temporal logic

propositional logic extended with temporal operators referring to past and future

CS 119 includes slides by Grigore Rosu
The Temporal Logic of Reactive and Concurrent Systems

Zohar Manna
Amir Pnueli

Springer-Verlag
future time properties

• If A happens now

B must happen

□(A → ◊ B)
past time properties

• If $A$ happens now

$B$ must have happened

$\Box (A \rightarrow \Diamond B)$
instances of MOP
## Chomsky’s language hierarchy

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Languages</th>
<th>Automaton</th>
<th>Production rules (constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>Recursively enumerable</td>
<td>Turing machine</td>
<td>$\alpha \rightarrow \beta$ (no restrictions)</td>
</tr>
<tr>
<td>Type-1</td>
<td>Context-sensitive</td>
<td>Linear-bounded non-deterministic Turing machine</td>
<td>$\alpha A\beta \rightarrow \alpha \gamma \beta$</td>
</tr>
<tr>
<td>Type-2</td>
<td>Context-free</td>
<td>Non-deterministic pushdown automaton</td>
<td>$A \rightarrow \gamma$</td>
</tr>
<tr>
<td>Type-3</td>
<td>Regular</td>
<td>Finite state automaton</td>
<td>$A \rightarrow a$ and $A \rightarrow aB$</td>
</tr>
</tbody>
</table>

temporal logic for finite traces: subset of regular languages

\( p \) is even in every other state

regular

temporal

even \( \land \) always(even implies (next next even))

does not work
properties of temporal logic

- some properties can be stated more succinctly in TL.
- but TL properties can be hard to write and read for non-trivial scenarios.

- now, many scenarios are trivial.

- it is of course a debate at a practical engineering level what notation is most suitable in practice. Number of characters is one possible measure.
future and past time
temporal logic
semantics and algorithms
future time

semantics and algorithm
monitoring Future Time LTL

Syntax – Propositional Calculus plus

- $o F$ (next)
- $\square F$ (always)
- $\diamond F$ (eventually)
- $F \cup F'$ (until)

Executable Semantics – Rewriting

$\{\_\} : \text{Formula} \times \text{Event} \rightarrow \text{Formula}$ ("consume" event $e$)

$F\{e\}$ formula that should hold after processing $e$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p{e}$</td>
<td>$p$ true on $e$?</td>
</tr>
<tr>
<td>$(F _o_ F'){e}$</td>
<td>$F{e} _o_ F'{e}$</td>
</tr>
<tr>
<td>$(o F){e}$</td>
<td>$F$</td>
</tr>
<tr>
<td>$(\square F){e}$</td>
<td>$F{e} \land (\square F)$</td>
</tr>
<tr>
<td>$(\diamond F){e}$</td>
<td>$F{e} \lor (\diamond F)$</td>
</tr>
<tr>
<td>$(F \cup F'){e}$</td>
<td>$F'{e} \lor (F{e} \land (F \cup F'))$</td>
</tr>
</tbody>
</table>
Future Time LTL - example

Event stream: \( \text{red yellow green yellow green red ...} \)

\[ \square(\text{green} \rightarrow \neg \text{red U yellow}) \ \{\text{yellow}\} \]
\[ \downarrow^* \]

\[ \square(\text{green} \rightarrow \neg \text{red U yellow}) \ \{\text{green}\} \]
\[ \downarrow^* \]

\[ ((\neg \text{red U yellow}) \land \square(\text{green} \rightarrow \neg \text{red U yellow})) \{\text{yellow}\} \]
\[ \downarrow^* \]

\[ \square(\text{green} \rightarrow \neg \text{red U yellow}) \ \{\text{green}\} \]
\[ \downarrow^* \]

\[ ((\neg \text{red U yellow}) \land \square(\text{green} \rightarrow \neg \text{red U yellow})) \{\text{red}\} \]
\[ \downarrow^* \]

\[ (\text{yellow}\{\text{red}\} \lor (\neg \text{red}\{\text{red}\} \land \neg \text{red U yellow})) \land ... \ \Rightarrow \text{false} \land ... \ \Rightarrow \text{false} \]

Formula was violated!
timed temporal logic

- add real time (RTL, MiTL, timed automata, etc.)

\( \square (start \rightarrow 5 stop) \)

\( (tF)\{e: \delta\} \Rightarrow (\delta \leq t) \land (F\{e: \delta\} \lor t-\delta F) \)
rewriting using Maude

• implemented the algorithm above in PaX
  – Maude as rewriting engine
    • 15 lines of obviously correct code!
• monitored 100 million events on 1.7GHz PC
  – 185 seconds, 220 million rewrites
  – Faster than modified Büchi automaton in Java
    (1,500 lines of code)
    • Is this 1,500 LOC Java program correct?
• I/O + buffering take longer than rewriting …
building a minimal BTT_FSM

• idea
  – do the rewrites for \textit{all} possible values of predicates
  – get a finite state machine
    • nodes are LTL formulae
    • optimize using a validity checker ($F \leftrightarrow F'$: one state)
    • edges are propositions
    • assign numbers to states
    • replace edges by \textit{Binary Transition Trees}
Binary Transition Tree FSM

- We can build minimal FSMs *statically* for LTL

<table>
<thead>
<tr>
<th>Formula</th>
<th>□(green → ¬red U yellow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>1</td>
</tr>
<tr>
<td>BTT</td>
<td>yellow ? 1 : green ? (red ? false : 2) : 1</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>yellow</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>red</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>false</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>false</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
```
MOP does not operate with “the end of a trace”

• so we cannot even write a formula of the form: □ (F → ◇ F’) and expect it to fail on for example the trace: F F’ F.

• instead, we have to assume an end event E and write: □ (F → !E U F’), meaning: if we see an F then we should see no E before an F’.

in MOP F U F’ does not mean that F’ eventually must happen. This is in contrast to traditional interpretations.
statistical approach
just a thought

Formula: \( \varphi = \Box (A \rightarrow !E \cup B) \)

Trace: A A A A A A A B E satisfies \( \varphi \)
Trace: A A A A A A A A A A E violates \( \varphi \)
Trace: A B A B A B A B A B A E violates \( \varphi \)

is this trace better?
was ok 4 out of 5 times

maintain statistical information?
past time

semantics and algorithm
monitoring safety

• Example:

**Safe Landing**
Land the space craft only after approval from ground and only if, since then, the radio signal has not been lost

or formally

\[ \uparrow \text{Landing} \rightarrow \{\text{Approved, } \downarrow \text{Radio}\} \]

where (MAC)

\[ \uparrow F \text{ means start } F, \]
\[ \downarrow F \text{ means end } F, \]
\[ \{F,F'\} \text{ means } F \text{ but not } F' \text{ since then} \]
past time operators

Basic – Propositional Calculus plus

- $\circ F$ (prev.)  $F \mathcal{S} F'$ (since)  $\box F$ (always)  $\Diamond F$ (eventually) in past

Special – Suitable for monitoring (MaC)

- $\uparrow F$ - start of $F$
- $\downarrow F$ - end of $F$
- $[F,F']$ - $F$ but not $F'$

Theorem: $\uparrow, \downarrow, [\_ ,\_ ]$ and $\circ, \mathcal{S}$ defined in terms of each other! F.ex:

\[
[F,F'] = (\neg F') \mathcal{S} F,
\]
\[
\uparrow F = F \land \circ \neg F
\]

Safety property: $\box F$, where $F$ is a past time LTL formula
semantics

• standard semantics

\[ e_1 e_2 \ldots e_{n-1} e_n \models [F,F'] \text{ iff } \]
there is \( 1 \leq i \leq n \) such that \( e_1 e_2 \ldots e_i \models F \) and
for all \( i \leq j \leq n \), \( e_1 e_2 \ldots e_j \not\models F' \)

• recursive Semantics

\( t e \models [F,F'] \text{ iff } \]
\[ t e \not\models F' \text{ and } ( t e \models F \text{ or } t \models [F,F'] ) \]
**dynamic programming algorithm**

Formula: $\uparrow_{\text{Landing}} \rightarrow \{\text{Approved, } \downarrow_{\text{Radio}}\}$

Trace: $e_1 e_2 \ldots e_{n-1} e_n$

**Step 4: Further Optimization**

Global bits $b_1, b_2, b_3$

Temporary bits $t_1, t_2, t_3$

$t_1 = \text{holds(Radio)}$
$t_2 = \text{holds(Landing)}$
$t_3 = (\text{not } b_1 \text{ or } t_1) \text{ and }$
$(\text{holds(Approved) or } b_3)$
if $(t_2 \text{ and not } (b_2 \text{ or } t_3))$
then “error”

$(b_1, b_2, b_3) = (t_1, t_2, t_3)$

Time: $\leq 6$ CPU clocks!
the monitoring matrix

<table>
<thead>
<tr>
<th></th>
<th>MATCH/VALIDATE</th>
<th>FAIL/VIOLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUFFIX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
comparing ERE with LTL

**property:**
An $p$ should be followed by a $q$

$$\text{[]}(p \rightarrow <>q)$$

<table>
<thead>
<tr>
<th></th>
<th>TOTAL + FAIL</th>
<th>SUFFIX + MATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTLTL</td>
<td>$\text{[]}(p \rightarrow !E \cup q)$</td>
<td></td>
</tr>
<tr>
<td>PTLTL</td>
<td>$E$ implies $((! &lt;\Rightarrow p) \lor (! p) S \ q))$</td>
<td></td>
</tr>
<tr>
<td>ERE</td>
<td>$(q^* (\varepsilon + (p \ p^* q)))^* E$</td>
<td></td>
</tr>
<tr>
<td>ERE</td>
<td>$p \ p^* E$</td>
<td></td>
</tr>
</tbody>
</table>
Past Time LTL syntax

BNF below is extended with {p} for zero or more and [p] for zero or one repetitions of p

```plaintext
<PTLTL Plugin Input> ::= {"event" <Event Name>}"ptltl:"<PTLTL>

<PTLTL> ::= "true" | "false" | <Event Name> | <PTLTL> "and" <PTLTL>
<PTLTL> "or" <PTLTL> | "not" <PTLTL> | <PTLTL> "implies" <PTLTL>

!-- Normal boolean constants and operators

"(*)" <PTLTL> | <PTLTL> "S" <PTLTL>
"<>" <PTLTL> | "[*]" <PTLTL> | [<PTLTL>,<PTLTL>]

!-- temporal operators

<LOGIC State> ::= "violation" | "validation"
```

for writing handlers, for example: @violation{...}
<table>
<thead>
<tr>
<th>formula</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>a</td>
<td>event a occurs</td>
</tr>
<tr>
<td>P and Q</td>
<td>Boolean conjunction</td>
</tr>
<tr>
<td>P or Q</td>
<td>Boolean disjunction</td>
</tr>
<tr>
<td>not P</td>
<td>Boolean negation</td>
</tr>
<tr>
<td>P implies Q</td>
<td>Boolean implication</td>
</tr>
<tr>
<td>(*) P</td>
<td>in previous state P</td>
</tr>
<tr>
<td>P S Q</td>
<td>P since Q</td>
</tr>
<tr>
<td>&lt;*&gt; P</td>
<td>P sometime in the past</td>
</tr>
<tr>
<td>[*] P</td>
<td>always in the past P</td>
</tr>
<tr>
<td>[P,Q}</td>
<td>P sometime in the past and Q never since then</td>
</tr>
</tbody>
</table>
Future Time LTL syntax

for writing handlers, for example:
@violation{...}
<table>
<thead>
<tr>
<th>Boolean Truth Values</th>
<th>Boolean Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>a</td>
<td>event a occurs</td>
</tr>
<tr>
<td>! P</td>
<td>Boolean negation</td>
</tr>
<tr>
<td>P ∧ Q</td>
<td>Boolean conjunction</td>
</tr>
<tr>
<td>P ∨ Q</td>
<td>Boolean disjunction</td>
</tr>
<tr>
<td>P ++ Q</td>
<td>Boolean exclusive disjunction</td>
</tr>
<tr>
<td>P -&gt; Q</td>
<td>Boolean implication</td>
</tr>
<tr>
<td>P &lt;-&gt; Q</td>
<td>Boolean bi-implication</td>
</tr>
<tr>
<td>[] P</td>
<td>always P</td>
</tr>
<tr>
<td>&lt;&gt; P</td>
<td>eventually P</td>
</tr>
<tr>
<td>o P</td>
<td>next P</td>
</tr>
<tr>
<td>P U Q</td>
<td>P until Q</td>
</tr>
</tbody>
</table>
MOP Past Time Linear Temporal Logic (PTLTL) Plugin

This page discusses the MOP past-time linear temporal logic (PTLTL) plugin. It also allows to type in an PTLTL specification and then generate a monitor for it. PTLTL is a logic for specifying properties of reactive and concurrent systems. PTLTL provides temporal operators that refer to the past states of an execution trace relative to a current point of reference. The logic plugin here is based on an rewriting-based algorithm for generating an optimized monitoring program from an PTLTL formula.

Note: If there are any technical difficulties please alert pmmerdiv@cs.uiuc.edu

Choose an example:
- HasNext
- NoDisableWhileConverting
- Resource
- SafeDivModify
- SafeIterator

Please press the Run button once and wait. It may take a few seconds to run PTLTLPlugin. The execution of PTLTLPlugin using this web interface is limited to 2 minutes of CPU time and 500 MB of RAM.
Note: If there are any technical difficulties please alert pmereidt@cs.uiuc.edu

Choose an example:

- HasNext
- NoDisableWhileConverting
  - Resource
  - SafeDivMod
  - SafeIterator

Event p

Event q

Event E

\( \text{ptl} : E \implies (\neg a \lor p) \lor (\neg a \land q) \)

Number of generated monitors (since 14 December 2008): 1044 (MOP), 62 (PTLTL), 49 (Language-neutral PTLTL)

The textbox below shows the generated monitor; see PTLTL Plugin Output Syntax (opens new window)

Text Output:

```c
// 2 bits
// initialization of monitor state (nil
b[0] := false
b[1] := false);
// sequential monitoring code (b[0] := 'q or not 'p and b[0]) ;
b[1] := 'p or b[1]) ;
output('E or not b[1] or b[0])
```
This page discusses the MOP future-time linear temporal logic (FTLTL) plug-in. It also allows to type in an FTLTL specification and then generate a monitor for it. FTLTL is a logic for specifying properties of reactive and concurrent systems. FTLTL provides temporal operators that refer to the future states of an execution trace relative to a current point of reference. The logic plugin here is based on a rewriting-based algorithm for generating an optimized monitoring program from an FTLTL formula.

Note: if there are any technical difficulties please alert pmored@cs.uuc.edu
This page discusses the MOP future-time linear temporal logic (FTLTL) plugin. It also allows to type in an FTLTL specification and then generate a monitor for it. FTLTL is a logic for specifying properties of reactive and concurrent systems. FTLTL provides temporal operators that refer to the future states of an execution trace relative to a current point of reference. The logic plugin here is based on a rewriting-based algorithm for generating an optimized monitoring program from an FTLTL formula.

Choose an example:
- HashSet
- TrafficLight

```
event p
event q
event E

fttl : \[\lbrack p \rightarrow \lbrack E \cup q\rbrack \rbrack
```

Number of generated monitors (since 14 December 2008): 1026 (MOP), 49 (FTLTL), 23 (Language-neutral FTLTL)

The textbox below shows the generated monitor; see FTLTL Plugin Output Syntax (opens new window)

Done.

Text Output:

```
Starting state: 1
Transitions:
switch (state) {
case 1: state <- q ? 1 : p ? 1 : E ? false : 1
  case 2: state <- q ? 1 : E ? false : 2
}
```
properties of Java library APIs

R1: There should be no two calls to `next()` without a call to `hasNext()` in between, on the same iterator.
public class Test2 {
    public static void main(String[] args) {
        Vector<Integer> v1 = new Vector<>();
        Vector<Integer> v2 = new Vector<>();
        v1.add(1); v1.add(3); v2.add(5); v2.add(7);
        Iterator it1 = v1.iterator();
        Iterator it2 = v2.iterator();
        int sumFirstTwo_v2 = 0;
        if (it2.hasNext())
            sumFirstTwo_v2 += (Integer)it2.next();
        if (it1.hasNext())
            sumFirstTwo_v2 += (Integer)it2.next();
        System.out.println("sumFirstTwo_v2 = " + sumFirstTwo_v2);
    }
}

our hasNext example again

should have been:
if(it2.hasNext())
recall ERE 1

i) **suffix** trace semantics

ii) looking for **match**

note: match = validation, fail = violation (MOP has changed)
recall ERE 2

(hasNext hasNext* next)*

i) total trace semantics

ii) looking for violation
LTL

next implies (*)(!next S hasnext)

PTLTL

next implies (*)[hasnext,next]

next implies (*)([hasnext,next} or !<*>next)

FTLTL

[](next -> o(!next U hasnext))

i) total trace semantics

ii) looking for violation
package paramere hasNext;

import java.io.*;
import java.util.*;

HasNext(Iterator i) {
    event hasNext before(Iterator i) : call(* Iterator.hasNext()) && target(i) {}
    event next before(Iterator i) : call(* Iterator.next()) && target(i) {}

    ptttl : next implies (*)(!next S hasNext)
}

@violation {
    System.err.println("! hasNext not called before next");
    ___RESET;
}
}
package paramere.hasnext;

import java.io.*;
import java.util.*;

HasNext(Iterator i) {
    event hasNext before(Iterator i) : call(* Iterator.hasNext()) && target(i) {}
    event next before(Iterator i) : call(* Iterator.next()) && target(i) {}
}

@violation{
    System.err.println("! hasNext not called before next");
    ___RESET;
}
}
properties of Java library APIs

R2: An enumeration should not be propagated after the underlying vector has been changed.

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hasMoreElements()</code></td>
<td>Tests if this enumeration contains more elements.</td>
</tr>
<tr>
<td><code>nextElement()</code></td>
<td>Returns the next element of this enumeration object has at least one more element to provide.</td>
</tr>
</tbody>
</table>
Vector v1 = new Vector();
Vector v2 = new Vector();
v1.add(1); v1.add(2); v2.add(4); v2.add(5);
Enumeration e1 = v1.elements();
Enumeration e2 = v1.elements();
Enumeration e3 = v2.elements();
while(e1.hasMoreElements()) print(e1.nextElement());
v1.add(99);
while(e2.hasMoreElements()) print(e2.nextElement());
while(e3.hasMoreElements()) print(e3.nextElement());
package paramere.safeenum;

import java.io.*;
import java.util.*;

suffix SafeEnumeration(Vector v, Enumeration e) {
    event create after(Vector v) returning(Enumeration e) :
        call(Enumeration Vector.elements()) && target(v) {}

    event updatesource before(Vector v):
        (call(* Vector.remove*(..)) || call(* Vector.add*(..)) ) && target(v) {}

    event next before(Enumeration e):
        call(* Enumeration.nextElement()) && target(e) {}

    ere : create next* updatesource updatesource* next

    @match {
        System.out.println("improper enumerator usage");
    }
}
package paramere.safeenum;

import java.io.*;
import java.util.*;

SafeEnumeration(Vector v, Enumeration e) {
    event create after(Vector v) returning(Enumeration e) :
        call(Enumeration Vector.elements()) && target(v) {}

    event updatesource before(Vector v) :
        (call(* Vector.remove*(..)) || call(* Vector.add*(..))) && target(v) {}

    event next before(Enumeration e) :
        call(* Enumeration.nextElement()) && target(e) {}

    *ftltl* (create -> [](updatesource -> !<>next))

    @violation {
        System.out.println("improper enumerator usage");
    }
}
end