Vérification formelle de la plate-forme *Java Card*

• Thèse de doctorat •

Guillaume Dufay

INRIA Sophia Antipolis

Cartes à puce intelligentes

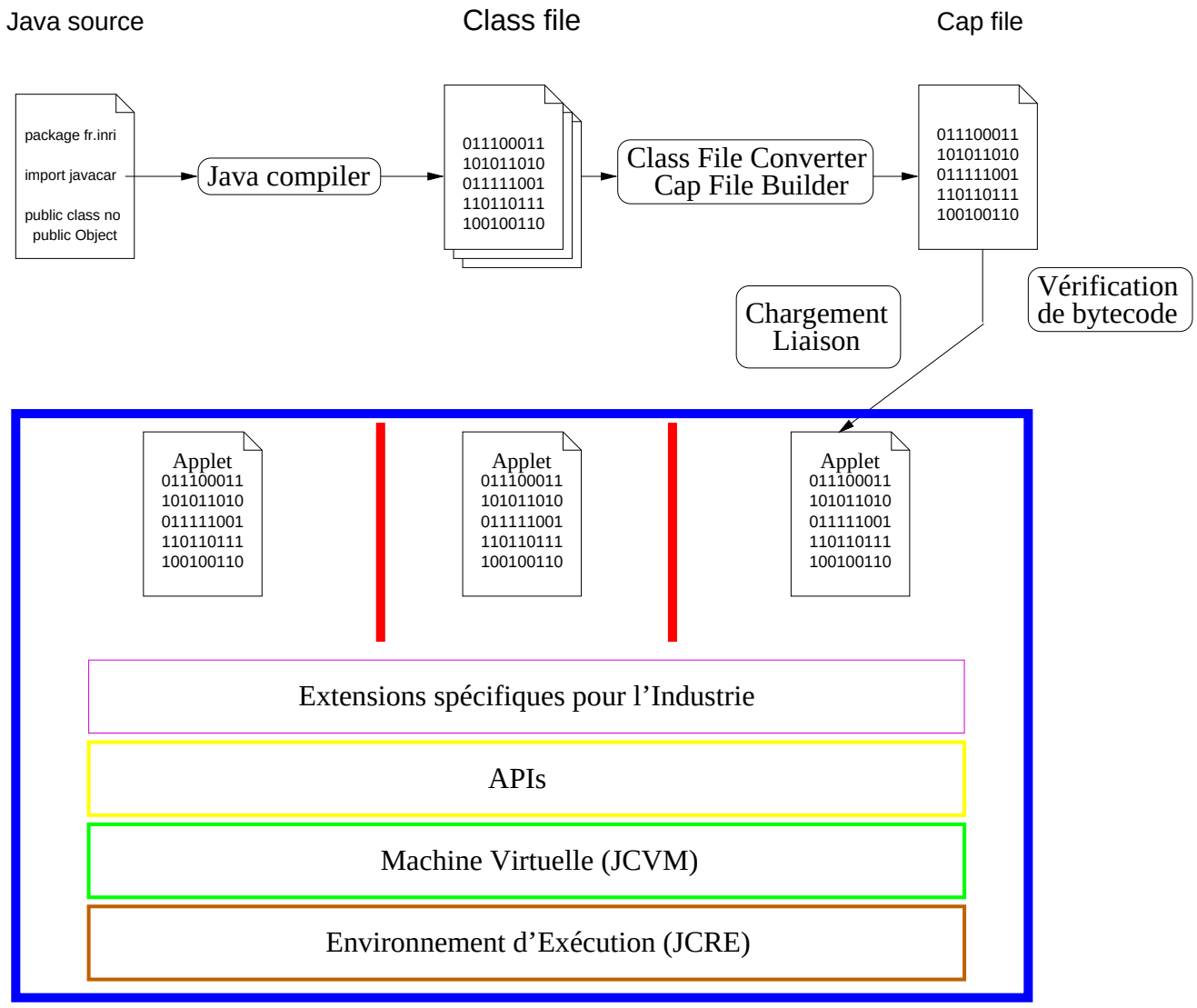
Java Card : Environnement de programmation dédié.

Dernières générations de cartes à puce pourvues d'un *microprocesseur* et d'un *système d'exploitation*.

- Stockent et gèrent de l'information
- Communiquent
- Contiennent plusieurs applications
- Partagent de l'information

Information manipulée *sensible* ou *confidentielle*.

Architecture Java Card



Problématique

Assurer la sécurité de la plate-forme.

- Formaliser la plate-forme pour raisonner sur celle-ci
- Méthodologie pour vérifier des propriétés statiques de la plate-forme
- Automatisation pour faciliter l'application de la méthodologie

Méthodes formelles

Exigées pour garantir la sécurité de systèmes complexes.

- Langage de spécification suffisamment expressif
- Système logique et techniques de démonstration
- Génération de code exécutable et certifié

Utilisation de l'outil COQ.

Méthodologie

Vérification d'une propriété statique P.

Machine P-défensive

proche des spécifications

P vérifié à l'exécution

Machine P-offensive

proche de l'implémentation

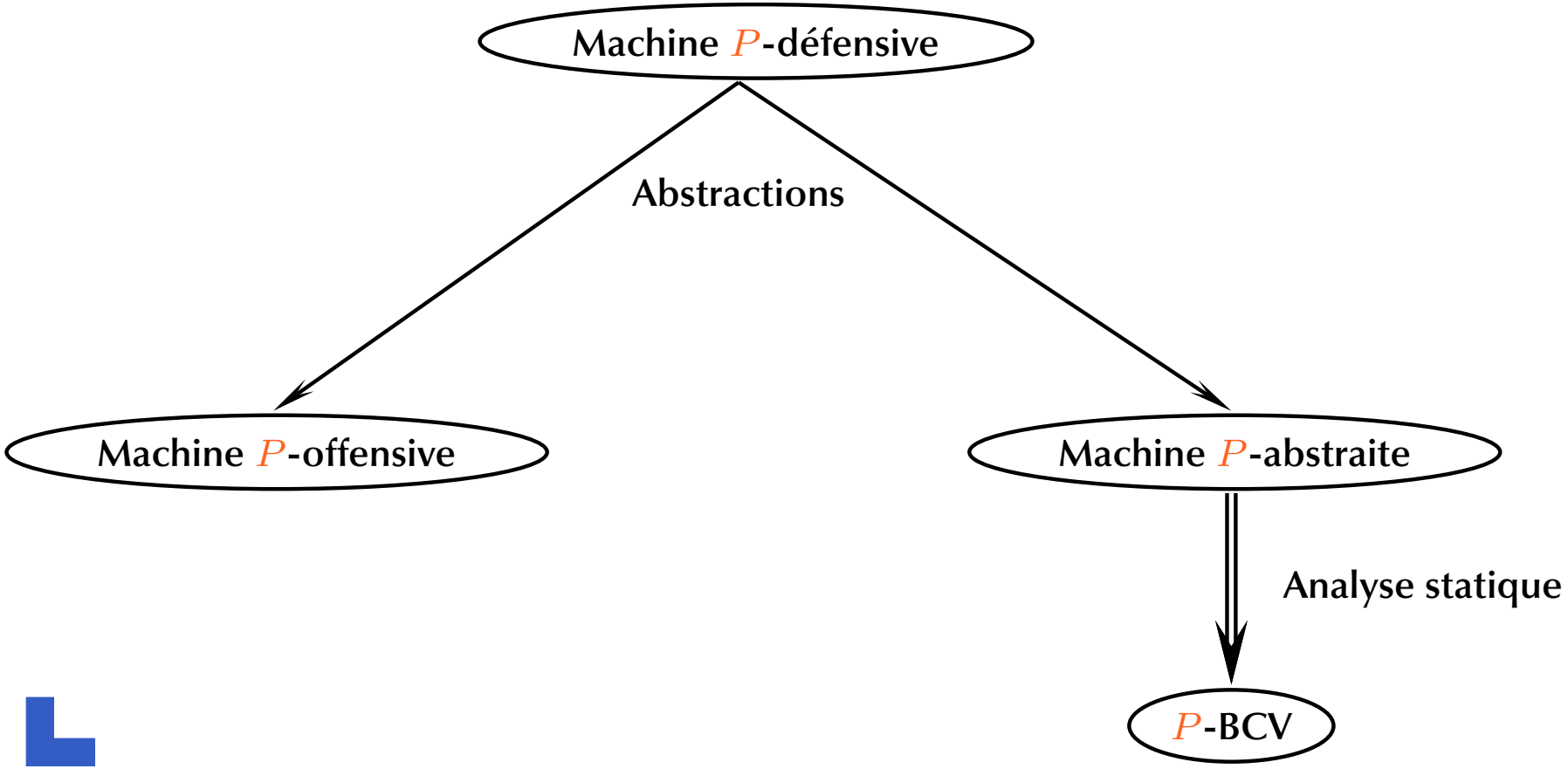
suppose P déjà vérifié

Machine P-abstraite

vérifie P uniquement

Méthodologie

Vérification d'une propriété statique P.



Méthodologie

Application à la sûreté du typage.

Machine $Type$-défensive

manipule : valeurs typées
types : vérifiés à l'exécution

Machine $Type$-offensive

manipule : valeurs non typées
correction typage assuré par BCV

Machine $Type$-abstraite

manipule : types comme valeurs

Plan

- Introduction
- *Formalisation de la JCVM*
  - *Programmes*
  - *Mémoire*
  - *Instructions*
- Abstractions
- Vérification de bytecode
- Conclusion

Réalisations

Formalisation en Coq de la JCVM et du JCRE.

- Intégralité des instructions
  (sémantique opérationnelle à petit pas)
- Environnement d'exécution (sauf méthodes natives)
- Spécifications fonctionnelles et exécutables

Comparable aux machines virtuelles de référence.

Programmes

Représentation des programmes après la phase de liaison (réalisée par le JCVM Tools).

```
Record jcprogram : Set := {
  classes    : (list Class);
  methods    : (list Method);
  interfaces : (list Interface);
  sheap_type : (list type)
}.
```

Programmes

Représentation des programmes après la phase de liaison (réalisée par le JCVM Tools).

```
Record Method : Set := {
  signature    : ((list type)*type);
  bytecode     : (list Instruction);
  handler_list : (list handler_type);
  is_static    : bool;
  ...
}.
```

Valeurs

Valeurs typées et vérification des types à l'exécution.

```
Inductive valu_prim : Set :=
  vByte          : Z → valu_prim |
  vShort         : Z → valu_prim |
  vInt           : Z → valu_prim |
  vBoolean       : Z → valu_prim |
  vReturnAddress : bytecode_idx → valu_prim.
```

États

Représentation du contenu de la mémoire.

```
Record frame : Set := {
  opstack : (list valu);
  locvars : (list (option valu));
  p_count : bytecode_idx;
  owner   : package;
  ...
}.
```

États

Représentation du contenu de la mémoire.

```
Record state : Set := {
  stack : (list frame);
  heap  : (list obj);
  sheap : (list valu);
}.
```

Exemple d'Instruction

```
Definition IF_COND := [b:bytecode_idx][st:state]
```

Exemple d'Instruction

```
Definition IF_COND := [b:bytecode_idx][st:state]
Cases (stack st) of
| nil ⇒ (AbortCode state_error st)
| (cons f lf) ⇒
  Cases (head (opstack f)) of
  | None ⇒ (AbortCode opstack_error st)
  | (Some v) ⇒
```

Exemple d'Instruction

```
Definition IF_COND := [b:bytecode_idx][st:state]
Cases (stack st) of
| nil ⇒ (AbortCode state_error st)
| (cons f lf) ⇒
  Cases (head (opstack f)) of
  | None ⇒ (AbortCode opstack_error st)
  | (Some v) ⇒
    Cases v of
    | (vPrim (vShort vx)) ⇒




    | _ ⇒ (AbortCode type_error st)
```

Exemple d'Instruction

```
Definition IF_COND := [b:bytecode_idx][st:state]
Cases (stack st) of
| nil ⇒ (AbortCode state_error st)
| (cons f lf) ⇒
  Cases (head (opstack f)) of
  | None ⇒ (AbortCode opstack_error st)
  | (Some v) ⇒
    Cases v of
    | (vPrim (vShort vx)) ⇒
      Cases (compare vx '0') of
      | true ⇒ (update_pc b ... st)
      | false ⇒ (update_pc (S (p_count f)) ... st)
      end
    | _ ⇒ (AbortCode type_error st)
end end end.
```

Plan

- Introduction
- Formalisation de la JCVM
- *Abstractions*
  - *JCVM offensive*
  - *JCVM abstraite*
  - *Génération automatique*
- Vérification de bytecode
- Conclusion

JCVM offensive

Manipule des valeurs non typées.

```
Definition valu := Z.
```

Reste de la mémoire inchangé.

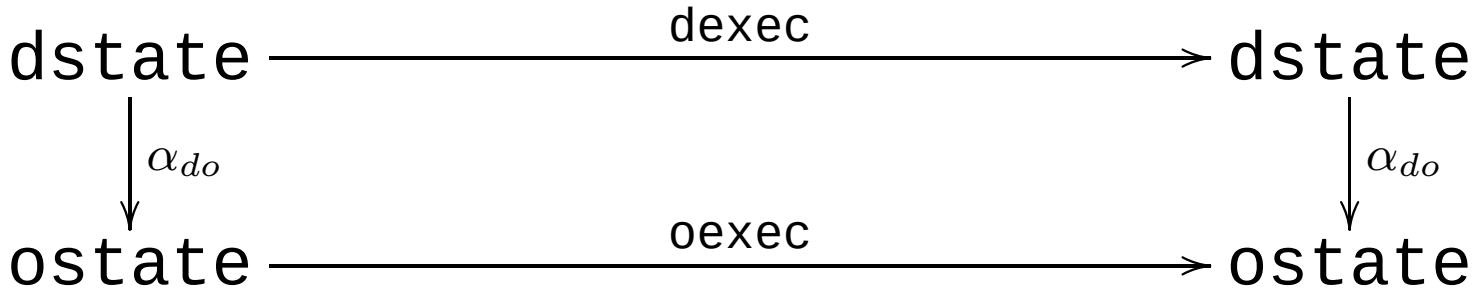
Exemple d'instruction

```
Definition IF_COND := [b:bytecode_idx][st:state]
Cases (stack st) of
| nil ⇒ (AbortCode state_error st)
| (cons f lf) ⇒
  Cases (head (opstack f)) of
  | None ⇒ (AbortCode opstack_error st)
  | (Some v) ⇒
    (* Cases v of *)
    (* | (vPrim (vShort vx)) ⇒ *)
      Cases (compare v '0') of
      | true ⇒ (update_pc b ... st)
      | false ⇒ (update_pc (S (p_count f)) ... st)
      end
    (* | _ ⇒ (AbortCode type_error st) *)
(* end *) end end.
```

Validation croisée

Assurer la correction de l'abstraction réalisée.

Fonction d'abstraction α_{do}



Le diagramme commute si la VM défensive ne lève pas d'erreurs de typage.

JCVM abstraite

Manipule pour valeurs des types.

```
Inductive valu_prim : Set :=
  vByte          : (* Z → *) valu_prim |
  vShort         : (* Z → *) valu_prim |
  vInt           : (* Z → *) valu_prim |
  vBoolean       : (* Z → *) valu_prim |
  vReturnAddress : bytecode_idx → valu_prim.
```

Mémoire

- Modèle de mémoire simplifié

```
Definition state := frame.
```

- Pas de gestion des exceptions
- Possible apparition de non-déterminisme

Exemple d'instruction

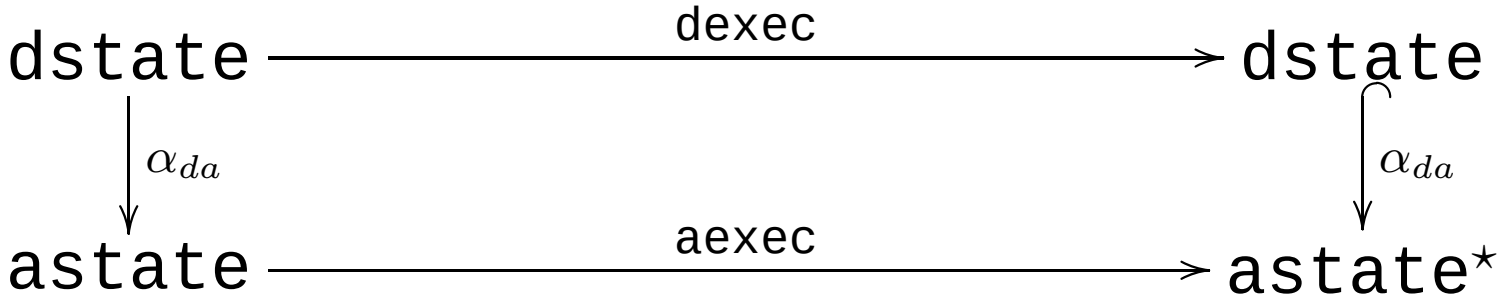
```
Definition IF_COND := [b:bytecode_idx][st:state]
(* Cases (stack st) of
| nil ⇒ (AbortCode state_error st)
| (cons f lf) ⇒ *)
  Cases (head (opstack st)) of
  | None ⇒ (AbortCode opstack_error st)
  | (Some v) ⇒
    Cases v of
    | (vPrim vShort) ⇒
      (* Cases (compare vx '0') of *)
      | true ⇒ (update_pc b ... st)
      | false ⇒ (update_pc (S (p_count f)) ... st)
      (* end *)
    | _ ⇒ (AbortCode type_error st)
end end (* end *).
```

Exemple d'instruction

```
Definition IF_COND := [b:bytecode_idx][st:state]
(* Cases (stack st) of
| nil ⇒ (AbortCode state_error st)
| (cons f lf) ⇒ *)
  Cases (head (opstack st)) of
  | None ⇒ (cons (AbortCode opstack_error st) (nil ?))
  | (Some v) ⇒
    Cases v of
    | (vPrim vShort) ⇒
      (* Cases (compare vx '0') of *)
      (cons (update_pc b ... st)
        (cons (update_pc (S (p_count f)) ... st) (nil ?)))
      (* end *)
    | _ ⇒ (cons (AbortCode type_error st) (nil ?))
end end (* end *).
```

Validation croisée

Fonction d'abstraction α_{da}



Le diagramme commute si l'exécution reste dans le même contexte et ne lève pas d'exceptions.

Jakarta

Outil pour la génération automatique d'abstractions et de preuves de correction.

- Format de spécification spécifique (avec passerelles)
- Moteur d'abstraction par traduction syntaxique
- Outil de génération d'énoncés de lemmes et de preuves
- Langage de script pour diriger l'abstraction

Abstractions réalisées par des scripts d'environ 50 lignes pour la VM offensive et 200 lignes pour la VM abstraite.

Plan

- Introduction
- Formalisation de la JCVM
- Abstractions
- *Vérification de bytecode*
  - *Principes*
  - *Implémentation*
  - *Instanciation*
- Conclusion

Principes

Analyse de flot de données du code d'un programme.

- Utilisation des valeurs avec des types corrects
- Pile d'opérande ayant la taille attendue
- Initialisation des registres et objets avant utilisation
- Bonne formation du programme

Assurer un comportement correct lors de l'exécution.

Réalisation

Analyse basée sur l'algorithme de KILDALL.

- Historique des états pour chacun des points de programme
- Existence d'une fonction d'unification sur les états
- Recherche d'un point fixe sur la relation d'exécution de la machine virtuelle abstraite
- Vérification méthode par méthode

Différents types d'analyses possibles (monovariante, polyvariante, ...).

Exemple de vérification

PC	instruction	1re passe
0:	`jsr 10`	(),[⊤;⊤]
1:	`iconst_0`	
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	
11:	`ret 1`	

États : (Pile d'opérande), [Variables locales]

Exemple de vérification

PC	instruction	1re passe
0:	`jsr 10`	(),[⊤;⊤]
1:	`iconst_0`	
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(RA 0),[⊤;⊤]
11:	`ret 1`	

États : (Pile d'opérande), [Variables locales]

Exemple de vérification

PC	instruction	1re passe
0:	`jsr 10`	(),[⊤;⊤]
1:	`iconst_0`	
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(RA 0),[⊤;⊤]
11:	`ret 1`	(),[⊤;RA 0]

États : (Pile d'opérande), [Variables locales]

Exemple de vérification

PC	instruction	1re passe
0:	`jsr 10`	`()`,`[⊤;⊤]`
1:	`iconst_0`	`()`,`[⊤;RA 0]`
2:	`istore_0`	
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	`(RA 0)`,`[⊤;⊤]`
11:	`ret 1`	`()`,`[⊤;RA 0]`

États : (Pile d'opérande), [Variables locales]

Exemple de vérification

PC	instruction	1re passe
0:	`jsr 10`	(),[⊤;⊤]
1:	`iconst_0`	(),[⊤;RA 0]
2:	`istore_0`	(int),[⊤;RA 0]
3:	`jsr 10`	
4:	`iload_0`	
10:	`astore_1`	(RA 0),[⊤;⊤]
11:	`ret 1`	(),[⊤;RA 0]

États : (Pile d'opérande), [Variables locales]

Exemple de vérification

PC	instruction	1re passe
0:	`jsr 10`	`()`,`[`$\top$`;`$\top$`]`
1:	`iconst_0`	`()`,`[`$\top$`;RA 0]`
2:	`istore_0`	`(int)`,`[`$\top$`;RA 0]`
3:	`jsr 10`	`()`, `[int;RA 0]`
4:	`iload_0`	
10:	`astore_1`	`(RA 0)`,`[`$\top$`;`$\top$`]`
11:	`ret 1`	`()`,`[`$\top$`;RA 0]`

États : (Pile d'opérande), [Variables locales]

Exemple de vérification

PC	instruction	1re passe	2e passe
0:	`jsr 10`	`()`,`[⊤;⊤]`	
1:	`iconst_0`	`()`,`[⊤;RA 0]`	
2:	`istore_0`	`(int)`,`[⊤;RA 0]`	
3:	`jsr 10`	`()`, `[int;RA 0]`	
4:	`iload_0`		
10:	`astore_1`	`(RA 0)`,`[⊤;⊤]`	`(RA 3)`,`[int;RA 0]`
11:	`ret 1`	`()`,`[⊤;RA 0]`	

Choix du type de vérification

Exemple de vérification

PC	instruction	1re passe	2e passe
0:	`jsr 10`	`()`,`[⊤;⊤]`	
1:	`iconst_0`	`()`,`[⊤;RA 0]`	
2:	`istore_0`	`(int)`,`[⊤;RA 0]`	
3:	`jsr 10`	`()`, `[int;RA 0]`	
4:	`iload_0`		
10:	`astore_1`	`(RA 0)`,`[⊤;⊤]`	`(RA 3)`,`[int;RA 0]`
11:	`ret 1`	`()`,`[⊤;RA 0]`	

Échec de la vérification monovariante

Exemple de vérification

PC	instruction	1re passe	2e passe
0:	`jsr 10`	`()`,`[⊤;⊤]`	
1:	`iconst_0`	`()`,`[⊤;RA 0]`	
2:	`istore_0`	`(int)`,`[⊤;RA 0]`	
3:	`jsr 10`	`()`, `[int;RA 0]`	
4:	`iload_0`		
10:	`astore_1`	`(RA 0)`,`[⊤;⊤]`	`(RA 3)`,`[int;RA 0]`
11:	`ret 1`	`()`,`[⊤;RA 0]`	

Vérification polyvariante

Exemple de vérification

PC	instruction	1re passe	2e passe
0:	`jsr 10`	`(),[⊤;⊤]`	
1:	`iconst_0`	`(),[⊤;RA 0]`	
2:	`istore_0`	`(int),[⊤;RA 0]`	
3:	`jsr 10`	`(),[int;RA 0]`	
4:	`iload_0`		
10:	`astore_1`	`(RA 0),[⊤;⊤]`	`(RA 3),[int;RA 0]`
11:	`ret 1`	`(),[⊤;RA 0]`	`(),[int;RA 3]`

Vérification polyvariante

Exemple de vérification

PC	instruction	1re passe	2e passe
0:	`jsr 10`	`()`,`[⊤;⊤]`	
1:	`iconst_0`	`()`,`[⊤;RA 0]`	
2:	`istore_0`	`(int)`,`[⊤;RA 0]`	
3:	`jsr 10`	`()`,`[int;RA 0]`	
4:	`iload_0`	`()`,`[int;RA 3]`	
10:	`astore_1`	`(RA 0)`,`[⊤;⊤]`	`(RA 3)`,`[int;RA 0]`
11:	`ret 1`	`()`,`[⊤;RA 0]`	`()`,`[int;RA 3]`

Vérification polyvariante

Implémentation

Réalisée en Coq sous forme de modules.

```
Module Type BCV.
Parameter state : Set.
Parameter exec  : (relation state).
Parameter err   : (predicate state).
Parameter check : (predicate state).

Axiom (decidable check).
Axiom ∀ a:state.(check a) →¬(reaches err exec a).
End BCV.
```

Implémentation

Réalisée en Coq sous forme de modules.

```
Module Type Abstract_VM.
Parameter state : Set.
Parameter err   : (predicate state).
Parameter loc   : Set.
Parameter locs  : (list loc).
Parameter succs : loc → state → (list loc).
Parameter exec  : loc → state → state.
End Abstract_VM.
```

Implémentation

Paramétrée par le type de vérification souhaité.

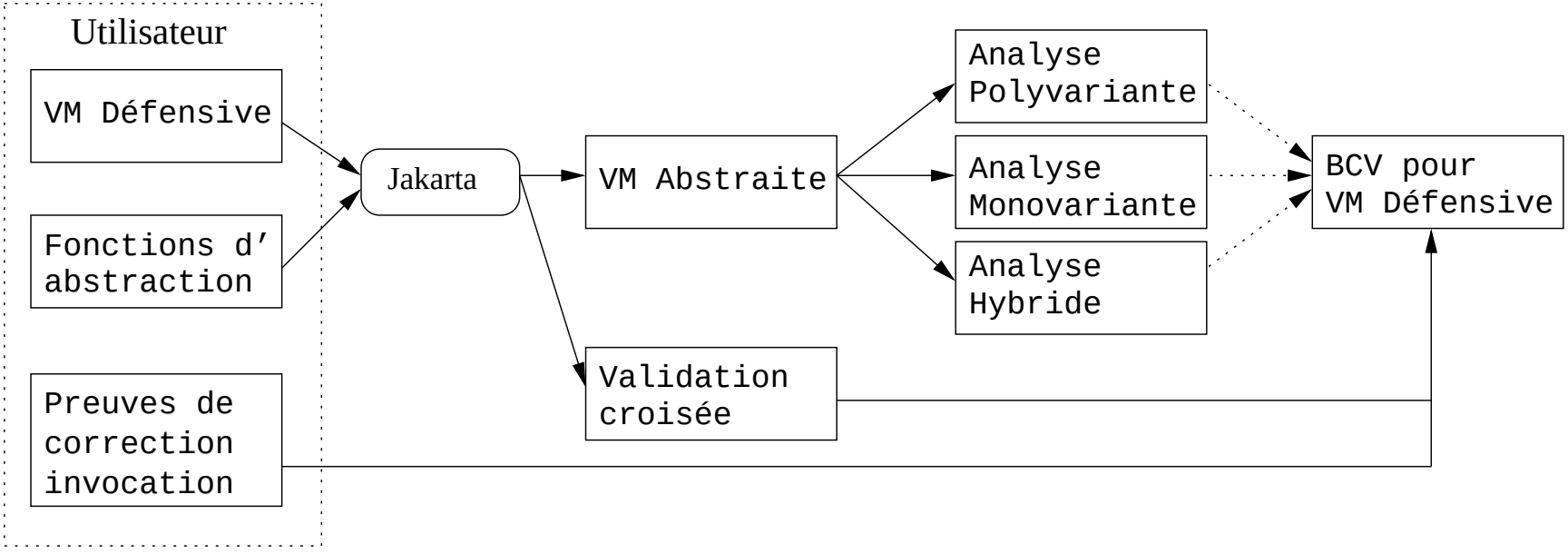
```
Module Type Polyvariant_Struct.
Declare Module avm : Abstract_VM.

Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).

Axiom ∀ l:loc.(monotone <_state (exec l)).
End Polyvariant_Struct.


Module PVS2SMS [pvs:Polyvariant_Struct] <: Stackmap_Struct.
Module BCV_aexec [sms:Stackmap_Struct] <: BCV.
```

Compositionnalité

Invariant de l'exécution défensive établissant une liaison avec l'exécution abstraite méthode par méthode.

- *instructions intra-procédurales* : résultats de validation croisée
- *appels de méthodes* : types des arguments compatibles avec la signature de la méthode
- *retour de méthodes* : type de retour compatible avec la signature de la méthode
- *lancement d'exceptions* : état attendu à l'entrée d'un gestionnaire

Résultats



Théorèmes obtenus :

$\forall$ s:dstate . (check s) $\rightarrow$ $\neg$(reaches err dexec s).

$\forall$ s:dstate . (check s) $\rightarrow$(α_{do} (dexec s))=(oexec (α_{do} s)).

Plan

- Introduction
- Formalisation de la JCVM
- Abstractions
- Vérification de bytecode
- *Conclusion*

Réalisations

- Machine virtuelle Java Card complète, avec son environnement d'exécution
- Méthodologie adaptée à la vérification de propriétés sur la plate-forme
  - Appliquée à la vérification du typage
  - Automatisation des tâches
- Modèle générique et paramétré de BCV

Intérêts de l'étude

- Utilisation des méthodes formelles (Critères Communs)
- Enjeux économiques liés à la sécurité du système
- Langage de programmation réaliste
- Originalité de l'approche

Travaux connexes

- Java en ACL 2
- Projet Bali : Java_{light} en Isabelle/HOL
- Gemplus : BCV embarquable pour Java Card en Atelier B
- Axalto et Trusted Logic : Plate-forme Java Card en Coq
- Secsafe : Sémantique opérationnelle de *Carmel* pour PVS
- JBook : Java en ASM-Gofer

Perspectives

- Application de la méthodologie à d'autres propriétés
  - Initialisation des objets
  - Non-interférence
  - Disponibilité des ressources
- Vérificateurs de bytecode optimisés
- Vérificateurs de bytecode embarqués
- Passage à Java, C♯, .Net

Merci de
votre Attention

?

?

!