A Circuit Approach to LTL Model Checking

Koen Claessen
Chalmers University of Technology

Niklas Een, Baruch Sterin
UC Berkeley

PLTL – Linear Temporal Logic w/Past Operators

LTL and PLTL are used to model and specify system behavior

Atomic Propositions

p, q, ….

Boolean Operators

&, |, !, –>, …

PLTL – Linear Temporal Logic w/Past Operators

Some future temporal operators

Xf	**f** holds in the next cycle
Ff	**f** holds sometime in the future
Gf	**f** holds forever
f**U**g	**g** holds sometime in the future, and until then, **f** holds

Some past temporal operators

Yf	**f** held in the previous cycle
Of	**f** held sometime in the past
Hf	**f** held until now

A Few LTL Formulas

G !err

The error signal is never raised

F err

The error signal will eventually be raised

G(req –> **X F** ack)

Every request must be eventually acknowledged

F (req & **X G** !ack)

There will eventually be a request that is never acknowledged

Monotonicity

LTL operators are monotone

For example:

p	: 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 …
Xp	: 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 …

Monotonicity

LTL operators are monotone

For example:

p : 0 0 1 0 0 1 0 **1** 0 0 1 1 0 **1** 0 0 0 **1** 0 0 1 …

Xp : 0 1 0 0 1 0 **1** 0 0 1 1 0 **1** 0 0 0 **1** 0 0 1 …

If p holds in more places, then **X**p holds in more places

Automata-Theoretic Approach[VW86]

Every LTL formula **f** has a Büchi automaton $\mathbf{A_f}$ (**monitor**) that accepts all traces that satisfy **f**

To check whether **f** hold on every trace of **M:**

- Build a Büchi automaton $\mathbf{A_{!f}}$ (monitor for **!f**)
- Check if $\mathbf{M \times A_{!f}}$ is empty

LTL Model Checking

Directly construct Büchi Automata [VW86]

Construct an Alternating Büchi Automata, convert to Regular Büchi [V95]

- Beautiful, clean and elegant
- Alternating automata and their conversion to Büchi automata are nontrivial

Temporal Testers [PZ06]

Transforming the Formula

Assume formula in NNF, with only |, & as boolean operators

For every node ***f** or **f*g** in the parse tree:

- Introduce a new **activator** variable $\mathbf{z_i}$
- Replace the node with that variable
- Maintain correctness by adding a **conjunct** $\mathbf{G}(\mathbf{z_i}$ <–> ***f**) or $\mathbf{G}(\mathbf{z_i}$ <–> **f*g**)

Add a conjunct $\mathbf{z_0}$ for the top level activator

Example

Formula

F (req & **X** **G** !ack)

Conjuncts

Example

Formula	**Conjuncts**
$\mathbf{F}$ (req & $\mathbf{X}$ $\mathbf{z_3}$)	$\mathbf{G}$($\mathbf{z_3}$ <–> $\mathbf{G}$!ack)

Example

Formula

$\mathbf{F}\ (\ \mathrm{req}\ \&\ \mathbf{z_2}\ )$

Conjuncts

$\mathbf{G}(\ \mathbf{z_2} <\!\!-\!\!> \mathbf{X}\ \mathbf{z_3}\ )\ \&$

$\mathbf{G}(\ \mathbf{z_3} <\!\!-\!\!> \mathbf{G}\ !\mathrm{ack}\ )$

Example

Formula	Conjuncts
F $\mathbf{z_1}$	**G**($\mathbf{z_1}$ <–> req & $\mathbf{z_2}$) &
	G($\mathbf{z_2}$ <–> **X** $\mathbf{z_3}$) &
	G($\mathbf{z_3}$ <–> **G** !ack)

Example

Formula	**Conjuncts**
$\mathbf{z_0}$	$\mathbf{G}(\ \mathbf{z_0} <\!\!-\!\!> \mathbf{F}\ \mathbf{z_1}\ )\ \&$
	$\mathbf{G}(\ \mathbf{z_1} <\!\!-\!\!> \mathrm{req}\ \&\ \mathbf{z_2}\ )\ \&$
	$\mathbf{G}(\ \mathbf{z_2} <\!\!-\!\!> \mathbf{X}\ \mathbf{z_3}\ )\ \&$
	$\mathbf{G}(\ \mathbf{z_3} <\!\!-\!\!> \mathbf{G}\ !\mathrm{ack}\ )$

Example

Formula	**Conjuncts**
	$\mathbf{z_0}$ &
	$\mathbf{G}(\mathbf{z_0} \text{ <-> } \mathbf{F}\ \mathbf{z_1})$ &
	$\mathbf{G}(\mathbf{z_1} \text{ <-> } \text{req} \ \&\ \mathbf{z_2})$ &
	$\mathbf{G}(\mathbf{z_2} \text{ <-> } \mathbf{X}\ \mathbf{z_3})$ &
	$\mathbf{G}(\mathbf{z_3} \text{ <-> } \mathbf{G}\ !\text{ack})$

<–> –> –>

We can replace the <–> with a simple –>

Given a trace satisfying a conjunct **G**($\mathbf{z}_i$ <–> **f*g**)
Then it satisfies **G**($\mathbf{z}_i$ –> **f*g**)

Given a trace satisfying a conjunct **G**($\mathbf{z}_i$ –> **f*g**)
Then we change $\mathbf{z}_i$ to 1 whenever **f*g** holds

Because LTL operators are monotonic, and in NNF we only have monotonic boolean operators, this trace now satisfies **G**($\mathbf{z}_i$ <–> **f*g**)

Example

$\mathbf{Z_0}$ &

$\mathbf{G}(\mathbf{z_0} \text{ –> } \mathbf{F}\ \mathbf{z_1})$ &

$\mathbf{G}(\mathbf{z_1} \text{ –> } \text{req \& } \mathbf{z_2})$ &

$\mathbf{G}(\mathbf{z_2} \text{ –> } \mathbf{X}\ \mathbf{z_3})$ &

$\mathbf{G}(\mathbf{z_3} \text{ –> } \mathbf{G}\ \text{!ack})$

This new formula is satisfiable iff the original formula is satisfiable

It is easy to construct monitors for each conjunct

Monitors

Monitors

pending:

Holds if the monitor has an outstanding requirement

failed:

Holds if a violation has been detected

accept:

Must hold infinitely often for a trace to be valid

In most cases **accept** = **!pending**

Example Monitors

G(z –> X a)

pending = z

failed = **prev**(z) & !a

G(z –> **G** a)

pending = **prev**(pending) | z

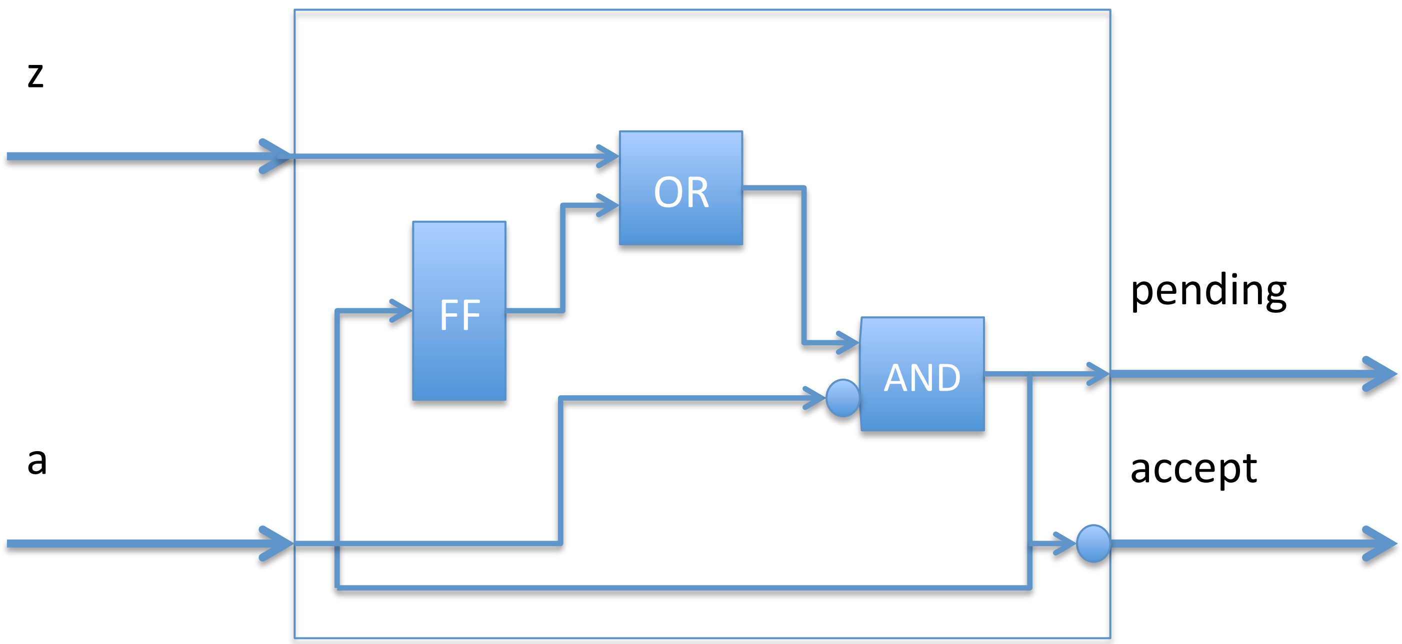
failed = pending & !a

G(z –> **F** a)

pending = (z | **prev**(pending)) & !a

accept = pending & !a

Monitor for G(z -> Fa)



Summary

Negate the formula **f**

Put **!f** in NNF form

Expand !**f** to its conjuncts

Replace <–> with –>

Construct monitors for the conjuncts

Mark all !**failed** signals as constraints

Replace the top-level $\mathbf{z_0}$ with **is_init**

Mark all **accept** signals as fairness constraints

Finite Traces

What happens if all pending signals become 0?

The trace can be extended to an infinite trace, by setting all activators to 0 going forward

This gives a safety property (!failed & !pending), which catches all **informative prefixes** [KV99]

Assumptions and Assertions

LTL formulas are used to either

- Specify behavior – **Assertions**
- Model the environment – **Assumptions**

In practice, infinite traces are expensive to find (finding a loop is hard)

Sometimes, a reasonable compromise for safety assertions, is to only use the **failed** signal of the assumptions (ignoring **accept**)

Deadlock and Acceptable States

Deadlock States:

- States which will eventually reach a **failed** signal
- Transitive strong preimage of **failed**
- Detect **failed** faster

Acceptable States:

- States that can reach all **accept** signals
- Intersection of the all the transitive (weak) preimage of each **accept** signals
- Restrict search to a small set

Reachable States

We can compute the reachable state space of the monitor

Can be added as a constraint to improve k-induction and PDR performance

Provide similar benefit to determinizing the automaton

Experimental Results

Benchmarks from [BHJLS06]

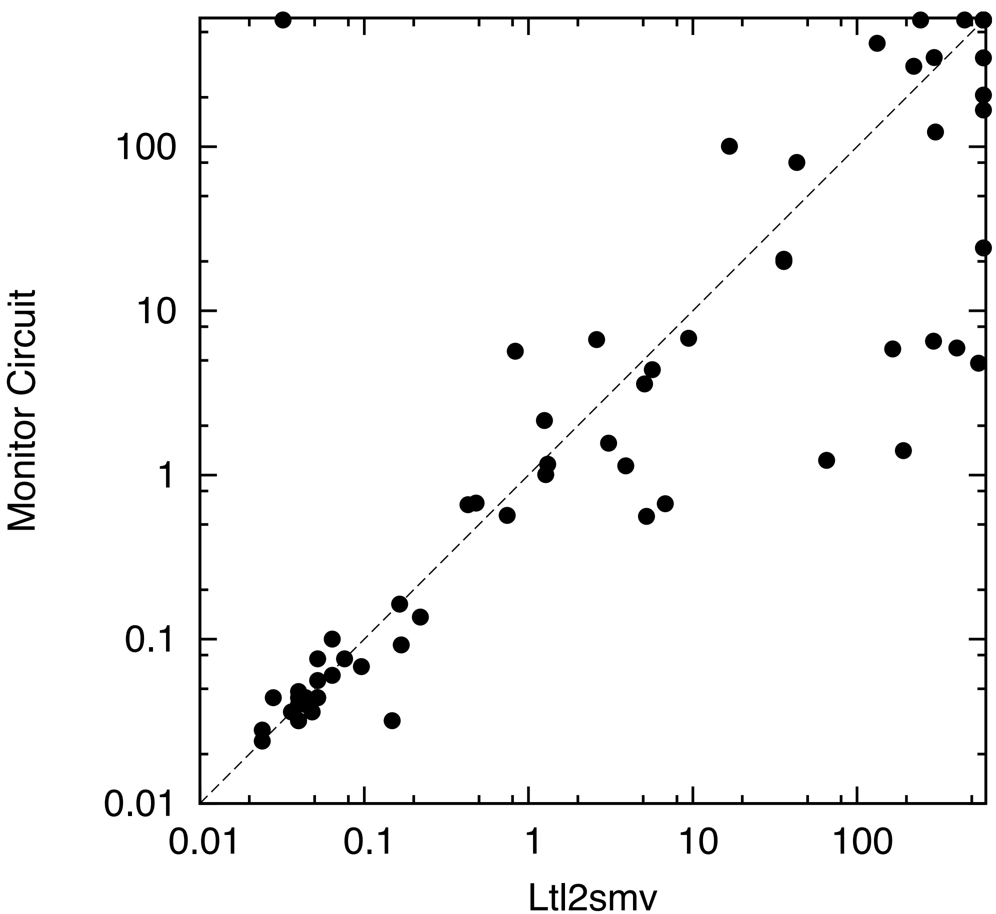
- SMV files and PLTL formulas
- Except for **1394**, **csmacd** (could not translate)

Compared to **LTL2SMV**

- now part of the NuSMV distribution

Converted from SMV to AIGER using our own tool

Experimental Results



Conclusions

Our approach is competitive with existing methods

Its (relative) simplicity makes it a good option for industrial use