A Framework for Asynchronous Circuit Modeling and Verification in ACL2

Cuong Chau

ckcuong@cs.utexas.edu

Department of Computer Science

The University of Texas at Austin

October 20, 2017

Outline

Outline

Introduction

Synchronous circuits (or clock-driven circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (or self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communications between state-holding elements are performed via **local communication protocols**.

Introduction

Synchronous circuits (or clock-driven circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (or self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communications between state-holding elements are performed via **local communication protocols**.

Why asynchronous?

Introduction

Synchronous circuits (or clock-driven circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (or self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communications between state-holding elements are performed via **local communication protocols**.

Why asynchronous?

- Low power consumption,
- High operating speed,
- Elimination of clock skew problems,
- Better composability and modularity for large systems,
- ...

Introduction

Our goal: developing scalable methods for reasoning about the functional correctness of self-timed systems using ACL2.

Introduction

Our goal: developing scalable methods for reasoning about the functional correctness of self-timed systems using ACL2.

- We use the DE system [Hunt:2000], which is built in ACL2, to specify and verify self-timed circuit designs.

Introduction

Our goal: developing scalable methods for reasoning about the functional correctness of self-timed systems using ACL2.

- We use the DE system [Hunt:2000], which is built in ACL2, to specify and verify self-timed circuit designs.
- Developing a hierarchical verification approach to support scalability.

Introduction

Our goal: developing scalable methods for reasoning about the functional correctness of self-timed systems using ACL2.

- We use the DE system [Hunt:2000], which is built in ACL2, to specify and verify self-timed circuit designs.
- Developing a hierarchical verification approach to support scalability.
- Exploring strategies for reasoning with non-deterministic circuit behavior.

Outline

The DE System

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

The DE System

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

The DE system supports hierarchical verification:

- Each time a module is specified, there are two lemmas need be proven: a value lemma specifying the module's outputs and a state lemma specifying the module's next state.

The DE System

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

The DE system supports hierarchical verification:

- Each time a module is specified, there are two lemmas need be proven: a value lemma specifying the module's outputs and a state lemma specifying the module's next state.
- If a module doesn't have an internal state (purely combinational), only the value lemma need be proven.

The DE System

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

The DE system supports hierarchical verification:

- Each time a module is specified, there are two lemmas need be proven: a value lemma specifying the module's outputs and a state lemma specifying the module's next state.
- If a module doesn't 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**.

The DE System

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

The DE system supports hierarchical verification:

- Each time a module is specified, there are two lemmas need be proven: a value lemma specifying the module's outputs and a state lemma specifying the module's next state.
- If a module doesn't 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**.
- This approach has been demonstrated its **scalability** to large systems, as shown on contemporary x86 designs at Centaur Technology [Slobodova et al.:2011].

Outline

Modeling

- No global clock signal
- Local communication protocols
- Non-deterministic behavior due to variable delays in wires and gates

Modeling

- No global clock signal
  $\Rightarrow$ Adding local signaling to state-holding devices
- Local communication protocols

- Non-deterministic behavior due to variable delays in wires and gates

Modeling

- No global clock signal
  $\Rightarrow$ Adding local signaling to state-holding devices
- Local communication protocols
  $\Rightarrow$ Modeling the link-joint model introduced by Roncken et al., a universal communication model for various self-timed circuit families [Roncken et al.:2015]
- Non-deterministic behavior due to variable delays in wires and gates

Modeling

- No global clock signal
  $\Rightarrow$ Adding local signaling to state-holding devices
- Local communication protocols
  $\Rightarrow$ Modeling the link-joint model introduced by Roncken et al., a universal communication model for various self-timed circuit families [Roncken et al.:2015]
- Non-deterministic behavior due to variable delays in wires and gates
  $\Rightarrow$ Employing an oracle, which we call a collection of go signals. These signals are part of the input.

The Link-Joint Model

We model self-timed systems as finite state machines (FSMs) representing networks of communication links.

Links communicate with each other locally via **handshake components**, which are called joints, using the link-joint model.

The Link-Joint Model

We model self-timed systems as finite state machines (FSMs) representing networks of communication links.

Links communicate with each other locally via **handshake components**, which are called joints, using the link-joint model.

- Links are communication channels in which **data** and **full/empty states** are stored.
- Joints are handshake components that implement **flow control** and **data operations**.

The Link-Joint Model

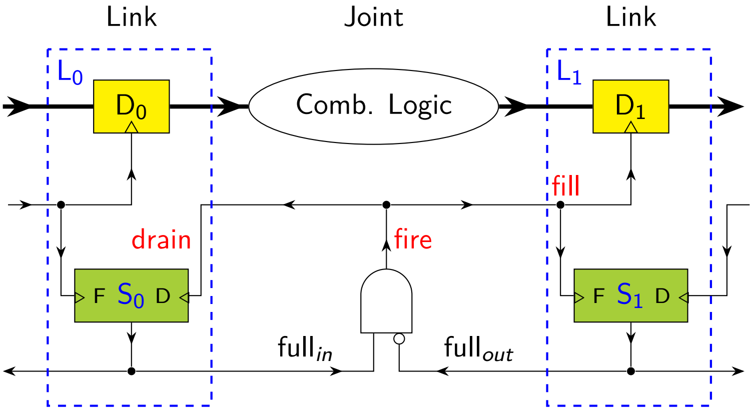
We model self-timed systems as finite state machines (FSMs) representing networks of communication links.

Links communicate with each other locally via **handshake components**, which are called joints, using the link-joint model.

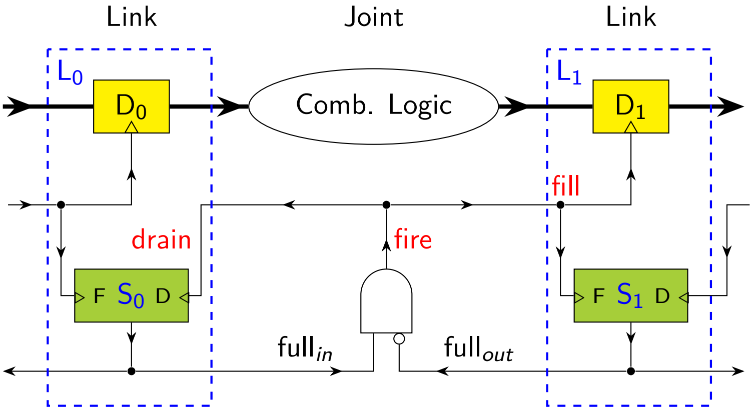
- Links are communication channels in which **data** and **full/empty states** are stored.
- Joints are handshake components that implement **flow control** and **data operations**.

Joints are the meeting points for links to **coordinate states** and **exchange data**.

The Link-Joint Model



The Link-Joint Model

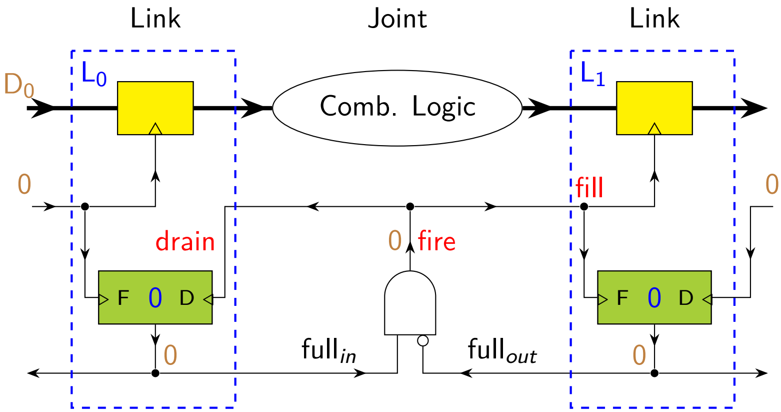


A joint can have several input and output links connected to it.

A joint can have multiple (guarded) mutually exclusive actions.

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

The Link-Joint Model

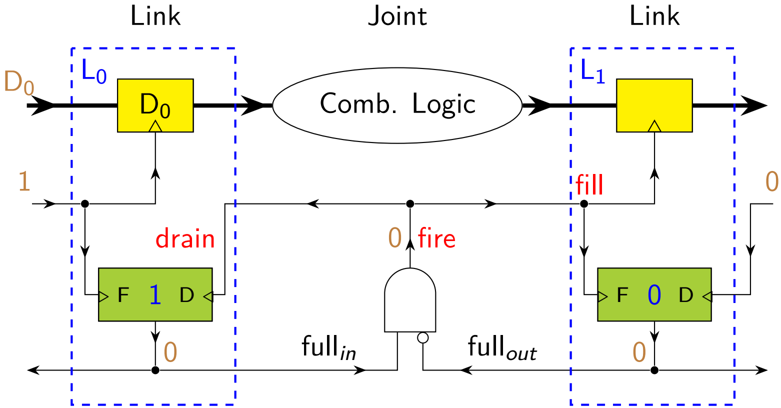


A joint can have several input and output links connected to it.

A joint can have multiple (guarded) mutually exclusive actions.

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

The Link-Joint Model

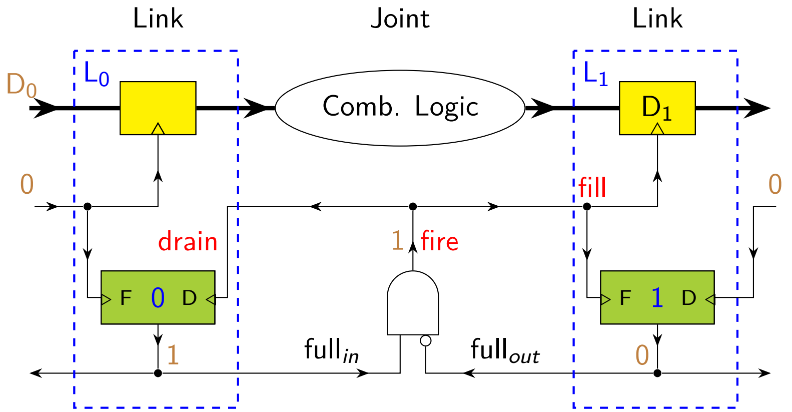


A joint can have several input and output links connected to it.

A joint can have multiple (guarded) mutually exclusive actions.

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

The Link-Joint Model

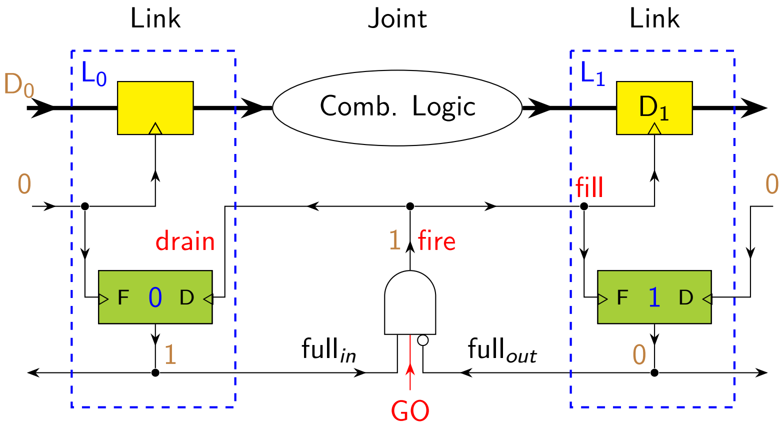


A joint can have several input and output links connected to it.

A joint can have multiple (guarded) mutually exclusive actions.

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

The Link-Joint Model

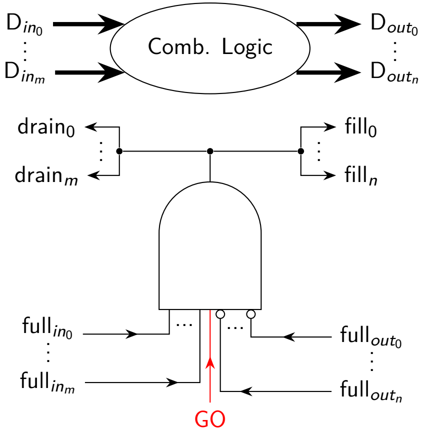


A joint can have several input and output links connected to it.

A joint can have multiple (guarded) mutually exclusive actions.

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

The Link-Joint Model



When a joint-action fires, three tasks will be executed in parallel:

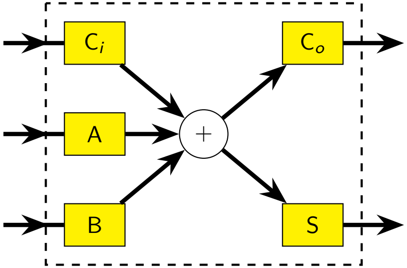
- transfer data computed from the input links to the output links,
- fill the output links, make them **full**,
- drain the input links, make them **empty**.

Verification

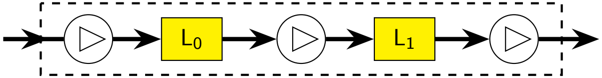
Our framework applies a hierarchical verification approach to formalizing single transitions of circuit behavior (simulated by `se` and `de` functions).

- The output and next state of a module are formalized using the formalized outputs and next states of submodules, without delving into details about the submodules.
- Self-timed modules can be abstracted as “complex” links or “complex” joints.

Self-Timed Modules



A complex link: an adder



A complex joint: a queue $Q2$ of two links

Verification

Compositional reasoning:

- Functional properties of self-timed systems may involve multi-step executions that are quite burdensome to establish directly.
- Decompose the executions into smaller steps in such a way that sub-properties after executing each of these smaller steps can be carried out much easier.
- The desired properties are then established by simply composing these sub-properties.

Verification

Compositional reasoning:

- Functional properties of self-timed systems may involve multi-step executions that are quite burdensome to establish directly.
- Decompose the executions into smaller steps in such a way that sub-properties after executing each of these smaller steps can be carried out much easier.
- The desired properties are then established by simply composing these sub-properties.

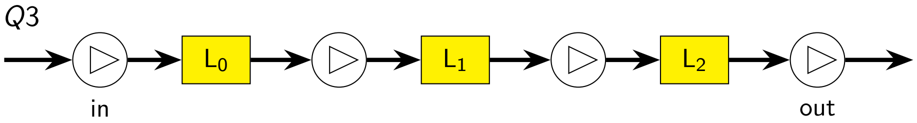
Induction:

- We use induction to prove properties of systems over time, for instance, the relationship between input and output sequences.
- We also apply induction to establishing **loop invariants** of iterative circuits, i.e., circuits with feedback loops.

Outline

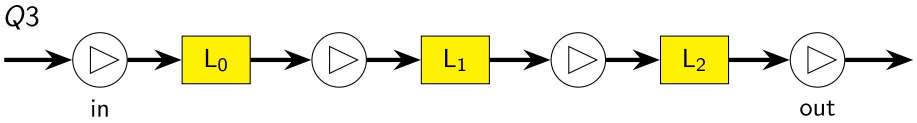
Outline

Example 1



For self-timed circuits with no feedback loops, we verify their functional correctness in terms of the relationship between their input and output sequences.

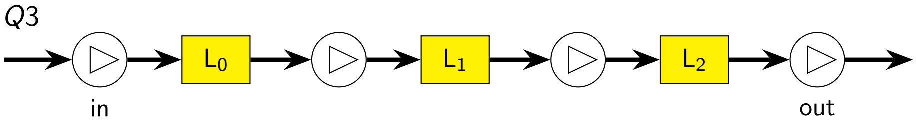
Example 1



For self-timed circuits with no feedback loops, we verify their functional correctness in terms of the relationship between their input and output sequences.

Let **in-act** denote the **fire** signal from the AND gate in the control logic of joint **in**, **out-act** denote the **fire** signal from the AND gate in the control logic of joint **out**.

Example 1

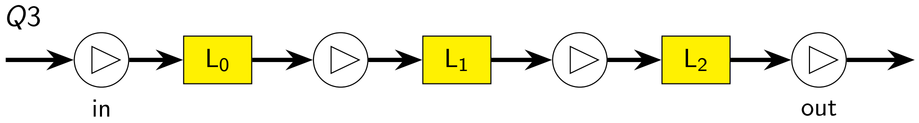


For self-timed circuits with no feedback loops, we verify their functional correctness in terms of the relationship between their input and output sequences.

Let **in-act** denote the **fire** signal from the AND gate in the control logic of joint **in**, **out-act** denote the **fire** signal from the AND gate in the control logic of joint **out**.

Module *Q*3 will accept a new input if the **in-act** signal is high. In this case, we call this input valid. We define a (valid) input sequence as a sequence of valid inputs.

Example 1



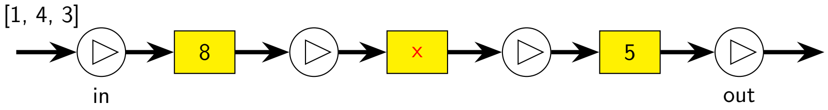
For self-timed circuits with no feedback loops, we verify their functional correctness in terms of the relationship between their input and output sequences.

Let **in-act** denote the **fire** signal from the AND gate in the control logic of joint **in**, **out-act** denote the **fire** signal from the AND gate in the control logic of joint **out**.

Module $Q3$ will accept a new input if the **in-act** signal is high. In this case, we call this input valid. We define a (valid) input sequence as a sequence of valid inputs.

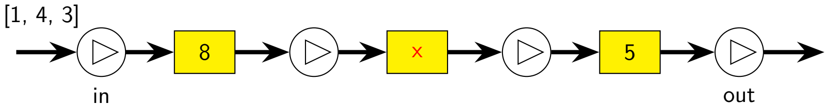
Similarly, we define a (valid) output sequence as a sequence of valid outputs. $Q3$ will report a valid output when the **out-act** signal is high.

Example 1

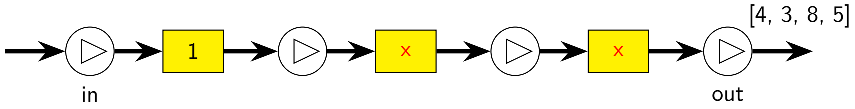


$$[1, 4, 3] ++ [8, 5]$$

Example 1

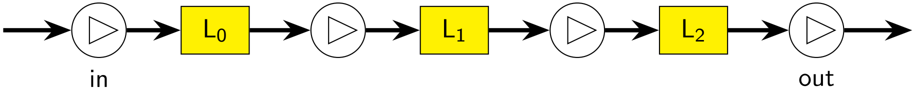


$$[1, 4, 3] ++ [8, 5]$$



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

Example 1



We define the function **q3$extract-data(st)** that extracts valid data from state *st* of *Q*3, i.e. extracts data from links that are **full** at state *st*. The following equation states the functional correctness of *Q*3:

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

where

```
$q3\$run(input\text{-}list, st, n) :=$
  if $(n \leq 0)$ $st$
  else $q3\$run(rest(input\text{-}list),$
               $q3\$step(first(input\text{-}list), st),$
               $n - 1)$
```

Example 1

$$q3\$extract\text{-}data(q3\$run(input\text{-}list, st, n)) ++ out\text{-}seq = in\text{-}seq ++ q3\$extract\text{-}data(st) \tag{4.1}$$

Our ACL2 proof of (4.1) uses **induction** and the following single-step-update property of $Q3$ as a supporting lemma:

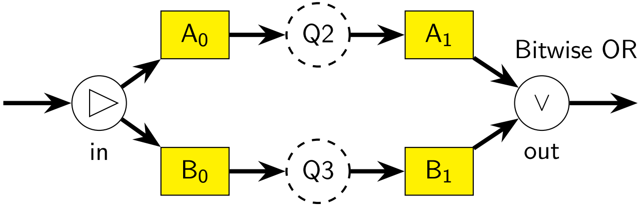
$$q3\$extract\text{-}data(q3\$step(input, st)) = q3\$step\text{-}spec(input, st) \tag{4.2}$$

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

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

Example 2

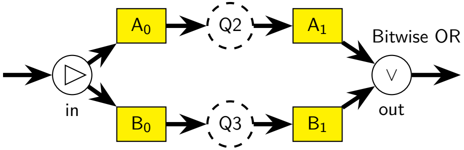
C



In terms of input-output relationship, C simply performs the bitwise OR operation on the two input operands. The operation of C over the input sequence is defined as follows:

```
$c\$op(in\text{-}seq) :=$
  if $(in\text{-}seq = NULL)$ $nil$
  else
    let $input := first(in\text{-}seq)$
    return $[v\text{-}or(input.a, input.b)] ++ c\$op(rest(in\text{-}seq))$
```

Example 2



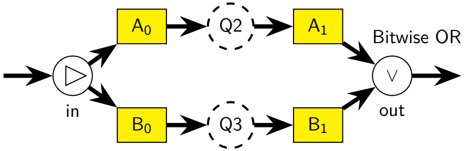
If the current state *st* of *C* satisfies the following condition:

$$c\$inv(st) := \Big(size\big(extract\text{-}data([st.A_0] \mathbin{++} st.Q2 \mathbin{++} [st.A_1])\big) = size\big(extract\text{-}data([st.B_0] \mathbin{++} st.Q3 \mathbin{++} [st.B_1])\big)\Big)$$

then the following functional property of *C* holds:

$$c\$extract\text{-}data(c\$run(input\text{-}list, st, n)) \mathbin{++} out\text{-}seq = c\$op(in\text{-}seq) \mathbin{++} c\$extract\text{-}data(st) \tag{4.3}$$

Example 2



If the current state *st* of *C* satisfies the following condition:

$$c\$inv(st) := \Big(size\big(extract\text{-}data([st.A_0] \mathbin{++} st.Q2 \mathbin{++} [st.A_1])\big) = size\big(extract\text{-}data([st.B_0] \mathbin{++} st.Q3 \mathbin{++} [st.B_1])\big)\Big)$$

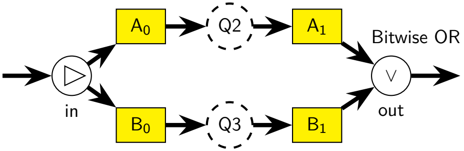
then the following functional property of *C* holds:

$$c\$extract\text{-}data(c\$run(input\text{-}list, st, n)) \mathbin{++} out\text{-}seq = c\$op(in\text{-}seq) \mathbin{++} c\$extract\text{-}data(st) \tag{4.3}$$

$$c\$extract\text{-}data(st) := c\$op(extract\text{-}data([st.A_0] \mathbin{++} st.Q2 \mathbin{++} [st.A_1]) \otimes extract\text{-}data([st.B_0] \mathbin{++} st.Q3 \mathbin{++} [st.B_1]))$$

E.g., $[1,3,5] \otimes [2,4,6] = [[1,2],[3,4],[5,6]]$

Example 2



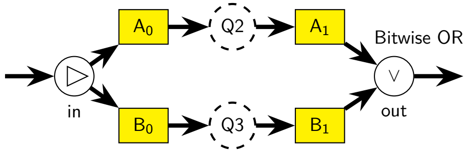
Our ACL2 proof of (4.3) uses **induction** and the single-step-update property (described below, given that **c$inv(st)** holds) as a supporting lemma.

$$c\$extract\text{-}data(c\$step(input, st)) = c\$step\text{-}spec(input, st) \qquad (4.4)$$

where $c\$step\text{-}spec(input, st) :=$

$$\begin{cases} c\$extract\text{-}data(st), \textbf{if } in\text{-}act = F \wedge out\text{-}act = F \\ [v\text{-}or(input.a, input.b)] ++ c\$extract\text{-}data(st), \textbf{if } in\text{-}act = T \wedge out\text{-}act = F \\ remove\text{-}last(c\$extract\text{-}data(st)), \textbf{if } in\text{-}act = F \wedge out\text{-}act = T \\ [v\text{-}or(input.a, input.b)] ++ remove\text{-}last(c\$extract\text{-}data(st)), \textbf{otherwise} \end{cases}$$

Example 2



Our ACL2 proof of (4.3) uses **induction** and the single-step-update property (described below, given that **c$inv(st)** holds) as a supporting lemma.

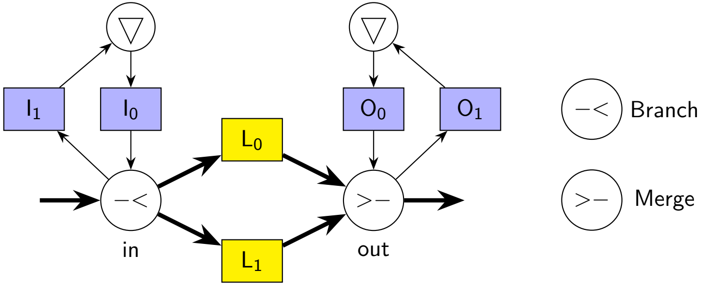
$$c\$extract\text{-}data(c\$step(input, st)) = c\$step\text{-}spec(input, st) \tag{4.4}$$

Our proof of (4.4) does not concern the details of $Q2$ and $Q3$. Instead, we use their single-step-update properties to prove (4.4).

In other words, we employ a hierarchical strategy in proving the single-step-update property of a self-timed module.

Example 3

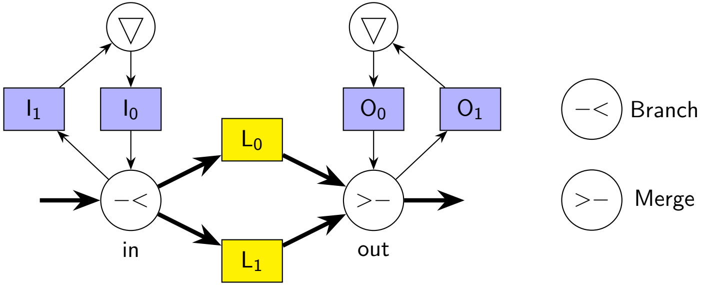
WW



WW alternately inputs data into links L_0 and L_1 and alternately outputs data from links L_0 and L_1.

Example 3

WW

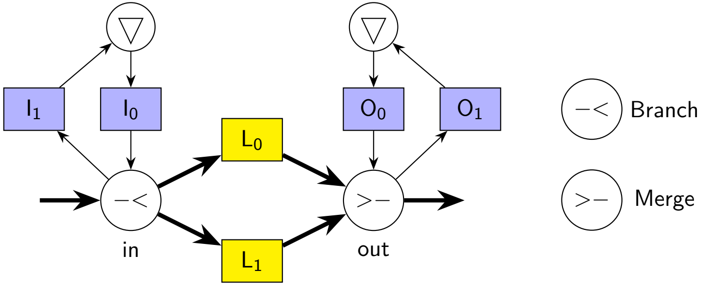


WW alternately inputs data into links L_0 and L_1 and alternately outputs data from links L_0 and L_1.

The (Boolean) value of links I_0 and O_0 indicate which of two links L_0 and L_1 will be input and output, respectively.

Example 3

WW

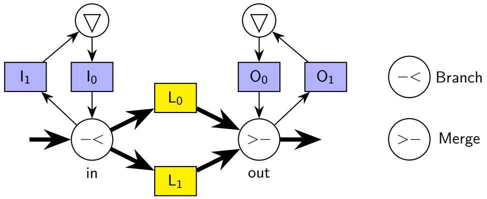


WW alternately inputs data into links L_0 and L_1 and alternately outputs data from links L_0 and L_1.

The (Boolean) value of links I_0 and O_0 indicate which of two links L_0 and L_1 will be input and output, respectively.

When the branch joint fires, it will **fill either** L_0 or L_1, but not both. Likewise, the merge joint will **drain either** L_0 or L_1 when it fires.

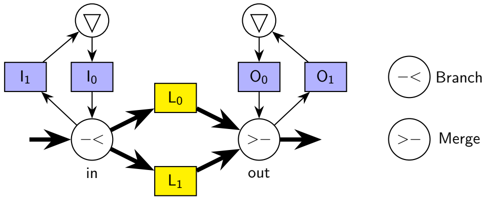
Example 3



The functionality of WW is equivalent to $Q2$, but potentially has higher performance due to shorter latencies:

- WW can input data into either L_0 or L_1, while $Q2$ can input data only into L_0;
- WW can output data from either L_0 or L_1, while $Q2$ can output data only from L_1.

Example 3

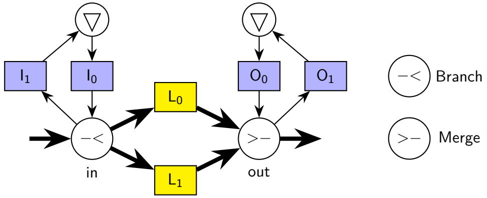


The functionality of WW is equivalent to $Q2$, but potentially has higher performance due to shorter latencies:

- WW can input data into either L_0 or L_1, while $Q2$ can input data only into L_0;
- WW can output data from either L_0 or L_1, while $Q2$ can output data only from L_1.

Our correctness proof of WW involves establishing an **invariant**.

Example 3

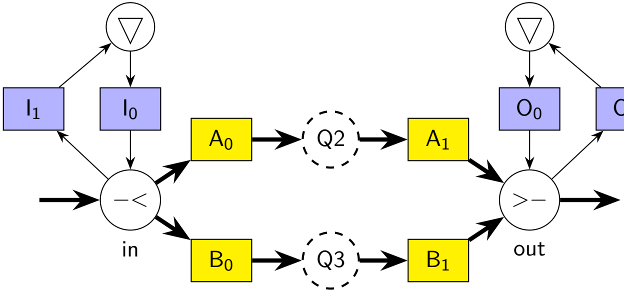


$$ww\$extract\text{-}data(ww\$run(input\text{-}list, st, n)) \mathbin{++} out\text{-}seq = in\text{-}seq \mathbin{++} ww\$extract\text{-}data(st) \qquad (4.5)$$

```
$ww\$extract\text{-}data(st) :=$
  if $full(st.O_0.status)$
    if $(st.O_0.data = T)$ $extract\text{-}data([st.L_0, st.L_1])$
    else $extract\text{-}data([st.L_1, st.L_0])$
  else if $(st.O_1.data = T)$ $extract\text{-}data([st.L_0, st.L_1])$
  else $extract\text{-}data([st.L_1, st.L_0])$
```

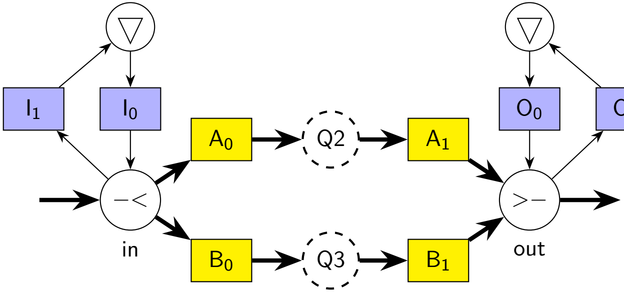
Example 4

RR



Example 4

RR

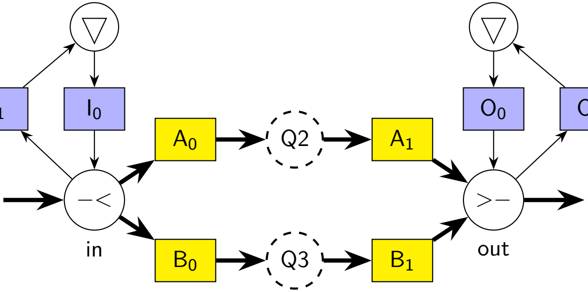


The single-step-update property:

$$rr\$extract\text{-}data(rr\$step(input, st)) = rr\$step\text{-}spec(input, st) \qquad (4.6)$$

Example 4

RR



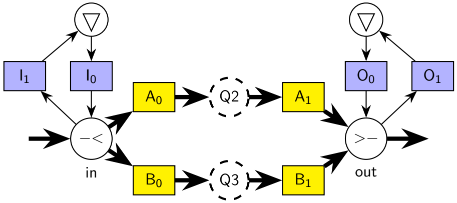
The single-step-update property:

$$rr\$extract\text{-}data(rr\$step(input, st)) = rr\$step\text{-}spec(input, st) \tag{4.6}$$

The relationship between input and output sequences:

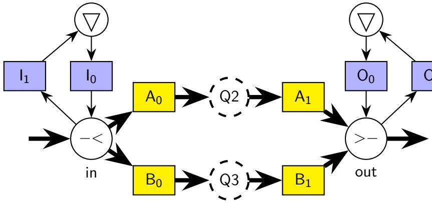
$$rr\$extract\text{-}data(rr\$run(input\text{-}list, st, n)) \mathbin{++} out\text{-}seq = \\ in\text{-}seq \mathbin{++} rr\$extract\text{-}data(st) \tag{4.7}$$

Example 4



The verification time of *RR* is about 15 minutes, while it only takes **5 seconds** to verify *WW* on a 2.9 GHz Intel Core i7 processor with 4MB L3 cache and 8GB memory.

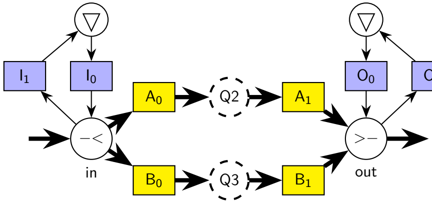
Example 4



The verification time of *RR* is about 15 minutes, while it only takes **5 seconds** to verify *WW* on a 2.9 GHz Intel Core i7 processor with 4MB L3 cache and 8GB memory.

There are many case splits in proving the invariant as well as the single-step-update property for *RR*. It takes 3.5 minutes to prove the invariant and 11.5 minutes to prove the single-step-update property.

Example 4

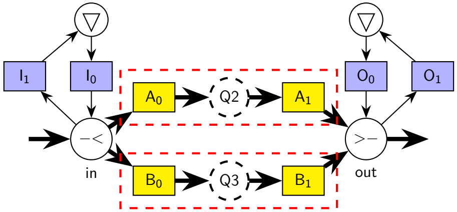


The verification time of *RR* is about 15 minutes, while it only takes **5 seconds** to verify *WW* on a 2.9 GHz Intel Core i7 processor with 4MB L3 cache and 8GB memory.

There are many case splits in proving the invariant as well as the single-step-update property for *RR*. It takes 3.5 minutes to prove the invariant and 11.5 minutes to prove the single-step-update property.

Can we reduce the number of case splits?

Example 4



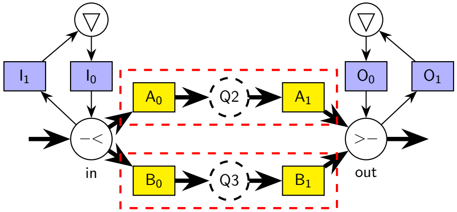
The verification time of *RR* is about 15 minutes, while it only takes **5 seconds** to verify *WW* on a 2.9 GHz Intel Core i7 processor with 4MB L3 cache and 8GB memory.

There are many case splits in proving the invariant as well as the single-step-update property for *RR*. It takes 3.5 minutes to prove the invariant and 11.5 minutes to prove the single-step-update property.

Can we reduce the number of case splits?

Solution: Abstract two queues ($A_0 \rightarrow Q2 \rightarrow A_1$) and ($B_0 \rightarrow Q3 \rightarrow B_1$) as two complex links.

Example 4



The verification time of *RR* is about 15 minutes, while it only takes **5 seconds** to verify *WW* on a 2.9 GHz Intel Core i7 processor with 4MB L3 cache and 8GB memory.

There are many case splits in proving the invariant as well as the single-step-update property for *RR*. It takes 3.5 minutes to prove the invariant and 11.5 minutes to prove the single-step-update property.

Can we reduce the number of case splits?

Solution: Abstract two queues ($A_0 \to Q2 \to A_1$) and ($B_0 \to Q3 \to B_1$) as two complex links.

$\Rightarrow$ The verification time of the new *RR* circuit is about 9 seconds.

In summary, for each self-timed module that has no feedback loops, we prove the following two properties:

1. the single-step-update property (proved by using **hierarchical reasoning**),
2. the relationship between the input and output sequences (proved by using **induction** and the single-step-update property).

Outline

Reasoning with highly non-deterministic behavior in self-timed systems with feedback loops is very challenging.

- Computing loop invariants in these systems becomes much more complicated than in synchronous systems.

Reasoning with highly non-deterministic behavior in self-timed systems with feedback loops is very challenging.

- Computing loop invariants in these systems becomes much more complicated than in synchronous systems.

We impose design restrictions on iterative circuits to reduce non-determinism, and consequently reduce the complexity of the set of execution paths:

- These restrictions enable our framework to verify loop invariants efficiently via **induction** and subsequently verify the **functional correctness** of self-timed circuit designs.

Reasoning with highly non-deterministic behavior in self-timed systems with feedback loops is very challenging.

- Computing loop invariants in these systems becomes much more complicated than in synchronous systems.

We impose design restrictions on iterative circuits to reduce non-determinism, and consequently reduce the complexity of the set of execution paths:

- These restrictions enable our framework to verify loop invariants efficiently via **induction** and subsequently verify the **functional correctness** of self-timed circuit designs.

Design restrictions: A module is ready to communicate with other modules only when it finishes all of its internal operations and becomes quiescent.

32-Bit Self-Timed Serial Adder Verification

We demonstrate our framework by modeling and verifying the functional correctness of a 32-bit self-timed serial adder [Chau:2017].

We prove that the self-timed serial adder indeed performs the addition under an appropriate initial condition.

- When the adder finishes its execution, the result is proven to be the sum of the two 32-bit input operands and the carry-in.

32-Bit Self-Timed Serial Adder Verification

We demonstrate our framework by modeling and verifying the functional correctness of a 32-bit self-timed serial adder [Chau:2017].

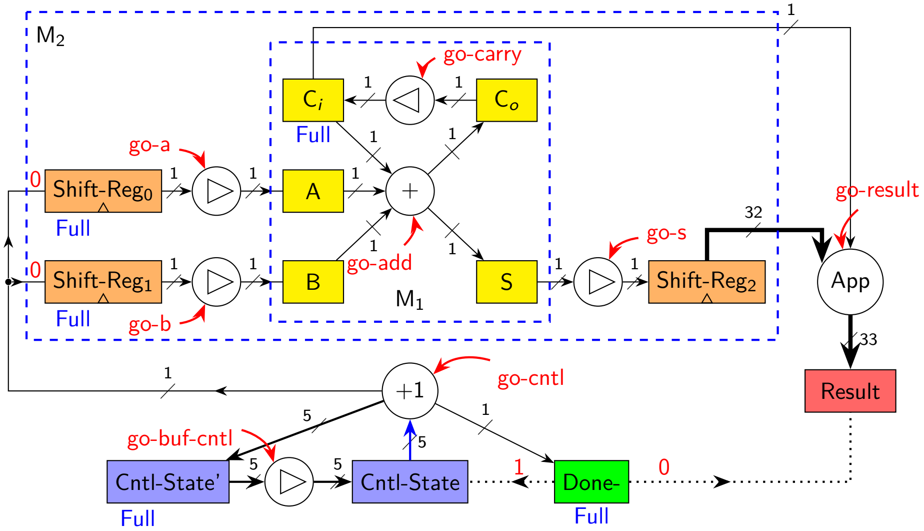
We prove that the self-timed serial adder indeed performs the addition under an appropriate initial condition.

- When the adder finishes its execution, the result is proven to be the sum of the two 32-bit input operands and the carry-in.

Our verification approach applies compositional reasoning.

- Divide the adder's execution into two parts: the loop part and the exit part (the execution after exiting the loop),
- Formalize a loop invariant for the loop part and the adder behavior during the exit part,
- Prove the functional correctness of the adder by glueing these two parts together.

Data Flow of a 32-Bit Self-Timed Serial Adder



Correctness Theorems

Theorem 1 (Partial correctness).

$$\begin{aligned}
& \mathit{async\_serial\_adder}(\mathit{netlist}) \wedge && (1)\\
& \mathit{init\_state}(\mathit{st}) \wedge && (2)\\
& (\mathit{operand\_size} = 32) \wedge && (3)\\
& \mathit{interleavings\_spec}(\mathit{input\text{-}list}, \mathit{operand\_size}) \wedge && (4)\\
& (\mathit{st}' = \mathit{run}(\mathit{netlist}, \mathit{input\text{-}list}, \mathit{st}, n)) \wedge && (5)\\
& \mathit{full}(\mathit{st}'.\mathit{result}.\mathit{status}) && (6)\\
\Rightarrow\ & \mathit{st}'.\mathit{result}.\mathit{data} = \mathit{st}.\mathit{shift\_reg\_0}.\mathit{data} + \\
& \qquad \mathit{st}.\mathit{shift\_reg\_1}.\mathit{data} + \\
& \qquad \mathit{st}.\mathit{ci}.\mathit{data}
\end{aligned}$$

Correctness Theorems

Theorem 2 (Termination).

$$\begin{aligned}
&\quad \mathit{async\_serial\_adder}(\mathit{netlist}) \wedge && (1)\\
&\quad \mathit{init\_state}(\mathit{st}) \wedge && (2)\\
&\quad (\mathit{operand\_size} = 32) \wedge && (3)\\
&\quad \mathit{interleavings\_spec}(\mathit{input\text{-}list}, \mathit{operand\_size}) \wedge && (4)\\
&\quad (\mathit{st}' = \mathit{run}(\mathit{netlist}, \mathit{input\text{-}list}, \mathit{st}, n)) \wedge && (5)\\
&\quad (n \geq \mathit{num\_steps}(\mathit{input\text{-}list}, \mathit{operand\_size})) && (6')\\
\Rightarrow &\ \mathit{full}(\mathit{st}'.\mathit{result}.\mathit{status})
\end{aligned}$$

Outline

Future Work

We are developing a new proof technique for partial correctness of **iterative** self-timed circuits that does not have any conditions on the values of **go** signals.

Future Work

We are developing a new proof technique for partial correctness of **iterative** self-timed circuits that does not have any conditions on the values of **go** signals.

For termination proofs, we need a constraint on **go** signals guaranteeing that **delays are bounded**.

Future Work

We are developing a new proof technique for partial correctness of **iterative** self-timed circuits that does not have any conditions on the values of **go** signals.

For termination proofs, we need a constraint on **go** signals guaranteeing that **delays are bounded**.

We intend to follow a **hierarchical approach** to prove module-level properties of iterative circuits of the following form:

- Given an initial state of the module, the module's **final state** meets its specification after that module completes execution.

Conclusions

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

Our goal is to develop a methodology that is capable of verifying the functional correctness of self-timed circuit designs at large scale.

- This work also provides a library for analyzing self-timed systems in ACL2.

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

We model the **non-determinism of event-ordering** in self-timed circuits by associating each joint with an external go signal.

Our key proof techniques are hierarchical reasoning, compositional reasoning, and induction.

References

C. Chau, W. Hunt, M. Roncken, and I. Sutherland (2017)
A Framework for Asynchronous Circuit Modeling and Verification in ACL2
HVC 2017, to appear.

W. Hunt (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 (2011)
A Flexible Formal Verification Framework for Industrial Scale Validation
MEMOCODE 2011, 89 – 97.

Questions?