studies

Demonstrate our framework with data-loop-free self-timed circuits.

- Example 1: A FIFO Circuit
- Example 2: Hierarchical Reasoning
- Example 3: Complex Links

Example 1: A FIFO Circuit

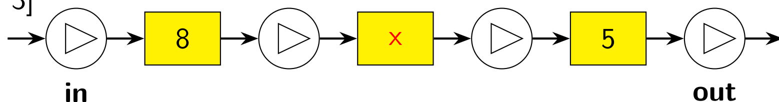
$Q3$



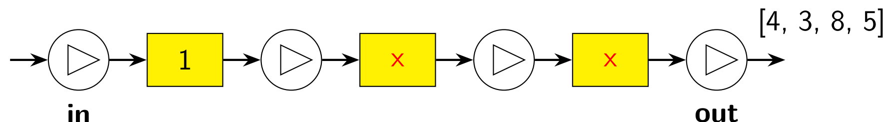
$$[1, 4, 3] \mathrel{++} [8, 5]$$

Example 1: A FIFO Circuit

$Q3$



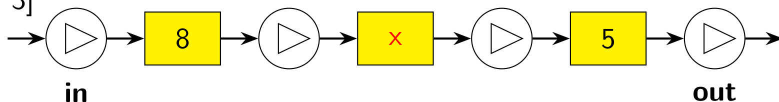
$$[1, 4, 3] ++ [8, 5]$$



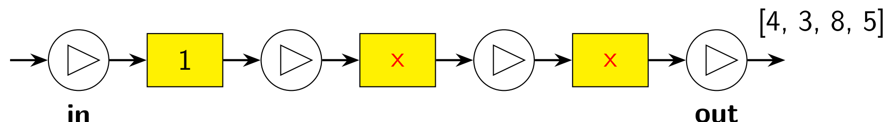
$$[1] ++ [4, 3, 8, 5]$$

Example 1: A FIFO Circuit

$Q3$



$$[1, 4, 3] ++ [8, 5]$$

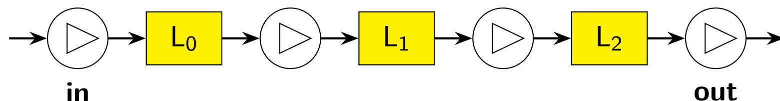


$$[1] ++ [4, 3, 8, 5]$$

$$[1] ++ [4, 3, 8, 5] = [1, 4, 3] ++ [8, 5]$$

Example 1: A FIFO Circuit

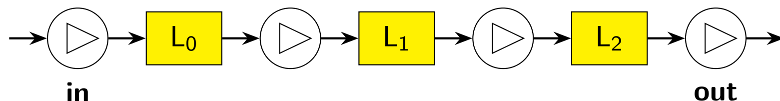
$Q3$



Let in-act and out-act denote the **act** signals from joints **in** and **out**, respectively.

Example 1: A FIFO Circuit

$Q3$

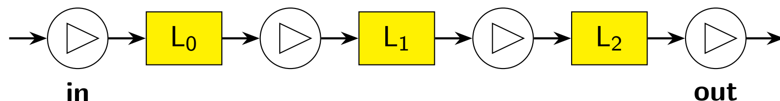


Let in-act and out-act denote the **act** signals from joints **in** and **out**, respectively.

$Q3$ accepts a new data item each time the in-act signal fires. We define in-seq, the **accepted input sequence**, as the sequence of data items that have passed joint **in**.

Example 1: A FIFO Circuit

*Q*3



Let in-act and out-act denote the **act** signals from joints **in** and **out**, respectively.

*Q*3 accepts a new data item each time the in-act signal fires. We define in-seq, the **accepted input sequence**, as the sequence of data items that have passed joint **in**.

Similarly, we define out-seq, the **valid output sequence**, as the sequence of data items that have passed through joint **out** while out-act fires.

Example 1: A FIFO Circuit

The relationship between $Q3$'s in-seq and out-seq.

$$q3\$extract(q3\$run(input\text{-}list, st, n)) ++ out\text{-}seq = \\ in\text{-}seq ++ q3\$extract(st)$$

$q3\$run(input\text{-}list, st, n) :=$

if $(n \leq 0)$ st

else

$q3\$run(rest(input\text{-}list),$

$q3\$step(first(input\text{-}list), st),$ // Return the next state of Q3

$n - 1)$

The extraction function ***q3$extract(st)*** extracts valid data from state ***st*** of $Q3$, i.e., extracts data from links that are **full** at state ***st***.

Example 1: A FIFO Circuit

The relationship between $Q3$'s in-seq and out-seq.

$$q3\$extract(q3\$run(input\text{-}list, st, n)) +\!+\ out\text{-}seq = in\text{-}seq +\!+\ q3\$extract(st)$$

$q3\$run(input\text{-}list, st, n) :=$
if $(n \leq 0)$ st
else
$q3\$run(rest(input\text{-}list),$
$q3\$step(first(input\text{-}list), st),$ // Return the next state of Q3
$n - 1)$

The extraction function ***q3\$extract(st)*** extracts valid data from state ***st*** of $Q3$, i.e., extracts data from links that are **full** at state ***st***.

$out\text{-}seq = in\text{-}seq$ when the initial and final states contain no valid data.

Example 1: A FIFO Circuit

$$q3\$extract(q3\$run(input\text{-}list, st, n)) ++ out\text{-}seq = in\text{-}seq ++ q3\$extract(st) \quad (1)$$

Our ACL2 proof of (1) uses **induction** and the following single-step-update property of $Q3$ as a supporting lemma,

$$q3\$extract(q3\$step(input, st)) = q3\$extracted\text{-}step(input, st) \quad (2)$$

Example 1: A FIFO Circuit

$$q3\$extract(q3\$run(input\text{-}list, st, n)) \mathbin{++} out\text{-}seq = in\text{-}seq \mathbin{++} q3\$extract(st) \quad (1)$$

Our ACL2 proof of (1) uses **induction** and the following single-step-update property of $Q3$ as a supporting lemma,

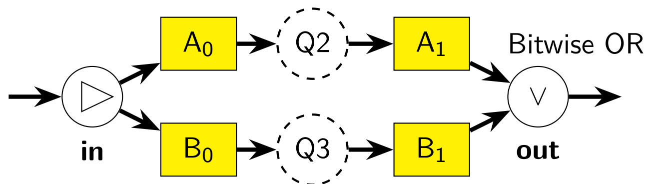
$$q3\$extract(q3\$step(input, st)) = q3\$extracted\text{-}step(input, st) \quad (2)$$

where $q3\$extracted\text{-}step(input, st) :=$

$$\begin{cases} q3\$extract(st), \textbf{if } in\text{-}act = nil \land out\text{-}act = nil \\ [input.data] \mathbin{++} q3\$extract(st), \textbf{if } in\text{-}act = t \land out\text{-}act = nil \\ remove\text{-}last(q3\$extract(st)), \textbf{if } in\text{-}act = nil \land out\text{-}act = t \\ [input.data] \mathbin{++} remove\text{-}last(q3\$extract(st)), \textbf{otherwise} \end{cases}$$

Example 2: Hierarchical Reasoning

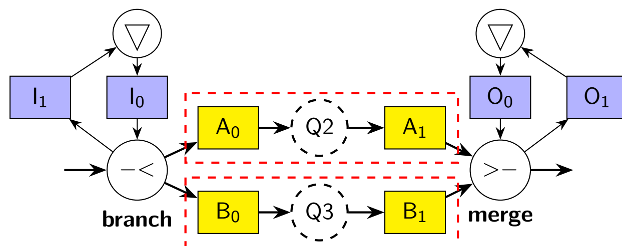
C



The step and extraction functions of C are defined **hierarchically** in terms of those functions of $Q2$ and $Q3$, respectively.

The single-step-update property of C is proved by using the single-step-update properties of $Q2$ and $Q3$.

RR



By abstracting two queues ($A_0 \rightarrow Q2 \rightarrow A_1$) and ($B_0 \rightarrow Q3 \rightarrow B_1$) as two complex links, the reasoning is more efficient as the number of case splits reduced in proving the invariant as well as the single-step-update property for *RR*.

$\Rightarrow$ The verification time of *RR* is reduced from more than 23.5 minutes to 14 seconds by using the complex links.

Outline

1. DE System
2. Modeling and Verification Approach
3. Case Studies
4. Conclusions and Future Work

Conclusions

We have presented a framework for formally modeling and verifying self-timed circuit designs using the DE system.

This work resulted in an ACL2 library for analyzing self-timed systems.

We model self-timed systems as networks of links communicating with each other locally via joints, using the link-joint model.

We model the **non-determinism of event-ordering** in self-timed circuits by associating each joint with an external go signal that, when disabled, prevents a joint from **firing**.

We have developed a hierarchical reasoning method that is capable of verifying the **functional correctness** of self-timed circuit designs at scale.

Future Work

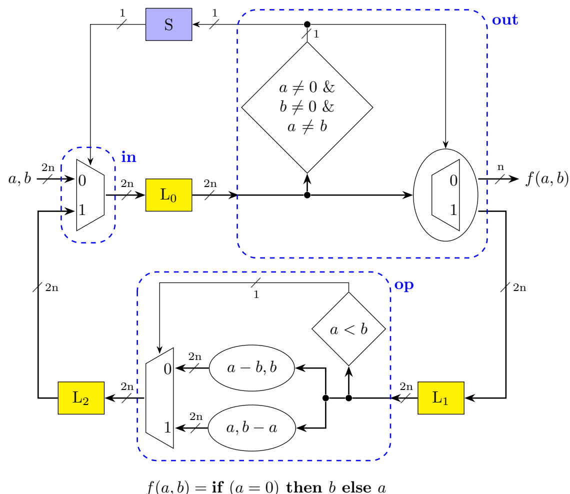
Enhance the effectiveness of our framework by increasing automation through the further introduction of **proof idioms** using **macros**.

Apply our verification methodology to iterative self-timed circuits.

- Verify a self-timed **serial adder** model without imposing the design restrictions presented in our previous work [Chau et al.:2017].

Analyze self-timed circuit models performing arbitrated merge operations that grant **mutually exclusive access** to a shared resource on a **first-come-first-served** (FCFS) basis.

A Greatest-Common-Divisor (GCD) Circuit



$f(a, b) = \textbf{if } (a = 0) \textbf{ then } b \textbf{ else } a$

gcd-alg$(a, b) :=$

while $(a \neq 0) \wedge (b \neq 0) \wedge (a \neq b)$ **do**

if $(a < b)$

then $b := b - a$

else $a := a - b$

return

if $(a = 0)$ **then** b **else** a

References

B. Brock and W. Hunt (1997)

The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor

Formal Methods in System Design, 11, 71 – 104.

C. Chau, W. A. Hunt Jr., M. Roncken, and I. Sutherland (2017)

A Framework for Asynchronous Circuit Modeling and Verification in ACL2

HVC 2017, 3 – 18.

W. A. Hunt Jr. (2000)

The DE Language

Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers Norwell, MA, USA, 151 – 166.

M. Roncken, S. Gilla, H. Park, N. Jamadagni, C. Cowan, I. Sutherland (2015)

Naturalized Communication and Testing

ASYNC 2015, 77 – 84.

A. Slobodova, J. Davis, S. Swords, and W. Hunt Jr. (2011)

A Flexible Formal Verification Framework for Industrial Scale Validation

MEMOCODE 2011, 89 – 97.

Questions?