
JACK – a tool for validation of security and behaviour of Java applications

Gilles Barthe

Lilian Burdy

Julien Charles

Benjamin Grégoire

Marieke Huisman

Jean-Louis Lanet

Mariela Pavlova

Antoine Requets

gemalto & INRIA Sophia Antipolis, France

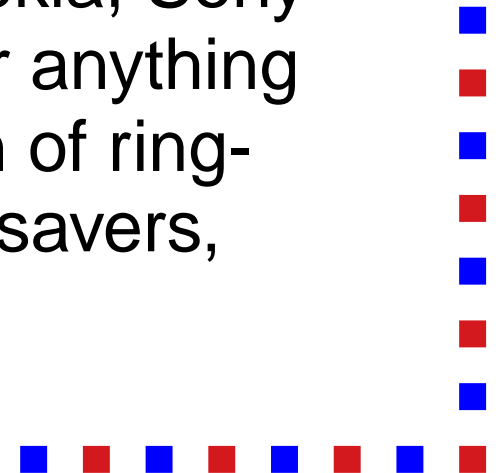


Google on “mobile phone games”

Welcome to Imserba

The best mobile phones portal and community in the world
Mobile phones Portal and Community

Imserba brings you the latest mobile phones related news, informations, stuffs you need for your phones. No matter which phone you are using: Nokia, Sony erricson, Siemens, Samsung, Motorola or anything else, here you can find our best collection of ring-tones, cell phone games, themes, screensavers, backgrounds.



Google on “mobile phone games”

Welcome to Imserba

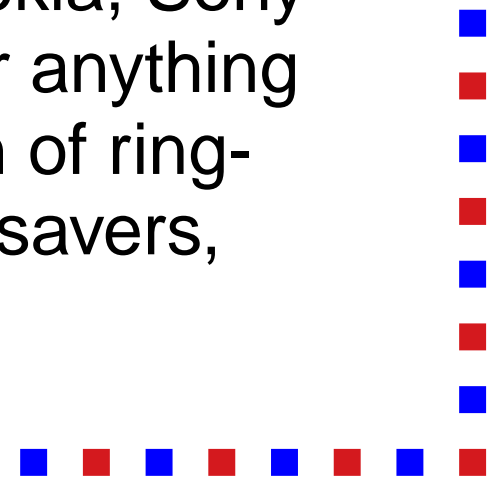
The best mobile phone
world Mobile phone

Imserba brings you
news, information
No matter which phone
ericson, Siemens
else, here you can
tones, cell phone games, themes, screensavers,
backgrounds.



community in the
community

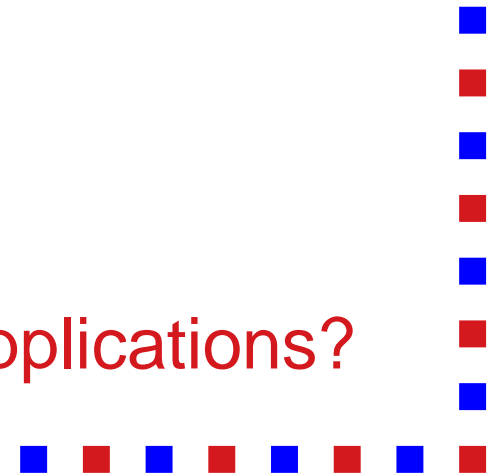
phones related
or your phones.
: Nokia, Sony
la or anything
ction of ring-



Google on “mobile phone games”

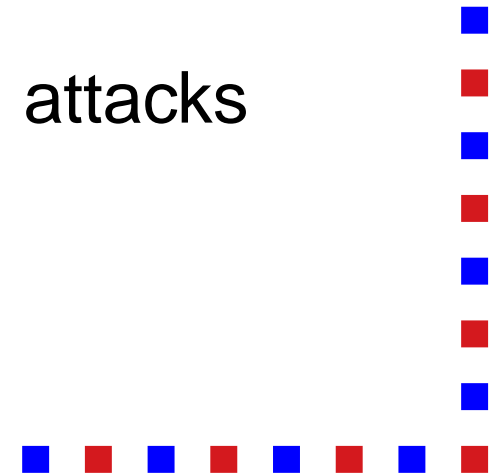


Are you sure that you can trust these applications?



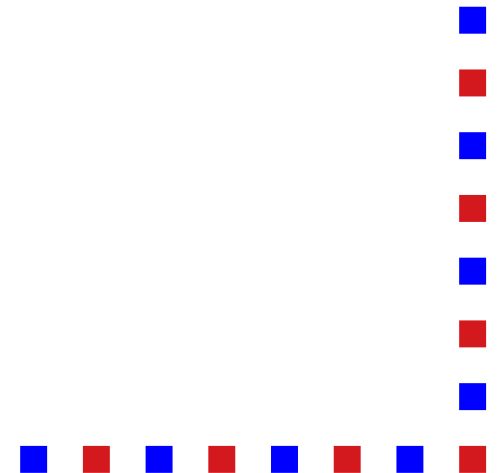
Security for trusted personal devices

- **Trusted personal devices:** phones, smart cards, pda's, set top boxes, . . .
- Used for security-sensitive applications
- Network connected
- Support for complex applications (contain a full JVM)
- Shift from hardware attacks to logical attacks



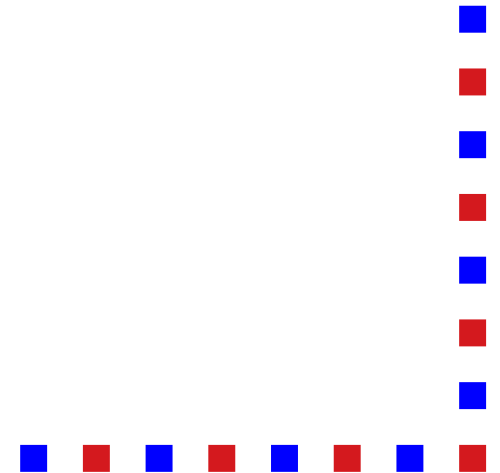
Guaranteeing security

- Formal specification and verification
- Java Modeling Language (JML) able to express security properties
- Classical program calculi can be used
- Large body of theory on sound modular verification
- Proof Carrying Code paradigm



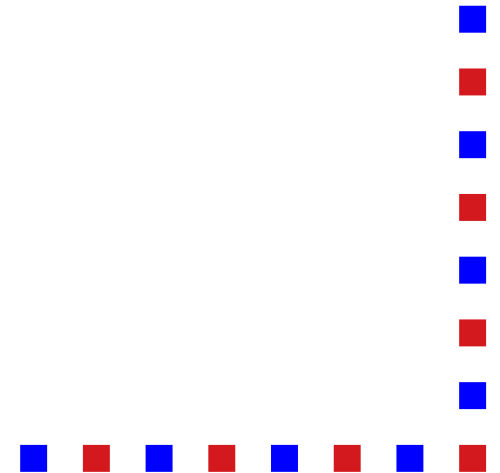
But how to convince developers do this?

- Seamless integration in standard development environment



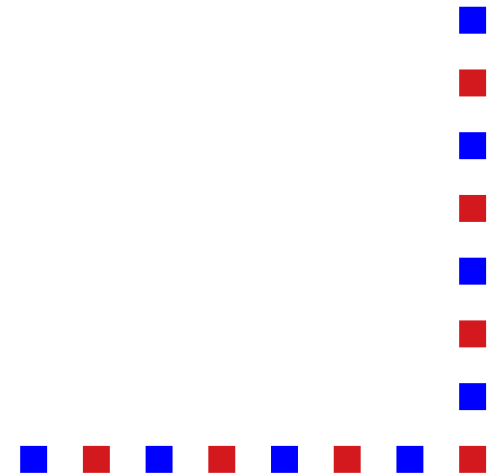
But how to convince developers do this?

- Seamless integration in standard development environment
- Small overhead in specification writing: annotation generation



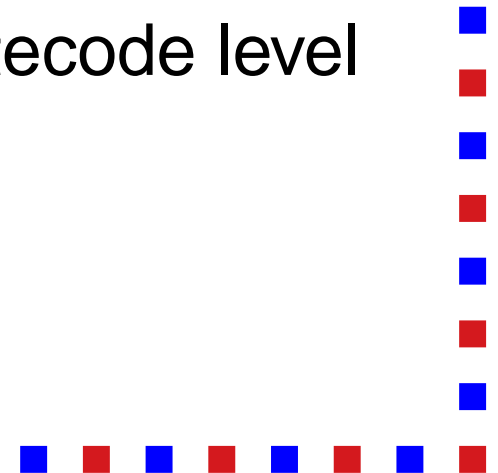
But how to convince developers do this?

- Seamless integration in standard development environment
- Small overhead in specification writing: annotation generation
- Verification conditions automatically generated, proven by automatic theorem prover



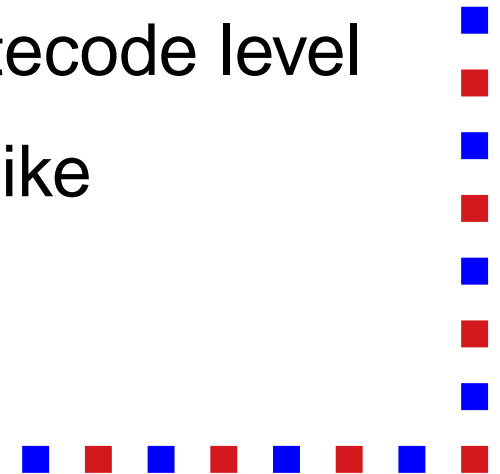
But how to convince developers do this?

- Seamless integration in standard development environment
- Small overhead in specification writing: annotation generation
- Verification conditions automatically generated, proven by automatic theorem prover
- Reasoning at source code *and* at bytecode level

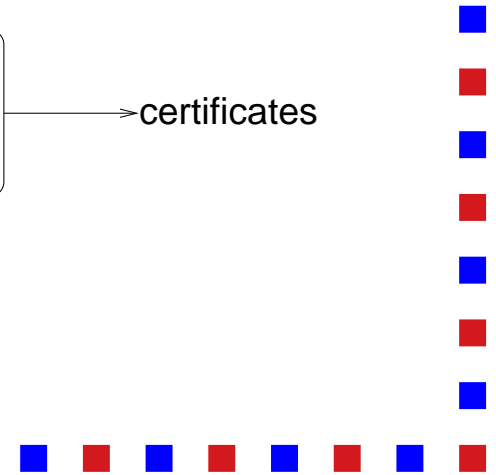
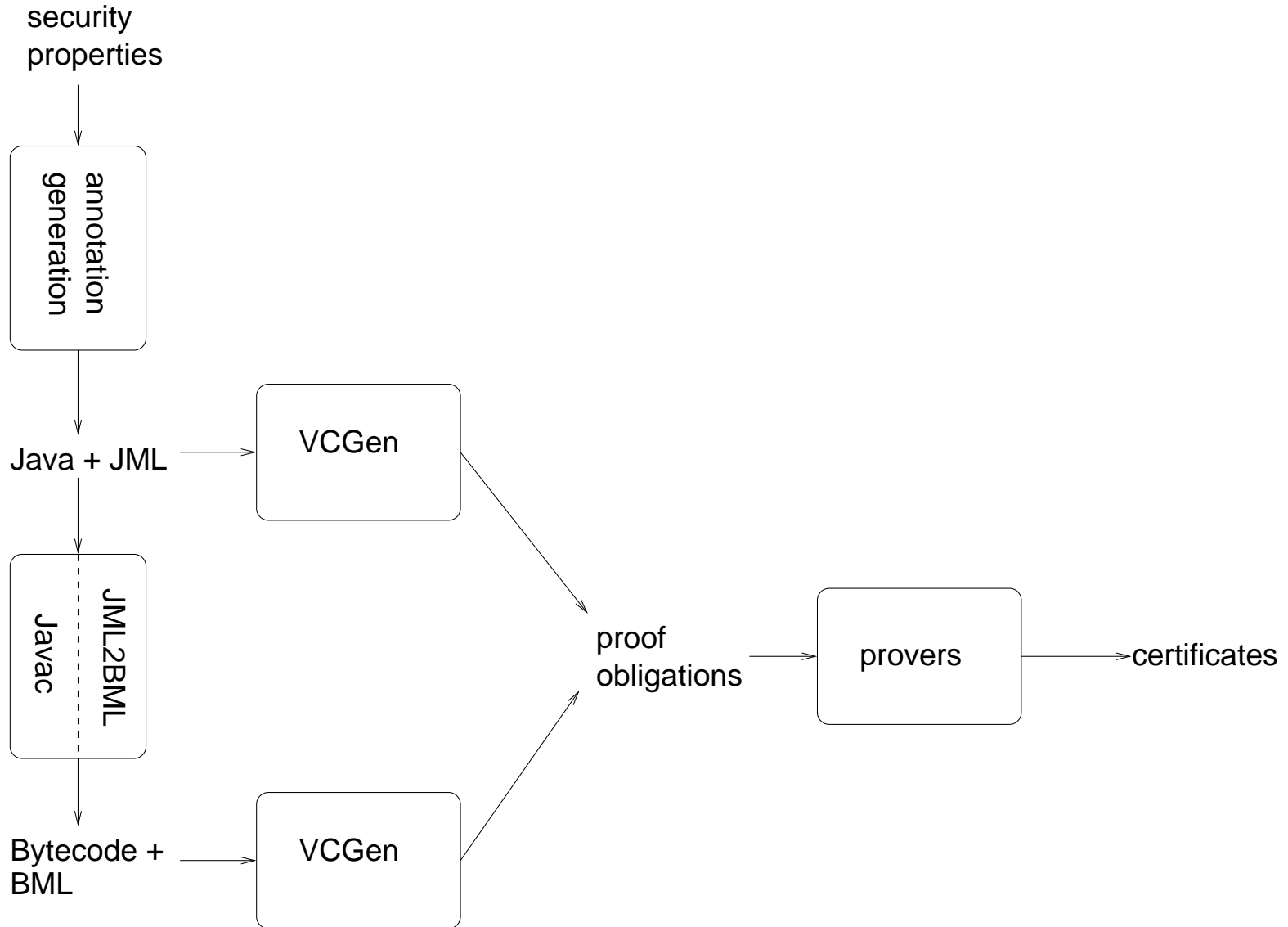


But how to convince developers do this?

- Seamless integration in standard development environment
- Small overhead in specification writing: annotation generation
- Verification conditions automatically generated, proven by automatic theorem prover
- Reasoning at source code *and* at bytecode level
- Advanced support for difficult tasks (like interactive proving)

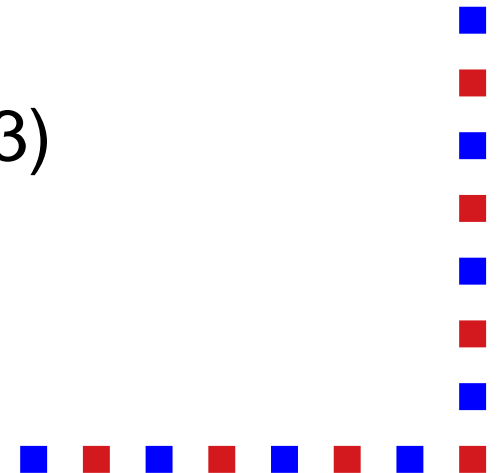


JACK: Java Applet Correctness Kit



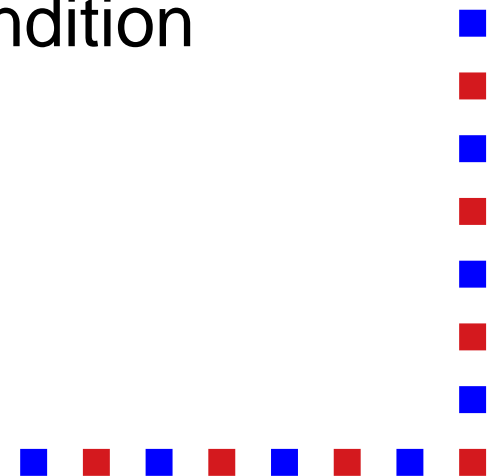
History of JACK

- Development started at Gemplus (Jan 2002 to April 2003)
Objective: Give developers tools that help them to provide and be accountable for quality of their code
 - Conform to specification requirements
 - Well-documented
 - Without bugs
- Transferred to INRIA (September 2003)
 - Correctness stays major concern
 - More features & plugins



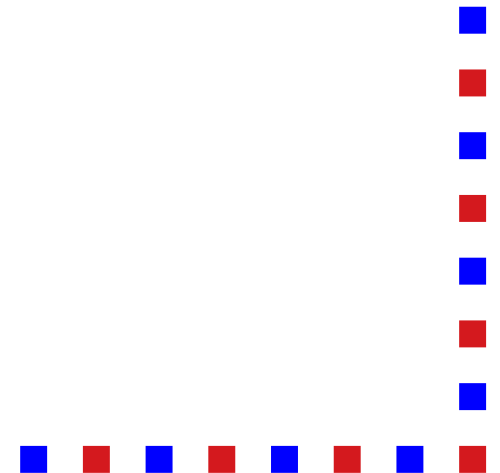
Features of JACK

- Tight integration with IDE Eclipse
- JML used as annotation language
- Different means of validation possible
- Support for Simplify (automatic) and Coq (interactive) prover
- Special JACK view for verification condition browsing

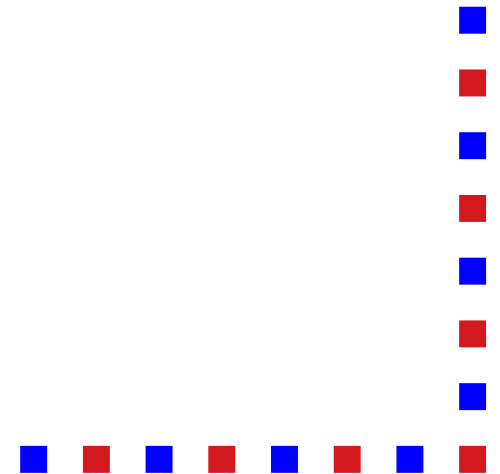


And more features of JACK

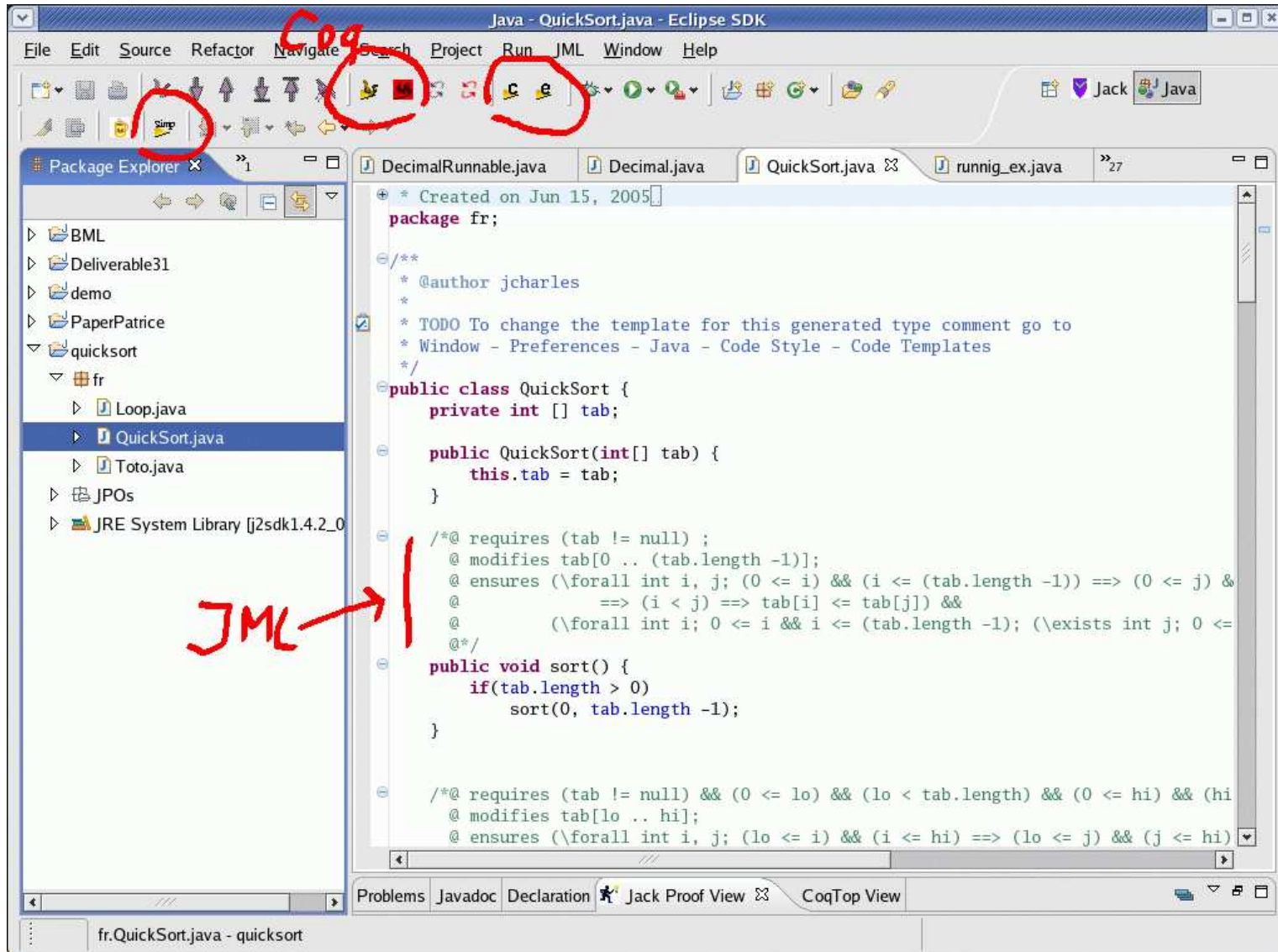
- Generation and propagation of annotations, based on implementation of verification condition generator
- JML specifications compiled into BML (Bytecode Modeling Language)
- Support for verification of bytecode



Integration with Eclipse



Developing an application in Eclipse



Using Simplify

The screenshot shows the Eclipse IDE with the Jack Proof View open. A red arrow points from the Package Explorer to the Jack Proof View. The Jack Proof View table shows the following data:

File	Status	Prover	PO Coun	PO Tried	PO Proved	% Tried	% Proved	
QuickSort.java	ERROR	Simplify	208	208	75	100	36	1

Proof obligation viewer

Jack - QuickSort.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run JML Window Help

Case Explorer - Q... x

- QuickSort
 - sort()
 - Case 1
 - Goal 2
 - Case 2
 - Goal 2
 - Goal 4
 - sort(int, int)
 - Case 1
 - Case 2
 - Case 3
 - Goal 1

Case Viewer - Lemma: 208 - Proved: 75 (36%) - Checked: 0 (0%)

```
this.tab = tab;
}

/*@ requires (tab != null) ;
@ modifies tab[0 .. (tab.length - 1)];
@ ensures (\forallall int i, j; (0 <= i) && (i <= (tab.length - 1)) ==> (0 <= j) && (j <= (tab.length - 1))
@ ==> (i < j) ==> tab[i] <= tab[j]) &&
@ (\forallall int i; 0 <= i && i <= (tab.length - 1); \exists int j; 0 <= j && j <= (tab.length - 1) && \old(tab[j]) == tab[i]);
@*/
public void sort() {
    if(tab.length > 0)
        sort(0, tab.length - 1);
    is_false

/*@ requires (tab != null) && (0 <= lo) && (lo < tab.length) && (0 <= hi) && (hi < tab.length);
@ modifies tab[lo .. hi];
@ ensures (\forallall int i, j; (lo <= i) && (i <= hi) ==> (lo <= j) && (j <= hi)
@ ==> (i < j) ==> tab[i] <= tab[j]) &&
```

Jack Metrics View Jack Proof View Check ensures clause of the current method Progress

this.tab != null
!this.tab.length > 0
!(this.tab == null)
REFERENCES this
(this) : (instances)
(typeof(this)) <: QuickSort

Java Coq Simplify PVS

forall int i; 0 <= i
&& i <= this.tab.length - 1
==> \exists int j; 0 <= j
&& j <= this.tab.length - 1
&& \old(this.tab[j]) == this.tab[i]

QuickSort.java x runnig_ex.java »29 CoqTop View x

Reasoning with method calls

The screenshot displays the Eclipse IDE with the Jack proofing environment. The main editor shows the source code for `QuickSort.java` with several JML annotations. The `Case Explorer` on the left shows a tree of cases and goals. The `Case Viewer` on the right shows the current case being worked on, with a progress indicator. The `Jack Metrics View` at the bottom left lists various goals, including `this.tab != null`, `0 <= lo`, `lo < this.tab.length`, `0 <= hi`, `hi < this.tab.length`, and `!(this.tab == null)`. The `CoqTop View` at the bottom right shows the Coq editor version 1.1.2. Red arrows indicate the flow of information: one arrow points from the `requires` clause of the `sort` method signature in the source code to the `0 <= lo` goal in the metrics view, and another arrow points from the `lo = left + 1` goal in the metrics view to the `sort(left + 1, hi);` call in the source code.

```
if(left > 0)
  sort(lo, left - 1);
/*@ assert (forall int i, j; (lo <= i) && (i < left) ==> (lo <= j) && (j < left)
=> (i < j) ==> tab[i] <= tab[j]) &&
@
(forall int i; (lo <= i) && (i < left)
=> tab[i] <= tab[left]) &&
@
(forall int j; (left < j) && (j <= hi)
=> tab[left] <= tab[j]) &&
@
(forall int i; lo <= i && i <= hi; (exists int j; lo <= j && j <= hi && \old(tab[j]) == tab[i]));
@*/
if(left + 1 < tab.length)
  sort(left + 1, hi);
/*@ assert (forall int i, j; (lo <= i) && (i <= hi) ==> (lo <= j) && (j <= hi)
=> (i < j) ==> tab[i] <= tab[j]) &&
@
(forall int i; lo <= i && i <= hi; (exists int j; lo <= j && j <= hi && \old(tab[j]) == tab[i]));
@*/
}
```

```
fr.QuickSort.sort(int lo, int hi)
requires this.tab!=null&&
0<=lo&&
lo<this.tab.length&&
0<=hi&&
hi<this.tab.length;
requires true;
```

```
0 <= left(after loop at line 55) + 1
lo = left + 1
```

Reasoning about exceptions

Jack - QuickSort.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run JML Window Help

Case Explorer - Q... Case Viewer - Lemma: 208 - Proved: 75 (36%) - Checked: 0 (0%)

```
/*@ requires (tab != null) && (0 <= lo) && (lo < tab.length) && (0 <= hi) && (hi < tab.length);
@ modifies tab[lo .. hi];
@ ensures (\forall int i, j; (lo <= i) && (i <= hi) ==> (lo <= j) && (j <= hi)
@      ==> (i < j) ==> tab[i] <= tab[j]) &&
@      (\forall int i; lo <= i && i <= hi; (\exists int j; lo <= j && j <= hi && \old(tab[j]) == tab[i]));
@*/
private void sort(int lo, int hi) {
    int left, right, pivot;
    if(!(lo < hi)) return;
    left = lo;
    right = hi;
    pivot = tab[hi];
    /*@ loop_invariant left, right, tab[lo..(hi - 1)];
@ loop_invariant (lo <= left) && (left <= right) && (right <= hi) &&
@ (\forall int m; (lo <= m) && (m < left) ==> tab[m] <= pivot)
@ && (\forall int n; (right < n) && (n <= hi) ==> pivot <= tab[n]) &&
@ tab[right] >= pivot &&
@ (\forall int i; lo <= i && i <= hi - 1; (\exists int j; lo <= j && j <= hi && \old(tab[j]) == tab[i]));
@*/
}
```

is null

exsures false

Jack Metrics View Jack Proof View Check exsures clause of the method Progress

- this.tab != null
- 0 <= lo
- lo < this.tab.length
- 0 <= hi
- hi < this.tab.length
- !(newObject_30) : (instances)

Java Coq Simplify PVS

QuickSort.java runnig_ex.java CoqTop View

Proof obligations

The screenshot displays the Eclipse IDE interface for a Jack-annotated QuickSort.java file. The Case Explorer on the left shows a hierarchy of cases (Case 1 to Case 14) and goals (Goal 1, Goal 2). The Case Viewer on the right shows the source code with Jack annotations, including `@requires`, `@modifies`, `@ensures`, `@loop_invariant`, and `@decreases`. The bottom panel shows a list of proof obligations, with the selected obligation `i < this.tab.length` highlighted. The corresponding code snippet for this obligation is shown on the right side of the bottom panel.

```
/*@ requires (tab != null) && (0 <= lo) && (lo < tab.length) && (0 <= hi) && (hi < tab.length);
@ modifies tab[lo .. hi];
@ ensures (forall int i, j; (lo <= i) && (i <= hi) ==> (lo <= j) && (j <= hi)
==> (i < j) ==> tab[i] <= tab[j]) &&
@ (forall int i; lo <= i && i <= hi; (exists int j; lo <= j && j <= hi && (old(tab[j]) == tab[i]));
@*/
private void sort(int lo, int hi) {
int left, right, pivot;
if(!(lo < hi)) return;
left = lo;
right = hi;
pivot = tab[hi];

/*@ loop_modifies left, right, tab[lo..(hi - 1)];
@ loop_invariant (lo <= left) && (left <= right) && (right <= hi) &&
@ (forall int m; (lo <= m) && (m < left) ==> tab[m] <= pivot)
@ && (forall int n; (right < n) && (n <= hi) ==> pivot <= tab[n]) &&
@ tab[right] >= pivot &&
@ (forall int i; lo <= i && i <= hi - 1; (exists int j; lo <= j && j <= hi && (old(tab[j]) == tab[i]));
@ decreases (right - left);
@*/
while(left < right) {
//@ ghost int oldright;
//@ ghost int oldleft;

//@ set oldright = right;
//@ set oldleft = left;

forall int i, j; lo <= i
&& i <= hi
==> (lo <= j
&& j <= hi
==> (i < j
==> this.tab[i] <= this.tab[j]))
```

Proof obligations in Coq

The screenshot displays the Eclipse IDE interface for a Coq project named 'Jack - QuickSort.java'. The main editor shows the source code for a `sort` function with several annotations: `@requires`, `@modifies`, `@ensures`, `@loop_invariant`, and `@decreases`. The left-hand 'Case Explorer' pane shows a tree structure with 'Goal 1' selected. The bottom pane, titled 'Check assertion', lists various proof obligations (e.g., `(f_tab this) <> null`, `(j_le 0 l_lo)`, `(j_lt l_lo (arraylength (f_tab this)))`) and their corresponding logical expressions in Coq notation.

```
/*@ requires (tab != null) && (0 <= lo) && (lo < tab.length) && (0 <= hi) && (hi < tab.length);
@ modifies tab[lo .. hi];
@ ensures (forall int i, j; (lo <= i) && (i <= hi) ==> (lo <= j) && (j <= hi)
==> (i < j) ==> tab[i] <= tab[j]) &&
@ (forall int i; lo <= i && i <= hi; (exists int j; lo <= j && j <= hi && (old(tab[j]) == tab[i]));
@*/
private void sort(int lo, int hi) {
int left, right, pivot;
if(!(lo < hi)) return;
left = lo;
right = hi;
pivot = tab[hi];

/*@ loop_modifies left, right, tab[lo..(hi - 1)];
@ loop_invariant (lo <= left) && (left <= right) && (right <= hi) &&
&& (forall int m; (lo <= m) && (m < left) ==> tab[m] <= pivot)
&& (forall int n; (right < n) && (n <= hi) ==> pivot <= tab[n]) &&
@ tab[right] >= pivot &&
@ (forall int i; lo <= i && i <= hi - 1; (exists int j; lo <= j && j <= hi && (old(tab[j]) == tab[i]));
@ decreases (right - left);
@*/
while(left < right) {
/*@ ghost int oldright;
/*@ ghost int oldleft;

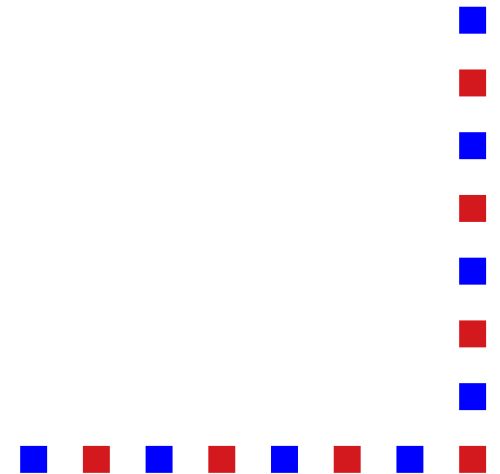
/*@ set oldright = right;
/*@ set oldleft = left;
```

Proof obligations in Simplify

The screenshot shows the Eclipse IDE with the following components:

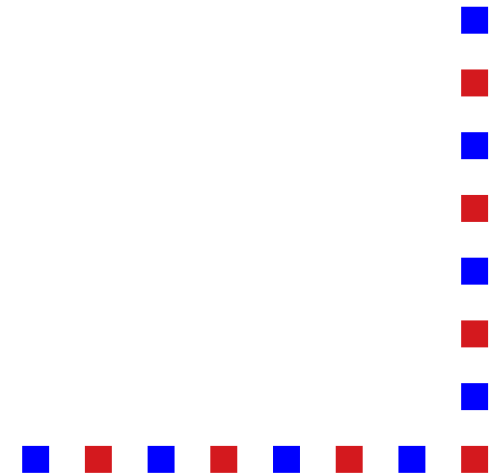
- Case Explorer:** A tree view on the left showing a project structure with folders for 'Case 1' through 'Case 14' and 'Goal 1' through 'Goal 2'.
- Case Viewer:** The main editor window displays the source code for 'QuickSort.java'. It includes JML annotations such as `@requires`, `@modifies`, `@ensures`, `@loop_invariant`, and `@decreases`. The code defines a `sort(int lo, int hi)` method.
- Jack Proof View:** A panel at the bottom left shows a list of proof obligations. The first obligation is `(NEQ (select |f_tab| |this|) |null|)`. The last obligation is circled in red.
- Progress View:** A panel at the bottom right shows the current proof obligation being processed: `(FORALL (|l_j0| |l_j10|) (IMPLIES (AND (<= |l_lo| |l_j0|) (<= |l_lo| |l_hi|)) (IMPLIES (AND (<= |l_lo| |l_j10|)`. A red arrow points to the `(FORALL` prefix with the text "prefix syntax".

Annotation generation



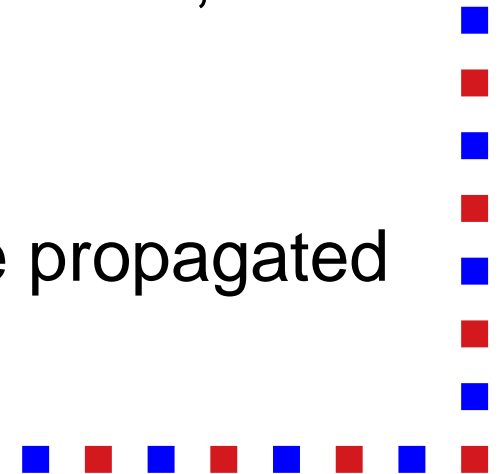
The cost of writing annotations

- Annotation writing labour-intensive and error-prone
- Much time spend on specifying obvious properties
- Annotations for a simple security property often scattered through the code
- For static verification, method specifications need to be relatively complete



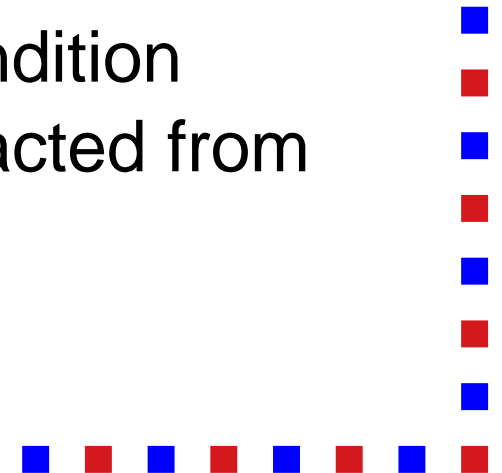
Runtime checking vs. static verification

- Method m has specification: requires P ; ensures Q
- Method use calls method m
- **Runtime checking**: at all calls to m the specification is tested
- **Static checking**: if use does not establish P , it needs to be propagated
Specification for use : requires P
- If use does not invalidate Q , it can be propagated

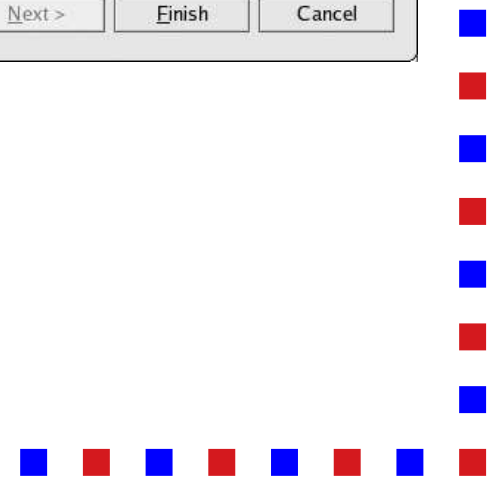
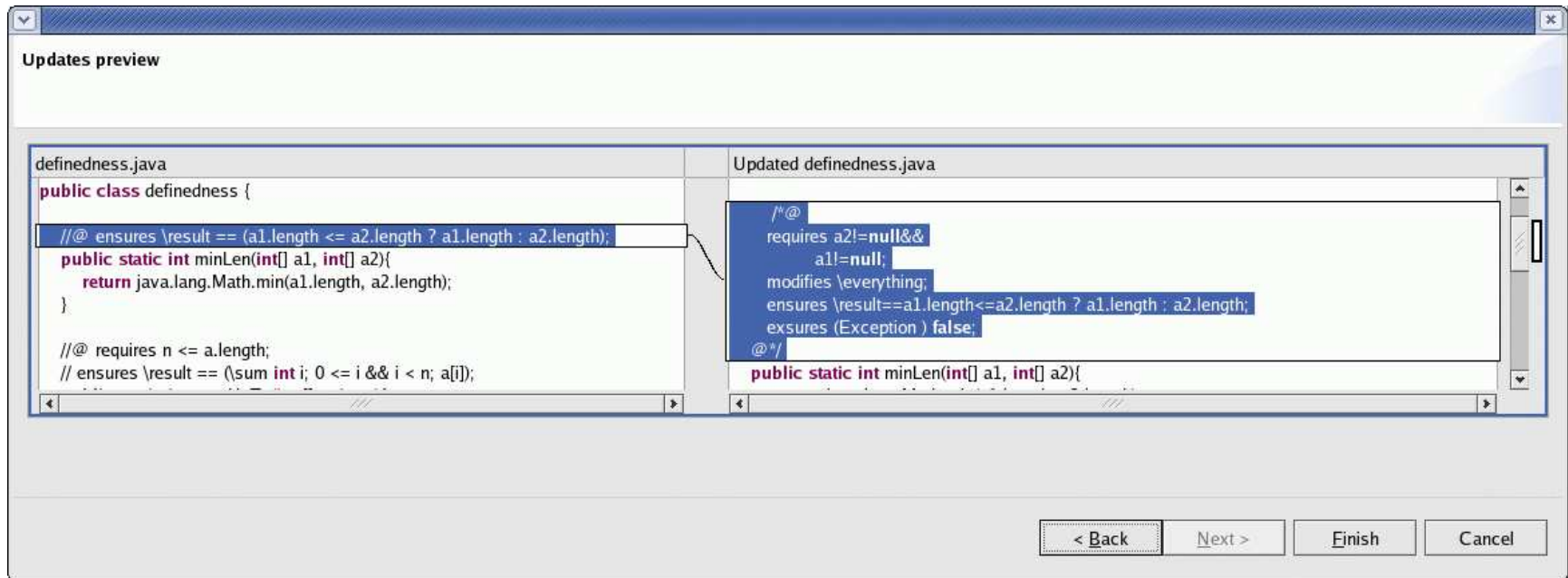


Annotation generation in JACK

- Precondition generation to avoid nullpointer exceptions and array index out of bound exceptions
- Assignable clause generation
- Annotation generation to capture security properties, with annotation propagation
- Implementation uses weakest precondition implementation: annotations are extracted from generated verification conditions



Generation of preconditions and assignable clauses



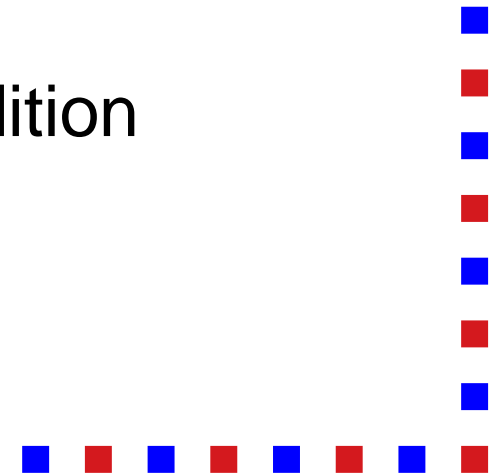
Annotation generation for security properties

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

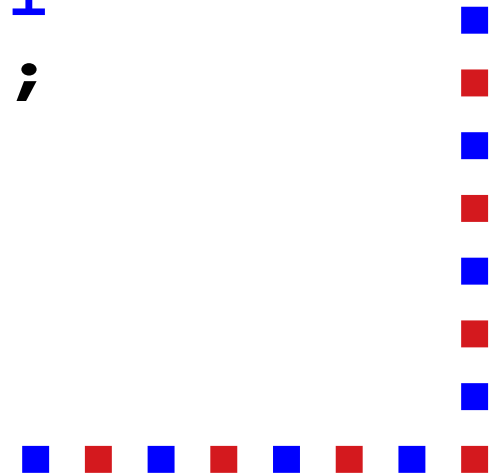
```
/*@ requires TRANSACT == 0;  
   @ assignable TRANSACT;  
   @ ensures TRANSACT == 1; @*/
```

```
public static native  
    void beginTransaction()  
        throws TransactionException;
```

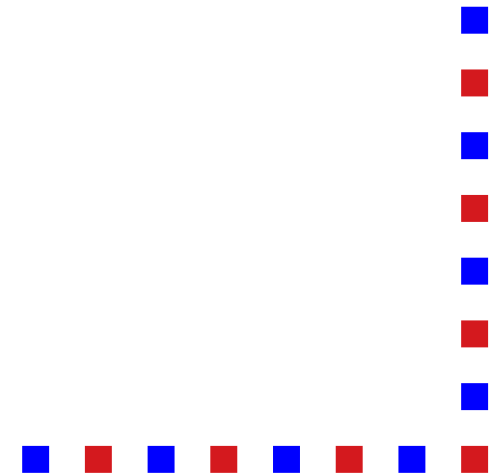
Similar annotations for `commitTransaction`,
`abortTransaction`

Preconditions for methods

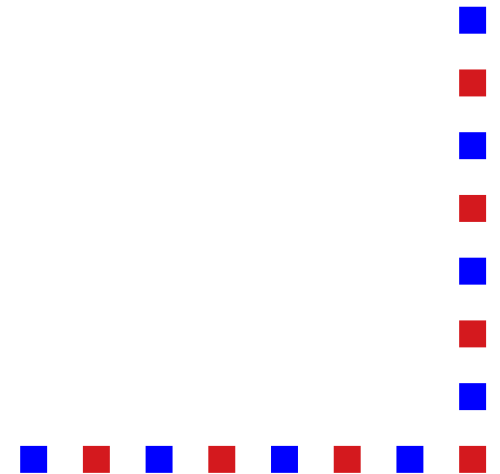
```
public void m() {  
    ...  
    // will require TRANSACT == 0  
    JCSsystem.beginTransaction();  
    // TRANSACT modified  
    // ensures TRANSACT == 1  
    ...  
    // will require TRANSACT == 1  
    JSSystem.commitTransaction();  
    // TRANSACT modified  
    // ensures TRANSACT == 0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



- 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

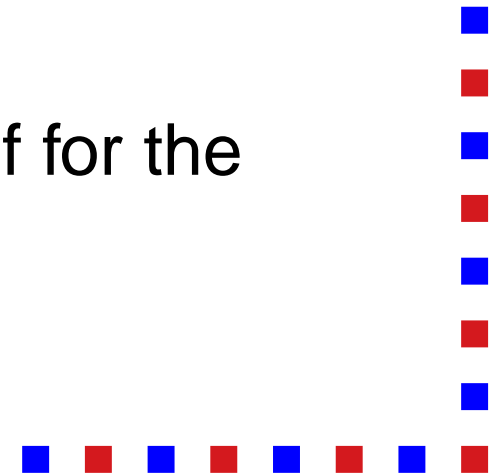


Support for bytecode



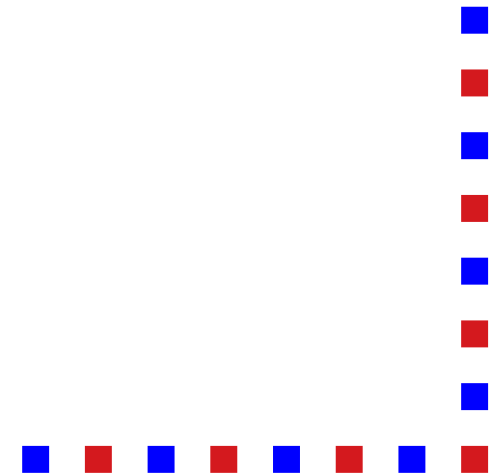
Proof carrying code

- Code producer
 - develops application and builds evidence for its correctness
 - ships application and evidence
- Code client
 - generates verification conditions for the application
 - checks that the evidence is a proof for the verification conditions

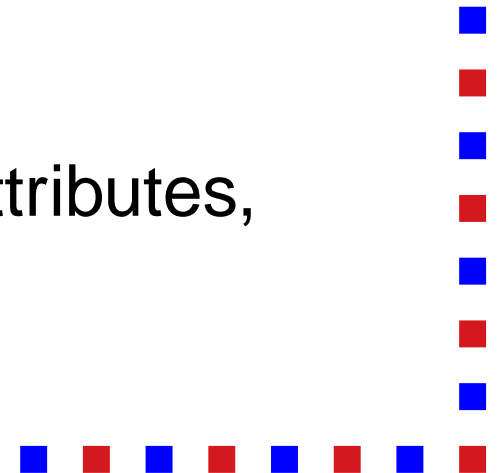


A framework for verification of bytecode

- Bytecode Modeling Language (BML)
- Compiler from JML to BML
- Verification condition generator
- Equivalence with source code verification



- Follows closely the syntax and semantics of JML
- Expression language extended with bytecode specific constructs (constant pool indexes, local variables, stack counter, stack expressions)
- Structural and type constraints, à la BCV
- Encoding in class file format
 - Java compiler independent
 - JVM compatibility: user-specific attributes, indexing to relevant program point
 - Efficiency of JVM not affected



Generated class file

The screenshot shows a 'Bytecode viewer' window for the file `/home/mhuisman/Research/SourceSpecVer/Talks/FMCO/QuickSort.class`. The left sidebar displays a tree view of the class structure. Two sections are circled in red: the 'Methods' section, specifically the '[0] Code' folder containing '[1] MethodSpecification', '[2] Assert', '[3] LoopSpecification', and '[4] BlockSpecification'; and the 'Attributes' section, containing '[0] SourceFile', '[1] ModelField', '[2] Model_Method', '[3] ClassInvariant', and '[4] Constraint'. The main pane shows 'Generic info' (Attribute name index: `cp_info #9`, Attribute length: `278`) and 'Specific info' with tabs for 'Bytecode', 'Exception table', and 'Misc'. The 'Bytecode' tab is active, displaying a list of instructions from index 0 to 40, including `iload_1`, `iload_2`, `if_icmplt 6 (+4)`, `return`, `istore_3`, `aload_0`, `getfield #14 <inherit/QuickSort.tab>`, `goto 82 (+63)`, `goto 28 (+6)`, `inc 3 by 1`, `istore 5`, `goto 51 (+20)`, and `aload_0`. A 'Copy to clipboard' button is located at the bottom right of the main pane.

Method specification in BML

Bytecode viewer

File Classpath Browse Window Help

/home/mhuisman/Research/SourceSpecVer/Talks/FMCO/QuickSort.class

General Information

Constant Pool

Interfaces

Fields

Methods

- [0] <init>
- [1] sort
 - [0] Code
 - [1] MethodSpecification
 - [2] Assert
 - [3] LoopSpecification
 - [4] BlockSpecification
- [2] sort
 - [0] Code
 - [1] MethodSpecification
 - [2] Assert
 - [3] LoopSpecification
 - [4] BlockSpecification
- [3] swap
- [4] withinBounds

Attributes

Generic info:

Attribute name index: [cp_info #40](#)

Attribute length: 0

Specific info:

Nr.	
0	requires this.#14 != null && 0 <= local(...)
1	{
2	requires true
3	modifies #14.this[local(1)..local(2)]
4	ensures (\forall #27#27;local(1) <= b...
5	exsures (#41) false
6	}

Loop specification in BML

The screenshot shows the Bytecode viewer interface for the class `QuickSort.class`. The left pane displays a tree view of the class structure, with the `LoopSpecification` attribute selected under the `sort` method. The right pane shows the BML for this attribute:

```
Generic info:  
Attribute name index: cp_info #43  
Attribute length: 0  
  
Specific info:  
loop: @82: modifies #14.this[local(1)..local(2) - 1],local(4),local(3) invariant l
```

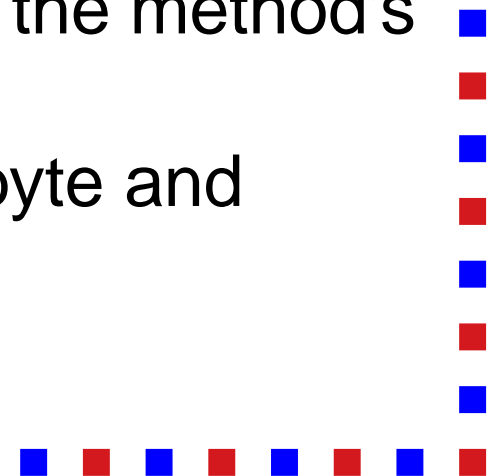
Red annotations highlight the BML components: an arrow points from the tree view to the `loop:` prefix; another arrow points to the `@82` value, labeled "index"; and a third arrow points to the `modifies` keyword. The `invariant l` part of the BML is underlined.

JML to BML compiler

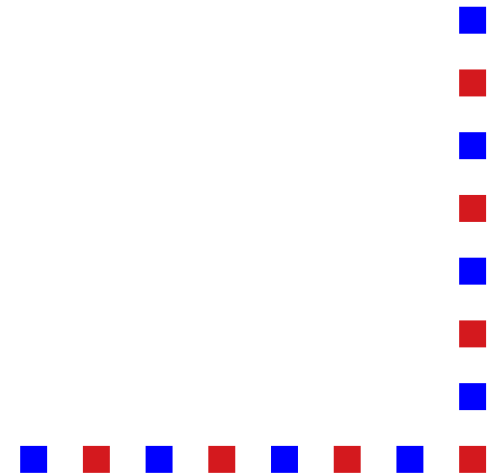
- Input:
 - Source file annotated with JML
 - Corresponding class file, decorated with **Local_Variable_Table** and **Line_Number_Table**
- Steps:
 - Declarations of ghost and model fields
 - Linking
 - Locating indexes for annotation statements
 - Compilation of JML predicates
 - Generation of user-specific class attributes

Relation between proof obligations on source and bytecode

- Verification condition generator proven sound under the hypothesis that the control flow graph is reducible
- For non-optimising compiler equivalence of proof obligations modulo:
 - names - Java names are compiled into indexes of the constant pool or elements in the method's local variable table
 - types - Java types integer, short, byte and boolean are compiled into integers



Support for interactive verification

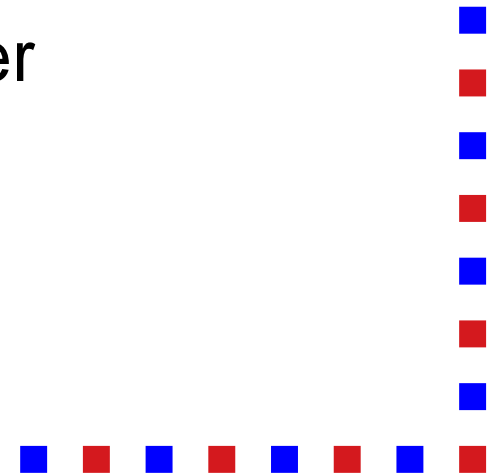


Specification and verification of complex properties

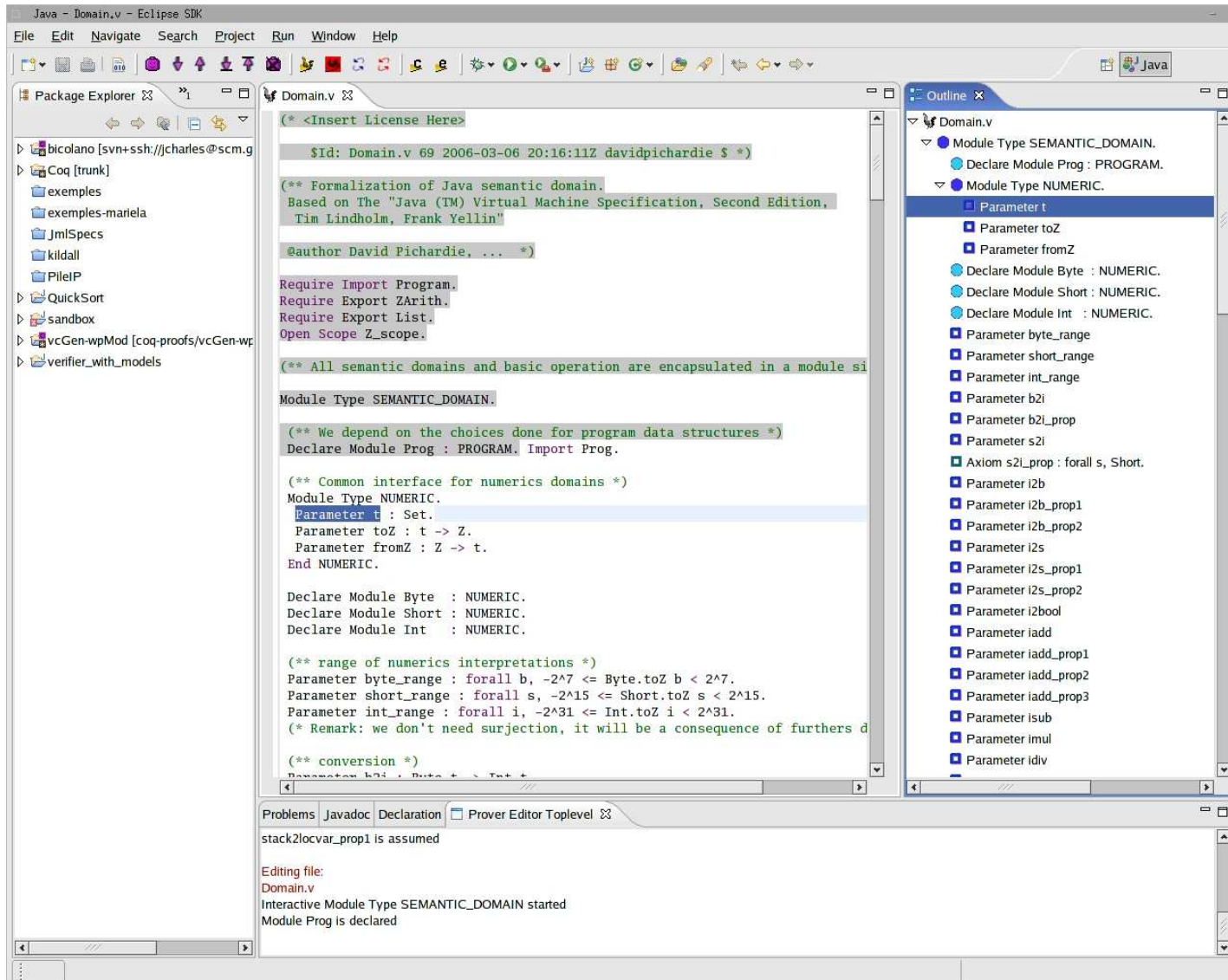
- For complex properties, automatic verification often not sufficient
- Such properties often use advanced specification techniques (JML model features)
- Interactive prover support necessary: [Coq](#)
- Introduction of [native](#) construct to bridge gap between JML models and logic of theorem prover



- **Syntax highlighting** for both Coq file and proof view window
- Same **keyboard shortcuts** as CoqIde
- Full integration within Eclipse
 - No pop-ups, except if user wants to use another editor
 - Management of proof files is easier
- Also usable with ESC/Java
- Handles large files (> 1 Mb)

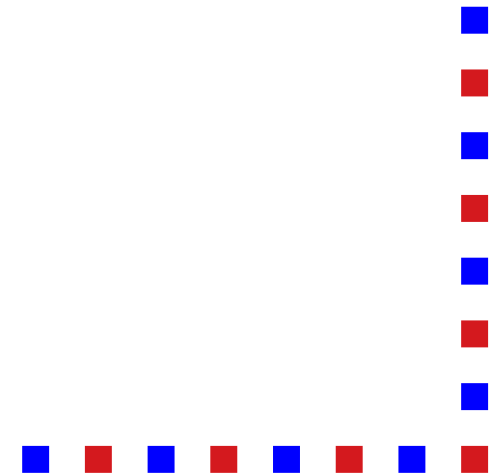


What it looks like



Specifications of complex properties: use of pure methods

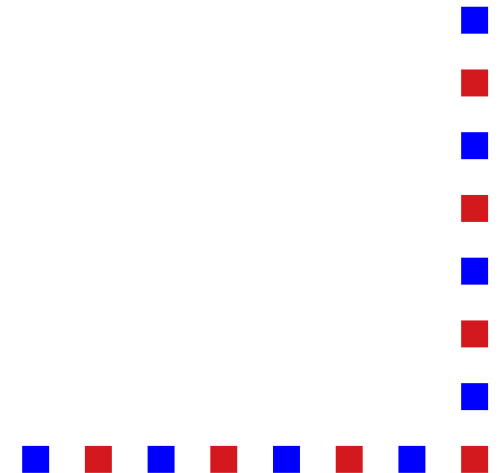
- A method is **pure** when it has no visible side effect
- Pure methods can be used in specifications
- Complicates verification: specification of pure method has to be used
- Our approach: define the pure method in Coq in directly



Specifications of complex properties: use of pure methods

- A method is **pure** when it has no visible side effect
- Pure methods can be used in specifications
- Complicates verification: specification of pure method has to be used
- Our approach: define the pure method in Coq in directly

Native specifications



Native methods

- In JML:

```
//@ public native boolean  
    withinBounds(Object[] tab, int i);
```

- In the Coq file `user_extensions.v`:

Definition `withinBounds` :

`Reference →`

`(Reference → t_int → Reference) →`

`t_int → bool :=`

fun `tab` `intelements` `value =>`

`and (tab != null) (and (0 <= value)`

`(value < (arraylength tab))).`



Native types

- To express complex properties, advanced data types useful
- Easily defined in Coq, not in JML
- **Native types:**
Coq types in JML:

```
//@ public native class ObjectSet;
```

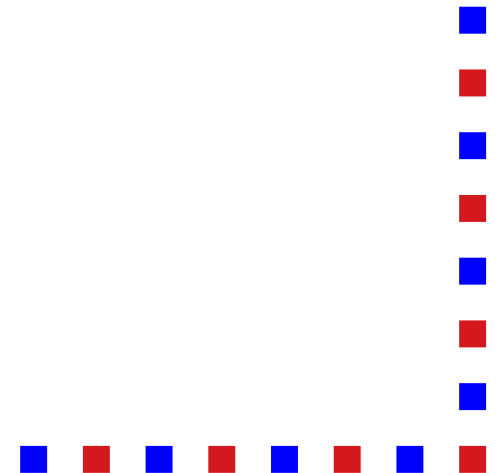
In the Coq file `user_extensions.v`:

```
Definition ObjectSet := set Reference.
```



What are native types?

- Native types are not standard Java/JML class types:
 - Do not inherit from Object
 - No constructors
 - No casts
 - No instance creation
- Native types are **functional type**:
 - Modifiers are 'static'
 - Modifiers create new objects



Example: set library

We can define a Coq set library to use in annotations
In JML we **declare**:

```
/*@ public native class ObjectSet {  
  @ public native static ObjectSet  
      create();  
  @ public native static ObjectSet  
  @      add(ObjectSet os, Object o);  
  @ public native boolean  
      member(Object o);  
  @ public static native ObjectSet  
  @      toSet(Object [] tab);  
  @ }  
@*/
```



Example: set library

In Coq we **define**:

```
Definition ObjectSet := set Reference.
```

```
Definition ObjectSet_create :=  
    empty_set.
```

```
Definition ObjectSet_add  
    (os: ObjectSet)  
    (o: Reference) :=  
    set_add o os.
```

```
Definition ObjectSet_member  
    (this: ObjectSet)  
    (o: Reference) :=  
    set_mem o this
```



To conclude...

- **JACK**: a tool for validating application security and behaviour
- Features:
 - Integration with Eclipse, developer-friendly environment
 - Reduces the burden of annotation writing, by implementing various annotation generation algorithms
 - Support for source code and bytecode verification
 - Support for complex properties, by providing support for interactive verification