temporal logic in JavaMOP

CS 119

propositional logic extended with temporal operators referring to past and future
past time properties

- If A happens now
  B must have happened

\(\square (A \rightarrow \diamond B)\)
future time properties

• If A happens now
  B must happen

\[ \square (A \to \diamond B) \]
Instances of MOP

- JavaMOP
- BusMOP
- HardwareMOP
- ...
http://fsl.cs.uiuc.edu/index.php/Special:PTLTLPlugin

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.

Input

PTLTL formulae use the following syntax, which is defined in BNF:

\[
\text{<PTLTL Plugin>} ::= \text{<Event Header>} \text{<PTLTL>}
\text{<PTLTL>} ::= \text{"formula:"} \text{<PTLTL Formula>}
\]

MOP Matrix: a clickable map of MOP pages.
http://fsl.cs.uiuc.edu/index.php/Special:FTLTLPlugin

MOP Future Time Linear Temporal Logic (FTLTL) Plugin

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 useful for stating properties about reactive and concurrent systems. It provides temporal operators that refer to the future/remaining part of an execution trace. The logic plugin here is based on a rewriting-based algorithm for generating a minimal special observer finite state machine (FSM), or an automaton, from an FTLTL formula.

Input

FTLTL formulae use the following syntax, which is defined in BNF:

```plaintext
<FTLTL Plugin> ::= <Event Header> <FTLTL>
<FTLTL> ::= "formula:" <FTLTL Formula>
```
# 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

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

- does not work
advantages of temporal logic

• some properties can be stated more succinctly
• where translation to automata will result in state explosion
• it will of course be a debate at a practical engineering level what notation is most suitable in practice
• everybody understand state machines right away and temporal properties can be hard to write and read for complex scenarios
past time and future time

temporal logic

semantics and algorithms
PathExplorer - overview
PathExplorer
the observer

```
paxmodules
module datarace = 'java pax.Datarace';
module deadlock = 'java pax.Deadlock';
module temporal = 'java pax.Temporal spec';
end
```

Event stream

Dispatcher

- datarace
- deadlock
- temporal

Specification Based Monitoring

warning ...

warning ...

warning ...
future time

semantics and algorithm
getting the events (Java)

```
class Light{
    void goRed(){
        color = 1;
    }
}
```

```
Instrumentation Script

predicate red    = (Light.color == 1);
predicate yellow = (Light.color == 2);
predicate green  = (Light.color == 3);

property p = [](green -> !red U yellow);
```

Diagram:
- **Program**: class Light
  - `void goRed(){
    color = 1;
  }
  ...
- **Bytecode**
  - Compiled from program
- **Instrumented Bytecode**
  - Instrumented bytecode
- **Java Virtual Machine**
  - Executed with instrumented bytecode
- **Instrumentation Script**
  - `predicate red    = (Light.color == 1);
predicate yellow   = (Light.color == 2);
predicate green    = (Light.color == 3);

  property p = [](green -> !red U yellow);`
monitoring Future Time LTL

Syntax – Propositional Calculus plus

- $\circ F$ (next)  $\Box F$ (always)  $\Diamond F$ (eventually)  $F \mathbin{U} F'$ (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>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p{e}$</td>
<td>is the atomic predicate $p$ true on $e$ ?</td>
</tr>
<tr>
<td>$(F \mathbin{op} F'){e}$</td>
<td>$F{e} \mathbin{op} F'{e}$</td>
</tr>
<tr>
<td>$(\circ F){e}$</td>
<td>$F{e}$</td>
</tr>
<tr>
<td>$(\Box F){e}$</td>
<td>$F{e} \land (\Box F)$</td>
</tr>
<tr>
<td>$(\Diamond F){e}$</td>
<td>$F{e} \lor (\Diamond F)$</td>
</tr>
<tr>
<td>$(F \mathbin{U} F'){e}$</td>
<td>$F'{e} \lor (F{e} \land (F \mathbin{U} F'))$</td>
</tr>
</tbody>
</table>
Future Time LTL - example

Event stream: $\text{red} \times \times \times \times \times \times \times \times \times \times \times \text{red}$ ...

$\Box (\text{green} \rightarrow \neg \text{red} U \text{yellow}) \{\text{yellow}\}$

$\downarrow^*$

$\Box (\text{green} \rightarrow \neg \text{red} U \text{yellow}) \{\text{green}\}$

$\downarrow^*$

$((\neg \text{red} U \text{yellow}) \land \Box (\text{green} \rightarrow \neg \text{red} U \text{yellow}))\{\text{yellow}\}$

$\downarrow^*$

$\Box (\text{green} \rightarrow \neg \text{red} U \text{yellow}) \{\text{green}\}$

$\downarrow^*$

$((\neg \text{red} U \text{yellow}) \land \Box (\text{green} \rightarrow \neg \text{red} U \text{yellow}))\{\text{red}\}$

$\downarrow^*$

$(\text{yellow}\{\text{red}\} \lor \neg\text{red}\{\text{red}\} \land \neg \text{red} U \text{yellow}) \land \ldots \Rightarrow \text{false} \land \ldots \Rightarrow \text{false}$

Formula was violated!
timed temporal logic

• Add real time (RTL, MiTL, timed automata, etc.)

\( \blacksquare (\text{start } \rightarrow ^{5} \text{stop}) \)

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

• 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 …
generating FSM observers

- There are applications where
  - Little monitoring overhead is allowed (real time)
  - Few resources available for monitoring
- **Challenge:** efficient and simple monitors!
- *Finite State Machine* observers can be built from formulae *before* monitoring
  - Lower runtime overhead
    - No inferences needed
    - Only some atomic predicates need to be evaluated
  - Higher start time overhead
building a minimal BTT_FSM

• Idea
  – Do the rewrites for all possible values of predicates
  – Get a finite state machine
    • Nodes are LTL formulae
    • Optimize using a validity checker (F ↔ F’ : one state)
    • Edges are propositions
    • Assign numbers to states
    • Replace edges by 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>

![Diagram of Binary Transition Tree FSM](image-url)
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

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

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

- $\Diamond F$ (prev.) $F \land F'$ (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 $\Diamond, \Box, S$ defined in terms of each other!

- $\Diamond F = (F \rightarrow \neg \uparrow F) \land (\neg F \rightarrow \downarrow F), [F,F') = (\neg F') S F, \uparrow F = F \land \Diamond \neg F$

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

• standard semantics

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

• recursive Semantics

\[ te \models [F,F'] \text{ iff } \ 
\begin{cases} 
  e \not\models F' \text{ and } (t e \models F \text{ or } t \models [F,F']) 
\end{cases} \]
Dynamic programming

*Optimal substructure* means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem. For example, the shortest path to a goal from a vertex in a graph can be found by first computing the shortest path to the goal from all adjacent vertices, and then using this to pick the best overall path, as shown in Figure 1. In general, we can solve a problem with optimal substructure using a three-step process:

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

The subproblems are, themselves, solved by dividing them into sub-subproblems, and so on, until we reach some simple case that is solvable in constant time.

To say that a problem has overlapping subproblems is to say that the same subproblems are used to solve many different larger problems. For example, in the Fibonacci sequence, $F_3 = F_1 + F_2$ and $F_4 = F_2 + F_3$ — computing each number involves computing $F_2$. Because both $F_3$ and $F_4$ are needed to compute $F_5$, a naive approach to computing $F_5$ may end up computing $F_2$ twice or more. This applies whenever overlapping subproblems are present: a naive approach may waste time recomputing optimal solutions to subproblems it has already solved.

In order to avoid this, we instead save the solutions to problems we have already solved. Then, if we need to solve the same problem later, we can retrieve and reuse our already-computed solution. This approach is called *memoization* (not *memorization*, although this term also fits). If we are sure we won’t need a particular solution anymore, we can throw it away to save space. In some cases, we can even compute the solutions to subproblems we know that we’ll need in advance.

In summary, dynamic programming makes use of:

- Overlapping subproblems
- Optimal substructure
- Memoization

**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 \( t_1, t_2, t_3 \)

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

**Time:** \( \leq 6 \) CPU clocks!
future time versus past time
liveness vs. safety

- Monitoring liveness properties ...

Formula: \( \varphi = \square (F \rightarrow \Diamond F') \)

Trace: \( F \ F' \ F \ F' \ F \ F' \ldots \ F \ F' \ F \bot \) violates \( \varphi \)

Trace: \( F \ F \ F \ F \ F \ F \ F' \ F \bot \ldots \) \( F \ F' \ F \bot \) does not violate \( \varphi \)

... keep statistical information for eventualities?

- Focus on safety: \( <>F \) can never be violated in JavaMOP
  - Often easier to express using past time LTL

\( \square F, \) where \( F \) is a past time LTL formula

- \( \square F \) equivalent to “monitor \( F \)”
past time

In JavaMOP
syntax

```
<PTLTL Plugin> ::= <Event Header> <PTLTL>
<PTLTL> ::= "formula:" <PTLTL Formula>
<PTLTL Formula> ::= "true" | "false"
| <Event Name> // events are atomic propositions
| "!" <PTLTL Formula> // negation
| <PTLTL Formula> \/* \* <PTLTL Formula> // conjunction
| <PTLTL Formula> "\" <PTLTL Formula> // disjunction
| <PTLTL Formula> "++" <PTLTL Formula> // XOR: eXclusive OR
| <PTLTL Formula> "->" <PTLTL Formula> // implies
| <PTLTL Formula> "<->" <PTLTL Formula> // if and only if
| "[*]" <PTLTL Formula> // always in the past
| "<*>" <PTLTL Formula> // eventually in the past
| "(*)" <PTLTL Formula> // previous
| <PTLTL Formula> "Ss" <PTLTL Formula> // strong since
| <PTLTL Formula> "Sw" <PTLTL Formula> // weak since
| "[" <PTLTL Formula> ";" <PTLTL Formula> "]s" // strong interval
| "[" <PTLTL Formula> ";" <PTLTL Formula> "]w" // weak interval
| "start("<PTLTL Formula>")" // a formula just starts to hold
| "end("<PTLTL Formula>")" // a formula which did hold just stopped holding
```
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.
class Test {
    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 sum = 0;
        if(it2.hasNext())
            sum += (Integer)it2.next();
        if(it1.hasNext())
            sum += (Integer)it2.next();
        System.out.println("sum(v2) = " + sum);
    }
}
recall the regular expression specification

/*@
partial centralized
scope = global
logic = ERE

HasNext(Iterator i)
{
    event hasnext<i> : end(call(* i.hasNext()));
    event next<i> : begin(call(* i.next()));
    formula : next next
}

validation handler{
    System.err.println("*** call hasNext() before next()"));
}

@*/
PTLTL Property in JavaMOP

```java
package ltl.hasnext;
import java.util.*;

/*@ 
centralized scope = global 
logic = PTLTL

HasNext(Iterator i) {
    event hasnext<i> : end(call(* i.hasNext()));
    event next<i> : begin(call(* i.next()));

    formula : next /
    \( (*)\text{next} \)
}

validation handler {
    System.err.println("*** call hasNext() before next()");
}
@*/

Warning: PTLTL and validation does not currently work in JavaMOP. Likely a bug.

partial matching does not make sense in PTLTL.

it does not look too different in this particular case, but PTLTL obviously does not show more succinct here.
PTLTL Property in JavaMOP

package ltl.hasnext;
import java.util.*;

/*@ 
centralized
scope = global
logic = PTLTL

HasNext(Iterator i) {
    event hasnext<i> : end(call(* i.hasNext()));
    event next<i> : begin(call(* i.next()));

    formula : !(next ∧ (*next)
}

violation handler {
    System.err.println("*** call hasNext() before next()";
}
@*/

In PTLTL:
validation = not violation

partial matching does not make sense in PTLTL.

less attractive but similarity to regular exp can still be noticed.
recall this other regular expression

\[(\text{hasNext hasNext}^* \text{ next})^*\]

which was slightly too strong, but let’s try to emulate it

i) total trace semantics
ii) looking for violation
PTLTL Property in JavaMOP - attempt 1

```java
package ltl.hasNext;
import java.util.*;

/*@ centralized
scope = global
logic = PTLTL
HasNext(Iterator i) {
    event hasnext<i> : end(call(* i.hasNext()));
    event next<i> : begin(call(* i.next()));

    formula : next -> <>hasnext
}

violation handler{
    System.err.println("*** call hasNext() before next()");
}
@*/
```

- no longer partial
- not correct, will allow for example: `it.hasNext(); it.next(); it.next()`
package ltl.hasnext;
import java.util.*;

/*@ 
centralized 
scope = global 
logic = PTLTL 
HasNext(Iterator i) { 
event hasnext<i> : end(call(* i.hasNext()));
event next<i> : begin(call(* i.next()));

  formula : next -> (*)(!next Ss hasnext)
} 
formula: next -> (*)[hasnext,next]s 
violation handler{
  System.err.println("*** call hasNext() before next()");
}
@*/
properties of Java library APIs

R₂: An enumeration should not be propagated after the underlying vector has been changed.
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());
recall ERE specification

/*@ partial centralized
scope = global
logic = ERE
SafeEnum (Vector v, Enumeration+ e) {
    event create<e, v> : end(call(Enumeration v.elements())) with (e);
    event updatesource<v> :
        end(call(* v.add*(..)))
        V end(call(* v.clear()))
        V end(call(* v.insertElementAt(..))) &&
        V end(call(* v.remove*(..)))
        V end(call(* v.retainAll(..)))
        V end(call(* v.set*(..)));
    event next<e> : begin(call(Object e.nextElement()));

    formula : create next* updatesource updatesource* next
}
validation handler {
    System.out.println("datasource changed during iteration!");
}
@*/
PTLTL Property in JavaMOP - using ‘Ss’

/*@ centralized scope = global logic = PTLTL

SafeEnum (Vector v, Enumeration+ e) {
    event create<e, v> : end(call(Enumeration v.elements())) with (e);
    event updatesource<v> : end(call(* v.add*(..))) ∨ ... ∨ end(call(* v.set*(..)));
    event next<e> : begin(call(Object e.nextElement()));

    formula : next -> !(<*>(updatesource ∧ <*>create))
}

violation handler {
    System.out.println("the datasource is changed during iteration!");
}
@*/
PTLTL Property in JavaMOP - using ‘[_,_]s’

/*@ 
centralized
scope = global
logic = PTLTL

SafeEnum (Vector v, Enumeration+ e) {
    event create<e, v> : end(call(Enumeration v.elements())) with (e);
    event updatesource<v> : end(call(* v.add*(..))) \ ... \ end(call(* v.set*(..)));
    event next<e> : begin(call(Object e.nextElement()));

    formula : next -> [create,updatesource)s
}

violation handler {
    System.out.println("the datasource is changed during iteration!");
}
*/
future time

In JavaMOP
syntax

```
<FTLTL Plugin> ::= <Event Header> <FTLTL>
<FTLTL> ::= "formula:" <FTLTL Formula>
<FTLTL Formula> ::= "true" | "false"

| <Event Name> | // events are atomic propositions
| "!" <FTLTL Formula> | // negation
| <FTLTL Formula> "/\" <FTLTL Formula> | // conjunction
| <FTLTL Formula> "\" <FTLTL Formula> | // disjunction
| <FTLTL Formula> "++" <FTLTL Formula> | // XOR: eXclusive OR
| <FTLTL Formula> "->" <FTLTL Formula> | // implies
| <FTLTL Formula> "<->" <FTLTL Formula> | // if and only if
| "[]" <FTLTL Formula> | // always
| "<>" <FTLTL Formula> | // eventually
| "o" <FTLTL Formula> | // next
| <FTLTL Formula> "U" <Formula> | // until
```
back to this one, now referring to future

R₂: An enumeration should not be propagated after the underlying vector has been changed.
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); next(e1);
while(e2.hasMoreElements()) print(e2.nextElement());
while(e3.hasMoreElements()) print(e3.nextElement());
recall past time formulation

`/*@
 centralized
 scope = global
 logic = PTLTL

 SafeEnum (Vector v, Enumeration+ e) {
   event create<e, v> : end(call(Enumeration v.elements())) with (e);
   event updatesource<v> : end(call(* v.add*(..))) ∨ … ∨ end(call(* v.set*(..)));
   event next<e> : begin(call(Object e.nextElement()));

   formula : next -> [create,updatesource]s
 }

 violation handler {
   System.out.println("the datasource is changed during iteration!");
 }
 @}/*`
SafeEnum (Vector v, Enumeration+ e) {
    event create<e, v> : end(call(Enumeration v.elements())) with (e);
    event updatesource<v> :
        end(call(* v.add*(..))) V
        end(call(* v.clear())) V
        end(call(* v.insertElementAt(..))) V
        end(call(* v.remove*(..))) V
        end(call(* v.retainAll(..))) V
        end(call(* v.set*(..)));
    event next<e> : begin(call(Object e.nextElement()));

    formula : [](create -> o [](updatesource -> [](!next)))
}

violation handler {
    System.out.println("the datasource is changed during iteration!");
}

@*/
end