A Hierarchical Approach to Formal Modeling and Verification of Asynchronous Circuits

Cuong Chau
cckuong@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin

March 15, 2019
Outline

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

1. Overview
2. DE System
3. Modeling and Verification Approach
4. Case Studies
5. Future Work and Conclusions
Synchronous vs. Asynchronous

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**.
Motivation

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

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

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

Scalable methods for self-timed system verification are highly desirable.
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 general-purpose, self-timed circuit designs.

Impact:

- Advance the state-of-the-art in self-timed circuit specification and verification, and provide a means to support building reliable complex hardware systems using the self-timed paradigm; and thus,
- Support a computing paradigm where systems can proceed at their best rate and no longer require clock signals.
Goals and Impact

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 general-purpose, self-timed circuit designs.

Impact:

- Advance the state-of-the-art in self-timed circuit specification and verification, and provide a means to support building reliable complex hardware systems using the self-timed paradigm; and thus,
- Support a computing paradigm where systems can proceed at their best rate and no longer require clock signals.
Approach

Extend the **DE**-based, synchronous-style verification system\(^1\) to one that is capable of analyzing self-timed system models.


Approach

Extend the **DE-based**, synchronous-style verification system\(^1\) to one that is capable of analyzing self-timed system models.

Apply the **link-joint model**\(^2\) to modeling self-timed circuit designs.


Approach

Extend the DE-based, synchronous-style verification system\(^1\) to one that is capable of analyzing self-timed system models.

Apply the link-joint model\(^2\) to modeling self-timed circuit designs.

Develop a hierarchical (compositional) reasoning approach that is amenable to verifying correctness of large, non-deterministic systems without a large growth of the time complexity.

---


Approach

Extend the DE-based, synchronous-style verification system\textsuperscript{1} to one that is capable of analyzing self-timed system models.

Apply the link-joint model\textsuperscript{2} to modeling self-timed circuit designs.

Develop a hierarchical (compositional) reasoning approach that is amenable to verifying correctness of large, non-deterministic systems without a large growth of the time complexity.

- Avoid exploring the operations \textit{internal to a verified submodule} as well as their interleavings.
- The \textit{input-output relationship} of a verified submodule is determined based on the communication signals at the submodule’s input and output ports, while \textit{abstracting away all execution paths internal to that submodule}.


Accomplishments

Extended the **DE system** to modeling self-timed circuit designs.

Extended the **DE primitive database** with a new primitive that coordinates the means to update the state of a (storage) link.
Accomplishments

Extended the **DE system** to modeling self-timed circuit designs.

Extended the **DE primitive database** with a new primitive that coordinates the means to update the state of a (storage) link.

Developed a **hierarchical verification** approach that scales well even as circuit size increases.

Developed **lemma libraries** and **strategies** for reasoning about **non-deterministic** circuit behavior efficiently.
Accomplishments

Extended the **DE system** to modeling self-timed circuit designs.

Extended the **DE primitive database** with a new primitive that coordinates the means to update the state of a (storage) link.

Developed a **hierarchical verification** approach that scales well even as circuit size increases.

Developed **lemma libraries** and **strategies** for reasoning about **non-deterministic** circuit behavior efficiently.

Successfully applied our modeling and verification approach to a variety of self-timed circuit models.

- Data-loop-free circuits [2]
- Iterative circuits [1]
- Circuits involving non-deterministically arbitrated merges [1]
Outline

1. Overview

2. DE System

3. Modeling and Verification Approach

4. Case Studies

5. Future Work and Conclusions
DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines.
DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines.

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.

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 data stored in a communication link.
DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing **Mealy machines**.

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**.

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

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines.

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.

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 data stored in a communication link.
Outline

1. Overview
2. DE System
3. Modeling and Verification Approach
4. Case Studies
5. Future Work and Conclusions
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.
We model self-timed systems as **Mealy machines** representing networks of **communication links** and **computation joints**.

![Diagram of link-joint model]

*Links* communicate with each other locally via *joints* using the **link-joint model**.

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

![Diagram of Link-Joint Model]

Links communicate with each other locally via joints using the link-joint model.

- 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.
The green boxes represent instances of our new \textit{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.
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.
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.
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, $Q_2$
Verification Flow

1. Functional spec
2. Gate-level netlist
Verification Flow

- Functional spec
- Extraction level
- Four-valued level
- Gate-level netlist
Verification Flow

Functional spec

Extraction level

Four-valued level

Value and state lemmas, Multi-step state lemma

Gate-level netlist
Verification Flow

Functional spec

Extraction level

Single-step-update properties

Four-valued level

Value and state lemmas, Multi-step state lemma

Gate-level netlist
Verification Flow

Functional spec

Multi-step input-output relationship

Extraction level

Single-step-update properties

Four-valued level

Value and state lemmas, Multi-step state lemma

Gate-level netlist
Verification Steps

Hierarchical reasoning & induction

Single-step-update properties

Induction

step

Value and state lemmas

Induction

Multi-step input-output relationship

run

Multi-step state lemma

Functional correctness

C. Chau (UT Austin)
Outline

1. Overview
2. DE System
3. Modeling and Verification Approach
4. Case Studies
5. Future Work and Conclusions
A FIFO Circuit Model

Q3

\[ [1, 4, 3] \]

\[ \begin{array}{c}
\text{in} \\
\downarrow \\
8 \\
\downarrow \\
\times \\
\downarrow \\
5 \\
\downarrow \\
\text{out}
\end{array} \]

\[ [1, 4, 3] \mathbin{++} [8, 5] \]
A FIFO Circuit Model

Q3

\[ [1, 4, 3] \rightarrow 8 \rightarrow x \rightarrow 5 \rightarrow \text{out} \]

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

\[ [1] \rightarrow 1 \rightarrow x \rightarrow x \rightarrow \text{out} \]

\[ [1] ++ [4, 3, 8, 5] \]
A FIFO Circuit Model

Q3

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

[1] ++ [4, 3, 8, 5] = [1, 4, 3] ++ [8, 5]
A FIFO Circuit Model

The relationship between $Q3$’s $\text{in-seq}$ and $\text{out-seq}$.

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

$\text{in-seq}$ is the sequence of input data extracted from $\text{inputs-seq}$ that are accepted by $Q3$.

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 $st$. 

C. Chau (UT Austin)
Async Circuit Modeling and Verification
March 15, 2019 18 / 28
A FIFO Circuit Model

The relationship between Q3’s \textit{in-seq} and \textit{out-seq}.

\[
q3$extract(q3$run(inputs-seq, st, n)) \text{ ++ out-seq = in-seq ++ q3$extract(st)}
\]

\textit{in-seq} is the sequence of input data extracted from \textit{inputs-seq} that are accepted by Q3.

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

\textit{out-seq = in-seq} when the initial and final states contain no valid data.
A FIFO Circuit Model

\[ q_3^\text{extract}(q_3^\text{run}(\text{inputs-seq}, st, n)) \mathbin{++} \text{out-seq} = \]
\[ \text{in-seq} \mathbin{++} q_3^\text{extract}(st) \]  \hspace{1cm} (1)

Our ACL2 proof of (1) uses \textbf{induction} and the following \textbf{single-step-update} property of Q3 as a supporting lemma,

\[ q_3^\text{extract}(q_3^\text{step}(\text{inputs}, st)) = q_3^\text{extracted-step}(\text{inputs}, st) \]  \hspace{1cm} (2)
A FIFO Circuit Model

\[ q3\text{extract}(q3\text{run}(\text{inputs-seq}, \text{st}, n)) \mathrel{++} \text{out-seq} = \text{in-seq} \mathrel{++} q3\text{extract}(\text{st}) \]

(1)

Our ACL2 proof of (1) uses \textbf{induction} and the following single-step-update property of \( Q_3 \) as a supporting lemma,

\[ q3\text{extract}(q3\text{step}(\text{inputs}, \text{st})) = q3\text{extracted-step}(\text{inputs}, \text{st}) \]

(2)

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

\[
\begin{cases}
q3\text{extract}(\text{st}), & \text{if } \text{in-act} = \text{nil} \land \text{out-act} = \text{nil} \\
[\text{inputs.data}] \mathrel{++} q3\text{extract}(\text{st}), & \text{if } \text{in-act} = \text{t} \land \text{out-act} = \text{nil} \\
\text{remove-last}(q3\text{extract}(\text{st})), & \text{if } \text{in-act} = \text{nil} \land \text{out-act} = \text{t} \\
[\text{inputs.data}] \mathrel{++} \text{remove-last}(q3\text{extract}(\text{st})), & \text{otherwise}
\end{cases}
\]
A Greatest-Common-Divisor (GCD) Circuit Model

\[ \text{gcd-alg}(a, b) := \]
\[ \text{if } (a = 0) \text{ then } b \]
\[ \text{else if } (b = 0) \text{ then } a \]
\[ \text{else if } (a = b) \text{ then } a \]
\[ \text{else if } (a < b) \text{ then } \text{gcd-alg}(b - a, a) \]
\[ \text{else } \text{gcd-alg}(a - b, b) \]
Hierarchical Reasoning

The module’s functionality still preserves when replacing its submodules with **functionally equivalent** ones, without the need to rework proofs.
Arbitrated merge is a well-known self-timed circuit model that provides mutually exclusive access to a shared resource.

Produce non-deterministic output sequences due to arbitrary arrival times of requests.

We formalize an arbitrated merge joint that provides mutually exclusive access to its output link from its two input links on a first-come-first-served basis\(^3\).

Circuits Performing Arbitrated Merges

**interl**

\[ in_0 \rightarrow Q’_{20a} \rightarrow A \rightarrow out \]

\[ in_1 \rightarrow Q’_{20b} \rightarrow \text{arbitrated merge} \]

**igcd**

\[ in_0 \rightarrow \text{interl} \rightarrow L \rightarrow \text{gcd} \rightarrow out \]

\[ in_1 \rightarrow \text{interl} \rightarrow L \rightarrow \text{gcd} \rightarrow out \]

**comp-interl**

\[ in_0 \rightarrow \text{interl}_0 \rightarrow L_0 \rightarrow \text{interl}_2 \rightarrow out \]

\[ in_1 \rightarrow \text{interl}_0 \rightarrow L_0 \rightarrow \text{interl}_2 \rightarrow out \]

\[ in_2 \rightarrow \text{interl}_1 \rightarrow L_1 \rightarrow \text{interl}_2 \rightarrow out \]

\[ in_3 \rightarrow \text{interl}_1 \rightarrow L_1 \rightarrow \text{interl}_2 \rightarrow out \]
The multi-step input-output relationship is established using the membership relation \((\in)\) and the interleaving operation \((\otimes)\).

interl$extract_0$ and interl$extract_1$ extract valid data from two complex links $Q'_{20a}$ and $Q'_{20b}$, respectively.

**Let** $st_f := \text{interl}$run$(\text{inputs-seq}, st, n)$,

\[
\forall x \in (\text{interl}$extract_0$(st_f) \otimes \text{interl}$extract_1$(st_f)).
\]

\[
(x ++ \text{out-seq}) \in \left( (\text{in}_0$-seq $++ \text{interl}$extract_0$(st)) \otimes (\text{in}_1$-seq $++ \text{interl}$extract_1$(st)) \right)
\]
Outline

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

Implement a syntactic checker that detects the link-joint topology violation in self-timed circuit designs.

Enhance the effectiveness of our framework by increasing automation through the further introduction of macros.

Automate the proofs of value and state lemmas.

Apply our methodology to modeling self-timed microprocessors and verifying their functional properties.

Model and verify a self-timed version of the **FM9001** microprocessor.

Develop methods for analyzing mixed self-timed, synchronous circuits and systems.
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, mechanized methodology that is capable of verifying the functional correctness of self-timed circuit designs at scale.
1. **Cuong Chau**, Warren A. Hunt Jr., Matt Kaufmann, Marly Roncken, and Ivan Sutherland

*A Hierarchical Approach to Self-Timed Circuit Verification*

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

*Data-Loop-Free Self-Timed Circuit Verification*

3. **Cuong Chau**, Warren A. Hunt Jr., Marly Roncken, and Ivan Sutherland

*A Framework for Asynchronous Circuit Modeling and Verification in ACL2*
HVC 2017, pp. 3-18.

4. Marly Roncken, Ivan Sutherland, Chris Chen, Yong Hei, Warren Hunt Jr., and **Cuong Chau**, with Swetha Mettala Gilla, Hoon Park, Xiaoyu Song, Anping He, and Hong Chen

*How to Think about Self-Timed Systems*
Questions?
Abstracting two queues \((A_0 \rightarrow Q_2 \rightarrow A_1)\) and \((B_0 \rightarrow Q_3 \rightarrow B_1)\) as two complex links makes reasoning more efficient by reducing case splits in proving the invariant as well as the single-step-update property for \(RR\).

The verification time of \(RR\) is reduced from more than 32.5 minutes to 22 seconds by using the complex links.