

# Geccoo

## Génération de code certifié pour des applications orientées objet

Spécification, raffinement, preuve et détection d'erreurs

Christine Paulin-Mohring

INRIA Futurs & Université Paris Sud

Novembre 2006 - PaRI-STIC - Nancy



## 1 Introduction

Équipes  
Objectifs  
Approche

## 2 Quelques résultats

Modélisation de propriétés de sécurité  
Raffinement, encapsulation  
Génération d'obligations de preuve  
Résolution d'obligations de preuve  
Détection d'erreurs

## 3 Conclusion

# Les équipes

Introduction

Équipes

Objectifs

Approche

Quelques  
résultats

Conclusion

TFC	LIFC, Besançon	F. Bellegarde F. Bouquet/A. Giorgetti S. Ranise
Cassis	LORIA, Nancy projet commun LIFC	
Everest	INRIA-Sophia Antipolis	M. Huisman
ProVal	INRIA Futurs-LRI, Saclay	C. Paulin
VaSCo	LSR, Grenoble	M.-L. Potet

## *Outils de développement de **code orienté objet certifié***

- Code critique: cartes à puce (JavaCard), terminaux

## *Couvrir la **chaîne de développement***

- Spécifier des propriétés de sécurité, les traduire en propriétés logiques
- Développement modulaire : raffinement et composition
- Génération et résolution automatique des obligations de preuve
- Détection d'erreurs : simulation et test

*Une approche **formelle accessible** aux programmeurs*

- Programmes **JAVA** annotés en **JML**.
- Automatisation des preuves.

*Des modèles **rigoureux de haut niveau** pour raisonner*

- Systèmes d'évènements, méthode **B**.
- Assistant de preuve **Coq**.

*Développement d'**outils***

# Exemple de programme JAVA annoté en JML

Introduction

Équipes

Objectifs

Approche

Quelques  
résultats

Conclusion

```

class Purse {
    int balance;

    //@ public invariant balance >= 0;
    /*@ public behavior
        @ requires s >= 0;
        @ modifiable balance;
        @ ensures s <= \old(balance)
            && balance == \old(balance) - s;
        @ signals (NoCreditException)
            s > balance && balance == \old(balance);
    @*/

    public void withdraw(int s) throws NoCreditException {
        if (balance >= s) { balance -= s; }
        else { throw new NoCreditException(); }
    }
}

```

# Une application test Demoney

Porte-monnaie électronique

Introduction

Équipes

Objectifs

Approche

Quelques  
résultats

Conclusion

(Trusted Logic, projet Secsafe)

*Propriétés de sécurité :*

- Absence d'**erreurs d'exécution** : division par zéro, déréréférencement du pointeur nul, accès en dehors des bornes d'un tableau. . .
- **Authentication** : différents niveaux d'identification requis pour chaque opération.
- **Enchaînement des opérations** : l'opération d'initialisation d'une transaction doit être immédiatement suivie de l'opération de complétion de la transaction; interruption par arrachage . . .

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

**1** Introduction

Équipes  
Objectifs  
Approche

**2** Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

**3** Conclusion



# Modélisation de propriétés de sécurité

Introduction

Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

Conclusion

## *Problématique*

- des politiques de sécurité de haut niveau
- des conditions logiques à vérifier sur le code

## *Comment engendrer les conditions ?*

- Prise en compte du raffinement

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

Analyse du comportement d'un système **B** événementiel  
N. Stouls, D. Bert, X. Morselli [BZ'2005], Grenoble

- **B** évènementiel
- Génération et visualisation d'un système de transitions symboliques
- Étapes de raffinement
- Vérification de propriétés de sécurité

Application à l'étude de cas Demoney

*L'initialisation de la transaction doit être immédiatement suivie de sa complétion.*

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

A. Giorgetti et J. Gros Lambert [FASE 2006], Besançon  
collaboration avec M. Huisman, Sophia-Antipolis

**Java Temporal Pattern Language** logique temporelle pour  
JML s'appuyant sur une sémantique de **traces**

**after** storeData() **normal always** storeData() **not enabled;**

- Traduction: propriétés temporelles vers assertions JML
- Propriétés de **sûreté** et de **vivacité**  
(hypothèse de progrès)
- Traduction vers **B**, technique directe en **JML**
- Préservation de propriétés par raffinement
- Test de propriétés de sécurité avec **JML-TT**

## A. Haddad [B 2006], Grenoble

### *Données*

- Modèles de sécurité pour le contrôle d'accès (ex:MAC, RBAC . . .)
- Instanciation pour un problème particulier  
Définition des droits
- Application particulière  
Définition des opérations

### *Résultat*

- Génération de **préconditions** pour les fonctions permettant d'assurer la politique de sécurité.

- Modèle de sécurité (RBAC):
  - Utilisateurs : `shop`, `bank`, `admterm`
  - Rôles : `debit` | `credit` | `admin`
  - Lien entre utilisateur et rôle  
(`shop,debit`) (`bank,credit`) (`admterm,admin`)
  - Lien entre rôle et opération
    - `set pin` : `admin`
    - `credit` : `credit`
    - `check pin, debit` : `debit`

Permet d'engendrer automatiquement une précondition pour chaque opération spécifiée qui vérifie les droits en fonction de l'utilisateur

# Raffinement et encapsulation

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

### **Raffinement, encapsulation**

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

- spécifications **JML** associées au code
- raisonner sur des modèles abstraits
- modulariser, factoriser les preuves

P.-A. Masson et J. Gros Lambert, Besançon

Spécification par **raffinement** de Demoney

- Renforcement des pré et post-conditions normales et exceptionnelles, pas d'ajout d'effet de bord, inclusion des types d'exceptions
- Travail sur la spécification indépendamment du code
- Chaque modèle correspond au détail d'une fonctionnalité (en tout 13 modèles)
  - personnalisation
  - niveau d'accès
  - séquençement des fonctions d'authentification
  - messages d'erreur
  - ...

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

**Raffinement, encapsulation**

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

*Retour d'expérience*

- Travail réalisé principalement par des étudiants de master pro
- Le dernier niveau correspondant aux communications par APDU est le plus gros (550 lignes)

*Suite à donner*

- Génération d'obligations de preuve de raffinement
- Lien avec le code
- Eviter l'explosion du modèle au niveau le plus bas



# Spécifications abstraites

## Introduction

### Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

### *Situation JML*

- spécification de **JML**: vérification dynamique
- introduction de méthodes pures, de modèles vus comme des classes et opérations **JAVA**
- manque de spécifications abstraites

### *Approche logique*

- Utiliser des types et des prédicats logiques interprétés dans le prouveur
- version J. Charles pour **Jack**: classes **native** [FTfJP'2006]
- version C. Marché pour **Krakatoa** : spécifications abstraites

# Spécifications abstraites en Krakatoa

Introduction

Quelques  
résultatsModélisation de  
propriétés de  
sécuritéRaffinement,  
encapsulationGénération  
d'obligations de  
preuveRésolution  
d'obligations de  
preuve

Détection d'erreurs

Conclusion

## *JAVA interface pour des ensembles finis*

```
class IntSet {
    // checks whether n belongs to this
    public boolean mem(int n);

    // adds n to this
    public void add(int n);
}
```

## *Spécification algébrique*

```
/*@ logic type intset;
   @ logic intset emptyset();
   @ logic intset singleton(int n);
   @ logic intset union(intset s1, intset s2);
   @ predicate in(int n, intset s);
   @*/
```

## Spécifications abstraites en

## Krakatoa

## Axiomes

Introduction

Quelques  
résultatsModélisation de  
propriétés de  
sécuritéRaffinement,  
encapsulationGénération  
d'obligations de  
preuveRésolution  
d'obligations de  
preuve

Détection d'erreurs

Conclusion

```

/*@ axiom in_empty :
  @ (\forall int n; !in(n,emptyset()));
  @
  @ axiom in_singleton :
  @ (\forall int n,k;
  @   in(n,singleton(k)) <==> n==k;
  @
  @ axiom in_union :
  @ (\forall int n; \forall intset s1,s2 ;
  @   in(n,union(s1,s2)) <==>
  @     in(n,s1) || in(n,s2);
  @
  @ axiom intset_ext :
  @ (\forall intset s1,s2 ;
  @   (\forall int n ; in(n,s1) <==> in(n,s2))
  @   ==> s1==s2) ;
  @*/

```

# Specification IntSet

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

```

class IntSet {
    int size;
    int t[];
    /*@ invariant
        @   t != null && 0 <= size <= t.length &&
        @   (\forallall int i,j; 0 <= i <= j < size; t[i] <= t[j]);
    @*/

    /*@ model intset my_model;
        @ predicate IntSet_models(intset s, IntSet this) {
        @   this != null &&
        @   (\forallall int n; in(n,s) <==>
        @       (\exists int k; 0 <= k < this.size; n==t[k])) }
        @ representati on i nvariant IntSet_models(my_model,this);
    @*/

    //@ ensures \result <==> in(n,my_model);

public boolean mem(int n);

    /*@ modi fi abl e my_model;
        @ model ensures
        @   my_model == union(\old(my_model), singleton(n));
    @*/

public void add(int n);
}

```

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

S. Boulmé et M-L Potet [B'2007], Grenoble en collaboration avec C. Marché, Saclay

- Problème important pour la modularité et la composition des preuves
- Interactions complexes avec le partage des données

```
class M { int x;
        public void incr () { x++ } }
class N extends M { //@ invariant x mod 2 = 0;
        public void incr2 () { incr(); incr(); } }
```

- L'opération `incr` ne préserve pas l'invariant de `N`.
- En `B` les règles strictes d'inclusion de machines permettent de préserver la modularité des preuves mais sont restrictives

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

Adaptation à **B** d'une technique développée par R. Leino pour **Spec#** reposant sur une relation de possession.

- Relation de possession : **M** possède **N** si l'invariant de **M** fait référence à des variables de **N**
- Chaque objet a un statut dynamique :
  - **valid** ou **committed** : invariant vérifié  
les objets que je possède sont en mode **committed**
  - **invalid** : variables modifiables
  - **committed** : au plus un maître
- Des opérations explicites pour changer les statuts des objets
  - Revenir en mode **valid** nécessite de revérifier l'invariant
- Une manière d'expliquer et d'étendre le modèle de **B**

# Génération et résolution d'obligations de preuve

## Introduction

## Quelques résultats

Modélisation de  
propriétés de  
sécurité

Raffinement,  
encapsulation

**Génération  
d'obligations de  
preuve**

Résolution  
d'obligations de  
preuve

Détection d'erreurs

## Conclusion

## *Génération*

- Capturer les propriétés essentielles dans le modèle
- Faciliter le travail dans les démonstrateurs

## *Résolution*

- Théories et stratégies adaptées

# Modèles mémoire pour la sécurité

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

N. Rousset et C. Marché [SEFM'06], Saclay  
Mécanismes JavaCard de transaction et mémoire volatile

```
byte bi;
byte bt[];
/*@ invariant isTransient(bt) != NOT_A_TRANSIENT_OBJECT;
byte bp[];
/*@ invariant isTransient(bp) == NOT_A_TRANSIENT_OBJECT;

/*@ requires JCSystem.getTransactionDepth() == 0;
   @ ensures bi == \old(bi) &&
   @         bt[0] == 1 && bp[0] == \old(bp[0]);
   @*/
public void test() {
    JCSystem.beginTransaction();
    bi = 1;
    bt[0] = 1;
    bp[0] = 1;
    JCSystem.abortTransaction();
}
```



# Transactions JavaCard

## Introduction

### Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

- Modélisation en **Why** en introduisant des copies de la mémoire
- Adaptation de la spécification `beginTransaction()`, `commitTransaction()`, et `abortTransaction()` pour chaque applet
- l'arrachage est modélisé grâce aux exceptions de **Why** possiblement déclenchées lors d'une mise à jour
- Test sur la vérification du code PIN
  - le code naïf ne fonctionne pas
  - le code correct est vérifié automatiquement par **SIMPLIFY**

# Génération d'obligations de preuve

Autres résultats

Introduction

Quelques  
résultats

Modélisation de  
propriétés de  
sécurité

Raffinement,  
encapsulation

Génération  
d'obligations de  
preuve

Résolution  
d'obligations de  
preuve

Détection d'erreurs

Conclusion

**Jack** : L. Burdy, J. Charles, J-L Lanet, M. Pavlova,  
Sophia-Antipolis.

- Module de développement pour **Coq** (modèle, tactiques)
- Traduction de spécification **JML** vers des annotations de byte-code [**SAC'06**]
- Génération automatique d'annotations : propriétés de sécurité, absence de menaces

**Krakatoa** : C. Marché, J-C. Filliâtre, C. Paulin, Saclay

- Découpage statique de la mémoire en zones
- Proposition de modèle pour les nombres flottants

# Résolution d'obligations de preuve

## Introduction

### Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

- Les obligations de preuves forment une classe de problèmes intéressante
  - **combinaison** de raisonnement propositionnel et de **procédures de décision** pour des types de données.
  - **quantifications** du premier ordre
  - raisonnement **interactif** (induction)
  - **stratégies** spécifiques à mettre au point
- Domaine très actif (**Yices**, **Boogie**, **CVC**, **Zenon** ...)
- Un outil ancien comme **Simplify** reste parmi les plus performants

# Résolution d'obligations de preuves

## Introduction

## Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

## Conclusion

**haRVey** : D. Déharbe, P. Fontaine, S. Ranise, Nancy

- Deux implémentations (**haRVey** et **RV-sat**)
- Reconstruction de preuves de **RV-sat** vers

# Résolution d'obligations de preuves

Introduction

Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

Conclusion

S. Conchon, E. Contejean, J-C. Filliâtre, Saclay

## *Une approche multi-prover*

- architecture générique pour la génération des théories, interface avec les prouveurs interactifs
- traduction de la logique de **Why** (multi-sortée, polymorphe) vers des formalismes plus faibles

## *Un nouveau prouveur : Ergo*

- nouvelle méthode de combinaison de théories (congruence closure, arithmétique)
- logique multi-sortée polymorphe
- une architecture petite et propre (3000 lignes de Ocaml, proches des règles logiques)
- reconstruction partielle de termes de preuves Coq

# Détecter les erreurs

**JML Testing-Tools** Animation et test de spécifications **JML**  
F. Dadeau, F. Bouquet, J. Gros Lambert, B. Legnard, Besançon.

- Adaptation de l'outils BZ Testing-Tools à **JML** [AFADL'06]
  - Utilisation du formal ensembliste interne BZP
  - Modélisation de la mémoire **JAVA**, et de **JML**
  - Moteur de résolution de contraintes
  - Outils d'animation
- Génération de tests
  - Activation des comportements décrits par la spécification [FM'06]
  - Propriétés de sécurité [FATES/RV'06]
  - Test aux limites
- Mise au point de spécifications
  - Traduction dans un modèle **B** [B'2007]
  - Expression de propriétés, satisfiabilité

Introduction

Quelques résultats

Modélisation de propriétés de sécurité

Raffinement, encapsulation

Génération d'obligations de preuve

Résolution d'obligations de preuve

Détection d'erreurs

Conclusion

Introduction

Quelques  
résultats

Conclusion

## 1 Introduction

Équipes  
Objectifs  
Approche

## 2 Quelques résultats

Modélisation de propriétés de sécurité  
Raffinement, encapsulation  
Génération d'obligations de preuve  
Résolution d'obligations de preuve  
Détection d'erreurs

## 3 Conclusion

# Avancées réalisées dans GECCOO

- Approche logique des propriétés de sécurité
  - Traduction de propriétés temporelles en annotations (**JAG**)
  - Génération de pre-post à partir de politiques de sécurité (**Meca**)
  - Mécanisme des transactions JavaCard
- Contributions à l'approche **JML**
  - Outils de simulation, génération de tests, analyse de spécification (**JML-TT**)
  - Spécifications abstraites (**native, model**)
  - Reflexion sur les invariants, héritage, encapsulation
- Modélisations autour de l'application **Demoney**
- Avancées sur les outils : **Jack, Krakatoa, haRVey**

La confrontation d'approches différentes de méthodes formelles par preuve a été fructueuse.