Verifying Timed, Asynchronous Circuits Using ACL2

Yan Peng[1] Mark R. Greenstreet[1]

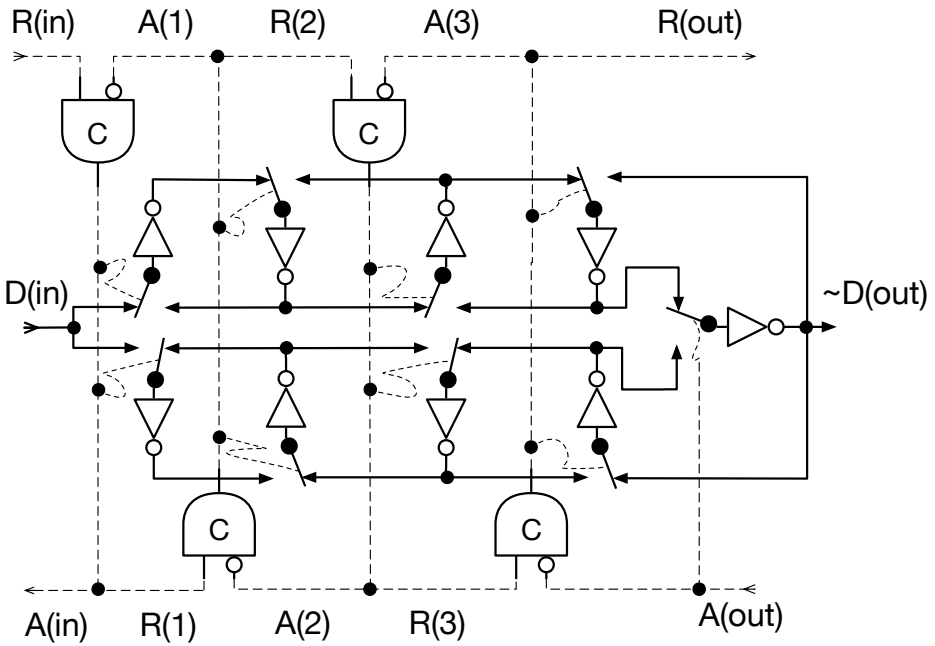
[1]Department of Computer Science
University of British Columbia

May 14th 2019

We verify timed circuits with the generality of ACL2 while achieving performance comparable to dedicated tools by using Smtlink, a SMT solver interface

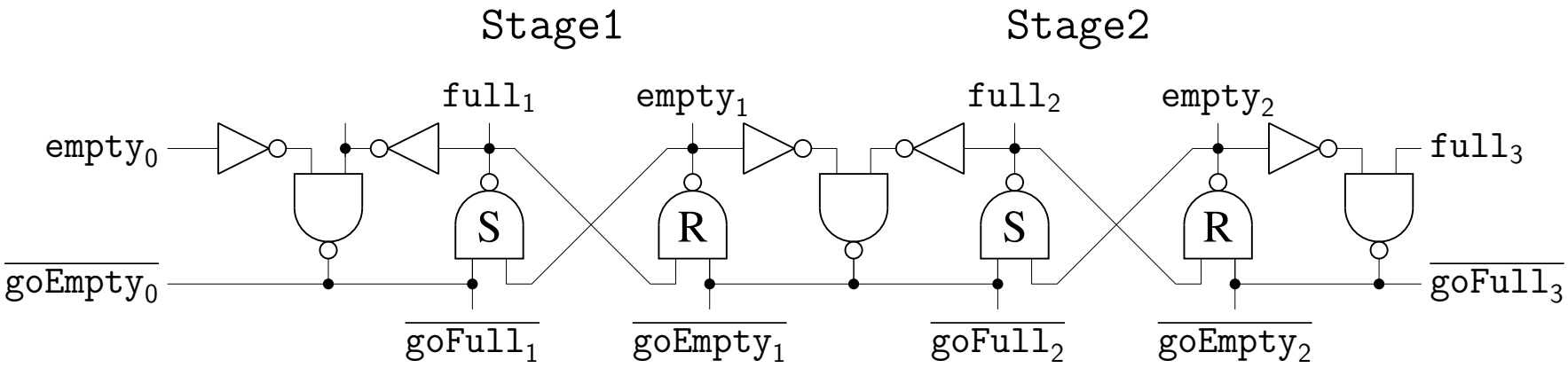
Timed Asynchronous Circuits are Great

- Timed asynchronous circuits exploit verifiable timing relationships in a circuit to improve performance and reduce power and area.
- Many timed pipeline designs have been proposed: micropipeline[Sut89], asP* pipelines[MJ+97], GasP[SF01], Mousetrap[SN07].



Timed Asynchronous Circuits are Challenging

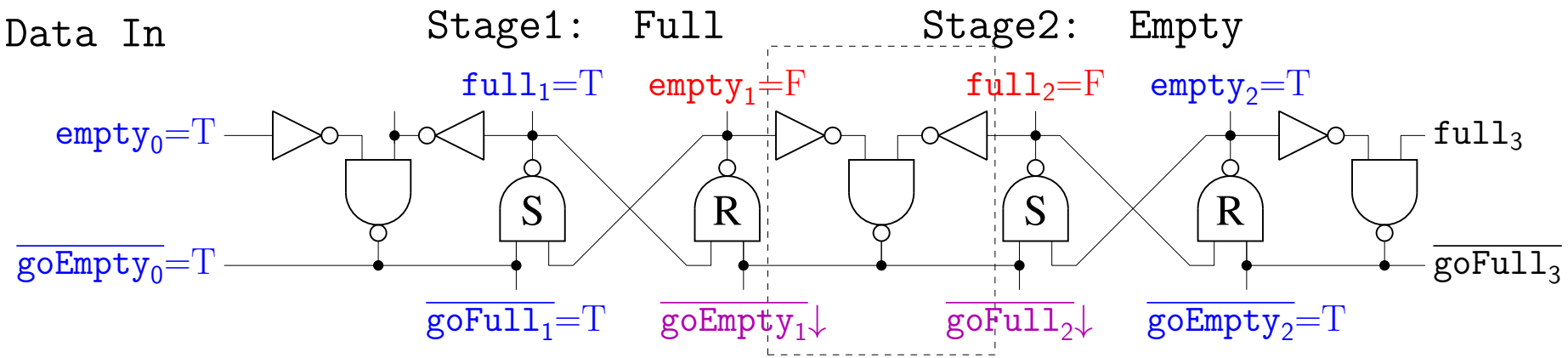
Let's look at the example of an asP* pipeline.



$\texttt{full}_i$ asserts the stage is full and $\texttt{empty}_i$ asserts the stage is empty.

Timed Asynchronous Circuits are Challenging

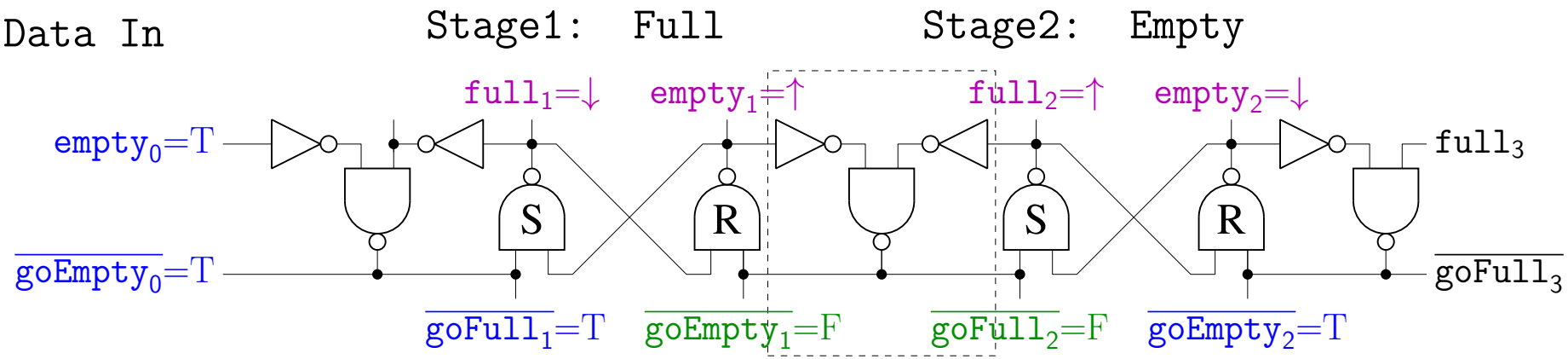
Let's look at the example of an asP* pipeline.



When the first stage is full, $\overline{\texttt{goEmpty}_1}$ and $\overline{\texttt{goFull}_2}$ are excited to go low.

Timed Asynchronous Circuits are Challenging

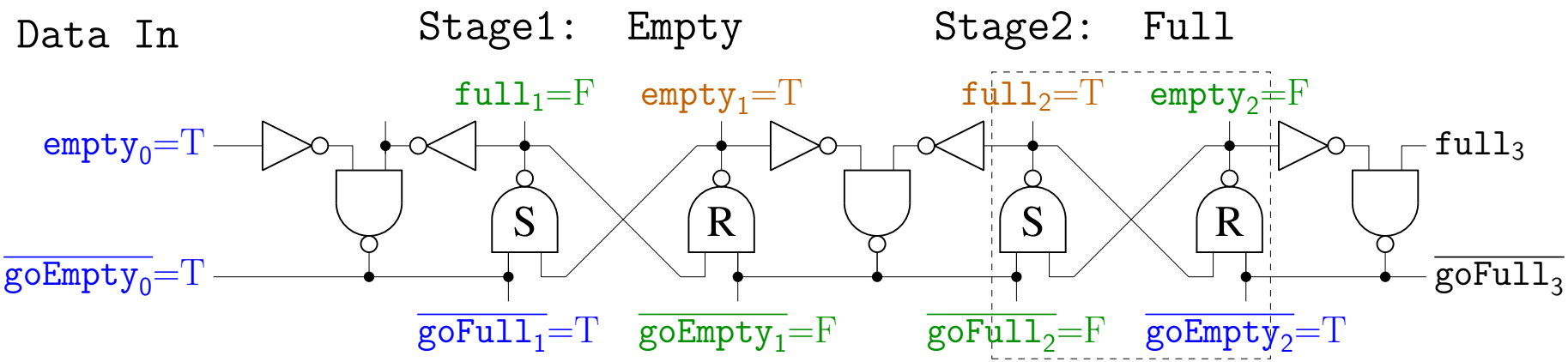
Let's look at the example of an asP* pipeline.



The SR-flops for stages 1 & 2 are both excited to change.
The pipeline transfers data from stage 1 to stage 2.

Timed Asynchronous Circuits are Challenging

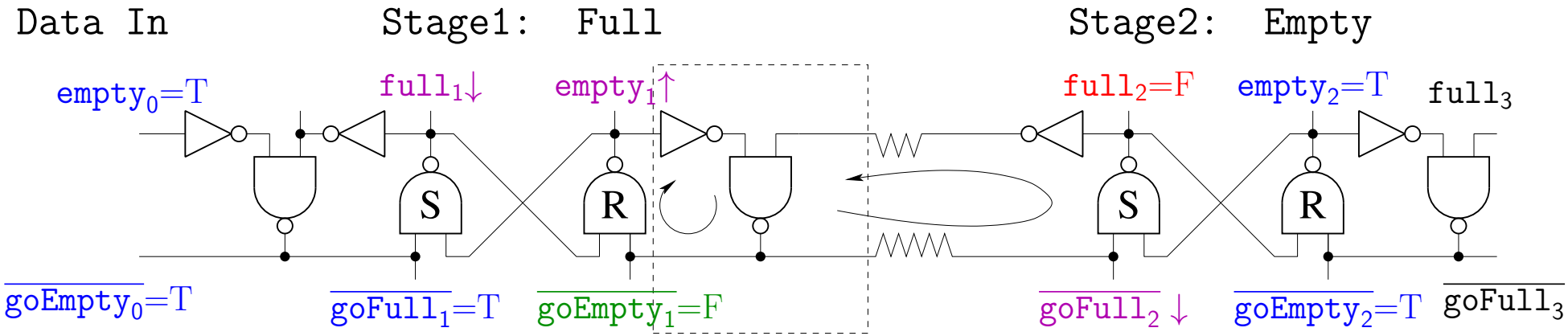
Let's look at the example of an asP* pipeline.



The first stage is now empty and the second stage full.
The data ripples through the pipeline.

Timed Asynchronous Circuits are Challenging

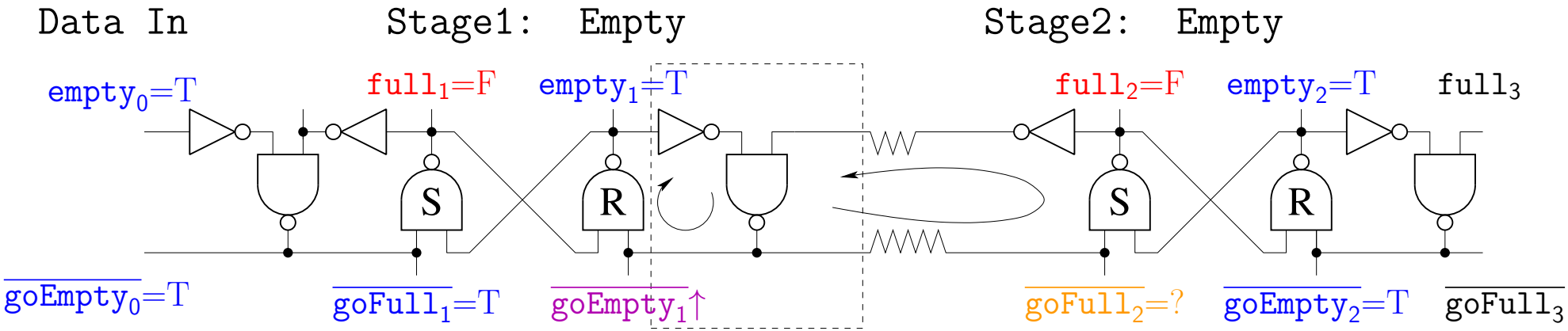
What can go wrong when timing constraints are violated?



Let's consider a scenario where the turn-around on the right of the NAND gate is significant longer than the left of it, for example, due to long wires in the layout.

Timed Asynchronous Circuits are Challenging

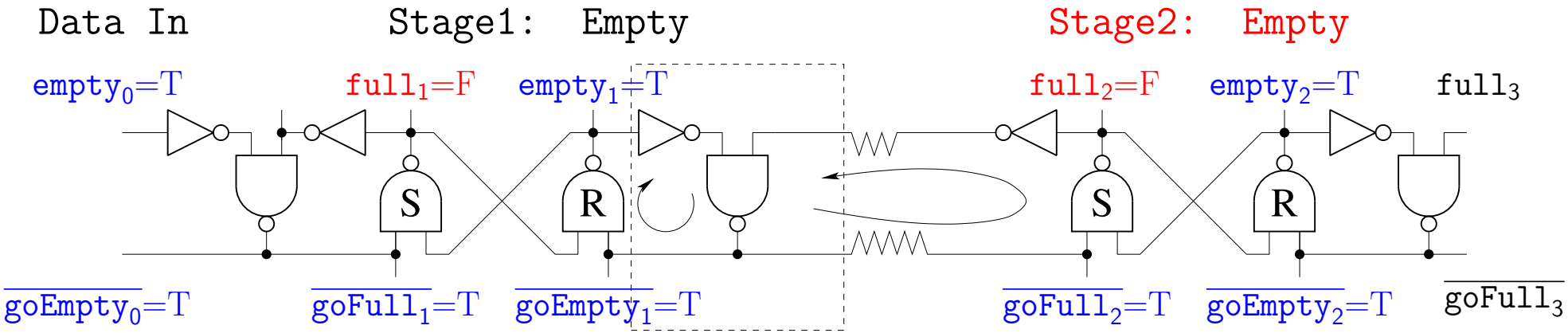
What can go wrong when timing constraints are violated?



$\texttt{empty}_1$ goes high, enabling $\overline{\texttt{goFull}_2}$ to return high before $\texttt{full}_2$ has a chance to change.

Timed Asynchronous Circuits are Challenging

What can go wrong when timing constraints are violated?



Now, $\texttt{full}_1$ has gone low, but $\texttt{full}_2$ never went high.
This disables stage two from going full.
A data value is lost. ☹

Timed Asynchronous Circuits are Challenging

Timing properties are crucial for correctness of timed asynchronous circuits. But how do we know we've defined enough timing constraints?

As is pointed out in [MJ+97]:

> *For the circuit presented here, determining the delay conditions that must be satisfied for reliable operation was difficult. We have used a mixture of ad hoc analysis and hSpice simulation to achieve our results ...*

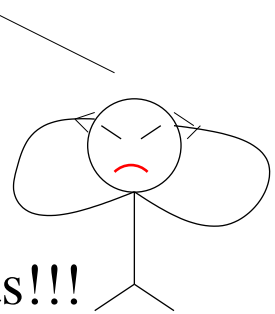
$$\overline{\texttt{goEmpty}_1}\downarrow \mapsto \texttt{empty}_1\uparrow \prec \overline{\texttt{goEmpty}_1}\uparrow$$

$$\overline{\texttt{goEmpty}_1}\downarrow \mapsto \texttt{full}_2\uparrow \prec \overline{\texttt{goEmpty}_1}\uparrow$$

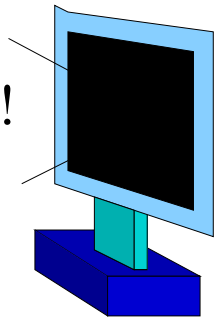
$$(\texttt{empty}_1\downarrow \wedge \texttt{full}_2\downarrow) \mapsto (\texttt{empty}_1\uparrow \wedge \texttt{full}_2\uparrow) \prec (\texttt{empty}_1\downarrow \wedge \texttt{full}_2\downarrow)$$

...

aaaaaaaahhhhhh My brain hurts!!!



Proved!

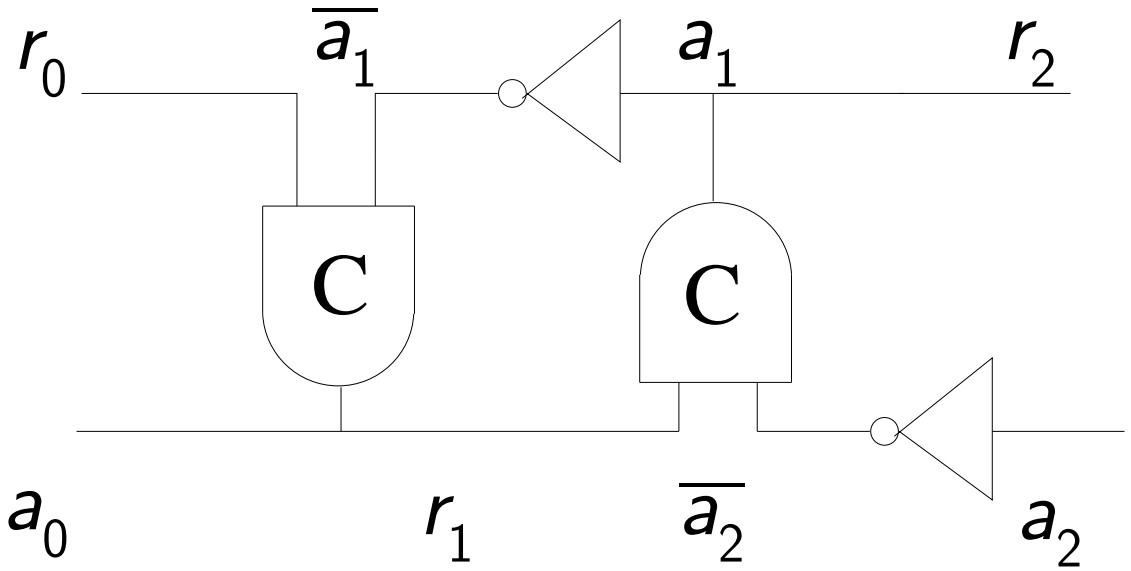


Contributions

- We propose a framework for modeling timed asynchronous circuits using *timed traces* and *trace recognizers*. It naturally models **nondeterminism**.
- We verified timing properties of three configurations of *asP* pipelines*, for each configurations:
  - We defined and proved the inductive **timing invariants**.
  - Using the invariants, we are able to prove **hazard-freedom**.
- Our model can handle **loop structures** and allows **parameterized verification**.

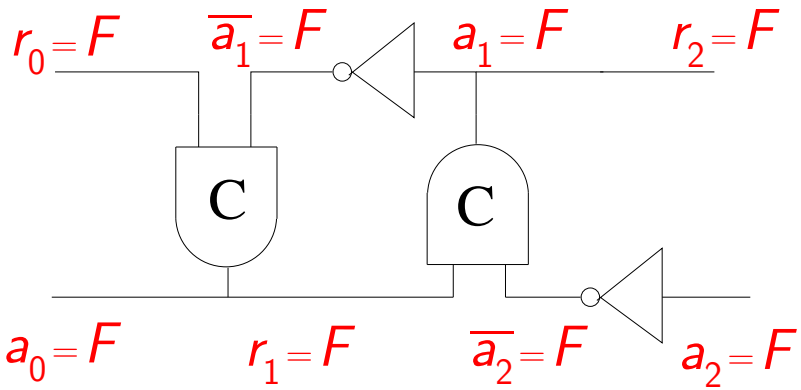
We verify timed circuits with the generality of ACL2 while achieving performance comparable to dedicated tools by using Smtlink, a SMT solver interface

Traces and Trace Recognizers



Every circuit contains a set of signal paths, representing nodes in the circuit.

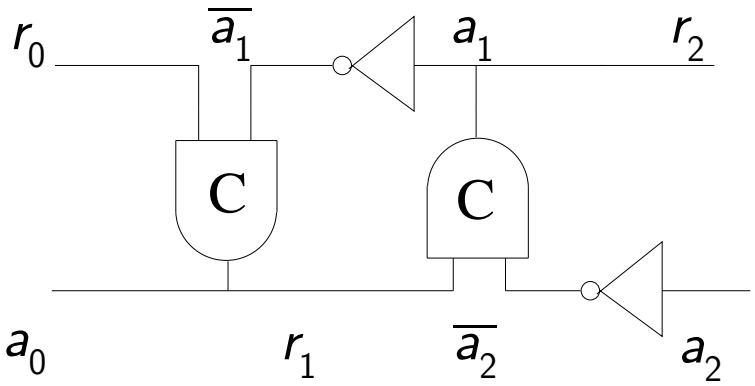
Traces and Trace Recognizers



r_0:F	a_0:F	r_1:F	a_1:F	$\overline{a_1}$:F	r_2:F	a_2:F	$\overline{a_2}$:F	...

- A state of the circuit is a table mapping from node paths to their values.
- A state can contain other signals that are outside of the component, allowing composability.

Traces and Trace Recognizers

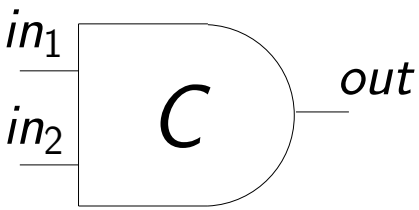


r_0:F	a_0:F	r_1:F	a_1:F	$\overline{a_1}$:F	r_2:F	a_2:F	$\overline{a_2}$:F	...
r_0:**T**	a_0:F	r_1:F	a_1:F	$\overline{a_1}$:**T**	r_2:F	a_2:F	$\overline{a_2}$:F	...
r_0:T	a_0:**T**	r_1:**T**	a_1:F	$\overline{a_1}$:T	r_2:F	a_2:F	$\overline{a_2}$:F	...
...	...	...	...	...	...	...	...	...

A trace of the circuit is a list of states.

Traces and Trace Recognizers

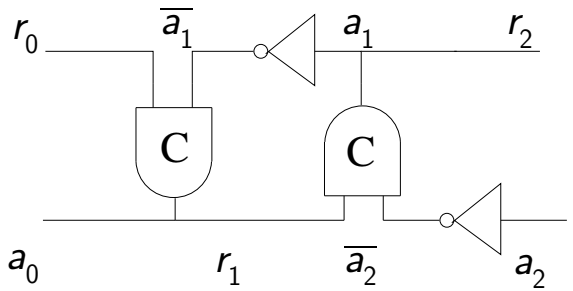
Trace recognizers are functions defined for each component. It specifies that for this component, every two consecutive states in the trace must make valid transitions:



$$\texttt{c\_step(prev,next)} \triangleq$$
$$\texttt{prev[out]} = \texttt{next[out]}$$
$$\vee\big(\texttt{prev[in1]} = \texttt{prev[in2]} \wedge \texttt{prev[out]} \neq \texttt{prev[in1]}\big)$$

- For example, for the component C-element,
  - Either the output hasn't changed, making no assumption about the inputs.
  - Or the output changes only when the inputs match and the output doesn't.
- The trace recognizers are receptive to all possible inputs[Dil88].
- It defines allowed behaviour on the outputs based on the inputs.

Introducing Time - Traces

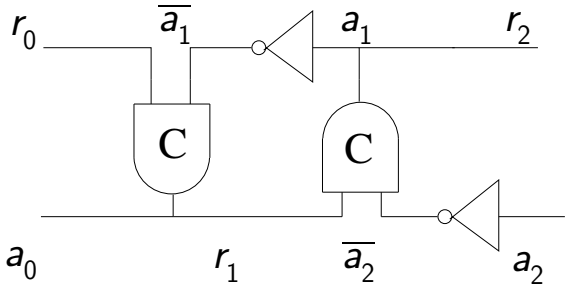


0	r_0:F,0	a_0:F,0	r_1:F,0	a_1:F,0	$\overline{a_1}$:F,0	...
0.3	r_0:**T,0.3**	a_0:F,0	r_1:F,0	a_1:F,0	$\overline{a_1}$:F,0	...
4	r_0:T,0.3	a_0:F,0	r_1:F,0	a_1:F,0	$\overline{a_1}$:**T,4**	...
10	r_0:T,0.3	a_0:**T,10**	r_1:**T,10**	a_1:F,0	$\overline{a_1}$:T,4	...
...	...	...	...	...	...	...

We extend the traces to include timing information.

- Add a time for each state.
- For each node signal in a state, we associate its value with a time representing the most recent time the signal acquired that value.

Introducing Time - Traces



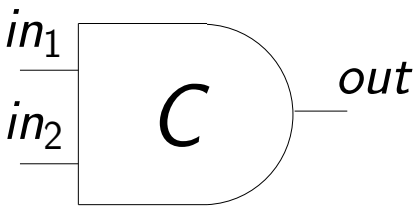
0	r_0:F,0	a_0:F,0	r_1:F,0	a_1:F,0	$\overline{a_1}$:F,0	...
0.3	r_0:**T,0.3**	a_0:F,0	r_1:F,0	a_1:F,0	$\overline{a_1}$:F,0	...
4	r_0:T,0.3	a_0:F,0	r_1:F,0	a_1:F,0	$\overline{a_1}$:**T,4**	...
10	r_0:T,0.3	a_0:**T,10**	r_1:**T,10**	a_1:F,0	$\overline{a_1}$:T,4	...
...	...	...	...	...	...	...

A few notes about time:

- Time is non-negative and monotonically increasing.
- Time of each signal must not be larger than the state time.
- Every trace can be extended so that time progresses without bound ("non-Zenoness" [AL94]).

Introducing Time - Trace Recognizers

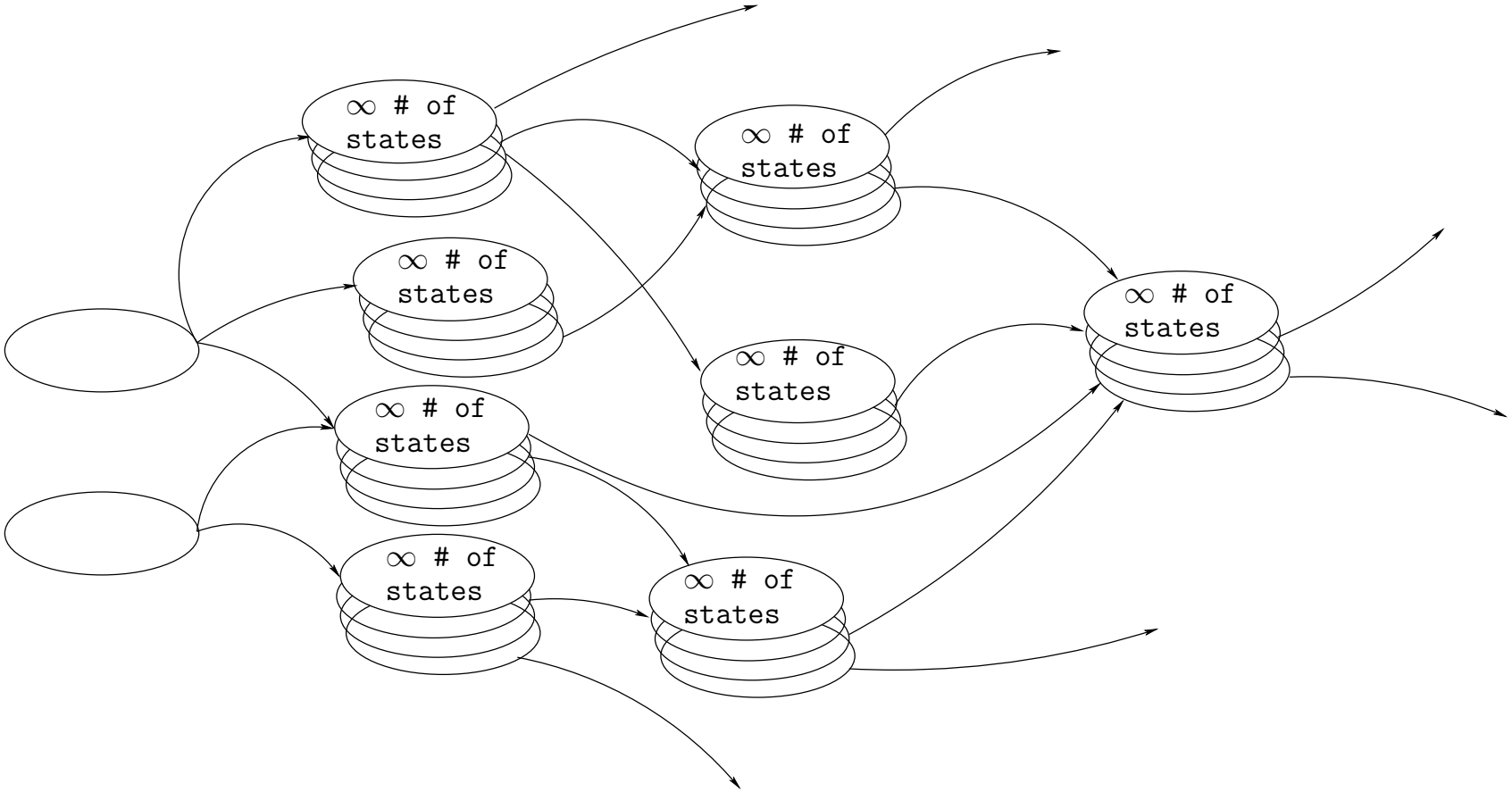
For each component, we use the delay model of a symbolic delay bound $[\delta_{lo}, \delta_{hi})$ where $\delta_{lo} > 0$ and $\delta_{lo} \leq \delta_{hi}$.
The trace recognizer now defines valid behaviour of a component in terms of both logic values and transition time. For example:



$$\texttt{c\_step(prev,next)} \triangleq$$
$$\begin{aligned}
&\bullet\ \bullet\ \bullet \\
&\vee\big(\texttt{prev[in1].v} = \texttt{prev[in2].v} \wedge \texttt{prev[out].v} \neq \texttt{prev[in1].v} \\
&\quad \wedge \texttt{prev[out].v} \neq \texttt{next[out].v} \\
&\quad \wedge \texttt{max}\big(\texttt{prev[in1].t}, \texttt{prev[in2].t}\big) + \delta_{lo} \leq \texttt{next.t} \\
&\quad \wedge \texttt{next[out].t} = \texttt{next.t} \\
&\quad \wedge \texttt{next.t} < \texttt{max}\big(\texttt{prev[in1].t}, \texttt{prev[in2].t}\big) + \delta_{hi}\big)
\end{aligned}$$

This says, when the output of the C-element changes, it should happen between delay bounds.

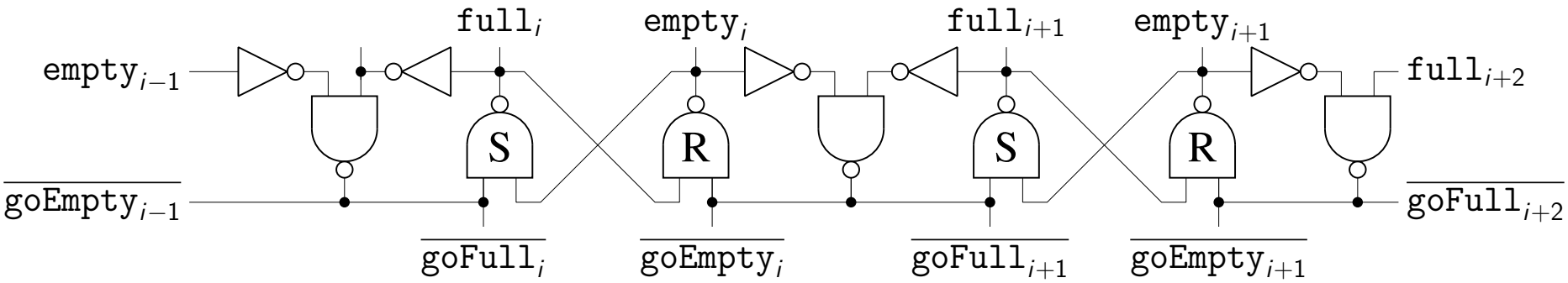
Nondeterminism



- Because of continuous time, a state in a trace can have an infinite number of successors.
- Our timed trace model naturally captures nondeterminism.

The asP* Pipelines

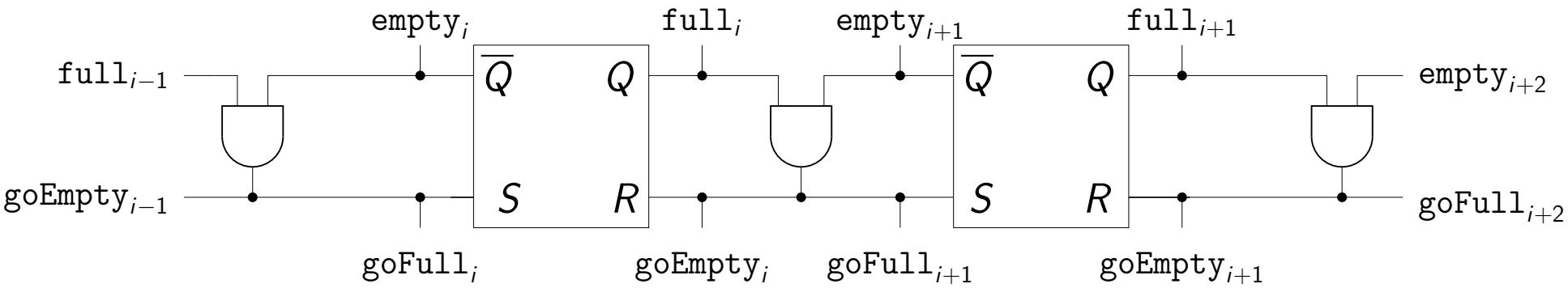
We recall the asP* pipeline is:



We want to find a way to break the pipeline into stages.

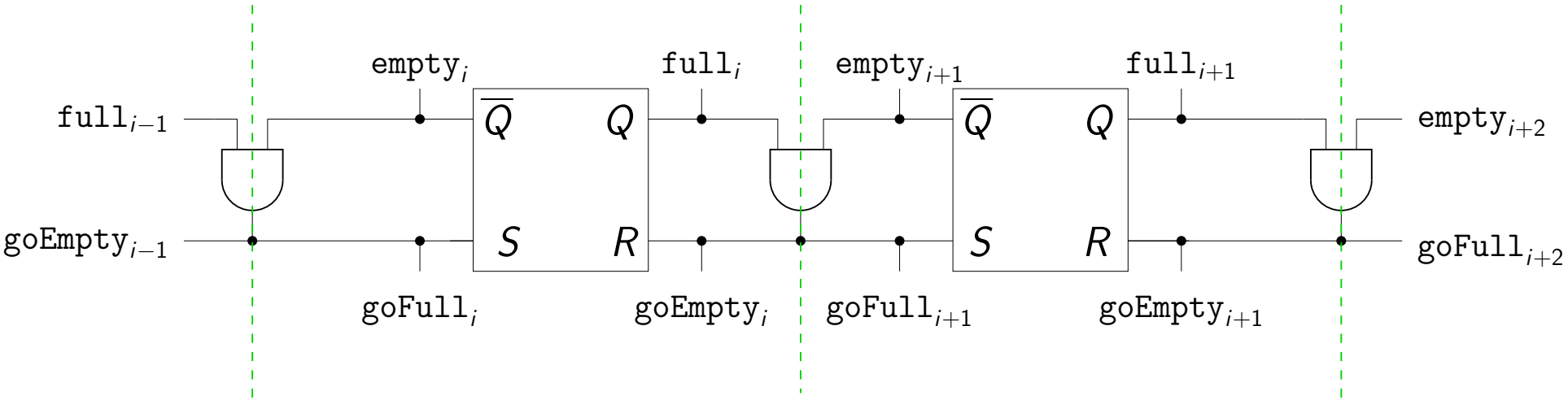
The asP* Pipelines

Apply a few logical equivalences to get:



The asP* Pipelines

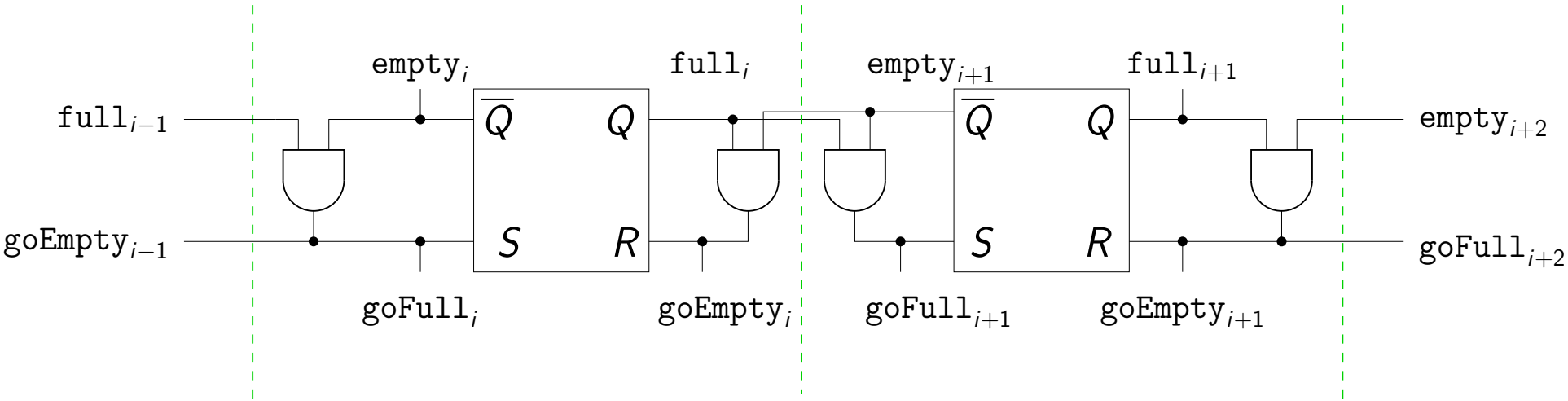
The obvious way is to split the AND gate into two halves:



But we don't want to model half of an AND gate.

The asP* Pipelines

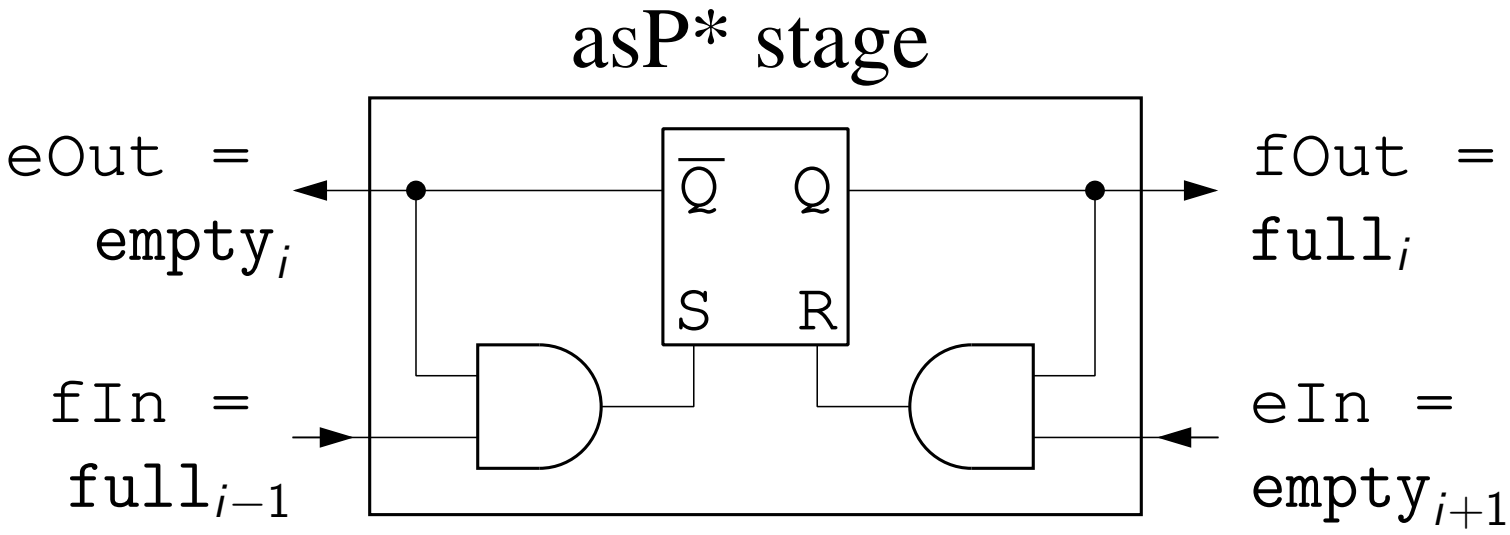
Instead we duplicate the AND gates:



This transformation includes all possible behaviours of the pipeline and additional behaviours due to the duplicated AND gates.

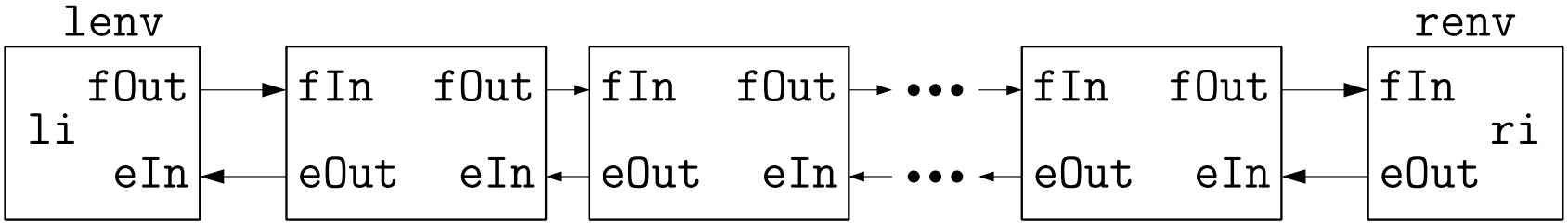
A Single Stage

So a single stage becomes:



- *eOut* stands for current stage being empty, and *fIn* stands for previous stage being full. In this case, the current stage can set
- *fOut* stands for current stage being full, and *eIn* stands for next stage being empty. In this case, the current stage can reset

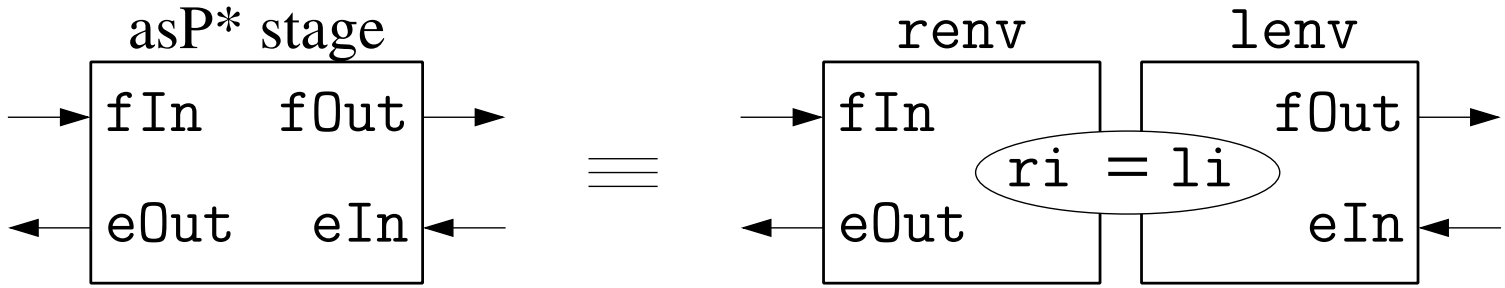
We Care About the Environment



- To verify a pipeline, we need to consider actions of the environment.
- A `lEnv` is a "left-environment"
  - It acts like an asP* stage,
  - But it doesn't have a left-neighbour.
  - An empty `lEnv` can spontaneously go full any time after a minimum delay after going empty.
  - An internal signal `li` allows the environment to have a pending output event.
- A `rEnv` is similar – it can spontaneously go empty.

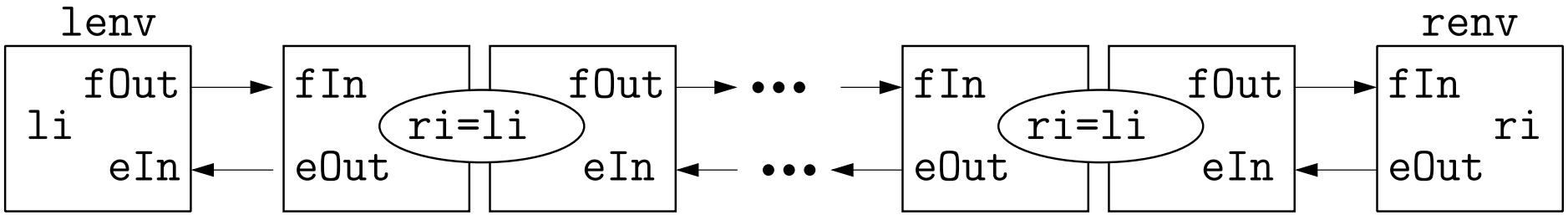
rEnv + lEnv = Stage

A stage can be built with a right and a left environment with the constraint that $ri == li$:



- Signal $ri == li$ represents the internal state of a stage when it has acquired the value but hasn't set/reset Q or $\overline{Q}$ yet.

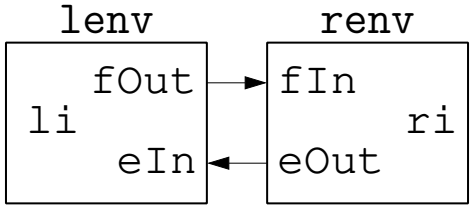
This gives us the final model for the asP* pipeline:



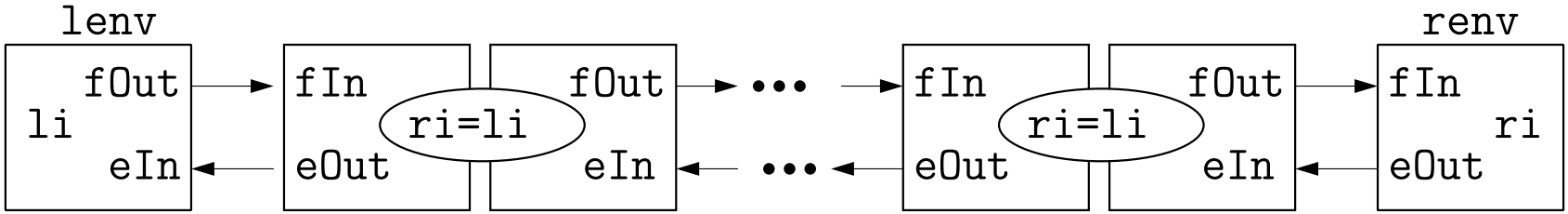
1. Motivation
2. Timed Traces
3. Modeling asP* Pipelines
4. Verifying the asP* Pipelines
5. Conclusions

Three configurations

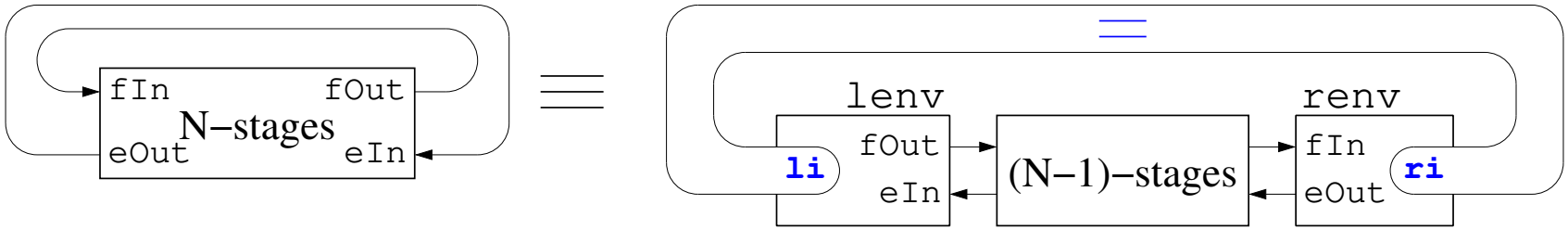
- Configuration 1: a lenv with a renv, 0 stage



- Configuration 2: a lenv, an arbitrary number of stages, and a renv



- Configuration 3: a ring with an arbitrary number of stages



Formalizing the Timing Invariant

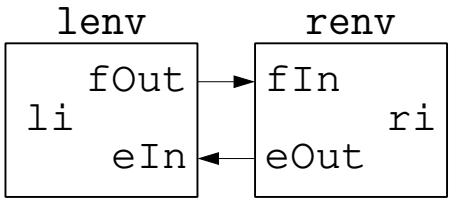
We define the timing invariant for an empty pipeline, and the invariant for the other two configurations can be defined recursively. The timing invariant is composed of two parts:

- Timing constraints within each individual component: lenv and renv – Obvious, based on the trace recognizers.
- Timing constraints on the interactions between components – The interesting part.

We prove the inductive invariant by proving the theorem:

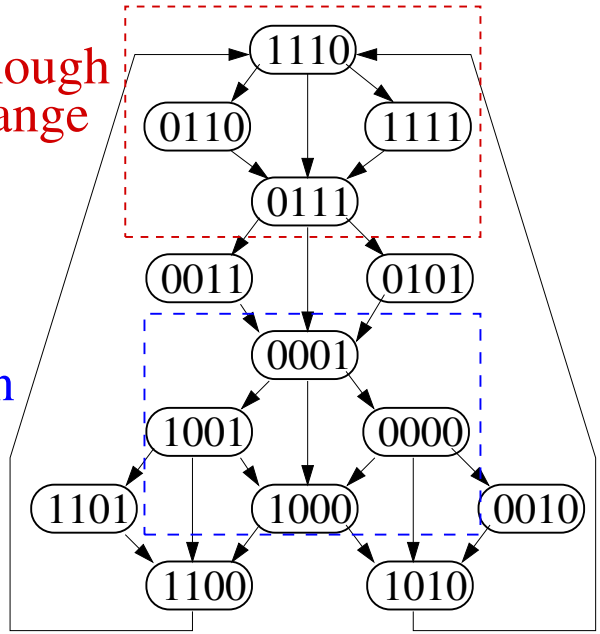
$$\begin{aligned} & \texttt{invariant\_env(s1)} \wedge \texttt{valid\_step(s1, s2)} \\ \Rightarrow \quad & \texttt{invariant\_env(s2)} \end{aligned} \tag{1}$$

Interaction invariants



f and e will hold long enough for both ri and li to change

f and e will both go low before either goes high again



ready to transfer

internal states reset

fOut and eOut reset

spontaneous action in lenv and renv

next enabling of fOut and eOut

(l f e r)

l = lenv internal state
r = right internal state

f = lenv.fOut = renv.fIn
e = lenv.eIn = renv.eOut

Proving Hazard-freedom

Definition 1 (Hazard)

Between two consecutive states, if some component output, y, is excited to change in the first state, but has neither changed nor is excited to make that change in the second state, we call this a hazard.

For example, the hazard-free step condition for signal `li` of `lenv` from state `s1` to state `s2` is:

$$\begin{aligned} & \texttt{s1[lenv.eIn].v} \wedge \texttt{s1[lenv.fOut].v} \wedge \\ & \quad \texttt{s1[lenv.li].v} \\ \Rightarrow \quad & \texttt{s2[lenv.eIn].v} \wedge \texttt{s2[lenv.fOut].v} \end{aligned} \tag{2}$$

Proving Hazard-freedom

Definition 1 (Hazard)

Between two consecutive states, if some component output, y, is excited to change in the first state, but has neither changed nor is excited to make that change in the second state, we call this a hazard.

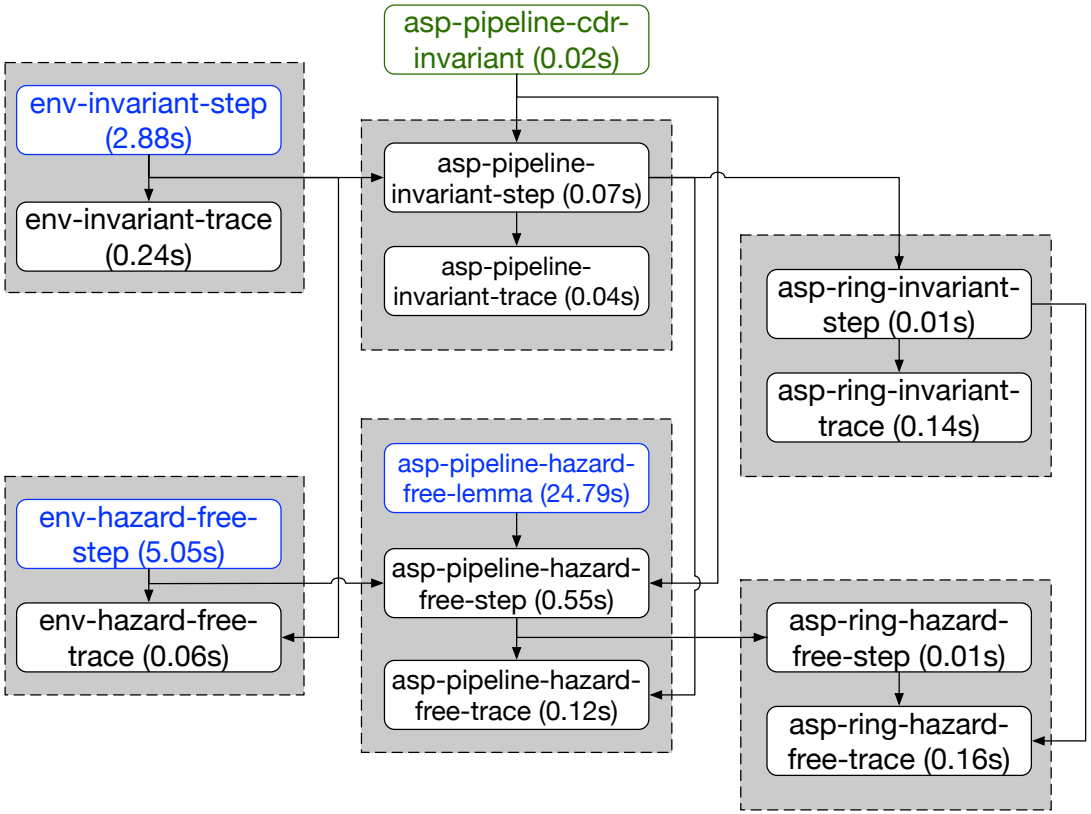
To show that a module's invariant ensures hazard freedom, we prove the theorem:

$$\begin{aligned} & \texttt{invariant\_mod(s1)} \wedge \texttt{valid\_step\_mod(s1,s2)} \\ \Rightarrow \quad & \texttt{hazard\_free\_mod(s1, s2)} \end{aligned} \tag{2}$$

Inductively, we prove that every step of a valid trace satisfies the hazard-free property.

The Proof Tree

We verify timed circuits with the generality of ACL2 while achieving performance comparable to dedicated tools by using Smtlink, a SMT solver interface



- This graph represents all theorems proved.
- The proof is greatly automated by using the Smtlink package.
- Counter-example driven manual learning of invariant.

Conclusion and Future Work

- ACL2 is a general purpose theorem prover that can be used to verify correctness of timed, asynchronous circuits.
- Using Smtlink, proofs are nearly automatic:
  - Prove each invariant is maintained by a single step (Smtlink).
  - Prove the invariant holds for a valid trace (ACL2 excels at induction).
- The performance is comparable to dedicated tools:
  - Most proofs complete within a second, the longest was less than 30 seconds.
  - The verification problem is formulated in the ACL2 logic rather than writing special-purpose timing analysis algorithms.
- We can verify parameterized designs:
  - An arbitrary number of pipeline or ring stages.
  - Symbolic constraints on timing delays.

Future Work

- Integrate with existing ACL2 Verilog front-end (SV and VL).
- Use model checking algorithms to automatically generate invariants.
- Explore more sophisticated timing models including metastability.
- Combine with verification of functionality:
  - Prove that timed designs implement untimed abstractions.
  - Reason about functionality in untimed model (e.g. [CCS19]).
- Continue to extend Smtlink and make proofs even more automatic.

Future Work

We verify timed circuits with the generality of ACL2 while achieving performance comparable to dedicated tools by using Smtlink, a SMT solver interface

- Integrate with existing ACL2 Verilog front-end (SV and VL).
- Use model checking algorithms to automatically generate invariants.
- Explore more sophisticated timing models including metastability.
- Combine with verification of functionality:
  - Prove that timed designs implement untimed abstractions.
  - Reason about functionality in untimed model (e.g. [CCS19]).
- Continue to extend Smtlink and make proofs even more automatic.

Thank You!

References I

Martín Abadi and Leslie Lamport.

An old-fashioned recipe for real time.

ACM Transactions on Programming Languages and Systems, 16(5):1543–1571, September 1994.

Matt Kaufmann Marly Roncken Cuong Chau, Warren Hunt and Ivan Sutherland.

A hierarchical approach to self-timed circuit verification.

May 2019.

David L. Dill.

Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.

PhD thesis, School of Computer Science, Carnegie Mellon University, 1988.

Published in book form as part of the ACM Doctoral Dissertation Award Series by the MIT Press, Cambridge, MA, 1989.

References II

Charles E. Molnar, Ian W. Jones, et al.

A FIFO ring oscillator performance experiment.

In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, April 1997.

Ivan Sutherland and Scott Fairbanks.

GasP: A minimal FIFO control.

In *Proceedings of the Seventh International Symposium on Asynchronous Circuits and Systems*, pages 46–53, April 2001.

M. Singh and S. M. Nowick.

Mousetrap: High-speed transition-signaling asynchronous pipelines.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15(6):684–698, June 2007.

References III

Ivan E. Sutherland.

Micropipelines.

Communications of the ACM, 32(6):720–738, June 1989.

Turing Award lecture.

Proving Deadlock-freedom

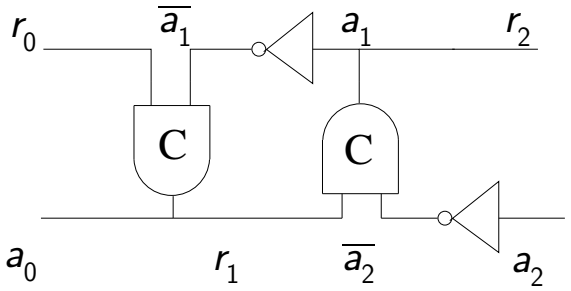
Definition 2 (Deadlock-freedom)

For all valid states, there always exist a valid next state that's different from the current state.

We prove deadlock-freedom by defining a witness function that provides a valid next state.
To show that a module's invariant ensures deadlock-freedom, we prove the theorem:

$$\begin{aligned} \texttt{invariant\_mod(s1)} \Rightarrow \quad & \texttt{witness\_mod(s1)} \neq \texttt{nil} \\ & \wedge \texttt{valid\_mod(s1, witness\_mod(s1))} \\ & \wedge \texttt{changed(s1, witness\_mod(s1))} \end{aligned} \tag{3}$$

Traces and Trace Recognizers



For example, this is an invalid trace:

r_0:F	a_0:F	r_1:F	a_1:F	$\overline{a_1}$:F	r_2:F	a_2:F	$\overline{a_2}$:F	...
r_0:**T**	a_0:F	r_1:F	a_1:F	$\overline{a_1}$:F	r_2:F	a_2:F	$\overline{a_2}$:F	...
r_0:T	a_0:**T**	r_1:F	a_1:F	$\overline{a_1}$:F	r_2:F	a_2:F	$\overline{a_2}$:F	...
...	...	...	...	...	...	...	...	...

a_0 as the output of the first C-element makes an unexplained transition, violating the trace recognizer for the first C-element:

$$\begin{aligned} &\texttt{c\_step(prev, next)} \triangleq \\ &\quad \texttt{next[out]} = \texttt{prev[out]} \\ &\vee \quad \texttt{prev[in1]} = \texttt{prev[in2]} \wedge \texttt{prev[in1]} \neq \texttt{prev[out]} \end{aligned} \tag{4}$$

Datatypes

In ACL2, we use algebraic datatypes for modeling traces:[1]

`Trace`	list of `TimedState`
`TimedState`	product of `Time` and `StateMap`
`StateMap`	map from `Path` to `TimedValue`
`Path`	list of `PathElement`
`PathElement`	product of `Name` and `Index`
`TimedValue`	product of `Time` and `Value`

Using this timed trace model, when proving properties of a circuit, usually we will prove a theorem in the shape:

$$\begin{aligned} & \texttt{Circuit(circuit)} \wedge \texttt{Trace(tr)} \\ & \wedge \texttt{valid\_circuit(circuit, tr)} \\ \Longrightarrow\ & \texttt{property(circuit, tr)} \end{aligned} \tag{5}$$

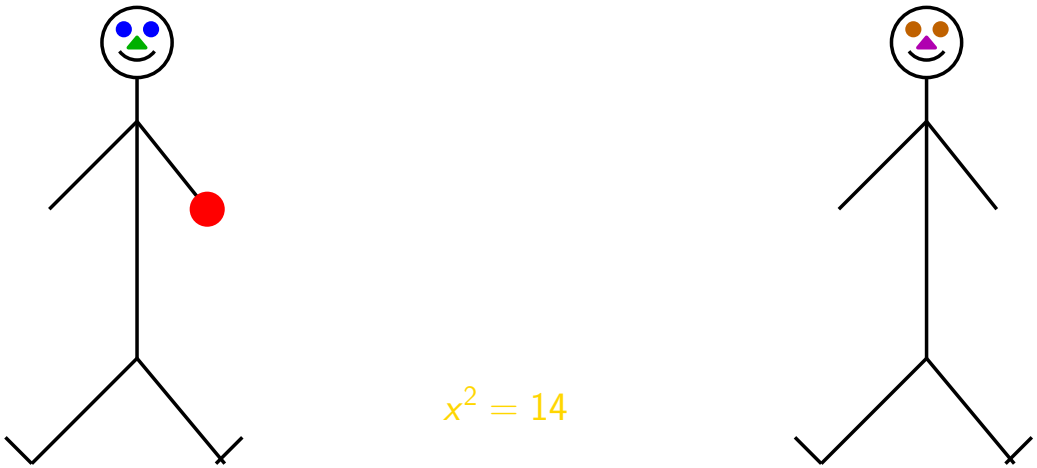
[1] Basic types: `Time` - `rationalp`; `Value` - `booleanp`; `Name` - `Symbolp`; `Index` - `integerp`

Related Work

Timed automaton based approaches, relative timing, theorem proving, other parameterized methods.

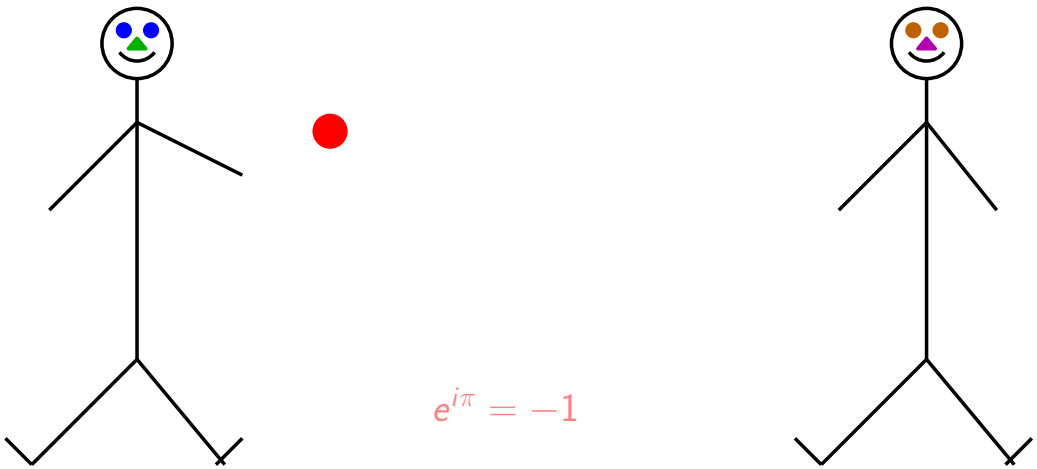
- Comparing to timed automaton based approaches, our method allows symbolic delays on gates and allows parameterized verification.
- Comparing to relative timing, our approach models full timing information that allows verification of a larger set of circuits.
- Comparing to theorem proving based methods, our use of an SMT solver at the backend greatly reduced human effort.
- Existing parameterized methods suffers from loss of precision, we can achieve precision by using induction proofs.

Cool Animation



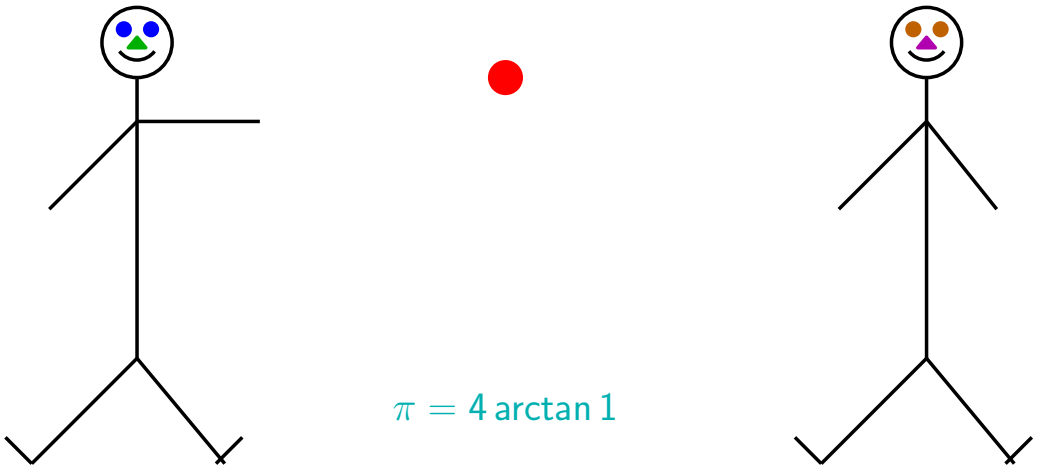
- This is an awesome animation.

Cool Animation



- This is an awesome animation.

Cool Animation



- This is an awesome animation.