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

May 22, 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 communication between state-holding elements is 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 communication between state-holding elements is 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 communication between state-holding elements is performed via **local communication protocols**.

Why asynchronous?

- Low power consumption,
- High operating speed,
- Low electromagnetic interference,
- Better composability and modularity in 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, a universal communication model for various 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, a universal communication model for various 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.

The Link-Joint Model

We model self-timed systems as finite-state-machine representations of networks of links communicating 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-machine representations of networks of links communicating 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**.

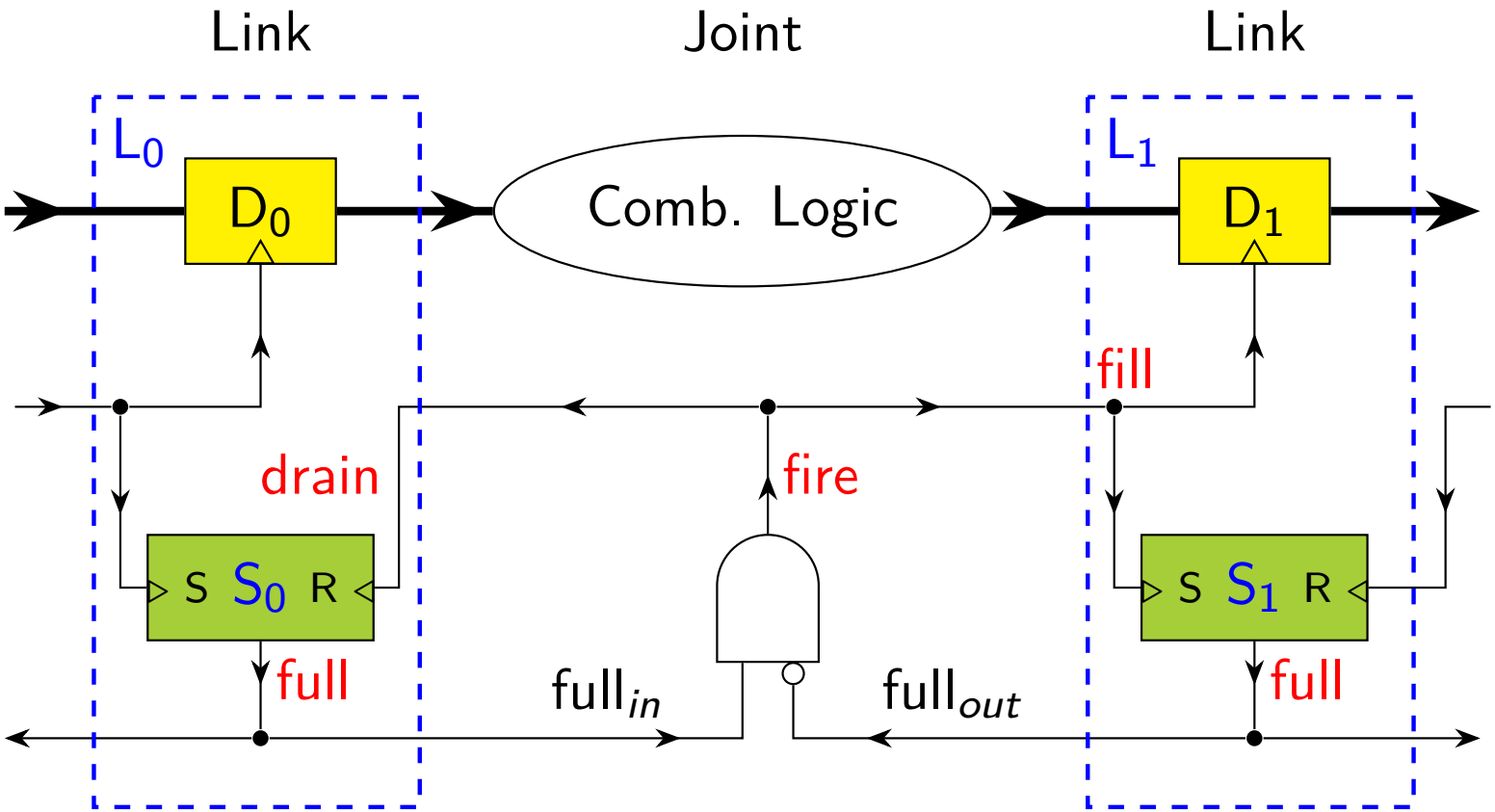
The Link-Joint Model

We model self-timed systems as finite-state-machine representations of networks of links communicating with each other locally via **handshake components**, which are called joints, using the link-joint model.

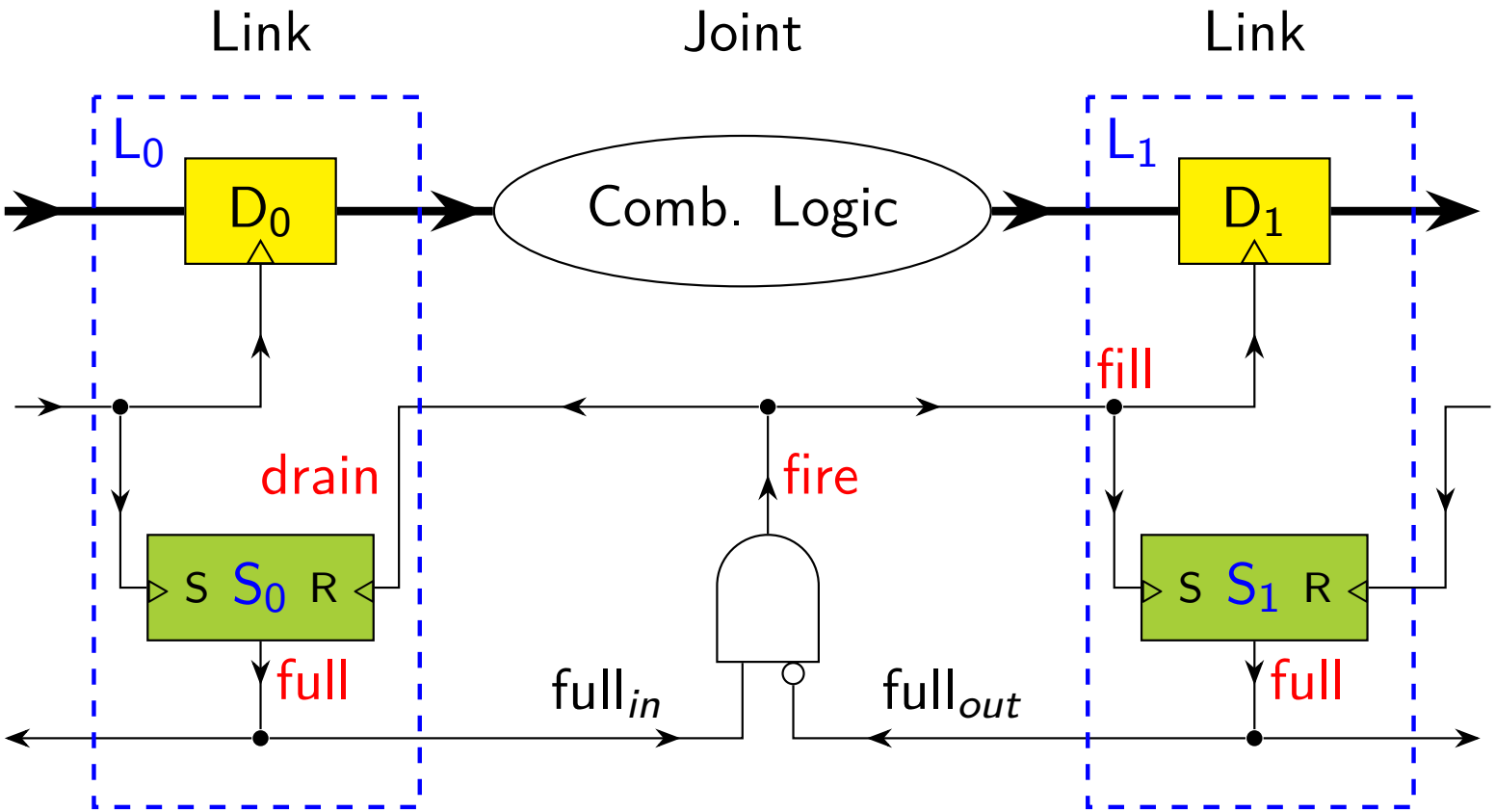
- Links are communication channels in which **data** and **full/empty states** are stored.
- Joints are handshake components that implement **flow control** and **data operations**.

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

The Link-Joint Model



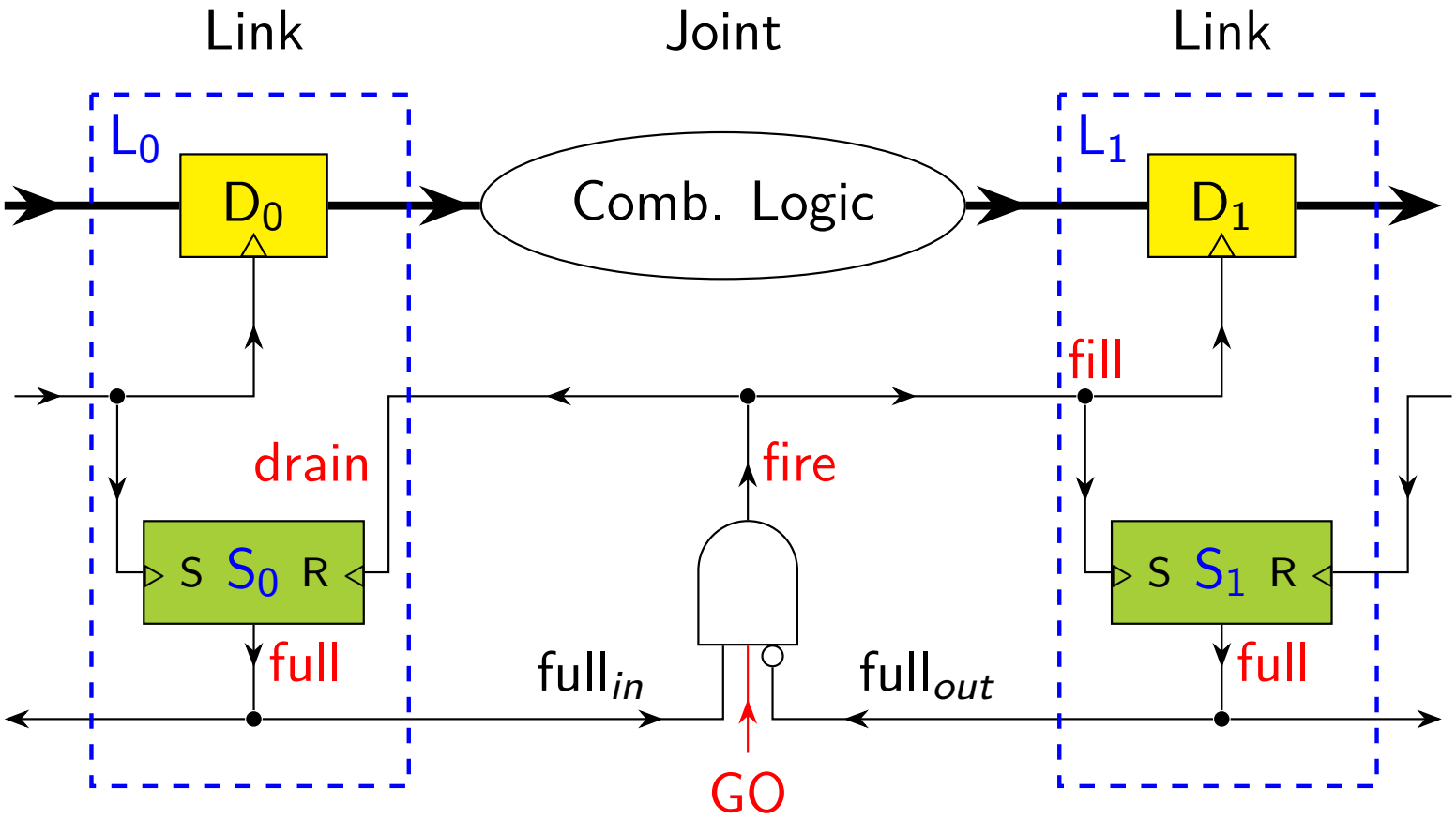
The Link-Joint Model



A joint can have several incoming and outgoing links connected to it.

Necessary conditions for a joint to fire: all of its incoming links are **full** and all of its outgoing links are **empty**.

The Link-Joint Model



A joint can have several incoming and outgoing links connected to it.

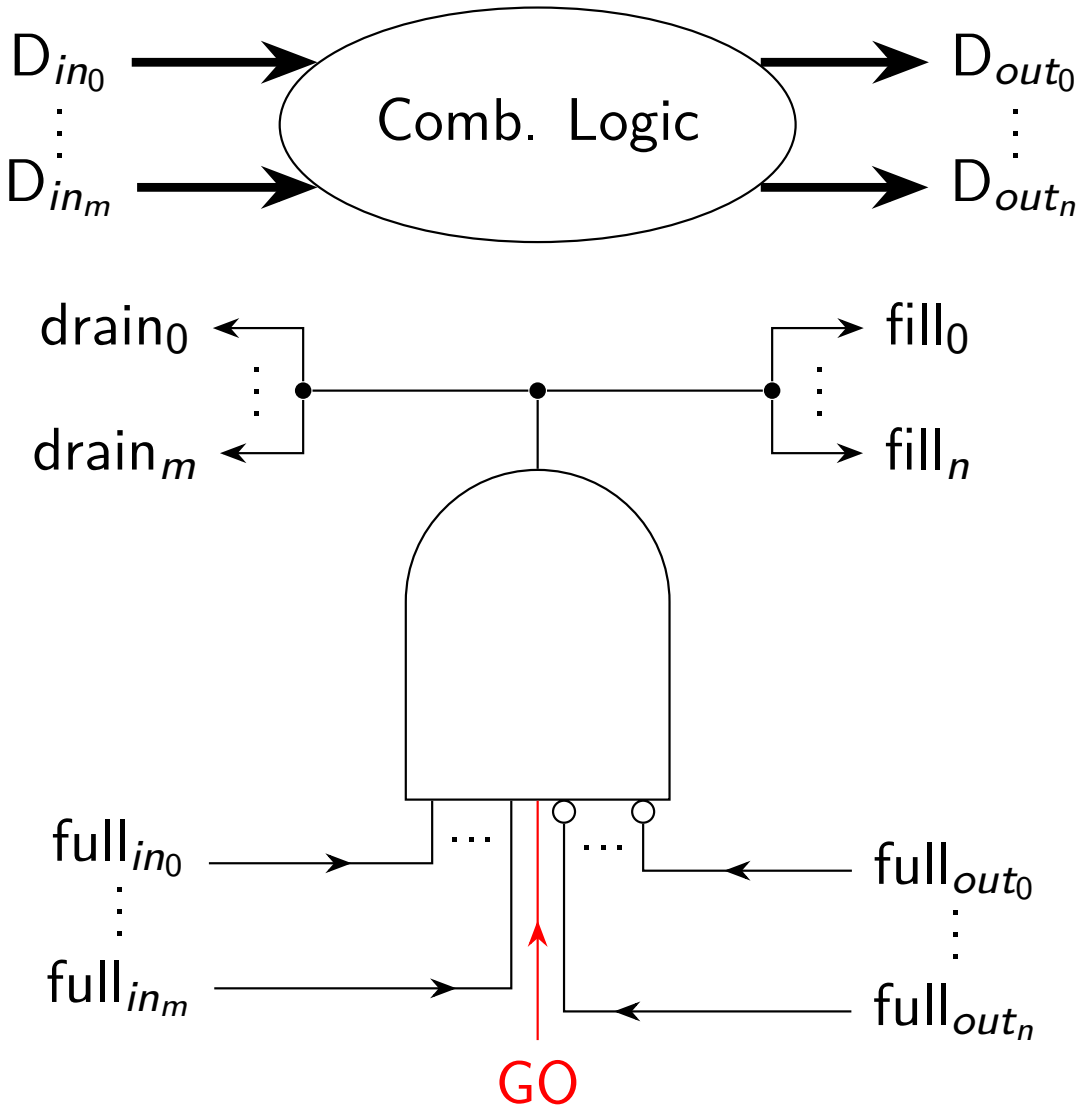
Necessary conditions for a joint to fire: all of its incoming links are **full** and all of its outgoing links are **empty**.

The Link-Joint Model

When a joint fires, the following three actions will be executed in parallel:

- transfer data computed from the incoming links to the outgoing links,
- fill the outgoing links, make them **full**,
- drain the incoming links, make them **empty**,

The Link-Joint Model



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.

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.

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

- Computing invariance properties in self-timed systems becomes much more complicated than in synchronous systems.

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.

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

- Computing invariance properties in self-timed systems becomes much more complicated than in synchronous systems.

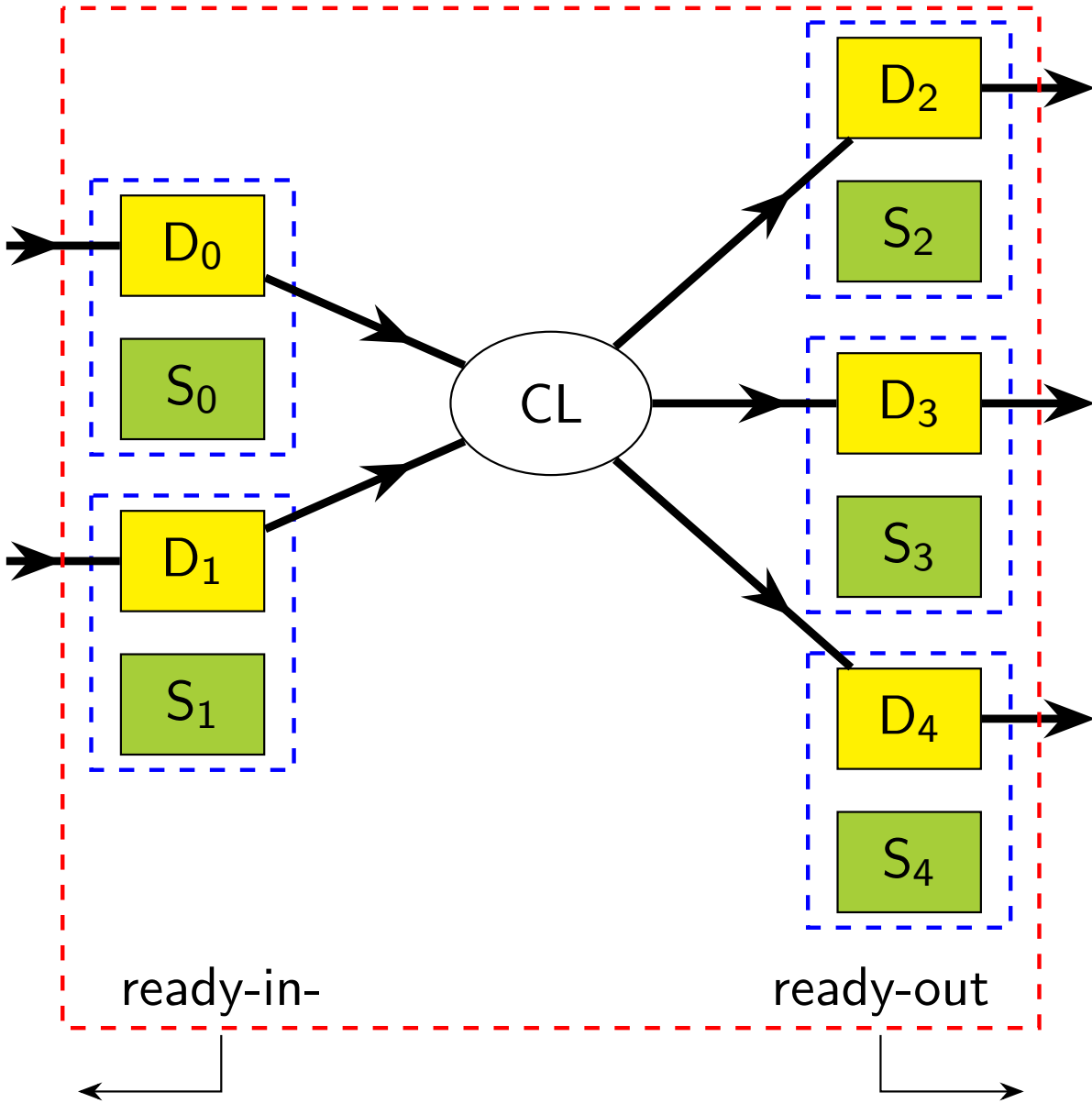
We impose design restrictions to reduce non-determinism, and consequently reduce the complexity of the set of execution paths:

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

Self-Timed Modules

Self-timed modules can be treated either as links or joints.

Our framework currently treats modules as "complex" links.

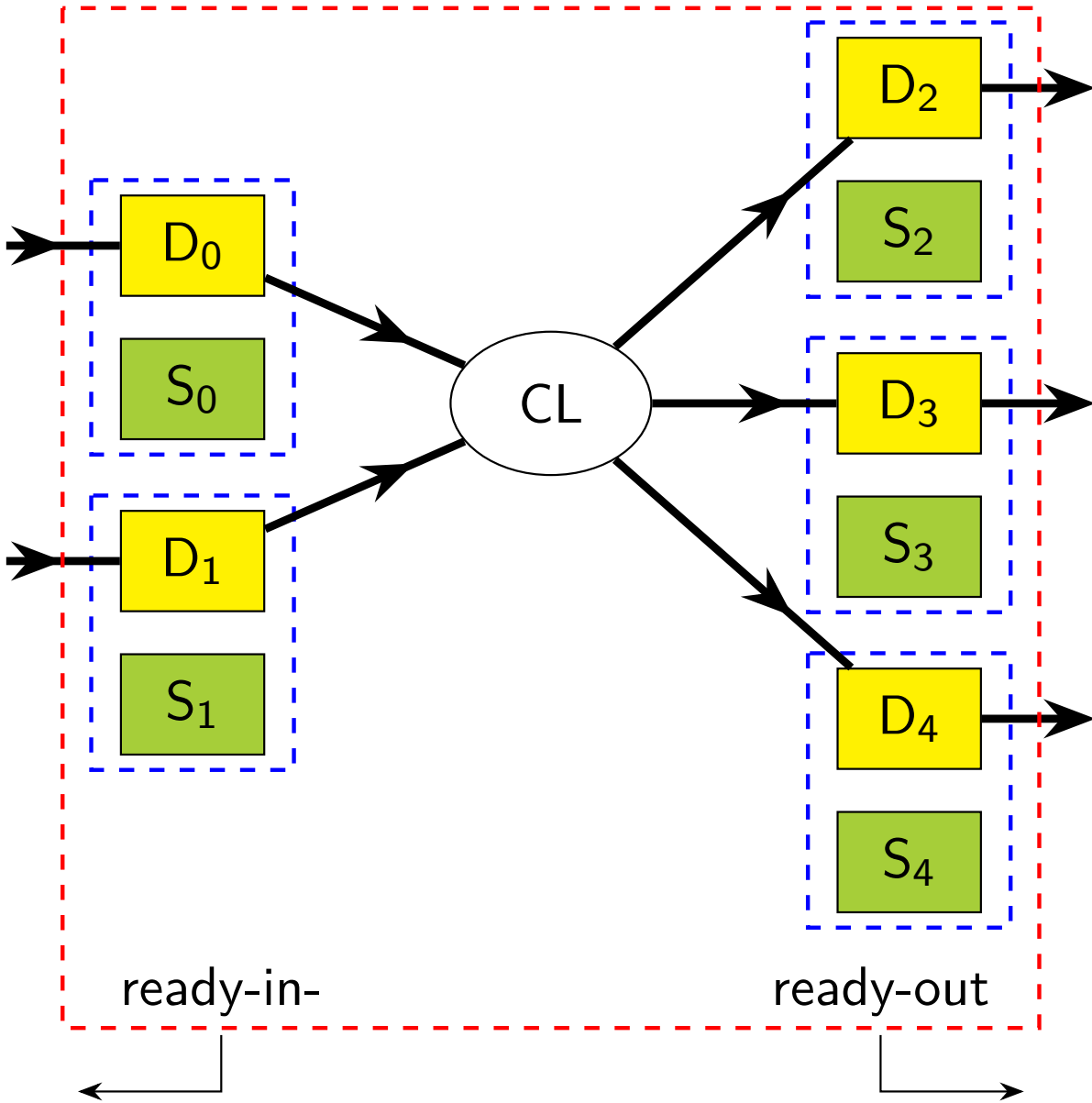


Self-Timed Modules

Self-timed modules can be treated either as links or joints.

Our framework currently treats modules as "complex" links.

$\Rightarrow$ Self-timed modules also report both **data** and **communication states** to the joints connecting them.

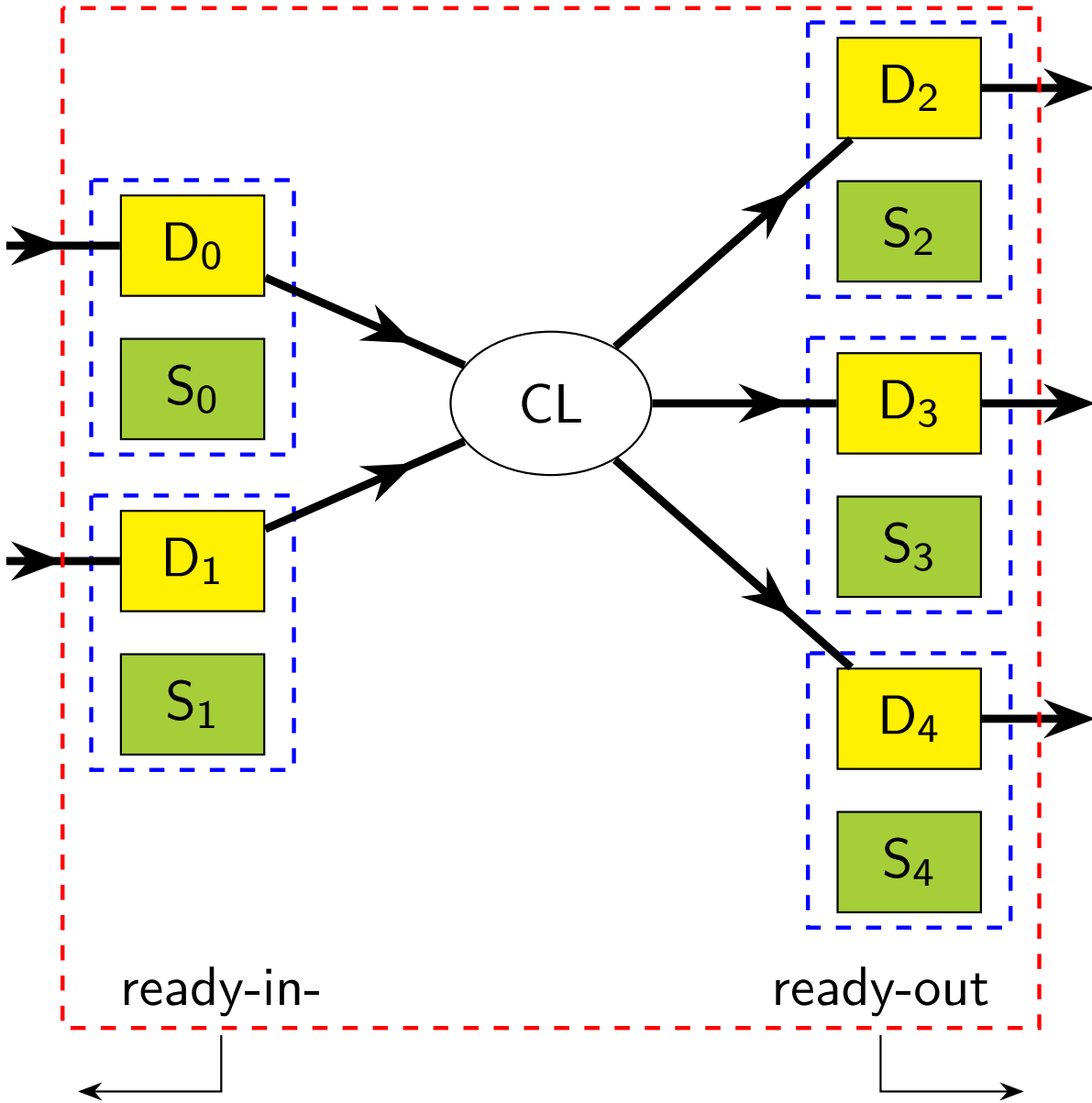


Self-Timed Modules

Self-timed modules can be treated either as links or joints.

Our framework currently treats modules as "complex" links.
$\Rightarrow$ Self-timed modules also report both **data** and **communication states** to the joints connecting them.

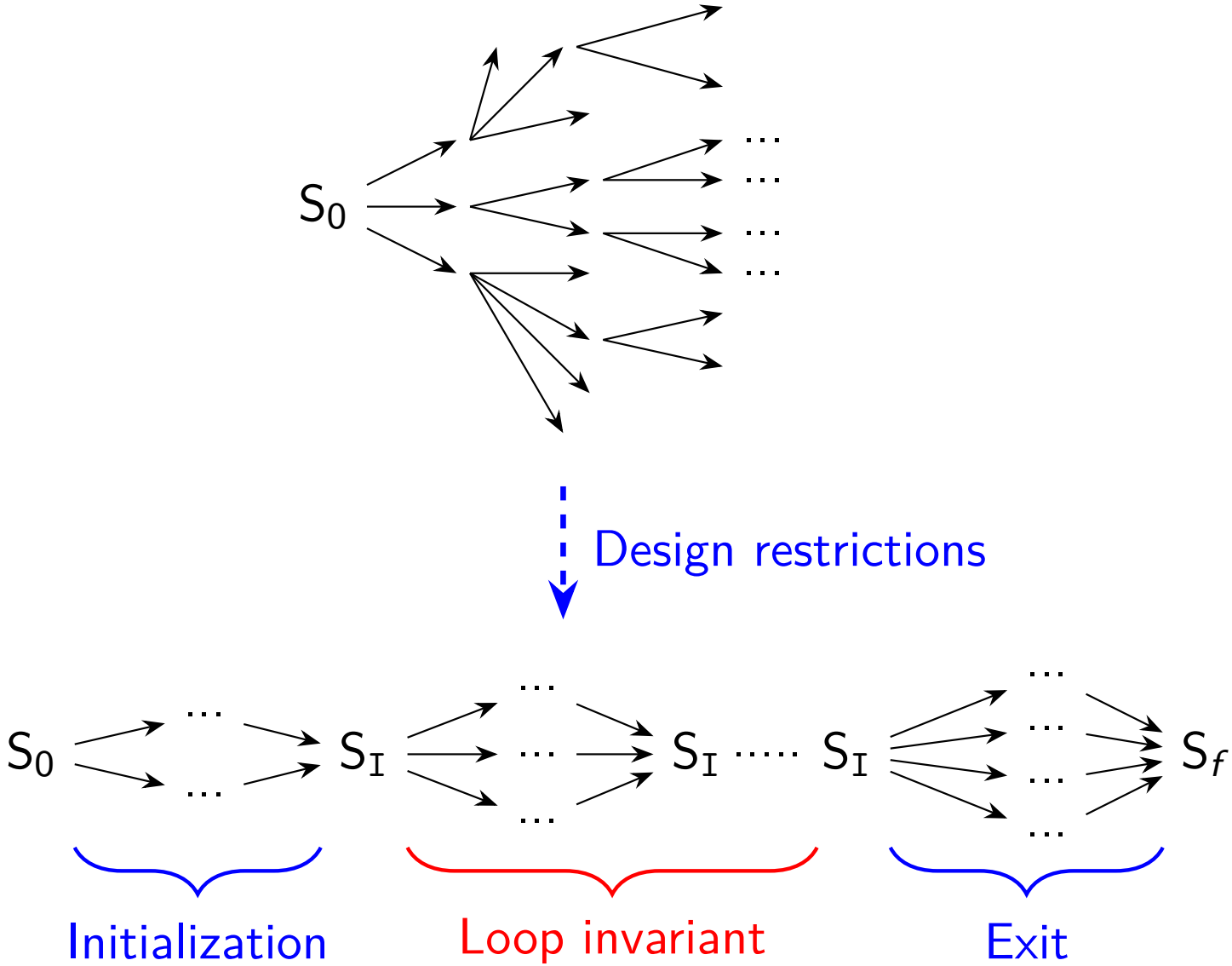
We plan to explore a notion of modules being treated as "complex" joints in the future.



Design Restrictions

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

State Space Reduction



S_0: initial state, S_I: invariant state, S_f: final state

Outline

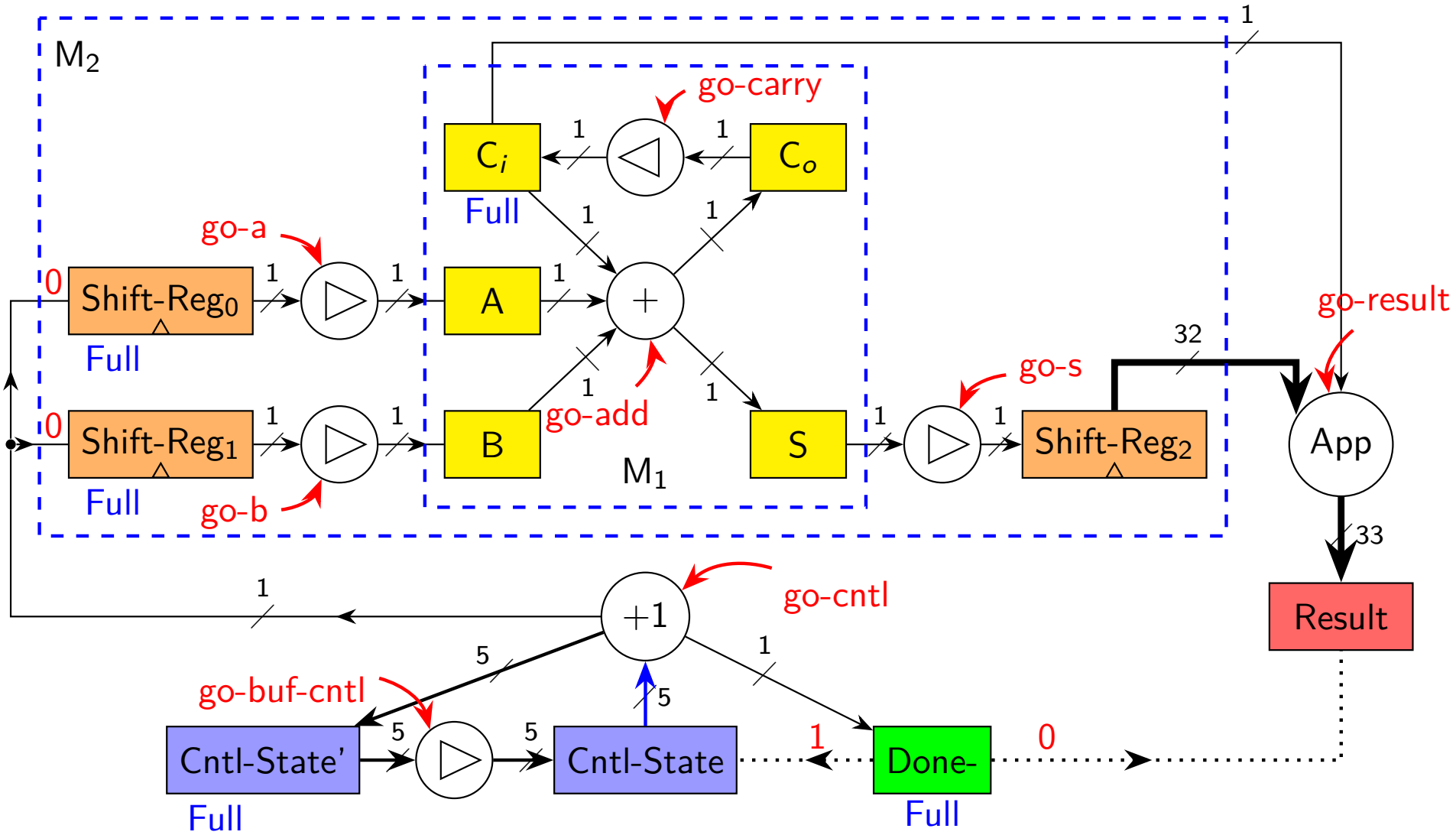
32-Bit Self-Timed Serial Adder Verification

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

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.

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



Correctness Theorems

Theorem 1 (Partial correctness).

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

Correctness Theorems

Theorem 2 (Termination).

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

Outline

Future Work

We plan to prove the partial correctness of the self-timed serial adder without specifying the interleavings of the go signals' values. In other words, we aim to remove **Hypothesis 4** from Theorem 1.

Future Work

We plan to prove the partial correctness of the self-timed serial adder without specifying the interleavings of the go signals' values. In other words, we aim to remove **Hypothesis 4** from Theorem 1.

For the termination theorem (Theorem 2), simply removing **Hypothesis 4** will make the theorem invalid.

- We need to add a constraint guaranteeing that delays are bounded in order to prove Theorem 2 without having Hypothesis 4.

Future Work

We plan to prove the partial correctness of the self-timed serial adder without specifying the interleavings of the go signals' values. In other words, we aim to remove **Hypothesis 4** from Theorem 1.

For the termination theorem (Theorem 2), simply removing **Hypothesis 4** will make the theorem invalid.

- We need to add a constraint guaranteeing that delays are bounded in order to prove Theorem 2 without having Hypothesis 4.

We also plan to investigate a notion of **modules with joints** at the interfaces, where two modules are connected by one or more external links.

Future Work

We plan to prove the partial correctness of the self-timed serial adder without specifying the interleavings of the go signals' values. In other words, we aim to remove **Hypothesis 4** from Theorem 1.

For the termination theorem (Theorem 2), simply removing **Hypothesis 4** will make the theorem invalid.

- We need to add a constraint guaranteeing that delays are bounded in order to prove Theorem 2 without having Hypothesis 4.

We also plan to investigate a notion of **modules with joints** at the interfaces, where two modules are connected by one or more external links.

We intend to follow a **hierarchical approach** to prove module-level properties 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.

Conclusions

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

We model a self-timed system as a network of links communicating with each other locally via handshake components, which are called joints, using the link-joint model.

Conclusions

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

We model a self-timed system as a network of links communicating with each other locally via handshake components, which are called joints, using the link-joint model.

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

Conclusions

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

We model a self-timed system as a network of links communicating with each other locally via handshake components, which are called joints, using the link-joint model.

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

Our verification framework is able to establish loop invariants using induction when the circuit behavior obeys the design restrictions we propose.

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)

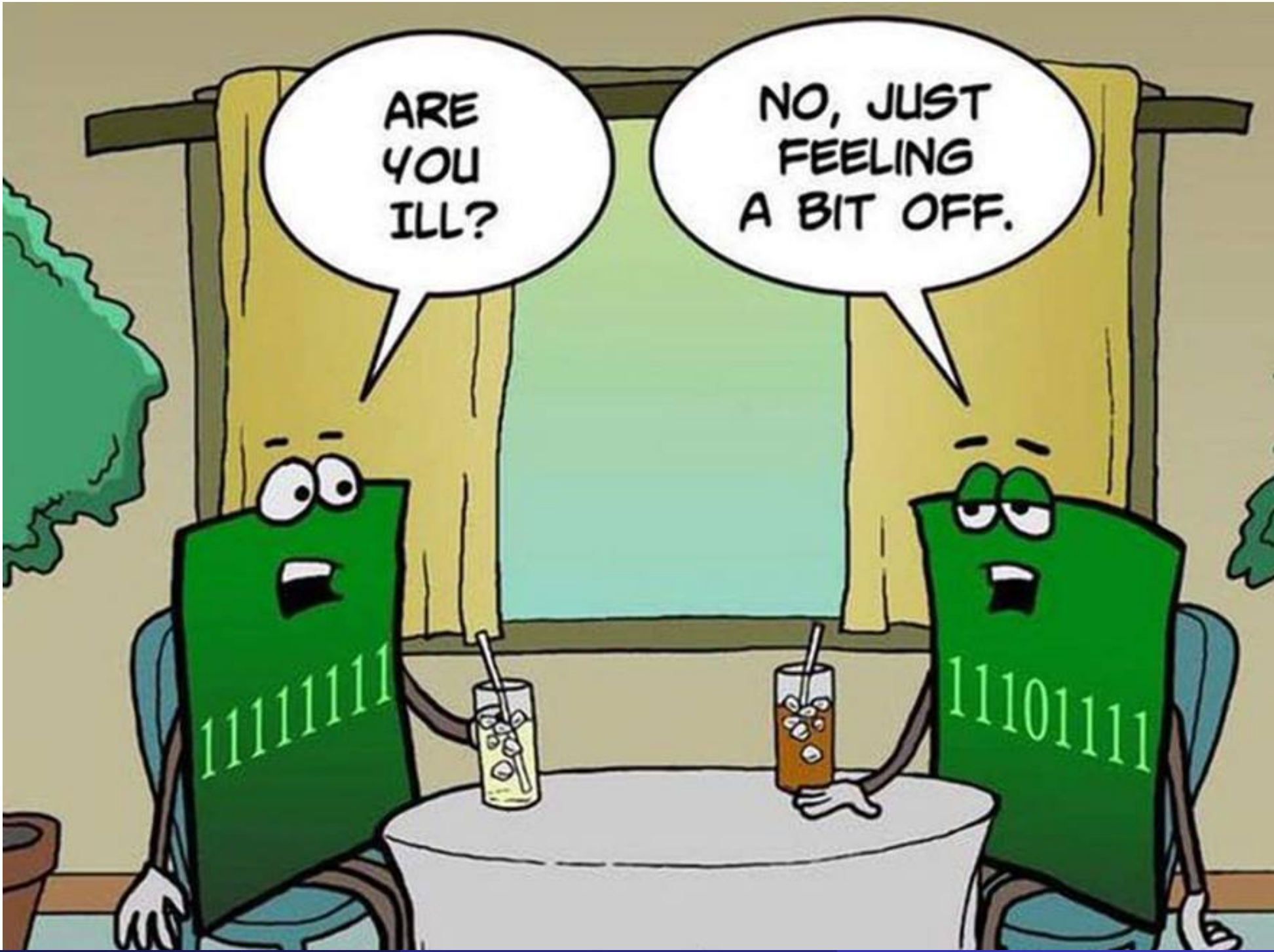
Naturalized Communication and Testing

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.


ARE YOU ILL?
NO, JUST FEELING A BIT OFF.
11111111
11101111

Questions?