# Formal Development of a Byte Code Verifier

Ludovic Casset
Gemplus Research Lab
7th January 2002

# Matisse European Project

- 1 goal: propose methodologies and techniques to use formal methods in industry
- 3 industrial case studies in 3 different fields: transportation, health care and smart card
- 7 European partners
- End of the project by the end 2002
- Web site: www.matisse.qinetiq.com



GEMPLUS

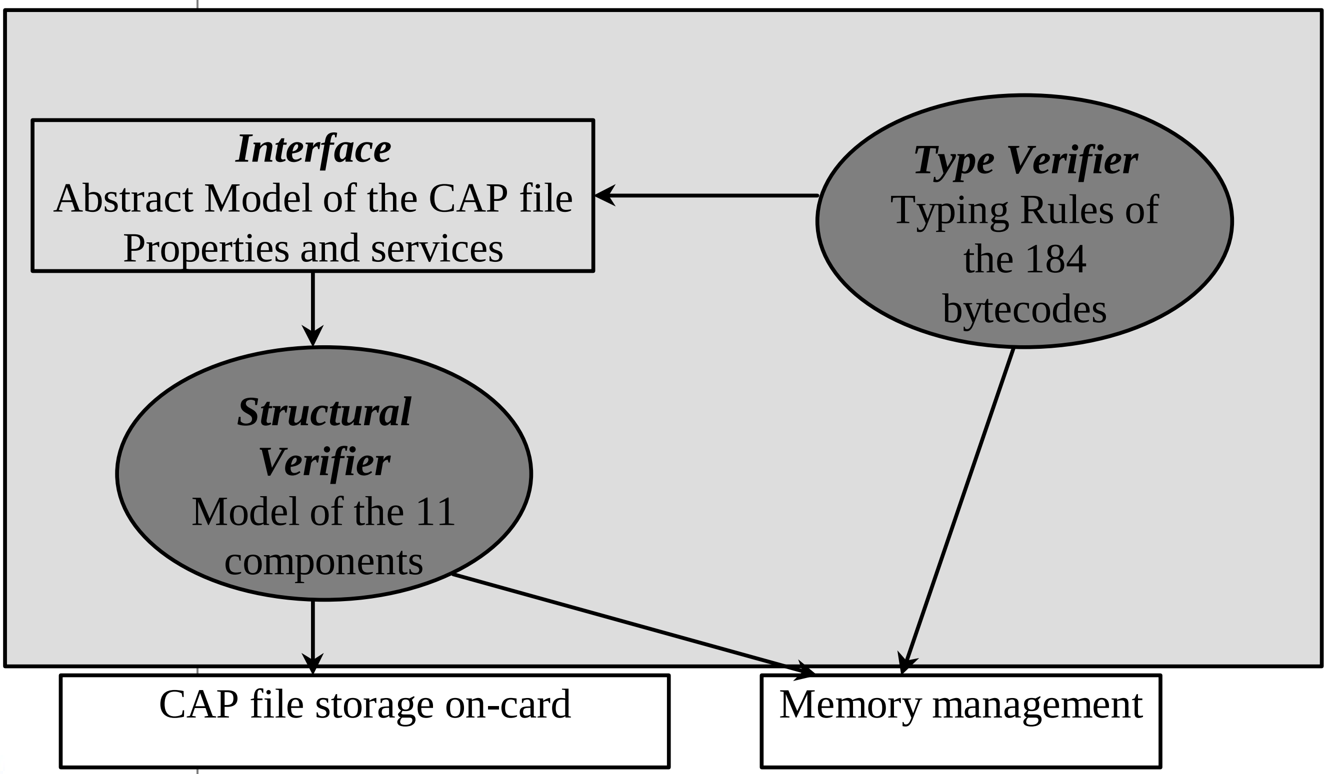
# JavaCard Bytecode Verifier Goals

- Ensures CAP file format
  - 11 standard components
- Ensures the enforcement of typing rules
  - a pointer cannot be forged from an integer
  - objects are accessed as what they are
- Ensures no stack over/underflow
- Ensures no memory violation
- ...

# Architecture of the Verifier

- A verifier divided into 2 parts
  - the type verifier
    - ensures that the Java Card typing rules are enforced
    - models each Java Card byte code
    - relies on the structural verifier to access data
  - the structural verifier
    - ensure that the byte stream represents a CAP file
    - models each CAP file component
    - provides access to data

# Architecture of the Verifier



# 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

# Type Verifier (cont.)

- Model of the sload byte code instruction

```
bb <-- verify_sload_n(idx) =
 PRE
     idx : t_byte
 THEN
     IF  size(stack)< method_maxstack(method_ref) &
          idx : dom(local_variable) &
          local_variable(idx) = c_short
     THEN
          bb := TRUE ||
          stack := stack <- c_short
     ELSE
          bb := FALSE
     END
  END;
```

# Type Verifier (Cont.)

- Concrete model
  - refines the abstract model
  - uses services provided by the interface
  - provides a proved implementation

# Type Verifier: Metrics

- Number of components : 34 (including mch, ref and imp)
- Number of lines of B: around 20 000
- Number of generated lemmas: around 18 160 POs
- Work Load : 5 mm

# Interface between Verifiers

- Abstract model of the CAP file

  - Properties of each component

```
method_returned_type: t_ptr +-> t_lattice_type &
c_uref /: ran(method_returned_type)
```

  - Services to access data within the CAP file

```
p_b <-- is_method_returning_value(p_desc_m) =
PRE    p_desc_m:ran(cp_token_desc_method)

THEN
       p_b:=bool(p_desc_m:dom(method_returned_type))
END
```



GEMPLUS

# Structural Verifier

- Implements the previous interface
- Specifies internal and external tests
  - the interface is not sufficient to define the structural verifier
  - it contains only properties related to the type verifier, not to the byte code interpreter
- This verifier relies on the model of the 11 standards components contained within the CAP file

# Structural Verifier

- Internal verifications
  - each component is modelled and checked
  - provide access to information
  - close to the hardware (memory representation)
  - not hard to specify, but hard to implement
  - proof hard to handle
  - bugs are not easily detected by the proof
    - bugs related with wrong offset when accessing data
    - tests not implemented (specification issues)
  - same result obtained with basic machines (see Class and Descriptor)



GEMPLUS

```
MACHINE cpn_component
VARIABLES
    Set of variables used to describe the component,
    Component_verified
INVARIANT
    Set of properties on variables previously defined &
    Component_verified : BOOL
INITIALISATION
    Initialisation of all variables describing the component
    Component_verified := FALSE
OPERATION
    Res ✍ component_internal_verif=
    PRE Component_verified = FALSE
    THEN (Component_verified = TRUE => the component is correct)
    END;
    Res ✍ other_services_1=
    PRE Component_verified = TRUE
    THEN …
    END
    …
END
```



GEMPLUS

# Structural Verifier (Cont.)

- External verifications
  - rely on services and properties of internal verifications
  - easier to specify and to implement
  - proof is also made easier thanks to properties provided by imported machines (internal verifications)
  - bugs that are found thanks to the proof
    - incoherence between components
    - wrong specification of components
    - properties missing
    - services missing

```
MACHINE
    Cpn_component_ext

SEES
    All cpn_components concerned by the consistency of
the component

OPERATIONS

    Res✍ test1=
    PRE Component_verified= TRUE &
        Component1_verified = TRUE &
             …
    THEN Res :: bool(Description of the property)
    END

    RES ✍ test2=
    PRE…
    THEN …
    END;

END
```



GEMPLUS

# Structural Verifier: Metrics

- Number of components : 116 (including mch, ref and imp)
- Number of lines of B: around 35 000
- Number of generated lemmas: around 11 700 POs
- Work Load: 8 mm
- Basic Machines: 6 (including the class and the descriptor)

# Bytecode Verifier Integration

- Not all implementations performed in B
  - use of Basic Machines
  - file loading and linking
- Need to represent
  - the card memory
  - packages already present in the card


GEMPLUS

# C Code Translator

- Developed within the G+ Lab
- Straight forward conversion into C code
  - no optimisation
  - in-lining possible
- Use only information available in the converted file
  - needs to add explicit typing information in the implementation
  - use typing information to distinguish *byte* from *short* and *int*

# Benefits from using Formal Methods

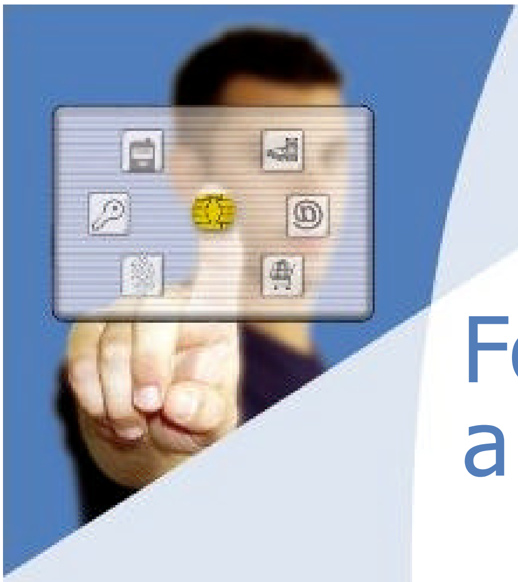
- Provides a complete and unambiguous specification of the byte code verifier
  - modelling activities help clarify the informal specification
- Provides a reference implementation of an on-card byte code verifier
  - a trusted implementation that conforms to its specification
- Provides elements for high level certification
  - the formal model of the byte code verifier is available

# Conclusion

- The code has been generated and loaded into an smart card chip
  - the code fits the smart card constraints
- The experience is conclusive
  - it is possible to develop code based on formal techniques and development that fits smart card constraints



GEMPLUS



# Formal Development of a Byte Code Verifier

Ludovic Casset
Gemplus Research Lab
7th January 2002