Monitoring-Oriented Programming

Feng Chen, Marcelo D'Amorim, and Grigore Roșu

Presenter: Marcelo B. D'Amorim

Programming Languages Seminar

Motivation: Reliable Software

- Problems
  - theorem proving: not fully mechanizable
  - model checking: poor scalability
  - testing: incomplete, but scalable
- Challenges
  - Critical applications (aircraft, hospital, etc.)
  - Larger programs
- MOP approach
  - Observe (in)correctness of **implementation** wrt specification

Runtime monitoring!

Monitoring in Engineering

- Most engineering disciplines take monitoring as basic design principle:
  - Fire alarms in buildings
  - Fuses in electronics
  - Watchdogs in hardware

- Why not doing the same for software?
  - Monitoring-oriented programming

What is MOP…

Programming paradigm in which specification and program are defined together; and specification is validated at runtime.

MOP vs. Runtime Assertion

- Existing runtime assertion techniques
  - Design By Contract (DBC),
  - Runtime Verification (MaC, PaX, etc.)

- Why to go further?
  - No fixed specification language
    - We believe there is no *"silver bullet"* logic for all domains
  - Easier to control the program
    - Warning or exception is helpful for debugging, but may not be enough for critical applications
  - It will become easier to define new annotation languages

How MOP works…

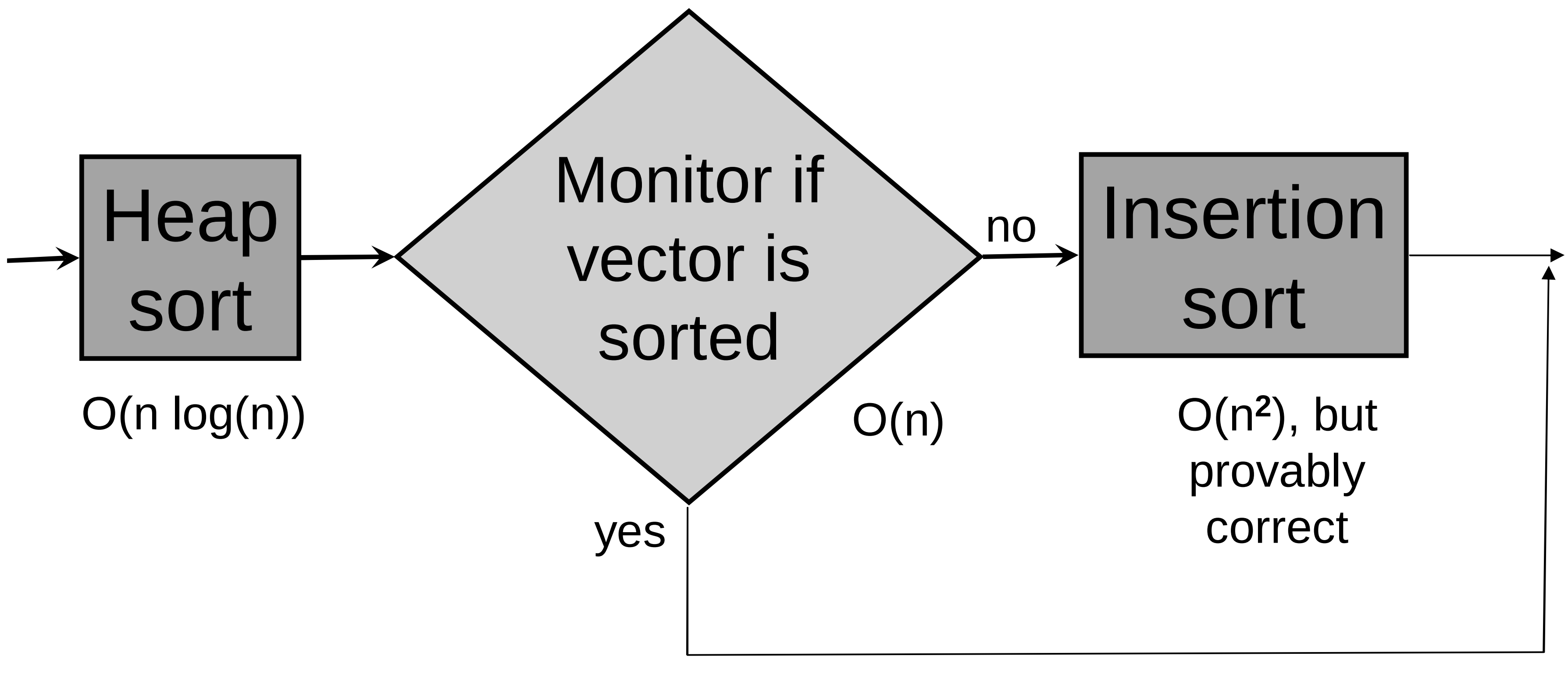
1. Programmer specifies requirements via annotations in a fixed format. These include constraints, violation and validation handlers

2. Annotations are compiled and the source code instrumented accordingly

3. Handlers may report errors or take actions during execution

Example (Simplex design)

- How to get an efficient and provably correct sorting algorithm?



MOP Ideas

- Do not modify host languages
  - Easily evolve with languages and compilers
- Language-independent
- Not limited to one logic
  - Allow users to develop and add their favorite logics (logic plug-ins)
- User-defined actions
  - May be taken at violation and/or validation of requirements

Example: An Inline LTL monitor

The following monitor can appear in a Java program

```
...
/* ftltl //MoP: after green comes yellow
   Predicate red = tlc.state.getColor() == 1;
   Predicate green = tlc.state.getColor() == 2;
   Predicate yellow = tlc.state.getColor() == 3;

   Formula : []( green -> (! red U yellow));
   Code : … any Java code
*/
...
```

Consumes resources of the host program

It is compiled into actual Java code (inline monitor)

```
...
switch(bttFsmState) {

     case 1 : bttFsmState = (tlc.state.getColor()==3) ? 1 :
                            (tlc.state.getColor()==2) ?
                            (tlc.state.getColor()==1) ? 0 : 2 : 1;
                   break;
     case 2 : bttFsmState = (tlc.state.getColor()==3) ? 1 :
                            (tlc.state.getColor()==1) ? 0 : 2;
              break;
}
   if(bttFsmState == 0) { … the "recovery" code …}
...
```

Annotation Syntax

Annotation schema

<LOGIC NAME> [{ATTRIBUTES}]

The main body of the specification

Violation Handler:

Actions when the specification is violated

Validation Handler:

Actions when the specification is validated

Annotation example

```
/*@ ptLTL {class !}

Event backup : backup() ;
Event logout : logout() ;
Predicate active : ActiveUserNum > 0 ;

Formula :
// Only after all users are logged out,
// the backup job will start.
[*] (start(backup)->[logout, active)s)

Violation Handler:
throw new RuntimeException(
               "Backup start failed!");

@*/
```

Annotation Semantics

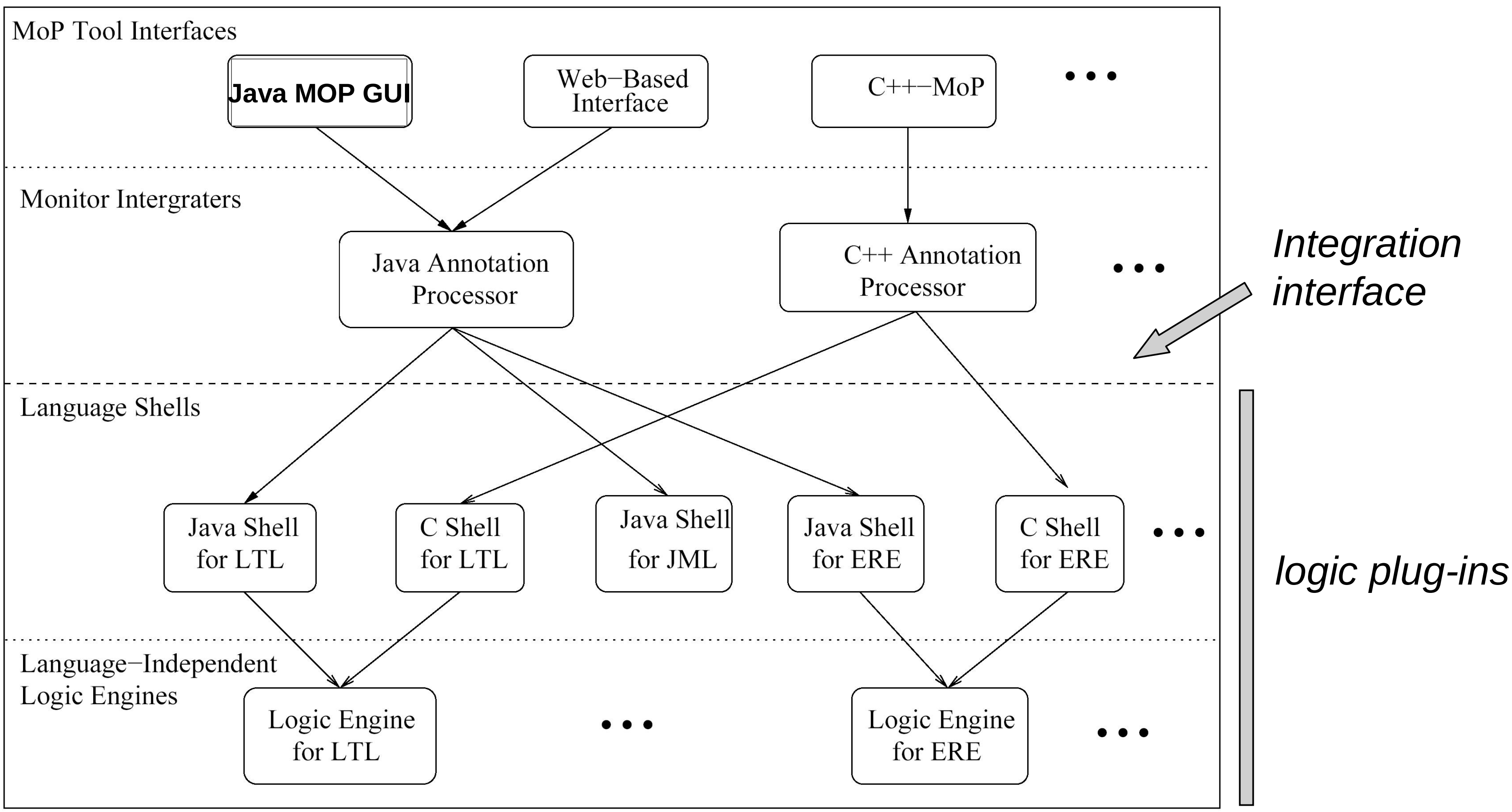
- Implementation attributes:
  - Inline, outline, online, offline, !
- Monitoring points attributes:
  - Class
  - Method (pre and post)
  - Block (pre and post)
  - Checkpoint

Inline vs. Outline Monitoring

- Outline monitoring
  - The monitor runs as a distinct process receiving events
  - Allows a centralized monitoring model
    - Can detect race conditions and deadlocks
  - Communication overhead

- Inline monitoring
  - Uses the resource of the monitored program
    - The monitoring code is inserted into the monitored program (not necessarily in the same position or thread)
  - Reduced communication

MOP layered architecture (dataflow)



Integration Layer

- Accepts the raw annotation as declared in the code
- Extracts just the formula and requests the logic plug-in to translate it to the source language
- The result is received in a standard format (*see next slide*)
- Instrumentation is carried out based on:
  - Formatted output from the logic plug-in
  - Attributes, predicates, events, and handlers from the annotation

Integration Interface

- Accepts a text formula as input
- Result is provided in the following format:

```
Declaration:
   …
Initialization:
   …
Monitoring Body:
   …
Success Condition:
   …
Failure Condition:
   …
```

Instrumentation

- Set up *monitoring points*:
  - Method calls, blocks, and checking points
  - Requires an aspect-oriented (like) language
- Merge monitor and source-code
  - Synthesize the monitoring server when monitoring outline

Logic plugin

- A logic plug-in = logic engine + language shell
- Logic engines generate intermediate code from formulae
- Language shells transform intermediate code to a target language

Code Generation for FTLTL

Input:

```
[] (green -> (! red) U yellow)
```

Output:

```
Declaration.
    int state;
Initialization.
    state = 1;
Monitoring Body.
    switch (state) {
      case 1 : state = yellow ? 1 : green ? (red ? -1:2):1;
      case 2 : state = yellow ? 1 : red ? -1 : 2;
    }
Failure Condition.
    state == -1
```

Code Generation for PTLTL

Input:

```
start(P) -> [Q, end(R \/ S))
```

Output:

```
Declaration.
    boolean now[3], pre[3]
Initialization.
    now[3] = R || S;
    now[2] = Q;
    now[1] = P;
Monitoring Body.
    now[3] = R || S;
    now[2] = (pre[2] || Q) && (now[3] || (! pre[3]));
    now[1] = P;
Failure Condition.
    now[1] && (! pre[1]) && (! now[2])
```

Code Generation for ERE

Input:

```
~((~ empty)(green red)(~ empty))
```

Output:

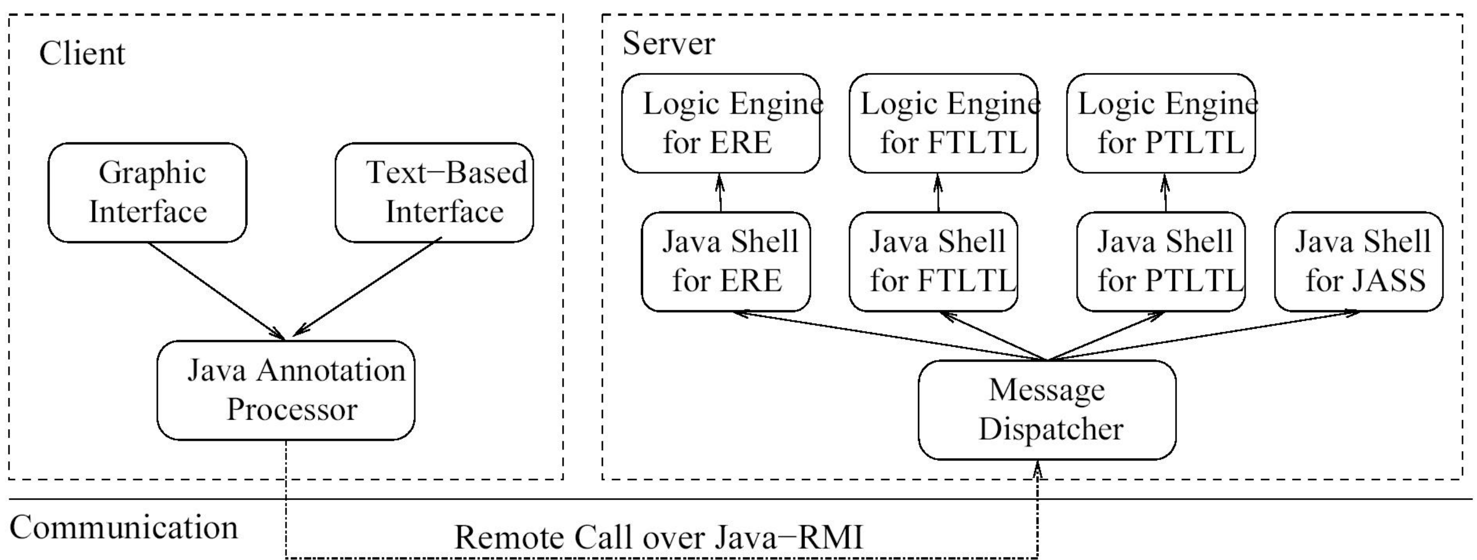
```
Declaration.
    int state;
Initialization.
    state = 0;
Monitoring Body.
    switch (state){
        case 0 : state = (yellow || red) ? 0 : green? 1:-1;
        case 1 : state = green ? 1 : yellow ? 0 : -1;
    }
Failure Condition.
    state == -1
```

Optimization

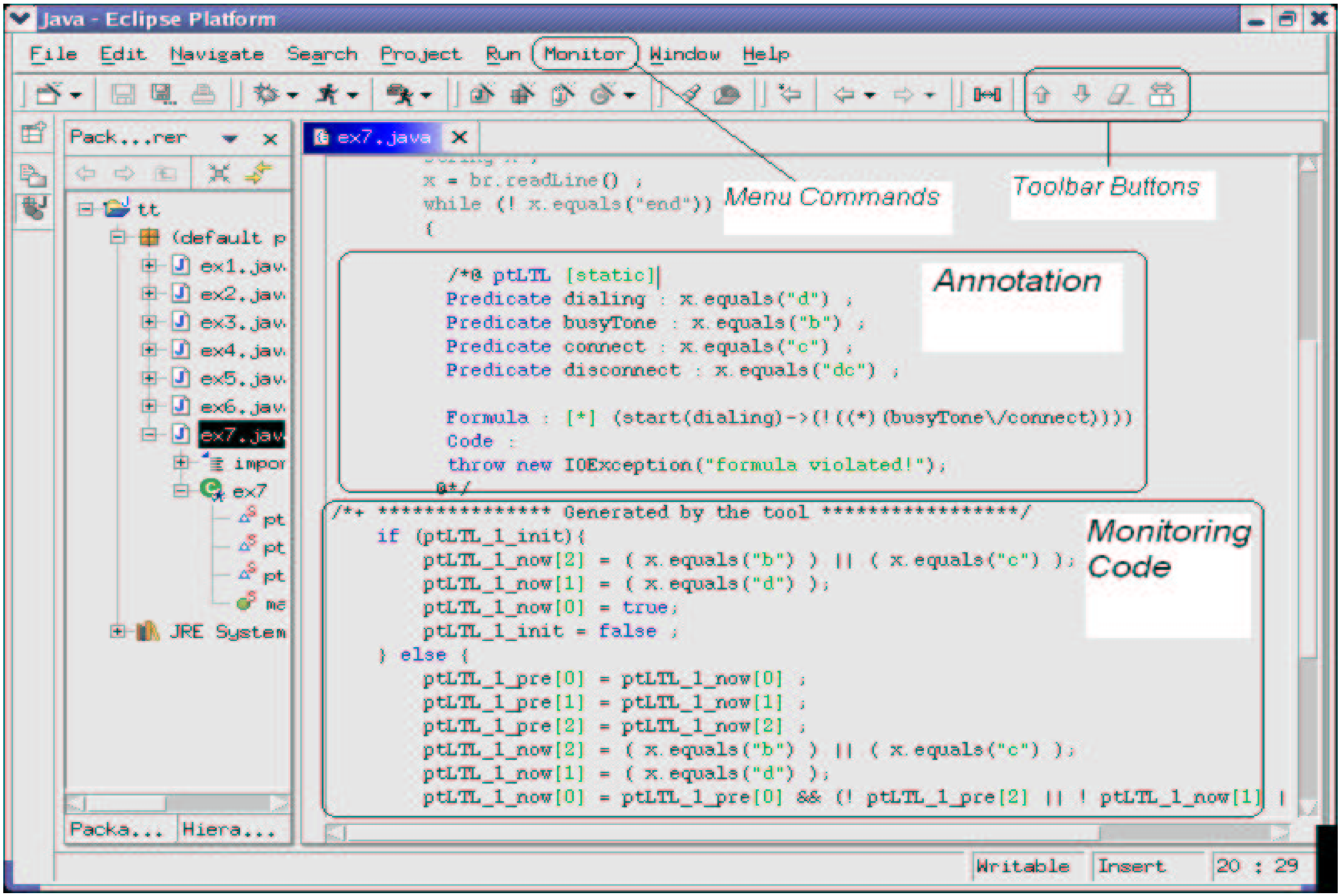
- Optimize monitoring code
  - Generate minimal state machines
- Reduce the number of monitors
  - Use static analysis to eliminate unnecessary monitoring points
- Reduce communication cost for outline monitoring
  - Decide if predicates are computed locally or remotely

Java-MoP

- Support online, inline, synchronous monitoring
- Support Jass, PTLTL, FTLTL, and ERE
- Distributed system based on Java-RMI



Snapshot for Java-MoP



Some techniques for Logic Engines

- Derivatives
  - Define new formulae denoting the requirements imposed by the first after an event is consumed. More formally,

    $\mathcal{L}(\Phi\{a\}) = \{w \mid aw \ \varepsilon \ \mathcal{L}(\Phi)\}$
  - Can be used to evolve the formula with the events, checking for validation and invalidation
  - Provides a recipe for outline monitors

Some techniques for Logic Engines

- Derivatives + Behavioral Equivalence
  - Behavioral Equivalence can be used to check if two regular expression are equivalent
    - Use Coinduction as a proof technique and accumulate equivalence predicates along the proof which is concerned with finding a closure (circularity) in which all proof steps are contained

Derivatives + Behavioral Equivalence (contd.)

- Allow one to generate a minimal DFA for ERE
  - Generate derivatives based on the alphabet. That is, for the initial expression R and alphabet {a,b}, check if R{a} and R{b} are equivalent to some r.e. visited. If it is, acumulate equivalences and update the transition relation (Don't create a new state). If not, create a new state and update the transition relation. Proceed in a depth-first-search.
- The same technique can be used to build automaton to validate LTL formulae

MOP Summary

- Formal specifications are inserted as code annotations
- Monitoring code is automatically generated and integrated
- User-defined action can be triggered by the monitors
- Three viewpoints of MOP
  1. Merging specification and implementation
  2. Extending languages with logic-based statements
  3. A light-weighted formal method

Future Work

- Define an AOP-based framework that language processors (2nd layer) can use
- Incorporate other specification languages
  - MTL, Eagle, CARET

Visit
http://fsl.cs.uiuc.edu/mop/
for experimenting the plug-ins for ERE, LTL and Jass

Questions…