Enforcing high-level security properties using JML

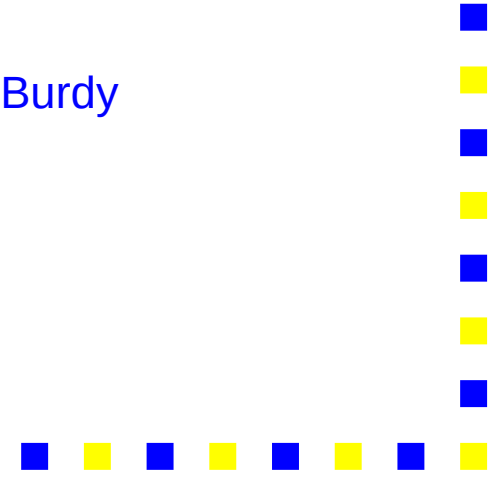
Marieke Huisman

Marieke.Huisman@inria.fr

INRIA Sophia Antipolis

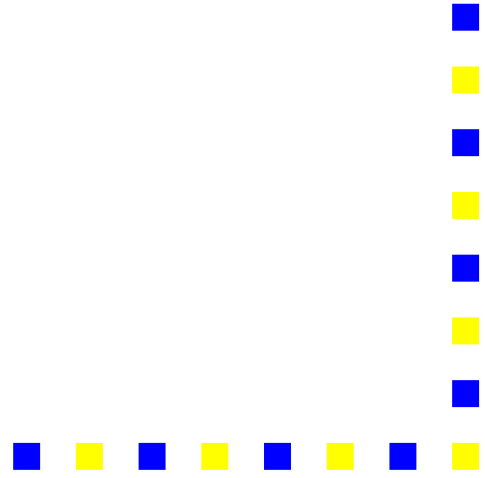
joint work with Mariela Pavlova, Gilles Barthe, Lilian Burdy

Jean-Louis Lanet & Igor Siveroni



Challenges within GECCOO

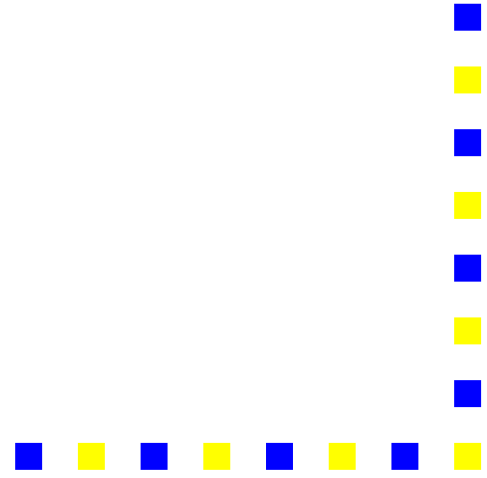
- Formally specifying high-level security properties
- Semantics of specifications
- Composing specifications



Challenges within GECCOO

- Formally specifying high-level security properties
- Semantics of specifications
- Composing specifications

This talk: focus on first point

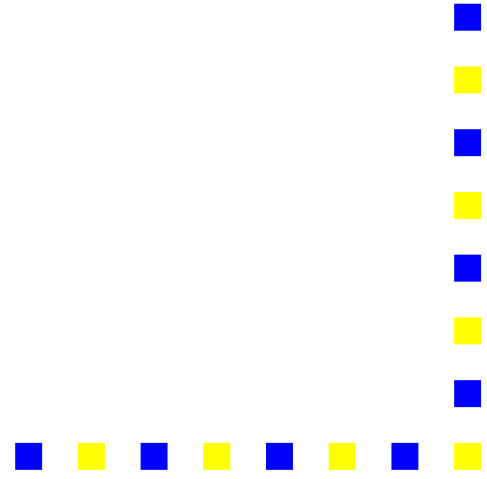


Contents

- JML: what is there already
- Security properties
- A method for annotation generation
- Towards security automata

Design goals JML

- Readable specifications
- Provides unambiguous documentation
- Usable for any Java application
- Rigorous, formal semantics - amenable to tool support

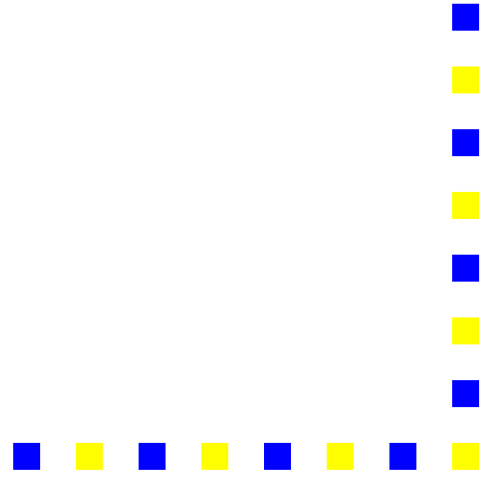


Design goals JML

- Readable specifications
- Provides unambiguous documentation
- Usable for any Java application
- Rigorous, formal semantics - amenable to tool support
  jmlc, ESC/Java(2), LOOP, Krakatoa, JACK

Limitations of JML

- Specifications restricted to a single class
- Specifications restricted to functional behaviour
- Not much support for refinement



Limitations of JML

- Specifications restricted to a single class
- Specifications restricted to functional behaviour
- Not much support for refinement

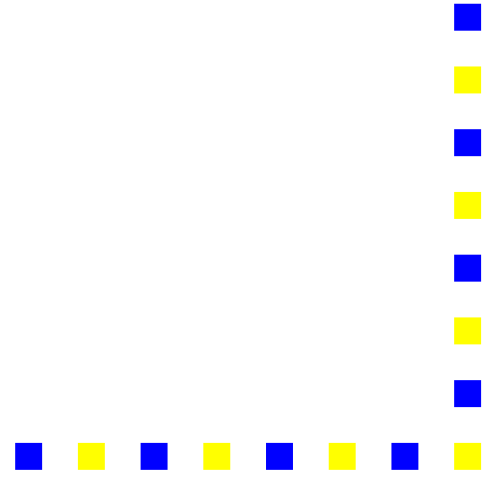
In order to verify security properties, one needs to overcome these limitations

Enforcing security properties

- Security expert: from global notions of security to set of rules
- Developers: try to obey rules
- Security audit: manual code inspection whether rules obeyed
- Need for tools to help security audit

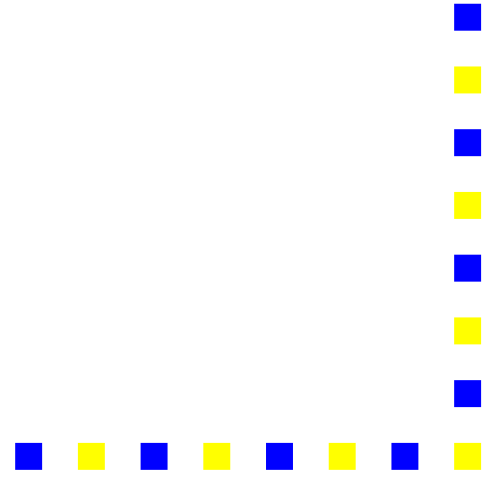
Questions to be addressed

- Relation between security notions and rules
- Guaranteeing rules are obeyed



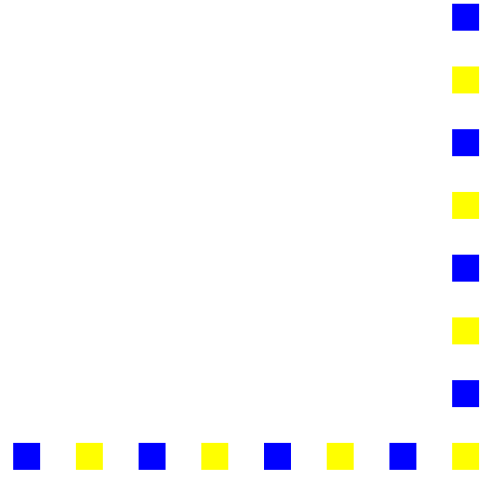
Questions to be addressed

- Relation between security notions and rules
- Guaranteeing rules are obeyed



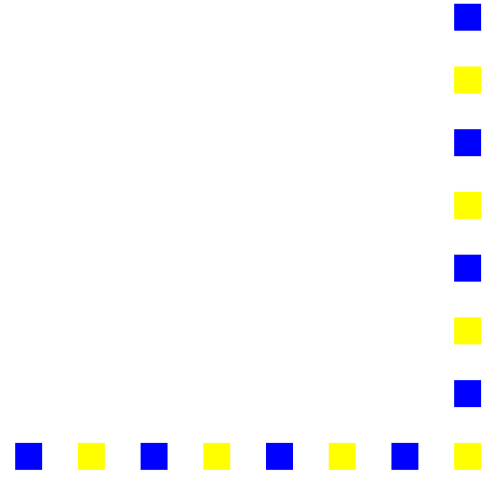
Questions to be addressed

- Relation between security notions and rules
- Guaranteeing rules are obeyed
- Extension towards security automata: *bridging* the gap between security notions and rules



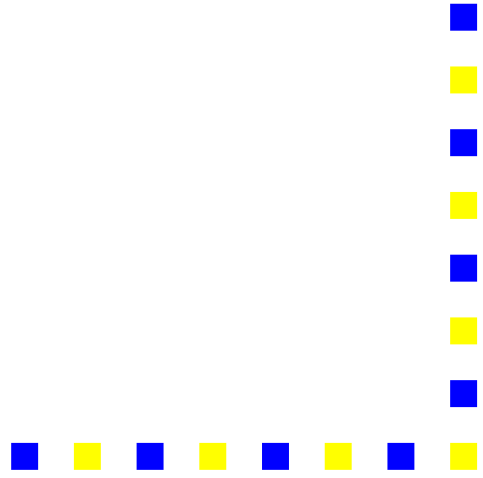
High-level security rules

- Atomicity
- Applet life cycle
- Exception handling
- Access control



High-level security rules

- Atomicity
  - No nested transactions
  - No uncaught exception in transactions
  - Bounded retries authentication
- Applet life cycle
- Exception handling
- Access control

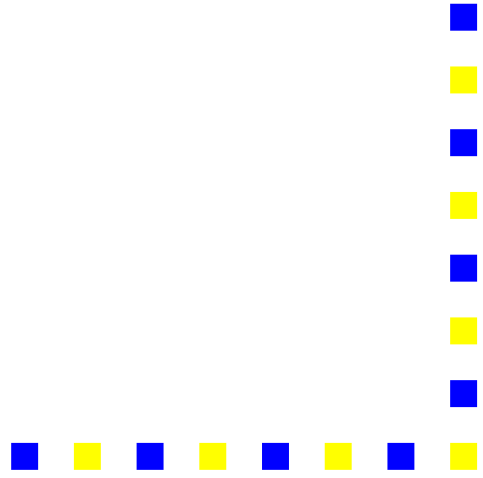


High-level security rules

- Atomicity
- Applet life cycle
  - Authenticated initialisation
  - Authenticated unblocking
  - Single personalisation
- Exception handling
- Access control

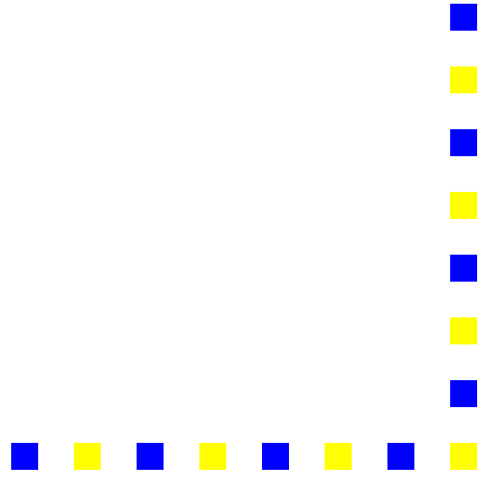
High-level security rules

- Atomicity
- Applet life cycle
- Exception handling
  - Only ISOExceptions at top-level
- Access control

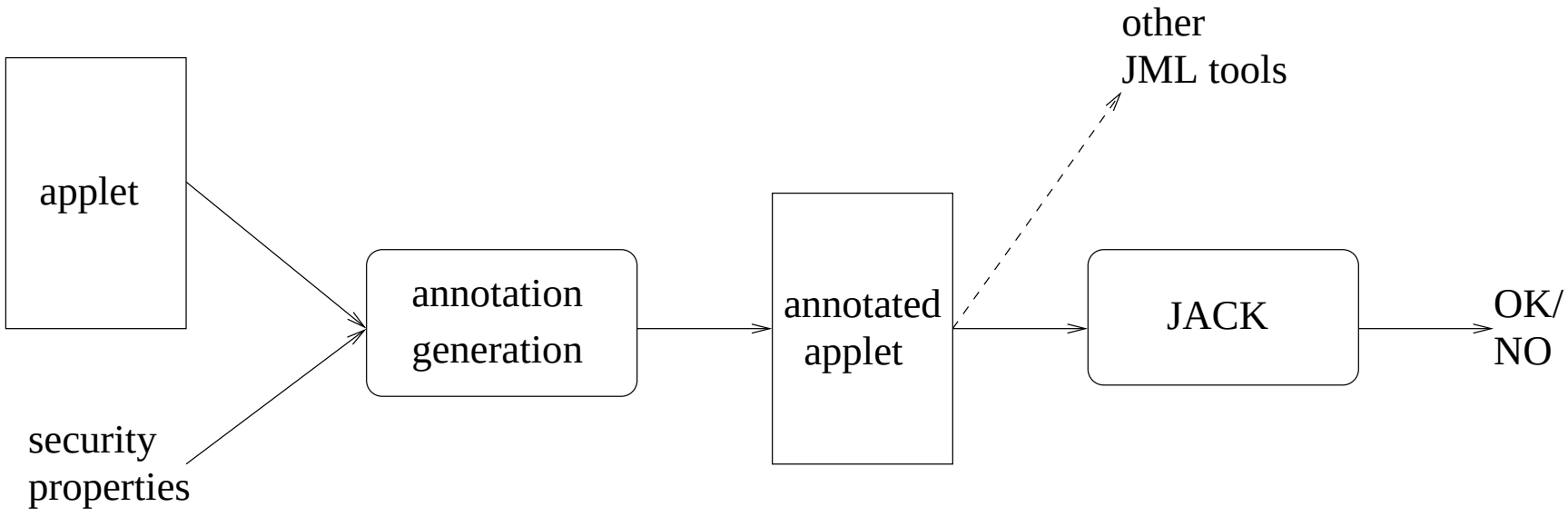


High-level security rules

- Atomicity
- Applet life cycle
- Exception handling
- Access control
  - Only selectable applications shareable



Architecture



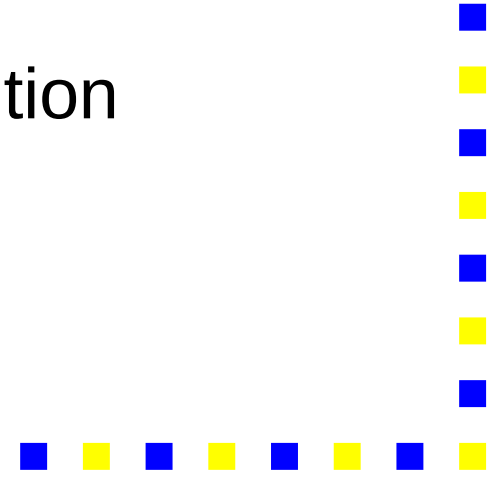
Assertion generation

Two phases:

- synthesising core-annotations
- weaving annotations throughout the application

Synthesising: for each property annotations have to be defined

Weaving: algorithm for pre- and postcondition generation



Example core-annotations

No nested transactions

```
/*@ static ghost int TRANSACT == 0; @*/
```

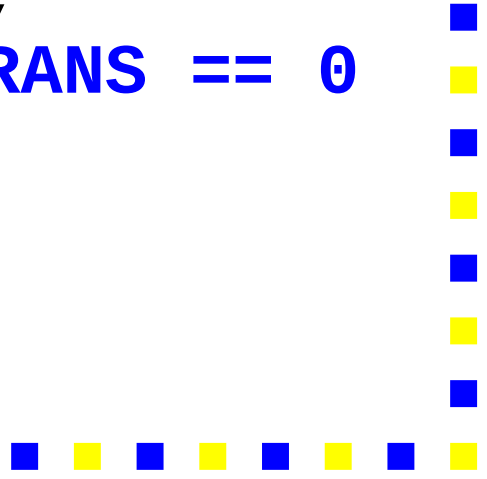
Method **beginTransaction**

```
/*@ requires TRANSACT == 0;
  @ assignable TRANSACT;
  @ ensures TRANSACT == 1; @*/
public static native
   void beginTransaction()
        throws TransactionException;
```

Similar annotations for **commitTransaction**, **abortTransaction**

Preconditions for methods

```
public void m() {
    ...
    // will require TRANS == 0
    JCSystem.beginTransaction();
    // TRANS modified, ensures TRANS == 1
    ...
    // will require TRANS == 1
    JSSystem.commitTransaction();
    // TRANS modified, ensures TRANS == 0
    ...
    }
```



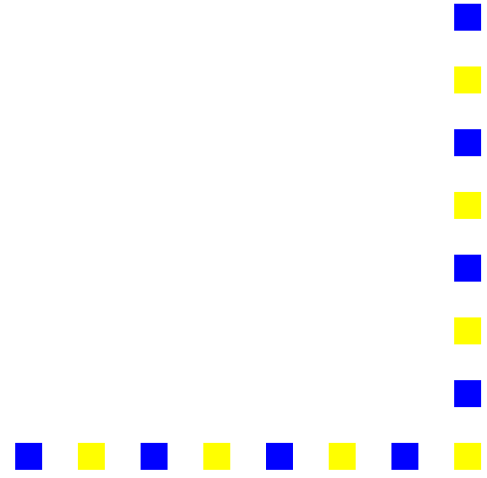
Preconditions for methods

```
public void m() {
    ...
    // will require TRANS == 0
    JCSystem.beginTransaction();
    // TRANS modified, ensures TRANS == 1
    ...
    // will require TRANS == 1
    JSSystem.commitTransaction();
    // TRANS modified, ensures TRANS == 0
    ...
    }
```

Thus: precondition for `m()`: `TRANS == 0`

Some assumptions we make

- Only variables in annotations are static and ghost of primitive type
- Appropriate modelling of programs
- Existence of function mod: set of static ghost variables modified by statement



Algorithm for precondition generation

- Method declaration

$$\mathsf{pre}(m) = \mathsf{pre}(\mathsf{body}(m), \emptyset)$$

- Composition

$$\mathsf{pre}(s_1;s_2, V) = \mathsf{pre}(s_1, V) \cup \mathsf{pre}(s_2, V \cup \mathsf{mod}(s1))$$

- Method call

$$\mathsf{pre}(\mathsf{call}(n), V) = \{p \mid p \in \mathsf{pre}(n) \wedge \mathsf{fv}(p) \cap V = \emptyset\}$$

Algorithm for precondition generation - branching statements

- Conditional

$$\begin{aligned}\mathsf{pre}(\texttt{if}\ (c)\ s_1\ \texttt{else}\ s_2, V) = \quad & \mathsf{pre}(c, V) \cup \\ & \mathsf{pre}(s_1, V \cup \mathsf{mod}(c)) \cup \\ & \mathsf{pre}(s_2, V \cup \mathsf{mod}(c))\end{aligned}$$

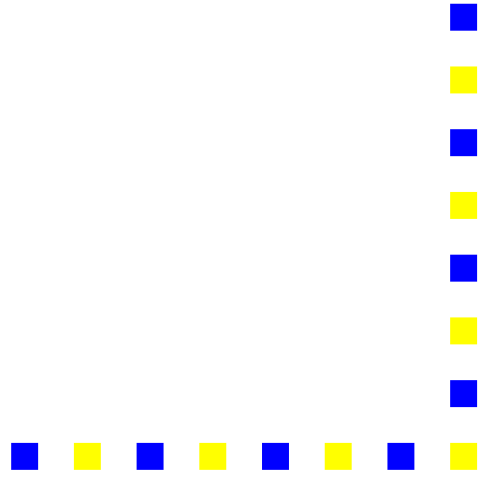
- Try catch

$$\begin{aligned}\mathsf{pre}(\texttt{try}\ s_1\ \texttt{catch}\ (E)\ s_2, V) = \quad & \mathsf{pre}(s_1, V) \cup \\ & \mathsf{pre}(s_2, V \cup \mathsf{mod}(s1))\end{aligned}$$

Experiments: checking atomicity properties

No uncaught exceptions in transactions

```
/*@ exsures (Exception) TRANSACT == 0; @*/
public static native void
    commitTransaction()
          throws TransactionException;
```



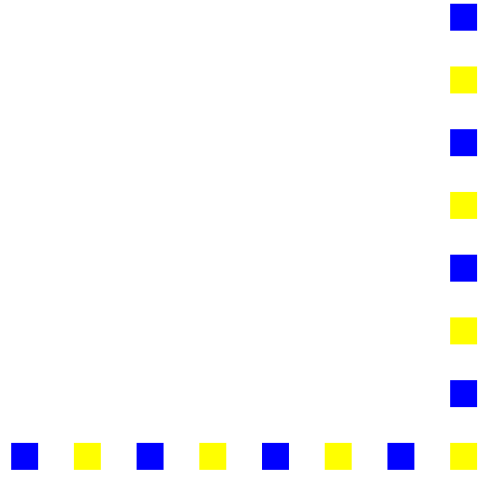
Experiments: checking atomicity properties

No pin-verification within transaction

```
/*@ requires TRANSACT == 0; @*/
public boolean check(byte[] pin,
                     short offset,
                     byte length);
```

Results

- Tested on several realistic smart card applications
- One core-annotation can give rise to many annotations in different classes (26 annotations, spread over 5 different classes)
- Several violations found: uncaught exceptions possible within transactions



Uncaught exception within transaction

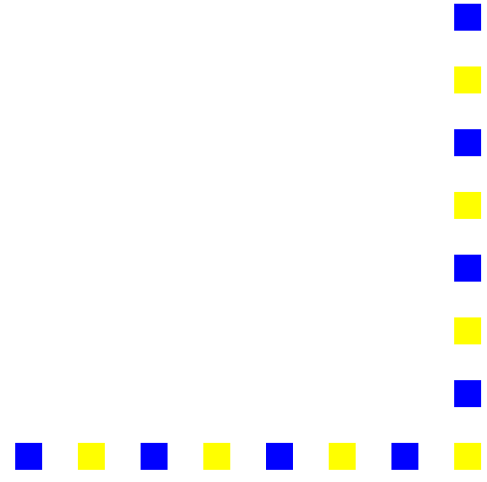
```
void appExchangeCurrency(...) {
    ...
    /*@ exsures (Exception) TRANSACT == 0; @*/
        ...
        JCSystem.beginTransaction();
        try {balance.setValue(decimal2);
                ...
        } catch (DecimalException e) {
            ISOException.throwIt(
                PurseApplet.OVERFLOW);
        }
        JCSystem.commitTransaction();
    }
    ...}
```

Towards more generality: Security automata

- Security automaton: monitors program behaviour, blocking/error upon unwanted behaviour
- Use security automata to express security properties
- Automatic generation of core-annotations
- Use any JML tool to check whether program respects the property

Correctness

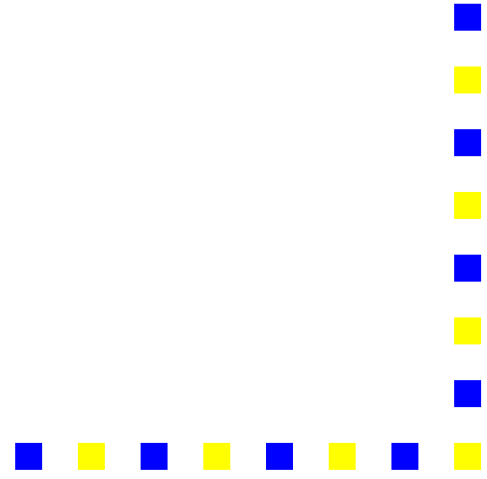
If program satisfies annotations
then it respects the security property



Correctness

If program satisfies annotations
then it respects the security property

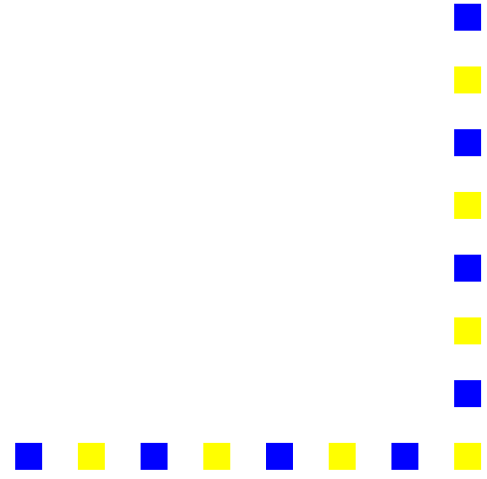
Igor Siveroni (post doc) works on this
First proof for simple while-language on the way



If program satisfies annotations then it respects the security property

Approach:

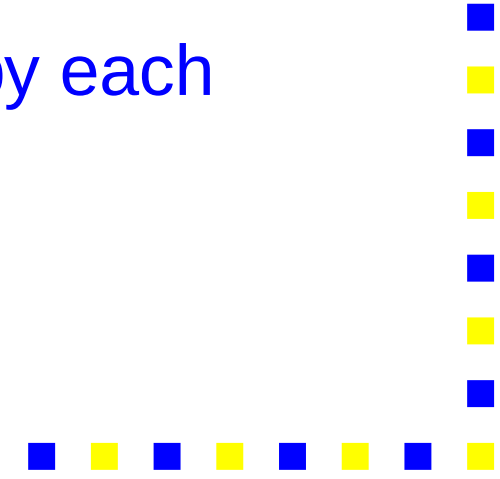
- If program satisfies annotations, run-time monitoring (with `jmlc`) will never block (known result)



If program satisfies annotations then it respects the security property

Approach:

- If program satisfies annotations, run-time monitoring (with `jmlc`) will never block (known result)
- If run-time monitoring (with `jmlc`) of annotated program never blocks, security automaton never blocks
  true for core-annotations, preserved by each propagation step



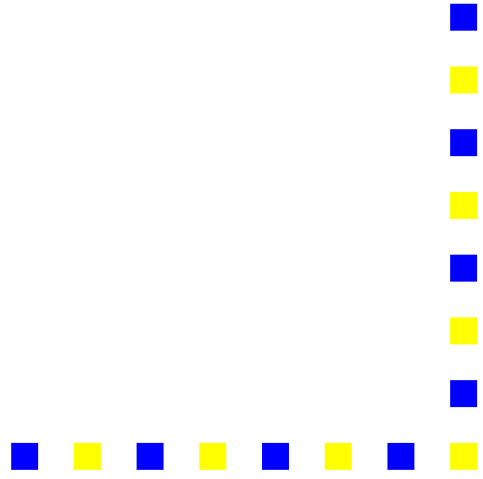
If program satisfies annotations then it respects the security property

Approach:

- If program satisfies annotations, run-time monitoring (with `jmlc`) will never block (known result)
- If run-time monitoring (with `jmlc`) of annotated program never blocks, security automaton never blocks
  true for core-annotations, preserved by each propagation step
- If security automaton does not block, program respects the security property (known result)

Future issues

- Other formalisms to express security properties (with translation into JML)?
- Translations into other formalisms
- Do we need liveness?



Theoretical foundations: abstract weakest precondition

Abstract wp-calculus $\mathsf{wp}^{\#}$: considers only static ghost variables
Example rules:

$$\mathsf{wp}^{\#}(\texttt{if}(c)s_1\ \texttt{else}\ s_2, Q) = \mathsf{wp}^{\#}(c, \mathsf{wp}^{\#}(s_1, Q)) \wedge \mathsf{wp}^{\#}(c, \mathsf{wp}^{\#}(s_2, Q))$$

$$\mathsf{wp}^{\#}(\mathsf{call}(m), Q) = \mathsf{requires}(m) \wedge \forall \mathsf{mod}(m).(\mathsf{ensures}(m) \Rightarrow Q)$$

Results

- Any proof in abstract wp-calculus, is proof in standard wp-calculus

$$\forall P, Q : \mathsf{Pred}, s : \mathsf{Stmt}.(P \Rightarrow \mathsf{wp}^{\#}(s, Q)) \Rightarrow (P \Rightarrow \mathsf{wp}(s, Q))$$

- Precondition generation computes the *free* part of the abstract weakest precondition

$$\exists F : \mathsf{Pred}.\mathsf{wp}^{\#}(s, \lambda x.\mathsf{true}) = (\mathsf{pre}(s, \emptyset) \wedge \forall \mathsf{mod}(s).F)$$