Une machine virtuelle en héritage multiple basée sur le hachage parfait

Julien Pagès

LIRMM - Université de Montpellier

M. Roland Ducournau	Professeur	LIRMM, Université de Montpellier	Directeur
M. Philippe Clauss	Professeur	INRIA, Université de Strasbourg	Rapporteur
M. Gaël Thomas	Professeur	Télécom SudParis	Rapporteur
M. Christophe Paul	Directeur de recherche	LIRMM, CNRS	Examinateur
M. Jean Privat	Professeur	Université du Québec à Montréal	Examinateur
M. Manuel Serrano	Directeur de recherche	INRIA Sophia Antipolis	Examinateur
M. Floréal Morandat	Maître de conférences	LaBRI, ENSEIRB-MATMECA	Invité

Sujet de la thèse

Deux principaux objectifs

- Preuve de concept d'une **machine virtuelle** pour un **langage à objet** en **héritage multiple**
  - Le modèle d'exécution est similaire à Java
- Études des **protocoles de compilation/recompilation** dans ce système
  - Les optimisations sont la clé de la performance des machines virtuelles
  - Mais sont souvent très peu décrites dans la littérature

La programmation objet

L'objet : données (attributs) et comportements (messages).

Classification

- Langage à classes et prototypes
- Typage statique et dynamique
- Héritage simple, sous-typage multiple, héritage multiple

Les machines virtuelles

Trois grandes familles de systèmes d'exécution : interpréteur, compilateur, machine virtuelle

Un lent processus de développement

- Quelques idées de la compilation à la volée sont présentes depuis LISP : fin des années 60
- Smalltalk et Self ont introduit les machines virtuelles
- Java puis C♯ ont popularisé ces systèmes

Caractéristiques actuelles d'une machine virtuelle

- Chargement dynamique (monde ouvert)
- Compilation à la volée
- Gestion automatique de la mémoire

État de l'art des systèmes actuels

Constat

- Il n'existe pas de machine virtuelle pour un langage en héritage multiple et typage statique
- Tous les langages en typage statique ont une forme d'héritage multiple : interfaces pour Java/C♯ et traits pour Scala

Héritage multiple

- L'héritage multiple est utile pour le programmeur
- Mais pose des problèmes d'implémentations

Le hachage parfait [Ducournau, 2008] est une solution.

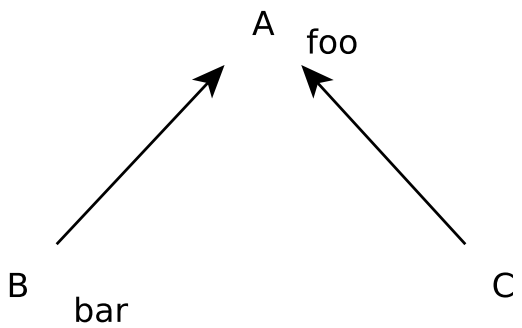
Mécanismes de la programmation objet

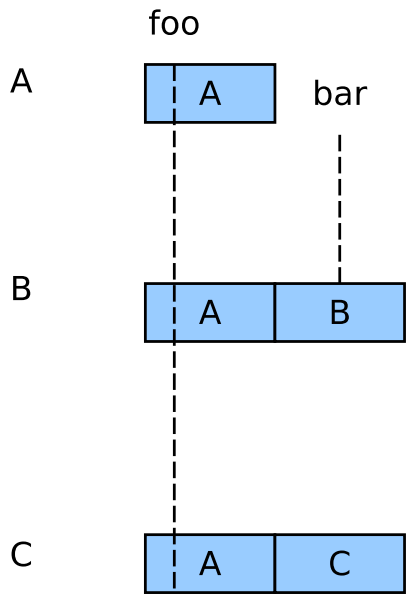
Trois principaux mécanismes

- Appel de méthode
- Accès aux attributs
- Test de sous-typage

Héritage simple (typage statique)

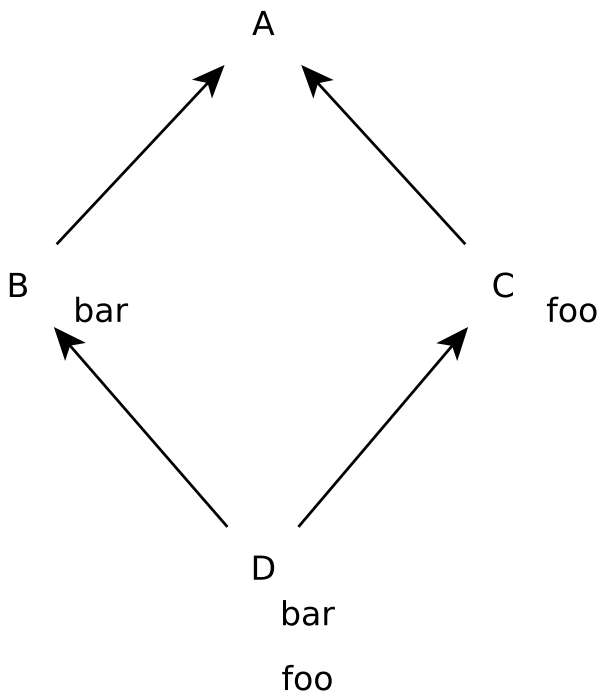
Position fixe pour chaque méthode

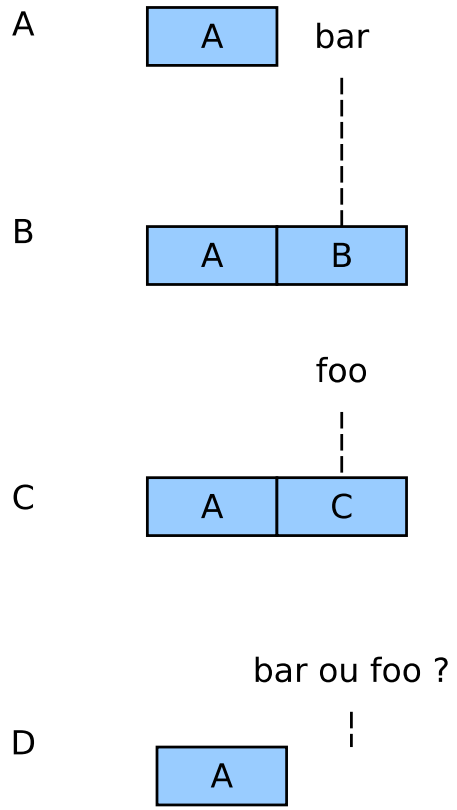




Héritage multiple

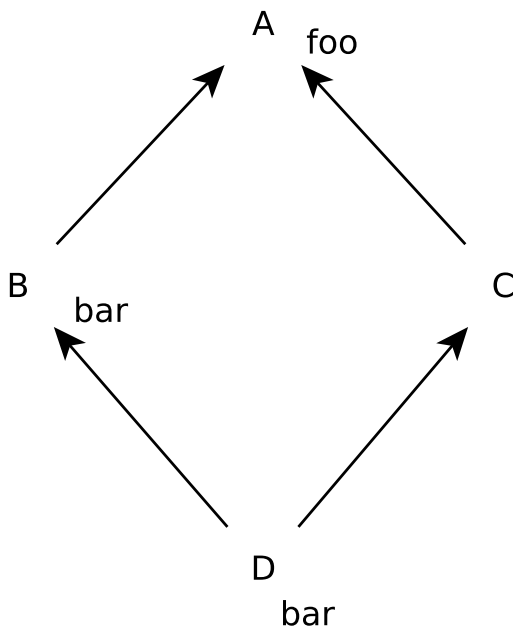
Problème : que faire en cas d'héritage multiple ?

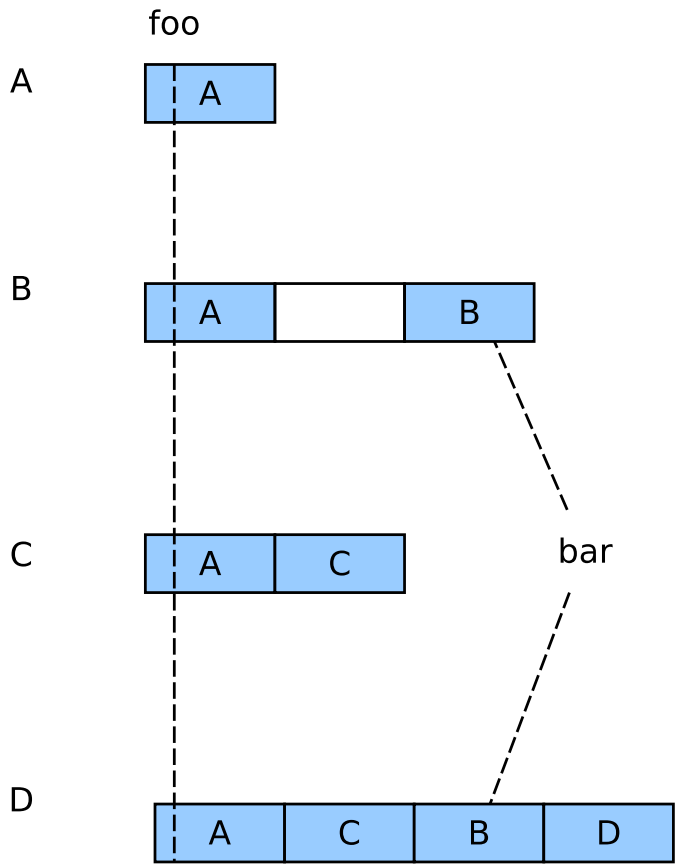




Solutions en monde fermé

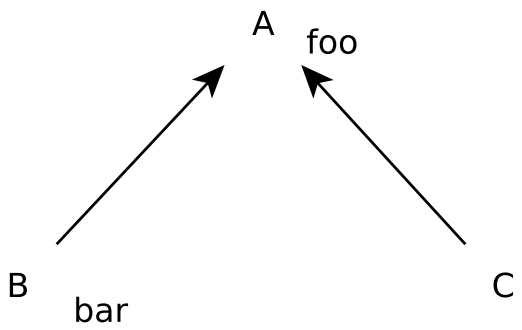
En monde fermé : la coloration serait utilisable

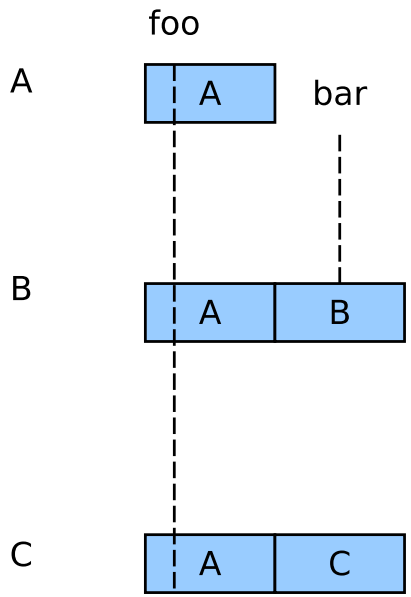




Chargement dynamique

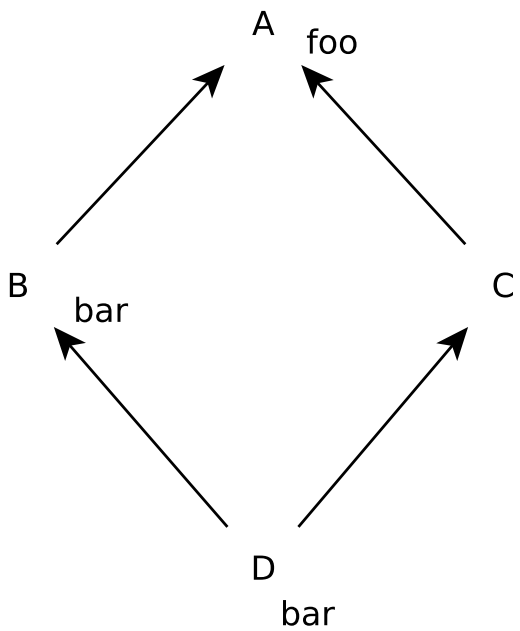
En monde ouvert (chargement dynamique) :

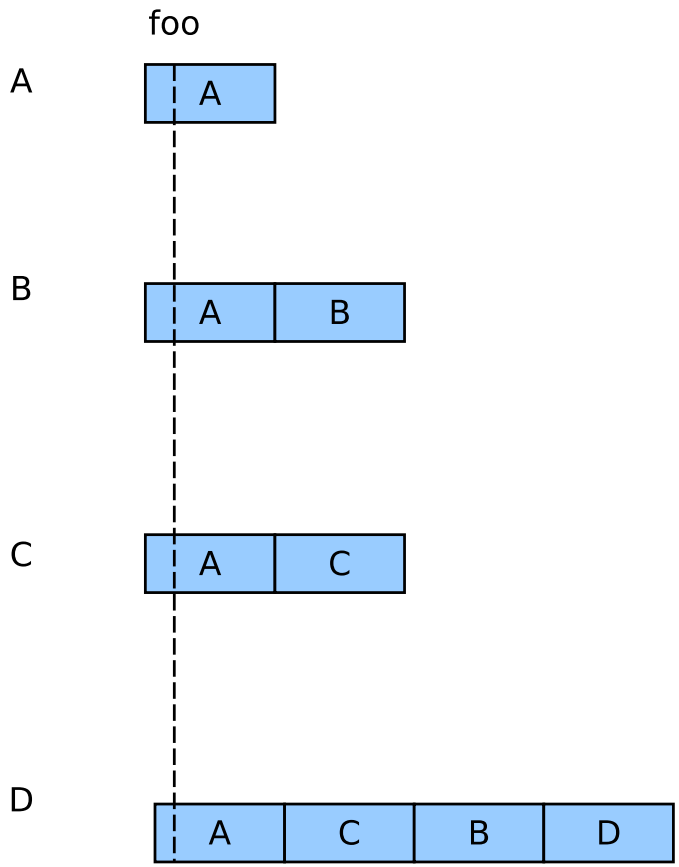




Chargement dynamique

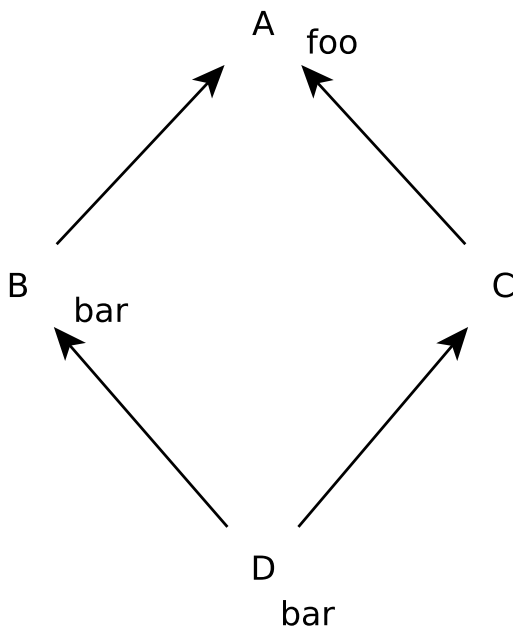
En monde ouvert (chargement dynamique) :

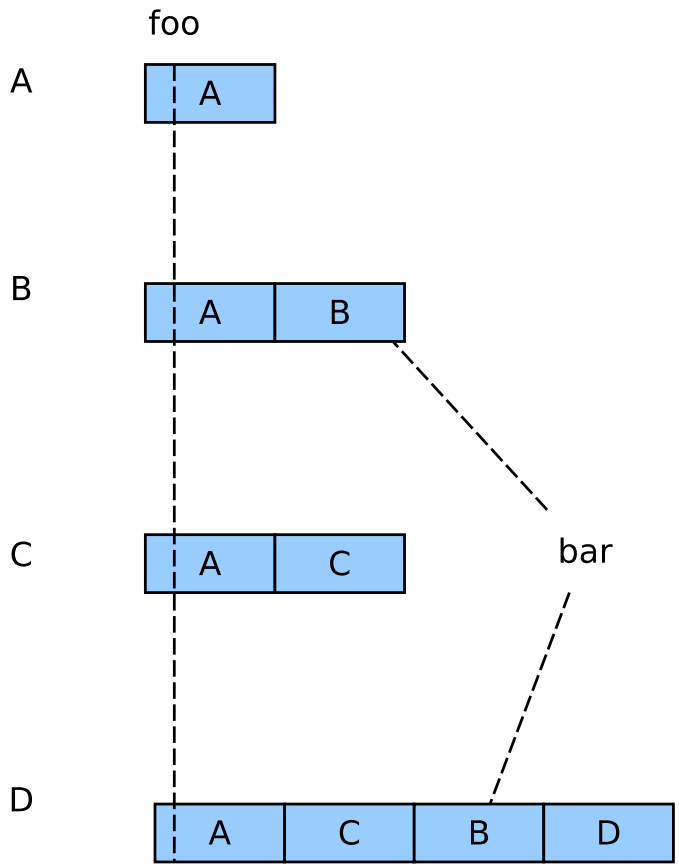




Chargement dynamique

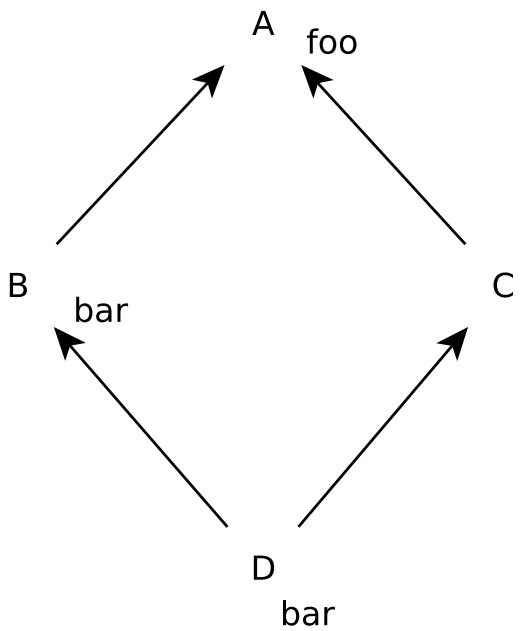
En monde ouvert (chargement dynamique) :

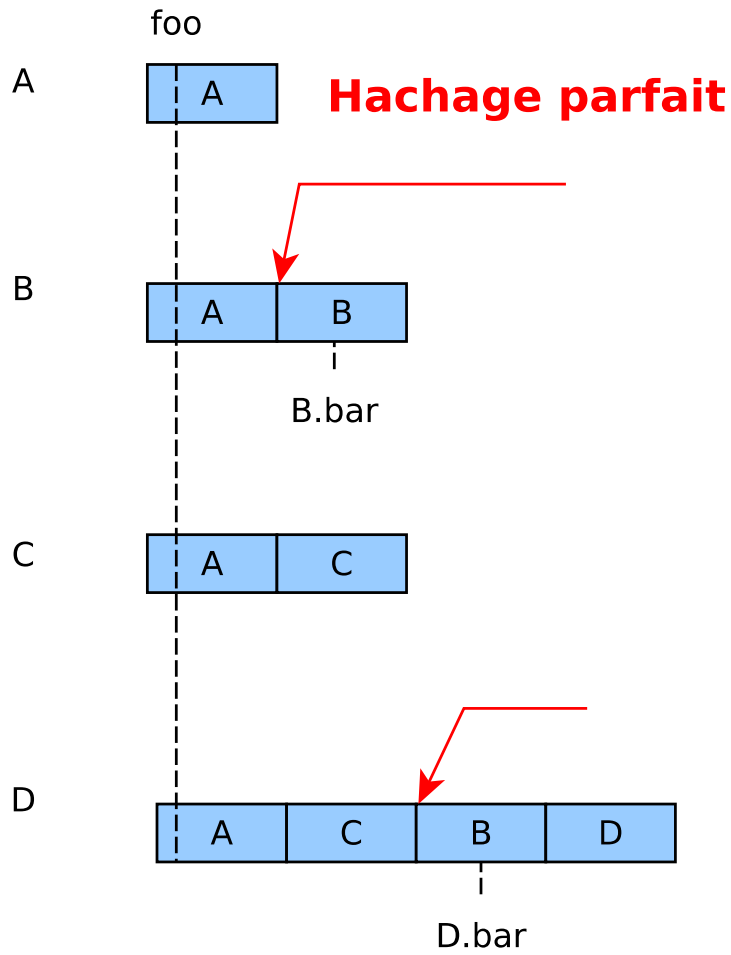




Hachage parfait

Solution compatible : le hachage parfait





Implémentations utilisées

Implémentations

Du moins efficace au plus efficace :

- Hachage parfait (héritage multiple) : marche dans tous les cas
- Héritage simple (*Single-Subtyping*, SST) : une seule position
- Statique : une seule méthode candidate
- *Inlining*

Comment faire mieux que le hachage parfait en monde ouvert ?

Démarche

Contexte et outils pour la machine virtuelle

- Le langage Nit [Privat, 2008]
- Développée à partir de l'interpréteur Nit
- Simulation de compilation à la volée du code

Le langage Nit

Caractéristiques du langage

- Tout est objet
- Héritage multiple
- Typage statique
- Généricité
- Modules, raffinement de classes

Une machine virtuelle à partir d'un interpréteur

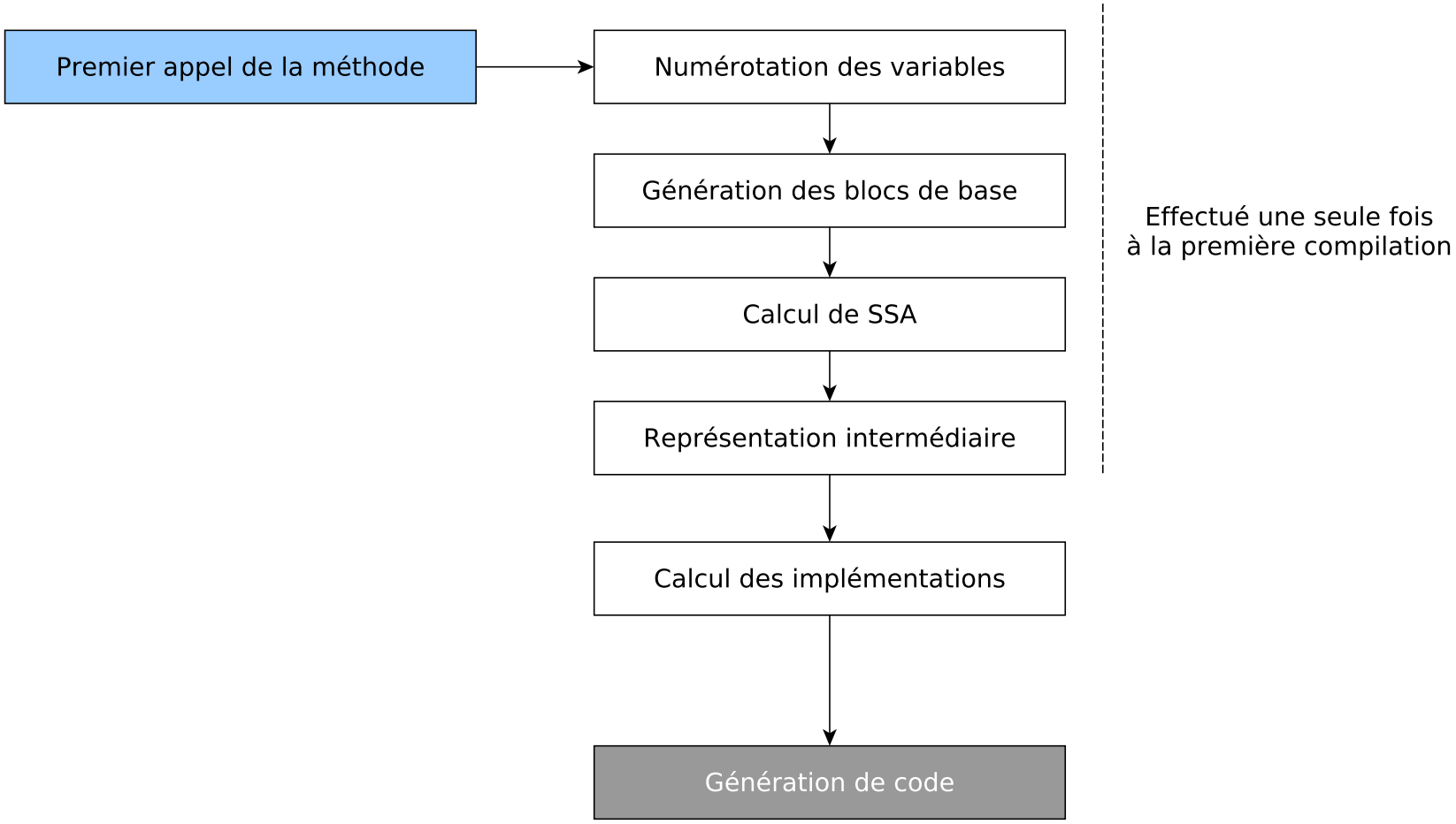
L'interpréteur Nit existant

- Fonctionne en **monde fermé**
- Pas optimisé du tout, son code est simple et réutilisable
- Interprète les programmes à partir de l'arbre syntaxique du programme (AST) décoré par le méta-modèle

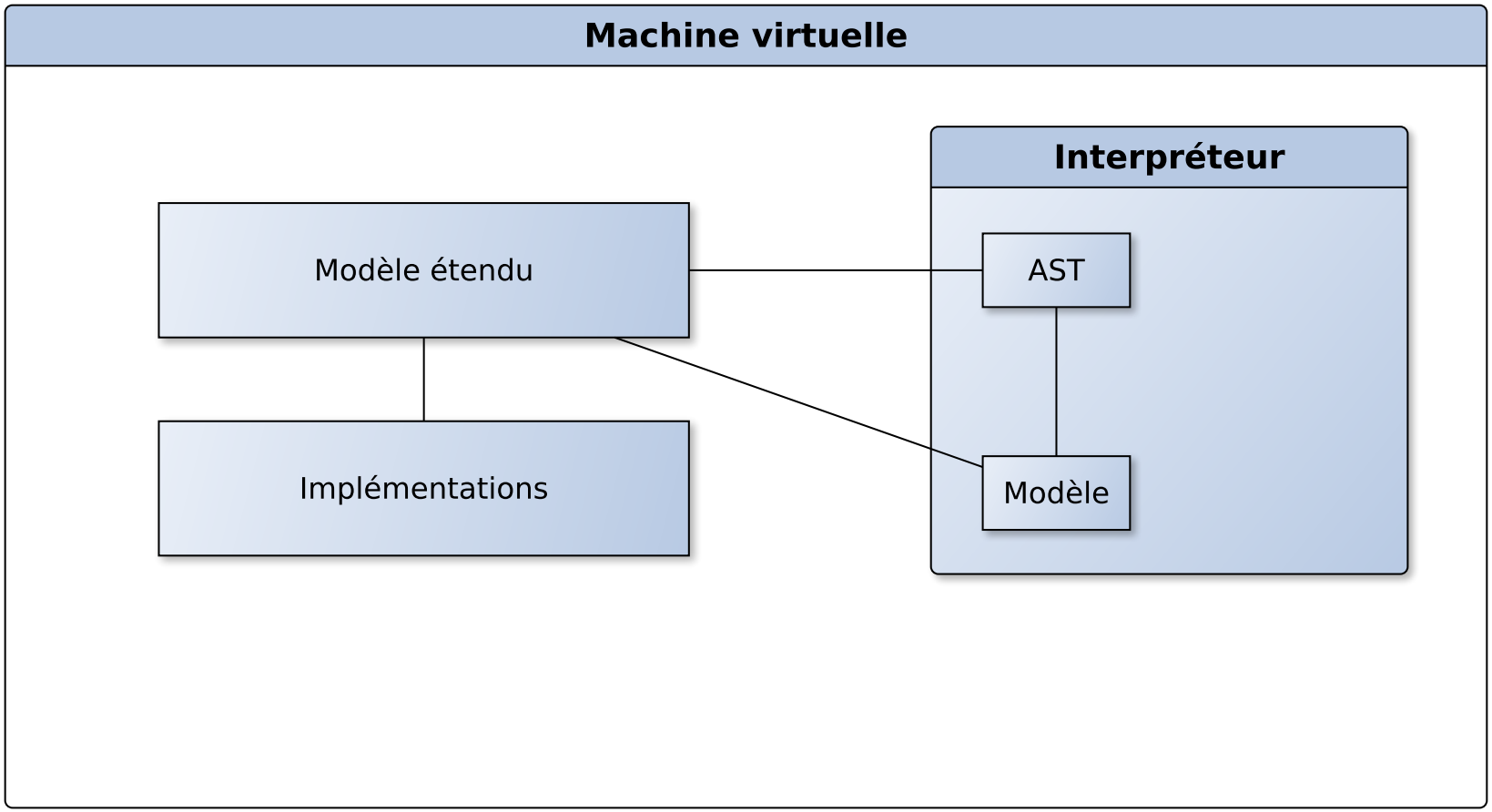
Méthodologie

- Développement par spécialisation/raffinement de l'interpréteur
- Ce qui n'est pas raffiné fonctionne encore
- Implémentation en C des structures de bas-niveau

Simulation de la compilation à la volée



Architecture globale de la machine virtuelle



Définition

Un protocole est un algorithme accompagné d'outils, contenant des éléments des trois catégories :

1. Collecte d'informations : modèle du programme, analyses statiques, profilage de l'exécution
2. Optimisations : SST, appels statiques (dévirtualisation), inlining
3. Recompilations et réparations

Problématique

Optimisations spéculatives/agressives

- Choix d'implémentations plus efficaces que l'implémentation conservatrice
- Ces choix sont valides lors de la compilation, mais peuvent être invalidés

Problématique

Optimisations spéculatives/agressives

- Choix d'implémentations plus efficaces que l'implémentation conservatrice
- Ces choix sont valides lors de la compilation, mais peuvent être invalidés

les dé-optimisations sont nécessaires, et provoquent des réparations/recompilations.

Problématique

Les recompilations sont insuffisantes si la méthode est active

Mécanismes de réparation :

- Gardes
- Code-patching
- On-stack replacement
- *Préexistence*

Objectif : Trouver un protocole efficace, et sans trop de réparations/recompilations

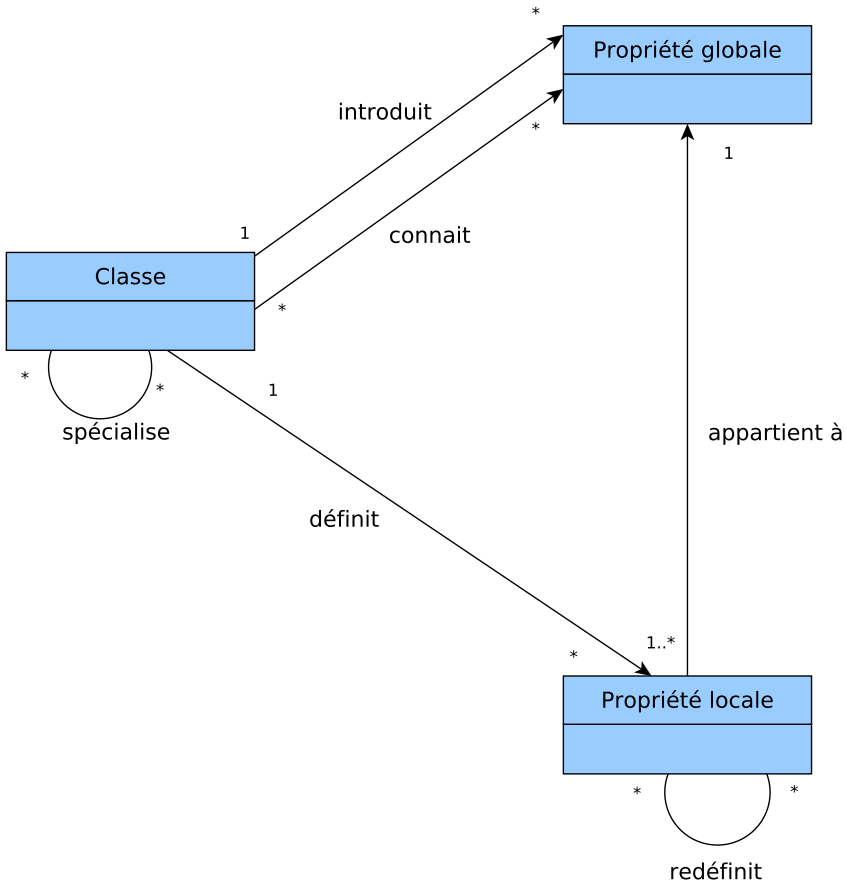
Modèle étendu de la machine virtuelle

Modèle et représentation intermédiaire

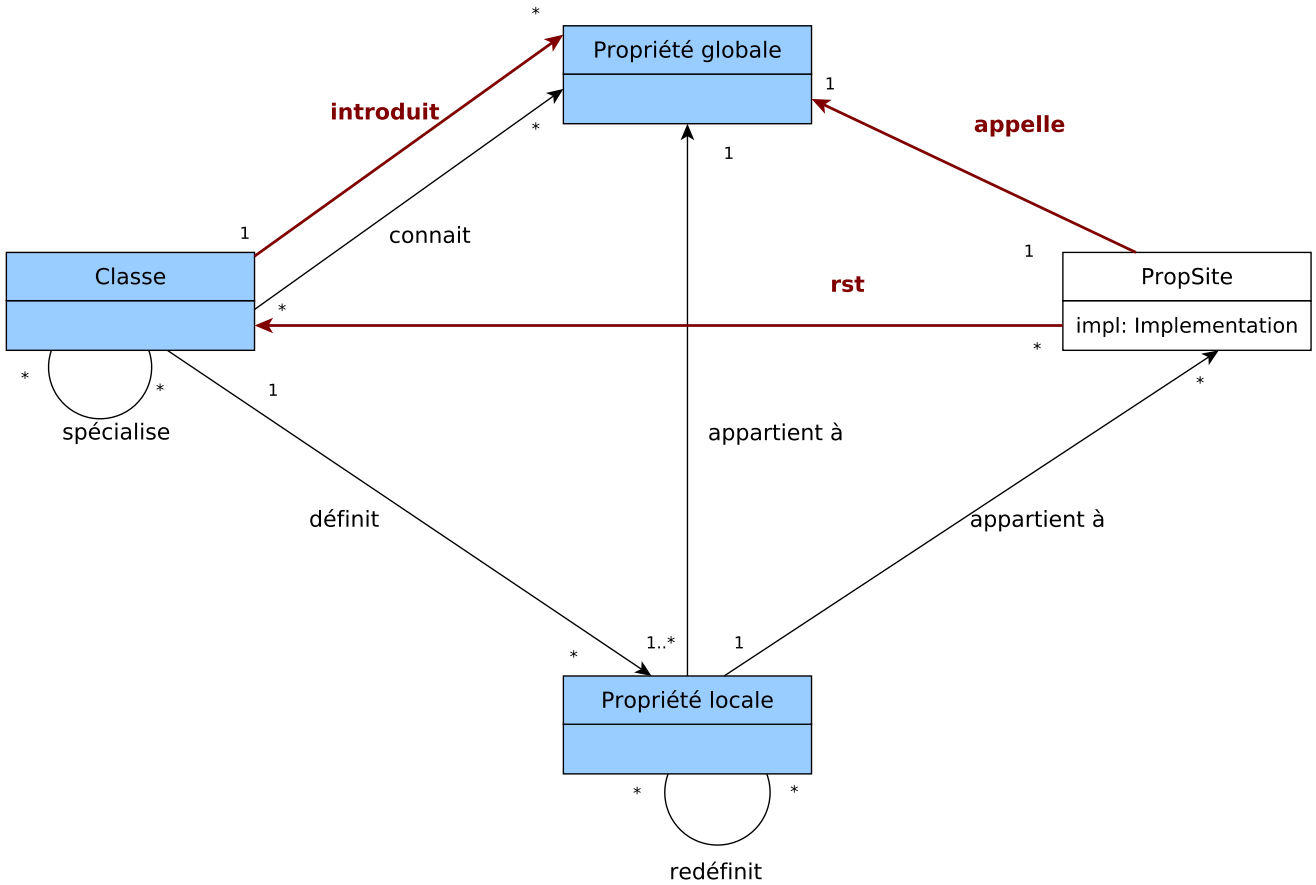
- Générés paresseusement à partir de l'AST décoré par le modèle
- Et de l'analyse de dépendances

Chaque nœud d'AST d'un mécanisme objet est représenté par son entité du modèle

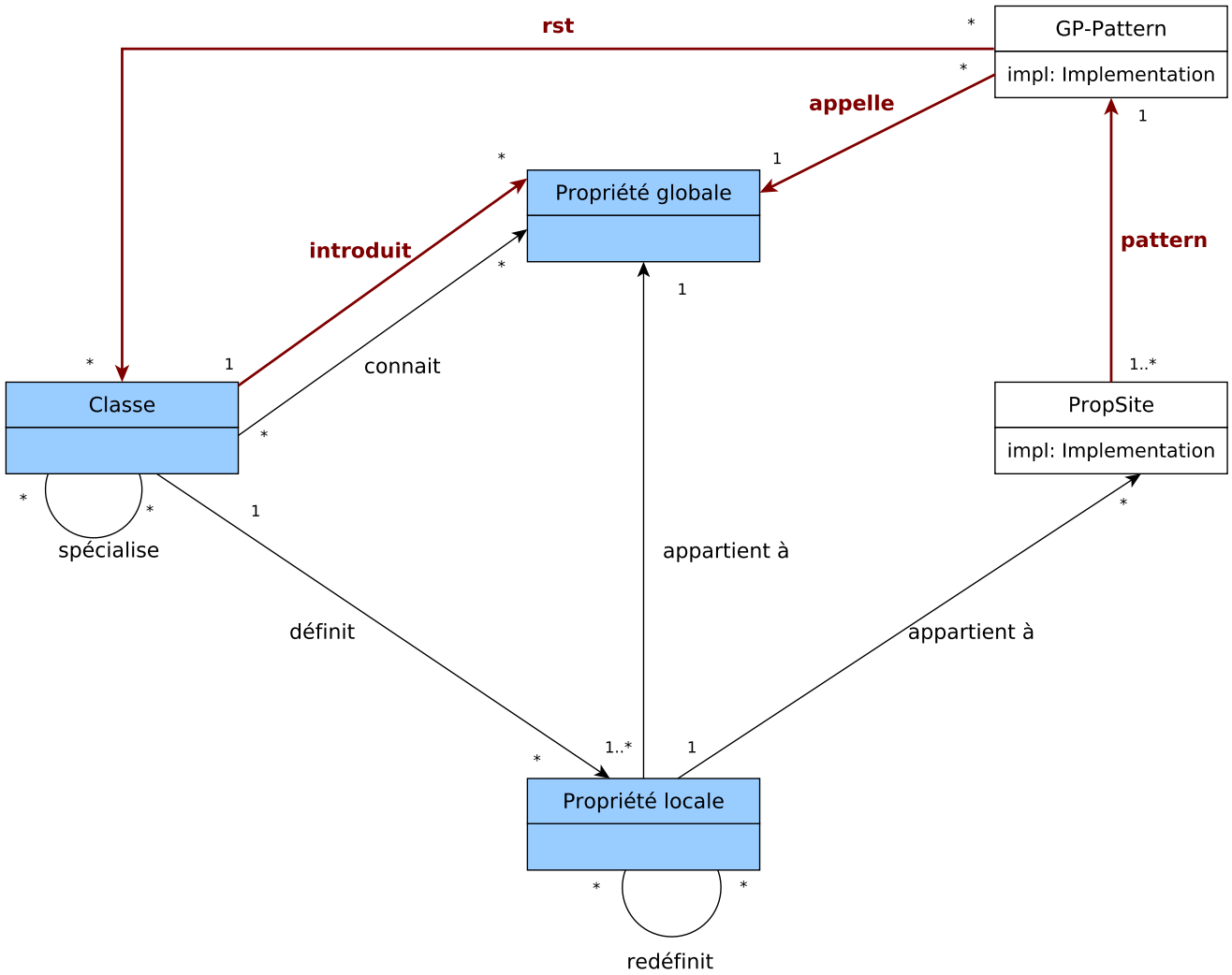
Méta-modèle de base de Nit



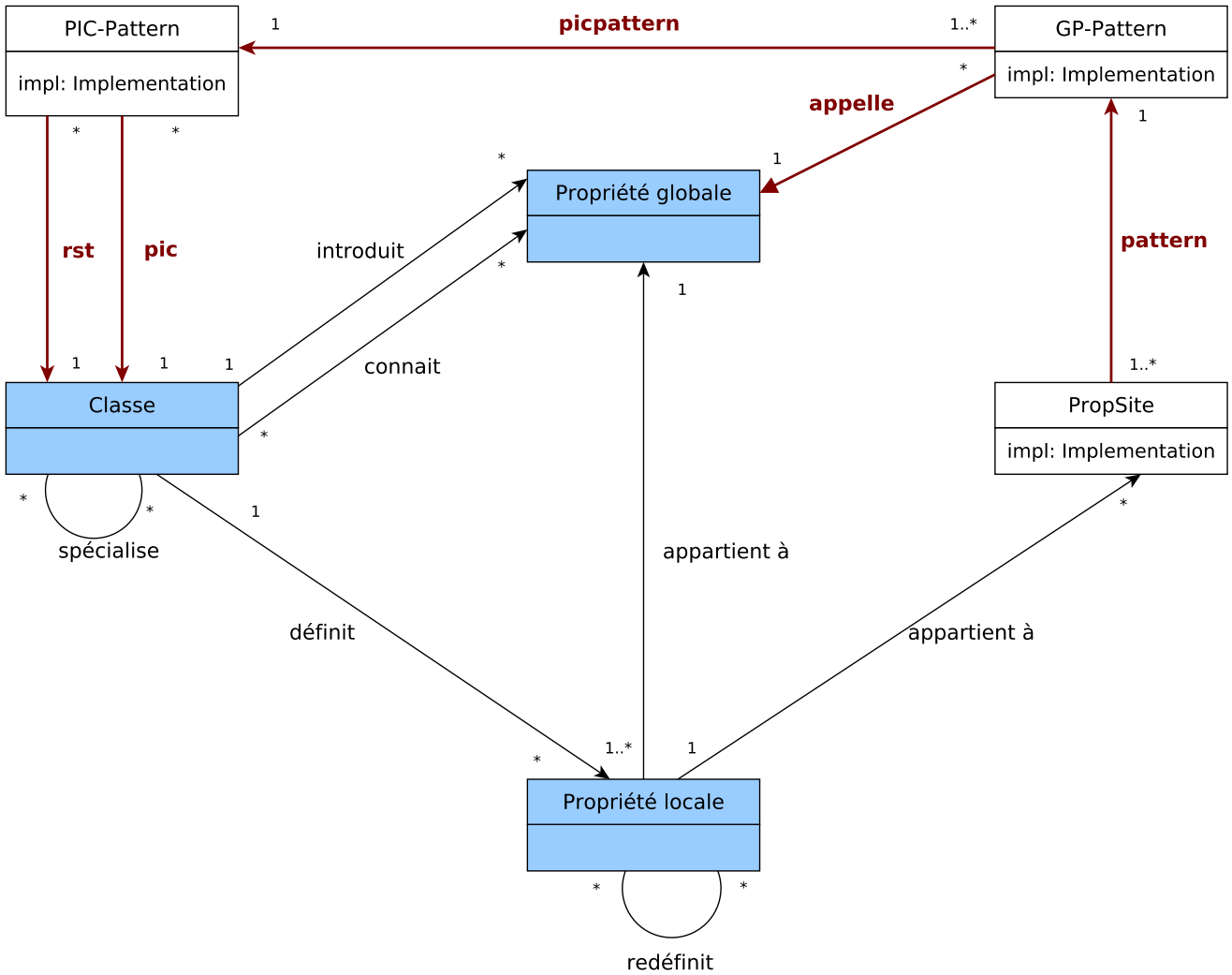
Site-Objets dans le modèle étendu



Patterns dans le modèle étendu



PIC-Patterns dans le modèle étendu



Calcul des implémentations optimistes

PIC-Pattern

- SST si position unique du *pic* dans les sous-classes du *rst*
- PH sinon

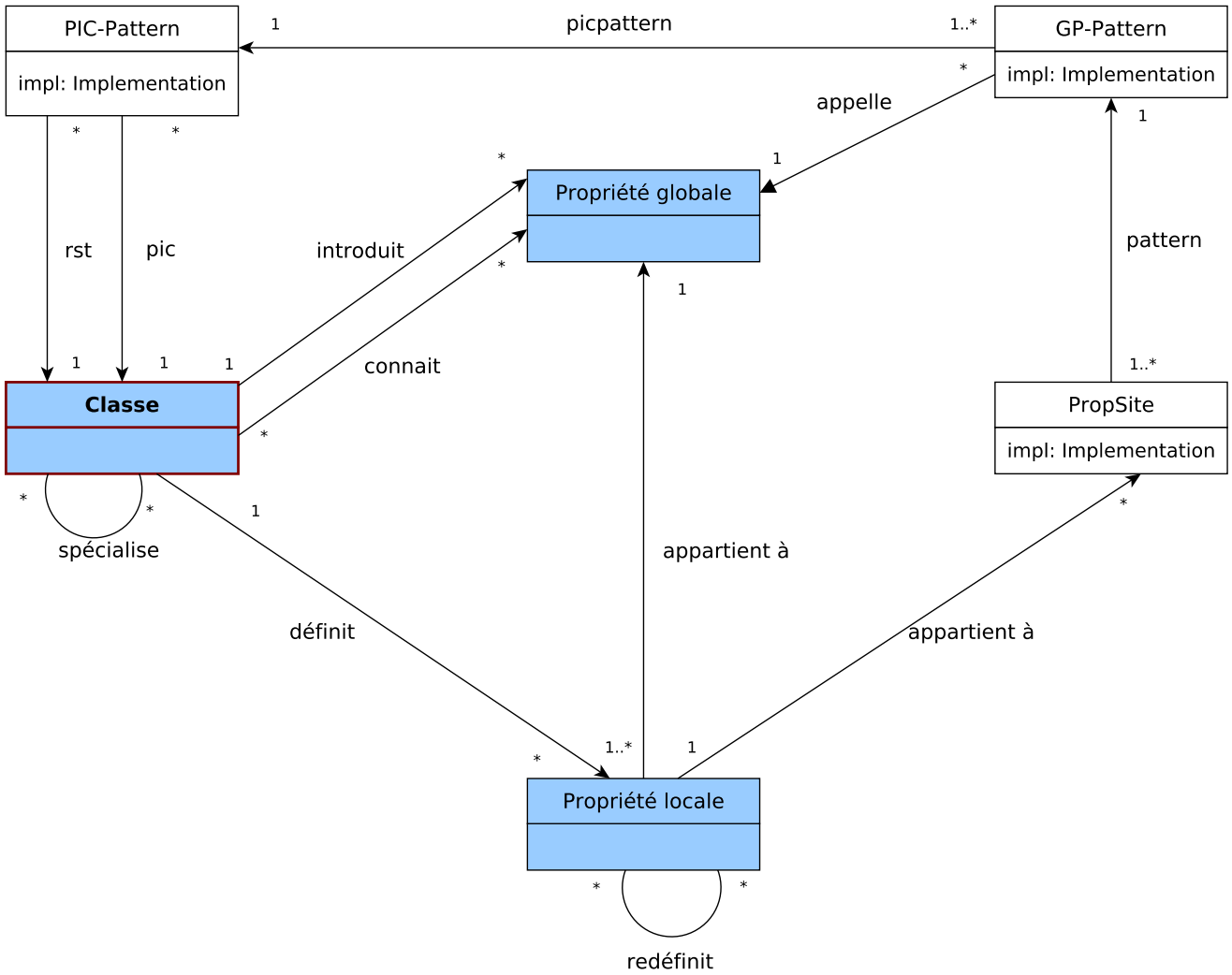
GP-Pattern

- Statique si une seule méthode est candidate
- Implémentation du PIC-Pattern sinon

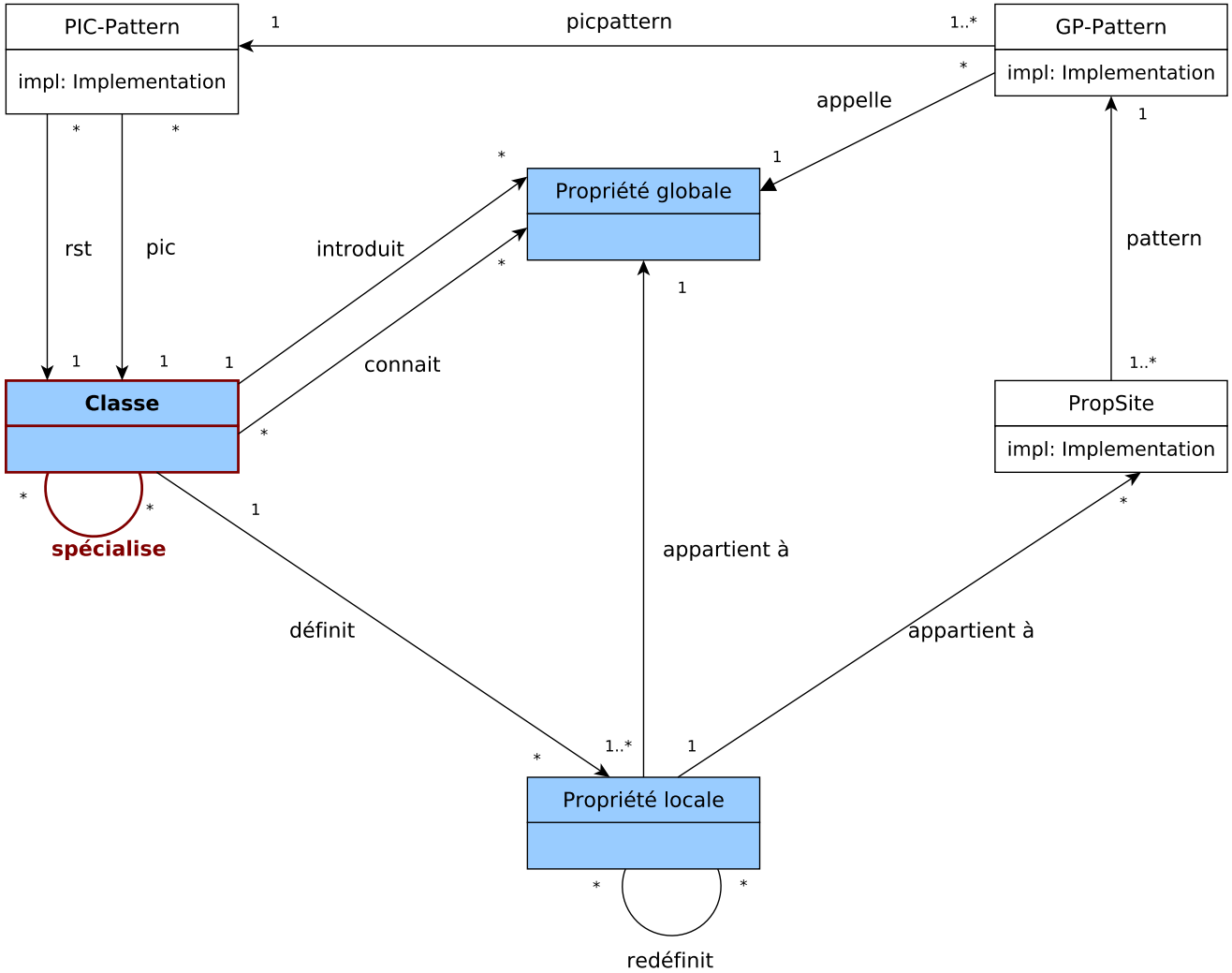
Site-objet

- Par défaut, implémentation du GP-Pattern

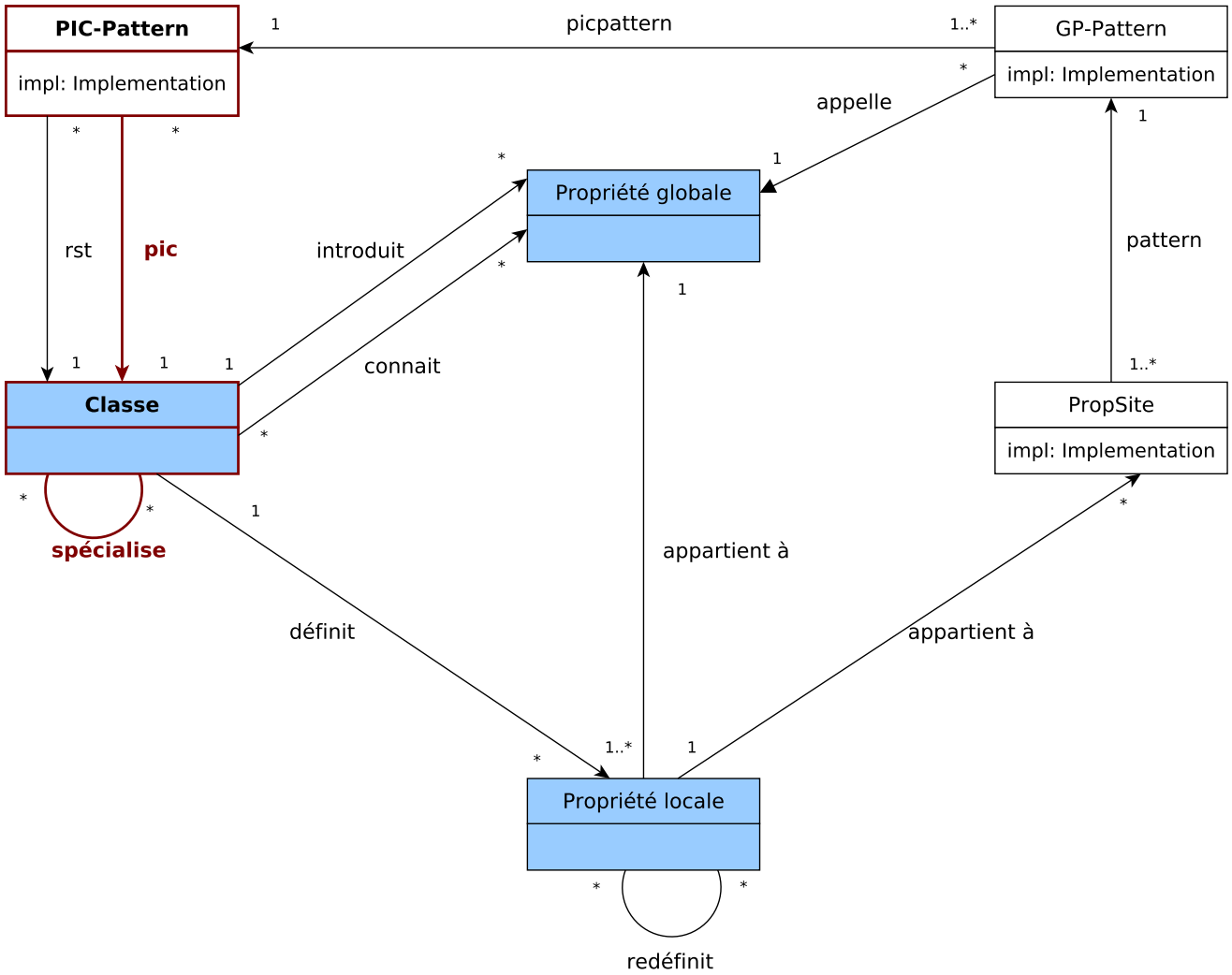
Propagation : chargement d'une classe



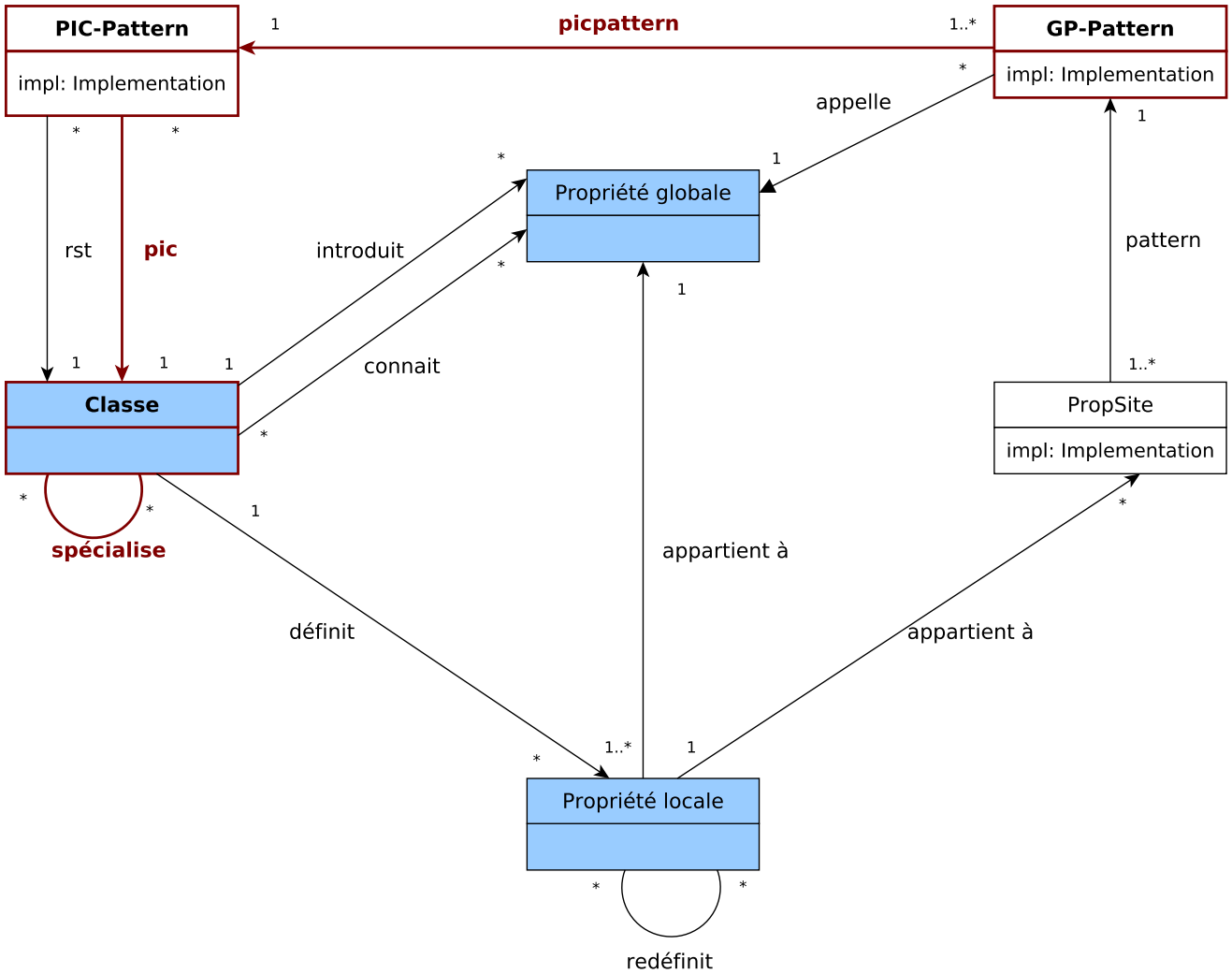
Propagation : chargement d'une classe



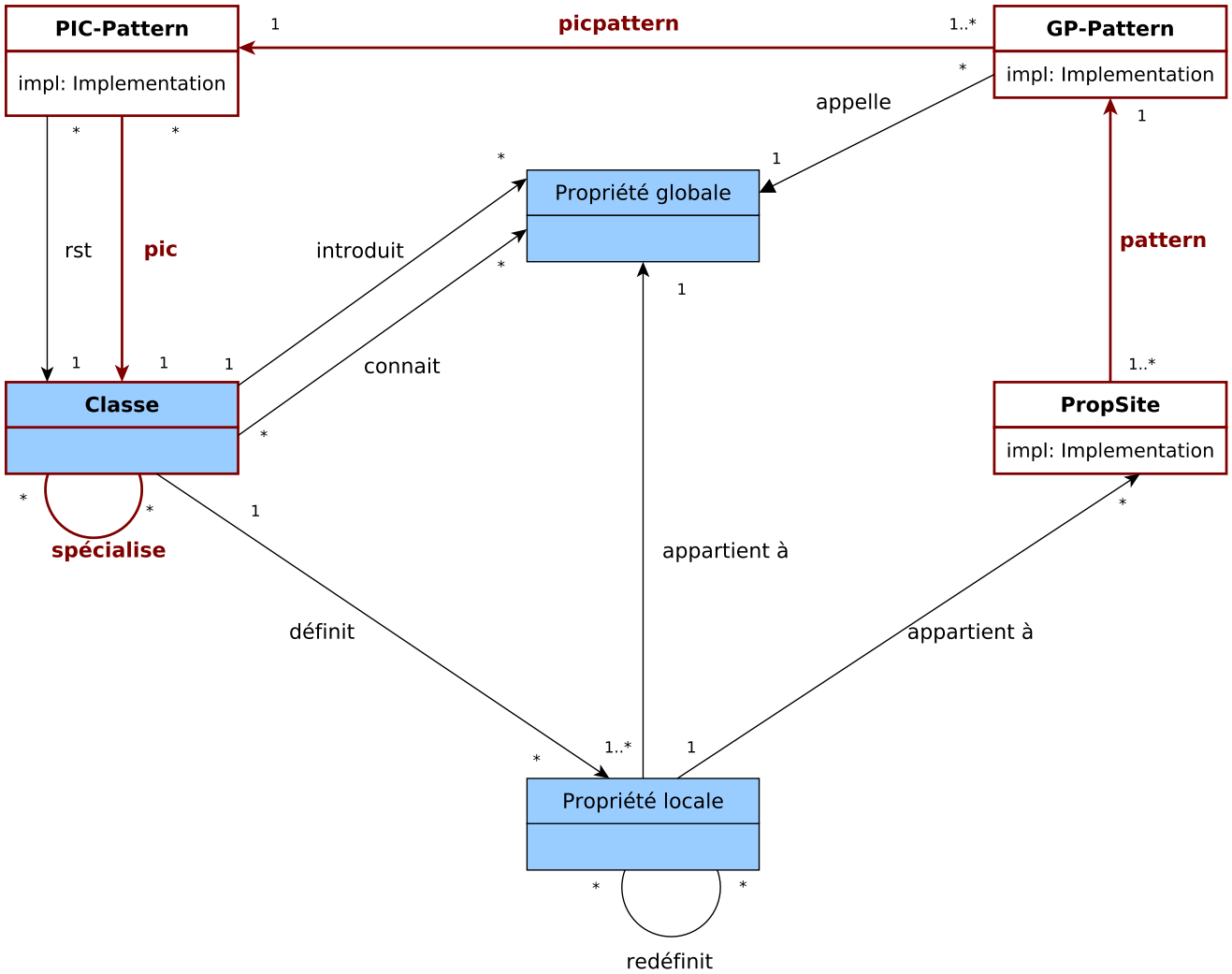
Propagation : chargement d'une classe



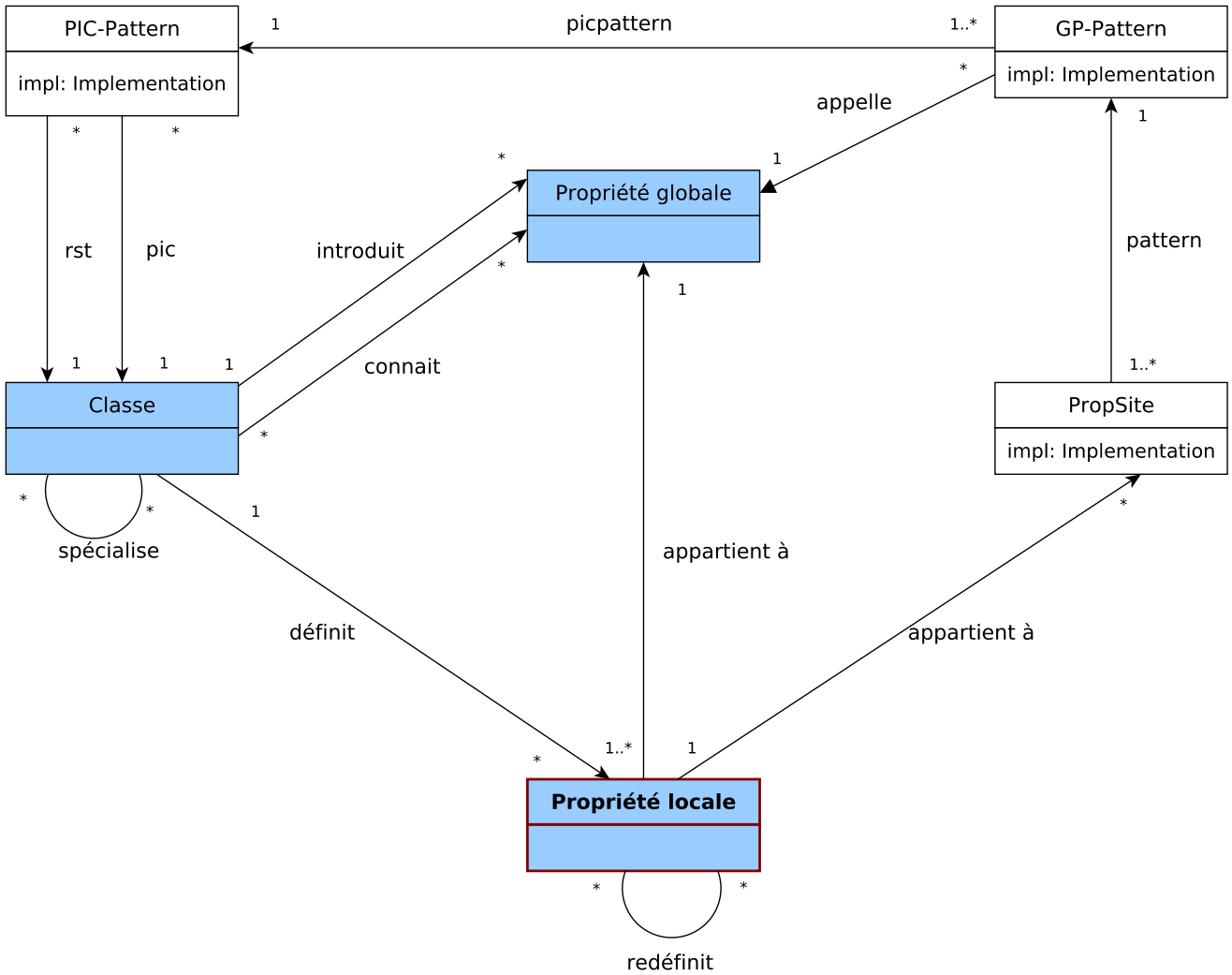
Propagation : chargement d'une classe



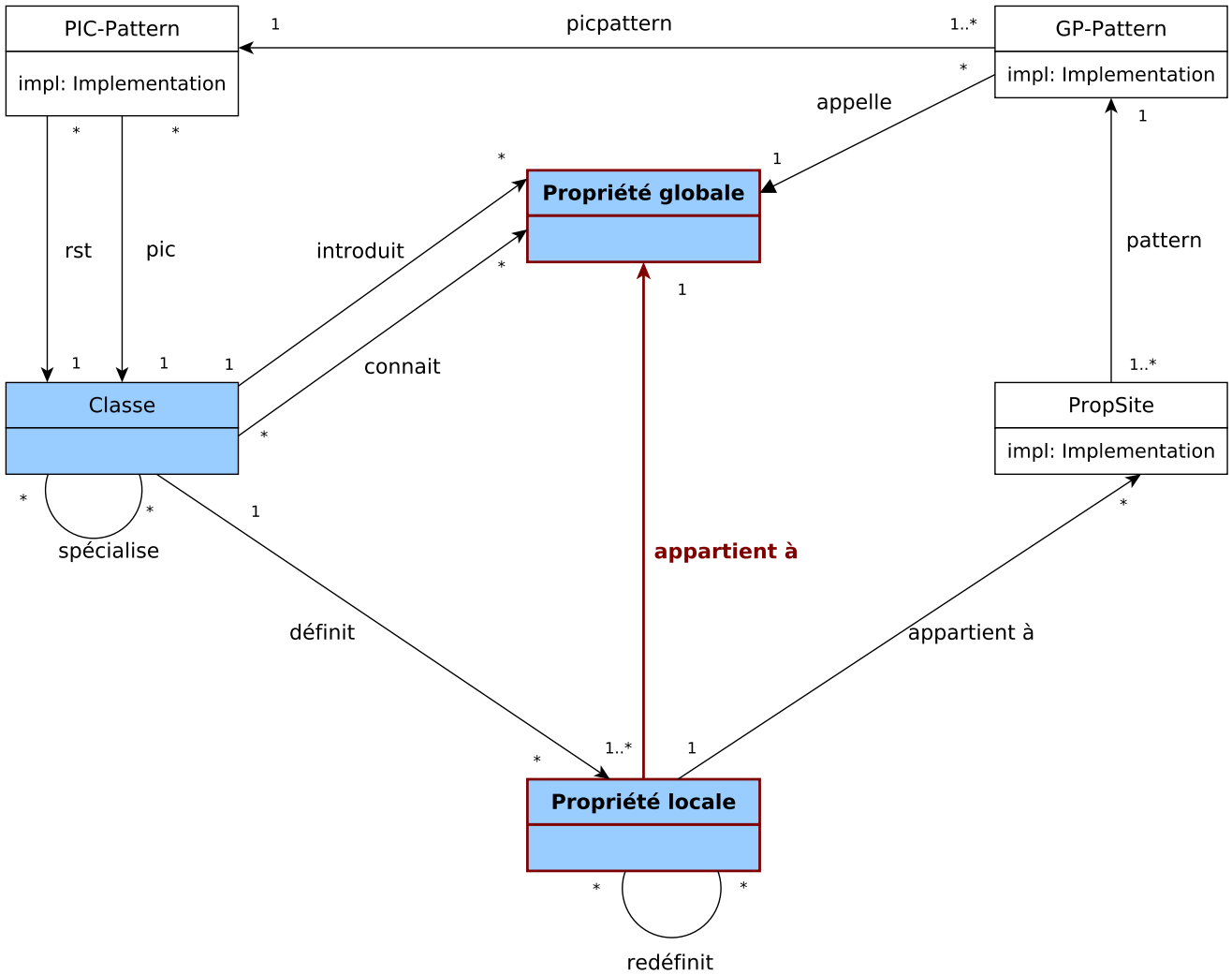
Propagation : chargement d'une classe



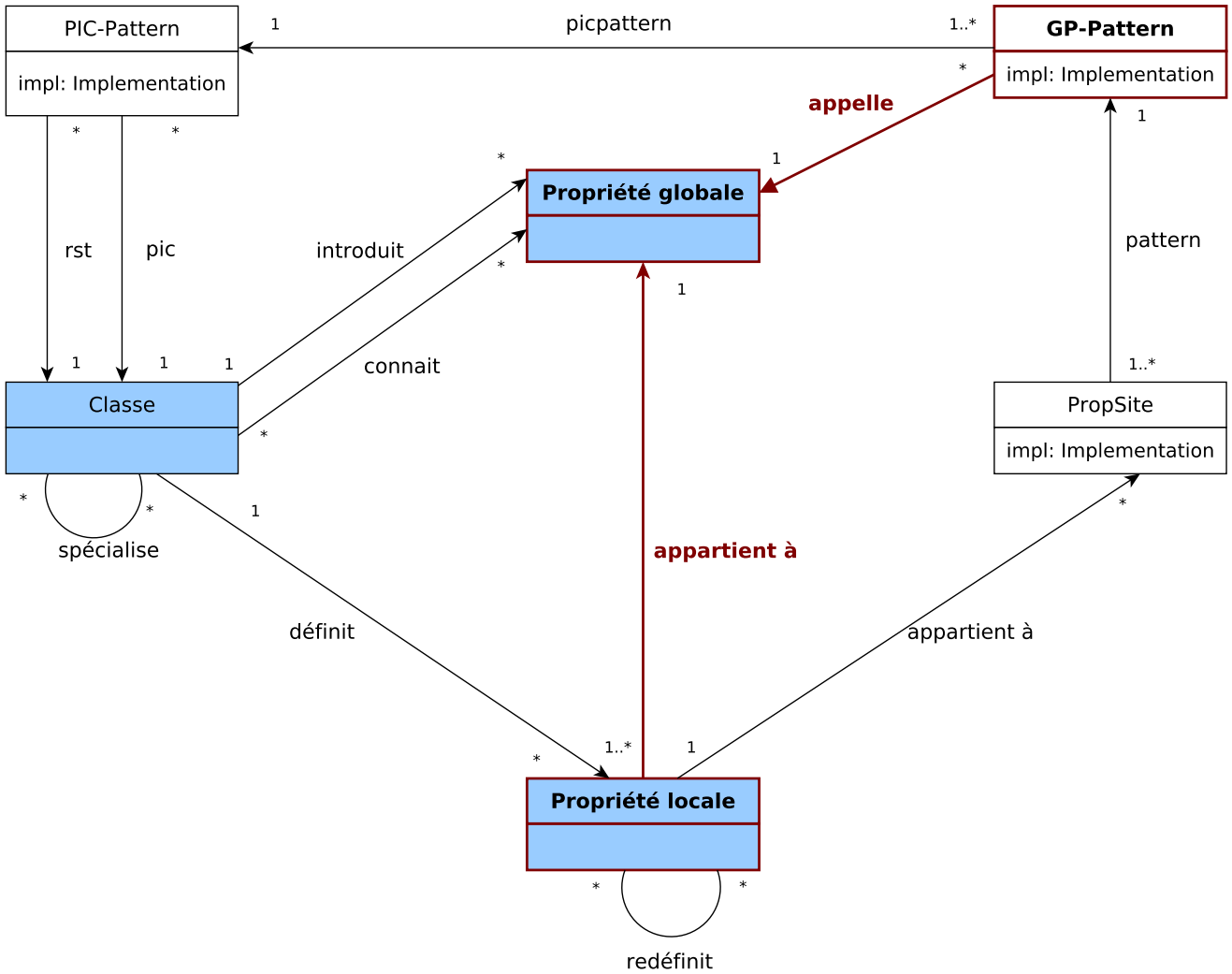
Propagation : nouvelle propriété locale



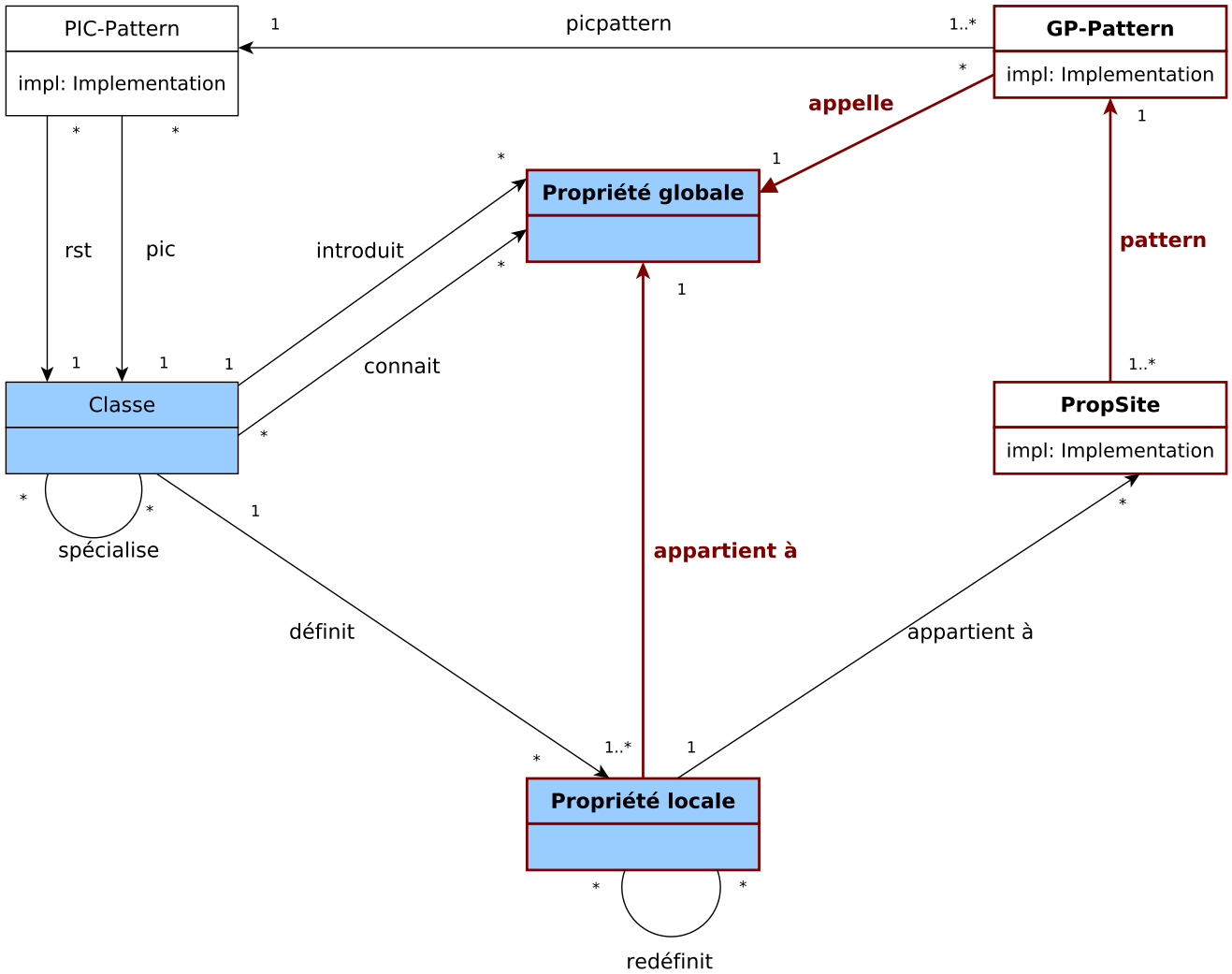
Propagation : nouvelle propriété locale



Propagation : nouvelle propriété locale



Propagation : nouvelle propriété locale



Préexistence

Propriété introduite par Detlefs et Agesen pour éviter les réparations à chaud

Définition

Dans une méthode, un receveur est préexistant s'il est garanti avoir été créé **avant** l'entrée dans la méthode.

Propriété

La préexistence du receveur assure que les éventuels chargements de classe provoqués par l'exécution courante ne provoqueront pas de réparation.

Exemple

```
fun foo(x: A)
do
    f() // Provoque le chargement de B
    x.bar() // x est préexistant
end
```

`x.bar()` peut être dévirtualisé et inliné sans gardes ou patchs.

Règles de la préexistence originelle

Règles de préexistence

- **Parameter-P**: un paramètre est toujours préexistant.
- **ImmutableAttribute-P**: une lecture d'attribut immutable est préexistante si son receveur l'est.

En pratique, la seconde règle a un effet très marginal.

Limites de la préexistence originelle

```
foo (x: A)
do
    var y: A
    if <condition> then
        y = new B()
        y.bar1()
    else
        y = x
        y.bar2()
    end
    y.bar3()
end
```

Limites de la préexistence originelle

```
foo (x: A)
do
    var y: A
    if <condition> then
        y = new B()
        y.bar1() // Préexistence de type si B est déjà chargée
    else
        y = x
        y.bar2()
    end
    y.bar3()
end
```

Limites de la préexistence originelle

```
foo (x: A)
do
    var y: A
    if <condition> then
        y = new B()
        y.bar1() // Préexistence de type si B est déjà chargée
    else
        y = x
        y.bar2() // Préexistence de valeur
    end
    y.bar3()
end
```

Limites de la préexistence originelle

```
foo (x: A)
do
    var y: A
    if <condition> then
        y = new B()
        y.bar1() // Préexistence de type si B est déjà chargée
    else
        y = x
        y.bar2() // Préexistence de valeur
    end
    y.bar3() // Préexistence de type
end
```

Préexistence étendue

Étendre la définition de la préexistence à :

- Toutes les expressions (pas uniquement les receveurs)
- Tous les mécanismes objet (pas seulement les appels de méthodes)
- Préexistence de type (pas seulement de valeur)

Dépendances des variables

- **Variable-P**: Une variable est préexistante si toutes les expressions dont elle dépend sont préexistantes.
- **Cast-P**: Un cast est préexistant si le receveur l'est.

Préexistence étendue

Préexistence de valeur

La **valeur** a été créée **avant** l'entrée dans la méthode.

Préexistence de type

Le **type dynamique** a été chargé **avant** l'entrée dans la méthode.

La préexistence de valeur implique la préexistence de type.

Type concret et préexistence

Type concret

Le type concret d'une expression est l'ensemble de ses types dynamiques possibles, si cet ensemble est connu statiquement.

Règles principales des types concrets

- **ConcreteType-P** : une expression est préexistante si toutes les classes du type concret sont chargées.
- **FinalType-CT** : quand le type statique d'une expression est final, le type concret est ce type final.
- **New-CT** : une instanciation a le type concret du *new*.
- **PrivateWrite-CT** : le type concret est l'union de toutes les affectations dans l'attribut privé.

Type concret et préexistence

Type concret

Le type concret d'une expression est l'ensemble de ses types dynamiques possibles, si cet ensemble est connu statiquement.

Règles principales des types concrets

- **ConcreteType-P** : une expression est préexistante si toutes les classes du type concret sont chargées.
- **FinalType-CT** : quand le type statique d'une expression est final, le type concret est ce type final.
- **New-CT** : une instanciation a le type concret du *new*.
- **PrivateWrite-CT** : le type concret est l'union de toutes les affectations dans l'attribut privé.

Amélioration de l'implémentation des sites.

Préexistence étendue

Extension à l'inter-procédural

- **Return-P**: Le retour d'une méthode a la préexistence de sa variable de retour
- **Call-P**: une expression d'invocation de méthode est préexistante si :
  1. Son receveur et ses arguments sont tous préexistants
  2. Les valeurs retournées de toutes les méthodes candidates à l'appel sont préexistantes
- **Return-CT**, **Call-CT**: similaires mais avec des types concrets

Préexistence étendue

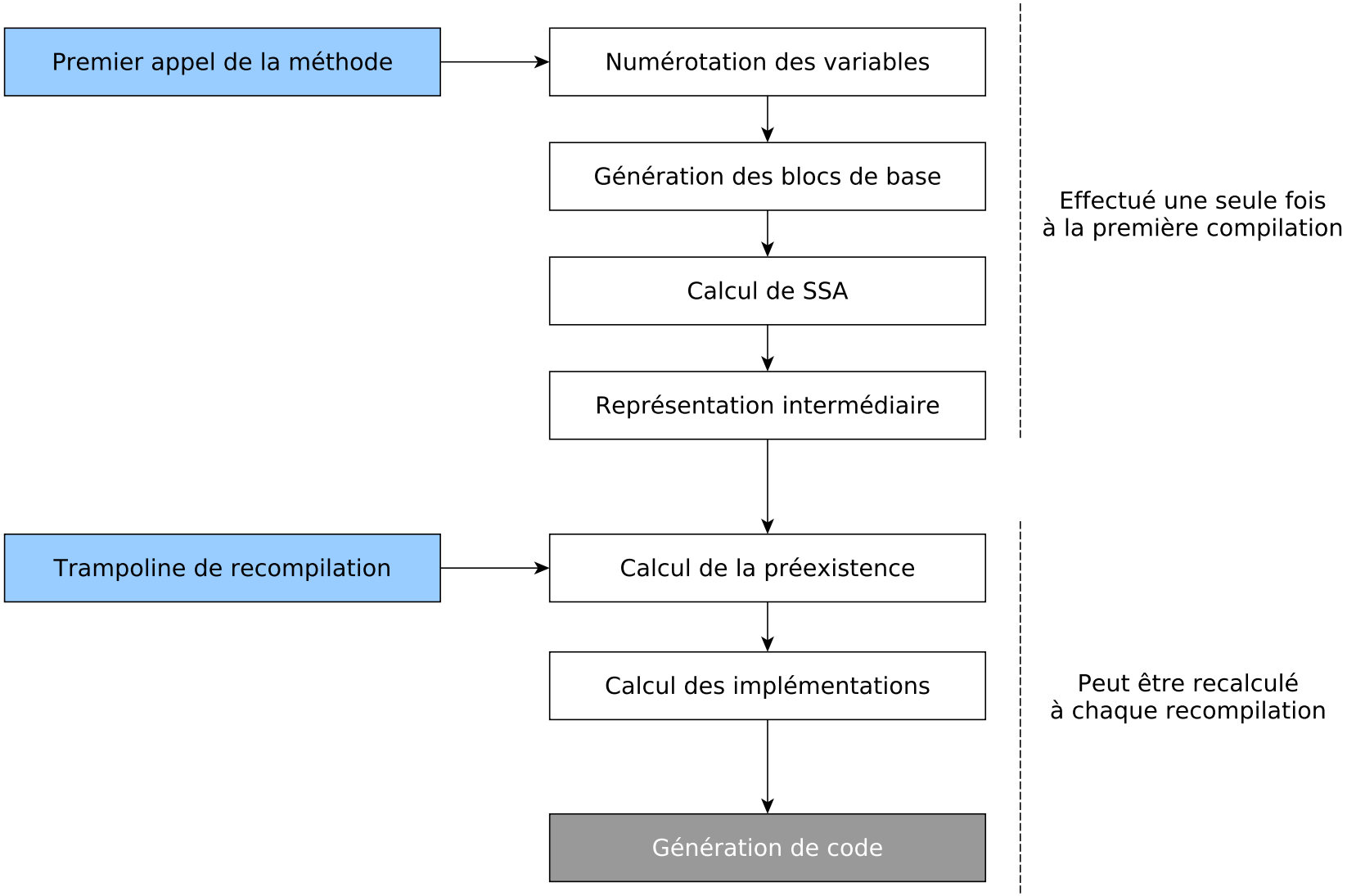
Extension à l'inter-procédural

- **Return-P**: Le retour d'une méthode a la préexistence de sa variable de retour
- **Call-P**: une expression d'invocation de méthode est préexistante si :
  1. Son receveur et ses arguments sont tous préexistants
  2. Les valeurs retournées de toutes les méthodes candidates à l'appel sont préexistantes
- **Return-CT**, **Call-CT**: similaires mais avec des types concrets

Les règles **Call-P** et **Call-CT** introduisent de la mutabilité dans la préexistence de ces expressions.

[Preexistence Revisited, ICOOOLPS, 2015]
[Preexistence And Concrete Type Analysis, PPPJ, 2016]

Simulation de la compilation à la volée



Principe des expérimentations

Les benchmarks

Des programmes Nit :

- Le compilateur statique, l'interpréteur, des outils divers

Statistiques et mesures

Le principe : comptage d'éléments à l'exécution

1. Compteurs statiques, nombre de sites suivant :
   - Implémentations
   - Préexistence
2. Compteurs dynamiques
   - Sites exécutés suivant leurs implémentations
   - Nombre de recompilations (de sites, de méthodes)

Trois protocoles implémentés

1. **Pur code-patching** : chaque méthode est compilée paresseusement avec des implémentations optimistes. Les changements d'implémentation sont propagés.
2. **Pur préexistence** : seuls les sites préexistants sont optimisés. Recompilation totale de la méthode au premier appel suivant.
3. **Mixte** : code-patching pour les appels de méthode et protocole de préexistence pour les attributs et les casts.

Décompte statique des méthodes

Benchmark	Original	Étendu	Amélioration	Total
nitc	4911	6101	24%	10184
niti	3682	4381	19%	7918
nitdoc	1723	2397	39%	4525
jwrapper	2061	2356	14%	2950
nitiwiki	267	391	46%	680
Total	12644	15626	24%	26228

Appels de méthodes polymorphes préexistants statiques dans le protocole pure-préexistence

Décompte statique des attributs

Benchmark	Original	Étendu	Amélioration	Total
nitc	3439	3726	8%	5436
niti	2274	2428	7%	3494
nitdoc	2293	2456	7%	3463
jwrapper	912	958	5%	1083
nitiwiki	363	379	4%	466
Total	9281	9947	7%	13942

Accès aux attributs polymorphes préexistants en SST dans le protocole pure-préexistence

Décompte dynamique

	Méthodes	Attributs	Casts	Total
monomorphes	33994	1835	0	35829
statique	62637	0	663	63301
SST	42757	63477	3625	109860
PH	21996	3372	752	26121
total	161386	68685	5041	235112

Nombre d'exécutions en milliers des sites objets dans le protocole préexistence dans *nitc*

- 14% de PH pour les méthodes
- 5% de PH pour les attributs

Contributions

Contributions

- Réalisation d'une machine virtuelle pour un langage en héritage multiple et en typage statique
- Simulation d'un compilateur à la volée
- Extension de l'analyse de préexistence associée à une analyse de types concrets
- Spécification de protocoles de compilation/recompilation basés sur la préexistence ou des patchs de code
- Expérimentations de ces protocoles par comptage d'éléments

Conclusion

Quelques conclusions :

1. La **préexistence étendue** augmente les opportunités d'optimisations
   - Bonne base pour les protocoles
   - Applicable à tous les langages similaires à Java

2. Une machine virtuelle en **héritage multiple**
   - Faible surcoût de l'héritage multiple

Perspectives

À court terme

- Plus de benchmarks
- Mesures plus fines du coût des protocoles
- Étudier les interactions entre la préexistence et l'inlining

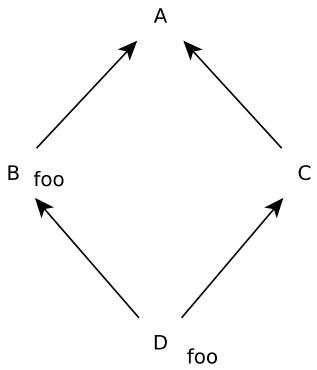
À moyen terme

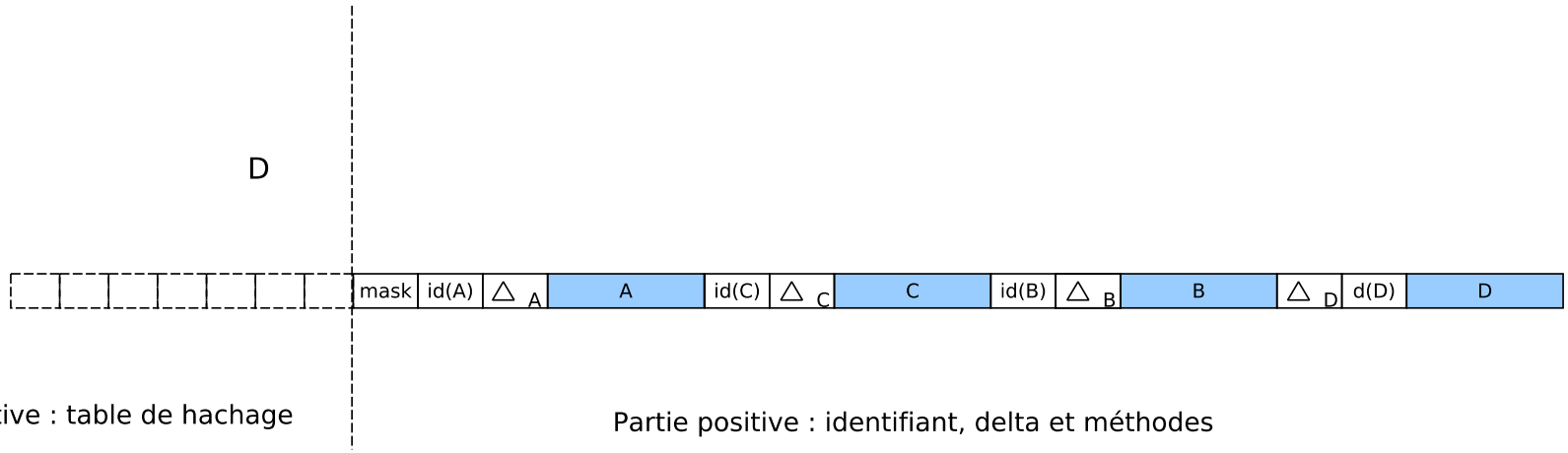
- Ajouter l'étape de la production de code à la machine virtuelle
- Intégrer d'autres optimisations, pas forcément liées à l'objet
- Étudier la préexistence étendue dans d'autres langages

Merci pour votre attention

Utilisation du hachage parfait

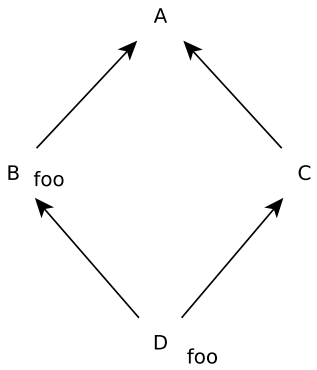
B x = new D()
x.foo()

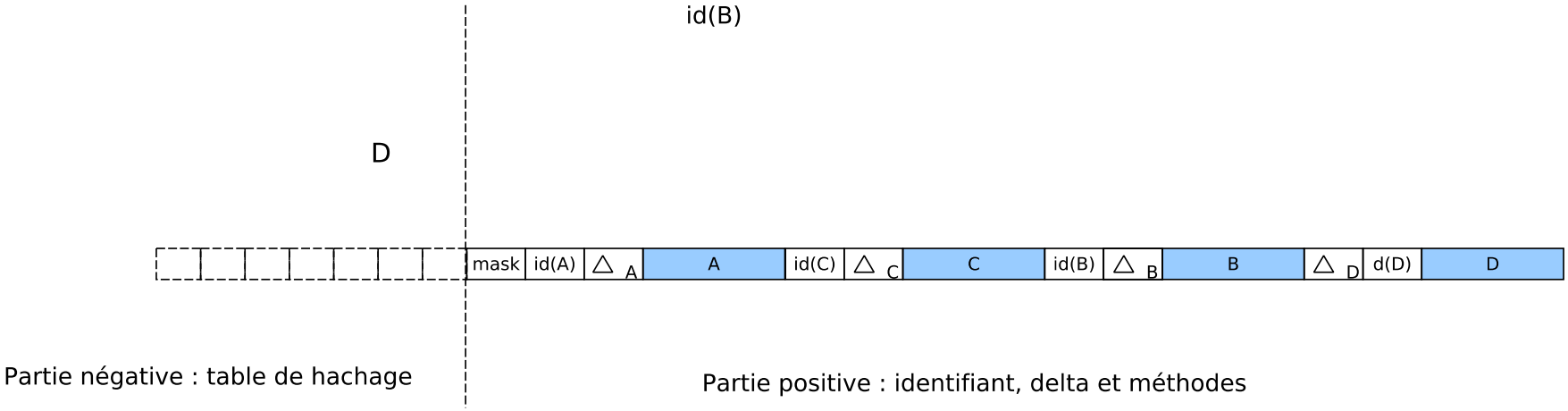




Utilisation du hachage parfait

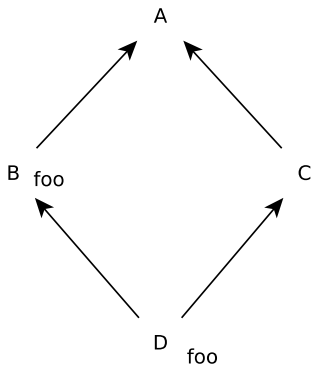
B x = new D()
x.foo()

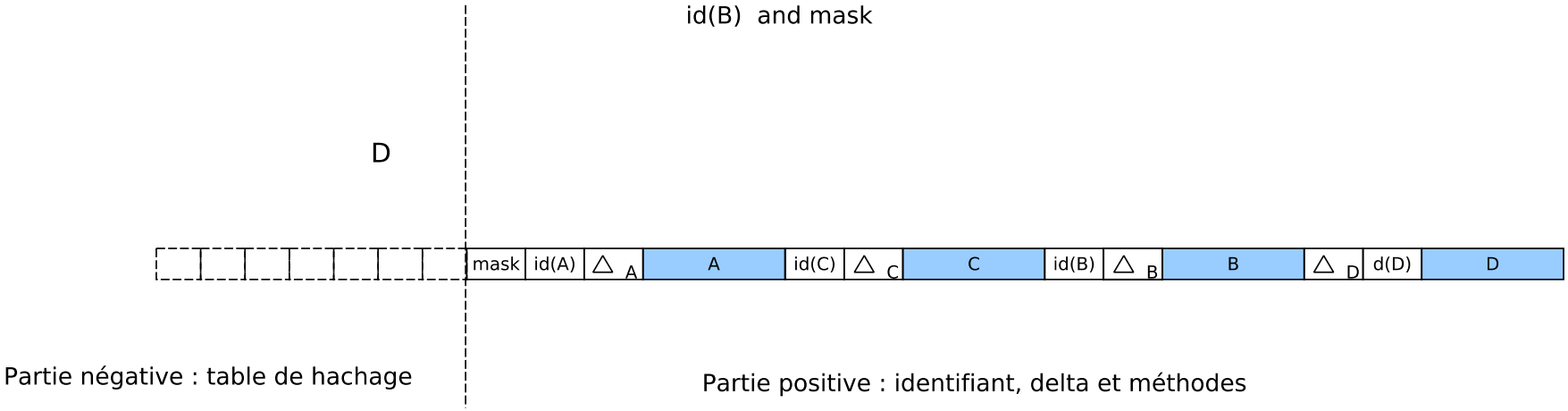




Utilisation du hachage parfait

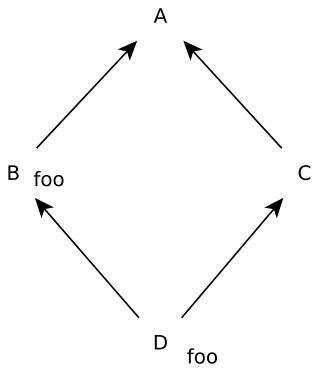
B x = new D()
x.foo()





Utilisation du hachage parfait

B x = new D()
x.foo()



id(B) and mask => position

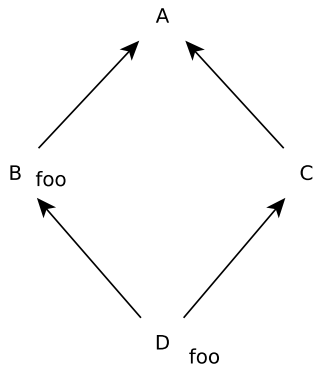
D

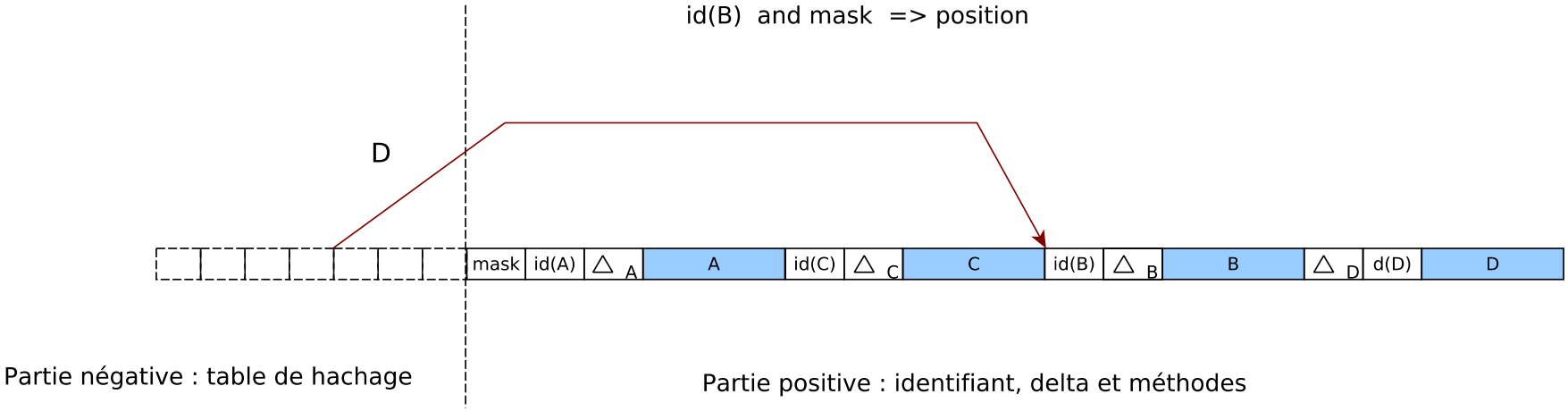
Partie négative : table de hachage

Partie positive : identifiant, delta et méthodes

Utilisation du hachage parfait

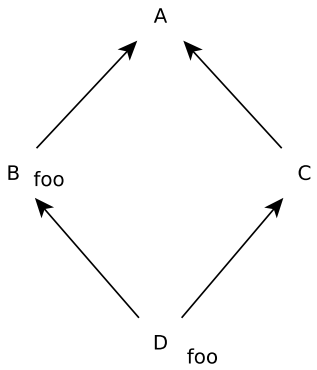
B x = new D()
x.foo()

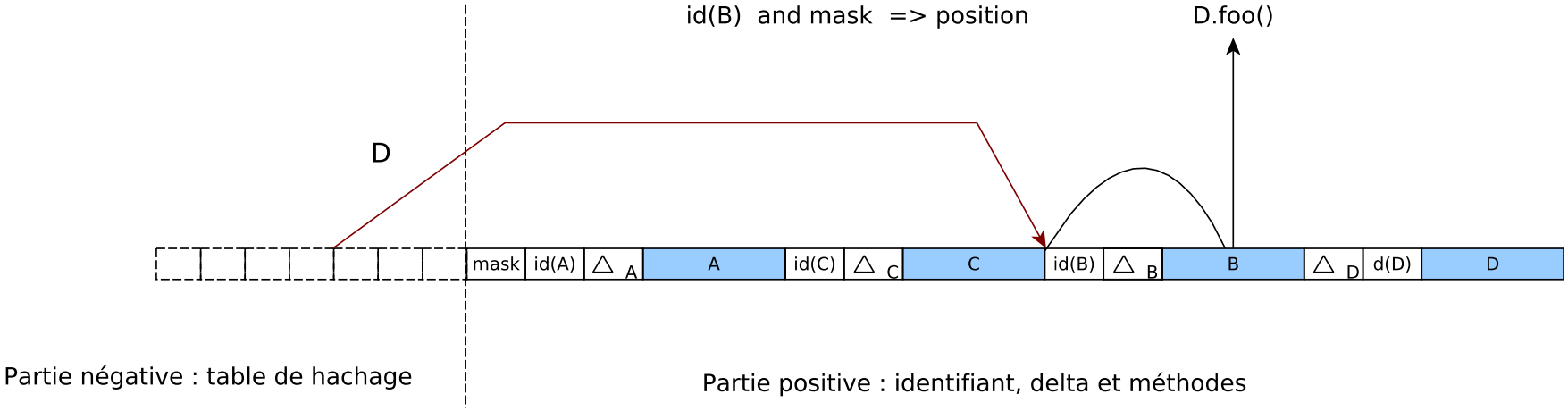




Utilisation du hachage parfait

B x = new D()
x.foo()





Le langage Nit : exemple

```
module fibonacci

redef class Int
    fun fibonacci: Int
    do
        if self < 2 then
            return 1
        else
            return (self-2).fibonacci + (self-1).fibonacci
        end
    end
end
end

print "Fibo(10) = " + 10.fibonacci
```


Detlefs, D. and Agesen, O. (1999).
Inlining of virtual methods.
In *ECOOP'99*, pages 258–277. Springer.


Ducournau, R. (2008).
Perfect hashing as an almost perfect subtype test.
ACM Trans. Program. Lang. Syst., 30(6):33:1–33:56.


Ducournau, R. and Morandat, F. (2011).
Perfect class hashing and numbering for object-oriented implementation.
Software: Practice and Experience, 41(6):661–694.


Ducournau, R., Pagès, J., Vidal, C., and Privat, J. (2015).
Preexistence revisited.
In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '15, pages 1–1. ACM.


Ducournau, R., Pagès, J., and Privat, J. (2016).

Preexistence and concrete type analysis in the context of multiple inheritance.
In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, pages 10:1–10:12.


Pagès, J. (2015).
A virtual machine for testing compilation/recompilation protocols in multiple inheritance.
In *ECOOP Doctoral Symposium '15*, pages 1–10.


Privat, J. (2008).
Nit language.
`http://nitlanguage.org/`.