A Formal Connection between Security Properties and JML Annotations

Marieke Huisman
INRIA Sophia Antipolis
Everest Project

joint work with
Alejandro Tamalet
Radboud University, Nijmegen, Netherlands

Goal of work

- Security of applications crucial for trusted devices
- Possible solution: enforce property at run–time
  - Monitoring executions
  - But how to recover from security violation?
- Ultimate goal: static verification of security properties
  - Properties need to be expressed in suitable format

This work

- Focus of work: Java (like) sequential programs
- Encode security property as JML annotations
- Use of JML provides means for
  - Run–time checking (jmlc)
  - Static verification (ESC/Java, Mobius tool set)
- Algorithm & formal correctness proof
- Restrictions on properties: only safety properties

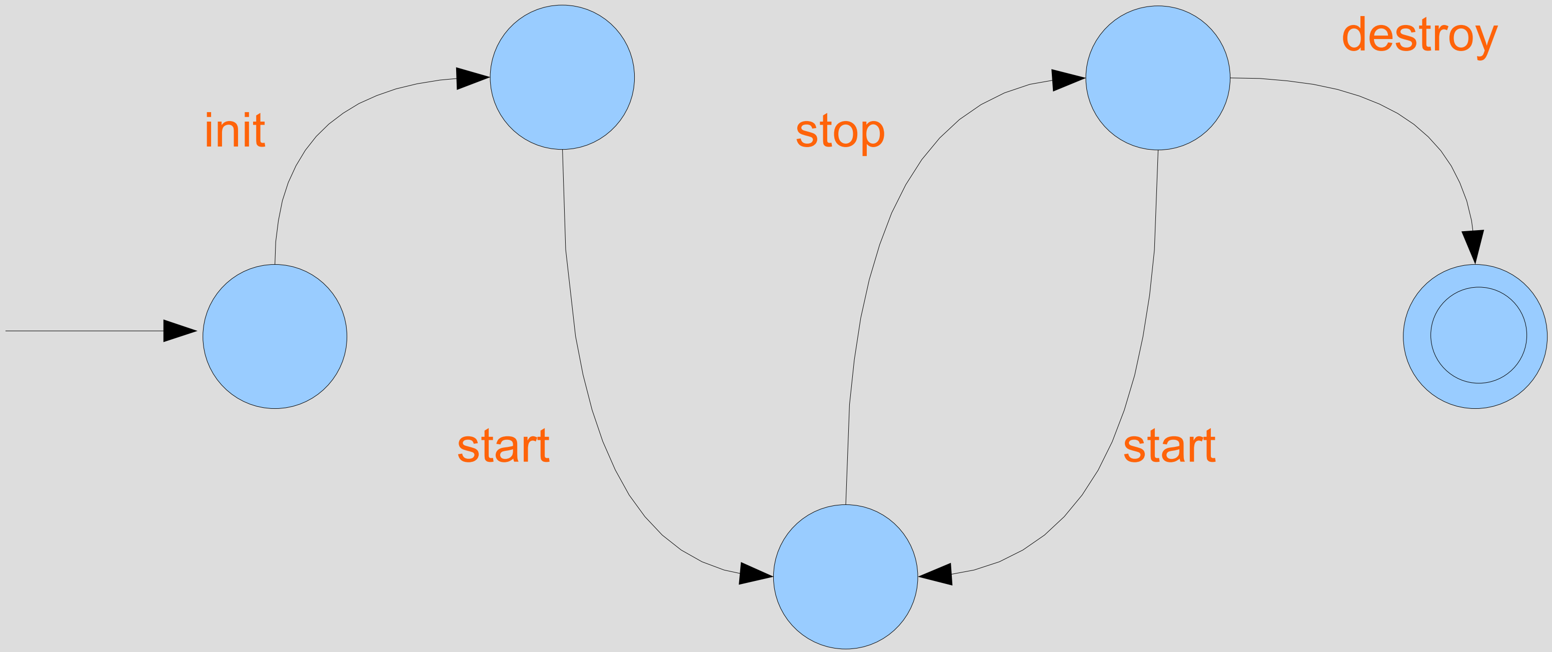
Outline

- Specifying monitors
- Translation of monitor into JML annotations
- Formalisation and correctness proof
- An unexpected subtlety with try-catch-finally
- Conclusions, related and future work

Security properties as automata

- High level view of properties
- Intuitive specifications
- Automaton specifies property of monitored class

Example: applet protocol expressed as automaton



init; (start; stop)+; destroy

Example due to
Cheon and Permendla

Applet protocol specified in JML

```
package java.applet

public class Applet {
  /*@ public static final ghost int
    @   PRISTINE = 1,
    @   INIT = 2,
    @   START = 3,
    @   STOP = 4,
    @   DESTROY = 5;
    @*/

//@ public ghost int state = PRISTINE;

//@ requires state == PRISTINE;
//@ ensures state == INIT;
public void init() {
  //@ set state = INIT;
  ...
}
```

```
//@ requires state == INIT || state == STOP;
//@ ensures state == START;
public void start() {
  //@ set state = START;
  ...
}

//@ requires state == START;
//@ ensures state == STOP;
public void stop() {
  //@ set state = STOP;
  ...
}

//@ requires state == STOP;
//@ ensures state == DESTROY;
public void destroy() {
  //@ set state = DESTROY;
  ...
}

...}
```

Multi-Variable Automata (MVA)

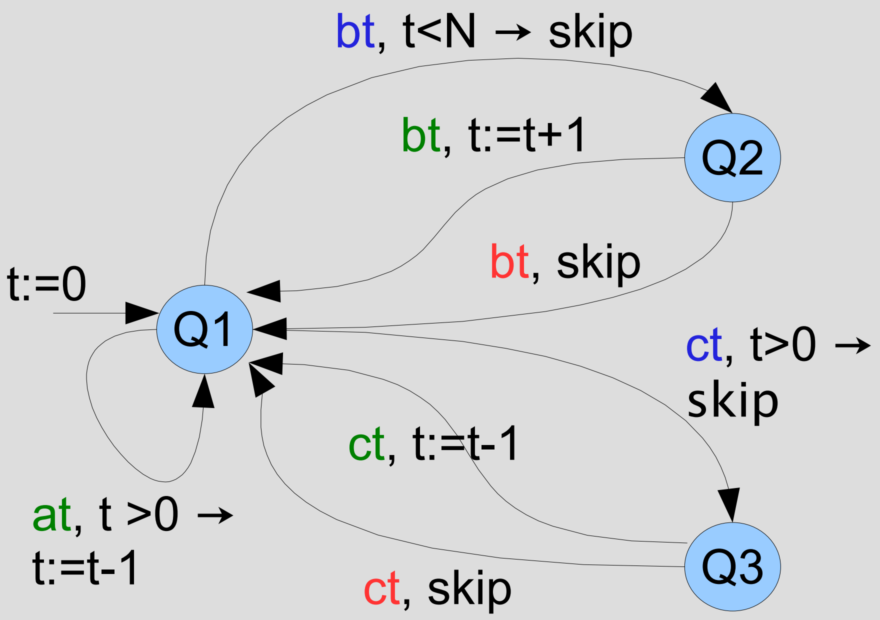
- Many interesting properties cannot be captured by regular automata
- For more expressivity: variables needed
- Inspection of program variables
- Updates of monitor-only variables

Transitions

- Transitions of MVA contain event, guard and actions
- Events can be entry or exit of methods
  Distinction between normal exit and exceptional exit
- Guards and actions may use automata variables and fields of monitored class
- Actions can only update automaton variables

Example: Embedded transactions

Property: At most N embedded transactions



bt = beginTransaction()
ct = commitTransaction()
at = abortTransaction()
entry
exit normal
exit exceptional

Automaton:
Monitored class: Transaction.java
$Q = \{Q1, Q2, Q3\}$
$\Sigma = \{bt, bt, bt, ct, ct, ct, at\}$
$vars_A = \{(t, int, 0)\}$
$vars_P = \{\}$

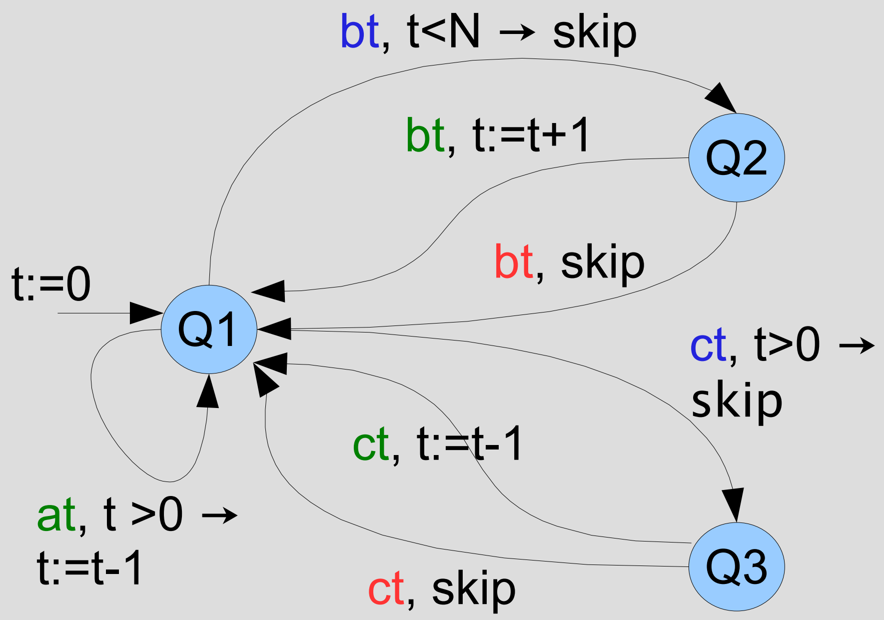
Typical example properties

- Enforce order in which methods are called: life cycle or object protocol
- Restrict the occurrence of a particular method call: m() can be called at most once
- Control-flow restrictions: method m1() can not or can only be called inside/after/before method m2()

Characteristics of MVA

- Automaton must be deterministic
- Transition relation completed by adding error state halted
- Add transitions to ensure halted is trap state
- No accepting states, i.e., no termination

Example: Completion of MVA



Abstract correctness property

P = program (possibly annotated)

A = monitoring automaton

|| = monitoring composition

≈ = equivalence relation

$$P \mid\mid A \approx \text{ann\_program}(P, A)$$

<u>Assumptions</u>:

P and A well-formed

P and A match

``P does not (implicitly or explicitly) catch JML exceptions''

Annotation generation algorithm

- Focus on correctness, rather than on efficiency of implementation
- Two step translation
  - Intermediate format, with set-statements in method specification
  - Transform method specifications into inline annotations

Code transformations

- Code transformations are needed in second step to model
  - monitoring of exceptions
  - methods with multiple returns
- Body should be enclosed in try–catch–finally block
- If code transformations are not allowed, automaton can only monitor method entry

Step 1 – 1: Add ghost variables

- New ghost variables declared to encode automaton
    - Control points (including halted): integer constants, initialised to unique value
    - Current control point (cp): integer initialised to initial control point
    - Automaton variables: type and initial value as specified for the automaton
- Note: program variables can be ignored

Step 1 – 1: Example

```
/*@ public static final ghost int
  @  HALTED = 0,
  @  Q1 = 1,
  @  Q2 = 2,
  @  Q3 = 3;
  @*/

//@ public ghost int cp = Q1;

//@ public ghost int t = 0;
```

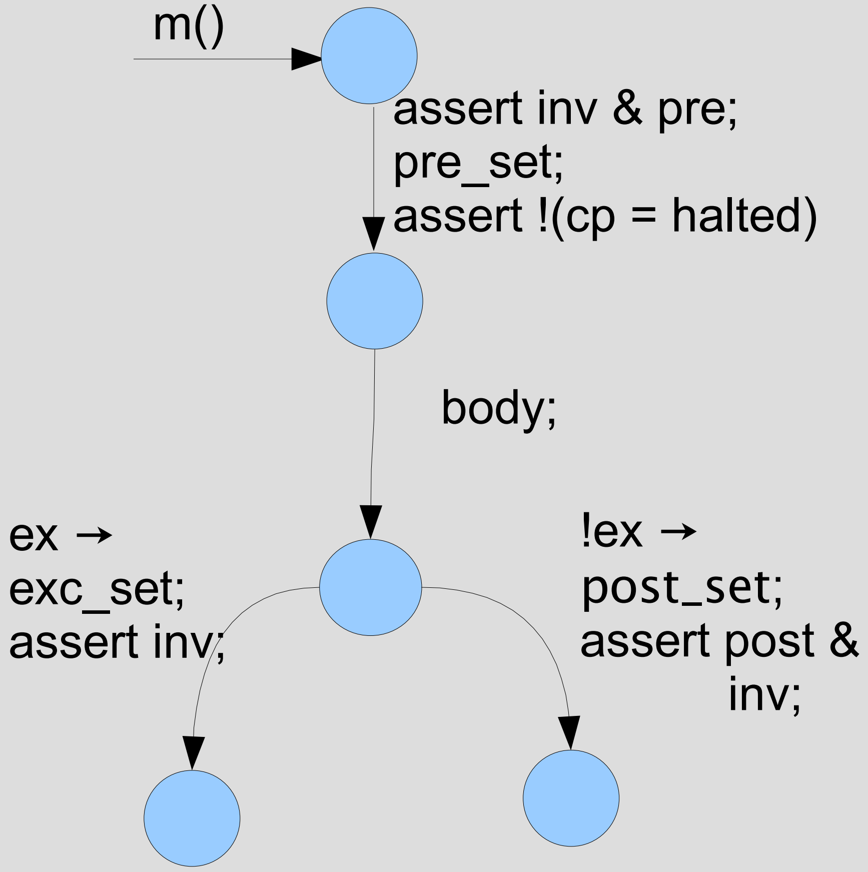
Step 1 – 2: Strengthen invariant

- Invariant is strengthened to assert that current control point has not reached the error state

```
//@ public invariant cp != halted;
```

Step 1 – 3: Annotate methods

```
//@ requires pre;
//@ ensures post;
m() {
  pre_set {
    /*@ annotations concerning
        m's entry @*/
  } body {
    m's body
  } post_set {
    /*@ annotations concerning
        m's normal exit @*/
  } exc_set {
    /*@ annotations concerning
        m's exceptional exit @*/
  }
}
```



m()

assert inv & pre;
pre_set;
assert !(cp = halted)

body;

ex →
exc_set;
assert inv;

!ex →
post_set;
assert post &
inv;

Step 1 – 4: Translate events

- Pre_set, post_set and exc_set encode actions of automaton
- Before entering body, check whether pre_set not reached trap state
- Multiple transitions can be associated to a single event – choice based on guard
- Special conditional ghost variable update construct to model this choice

Step 1 – 4: Example at

```
/*@ if (cp == Q1) {
  @   if (t > 0) {
  @     set t = t – 1;
  @     set cp = Q1;
  @   } else {
  @     set cp = HALTED;
  @ } else if (cp == Q2) {
  @   set cp = HALTED;
  @ } else if (cp == Q3) {
  @   set cp = HALTED;
  @ } else { // cp == HALTED
  @   set cp = HALTED;
  @ }
  @*/
```

```
/*@ if (cp == Q1 && t > 0) {
  @   set t = t – 1;
  @   set cp = Q1;
  @ } else {
  @   set cp = HALTED;
  @ }
  @*/
```

Step 2 – 1: Refine if – 1

- The conditional ghost variable updates are translated into a sequence of set annotations using conditional expressions

```
if (c) {
  set x := a;
  set y := b;
}
```

```
set x := c ? a : x;
set y := c ? b : y;
```

Step 2 – 1: Refine if – 2

- Auxiliary ghost variables are used to ensure that earlier updates do not affect later assignments

```
if (cp == Q1) {
  if (x >= 5) {
    set x = x–1;
    set cp = Q2;
  } if (x < 0) {
    set x = x+1;
    set cp = Q1;
  } else {
    set cp = HALTED;
  }
}
```

→

```
set contr = cp == Q1;
set guard = x >= 5;
set x =  contr && guard? x–1 : x;
set cp = contr && guard ? Q2 : cp;
set guard = !guard && x < 0;
set x = contr && guard ? x+1 : x;
set cp = contr && guard ? Q1 : y;
set guard = !guard;
set cp = contr && guard? HALTED : cp;
```

Step 2 – 2: Inline method set statements

```
m() {
  //@ ghost boolean ex;
  //@ pre_set;
  //@ assert cp != halted;
  try {
    body
  }
catch (Exception e) {
    //@ exc_set;
    //@ set ex = true;
    throw e;
  } finally {
    //@ if (!ex) { post_set; }
  }
}
```

Example: translation of the embedded transactions

```
public void beginTransaction() {
 //@ ghost boolean ex;
 //@ set cp = (cp == Q1  && t < N) ? Q2 : HALTED;
 //@ assert cp != HALTED;
 try {
   body
  } catch (Exception e) {
    //@ set cp = (cp == Q2) ? Q1 : HALTED;
    //@ set ex = true;
  } finally {
    //@ set t = (!ex && cp == Q2) ? t+1 : t;
    //@ set cp = (!ex && cp == Q2) ? Q1 : HALTED;
  }
}
```

An aside: the problem with Try-Catch-Finally

```
try{
  r := randomInt();
  decrypt(key, r);
}
finally{
   throw NullPointerException()
}
```

```
//@ requires inRange(arg);
decrypt(key, arg){
 ...
 }
```

- Run-time assertion checking will never return a JML Exception, but static checking will find this specification violation

Advantages of having a formalisation – 1

- Although the ideas are simple we found many subtleties:
  - assert at the end of the pre_set
  - formulation of new invariant
  - try–catch–finally needs special restrictions, to avoid that JMLExceptions are ignored
  - precise formulation of related states predicate: under which conditions does the program reach an exceptional state, when is correspondence maintained

Advantages of having a formalisation – 2

Makes all requirements explicit:

- no overlap between variable names of automaton and monitored class
- evaluation of expressions in guards or actions cannot have side effects or throw exceptions
- strictness of conjunction
- injective function needed to map control points to int

Related work

- FSM to annotations [Hubbers, Oostdijk, Poll]
- Temporal logic to annotations [Groslambert et al.]
- Midlet Navigation Graphs to JML, graph refinement [de Jong, Ravelo, Poll] Converting Midlet Navigation Graphs into JML
- Method call sequences as annotations [Cheon, Perumendla]
- Propagation of annotations [Pavlova et al.]

Implementations, but no formal proof

Conclusions

- Translation from monitors to annotations
- Correctness of transformation proven with help of theorem prover
- Modular semantics
- Formalisation helped to reveal unexpected problems (notably try–catch–finally)

Future work

- Formally prove correctness of second step
- Allow method parameters in monitor
- Generate preconditions and postconditions (now inline annotations generated)
- Towards static proving of security properties
  - Extend propagation algorithm of Mariela Pavlova
  - Formalise propagation algorithm in PVS
- Wider class of properties possible?
- Use for multi-threaded programs (under certain restrictions)