Survey on Trace Analyzers

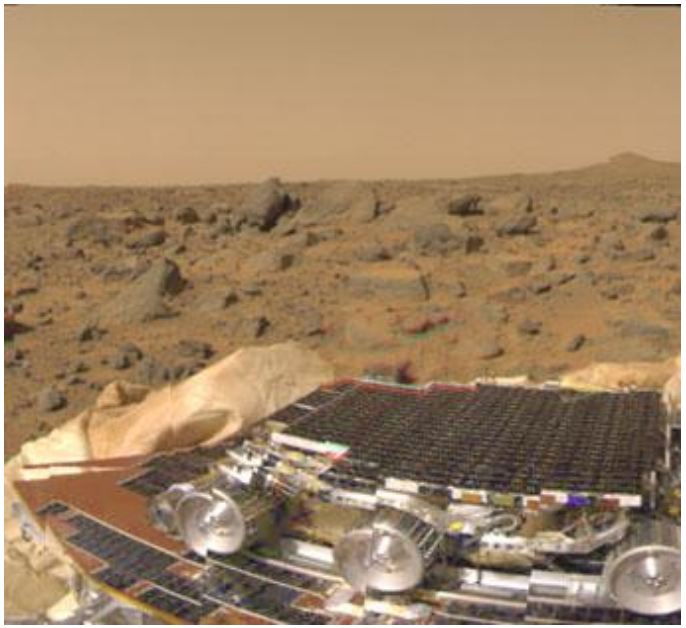
Hong, Shin

Contents

- Introduction
- JavaPathExplorer
- TemporalROVER
- Further study

Introduction (1/4)

- The importance of Software is getting increased.
  - ➔ Quality assurance of software is very important.
- Software is becoming more complex.
  - ➔ To assure quality of software is very hard.



Mars Pathfinder

landing on July 4, 1997

about 135,000 lines of code

costs $150,000,000

Introduction (2/4)

Approaches

- Model checking
  - Fail to apply on large programs.
  - Abstraction is a labor-some process.

- Traditional Testing
  - Do not allow formal specification.

Introduction (3/4)

Approach

Principle I. Do not involve heavy labor on developers.

Principle II. Use formal specification.

Principle III. Test or verify an executable program.

➔ Analyze a run-time trace of program with respect to formal specification.

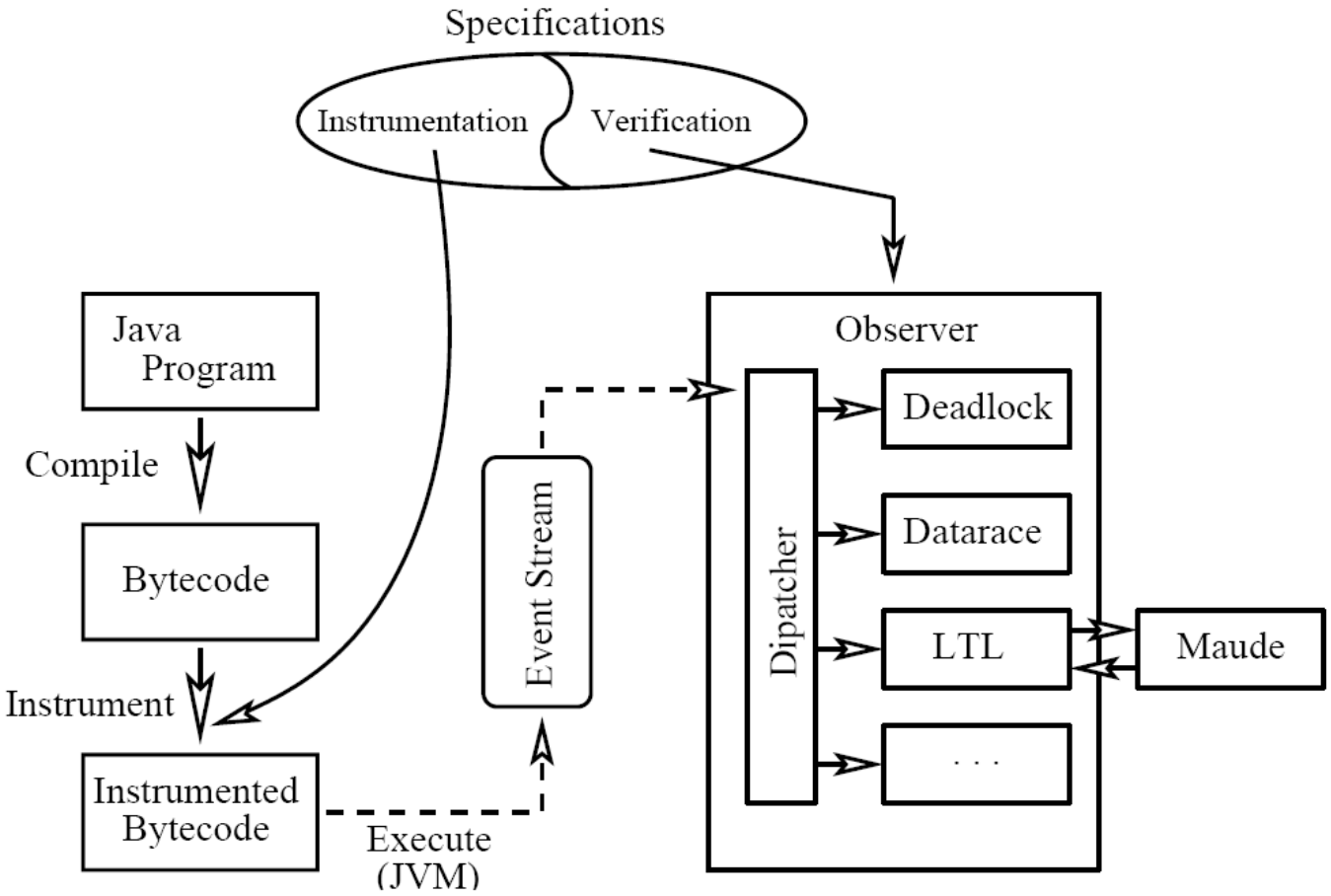
Introduction (4/4)

- Trace Analyzer
  - Use Formal specification
  - Extract interesting information while program is executing.
  - Verify traces of the program

Java Path Explorer (JPAX) 1/13

- A run time verification tool from NASA.
- Combine testing and formal method.
- Test execution traces against high level specification.
- Successfully applied on actual rover control program.

JPaX 2/13


Specifications
Instrumentation
Verification
Java Program
Compile
Bytecode
Instrument
Instrumented Bytecode
Execute (JVM)
Event Stream
Observer
Dipatcher
Deadlock
Datarace
LTL
. . .
Maude

JPaX 3/13

Work-flow

1. User writes formal specification.
2. Instruments probe on target program.
3. Extracting events from an executing programs.
4. Analyzing the events via remote observer.
5. Observer performs

   (1) logic-based monitoring

   (2) error-pattern analysis.

JPaX 4/13

- Logic based Monitoring
  - Maude
    - Meta-programming language
  - LTL
    - Safety property
    - End-of-Trace
    - Liveness property : $[](p \rightarrow <>q)$?
    - Observer need to take a decision regarding the validity of checked property.

JPaX 5/13

- Error-pattern analysis
  - Detect error potential.
  - Suggest problems from single execution trace.
  - JPaX supports Data race analysis and Deadlock analysis
  - Flexible manner.

JPaX 6/13

- Data Race analysis
  - Do not permit dirty read of shared variables in multiple readers and writers.

```
class Value {
    private int x = 1 ;
    public synchronized void add(Value v) { x = x + v.get() } ;
    public int get() { return x ; }
}

class Task extends Thread {
    Value v1 ; Value v2 ;
    public Task(Value v1, Value v2) {this.v1=v1;this.v2=v2;this.start()}
    public void run(){v1.add(v2)} ;
}

class Main {
    public static void main(String [] args) {
         Value d1 = new Value() ; Value d2 = new Value() ;
         new Task(d1, d2) ; new Task(d2, d1) ;
    }
}
```

JPAX 7/13

- Task 1

14 : start()

17 : **d1**.lock.acquire()

17 : **d1**. add(**d2**)

4 : **d1**.x = **d1**.x + **d2**.get()

4 : R_1 = **d2**.get() = 1

4 : **d1**.x = 1 + R_1 = 2

- Task 2

14 : start()

17 : **d2**.lock.acquire()

17 : **d2**. add(d1)

4 : **d2**.x = **d2**.x + **d1**.get()

4 : R2 = **d1**.get() = 1

4 : **d2**.x = 1 + R_2 = 2 **Incorrect d2.x = 3**

JPAX 8/13

- Thread-map
  - Keep set of locks that each thread holds
  - Thread-map : Thread T→ $\mathbb{P}$(Locks)
- Variable map
  - holds a set of locks for each shared variable that every thread which uses the shared variable commonly hold.
  - Variable-map : Variable V → $\mathbb{P}$(Locks)
  - For every visit of V of T
    - Variable-map[V] = Variable-map[V] ∩ Thread-map[T]
  - If variable map is empty, data race might be occurred.

JPAX 9/13

- Task 1 Task 2

14 : start()

17 : **d1**.lock.acquire() Thread-map[Task1] = {d1.lock}

17 : **d1**. add(**d2**)

14 : start()

17 : **d2**.lock.acquire() Thread-map[Task2] = {d2.lock}

17 : **d2**. add(d1)

4 : **d1**.x = **d1**.x + **d2**.get() Variable-map[d1] = {d1.lock}

4 : R_1 = **d2**.get() = 1 Variable-map[d2] = {d1.lock}

4 : **d1**.x = 1 + R_1 = 2

4 : **d2**.x = **d2**.x + **d1**.get() Variable-map[d1] = {}

4 : R2 = **d1**.get() = 2 Variable-map[d1] = {}

4 : **d2**.x = 1 + R_2 = 3

JPaX 10/13

- Deadlock analysis

```
class Value {
    private int x = 1 ;
    public synchronized void add(Value v) { x = x + v.get() } ;
    public synchronized int get() { return x ; }
}

class Task extends Thread {
    Value v1 ; Value v2 ;
    public Task(Value v1, Value v2) {this.v1=v1;this.v2=v2;this.start()}
    public void run(){v1.add(v2)} ;
}

class Main {
    public static void main(String [] args) {
        Value d1 = new Value() ; Value d2 = new Value() ;
        new Task(d1, d2) ; new Task(d2, d1) ;
    }
}
```

JPaX 11/13

- Task 1
  - 14 : start()
  - 17 : **d1**.lock.acquire()
  - 17 : **d1**. add(**d2**)
  - 4 : **d1**.x = **d1**.x + **d2**.get()
  - 4 : **d2**.lock.acquire()
- Task 2
  - 14 : start()
  - 17 : **d2**.lock.acquire()
  - 17 : **d2**. add(d1)
  - 4 : **d2**.x = **d2**.x + **d1**.get()
  - 4 : **d1**.lock.acquire()

Deadlock occurred!!

JPaX 12/13

- Thread-map
- Lock-graph
  - Nodes : all the locks
  - Edges : held lock → newly holding lock
  - When a thead T tries to acquire a lock L

```
For each lock H in Thead-map[T] {
   make <H,L> as an edge of the graph ; }
Thread-map[T] = Thread-map[T] ∪ {L} ;
```

- If the graph has a cycle, deadlock could be occurred.

JPAX 13/13

- Task 1 Task 2

14 : start()
17 : **d1**.lock.acquire() Thread-map[Task1] = {d1.lock}
17 : **d1**. add(**d2**)
4 : **d1**.x = **d1**.x + **d2**.get()
4 : **d2**.lock.acquire() Thread-map[Task1] = {d1.lock, d2.lock} d1.lock→ d2.lock
4 : R_1 = **d2**.get() = 1
4 : **d1**.x = 1 + R_1 = 2

14 : start()
17 : **d2**.lock.acquire() Thread-map[Task2] = {d2.lock}
17 : **d2**. add(d1)
4 : **d2**.x = **d2**.x + **d1**.get()
4 : **d1**.lock.acquire() Thread-map[Task1] = {d1.lock, d2.lock}
d2.lock→ d1.lock Cycle!!
4 : R2 = **d1**.get() = 2
4 : **d2**.x = 1 + R_2 = 3

TemporalROVER 1/7

- A commercial specification based verification tool.
- Use Linear-Time Temporal Logic and Metric Temporal Logic for formal specification.
- Target mainly on complex protocol and reactive systems where its behavior is time dependent.

TemporalROVER 2/7

- Formal Methods
  - LTL
    - Propositional logic + Time operators(such as X, U, [], <>)
  - MTL
    - Extends LTL by supporting the specification of relative time and real time constraints.
      - $\text{Always}_{<10}$({readySignal==1} implies {askSignal==0})
      - $\text{Eventually}_{\text{timer2} >= 20}$ ({ackSignal == 0})
    - TemporalROVER statement 'TRClock' maps virtual clocks to system call or system clock.

TemporalROVER 3/7

- Proprietary Operators
  - Repeated Until
    - ({signal == 0}) RepeatedUntil$_{<=5}$ ({ack == 1})
  - Reapeated
    - Reapeated$_{[4,7]}$ ({signal == 0})

TemporalROVER 4/7

- Assertion
  - TRAssertion {<formula>} => {<Four optional actions>}
  - <Four optional actions>
    1. Action to perform every cycle where formula succeeds.
    2. Action to perform every cycle where formula fails.
    3. Action to perform every cycle.
    4. Action to perform every cycle where the formula succeeds or fails in a final way.

- Write on source code as comment

TemporalROVER 5/7

```
void myScheduler() {
   while(1){ …
        trafficLightController() ; …}}

void trafficLightController() {
   /*TRBegin
   TRAssert {Always{Color_main==Green} implies {CameraOn==0}}
   =>
   {    printf("Assertion1: so far : Success\n") ;
        printf("Assertion1: so far : Fail\n") ;
        TRProcessLastResult("Assertion1\n") ;
        printf("Assertion1: Done! Your last result is Final!\n") ;
   }
   TREnd*/
}
```

TemporalROVER 6/7

- Eventually {x>0}
  - x = 0 until cycle 1000,
  - The program will not continue executing
  - The program will continue executing and possibly x>0 sometime thereafter.

TemporalROVER 7/7

- Embedded System Code Generator
  - Limited Memory
  - No storage
  - Real-time constraints

- Do not bother target program.
- Via serial port, RPC, target program only passes the result of basic computation of proposition to remote observer.

Conclusion

- Formal method that can specify target program's properties well and easily is key part of trace analyzer.

Further study

- Survey more trace analyzers.

- POTA : Partial Order Trace Analyzer
  - Computational Tree Logic
- JMPAX : Java Multithreaded Path Explorer

References

[1] Verisim:Formal Analysis of Network Simulations, IEEE TSE2002.

[2] Formal Verification of Simulation Traces Using Computation Slicing,

Sen&Garg, IEEE Transaction on Computing April 2007.

[3] Java PathExplorer-A Runtime Verification Tool, Havelund et al, 2001 RV.

[4] The Temporal Rover and ATG Rover, Drusinsky, 2000 Int'l SPIN workshop.

[5] Eraser: A Dynamic Data Race Detector for Multithreaded Programs,

Savage&Anderson, IEEE ToCS 97

Discussion