A Hierarchical Approach to Self-Timed Circuit Verification

Cuong Chau\textsuperscript{1}, Warren A. Hunt Jr.\textsuperscript{1}, Matt Kaufmann\textsuperscript{1}, Marly Roncken\textsuperscript{2}, and Ivan Sutherland\textsuperscript{2}

\{ckcuong,hunt,kaufmann\}@cs.utexas.edu,
mroncken@pdx.edu, ivans@cecs.pdx.edu

\textsuperscript{1} The University of Texas at Austin

\textsuperscript{2} Portland State University

May 14, 2019
Motivation and Goals

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.
- **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.
Motivation and Goals

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.
- 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.
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\(^1\) 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.

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


Contributions

Extend our previous framework\(^3\) to model and verify circuit generators with parameterized data sizes.

Demonstrate that our verification framework is applicable to circuits with loops as well.

Formalize an (non-deterministically) arbitrated merge joint that provides mutually exclusive access to its output link from its two input links.

Develop strategies for verifying the functional correctness of self-timed circuits performing arbitrated merges.

---

Outline

1. DE System
2. Modeling and Verification Approach
3. Case Studies
4. Future Work and Conclusions
DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing finite-state machines.
DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing finite-state 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.
DE System

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing finite-state 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 finite-state 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 extended the DE primitive database with a new primitive that models the validity of data stored in a communication link.
Outline

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

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

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** 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 \textbf{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 \textbf{full};
- drain (possibly a subset of) the input links, leaving them \textbf{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.
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.
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 joint: a queue of length two, $Q_2$

Complex link
Verification Flow

Functional spec

Gate-level netlist
Verification Flow

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

1. Functional spec
2. Extraction level
3. Four-valued level
   - Value and state lemmas,
   - Multi-step state lemma
4. 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

1. **Functional spec**
2. **Multi-step input-output relationship**
3. **Extraction level**
4. **Single-step-update properties**
5. **Four-valued level**
6. **Value and state lemmas, Multi-step state lemma**
7. **Gate-level netlist**
Verification Steps

Hierarchical reasoning

Single-step-update properties

Multi-step input-output relationship

Value and state lemmas

Multi-step state lemma

Functional correctness

Induction step

Induction run

✓

✓

✓

✓
Verification Steps

Hierarchical reasoning

Single-step-update properties

Induction

Value and state lemmas

Induction

Multi-step input-output relationship

Run

Multi-step state lemma

Functional correctness

✓: automated
1. DE System

2. Modeling and Verification Approach

3. Case Studies

4. Future Work and Conclusions
A Greatest-Common-Divisor (GCD) Circuit Model

\[ \text{gcd-alg}(a, b) := \]
\[ \begin{cases} 
  b & \text{if } (a = 0) \text{ then} \\
  a & \text{else if } (b = 0) \text{ then} \\
  a & \text{else if } (a = b) \text{ then} \\
  \text{gcd-alg}(b - a, a) & \text{else if } (a < b) \text{ then} \\
  \text{gcd-alg}(a - b, b) & \text{else}
\end{cases} \]
Arbitrated merge, or arbiter, 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\(^4\).

\[
\begin{array}{c}
in_0 \\
\downarrow \\
M \\
\uparrow \\
in_1 \\
\rightarrow \\
out
\end{array}
\]

---

Circuits Performing Arbitrated Merges

\[ \text{interl} \]

\[ \text{igcd} \]

\[ \text{comp-interl} \]
<table>
<thead>
<tr>
<th>Circuit</th>
<th>Proof time</th>
<th># go signals</th>
<th># go signals affecting reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>8s</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Q5’</td>
<td>8s</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Q10’</td>
<td>3s</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Q20’</td>
<td>3s</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>Q40’</td>
<td>3s</td>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>interl</td>
<td>5s</td>
<td>81</td>
<td>3</td>
</tr>
<tr>
<td>igcd</td>
<td>12s</td>
<td>84</td>
<td>5</td>
</tr>
<tr>
<td>comp-interl</td>
<td>23s</td>
<td>243</td>
<td>9</td>
</tr>
</tbody>
</table>
Outline

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

Implement a syntactic checker that detects link-joint topology violations 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.

E.g., 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 developed a hierarchical, mechanized methodology that is capable of verifying the functional correctness of self-timed circuit designs at scale.

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 action with an external go signal that, when disabled, prevents that action from firing.

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

- Data-loop-free circuits
- Iterative circuits
- Circuits involving non-deterministically arbitrated merges
Questions?
Arbitrated Merge Verification

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 $Q'_40a$ and $Q'_40b$, respectively.

let $st_f :=$ interl$\_run$(inputs-seq, $st$, $n$),

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

$(x ++ \text{out-seq}) \in ((\text{in}_0\text{-seq} ++ \text{interl}_{extract_0}(st)) \otimes$

$(\text{in}_1\text{-seq} ++ \text{interl}_{extract_1}(st))$)