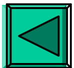
Utilisation de techniques formelles pour les cartes à puces

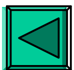
SEE SIC

Paris, 19 Juin 2003



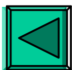
The domain : POPS

- Stands for "Petits Objets Portables de Sécurité",
  - smart card, palladium, GSM, wash machine...
  - every embedded mass product with or without security constraints
  - reasonable size for system or application software
  - non reasonable cost of upgrade linked to an on field bug discovery.
- Most of the attacks are hardware based,
- In a close future logical attacks will be the next nightmare.
- We need high quality software development.



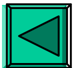
Avoiding vulnerabilities

- Logical attack : discover and use bugs
- In a security device, the bug is not acceptable...
- Correct construction of software
- Use of formal methods
  - Trusted smart card component: the BCV
  - B method and development evaluation



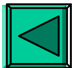
Objectives

- Fault avoidance
  - based on formal methods
  - increase the quality and the security of products by reducing vulnerabilities
- Fault removal
  - based on formal and semi formal methods
  - increase the productivity of the qualification process
- Side effect
  - the model obtained can be adapted and used in the certification process.



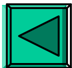
Formal Methods and Smart Cards

- Windows for Smart Card
  - Defensive Virtual Machine by Microsoft [Guerevitch]
  - Use of Abstract State Machine to describe their interpreter
  - Several memory area,
- The Multos e-purse and OS
  - Logica and University of York
  - Formal specification for certification purpose (Certified products)
  - Model of the abstract security properties behaviour and properties of the product using Z,
  - It’s a balance between model clarity and ease of proof.



Formal Methods and Smart Cards

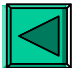
- dJVM by Cohen
  - Not complete Defensive Virtual Machine, important but unfinished work,
  - ACL-2 notation and theorem prover use,
  - Properties to prove: type safe execution, no implementation provided
- Java Card Verifier using a Model Checker [Posegga],
  - They transform each method of an applet into a state transition system (SMV),
  - They propose an abstraction (type),
  - Security properties as temporal formulae are verified with the model checker



Formal Methods and Smart Cards

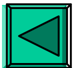
- And more recent works... :
  - Proof of a verifier using Coq on the F&M subset of the byte code [Bertot],
  - Modelling of a large subset of the Java Card Byte Code in Isabelle [Nipkow] and Coq [Jakubietz],
  - The Loop project at Niemegen University [Jacobs],
  - Formalisation of a byte code verifier at Kestrel Institute [Qian],
  - The Verificard project (www.verificard.org)

...and others...



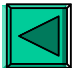
How to introduce FM in product development ?

- We claim that formal methods can be used in developing smart cards in such a way that gains in quality come at acceptable cost.
- Need to be demonstrated on a real case study…



The case study

- Embedding the byte code verifier is a real challenge:
  - the verifier is a key point of the security architecture,
  - we need the proof of correct implementation using a formal method.
- For the purpose of the evaluation we have developed two similar algorithms:
  - a PCC-based type verifier,
  - a standalone type verifier.



Java Card Architecture

Development Library

.jar

Java source code

Java Compiler

Java Class files

Card Image

Byte code verifier, converter, and signer

Off-card loader

.jar

*.java

Java Card Virtual Machine

Java Card files

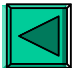
API

Interpreter

O.S.

On-card loader

.cap



Java Card Architecture

Development Library

.jar

Java source code

Java Compiler

Java Class files

Card Image

Converter

Off-card loader

*.java

.jar

Java Card Virtual Machine

Java Card files

No need to sign !!

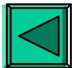
API

Interpreter

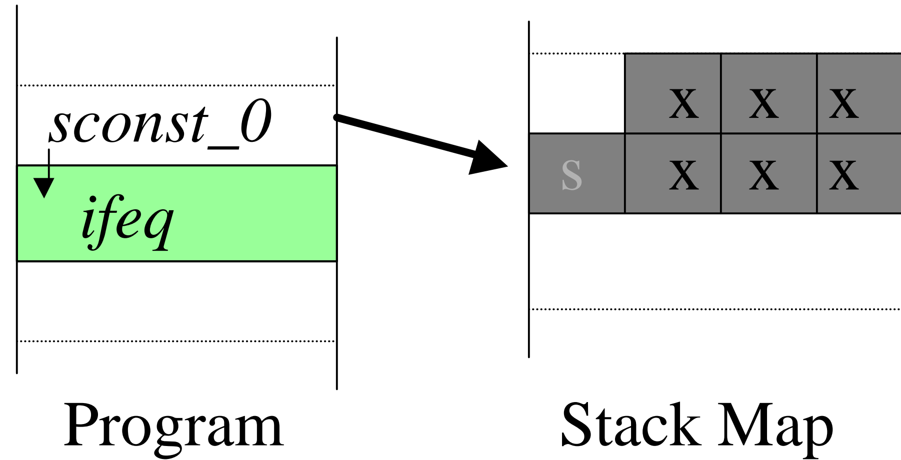
O.S.

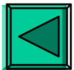
On-card loader, **byte code verifier**

.cap



Type Verification (cont.)





Type Verification (Cont.)

sconst_0

ifeq

	x	x	x
s	x	x	x

Program

Stack Map

sconst_0

ifeq

xx

iaload

goto xx

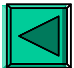
s	x	x	x

[i	s	x	x
i0	i	x	x
i0	i	x	x

TOP

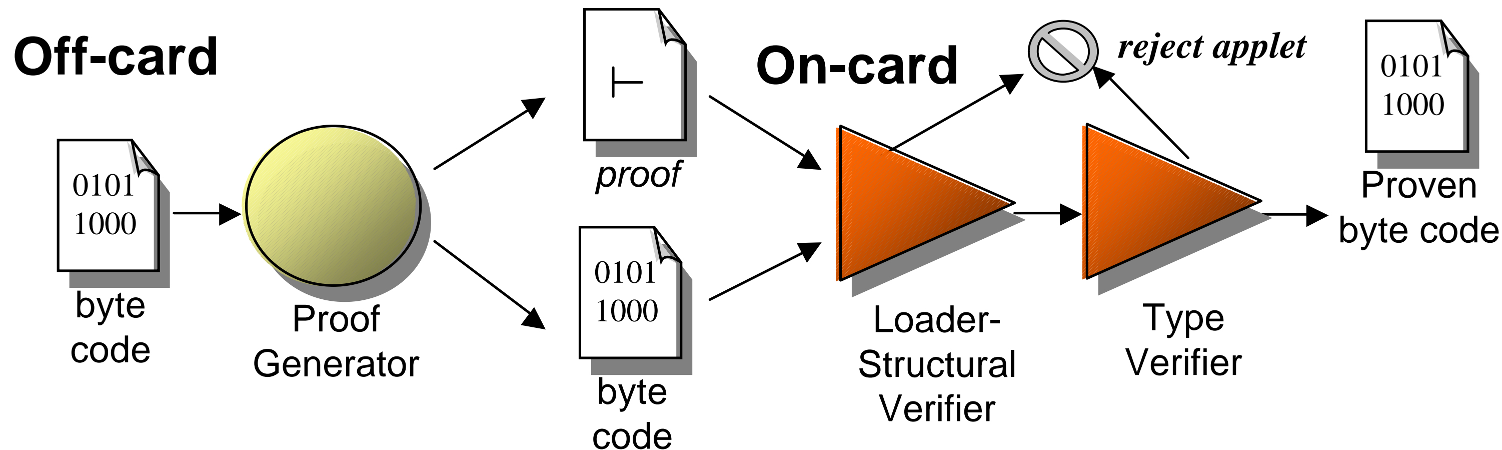
S i0

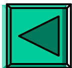
Compatible with



Two Solutions

- The **PCC Verifier** suitable for low-end chip,
  - small memory usage,
  - external pre-processing: stack map like KVM.

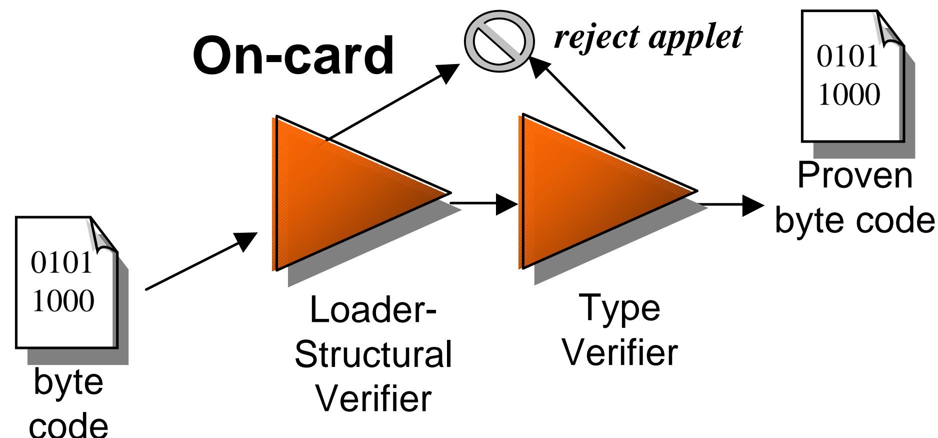


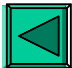


Two Solutions

- The **PCC Verifier** suitable for low-end chip,
  - small memory usage,
  - external pre-processing: stack map like KVM.
- The **Standalone Verifier** suitable for high end smart card,
  - no external preprocessing.

Off-card





The common part

Formal development

Cache mechanism
&
Graph algorithm
for type inference

PCC-based

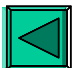
Type Verification

Type Verification

Standalone

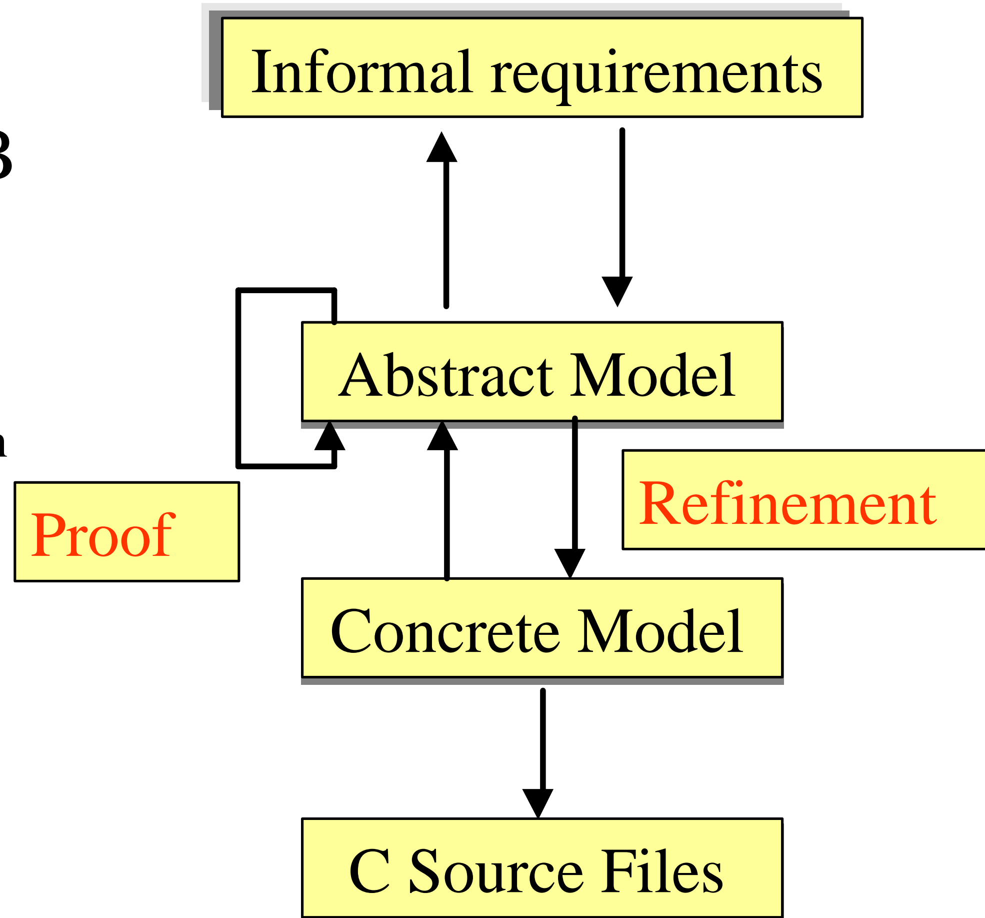
Structural
Verification

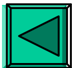
Conventional development



Formal development

- Development with the B formal method
  - definition of the architecture,
  - formalization of the specification in an abstract model,
  - refinement of the abstract model in a concrete model,
  - automatic code generation.





At the boundaries...

Informal specifications

Specification

Review

Requirements

Formalization

Review

B model

Refinement

Proof

B0 model

Translation

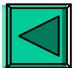
Test

C code

Optimization

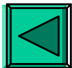
Test

Embedded code



Bias in the evaluation

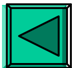
- We developed a prototype not a product,
  - not the same qualification process,
  - no field return,
- Skills of the teams,
  - different skills in type verification, Java Card…,
  - experts in B and C,
- Teams were not physically separated,
  - development, test, integration, review,
  - flow of knowledge,
  - they don't start exactly at the same time,
- Algorithms are different…
  - jsr, ret are pre-processed



Faults discovered by test

- In the 32 faults of the formal development, we have to retrieve the errors due to:
  - the translator ……………1
  - external tools…………….8
  - structural verification….9
- 14 errors remain related to the translation from informal to formal specification

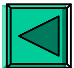
- In the 71 errors of the conventional development, two were linked with unification algorithm which is not included in the common part.



Real amount of discovered faults

	Formal	Conventional
Review	13	24
Proof	29	NA
Test	14	69
Total	56	93

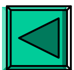
Zero default is unreachable !!!



Cost overhead is acceptable

- Development duration (in weeks)

	Formal	Conventional
Java understanding	Included in the next phase	4
Development	12	8
Proof	6	NA
Testing	1	3
Integration	1	2
Total	20 (weeks)	17 (weeks)

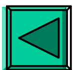


Automatically generated code fits the SC constraints

- Memory footprint:

	Formal	Conventional
Type ROM size (kb)	18	16
Structural ROM size (kb)	24	20
RAM (byte)	140	128-756*
Applet code overhead (%)	10-20	0

* RAM usage for this verifier is adjustable

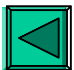


Automatically generated code fits the SC constraints

- Execution time (ms):

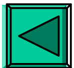
	Formal (6464)	Formal (3232)	Conventional (3232)
Wallet	811	411	318
Utils	2794	1312	1463
PacapInt	241	80	61
TicTacToe	3555	1232	1102

With the 3232, there is a direct access to the memory, while with the 6464, the B model needs a addition to each memory access



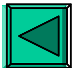
Methodology

- Easy integration of proved code/non proved code (C code),
- Efficient use of dedicated rules to simplify the proof process,
- Warning: we did not prove their correctness
  - need some more weeks to demonstrate them,
  - by the rule prover tool, by hand, with another prover...
- We still need to improve the code generation (currently too basic).



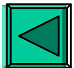
…but

- Software architecture constrained by the proof process
  - stronger architectural constraints than classical programming languages
  - modelling choices impact complexity of the proofs
    - requires taking the proof into account early
    - proof can require the specification and/or implementation rewriting
  - difficult to define an abstract model suitable for proving both implementation and specification
- Limit of the tool (wrong lemma, CPU usage,…).



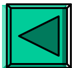
Conclusions

- Smart card is not a specific domain,
  - we obtain the same conclusion as in other domains,
    - overhead,
    - error in the formalization process.
  - tools used for railways run well for SC,
  - we have produced a SMART CARD.



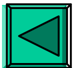
Introduction

- JACK : Java Applet Correctness Kit
  - Allow to formally prove applet correctness
  - Low-level design
- Challenge
  - Benefits from formal techniques without drawbacks
  - Let it affordable to developer
- Reduce unitary test cost
  - Gain formal assurance of code correctness



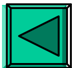
Strategy

- JML is the well-suited specification language
  - Language close to Java
  - Code is annotated by developers
- Used in LOGICAL and LEMME teams (INRIA)
- Let techniques accessible to developers
- Annotate source code



JML overview

- JML: **J**ava **M**odelling **L**anguage
  - Allow specifying Java programs
  - Similar to JavaDoc, but for specification
    - Embed specification in special comments
- JML benefits
  - Static analysis of applet using JML comments as hints/properties
    - Correctness proof
  - Generates reference manuals with formal specifications



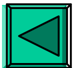
JML: Simple notation example

The array variable can never be null

The number of elements will never exceed the size of the array

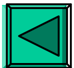
```
public class IntBag {
    /*@ non_null */ int [] array;
    int count;
    //@ invariant 0 <= count && count < array.length;

    //@ requires input != null;
    Bag(int [] input) {

        ...
```

This constructor must not be called with a null argument

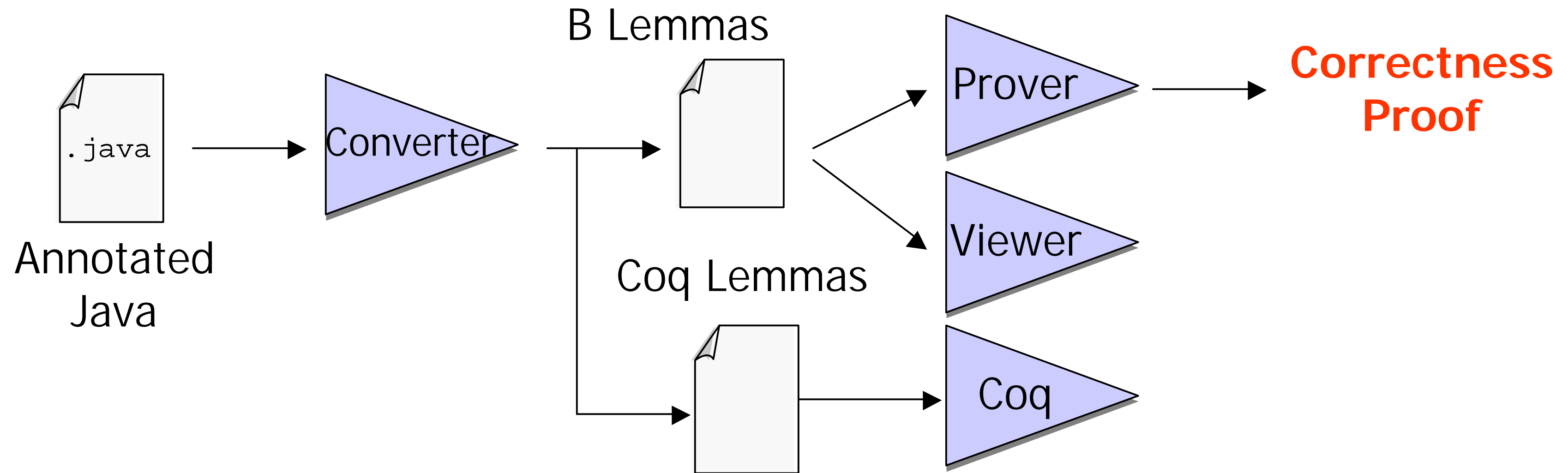


JML: Specification example

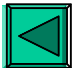
```
/**
  * Returns the minimum value of the three parameters.
 **/
/*@ ensures
  @ \result == a || \result == b || \result == c &&
  @ \result <= a && \result <= b && \result <= c;
  @*/
int min(int a, int b, int c) {
   return (a = (a <= b ? a : b)) <= c ? a : c;
}
```



Description of Jack

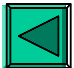


- Converter from JML annotated Java to formal lemmas (B, Coq, Simplify,…)
- Allows using the B automatic prover
- Lemma viewer hiding mathematics material



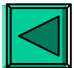
Limitations of the approach

- Native methods will still have to be validated separately
- VM-dependant properties are not taken into account
- Not all properties can be expressed in JML
  - Temporal properties, multithreading, abstraction, etc…



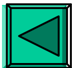
Formal method

- B Method
  - Industrially used
  - Automatic prover
- Coq
  - Interactive prover
- Drawbacks
  - Difficult to handle mathematical material
  - Need experts



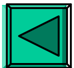
Viewer

- Allow developer to check lemma correctness
- Interface automatic prover
- Hide formal material
  - Lemmas are translated to Java
  - Prove become, in certain case, a simple click
- Need experts in last step to prove remaining lemmas or generate the adequate test suites.

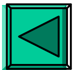


Conclusion

- Modeling implies thinking…,
- Formal method are affordable for POPS development,
- Formal techniques are no more a expert monopoly,
- Mixing different tools and techniques is probably the right way.

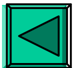


Annexes



Type Verifier

- Abstract model
  - the higher specification returns a boolean
  - defines the loop on all the methods
  - then, for each method, defines a loop on all the bytecodes
  - specifies the typing rules of the 184 different bytecodes
- Relies on the interface and the properties describing the CAP file
  - help defining the structural verifier



Byte code specification (1)

aaload (specification from Sun)

Load reference from array

Stack

..., *arrayref*, *index* .

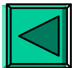
..., *value*

Description

The *arrayref* must be of type reference and must refer to an array whose components are of type reference. The *index* must be of type short. Both *arrayref* and *index* are popped from the operand stack. The reference *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

Runtime Exceptions

If *arrayref* is null, *aaload* throws a NullPointerException. Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an ArrayIndexOutOfBoundsException



Byte code specification (2)

aaload (informal specification rewriting)

[..., refarray class, short] => [..., ref class]

[..., null, short] => [..., null]

Pre-modification tests:

1. The stack must contain at least two elements
2. The two topmost elements of the stack have to be of types compatibles with refarray class and short.

Modifications:

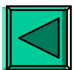
The two topmost elements of the stack are removed. If the second element was a refarray type, then a reference of the same class is pushed onto the stack. Otherwise a type null is pushed.

Post-modification tests:

None

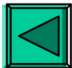
Throws

- NullPointerException
- ArrayOutOfBoundException
- SecurityException



Byte code specification (3)

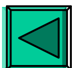
```
bb ← verify_aaload =
  PRE
    pc ∈ bytecode_ref_pkg
  THEN
    SELECT
      2 ≤ size(stack) ∧
      last(stack) = c_short ∧
      last(front(stack)) = c_refarray ∧
      (pc ∈ dom(exception_handler)
       ⇒
       ∀xx.(xx ∈ exception_handler(pc)
              ⇒
              COMPAT(loc_var,
                     loc_var_desc(proof_ref)(xx))) ∧
       c_uref ∉ ran(loc_var))
    THEN
      bb    := TRUE ||
      stack := front(front(stack)) ← c_ref
    WHEN
      size(stack) ≥ 2 ∧
      last(stack) = c_short ∧
      last(front(stack)) = c_null ∧
      (pc ∈ dom(exception_handler)
       ⇒
       ∀xx.(xx ∈ exception_handler(pc)
              ⇒
              COMPAT(loc_var,
                     loc_var_desc(proof_ref)(xx))) ∧
       c_uref ∉ ran(loc_var))
    THEN
      bb    := TRUE ||
      stack := front(front(stack)) ←c_null
    ELSE
      bb := FALSE
    END
  END
END
```



Byte code specification (4)

$bb \leftarrow$ **verify_aaload** $=$ **PRE** $pc \in bytecode_ref_pkg$ **THEN** **SELECT** $2 \leq$ **size**$(stack) \wedge$ **last**$(stack) = c_short \wedge$ **last**(**front**$(stack)) = c_refarray \wedge$ … **THEN** bb := **TRUE** || $stack :=$ **front**(**front**$(stack)) \leftarrow c_ref$	**WHEN** **size**$(stack) \geq 2 \wedge$ **last**$(stack) = c_short \wedge$ **last**(**front**$(stack)) = c_null \wedge$ … **THEN** bb := **TRUE** || $stack :=$ **front**(**front**$(stack)) \leftarrow c_null$ **ELSE** bb := **FALSE** **END** **END**

The two topmost elements of the stack are removed. If the second element was a refarray type, then a reference of the same class is pushed onto the stack. Otherwise a type null is pushed.



Byte code specification (5)

```
bb ← verify_aaload =
  PRE
    pc ∈ bytecode_ref_pkg
  THEN
    SELECT
      2 ≤ size(stack) ∧
      last(stack) = c_short ∧
      last(front(stack)) = c_refarray ∧
      (pc ∈ dom(exception_handler)
       ⇒
       ∀xx.(xx ∈ exception_handler(pc)
            ⇒
            COMPAT(loc_var,
                   loc_var_desc(proof_ref)(xx))) ∧
       c_uref ∉ ran(loc_var))
    THEN
      bb    := TRUE ||
      stack := front(front(stack)) ← c_ref
    WHEN
      size(stack) ≥ 2 ∧
      last(stack) = c_short ∧
      last(front(stack)) = c_null ∧
      (pc ∈ dom(exception_handler)
       ⇒
       ∀xx.(xx ∈ exception_handler(pc)
            ⇒
            COMPAT(loc_var,
                   loc_var_desc(proof_ref)(xx))) ∧
       c_uref ∉ ran(loc_var))
    THEN
      bb    := TRUE ||
      stack := front(front(stack)) ←c_null
    ELSE
      bb := FALSE
    END
  END
END
```