A Formal Connection between Security Properties and JML Annotations

Work in progress with Marieke Huisman

Alejandro Tamalet
Radboud University
Nijmegen, The Netherlands

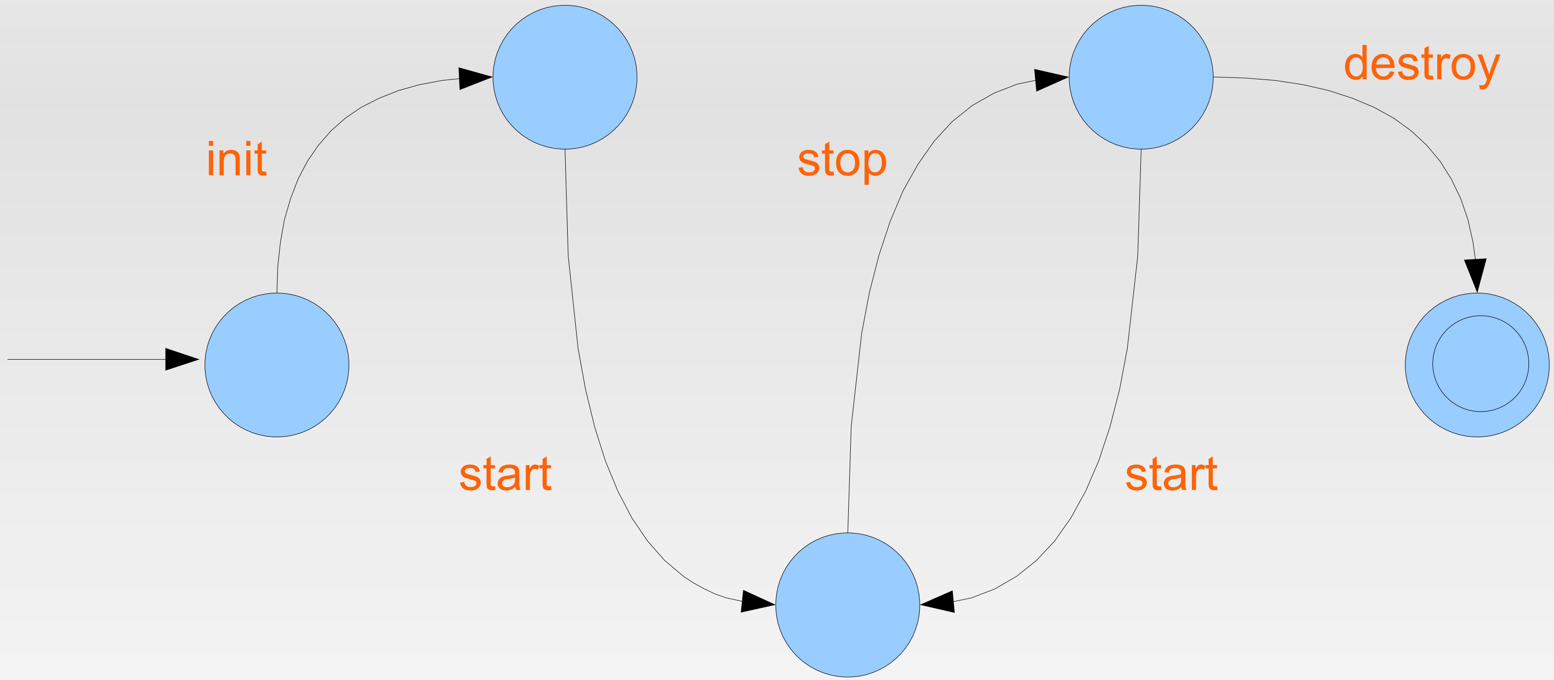
Introduction: The Goal

- Trusted devices (smart phones, PDA, smart cards) need a way to ensure the security of applications.
- We want to enforce (at runtime) a certain property. Ultimately, we would like to prove (statically) that it holds.
- We will work with Java or Java-like sequential programs.

Introduction: The Means

- One way to achieve this goal is to encode the property as JML annotations
- JML connects runtime checking (jmlc) and proving (ESC/Java2).
- This imposes restrictions on the kind of properties we can express: only safety properties (no liveness).

Example: An applet protocol as an automaton (Cheon and Perumendla)



init; (start; stop)+; destroy

Example: The applet protocol specified in JML (Cheon and Perumendla)

```
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)

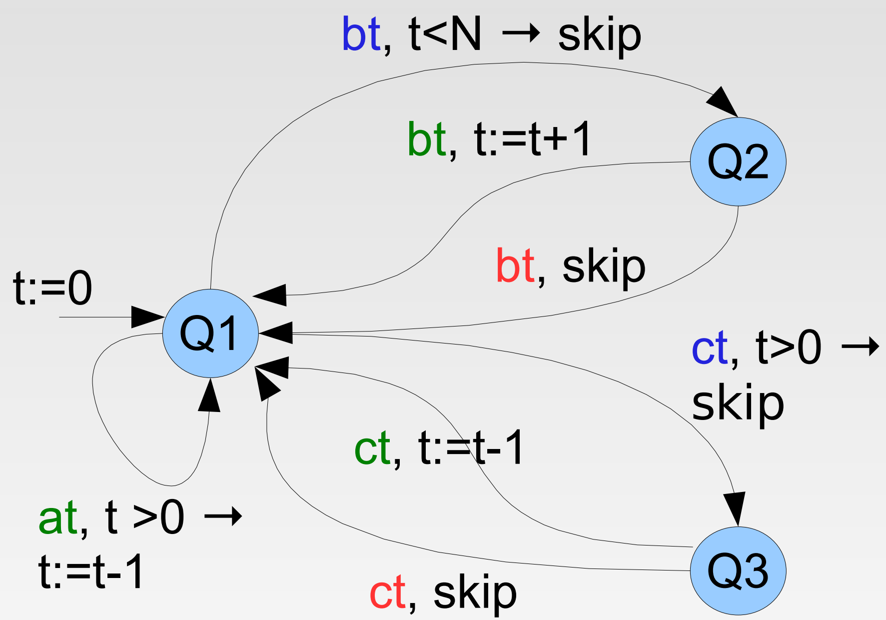
- We want to keep the high level view of these properties.
- Regular automata are not enough to express many interesting properties. We use automata with variables.
- An automaton specifies a property of a class called the monitored class.

Transitions

- Transitions of an MVA have an event, a guard and actions.
- The events can be entry to or exit of methods. We distinguish between a normal exit and an exceptional exit.
- Guards and actions may involve fields of the monitored class or parameters of the method. Actions can only update variables of the automaton.

Example: Embedded transactions

Property: At most N embedded transactions.



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

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

Other properties

- Enforce and order in which methods are called: life cycle or protocol of an object.
- Restrict the frequency of a particular method call. Example: m() can be called at most one time.
- Method m1() can not or can only be called inside method m2().

Characteristics of a MVA

- The automaton must be deterministic.
- We complete the transition function by adding an error state. We call it halted.
- Since we work with safety properties, halted is a trap state.
- We don't have accepted states.

Abstract correctness property

P = program (may already have annotations)

A = automaton describing a security property

|| = monitored by

≈ = equivalence relation

<u>Assumptions</u>: P does not throw nor catch JML exceptions

A is “*well formed*” and “*well behaved*”

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

Translation into JML... plus some code transformations

- Some code transformations are needed to treat exceptions. We have to enclose the body in a `try-catch-finally` block.
- If no code transformations are allowed we must restrict the expressiveness of the automata. We would only be able to talk about entry to methods.

ann_program: Two step translation

- For the following algorithm, we focus more in its correctness than in its actual implementation.
- For ease of verification, the translation is done in two steps. In the first step we do some abstractions and then we refine them in the second step.

Step 1 – 1: Add ghost variables

- New ghost variables are added to encode the automaton.
  - Control points (including halted): integers initialized to a unique value.
  - Current control point (cp): integer initialized to the value of the initial control point.
  - Variables of the automaton: their type and initial value are provided by the automaton.

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;
```

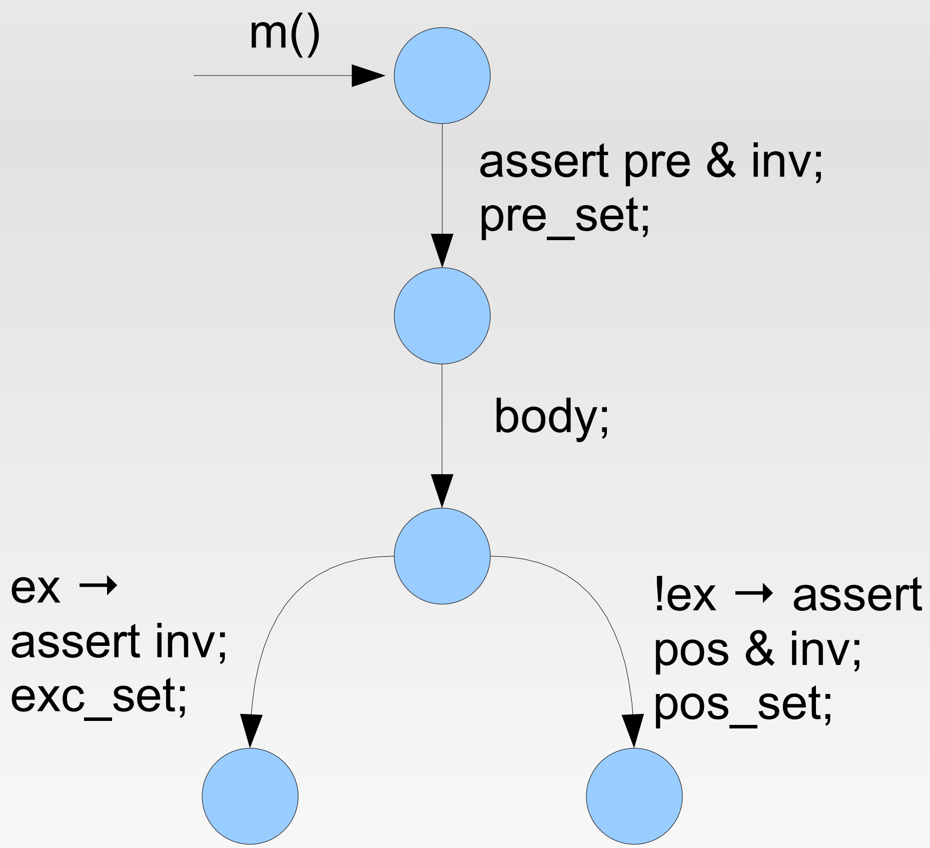
Step 1 – 2: Strengthen invariant

- The invariant is strengthened to assert that the current control point has not reached the error state.

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

Step 1 – 3: Annotate methods

```
//@ requires pre;
//@ ensures pos;
m() {
  pre_set {
    /*@ annotations regarding
        m's entry @*/
  } body {
    m's body
  } pos_set {
    /*@ annotations regarding
        m's normal exit @*/
  } exc_set {
    /*@ annotations regarding
        m's exceptional exit @*/
  }
}
```



Step 1 – 4: Translate events

- Each transition is translated independently of the type of its event (entry, exit normal or exit exceptional).
- We assume the existence of an `if` statement that works with ghost variables in the condition and in the branches.

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 `if` for ghost variables are translated into a sequence of `set` statements using conditional statements.

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

→

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

Step 2 – 1: Refine if - 2

- Two auxiliary ghost variables are used to ensure the independence of the branches.

```
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 b1 = cp == Q1;
set b2 = b1 && x >= 5;
set x = b2 ? x-1 : x;
set cp = b2 ? Q2 : cp;
set b2 = b1 && !b2 && x < 0;
set x = b2 ? x+1 : x;
set cp = b2 ? Q1 : y;
set b2 = b1 && !b2;
set cp = b2? HALTED : cp;
```

Step 2 – 2: Refine pre_set et al.

```
m() {
  //@ ghost boolean ex;
  try {
    //@ pre_set;
    //@ assert cp != halted;
    body
  }
```

```
catch (Exception e) {
    //@ exc_set;
    //@ set ex = true;
    throw e;
  } finally {
    //@ if (!ex) { pos_exc; }
  }
}
```

Example: translation of the embedded transactions

```
public void beginTransaction() {
  //@ ghost boolean ex;
  try {
    //@ set cp = (cp == Q1 && t < N) ? Q2 : HALTED;
    //@ assert cp != HALTED;
    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;
  }
}
```

Formalization

- Everything must be defined:
  - Automatons and their operational semantics.
  - (A subset of) Java programs with annotations and their operational semantics (big step, based on Von Oheimb's formalization).
  - A semantics for monitored programs.
  - A bisimulation relation.

PVS

- Provides an expressive specification language an interactive proof checker and other tools for managing and analysing specifications.
- Its logic is an extension of higher order logic with support for predicate subtyping and dependent types.
- Does not provide polymorphic types but theories are parametrizable.

A subset of Java-like programs - 1

- We formalized the syntax and semantics of a subset of Java relevant for our problem.
  - Types: int, boolean, void, references.
  - Exceptions: Throwable, NullPointer, JMLExc
  - Expressions: method calls, assignments, etc.
  - Statements: if, while, try-catch-finally, etc.
  - Annotations: set, assert, requires, ensures, invariant.

A subset of Java-like programs - 2

- We did some typical simplifications.
  - Methods have only one argument
  - Local variables declared at the beginning
  - No `return` instruction
- Some things where not modelled.
  - Only basic things of the inheritance apparatus were modelled (method lookup)
  - Static fields, static overloading, initialization

Characteristics of the specification - 1

- To deal with termination, the semantics requires the length of the derivation sequence.
- We have one parametric semantics that we instantiate to get the behaviour of annotated programs and (annotated) monitored programs.

Characteristics of the specification - 2

- The syntax of programs is described by a datatype with mutually recursive subtypes:

```
Body[Name: TYPE+]: DATATYPE WITH SUBTYPES Expr, Stmt
  Assign(target: Name, source: Expr): Assign? : Expr
  While(test: BoolExpr, body: Stmt): While?   : Stmt
```

- This allows us to have only one semantic function instead of two mutually recursive functions: one for expressions and one for statements.

Characteristics of the specification - 3

- The functions passed as parameters to the semantics theory to define `derive` need a way to do their own computations.
- PVS does not provide built-in support for mutual recursive functions. They are emulated by passing functions as arguments.

```
derive_type(n: nat): TYPE = [FullProgram →
  [Body, FullState, Val, FullState → [bellow(n) → bool]]]
derive_rec_type(n: nat): TYPE =
  [k: upto(n) → derive_type(k)]
```

States

```
MonitoredProgram: TYPE = [# mva: MVA, program: Program #]

Store: TYPE = [Name -> Val]

AState: TYPE = [# cp: CP, stA: Store #]

PState: TYPE = [# ex: lift[Excpt], fvs, lvs: Store #]

APState: TYPE = PState WITH [# gvs: Store #]

MPState: TYPE = APState WITH [# astate: AState #]
```

The equivalence relation - 1

```
MVA_modeled?(mp)(sA: AState, sAP: APState): boolean =
  MVA_cp_modeled?(mp)(sA, sAP) AND
  MVA_cps_modeled?(mp)(sAP) AND
  MVA_vars_modeled?(sA, sAP)

Program_modeled?(sMP: MPState, sAP: APState): boolean =
  pstate(sMP) = pstate(sAP) AND
  Program_gvs_modeled?(sMP, sAP)
```

The equivalence relation - 2

```
halted_implies_JMLExc(mp)(sMP: MPState, sAP: APState): boolean =
  cp(astate(sMP)) = halted IMPLIES
    (up?(ex(pstate(sAP))) AND down(ex(pstate(sAP))) = JMLExc)

related_states(mp)(sMP: MPState, sAP: APState): boolean =
  wf_state(mp)(sMP) AND
  wf_state(ann_program(mp))(sAP) AND
  MP_modeled?(mp)(sMP, sAP) AND
  halted_implies_JMLExc(mp)(sMP, sAP)
```

Correctness property in PVS

```
correctness_of_ann_program: THEOREM
  FORALL (mp)(main: Method, arg: int)
         (sMP: MPState, sAP: APState):
    well_behaved_MP(mp) IMPLIES
    run_monitored_program(mp)(main, arg)
                              (sMP) IMPLIES
    run_annotated_program(ann_program(mp))(main, arg)
                              (sAP) IMPLIES
      related_states(mp)(sMP, sAP)
```

The invariant

```
derive_maintains_related_states : THEOREM
  FORALL (mp)(b: Body, v1, v2: Val)
         (sMP1, sMP2: MPState, sAP1, sAP2: APState)
         (n1, n2 : nat):
    well_behaved_MP(mp) IMPLIES
    related_states(mp)(sMP1, sAP1) IMPLIES
    derive(mp)(b, sMP1, v1, sMP2)(n1) IMPLIES
    derive(ann_program(mp))(b, sAP1, v2, sAP2)(n2) IMPLIES
      related_states(mp)(sMP2, sAP2) AND v1 = v2
```

Sketch of the proof of step 1

- The initial states are equivalent.
- Prove `derive_maintains_related_states`.
  - The proof is by induction on the length of the derivation sequence.
  - The method call case is the interesting one. Here is where we have to show that `ann_program` is correct.
- Prove `correctness_of_ann_program`.

Advantages of having a formalization - 1

- Although the ideas are simple we found many subtleties.
  - `assert` at the end of the pre_set.
  - in the proof the `try-catch-finally` case is tricky.

Advantages of having a formalization - 2

- Makes all the requirements explicit.
  - No clash between variable names of the automaton and the monitored class.
  - The evaluation of expressions appearing on guards or actions can not have side effects nor throw exceptions.
  - There must be an injective function from the set of control points to i nt .

Future work

- Prove the correctness of the second step.
- Generate preconditions and postconditions.
- Prove that some properties can be checked statically.
  - Extend the propagation algorithm given by Mariela Pavlova.
  - Formalize it in PVS by extending this work and prove its correctness.

Related work - 1

- Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct Java card applets.
- Daan de Jong. Converting Midlet Navigation Graphs into JML
- Jesús Ravelo and Erik Poll. Work in progress about graph refinement.

Related work - 2

- Mariela Pavlova. Generation of JML specification for Java card applications.
- Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman and Jean-Louis Lanet. Enforcing high-level security properties for applets.
- Yoonsik Cheon and Ashaveena Perumendla. Specifying and checking method call sequences of Java programs.

The end

Thanks!

Questions?