# Jakarta

*a tool support for formal verification*


Gilles Barthe

Pierre Courtieu

Guillaume Dufay

Marieke Huisman

Simão Melo de Sousa

Sorin Stratulat

`FirstName.LastName@sophia.inria.fr`

INRIA Sophia-Antipolis

France

# Talk Overview

- Background
- Presentation of the JaKarTa toolset
- JaKarTa 's Preliminary Results
- Conclusion and Perspectives

# Background

CertiCarte: Formal executable specifi cation of the JavaCard Plateform including offensive and defensive Virtual Machine and a ByteCode Verifi er.

- Defi nitions are (a bit) cluttered and diffi cult to modify
- Case-distinctions make proofs tedious
- Low level of automation (both in proofs and writing specifi cation)
- Diffi cult to make variations on the specifi cation (such as abstractions)

However, all these problems are not insurmountable: JaKarTa is design to provide solutions.

# JaKarTa, a toolset for formal verification

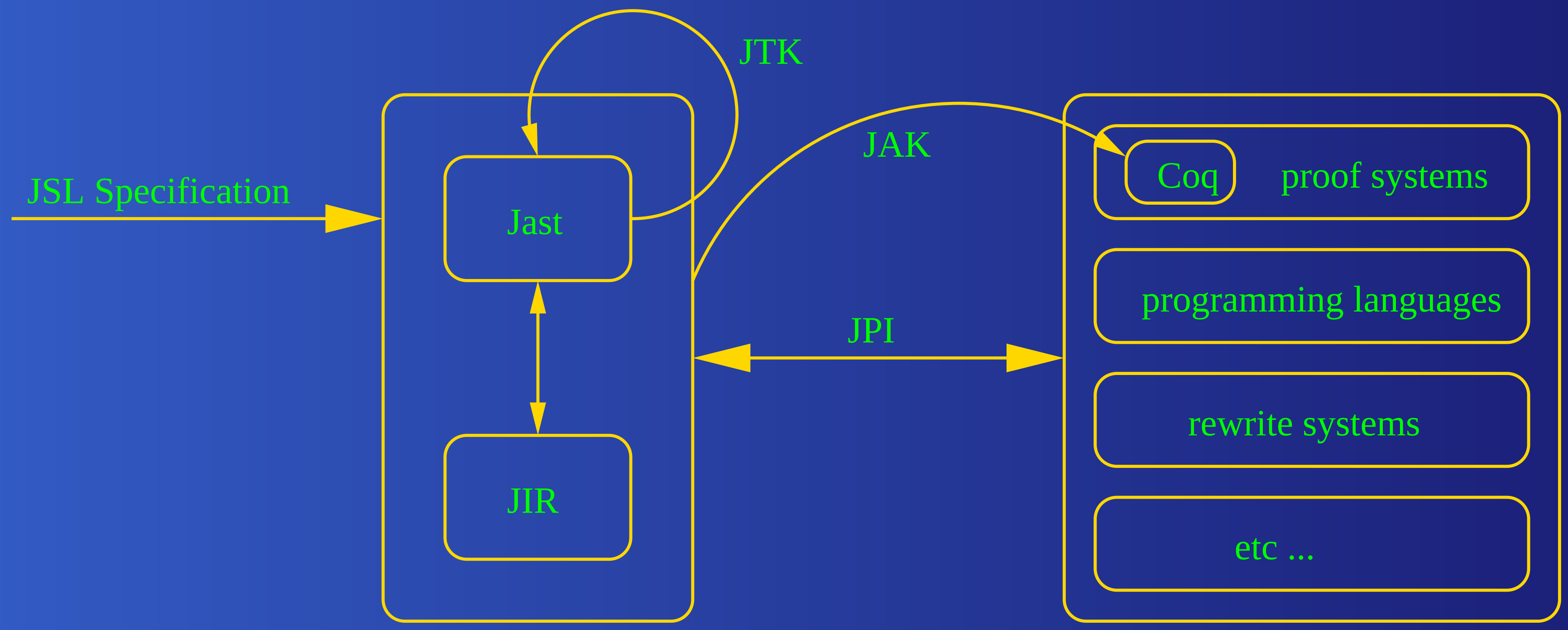
*A dedicated specification language can have a positive impact on formal specification and formal verification.*

Designed with the following goals in mind

- Clarity and Executablility of specifi cations
- Specifi cations easily transformable
- Tool independence
- Support for partial functions (by automatic transformation into total functions)
- Proof automation (generation of inversion principles)
- Support for refi nement and abstractions

INRIA SOPHIA ANTIPOLIS

# JaKarTa basic architecture




JaKarTa Basic Architecture

# JSL: *J*aKarTa *S*pecification *L*anguage

- JSL types: first-order polymorphic types
- JSL expressions: first-order algebraic terms built from
  - variables
  - constructor symbols (data type declarations)
  - defined symbols (function definitions)

$$\mathcal{E} := \mathcal{V} \mid \mathcal{E} == \mathcal{E} \mid c\,\vec{\mathcal{E}} \mid f\,\vec{\mathcal{E}}$$

INRIA SOPHIA ANTIPOLIS

# JSL: *J*aKarTa *S*pecification *L*anguage

Functions defi ned by set of rewrite rules:

*Constructor-based oriented conditional rewriting with extra variables*

$$l_1 \twoheadrightarrow r_1, \ \ldots \ , l_n \twoheadrightarrow r_n \Rightarrow g \rightarrow d$$

- $g = f\ \vec{x}$ with all $x_i \in \mathcal{V}$ pairwise distinct
- $r_i$ are patterns with fresh variables
- $\text{var}(l_i) \subseteq \text{var}(g) \cup \text{var}(r_1) \cup \ldots \cup \text{var}(r_{i-1})$
- $\text{var}(r_i) \cap \text{var}(g) = \emptyset$ and $\forall ij.\text{var}(r_i) \cap \text{var}(r_j) = \emptyset$

# JSL: *J*aKarTa *S*pecification *L*anguage

```
function take : nat -> list 'a -> list 'a :=
n -> Zero       =>(take n l)-> Nil;
n ->(Succ m),
l ->(Cons hd tl)=>(take n l)->(Cons hd (take m tl))
```

- Partial function
- First condition of second rule: binds `m`
- Second condition binds hd and `tl`
- Result uses fresh variables

# Interaction with other tools

2 internal data representations:

- Jast = *J*aKarTa *A*bstract *S*yntax *T*ree
- JIR = *J*aKarTa *I*ntermediate *R*epresentation = Jast complemented by a tree stuctured case distinction

2 kinds of translation:

- To "rewrite rule" based tools (such as ELAN or SPIKE) : Translation from Jast to the target language.
- To tools with tree stuctured case distinction languages (COQ, PVS or Ocaml): translation from JIR to the target language. For Coq, a "two-ways" translation is provided.

Consequence: JSL Specifi cation of CertiCarte for free.

INRIA SOPHIA ANTIPOLIS

# JAK: *J*aKarTa *A*utomation *K*it

- Automatic generation of appropriate theorems to be used in formal verifi cation
- Tailored towards specifi c theorem prover
- JAK's Current Focus: generation of adequate inversion principles for functions. This is particularly useful for Coq proofs.

# JTK: *J*aKarTa *T*ransformation *K*it

- For each datatype $\sigma$ define $\hat{\sigma}$ and $\lceil . \rceil_\sigma : \sigma \to \hat{\sigma}$
- For each defined function $f : \sigma \to \tau$, define $\hat{f} : \hat{\sigma} \to \hat{\tau}$ by transforming

$$l_1 \twoheadrightarrow r_1, \ \ldots \ , l_n \twoheadrightarrow r_n \Rightarrow g \to d$$

into

$$\lceil l_1 \rceil \twoheadrightarrow \lceil r_1 \rceil, \ \ldots \ , \lceil l_n \rceil \twoheadrightarrow \lceil r_n \rceil \Rightarrow \lceil g \rceil \to \lceil d \rceil$$

- Not a legal rule:substitution and cleaning required

# JTK: *J*aKarTa *T*ransformation *K*it

The user can:

- introduce is own solution to local abstraction. "Intelligence" in abstraction is introduced by this way
- introduce special guidance to optimize the treatment of
  - dead rules
  - type conversions
  - functions that become total

INRIA SOPHIA ANTIPOLIS

# Current JaKarTa Focus

- Input: Defensive Virtual Machine
- Output:
  - Offensive and Abstract Virtual Machines
  - Diagrams commute
  - Offensive and Defensive machines coincide on well-typed programs

Automating the correctness proof of the BCV is yet out of reach

INRIA SOPHIA ANTIPOLIS

# Current JTK Focus: Offensive Abstraction

```
data valu_prim  =
        VReturnAddress nat      |
        VBoolean z      |
        VByte z |
        VShort z        |
        VInt z.
```

becomes

```
type abs_valu_prim  = z.
```

# Current JTK Focus: Offensive Abstraction

```
function abstract_valu_prim
    : valu_prim -> abs_valu_prim :=
=>abstract_valu_prim (VReturnAddress v)
             -> (inject_nat v) ;
=>abstract_valu_prim (VBoolean v) -> v ;
=>abstract_valu_prim (VByte v) -> v ;
=>abstract_valu_prim (VShort v) -> v ;
=>abstract_valu_prim (VInt v) -> v .
```

INRIA SOPHIA ANTIPOLIS

# Current JTK Focus: Offensive Abstraction

```
<pUTSTATIC_rule_6>
(stack_f state)->(Cons h lf),
(head (opstack h))->(Value x),
(nth_elt (sheap_f state) idx)->(Value nod),
nod->(VPrim (VBoolean z0)),
t->(Prim Byte)
  => (pUTSTATIC t idx state cap)->
         (res_putstatic state x idx);
```

# Current Focus: Offensive Abstraction

```
<abstracted_pUTSTATIC_rule_6>
(abstracted_stack_f state)->(Cons h lf),
(head (abstracted_opstack h))->(Value x),
(nth_elt (abstracted_sheap_f state)
         idx)->(Value nod),
nod->z0,
t->(Prim Byte)
  => (abstracted_pUTSTATIC t idx state cap)->
         (abstracted_res_putstatic state x idx);
```

INRIA SOPHIA ANTIPOLIS

# Current Focus: Offensive Abstraction

```
<cONV_rule_2>
(stack_f state)->(Cons h lf),
(extr_from_opstack t (head (opstack h)))->(Value k)
   => (cONV t t’ state) ->
        (update_frame (update_opstack
           (Cons (VPrim (tpz2vp t’
                           (t_convert t t’ k)))
               (opstack h)) h) state);
```

INRIA SOPHIA ANTIPOLIS

# Current Focus: Offensive Abstraction

```
<abstracted_cONV_rule_2>
(abstracted_stack_f state)->(Cons h lf),
(head (abstracted_opstack h))->(Value k)
  => (abstracted_cONV t t' state)->
           (abstracted_update_frame
               (abstracted_update_opstack
                  (Cons (abstracted_tpz2vp t'
                                (t_convert t t' k))
                     (abstracted_opstack h)) h) state
```

# Current Focus: Offensive Abstraction

Script $\approx$ 40 lines $\Rightarrow$ whole offensive virtual machine

```
abstract exec_instruction with
    abstract_valu_prim  (etc...)
and inject_nat (etc...)
  (* user intervention directives start here *)
  conversion using inject_nat z2n
  (etc...)
  in cONV replace 2,2,1 by
          (head (abstracted_opstack h))
  (etc...)
  reject (abstracted_abortCode Type_error state)
into jcvm_off_functions    log jcvm_log.
```

# Conclusion

- right now JaKarTa  is proof of concept
- Tool independence ( translations to theorem provers, rewrite systems etc...)
- Generated offensive virtual machine, abstract machine underway
- Used JAK tactics to good effect
- Automation of equational reasoning is on the way

INRIA SOPHIA ANTIPOLIS