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 Imse

The best mobile p ommunity in the world Mobile pho munity

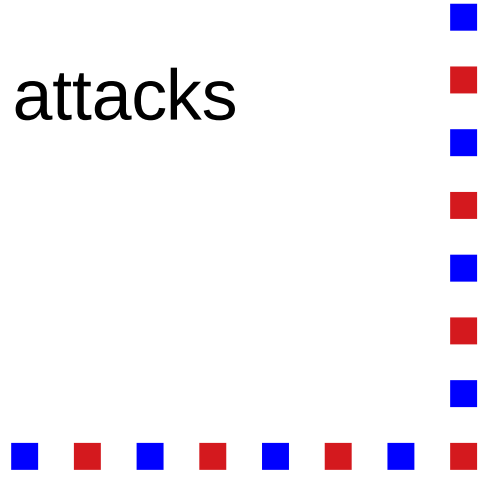
Imserba brings yo phones related news, information or your phones. No matter which p : Nokia, Sony erricson, Siemens la or anything else, here you ca ction of ring-tones, cell phone games, themes, screensavers, backgrounds.

Google on “mobile phone games”

Are you sure that you can trust these applications?

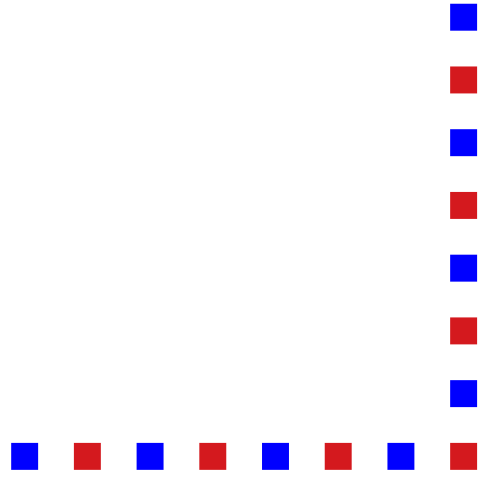
Security for trusted personal devices

- 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



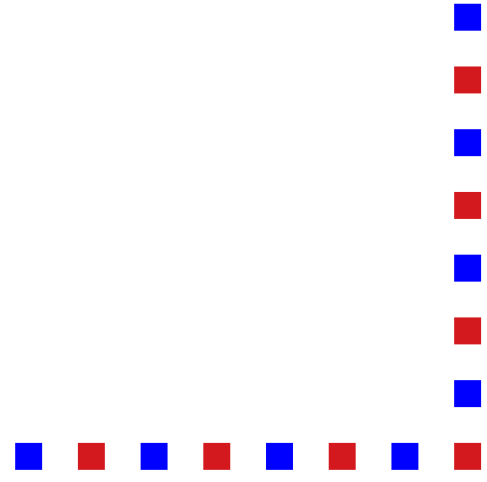
Guaranteeing security

- 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



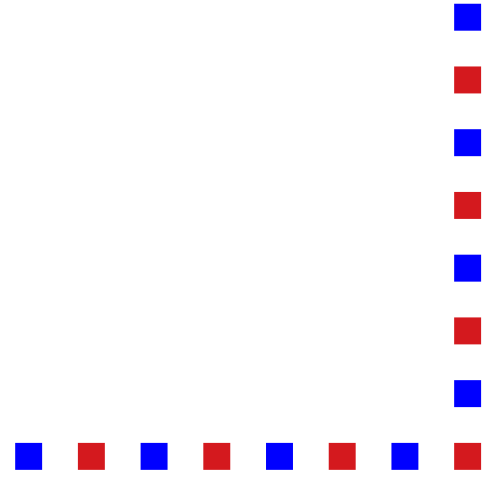
But how to convince developers do this?

- Seamless integration in standard development environment



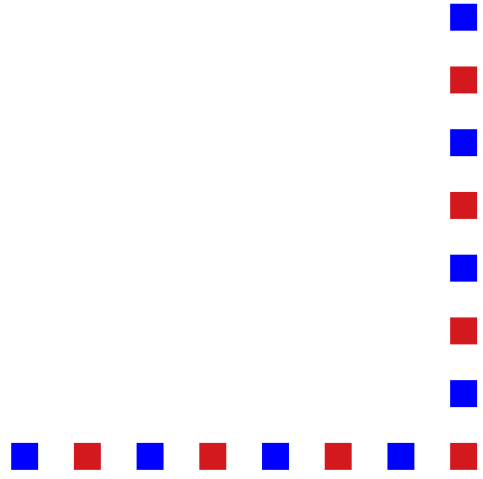
But how to convince developers do this?

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



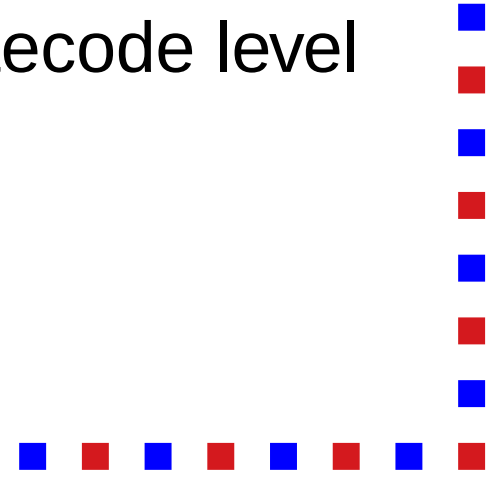
But how to convince developers do this?

- 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?

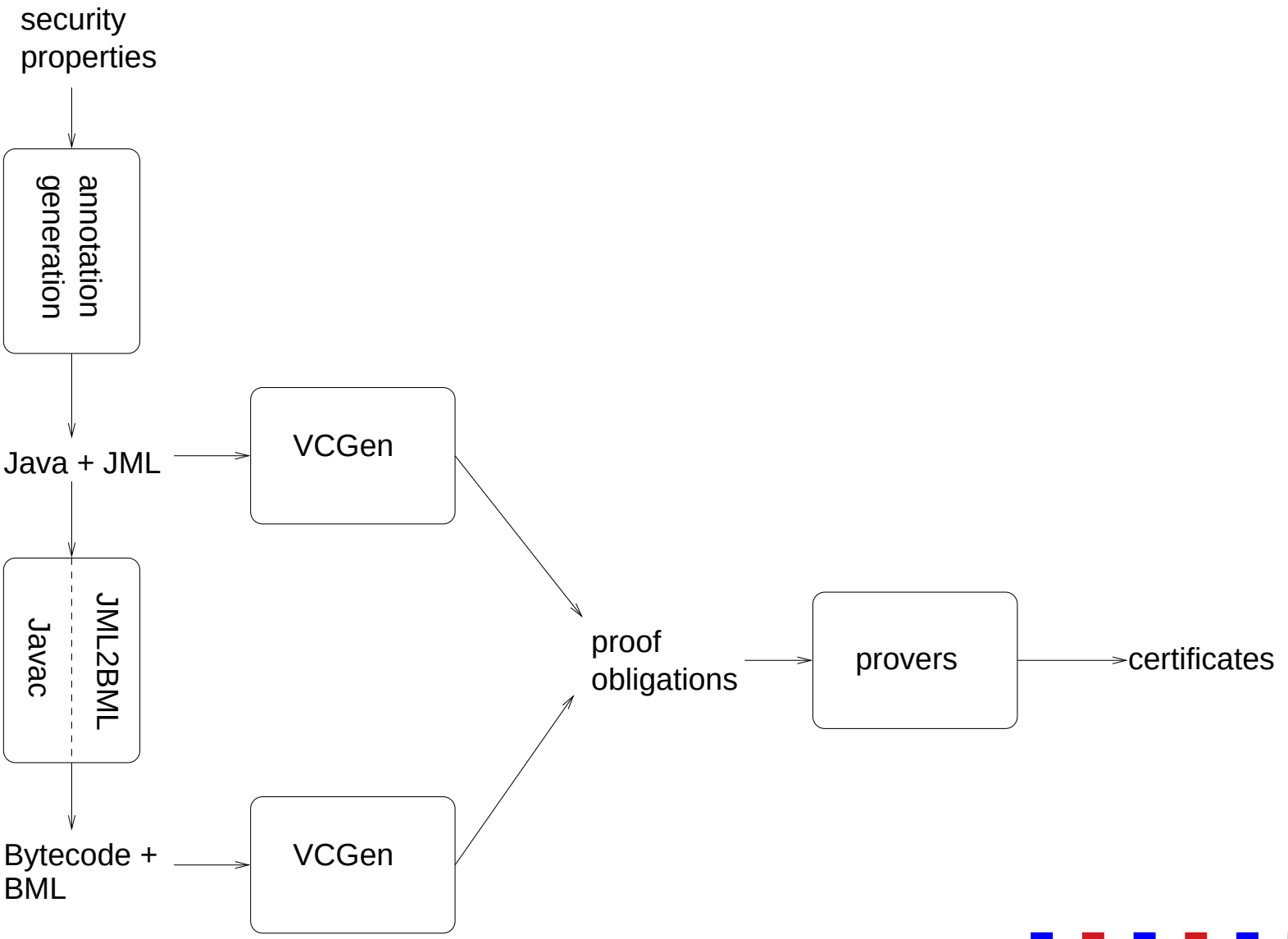
- Seamless integration in standard development environment
- Small overhead in specification writing: annotation generation
- Verification conditions automatically generated, proven by automatic theorem prover
- Reasoning at source code *and* at bytecode level



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

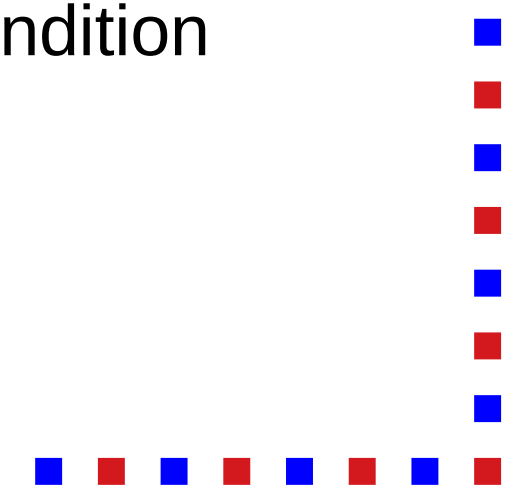


History of JACK

- 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
- Transfered to INRIA (September 2003)
  - Correctness stays major concern
  - More features & plugins

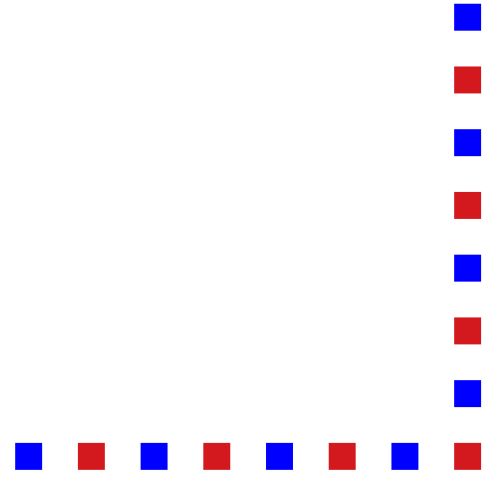
Features of JACK

- 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

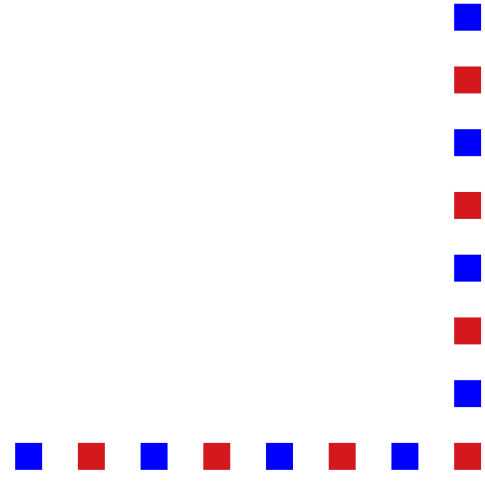


And more features of JACK

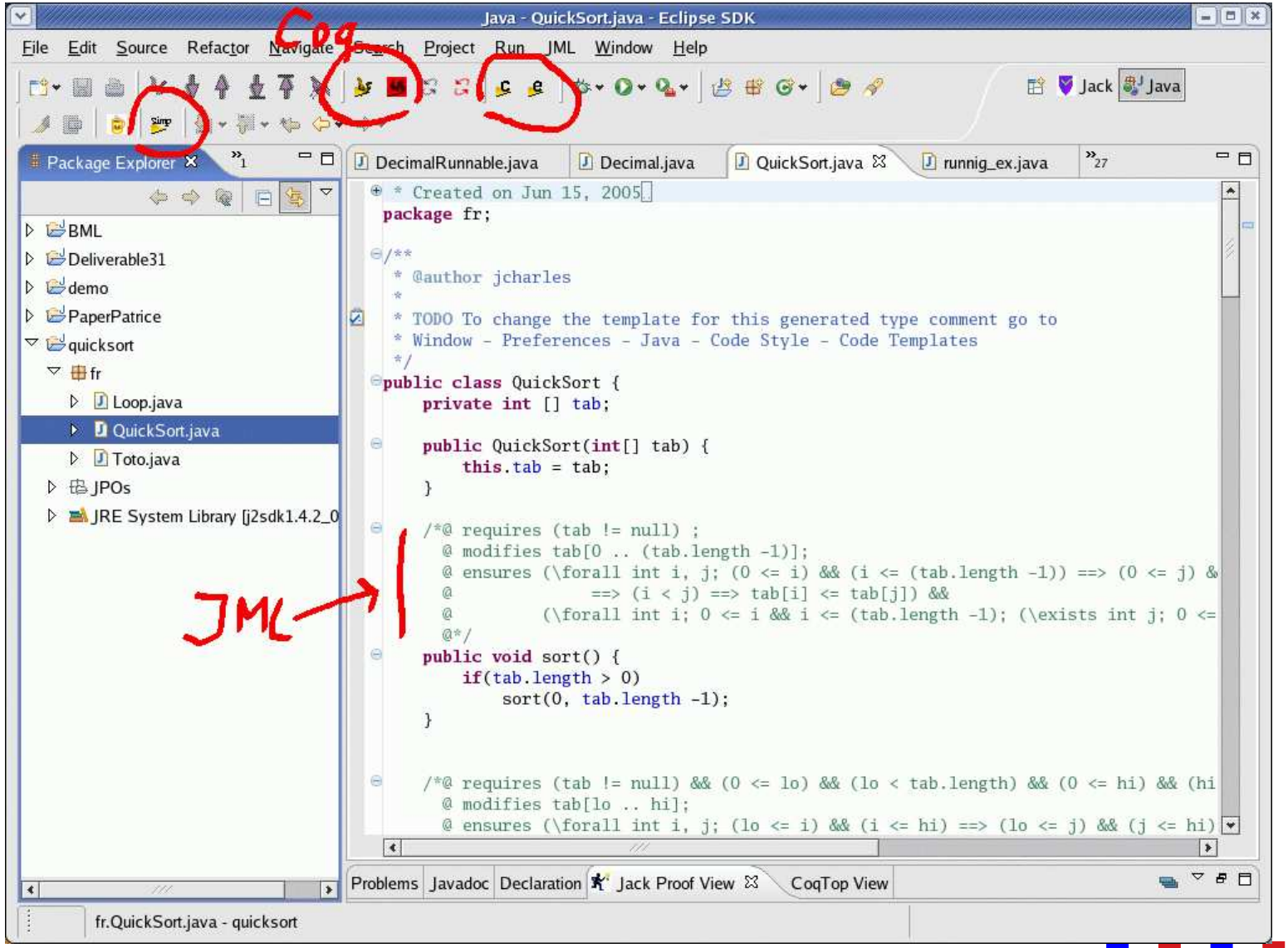
- Generation and propagation of annotations, based on implementation of verification condition generator
- JML specifications compiled into BML (Bytecode Modeling Language)
- Support for verification of bytecode



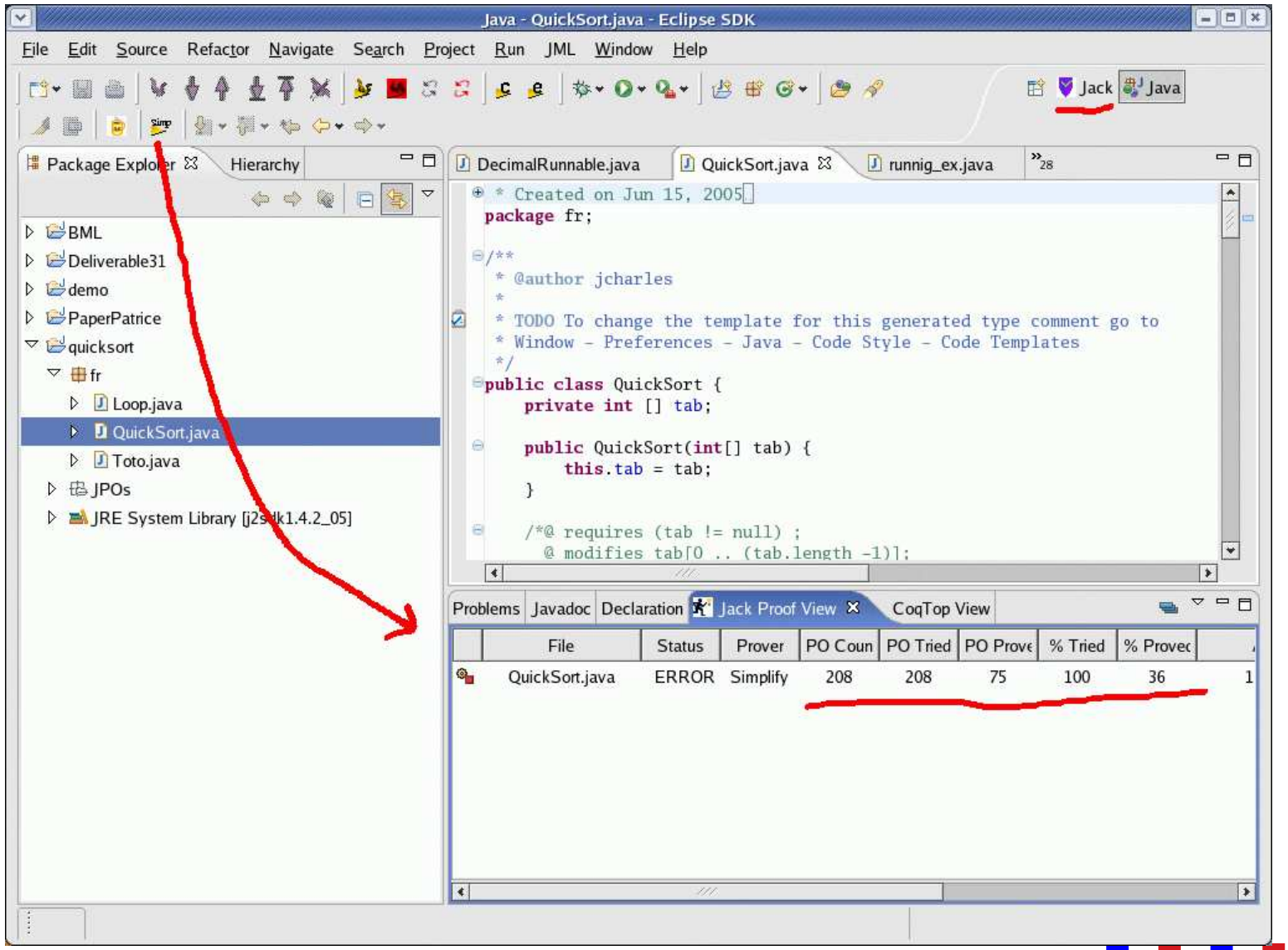
Integration with Eclipse



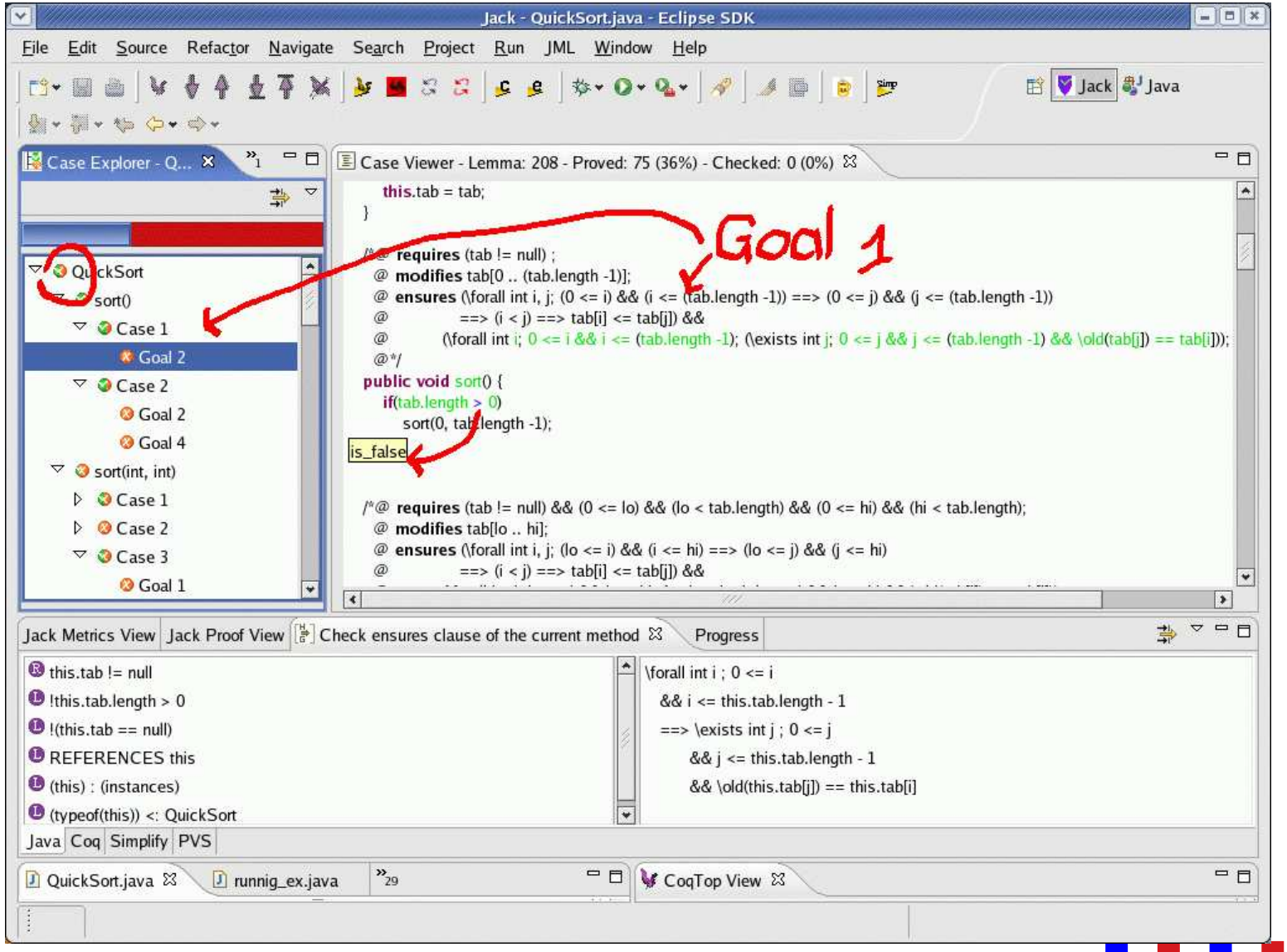
Developing an application in Eclipse



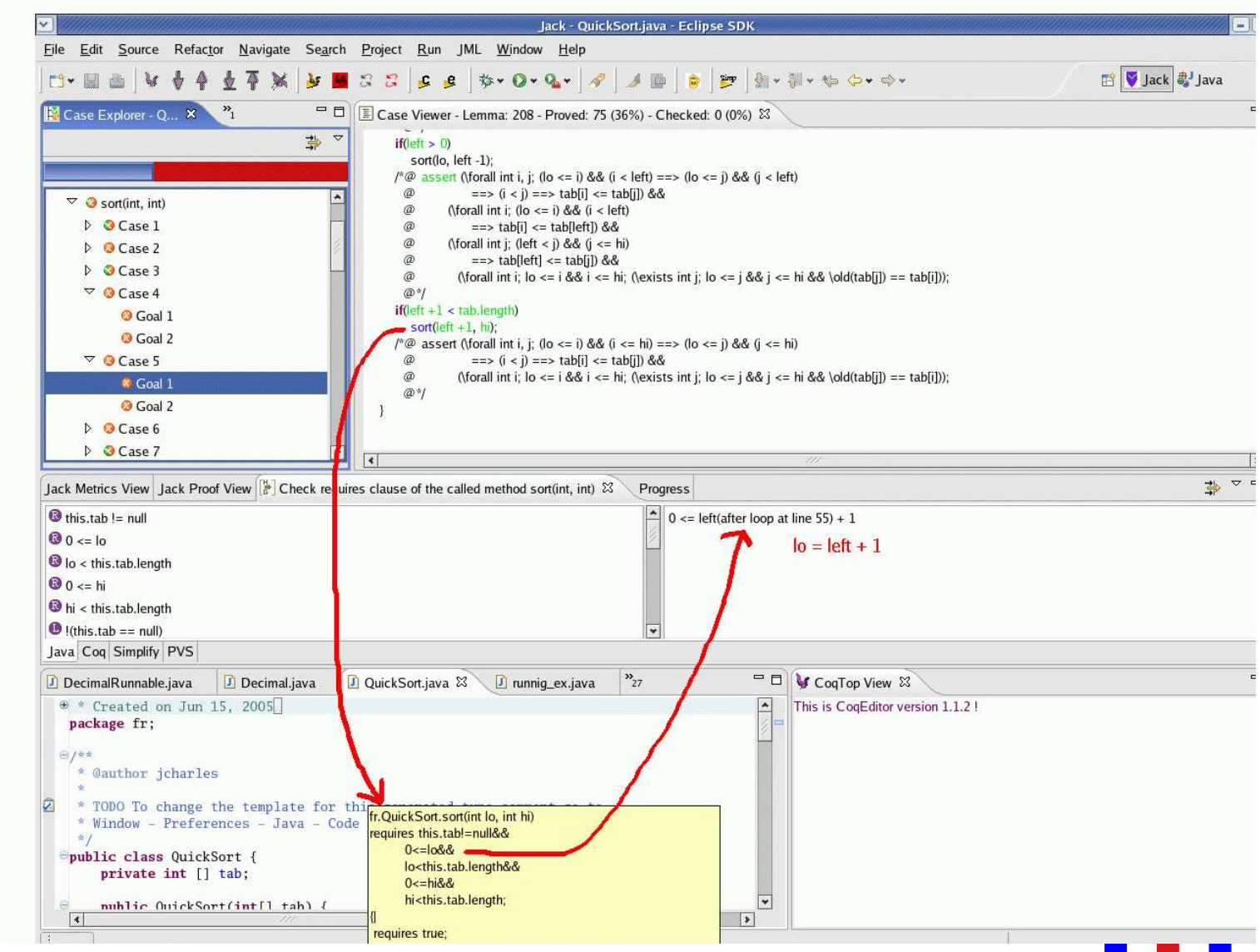
Using Simplify



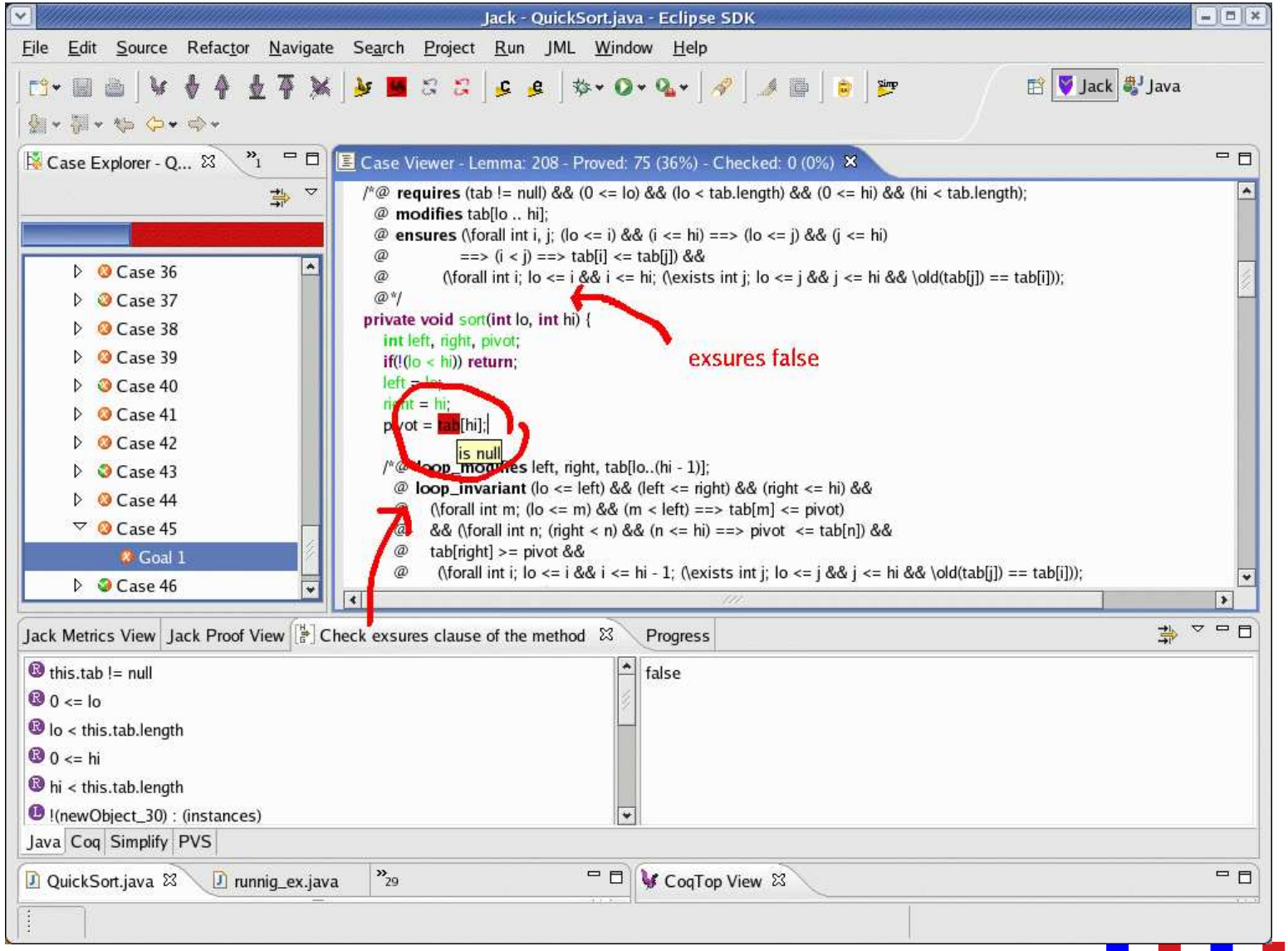
Proof obligation viewer



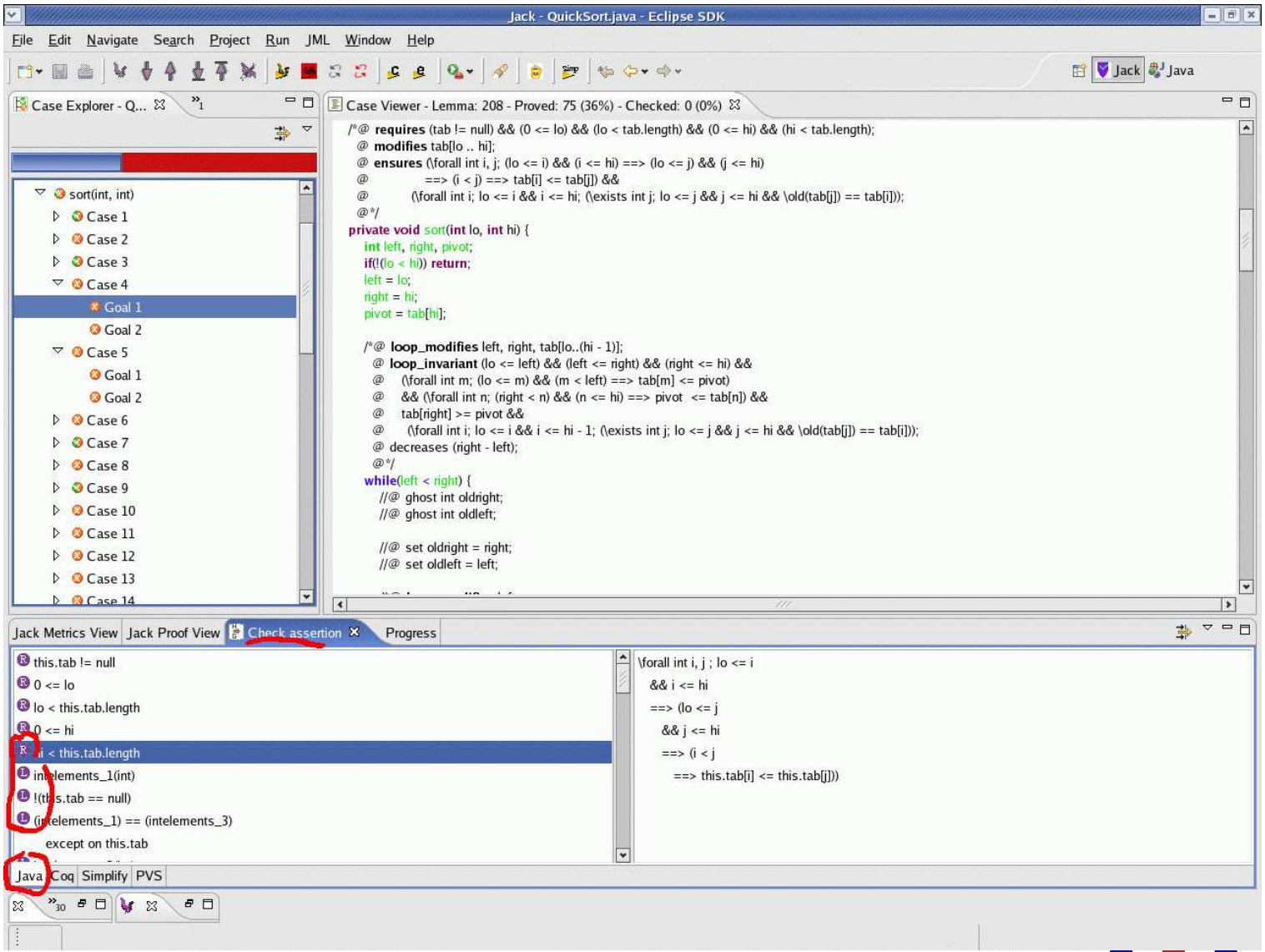
Reasoning with method calls



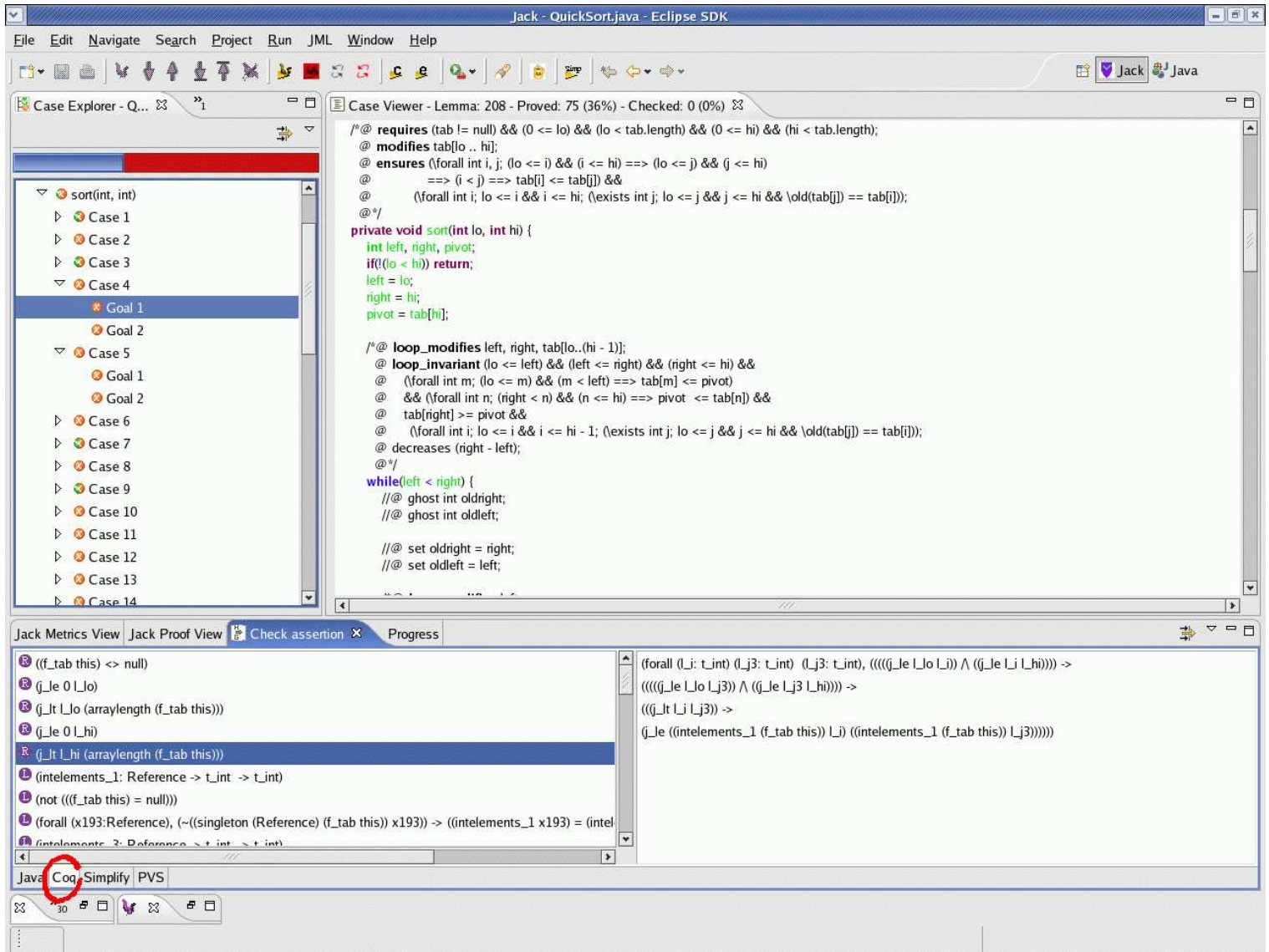
Reasoning about exceptions



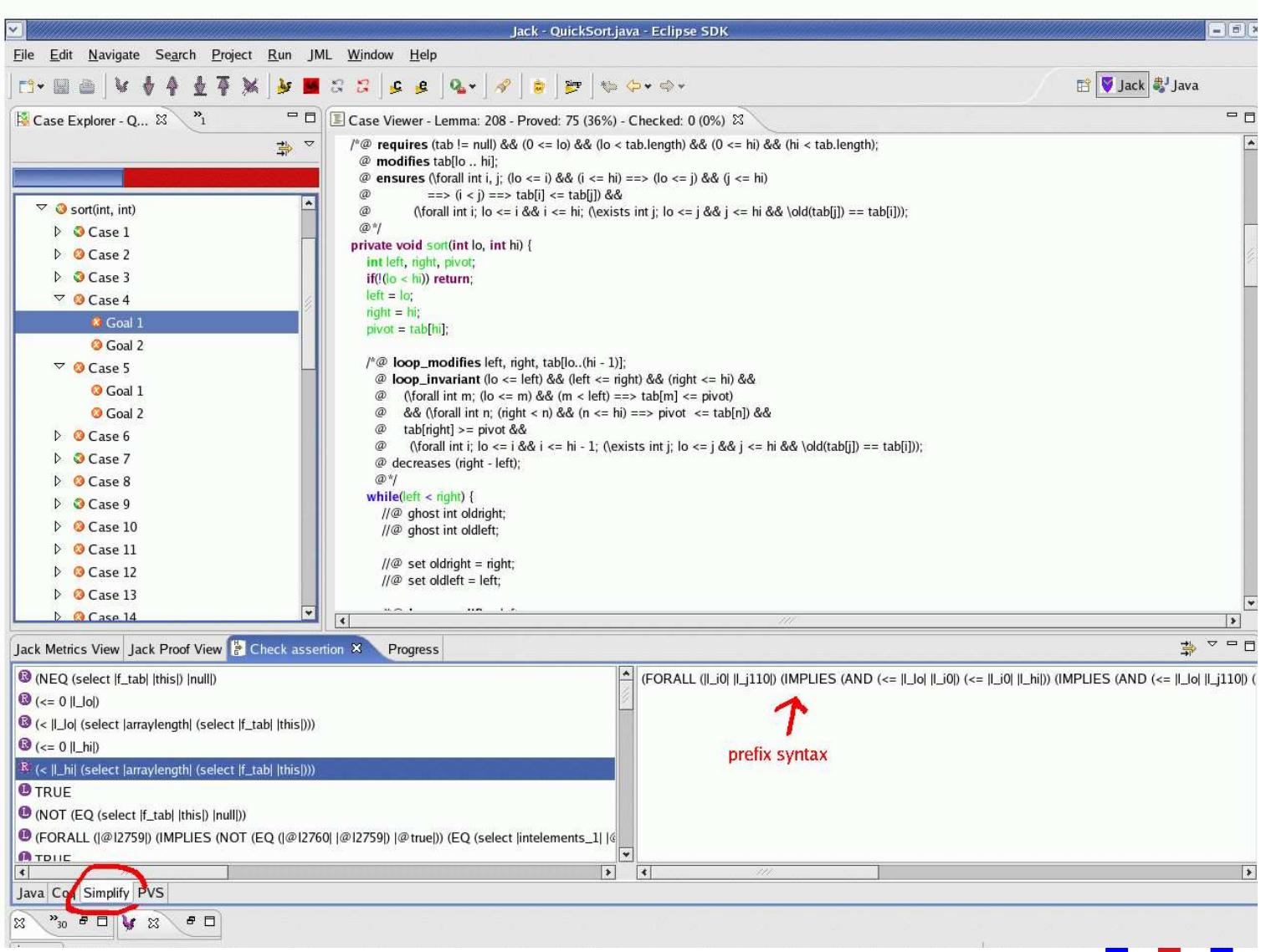
Proof obligations



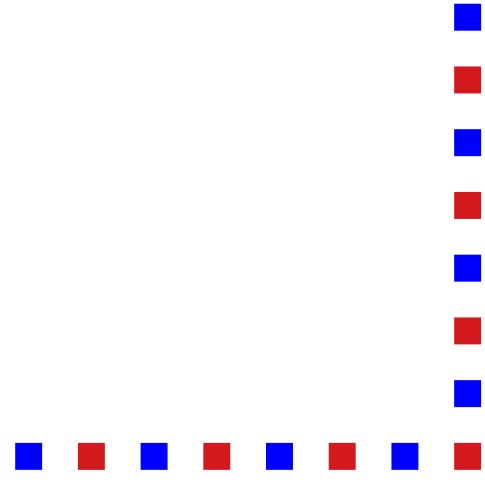
Proof obligations in Coq



Proof obligations in Simplify

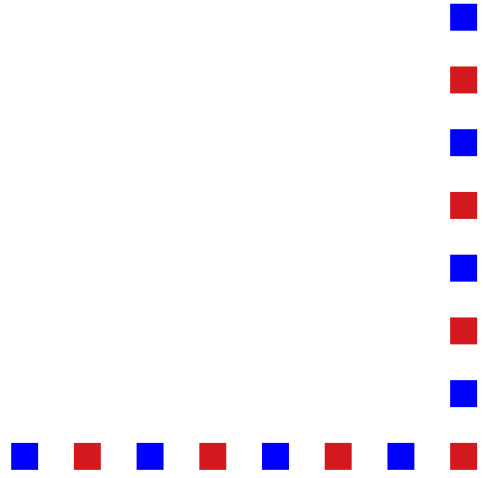


Annotation generation



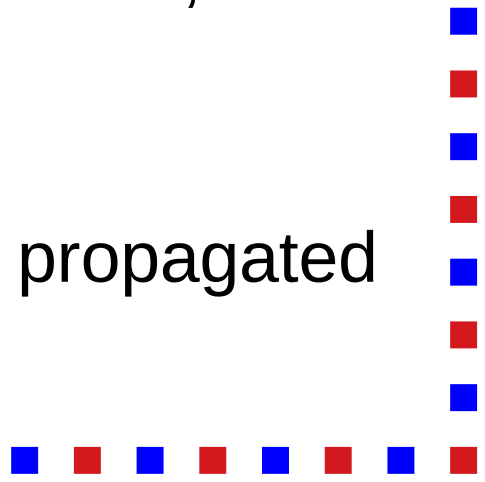
The cost of writing annotations

- 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



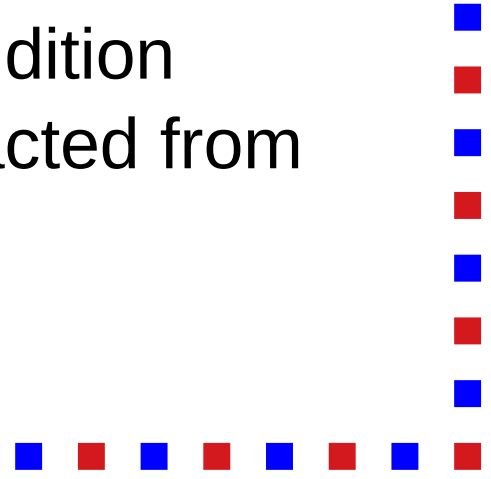
Runtime checking vs. static verification

- 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

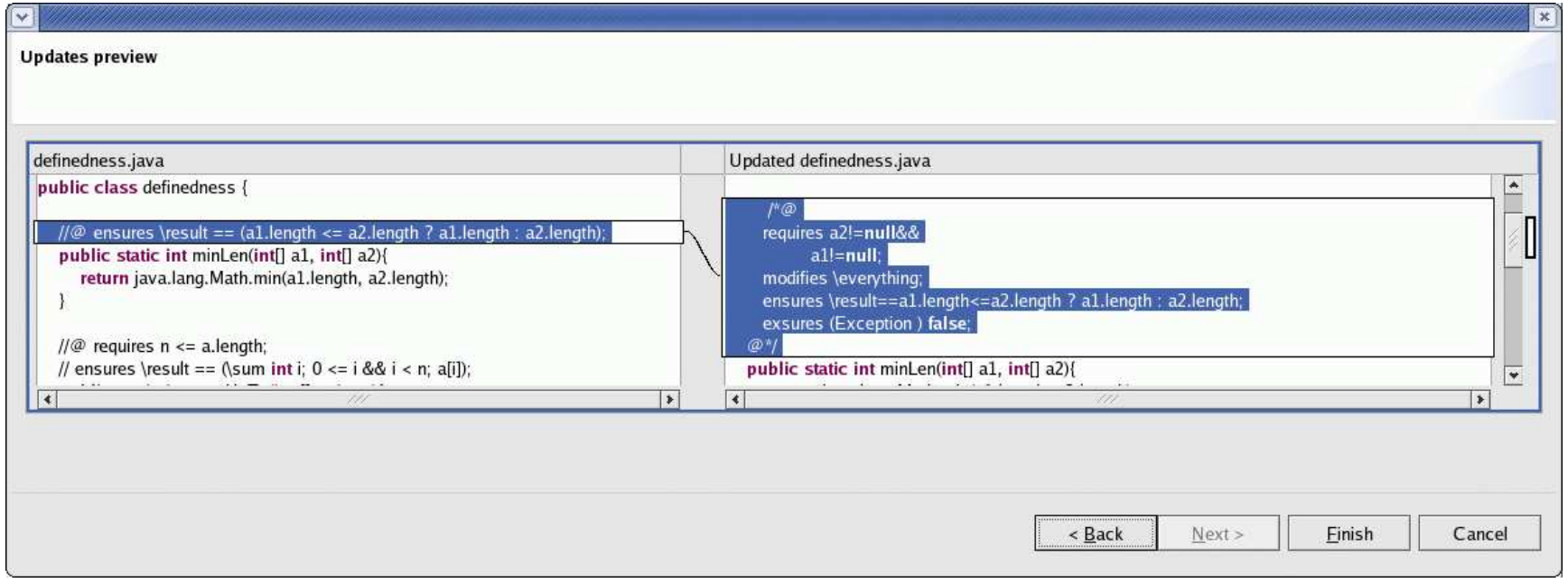


Annotation generation in JACK

- 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



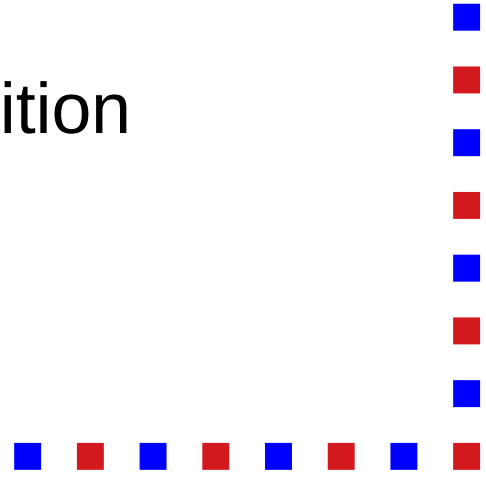
Annotation generation for security properties

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; @*/
```

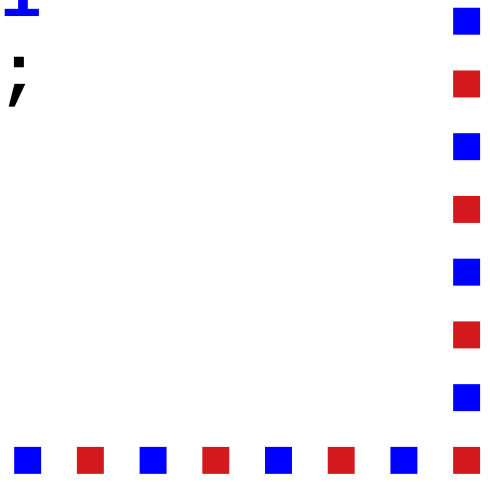
Method **beginTransaction**

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

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

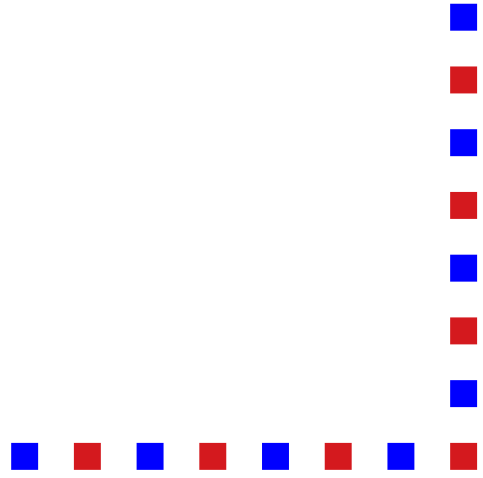
Preconditions for methods

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

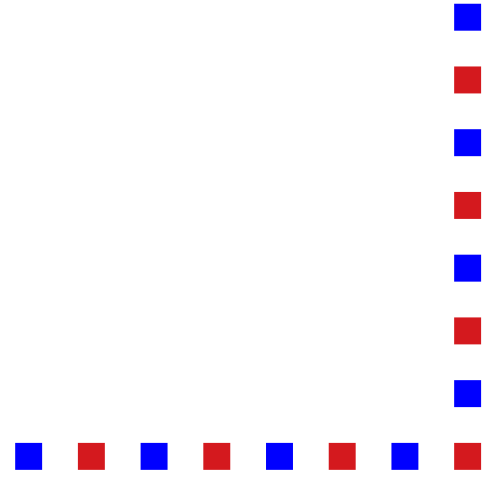


Results

- 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

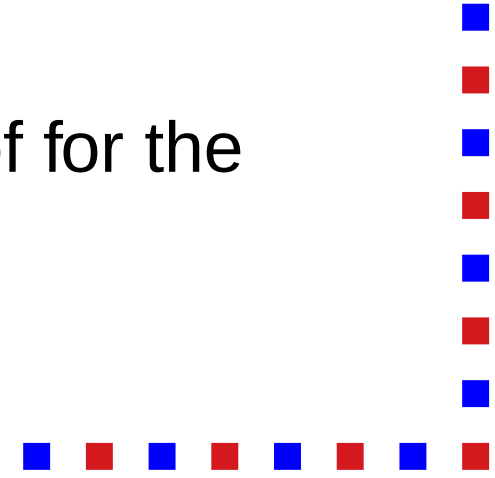


Support for bytecode



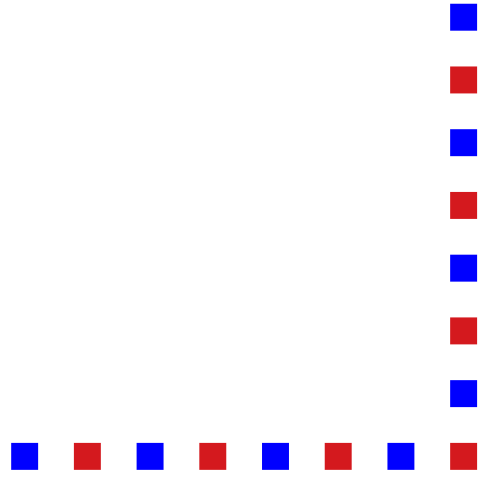
Proof carrying code

- 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



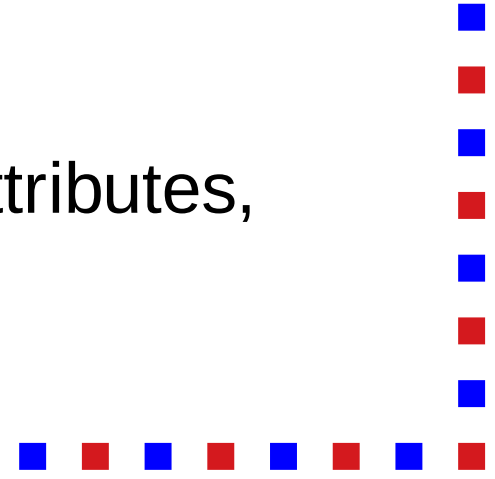
A framework for verification of bytecode

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

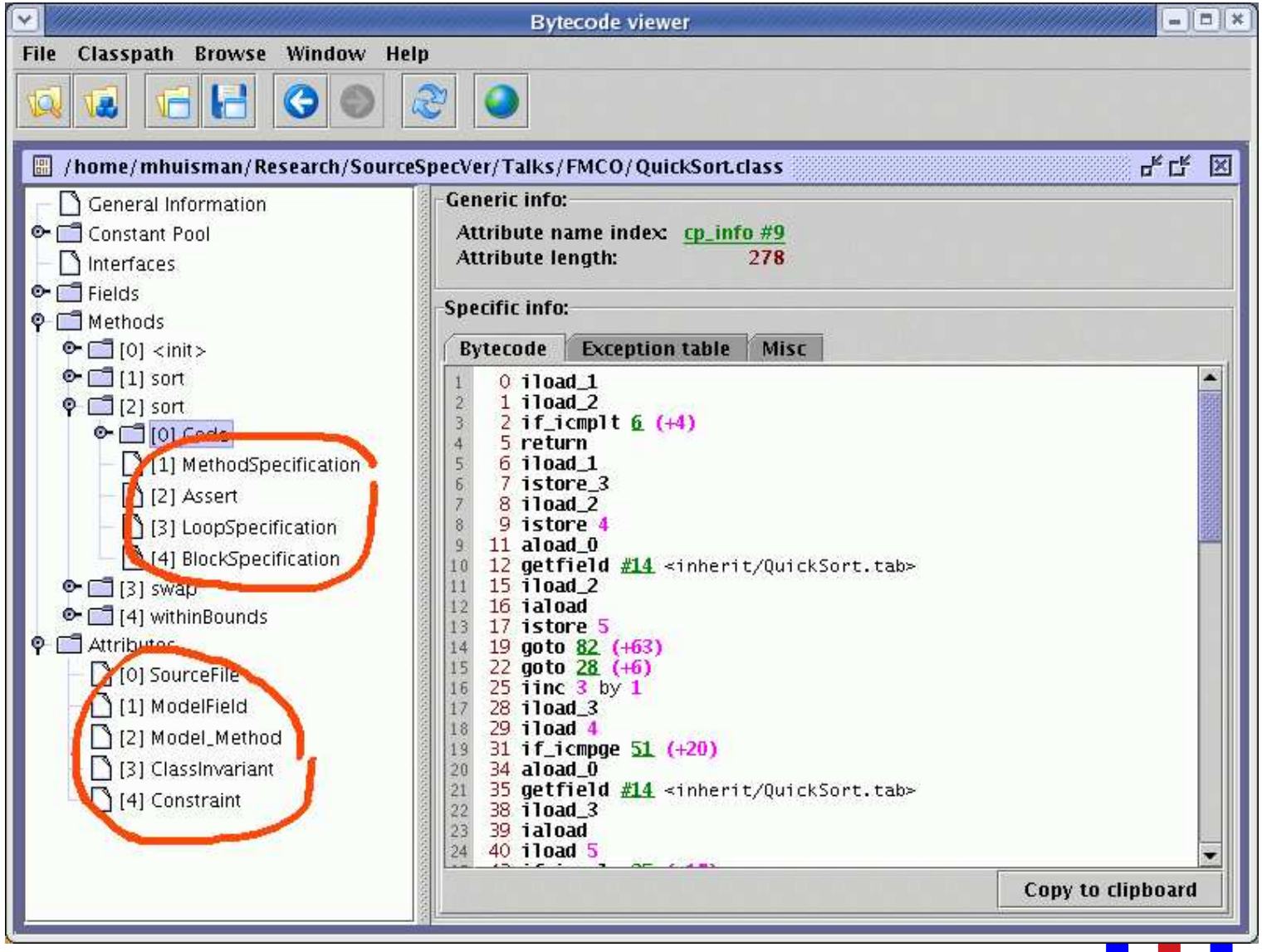


BML

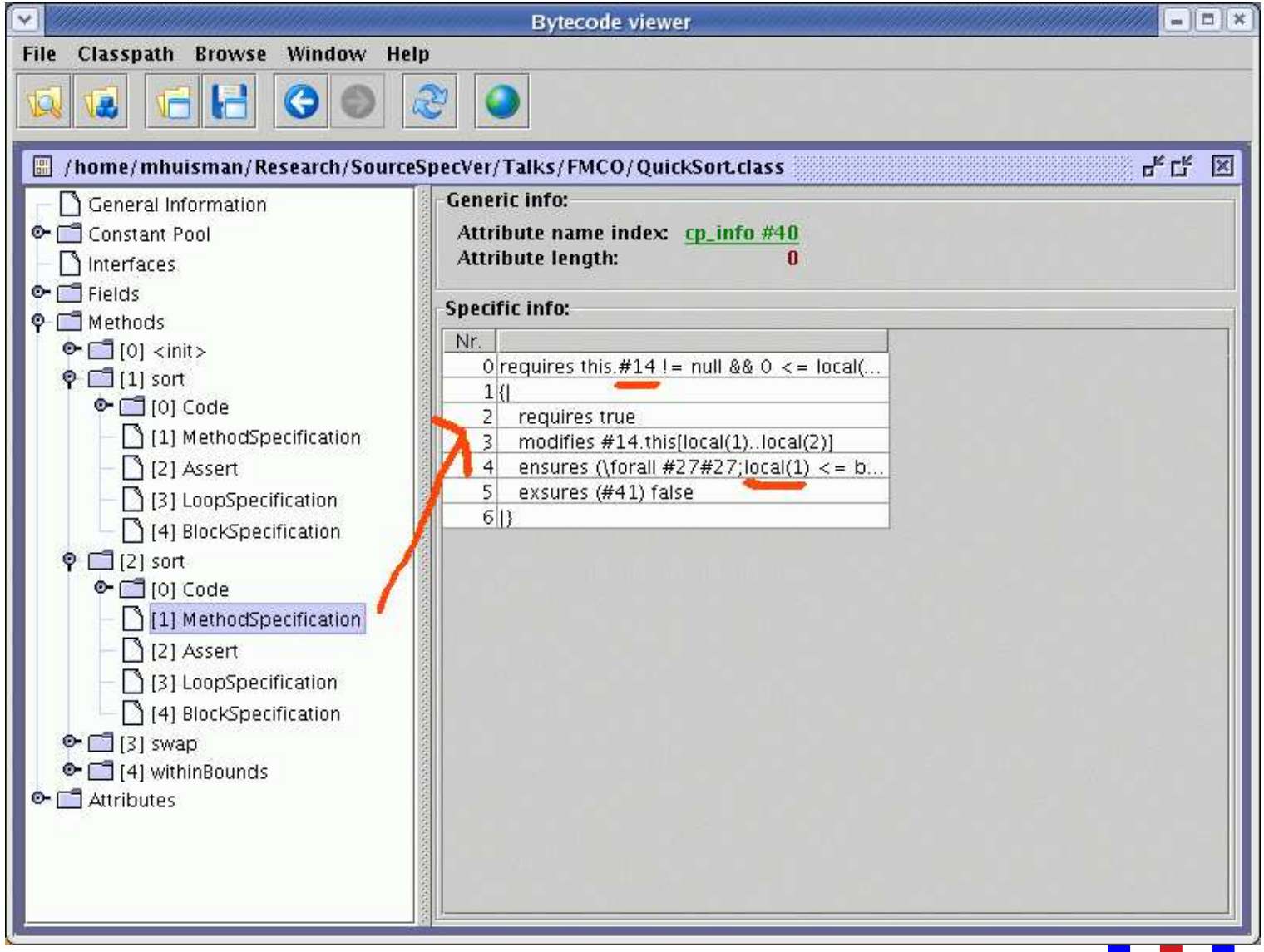
- Follows closely the syntax and semantics of JML
- Expression language extended with bytecode specific constructs (constant pool indexes, local variables, stack counter, stack expressions)
- Structural and type constraints, à la BCV
- Encoding in class file format
  - Java compiler independent
  - JVM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



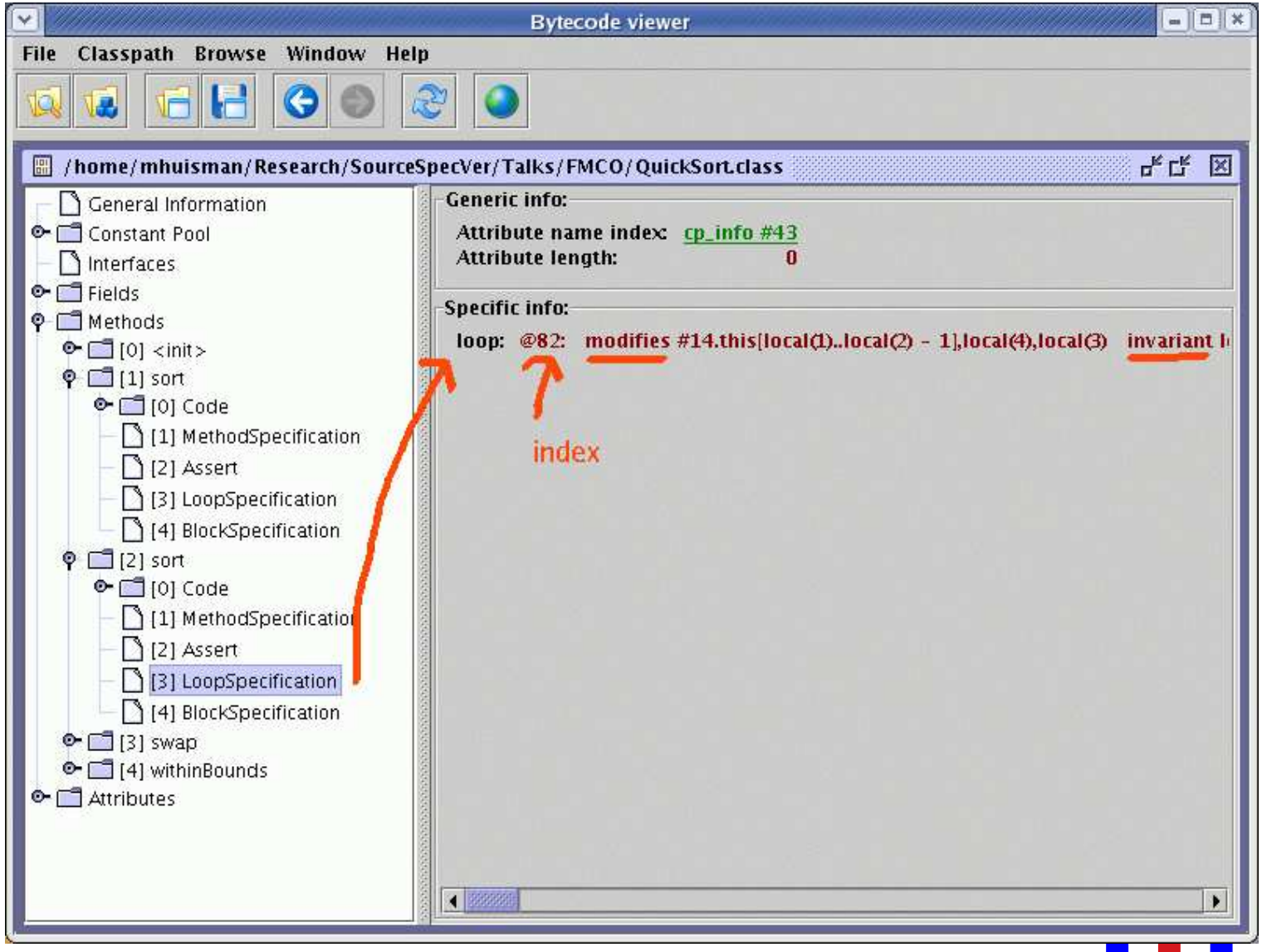
Generated class file



Method specification in BML

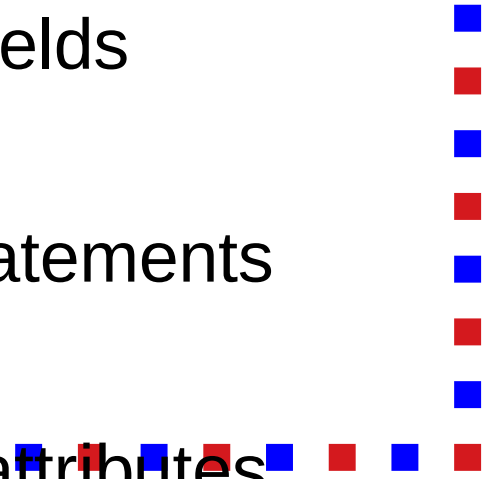


Loop specification in BML



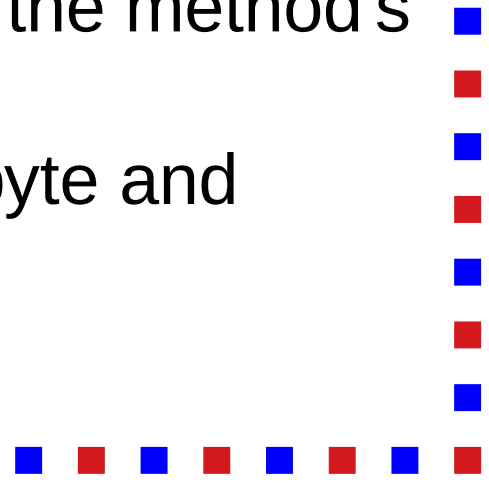
JML to BML compiler

- 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

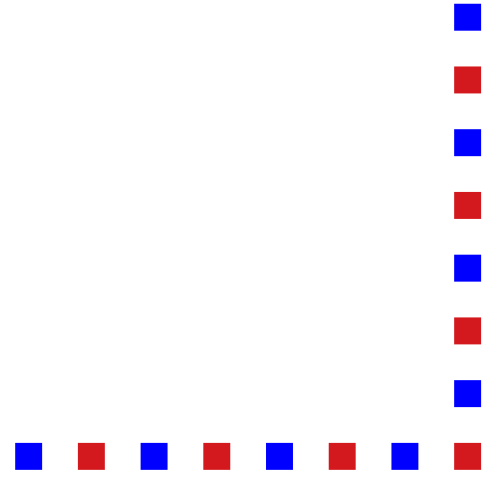


Relation between proof obligations on source and bytecode

- 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

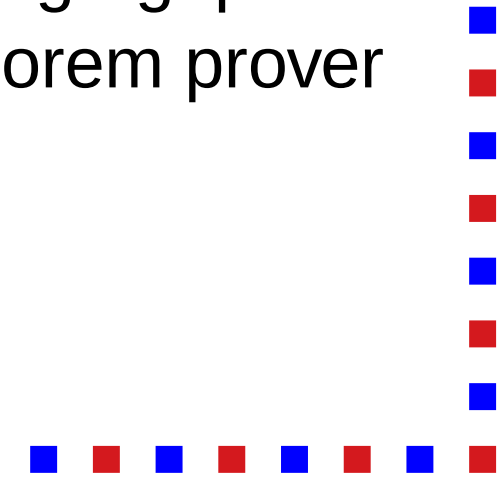


Support for interactive verification



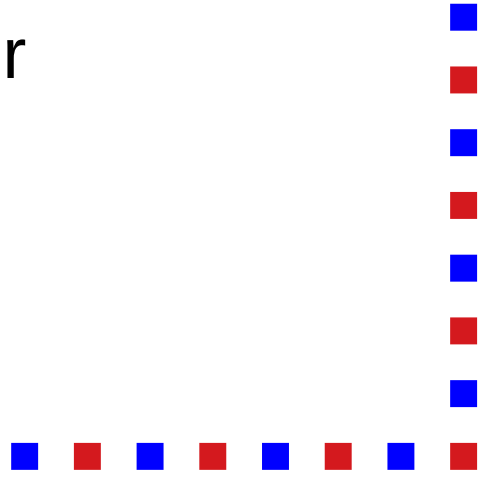
Specification and verification of complex properties

- 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

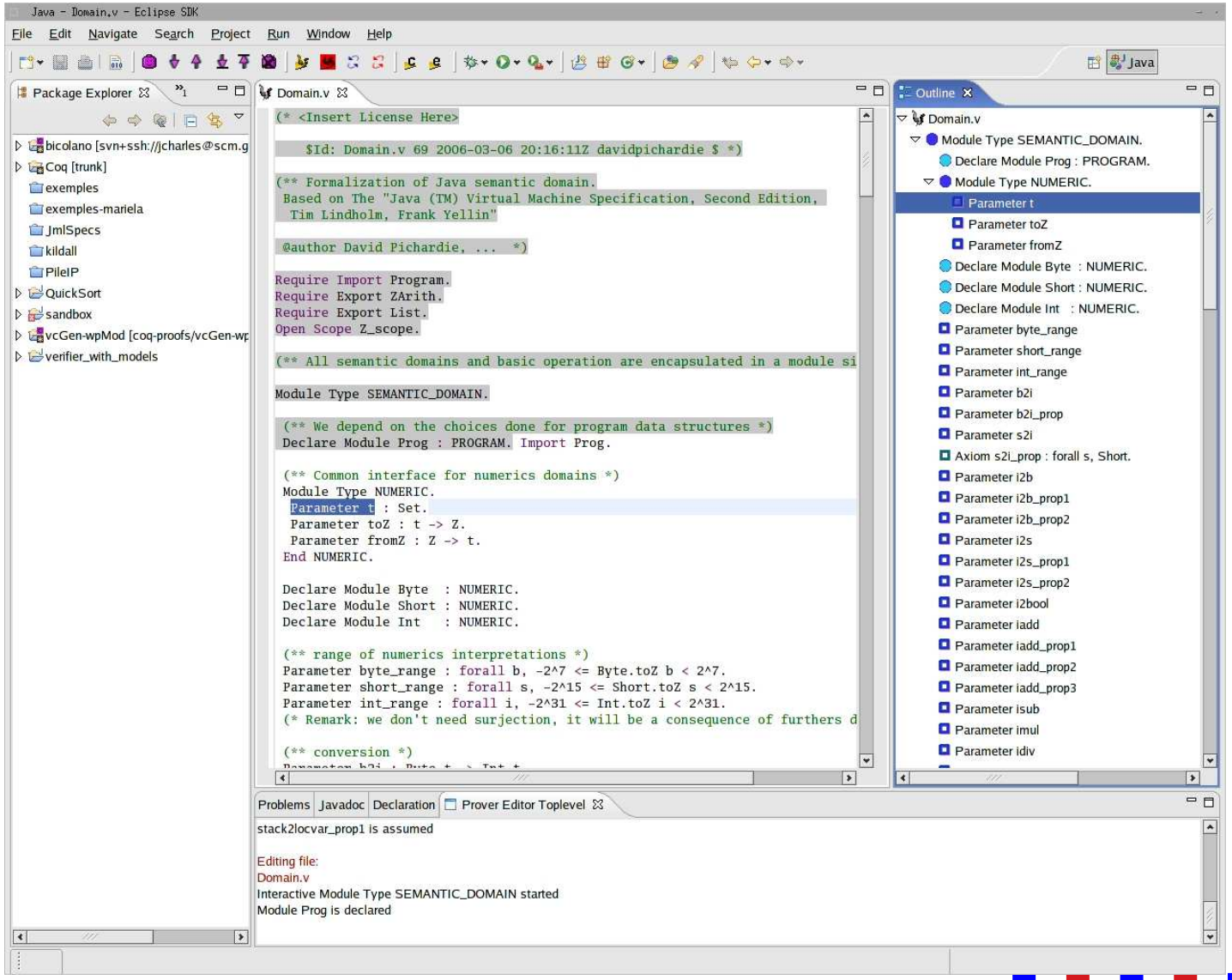


CoqProverEditor

- 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)

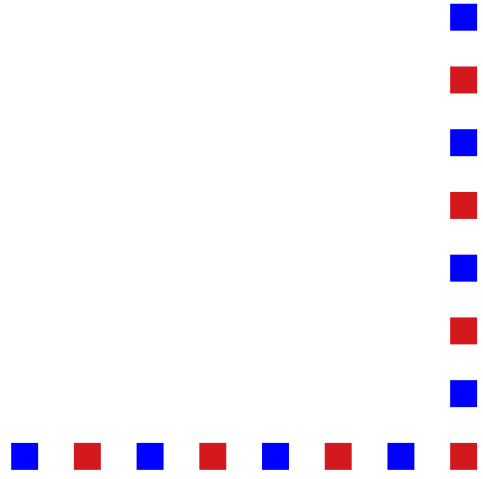


What it looks like



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

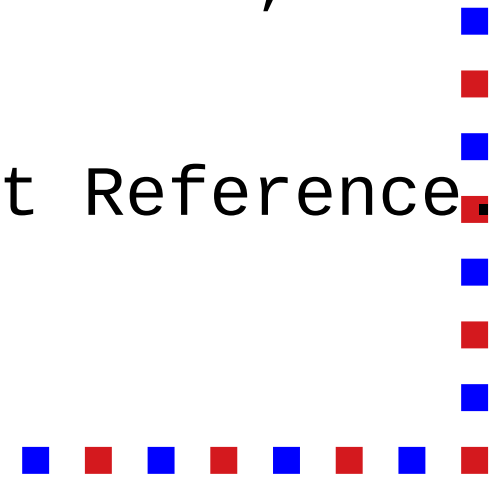
- To express complex properties, advanced data types useful
- Easily defined in Coq, not in JML
- Native types:

  Coq types in JML:

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

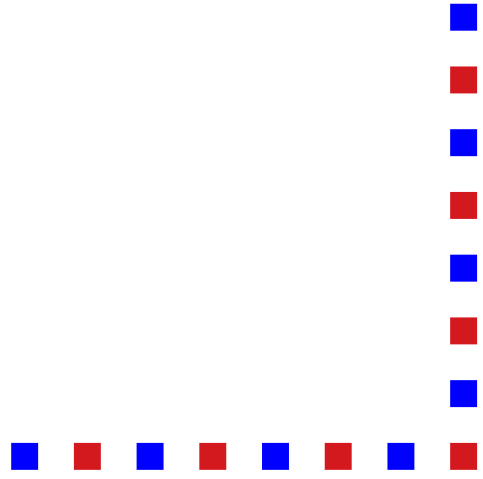
  In the Coq file user_extensions.v:

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



What are native types?

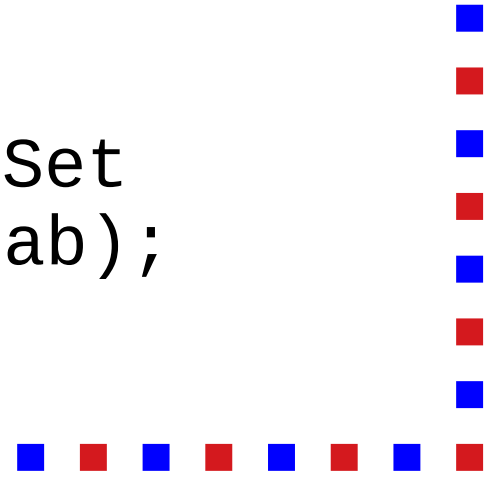
- Native types are not standard Java/JML class types:
  - Do not inherit from Object
  - No constructors
  - No casts
  - No instance creation
- Native types are functional type:
  - Modifiers are ‘static’
  - Modifiers create new objects



Example: set library

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

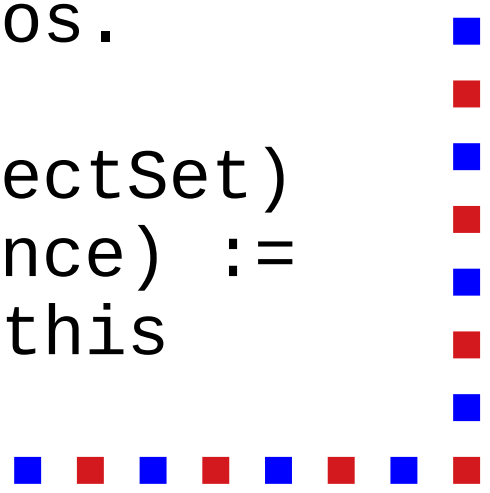
```
/*@ public native class ObjectSet {
  @ public native static ObjectSet
                  create();
  @ public native static ObjectSet
  @               add(ObjectSet os, Object o);
  @ public native boolean
                  member(Object o);
  @ public static native ObjectSet
  @               toSet(Object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