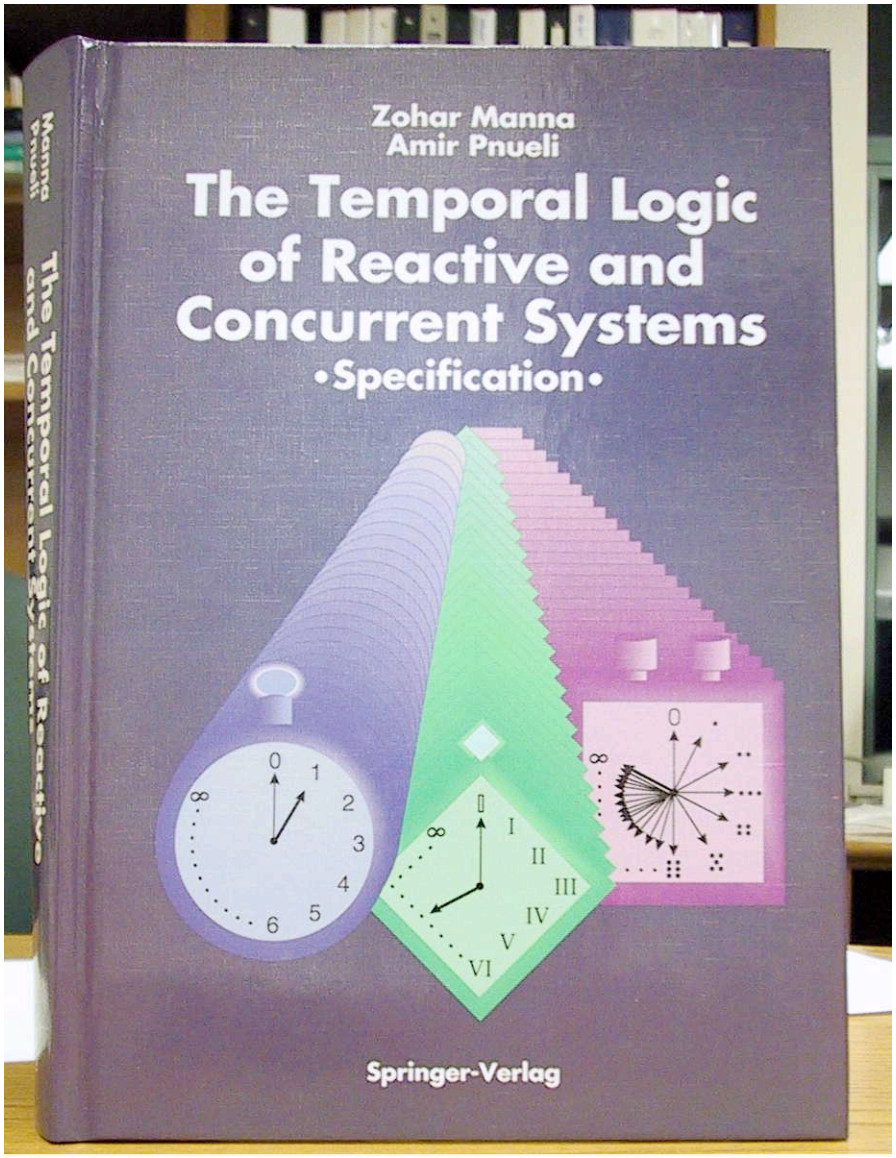
temporal logic in JavaMOP

CS 119

includes slides
by Grigore Rosu

propositional logic extended with
temporal operators referring to
past and future

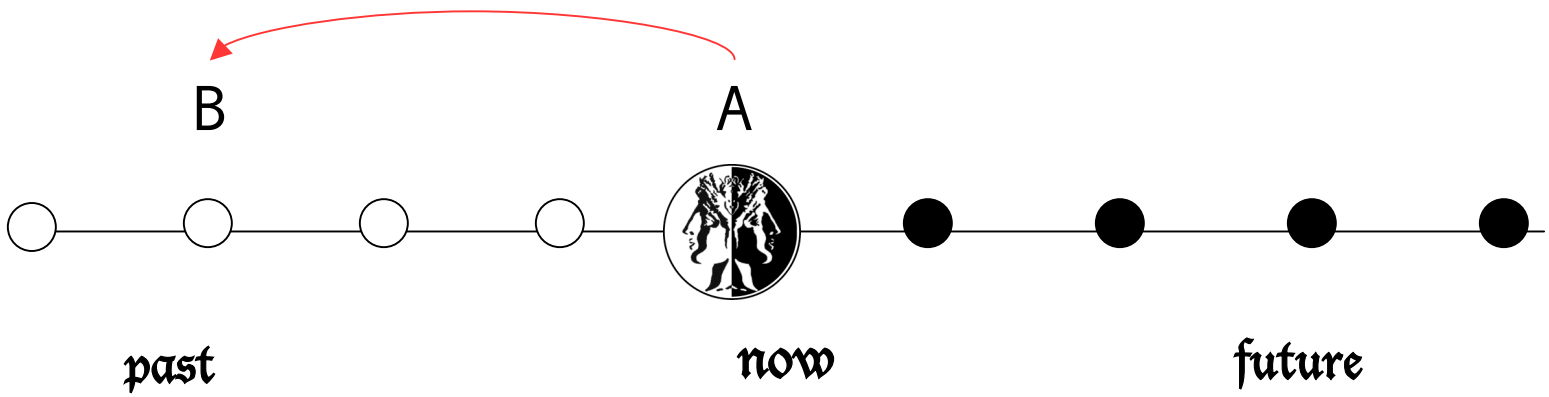

Zohar Manna
Amir Pnueli
The Temporal Logic of Reactive and Concurrent Systems
•Specification•
Springer-Verlag

past time properties

- If A happens now

  B must have happened

$$\Box(A \rightarrow \blacklozenge B)$$

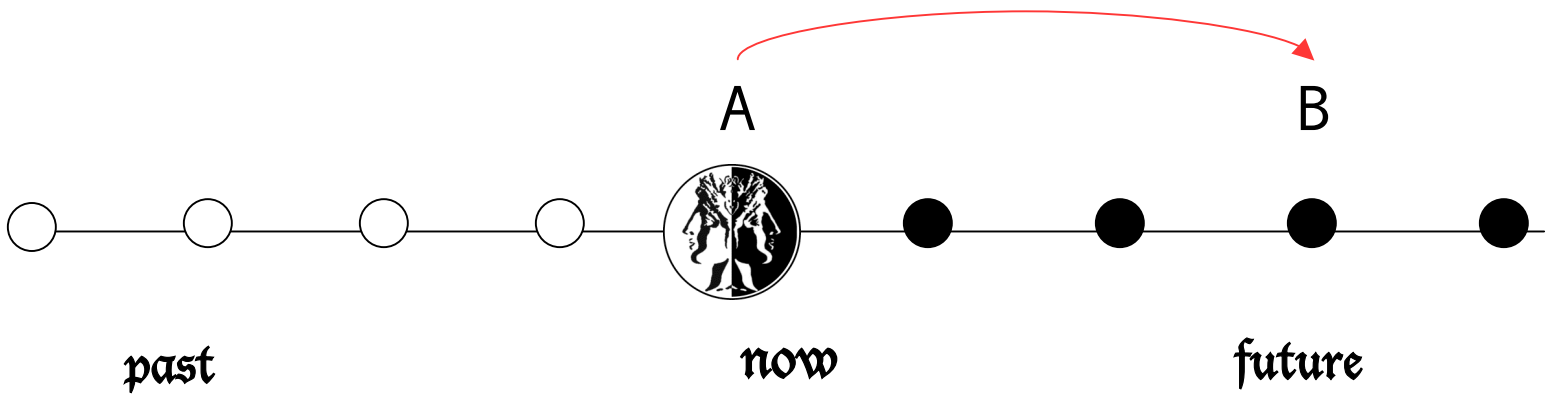


future time properties

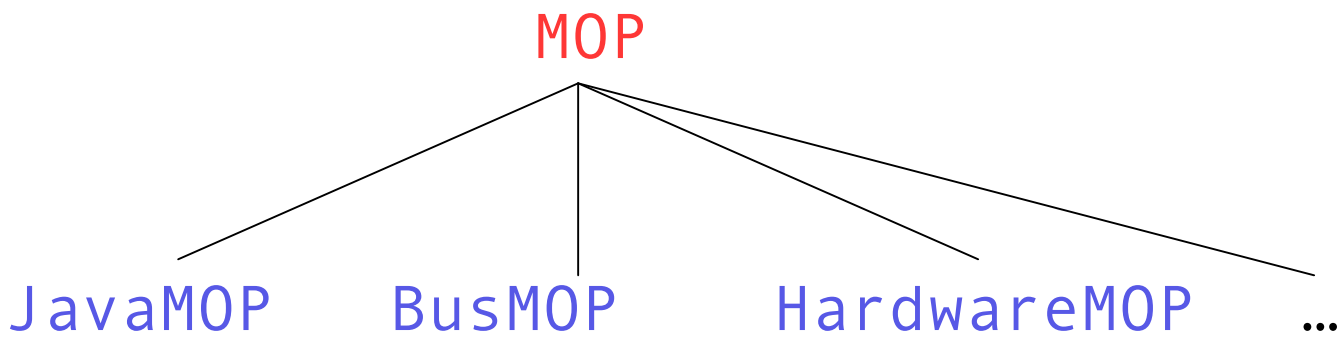
- If A happens now

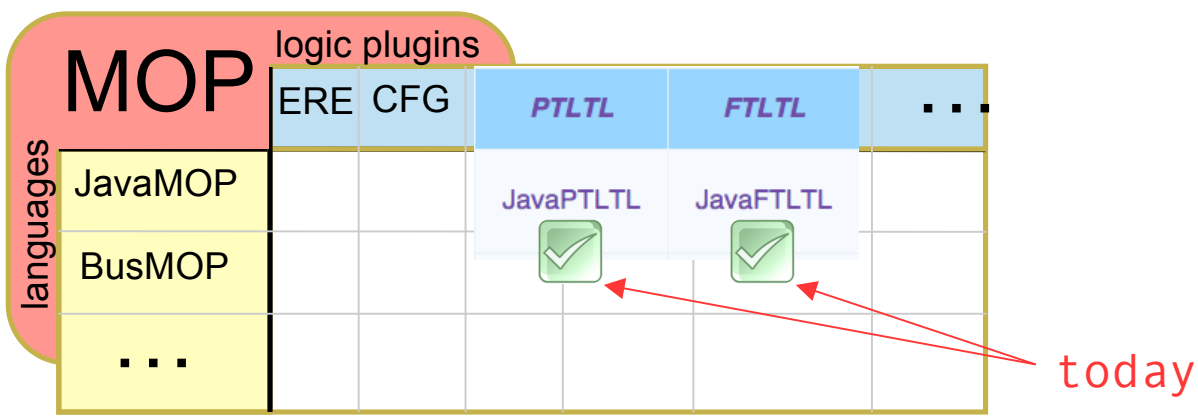
  B must happen

$$\Box(A \rightarrow \Diamond B)$$



Instances of MOP





http://fsl.cs.uiuc.edu/index.php/Special:PTLTLPlugin

MOP Past Time Linear Temporal Logic (PTLTL) Plugin – FSL

http://fsl.cs.uiuc.edu/index.php/Special:PTLTLPlugin

Vibrato Grill Jazz ... etc. | Joergen Ingman | TimeLife.com | Micro Site | Eventseer.net – Home | Free Website Polls | Richest Cou...n the World | Castle for Sale | Programming with C/C++ | Costa Rica, ...Estate Blogs

MOP Past Time Linear Tem...

Log in / create account

Formal Systems Laboratory

navigation

- Main Page
- People
- Projects
- Publications
- News
- Events

search

Go Search

MOP Past Time Linear Temporal Logic (PTLTL) Plugin

shortcuts: MOP

languages \ logic plugins	ERE	CFG	PTLTL	FTLTL	PTCaRet	...
MOP						
JavaMOP	JavaERE	JavaCFG	JavaPTLTL	JavaFTLTL	JavaPTCaRet	...
BusMOP	BusERE	...	BusPTLTL	...	...	...
...	...	...	...	...	...	...

MOP Matrix: a clickable map of MOP pages.

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:

```
<PTLTL Plugin> ::= <Event Header> <PTLTL>
    <PTLTL> ::= "formula:" <PTLTL Formula>
```

http://fsl.cs.uiuc.edu/index.php/Special:FTLTLPlugin

MOP Future Time Linear Temporal Logic (FTLTL) Plugin – FSL

http://fsl.cs.uiuc.edu/index.php/Special:FTLTLPlugin

Vibrato Grill Jazz ... etc. | Joergen Ingman | TimeLife.com | Micro Site | Eventseer.net – Home | Free Website Polls | Richest Cou...n the World | Castle for Sale | Programming with C/C++ | Costa Rica, ...Estate Blogs

MOP Future Time Linear Te...

Log in / create account

Formal System Laboratory

navigation

- Main Page
- People
- Projects
- Publications
- News
- Events

search

Go Search

MOP Future Time Linear Temporal Logic (FTLTL) Plugin

shortcuts: MOP

logic plugins

languages

MOP	ERE	CFG	PTLTL	FTLTL	PTCaRet	...
JavaMOP	JavaERE	JavaCFG	JavaPTLTL	JavaFTLTL	JavaPTCaRet	...
BusMOP	BusERE	...	BusPTLTL	...	...	...
...	...	...	...	...	...	...

MOP Matrix: a clickable map of MOP pages.

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:

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

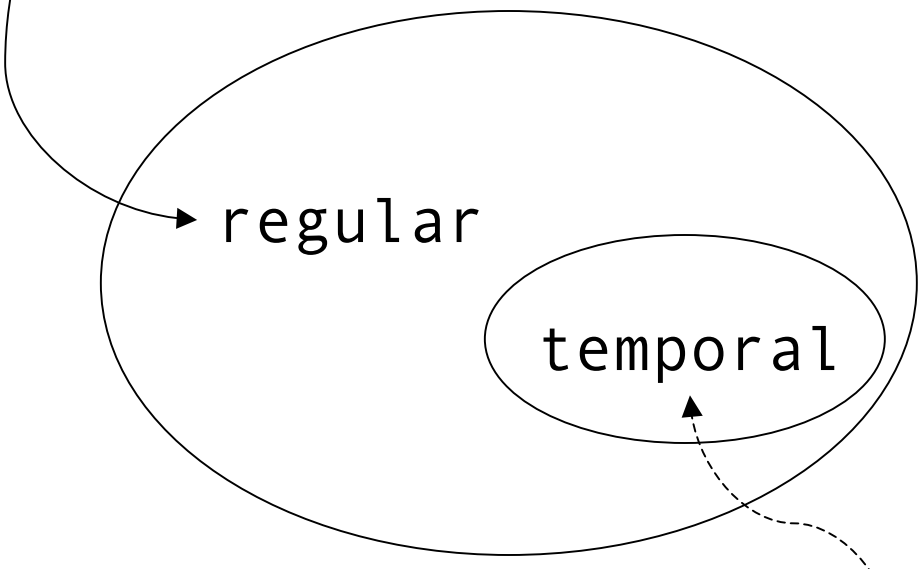
Chomsky's language hierarchy

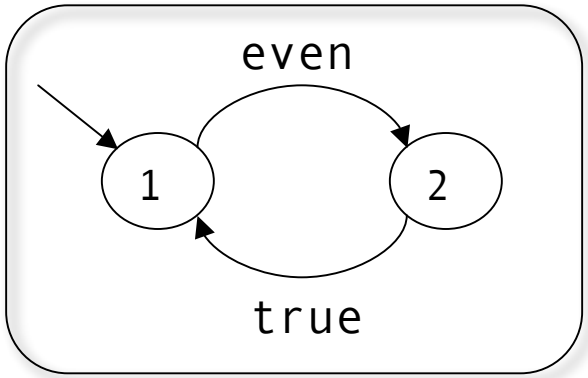
Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic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

http://en.wikipedia.org/wiki/Chomsky_hierarchy

temporal logic for finite traces: subset of regular languages

p is even in every other state





~~even /\ always(even implies (next next even))~~

does not work

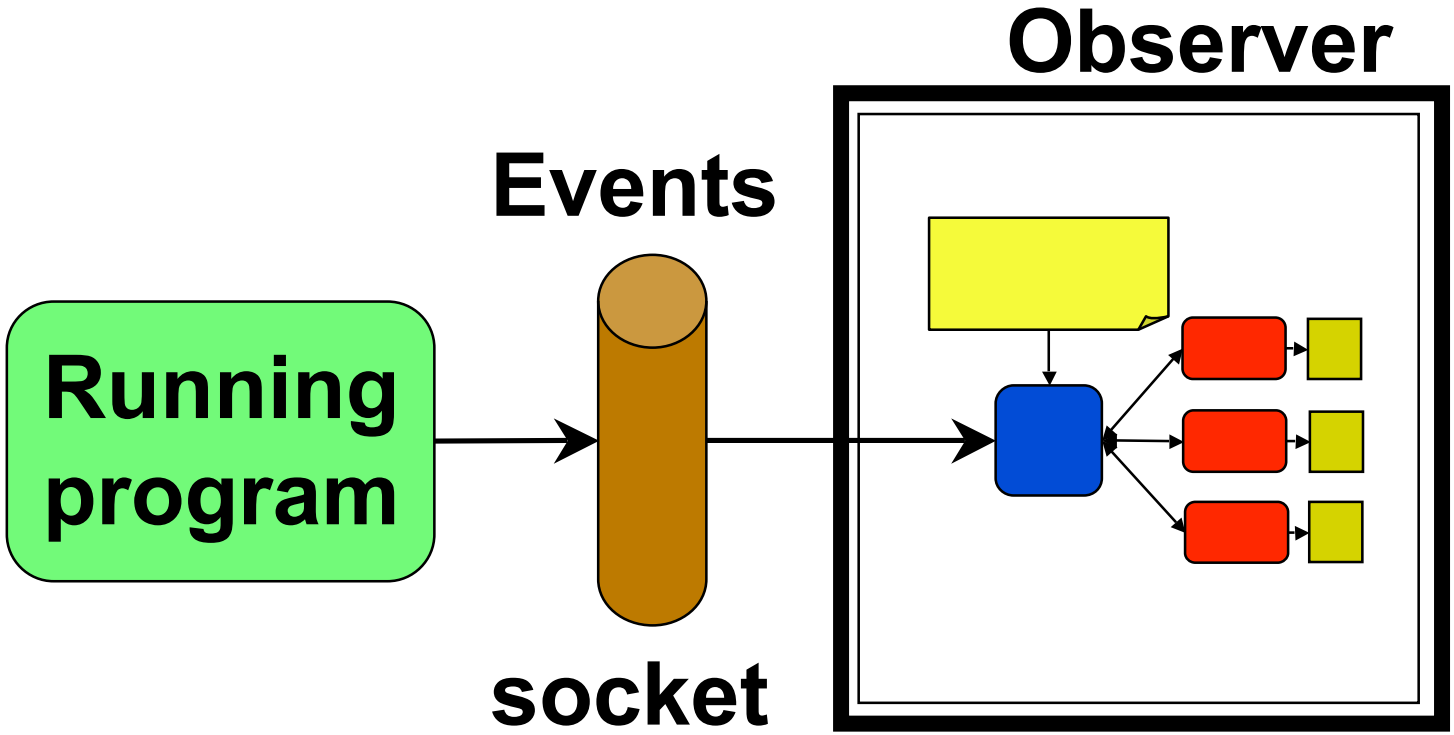
advantages of temporal logic

- 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

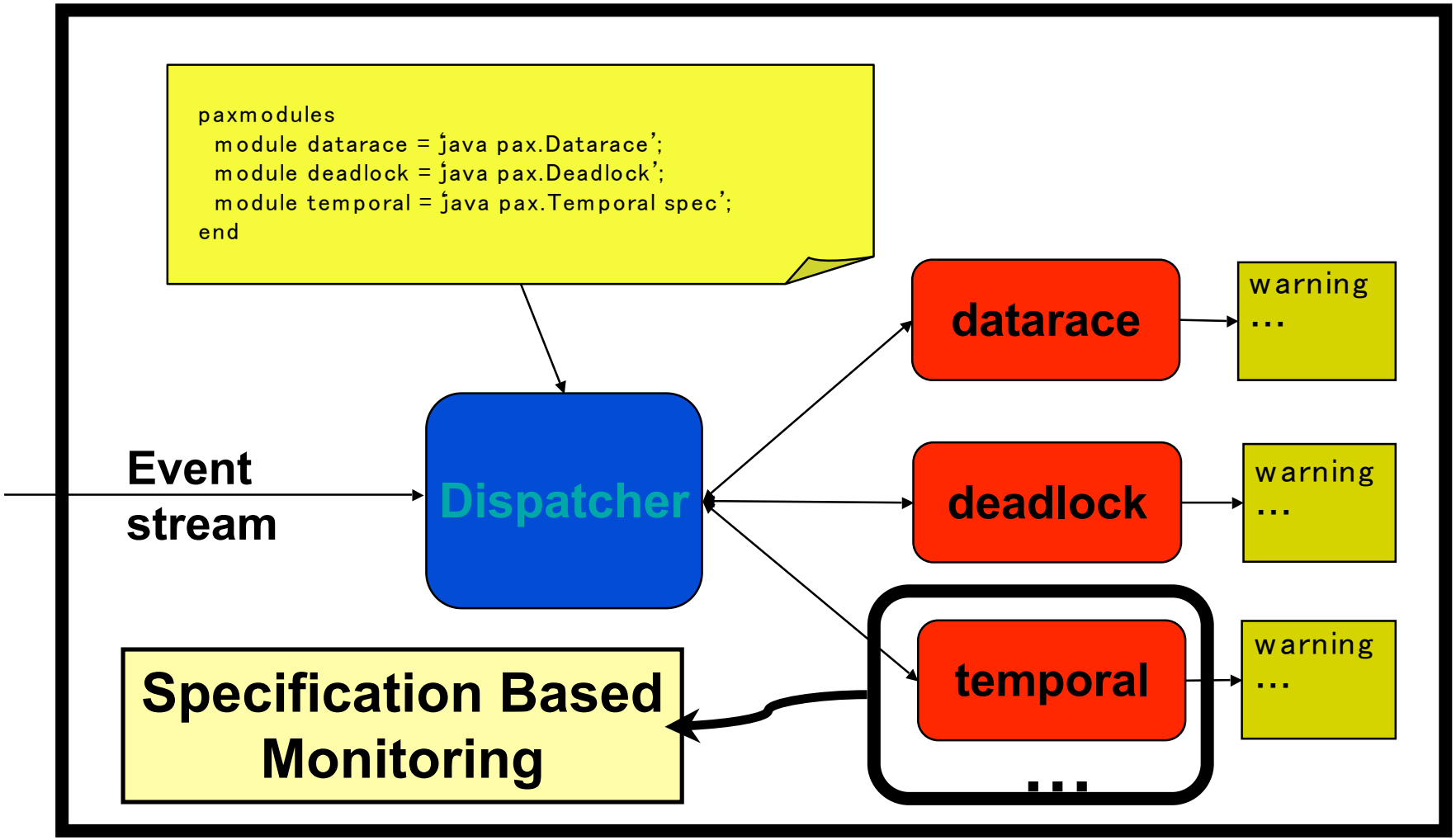
past time and future time temporal logic

semantics and algorithms

PathExplorer - overview



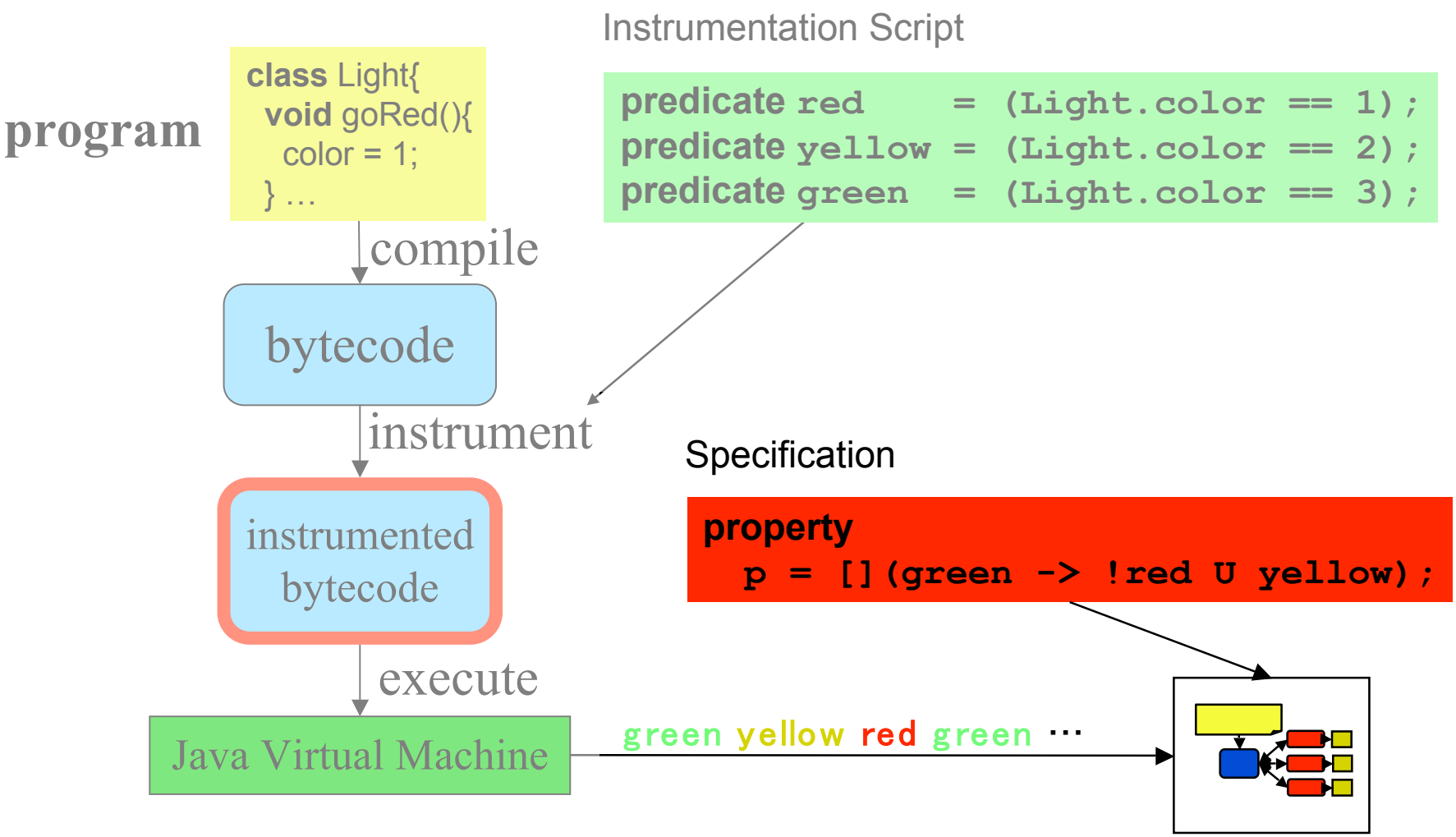
PathExplorer
the observer



future time

semantics and algorithm

getting the events (Java)



monitoring Future Time LTL

Syntax – Propositional Calculus plus

o F (next) □ F (always) ◇ F (eventually) F U F' (until)

Executable Semantics – Rewriting

_{_} : Formula x Event -> Formula ("consume" event e)

F{e} formula that should hold after processing e

p{e}	⇨	is the atomic predicate p true on e ?
(F *op* F')\{e\}	⇨	F{e} *op* F'{e}
(o F){e}	⇨	F
(□ F){e}	⇨	F{e} ∧ (□ F)
(◇ F){e}	⇨	F{e} ∨ (◇ F)
(F U F'){e}	⇨	F'{e} ∨ (F{e} ∧ (F U F'))

Future Time LTL - example

Event stream: **x** red yellow **x** green yellow **x** green **x** red ...

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

$\Downarrow *$

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

$\Downarrow *$

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

$\Downarrow *$

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

$\Downarrow *$

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

$\Downarrow *$

$(\text{yellow}\{\text{red}\} \vee \neg\text{red}\{\text{red}\} \wedge \neg\text{red U yellow}) \wedge \ldots \overset{*}{\Rightarrow} \text{false} \wedge \ldots \overset{*}{\Rightarrow} \text{false}$

Formula was violated!

timed temporal logic

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

$$\Box(\text{start} \rightarrow \Diamond_{5} \text{stop})$$

$$(\Diamond_{t} F)\{e{:}\delta\} \Rightarrow (\delta \leq t) \wedge (F\{e{:}\delta\} \vee \Diamond_{t-\delta} F)$$

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
- <u>Challenge:</u> 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

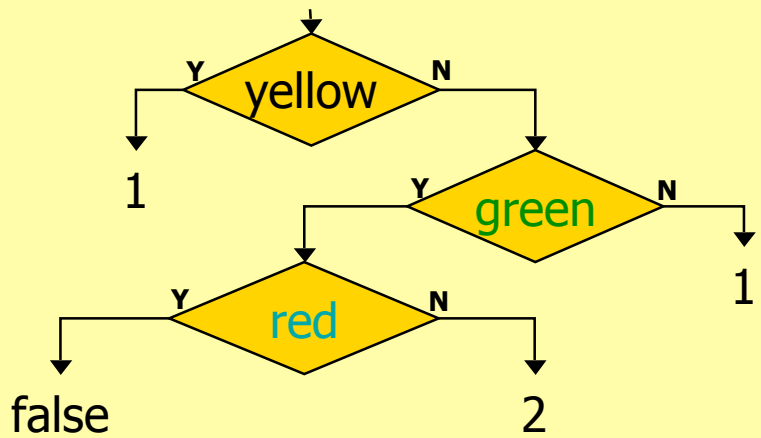
building a minimal BTT_FSM

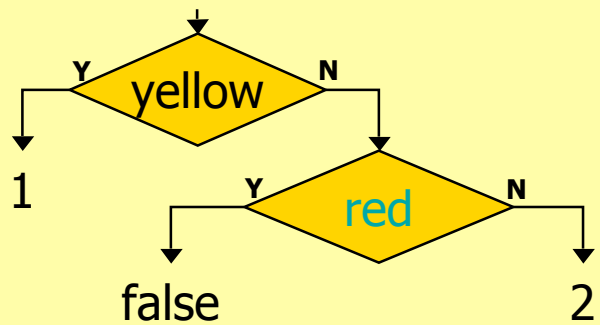
- Idea
  - Do the rewrites for *all* possible values of predicates
  - Get a finite state machine
    - Nodes are LTL formulae
    - Optimize using a validity checker (F ↔ F' : one state)
    - Edges are propositions
    - Assign numbers to states
    - Replace edges by *Binary Transition Trees*

Binary Transition Tree FSM

- We can build minimal FSMs *statically* for LTL

Formula	$\Box$(green $\rightarrow$ $\neg$red U yellow)	
State	1	2
BTT	yellow ? 1 : green ? (red ? false : 2) : 1	yellow ? 1 : (red ? false : 2)





past time

semantics and algorithm

monitoring safety

- Example:

<u>Safe Landing</u>

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

or formally

$\uparrow$Landing $\rightarrow$ [Approved, $\downarrow$Radio)

where (MAC)

$\uparrow$F means <u>start F</u>,

$\downarrow$F means <u>end F</u>,

[F,F') means <u>F butnot F'</u> since then

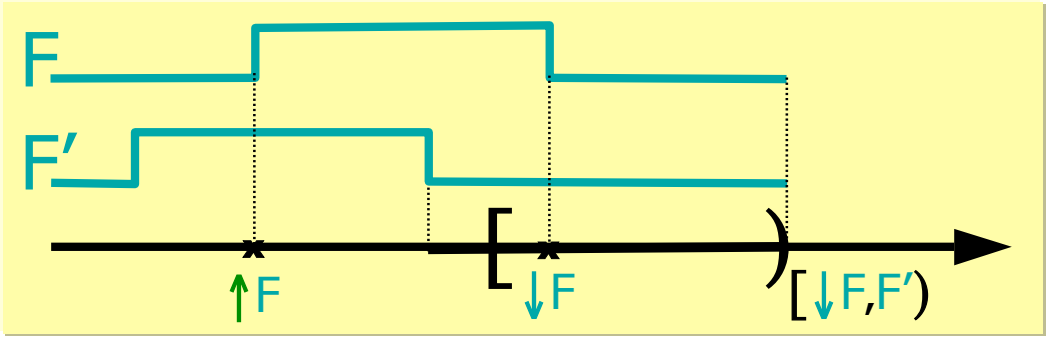
past time operators

Basic – Propositional Calculus plus

$\odot F$ (prev.) $F \; S \; F'$ (since) $\boxdot F$ (always) $\diamondminus F$ (eventually) in past

Special – Suitable for monitoring (MaC)

- $\uparrow F$ - start of F
- $\downarrow F$ - end of F
- $[F,F')$ - F butnot F'



<u>Theorem:</u> $\uparrow, \downarrow, [\_,\_)$ and $\odot, \_S\_$ defined in terms of each other!

- $\odot F = (F \rightarrow \neg\uparrow F) \wedge (\neg F \rightarrow \downarrow F)$, $[F,F') = (\neg F') \; S \; F$, $\uparrow F = F \wedge \odot\neg F$

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

semantics

- standard semantics

$e_1\, e_2 \ldots e_{n-1}\, e_n \models [F, F')$ 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

$t\, e \models [F, F')$ iff
$t\, e \not\models F'$ and ($t\, e \models F$ or $t \models [F, F')$)

article | discussion | edit this page | history

Dynamic programming

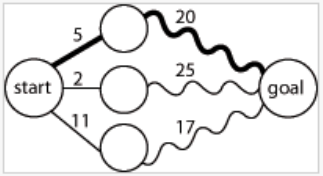


Figure 1. Finding the shortest path in a graph using optimal substructure; a straight line indicates a single edge; a wavy line indicates a shortest path between the two vertices it connects (other nodes on these paths are not shown); the bold line is the overall shortest path from start to goal.

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.

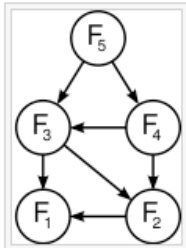


Figure 2. The subproblem graph for the Fibonacci sequence. That it is not a tree but a DAG indicates overlapping subproblems.

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:

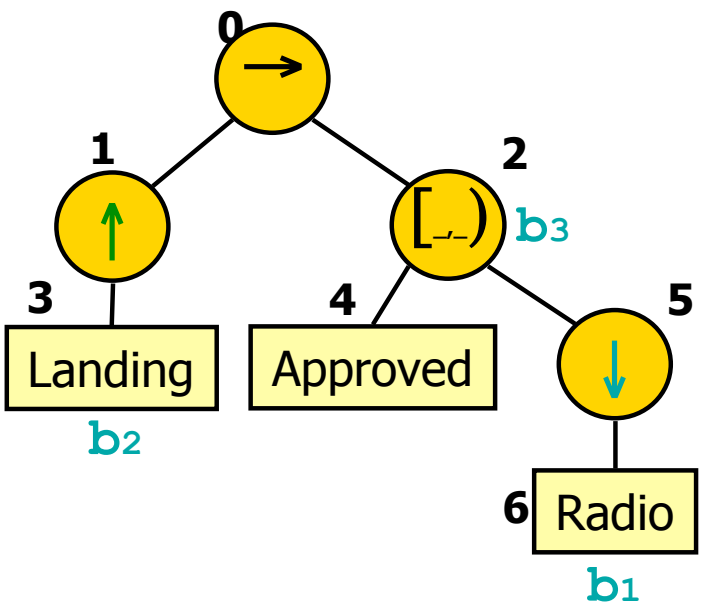
- Overlapping subproblems
- Optimal substructure
- Memoization

http://en.wikipedia.org/wiki/Dynamic_programming

dynamic programming algorithm

Formula: ↑Landing → [Approved, ↓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

```
t1 = holds(Radio)
t2 = holds(Landing)
t3 = (not b1 or t1) and
     (holds(Approved) or b3)
if (t2 and not (b2 or t3))
     then "error"
(b1,b2,b3) = (t1,t2,t3)
```

Time: ≤ 6 CPU clocks!

future time versus past time

liveness vs. safety

- Monitoring liveness properties ...

Formula: $\varphi = \Box (F \rightarrow \Diamond F')$

Trace $F\ F'\ F\ F'\ F\ F' \ldots F\ F'\ F\ \perp$ violates φ

Trace $F\ F\ F\ F\ F\ F \ldots F\ F'\ \perp$ does not violate φ

... keep statistical information for eventualities?

- Focus on safety `<>F can never be violated in JavaMOP`
  - Often easier to express using *past time LTL*

$\Box F$, where F is a past time LTL formula

- $\Box F$ 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

Iterator (Java 2 Platform SE 5.0)

http://java.sun.com/j2se/1.5.0/docs/api/

Joergen Ingman | TimeLife.com | Micro Site | Eventseer.net - Home | Free Website Polls | Richest Cou...n the World | Castle for Sale | Programming with C/C++ | Costa Rica, ...Estate Blogs | BASH Help - ...sh Tutorial

Try JavaMOP Online - FSL | Iterator (Java 2 Platform...

Java™ 2 Platform Standard Ed. 5.0

All Classes

Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font

IOException
IOR
IORHelper
IORHolder
IORInfo
IORInfoOperations
IORInterceptor
IORInterceptor_3_0
IORInterceptor_3_0Helper
IORInterceptor_3_0Holder
IORInterceptor_3_0Operations
IORInterceptorOperations
IRObject
IRObjectOperations
IstringHelper
ItemEvent
ItemListener
ItemSelectable
Iterable
Iterator
IvParameterSpec
JApplet
JarEntry
JarException
JarFile
JarInputStream
JarOutputStream
JarURLConnection
JButton
JCheckBox
JCheckBoxMenuItem
JColorChooser
JComboBox
JComboBox.KeySelectionManager

Overview | Package | Class | Use | Tree | Deprecated | Index | Help

PREV CLASS | NEXT CLASS | FRAMES | NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Java™ 2 Platform Standard Ed. 5.0

java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>

All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

R_1: There should be no two calls to next() without a call to hasNext() in between, on the same iterator.

public interface **Iterator<E>**

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
Collection, ListIterator, Enumeration

Method Summary	
boolean	hasNext() Returns true if the iteration has more elements.
E	next() Returns the next element in the iteration.
void	remove() Removes from the underlying collection the last element returned by the iterator (optional operation).

Method Detail

hasNext

our hasNext example again

```
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);
  }
}
```

should have been:
if(it2.hasNext())

⇓

unguarded call:
it2.next()

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()");
}

@*/
```

partial, matching against suffix trace

validation (not violation)

PTLTL Property in JavaMOP

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

```
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
}

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

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:

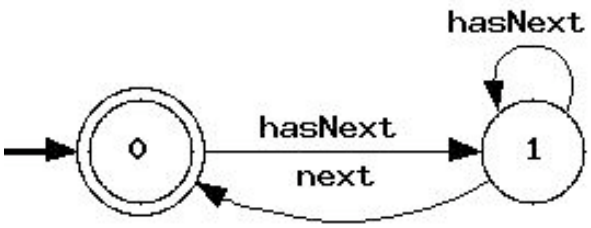
validation
=
not violation

partial matching does not make sense in PTLTL.

less attractive but simularity to regular exp can still be noticed.

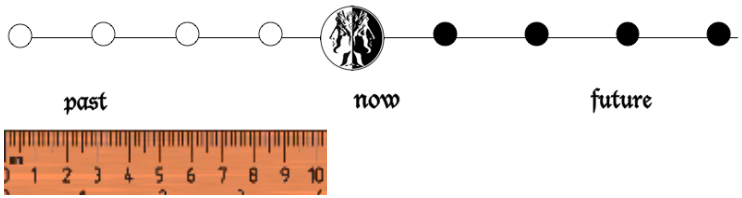
recall this other regular expression

(hasNext hasNext* 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

```
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()
```

PTLTL Property in JavaMOP - attempt 2

```
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()");
}
@*/
```

correct,
will not allow:

```
it.hasNext();
it.next();
it.next()
```

properties of Java library APIs

Enumeration (Java 2 Platform SE 5.0)

http://java.sun.com/j2se/1.5.0/docs/api/

Java™ 2 Platform Standard Ed. 5.0

All Classes

Packages

java.applet

Entity
EntityReference
EntityResolver
EntityResolver2
Enum
EnumConstantNotPrese
EnumControl
EnumControl.Type
Enumeration
EnumMap
EnumSet
EnumSyntax
Environment
EOFException
Error
ErrorHandler
ErrorListener
ErrorManager
EtchedBorder
Event

Overview Package Class Use Tree Deprecated Index Help

Java™ 2 Platform Standard Ed. 5.0

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.util

Interface Enumeration<E>

All Known Subinterfaces:

NamingEnumeration<T>

All Known Implementing Classes:

StringTokenizer

R_2: An enumeration should not be propagated after the underlying vector has been changed.

Method Summary

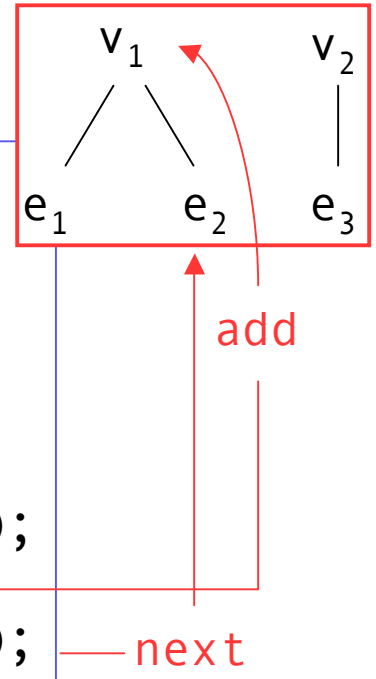
boolean	hasMoreElements() Tests if this enumeration contains more elements.
E	nextElement() Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

Find: 97 Next Previous Highlight all Match case

Downloads Clear

many vectors and enumerators

```
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());
```



events:

```
create(v,e)
next(e)
updatesource(v)
```

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*(..)))
                     \/ end(call(* v.clear()))
                     \/ end(call(* v.insertElementAt(..))) &&
                     \/ end(call(* v.remove*(..)))
                     \/ end(call(* v.retainAll(..)))
                     \/ 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

Enumeration (Java 2 Platform SE 5.0)

http://java.sun.com/j2se/1.5.0/docs/api/

Overview Package Class Use Tree Deprecated Index Help

Java™ 2 Platform Standard Ed. 5.0

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Java™ 2 Platform Standard Ed. 5.0

All Classes

Packages

java.applet

EntityReference
EntityResolver
EntityResolver2
Enum
EnumConstantNotPrese
EnumControl
EnumControl.Type
Enumeration
EnumMap
EnumSet
EnumSyntax
Environment
EOFException
Error
ErrorHandler
ErrorListener
ErrorManager
EtchedBorder
Event

java.util

Interface Enumeration<E>

All Known Subinterfaces:

NamingEnumeration<T>

All Known Implementing Classes:

StringTokenizer

R_2: An enumeration should not be propagated after the underlying vector has been changed.

Method Summary

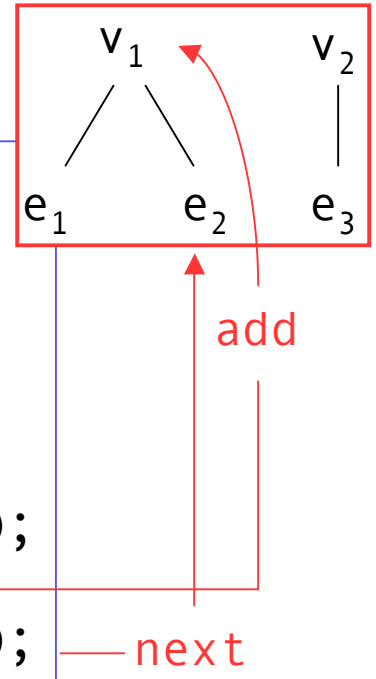
boolean	hasMoreElements() Tests if this enumeration contains more elements.
E	nextElement() Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

Find: 97 Next Previous Highlight all Match case

Downloads

many vectors and enumerators

```
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());
```



events:

```
create(v,e)
next(e)
updatesource(v)
```

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!");
}
@*/
```

FTLTL specification

```
/*@
centralized
scope = global
logic = FTLTL

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.clear()))  \/
            end(call(* v.insertElementAt(..)))  \/
            end(call(* v.remove*(..))) \/
            end(call(* v.retainAll(..))) \/
            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