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)

\[
\text{init; (start; stop)+; destroy}
\]
Example: The applet protocol specified in JML (Cheon and Perumendla)

```java
package java.applet

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

    //@ public ghost int state = PRI STI NE;
    //@ requires state == PRI STI NE;
    //@ ensures state == INIT;
    public void init() {
        //@ set state = INIT;
        ...
    }

    //@ requires state == INIT;
    //@ 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;
        ...
    }

    ...
```
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 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.

Automaton:
Monitored class: transactions.java
Q = \{Q1, Q2, Q3\}
\(\Sigma = \{bt, bt, bt, ct, ct, ct, at\}\)
vars\(_A\) = \{(t, int, 0)\}
vars\(_P\) = \{\}
Other properties

- Enforce and order in which methods are called: **lifecycle** 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

\[ P = \text{program (may already have annotations)} \]
\[ A = \text{automaton describing a security property} \]
\[ \parallel = \text{monitored by} \]
\[ \approx = \text{equivalence relation} \]

**Assumptions:** P does not throw nor catch JML exceptions

\[ A \text{ is “well formed” and “well behaved”} \]

\[ P \parallel A \approx \text{ann\_program}(P, A) \]
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.
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

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

//@ public ghost int cp = Q1;
//@ public ghost int t = 0;
```
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

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

m()
assert pre & inv;
pre_set;
body;
ex → assert inv;
exc_set;
!ex → assert pos & inv;
pos_set;
```
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

```c
/*@ 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;
  @}
  @*/
```
The `if` for ghost variables are translated into a sequence of `set` statements using conditional statements.

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

```plaintext
set x := c ? a : x;
set y := c ? b : y;
```
Two auxiliary ghost variables are used to ensure the independence of the branches.

```plaintext
if (cp == Q4) {
    if (x >= 5) {
        set x = x-1;
        set cp = Q2;
    } else {
        set x = x+1;
        set cp = Q4;
    }
}
else {
    set cp = HALTED;
}
```

```plaintext
set b1 = cp == Q4;
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 ? Q4 : y;
set b2 = b1 && !b2;
set cp = b2 ? HALTED : cp;
```
Step 2 – 2: Refine pre_set et al.

```java
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

```java
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;
    }
}
```
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, NullPointerException, JMLExc
  - Expressions: method calls, assignments, etc.
  - Statements: if, while, try-catch-finally, etc.
  - Annotations: set, assert, requires, ensures, invariant.
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
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.
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: Bool Expr, 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.
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.

\[
\text{derive}_\text{type}(n: \text{nat}): \text{TYPE} = [\text{FullProgram} \rightarrow \\
[\text{Body, FullState, Val, FullState} \rightarrow [\text{below}(n) \rightarrow \text{bool}]]] \\
\text{derive}_\text{rec_type}(n: \text{nat}): \text{TYPE} = \\
[ k: \text{upto}(n) \rightarrow \text{derive}_\text{type}(k) ]
\]
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_model ed?( mP)( sA: AState, sAP: APState): boolean =
MVA_cp_model ed?( mP)( sA, sAP) AND
MVA_cps_model ed?( mP)( sAP) AND
MVA_vars_model ed?( sA, sAP)

Program_model ed?( sMP: MPState, sAP: APState): boolean =
pstate( sMP) = pstate( sAP) AND
Program_gvs_model ed?( sMP, sAP)
The equivalence relation - 2

\[
\text{halted_implies_JMLExc}(mp)(sMP; MPState, sAP; APState): \text{boolean} = \\
\text{cp}\(\text{astate}(sMP)\) = \text{halted IMPLIES} \\
\text{up?}(\text{ex}(\text{pstate}(sAP))) \text{ AND } \text{down}(\text{ex}(\text{pstate}(sAP))) = \text{JMLExc}
\]

\[
\text{related_states}(mp)(sMP; MPState, sAP; APState): \text{boolean} = \\
\text{wf_state}(mp)(sMP) \text{ AND} \\
\text{wf_state}(\text{ann_program}(mp))(sAP) \text{ AND} \\
\text{MP_modeled?}(mp)(sMP, sAP) \text{ AND} \\
\text{halted_implies_JMLExc}(mp)(sMP, sAP)
\]
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

\[
\text{derive_maintains_related_states} : \ \text{THEOREM}
\]

\[
\text{FORALL } (\ mp)(b: \ \text{Body}, \ v_1, \ v_2: \ \text{Val})
\]

\[
(\ sMP_1, \ sMP_2: \ \text{MPState}, \ sAP_1, \ sAP_2: \ \text{APState})
\]

\[
(\ n_1, \ n_2: \ \text{nat}): \text{well_behaved_MP}(\ mp) \ \text{IMPLIES}
\]

\[
\text{related_states}(\ mp)(sMP_1, \ sAP_1) \ \text{IMPLIES}
\]

\[
\text{derive}(\ mp)(b, \ sMP_1, \ v_1, \ sMP_2)(n_1) \ \text{IMPLIES}
\]

\[
\text{derive}(\ \text{ann_program}(\ mp))(b, \ sAP_1, \ v_2, \ sAP_2)(n_2) \ \text{IMPLIES}
\]

\[
\text{related_states}(\ mp)(sMP_2, \ sAP_2) \ \text{AND} \ v_1 = v_2
\]
Sketch of the proof of step 1

- The initial states are equivalent.

- Prove derive_maintains_relations.
  - 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
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 int.
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.
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?