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

\( X_f \quad f \) holds in the next cycle
\( F_f \quad f \) holds sometime in the future
\( G_f \quad f \) holds forever
\( f U g \quad g \) holds sometime in the future, and until then, \( f \) holds

Some past temporal operators

\( Y_f \quad f \) held in the previous cycle
\( O_f \quad f \) held sometime in the past
\( H_f \quad f \) held until now
A Few LTL Formulas

\( \mathbf{G} \neg \text{err} \)

The error signal is never raised

\( \mathbf{F} \text{err} \)

The error signal will eventually be raised

\( \mathbf{G}( \text{req} \rightarrow \mathbf{X} \mathbf{F} \text{ack} ) \)

Every request must be eventually acknowledged

\( \mathbf{F}( \text{req} \& \mathbf{X} \mathbf{G} \neg \text{ack} ) \)

There will eventually be a request that is never acknowledged
Monotonicity

LTL operators are monotone

For example:

\[ p : 001001000011000000001 \ldots \]

\[ Xp : 010010000110000000001 \ldots \]
Monotonicity

LTL operators are monotone

For example:

\[ p : 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ \ldots \]

\[ Xp : 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ \ldots \]

If \( p \) holds in more places, then \( Xp \) holds in more places
Automata-Theoretic Approach [VW86]

Every LTL formula $f$ has a Büchi automaton $A_f$ (monitor) that accepts all traces that satisfy $f$

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

1. Build a Büchi automaton $A_{\neg f}$ (monitor for $\neg f$)
2. Check if $M \times A_{\neg f}$ 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 \( z_i \)
- Replace the node with that variable
- Maintain correctness by adding a conjunct \( G(z_i \leftrightarrow *f) \) or \( G(z_i \leftrightarrow f*g) \)

Add a conjunct \( z_0 \) for the top level activator
Example

<table>
<thead>
<tr>
<th>Formula</th>
<th>Conjuncts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F (\text{req} &amp; X G \text{!ack})$</td>
<td></td>
</tr>
</tbody>
</table>
Example

Formula: \( F (\text{req} \& \ X \ z_3) \)

Conjuncts: \( G( z_3 \leftrightarrow G \text{!ack} ) \)
Example

**Formula**

\[ F(\text{req} \& z_2) \]

**Conjuncts**

\[ G(z_2 \leftrightarrow X z_3) \& G(z_3 \leftrightarrow G \text{!ack}) \]
<table>
<thead>
<tr>
<th>Formula</th>
<th>Conjuncts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Fz_1$</td>
<td>$G(\ z_1 \longleftrightarrow \text{req} \ &amp; \ z_2 ) \ &amp;$</td>
</tr>
<tr>
<td></td>
<td>$G(\ z_2 \longleftrightarrow X \ z_3 ) \ &amp;$</td>
</tr>
<tr>
<td></td>
<td>$G(\ z_3 \longleftrightarrow G \ \text{!ack} )$</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Formula</th>
<th>Conjuncts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z_0$</td>
<td>$G( z_0 \leftrightarrow F z_1 )$ &amp;</td>
</tr>
<tr>
<td></td>
<td>$G( z_1 \leftrightarrow req &amp; z_2 )$ &amp;</td>
</tr>
<tr>
<td></td>
<td>$G( z_2 \leftrightarrow X z_3 )$ &amp;</td>
</tr>
<tr>
<td></td>
<td>$G( z_3 \leftrightarrow G !ack )$</td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Formula</th>
<th>Conjuncts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z_0$ &amp;</td>
<td></td>
</tr>
<tr>
<td>$G( z_0 \leftrightarrow F z_1 )$ &amp;</td>
<td></td>
</tr>
<tr>
<td>$G( z_1 \leftrightarrow \text{req} &amp; z_2 )$ &amp;</td>
<td></td>
</tr>
<tr>
<td>$G( z_2 \leftrightarrow X z_3 )$ &amp;</td>
<td></td>
</tr>
<tr>
<td>$G( z_3 \leftrightarrow G \text{!ack} )$</td>
<td></td>
</tr>
</tbody>
</table>
We can replace the $\langle\rightarrow\rangle$ with a simple $\rightarrow$.

Given a trace satisfying a conjunct $G(z_i \langle\rightarrow\rangle f \ast g)$
Then it satisfies $G(z_i \rightarrow f \ast g)$

Given a trace satisfying a conjunct $G(z_i \rightarrow f \ast g)$
Then we change $z_i$ to 1 whenever $f \ast g$ holds.

Because LTL operators are monotonic, and in NNF we only have monotonic boolean operators, this trace now satisfies $G(z_i \langle\rightarrow\rangle f \ast g)$.
Example

$Z_0$ &
$G( z_0 \rightarrow F z_1 )$ &
$G( z_1 \rightarrow req \& z_2 )$ &
$G( z_2 \rightarrow X z_3 )$ &
$G( z_3 \rightarrow G \!ack )$

This new formula is satisfiable iff the original formula is satisfiable

It is easy to construct monitors for each conjunct
Monitors

\[ f \ast g \]

- \( z \)
- \( f \)
- \( g \)
- error
- pending
- accept
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

\( G( z \rightarrow X a ) \)

\begin{align*}
\text{pending} &= z \\
\text{failed} &= \text{prev}(z) \& !a
\end{align*}

\( G( z \rightarrow G a ) \)

\begin{align*}
\text{pending} &= \text{prev}(\text{pending}) \mid z \\
\text{failed} &= \text{pending} \& !a
\end{align*}

\( G( z \rightarrow F a ) \)

\begin{align*}
\text{pending} &= (z \mid \text{prev}(\text{pending}) ) \& !a \\
\text{accept} &= \text{pending} \& !a
\end{align*}
Monitor for $G(z \rightarrow Fa)$
Summary

Negate the formula $f$
Put $\neg f$ in NNF form
Expand $\neg f$ to its conjuncts
Replace $\iff$ with $\rightarrow$
Construct monitors for the conjuncts
Mark all $\neg$failed signals as constraints
Replace the top-level $z_0$ 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 (\(!\text{failed} \& \!\text{pending}\) ), 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