A Framework for Asynchronous Circuit Modeling and Verification in ACL2

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

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

marly.roncken@gmail.com, ivans@cecs.pdx.edu

[1] The University of Texas at Austin

[2] Portland State University

November 16, 2017

Outline

1. Introduction
2. The DE System
3. Modeling and Verification Approach
4. 32-Bit Self-Timed Serial Adder Verification
5. Future Work and Conclusions

Outline

1. Introduction
2. The DE System
3. Modeling and Verification Approach
4. 32-Bit Self-Timed Serial Adder Verification
5. Future Work and Conclusions

Introduction

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

Asynchronous circuits (or self-timed circuits): no global clock signal. The communications between storage elements are performed via **local communication protocols**.

Introduction

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

Asynchronous circuits (or self-timed circuits): no global clock signal. The communications between storage elements are performed via **local communication protocols**.

Why asynchronous?

Introduction

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

Asynchronous circuits (or self-timed circuits): no global clock signal. The communications between storage 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.

- Using 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.

- Using 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.

- Using 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:

- Prove the following two lemmas **hierarchically** for each module: 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:

- Prove the following two lemmas **hierarchically** for each module: 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:

- Prove the following two lemmas **hierarchically** for each module: 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:

- Prove the following two lemmas **hierarchically** for each module: 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
  ⇒ Adding local signaling to state-holding devices
- Local communication protocols
  ⇒ 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
  ⇒ 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

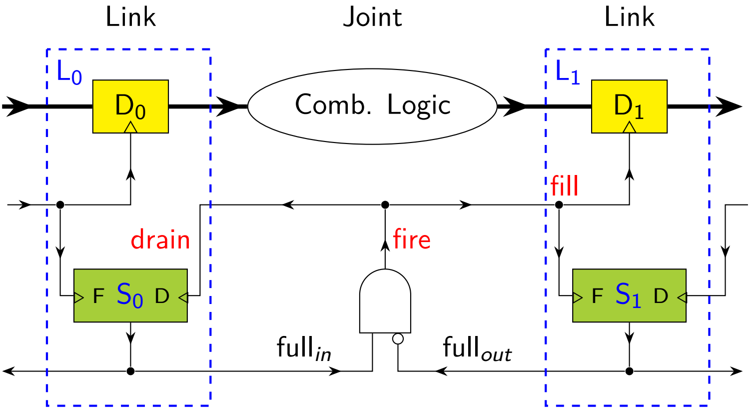
We model self-timed systems as finite state machines (FSMs) representing networks of communication links.

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

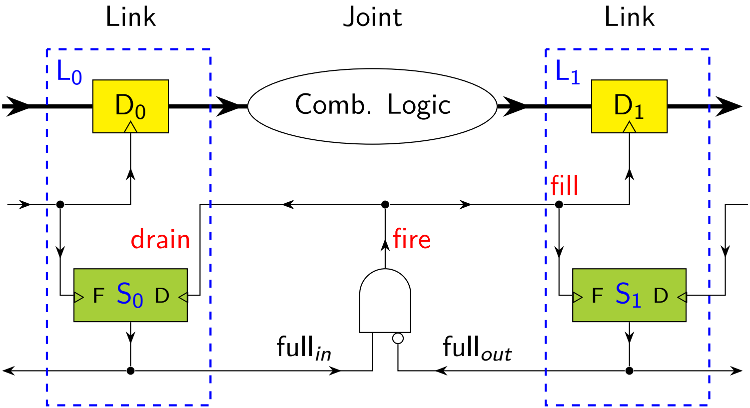
- Links are communication channels in which **data** and **full/empty states** are stored.
- Joints are handshake components that implement **flow control** and **data operations**.

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

The Link-Joint Model



The Link-Joint Model

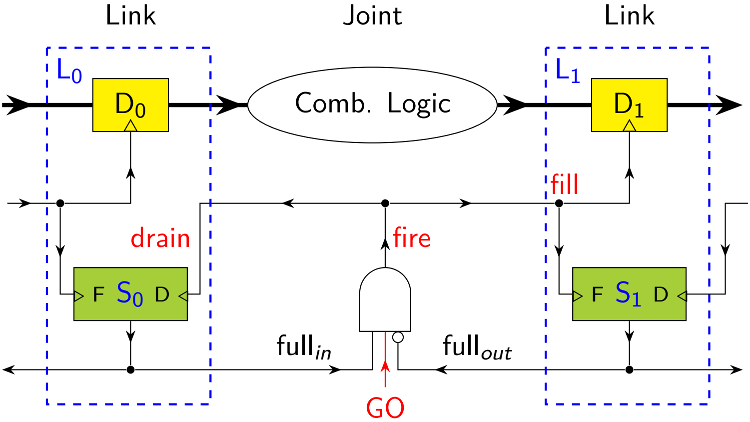


A joint can have several input and output links connected to it.

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

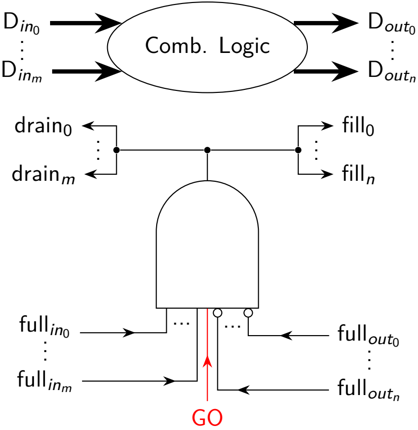


A joint can have several input and output links connected to it.

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



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

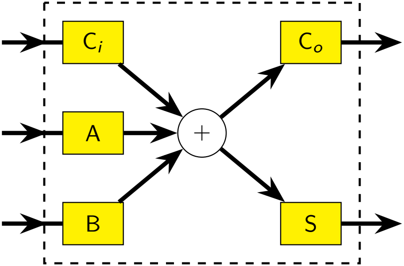
- transfer data computed from the input links to the output links,
- fill the output links, make them **full**,
- drain the input links, make them **empty**.

Verification

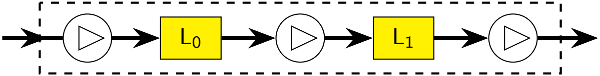
Hierarchical reasoning:

- 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 of two links

Verification

Multi-step decomposition reasoning:

- Functional properties of self-timed systems may involve multi-step executions that are quite burdensome to establish directly.

Verification

Multi-step decomposition reasoning:

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

Verification

Multi-step decomposition reasoning:

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

Verification

Multi-step decomposition reasoning:

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

Induction:

- We apply induction to establishing loop invariants of **iterative circuits**, i.e., circuits with feedback loops in their dataflows.

Verification

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

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

Verification

Reasoning with highly non-deterministic behavior in iterative self-timed systems 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.

Verification

Reasoning with highly non-deterministic behavior in iterative self-timed systems 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.

Outline

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

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.

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

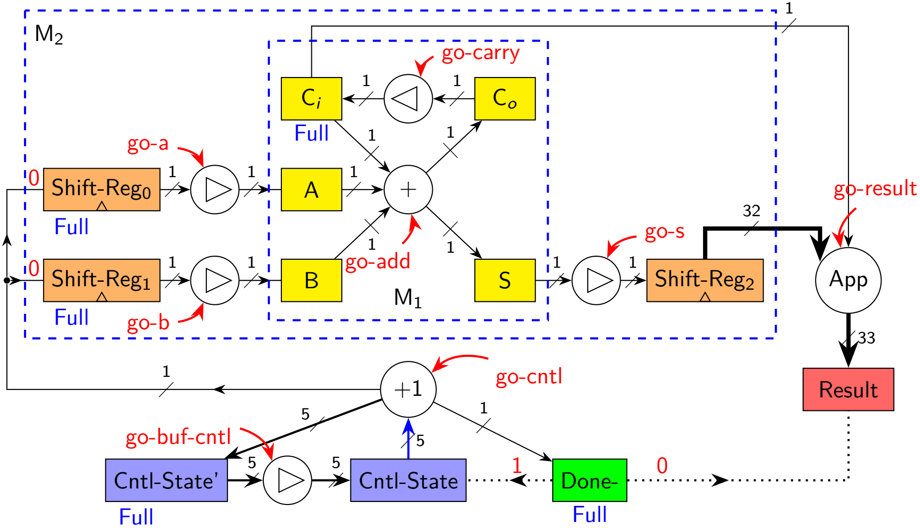
We prove that the self-timed serial adder indeed performs the addition under an appropriate initial condition.

- 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.

Multi-step decomposition 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.

Dataflow of a 32-Bit Self-Timed Serial Adder



Correctness Theorems

Theorem 1 (Partial correctness).

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

Correctness Theorems

Theorem 2 (Termination).

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

Outline

Future Work

We are developing new proof techniques for partial correctness of self-timed circuit designs that DO NOT have any conditions on the values of **go** signals.

- Our new method does not impose the aforementioned design restrictions on **loop-free** circuits.

Future Work

We are developing new proof techniques for partial correctness of self-timed circuit designs that DO NOT have any conditions on the values of **go** signals.

- Our new method does not impose the aforementioned design restrictions on **loop-free** circuits.

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

Future Work

We are developing new proof techniques for partial correctness of self-timed circuit designs that DO NOT have any conditions on the values of **go** signals.

- Our new method does not impose the aforementioned design restrictions on **loop-free** circuits.

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, multi-step decomposition reasoning, and induction.

References

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?