Fevrier 98

La méthode Cliff Jones orientée objet pour le développement formel de programmes concurrents

Brahim Mammass
LIP6, équipe SPI
Université Pierre et Marie Curie

Table des matières

1 Introduction **5**

2 Le langage de conception **7**
2.1 L'objectif 7
2.2 Quelques éléments du langage 7

3 La logique de preuves **10**
3.1 Les assertions 10
3.2 Le langage logique 11
3.3 Quelques règles d'inférence 12
3.4 Les propriétés vérifiables 13

4 Une méthode pour contraindre les interférences **14**
4.1 La spécification 14
4.2 Le choix des structures de données 15
4.2.1 Les structures de données 15
4.2.2 L'invariant du programme 16
4.2.3 La nouvelle spécification 17
4.2.4 La relation entre les représentations abstraite et concrète 18
4.3 La décomposition des opérations 18
4.4 Introduction de la concurrence 19
4.4.1 Les règles de transformation 19
4.4.2 Application à la file d'attente 20

5 Une méthode pour raisonner sur les interférences **24**
5.1 La spécification 24
5.2 Le choix des structures de données 25
5.3 La décomposition des opérations 27
5.3.1 Décomposition de la classe Vector 27
5.3.2 Décomposition de la classe Primes 28
5.3.3 Décomposition de la classe Rem 30
5.4 Introduction de la concurrence 30

6 Sémantique opérationnelle de $\pi\o\beta\lambda$ **31**
6.1 Les transitions locales 31
6.1.1 Sémantique de l'affectation 31
6.1.2 Sémantique de l'instruction If-then-else 32
6.2 Les transitions globales 32
6.2.1 Perception d'une transition locale au niveau global 33
6.2.2 Sémantique de l'instruction new 33
6.2.3 Sémantique de l'invocation d'une méthode 34
6.2.4 Sémantique de l'instruction return 34

7 Sémantique π-calcul de $\pi\o\beta\lambda$ **35**
7.1 Syntaxe et sémantique informelle du π-calcul polyadique 35
7.2 Le codage des booléens 36
7.3 Le codage des entiers naturels 38
7.4 Sémantique des classes sans variables 39
7.5 Sémantique des classes avec variables 40
7.5.1 Variables d'instance booléennes 41
7.5.2 Variables de type référence 42

8 Conclusions et perspectives **43**
8.1 Le langage de conception 43
8.2 La logique de preuves 43
8.3 Les méthodes de développement 43
8.4 La sémantique SOS de $\pi\o\beta\lambda$ 44
8.5 La sémantique π-calcul de $\pi\o\beta\lambda$ 45

Table des figures

1 Liste chaînée des instances de Priq 15
2 Transfert de contrôle des méthodes (cas séquentiel) 20
3 Comportement de add (cas séquentiel) 21
4 Comportement de add (cas parallèle) 21
5 Comportement de test (cas séquentiel) 22
6 Comportement de test (cas parallèle) 22

1 Introduction

Ce papier présente une synthèse des travaux de Cliff Jones (université de Manchester). Nous nous sommes fondés sur des références qui s'échellonent sur plusieurs années et nous avons essayé de présenter une sorte d'union et de mise à jour de ces papiers. Ces travaux s'inspirent de l'expérience de Cliff Jones dans le développement de la méthode VDM (Vienna Development Method) et s'inscrivent dans le cadre de l'utilisation des méthodes formelles pour le génie logiciel et plus particulièrement celui de la concurrence. L'objectif de ces travaux est d'élaborer une méthode de développement compositionnelle pour les programmes concurrents qui ne s'appuie pas sur des variables partagées. Une méthode de développement est compositionnelle si le fait qu'une étape de conception satisfait une spécification donnée peut être justifié sur la base de la spécification de tous les composants sans aucune connaissance de leur construction. Une méthode compositionnelle est naturellement modulaire puisque le système est décomposé en plusieurs modules facilitant ainsi son développement. Les premières implantations des programmes concurrents utilisaient des variables partagées, ce qui conduit à des difficultés de conception évoquées plus loin.

L'avantage de la compositionnalité est que le raisonnement sur la corection des programmes peut être local: la preuve de la validité d'une étape de conception peut être faite sans connaissance de la structure interne de ses composantes. Ainsi, les erreurs de spécification peuvent être détectées dès les premières phases du développement. La recherche de méthodes de développement compositionnelles est très difficile à cause des interférences qui peuvent apparaître dans les systèmes concurrents [Jon96a]. En effet, dans certaines applications, plusieurs utilisateurs peuvent interagir en même temps avec le système (application de réservation d'avion ou application bancaire). La spécification de ces applications doit tenir compte des interférences possibles entre les différents utilisateurs.

Dans les programmes concurrents avec variables partagées, les interférences se manifestent par le changement de l'état de ces variables. Afin d'analyser des programmes avec possibilité d'interférence, il est nécessaire de faire des hypothèses sur la granularité de ces programmes. Cela impose de décider quelles vont être les instructions atomiques (souvent, on décide que l'instruction d'affectation est atomique). Les programmes résultant de cette approche ne peuvent pas toujours être implémentés de manière efficace car la méthode de développement impose au concepteur une atomicité qui n'est pas celle qu'il aurait besoin d'utiliser pendant le développement.

En utilisant certains concepts orientés-objet, Cliff Jones propose une méthode compositionnelle pour les systèmes concurrents qui évite de faire des hypothèses d'atomicité [Jon95b]. Ainsi, le contrôle de la granularité peut être complètement géré par le développeur et n'est donc plus figé une fois pour toutes par la méthode de développement. Dans un monde orienté-objet, toute variable est obligatoirement encapsulée par un objet. Les variables d'un objet ne sont accessibles que par les méthodes de l'objet lui-même. De plus, si une seule méthode est active dans un objet à un moment donné, on peut dire que l'exécution d'une méthode est atomique et que les variables d'un objet sont protégées des interférences. Les interactions entre les objets sont implémentées par des appels de méthodes. Mais, les objets peuvent aussi évoluer indépendamment, chacun exécutant l'une de ses méthodes. On peut structurer les interactions dans une application en posant des contraintes sur les possibilités de communication entre les objets, ce qui permet de minimiser et même d'éviter parfois les interférences dans l'application. Ainsi, on peut implémenter par exemple une variable globale par un objet et assurer que l'accès en lecture et en modification de la variable sont atomiques en définissant les méthodes appropriées. On peut alors définir une méthode qui incrémente la variable tout en évitant tout danger d'interférence, ce qu'on ne peut pas faire avec l'instruction d'affectation dans le cas d'une variable partagée. Dans ce dernier cas, il faut prévoir un mécanisme de gestion d'accès à la variable (par exemple un sémaphore).

La section 2 présente le langage de conception utilisé pour décrire les systèmes concurrents. La section 3 présente la logique utilisée pour exprimer les propriétés du système spécifié et pour faire les preuves de ces propriétés. La section 4 présente une première méthode de développement pour des applications peu concurrentes permettant de restreindre et d'éviter les interférences en structurant les données de l'application. La section 5 décrit une deuxième méthode de développement dédiée aux applications complexes où les risques d'interférences ne peuvent être évités; cette méthode permet donc de raisonner sur les interférences. Ces deux méthodes procèdent par transformation de programme (raffinement) et introduisent la concurrence à la dernière étape du développement. La section 6 décrit la sémantique opérationnelle du langage de conception. La section 7 présente une autre sémantique du langage dans le π-calcul [Mil89]. Enfin, la dernière section présente quelques critiques ainsi que les perspectives de ce travail.

2 Le langage de conception

Dans cette section, nous allons décrire le langage de conception qui sert à décrire les applications concurrentes.

2.1 L'objectif

Le but de Cliff Jones n'est pas de créer un nouveau langage orienté-objet concurrent. Il s'agit seulement d'utiliser certains concepts orientés-objet qui rendraient la conception des programmes concurrents plus attractive qu'avec les langages à variables partagées. La notation de conception utilisée est appelée $\pi\o\beta\lambda$ et a été fortement inspirée du langage de programmation orienté-objet parallèle POOL [Ame89].

2.2 Quelques éléments du langage

Une syntaxe réduite du langage de conception, décrite à la BNF, est présentée à la fin de la section. On peut remarquer que le langage est typé.

Une application est spécifiée à l'aide de classes. Une **classe** décrit la structure et le comportement d'une famille d'objets. Elle liste les variables d'instance avec les types associés et les valeurs initiales ainsi que les méthodes et définit pour chaque méthode les paramètres, le type du résultat retourné, les variables locales et le corps. Il n'existe pas de variables de classe ni de méthodes de classe et toute classe possède un constructeur unique appelé *new* qui n'a pas à être déclaré car les variables d'instance doivent avoir une valeur par défaut. Quand la méthode **new** est appelée, elle crée un nouvel objet, lui associe une nouvelle référence et retourne cette référence comme résultat. L'objet peut ainsi être accédé à l'aide de cette référence. Chaque objet créé possède ses propres copies des variables d'instance et ce sont ces copies qui sont manipulées par ses méthodes.

Une variable d'instance peut être de type simple ou composé. Les types simples sont les types **Integer** ($\mathcal{N}$), **Boolean** ($\mathcal{B}$) ou **référence typée** (Ref). Une variable de type référence aura pour contenu la référence d'un objet. Les types composés **bag** (multi-set), **seq** (séquence), **set** (ensemble), ... et les opérateurs associés sont fournis par le langage de spécification.

Une référence peut être de type **Private** (privée) ou **Shared** (partagée). Quand elle est de type *Private*, elle n'est connue que de l'objet qui l'a créée et ne peut être connue par les autres objets. Cette restriction protège l'objet *Private* de tout danger d'interférence. Quand elle est de type *Shared*, elle est connue par l'objet qui l'a créée ainsi que par tout autre objet. Par exemple, deux méthodes actives dans deux objets clients distincts, peuvent interférer entre elles par les invocations de méthodes qu'elles peuvent faire à un serveur dont elles partagent la référence (qui est de type *Shared*). Ces interférences se manifesteront par des modifications des variables d'instance du serveur, de manière à priori non déterministe.

Un objet peut invoquer les méthodes de tout autre objet dont il connaît la référence. Quand l'objet serveur accepte l'invocation du client, le client est bloqué dans un **rendez-**

vous. Ce rendez-vous est achevé quand une valeur est retournée par le serveur. Ce retour de résultat peut être réalisé simplement par l'instruction **return**. Une autre instruction, **delegate**, permet à un objet serveur de transférer la responsabilité de réponse à un autre objet sans que lui-même attende le résultat. L'objet serveur peut ainsi anticiper la fin de la première invocation et accepter d'autres invocations de méthodes qui seront traitées en parallèle avec la première.

Un objet peut donc être dans l'un des trois états suivants. À l'état **dormant**, il ne fait rien. Quand il est à l'état **attente**, il est bloqué dans un rendez-vous suite à une invocation de méthode. Enfin, quand il est à l'état **actif**, il exécute le corps de l'une de ses méthodes. Un objet ne peut donc répondre à l'invocation de l'une de ses méthodes que s'il est à l'état dormant. Il en découle qu'il y a au plus une méthode active dans un objet à un moment donné.

En plus des instructions évoquées ci-dessus, $\pi\o\beta\lambda$ fournit d'autres instructions simples telles que l'affectation, if-then-else, while, ... La syntaxe complète du langage est décrite dans [Jon92]. Une syntaxe réduite est présentée ci-dessous.

$System$ = $Id \rightarrow Cdef$ (système = ensemble de classes)

$Cdef$ ::= $ivars\colon\ Id \rightarrow Type$ (variables de la classe et leur type)
$init\colon\ Id \rightarrow Val$ (valeurs initiales des variables)
$mm\colon\ Id \rightarrow Mdef$ (méthodes de la classe)

$Type$ = $BOOL \mid INT \mid Ref$

(les types simples)

Ref ::= $PrivateRef \mid SharedRef$ (les références)

$PrivateRef$ ::= Id (référence privée)

$SharedRef$ ::= Id (référence partagée)

$Mdef$ ::= $r\colon\ Type$ (type du résultat retourné par la méthode)
$pl\colon\ (Id \times Type)^*$ (liste des paramètres)
$tvars\colon\ Id \rightarrow Type$ (variables locales)
$body\colon\ Stmt$ (corps de la méthode)

$Stmt$ = $Assign \mid Compound \mid If \mid While \mid Return \mid Delegate \mid New \mid Call$

$Assign$ ::= $lhs\colon\ Id$ (l'affectation)
$rhs\colon\ Expr$

$Compound$::= $sl\colon\ Stmt^*$ (la séquence)

If ::= b: $Expr$ (l'instruction If-then-else)
$then$: $Stmt$
$else$: $Stmt$

$While$ ::= b: $Expr$ (la boucle while)
s: $Stmt$

$Return$::= r: $Expr$ (l'instruction return)

$Delegate$::= r: $Mref$ (la délégation du retour)

New::= lhs: Id (l'instruction new)
cn : Id

$Call$::= lhs: Id (l'appel d'une méthode)
$call$: $Mref$

$Mref$::= obj: Id (nom de l'objet appelé)
mn: Id (nom de la méthode)
al: $Expr$ (liste des arguments)

$Expr$ = $Id \mid \mathcal{B} \mid \mathcal{N} \mid Nil \mid Bop \mid Mop \mid Cmp$ (les expressions)

Bop::= op: Id (opération binaire)
e_1: $Expr$
e_2: $Expr$

Mop::= op: Id (opération unaire)
e: $Expr$

Cmp ::= op: Id (la comparaison de 2 expressions)
e_1: $Expr$
e_2: $Expr$

3 La logique de preuves

Le langage de spécification permet de décrire la conception et l'implémentation des applications concurrentes. Il permet aussi de décrire les propriétés de ces applications en fournissant un langage logique que nous allons décrire.

3.1 Les assertions

La description des propriétés du système est faite par des assertions. Elles sont écrites dans une logique qui est dérivée de la logique présentée dans [Jon91a] et qui ressemble à TLA (Temporal Logic of Actions) de Lamport [Lam91] et à UNITY [CM88]. Pour décrire les méthodes d'une classe, on peut utiliser 2 types d'assertions que nous allons présenter.

Un objet possède un **état** qui est défini par les valeurs de ses variables. Cet état est noté ξ. L'**environnement** est constitué par l'ensemble des états des objets du système.

Dans les assertions, on peut utiliser des prédicats sur l'état d'un objet (p: $\xi \rightarrow \mathcal{B}$). Ainsi, la notation $p.b$ signifie que la valeur de b à l'état courant vérifie le prédicat p et la notation $p.\overline{b}$ signifie que la valeur de b à l'état précédent vérifie le prédicat p.

On peut également utiliser des relations sur des paires d'états de l'objet (r: $\xi_1 \times \xi_2 \rightarrow \mathcal{B}$). Ainsi, la notation $r.(\overline{b}, b)$ signifie que le système transite de la valeur de la variable b à l'état précédent à celle de l'état courant. Donc, $(\overline{b}, b)$ est une **transition** du système.

Rappelons enfin que les valeurs des variables sont accédées uniquement entre les invocations des méthodes qui sont atomiques.

Les **assertions locales** à un objet donné utilisent l'état de cet objet avant et après l'exécution de la méthode. Ces assertions sont données par *Pre* et *Post*. Comme exemple, supposons qu'on veut spécifier le type ensemble d'entiers. Si on veut supprimer un élément de l'ensemble, la pré-condition est que l'ensemble doit être non vide. La post-condition est que l'ensemble des entiers obtenu est celui de l'état précédent auquel il faut retirer cet élément. Ci-dessous $\overline{b}$ signifie le contenu de b à l'état précédent, b son contenu à l'état courant.

ensemble Class
    vars b: $\mathcal{N}$-*set* $\leftarrow \{\}$
    rem(e: $\mathcal{N}$) *method*
        pre $b \neq \{\}$
        post $b = \overline{b} - \{e\}$

Les **assertions globales** à plusieurs objets sont exprimées par des propriétés sur l'environnement et sont appelées rely/garantee-conditions ([Jon81], [Sto90]). Les rely/garantee-conditions sont la généralisation des pre/post-conditions utilisées dans les spécifications séquentielles aux spécifications parallèles. Leur syntaxe est la suivante: soit e un environnement et soit S une suite d'instructions. On note par S/e l'environnement obtenu suite à la modification de e par l'exécution de S. La formule $p_1(e) \Rightarrow p_2(S/e)$ signifie que si l'environnement

e satisfait la propriété p_1, alors l'exécution de la suite d'instructions S dans l'environnement e (i.e, S/e) satisfaira la propriété p_2. Un exemple peut être consulté dans la section 5.3.2.

Les pre/post-conditions et les rely/garantee-conditions ont l'avantage d'inciter le développeur à expliciter les conditions d'appel des méthodes, donc à spécifier plus finement les méthodes, ce qui peut limiter les obligations de preuves qui seront engendrées.

3.2 Le langage logique

Les formules peuvent être exprimées à l'aide des opérateurs logiques classiques *and*, *or*, *not*, $\Rightarrow$ et $\Leftrightarrow$. Des opérateurs temporels peuvent être également utilisés; nous allons les décrire dans ce qui suit.

Soit S une suite d'instructions, soit r une relation sur les paires d'états d'un objet (r: $\xi_1 \times \xi_2 \rightarrow \mathcal{B}$) et soit p un prédicat sur l'état d'un objet (p: $\xi \rightarrow \mathcal{B}$).

L'opérateur de base pour le raisonnement de sûreté est l'opérateur **links**. L'expression S *links* r signifie que toute exécution de la suite d'instructions S mène à des transitions qui satisfont la relation r.

D'autres opérateurs dérivés peuvent être utilisés:

- L'opérateur **fin**

  S *fin* p signifie que l'exécution de la suite d'instructions S se termine et que l'état final vérifie le prédicat p.

- L'opérateur **confirms**

  S *confirms* p signifie que si l'exécution de la suite d'instructions S mène à des transitions telles que: si le prédicat p est vérifié à l'état précédent alors p est vérifié à l'état courant; alors S confirme le prédicat p.

- L'opérateur **maintains**

  S *maintains* p signifie que si l'exécution de la suite d'instructions S mène à des transitions telles que: si le prédicat p est vérifié à l'état précédent alors p est vérifié à l'état courant et réciproquement; alors S maintient le prédicat p.

- L'opérateur **conserves**

  Soit f: $\xi \rightarrow Val$, une valuation qui associe une valeur à chaque variable d'un objet.

  S *conserves* f signifie que si l'exécution de la suite d'instructions S mène à des transitions telles que la valuation f à l'état précédent est identique à la valuation f à l'état courant; alors S conserve la valuation f.

3.3 Quelques règles d'inférence

Soit S une suite d'instructions, soit r une relation sur les paires d'états d'un objet (r: $\xi_1 \times \xi_2 \rightarrow \mathcal{B}$) et soit p un prédicat sur l'état d'un objet (p: $\xi \rightarrow \mathcal{B}$). Soit f: $\xi \rightarrow Val$, une valuation qui associe une valeur à chaque variable d'un objet.

On va décrire les règles d'inférence qui peuvent être utilisées pendant les preuves de programmes. Les trois règles d'inférence qui suivent sont déduites directement des opérateurs qu'on vient de décrire:

$$\frac{S\ links\ (\overline{p} \Rightarrow p)}{S\ confirms\ p}$$

$$\frac{S\ links\ (\overline{p} \Leftrightarrow p)}{S\ maintains\ p}$$

$$\frac{S\ links\ (f(\overline{\xi}) = f(\xi))}{S\ conserves\ f}$$

Les deux règles qu'on va décrire maintenant sont plus importantes car elles introduisent la concurrence:

- $\|$-$links$

  $$\frac{\bigwedge_i\ (S_i\ links\ r)}{(\|_i\ S_i)\ links\ r}$$

  signifie que si l'exécution de chaque séquence S_i $(i = 1..n)$ mène à des transitions qui vérifient la relation r, alors l'exécution parallèle des séquences S_i $(i = 1..n)$ mène à des transitions qui vérifient la relation r.

- $\|$-fin

  $$\frac{\bigwedge_i (S_i\ links\ r)\ , \bigwedge_i (e\ links\ r \Rightarrow S_i/e\ fin\ p_i)}{e\ links\ r \Rightarrow (\|_i\ S_i)/e\ fin\ (\bigwedge_i\ p_i)}$$

  signifie que:

  - si l'exécution de chaque séquence S_i $(i = 1..n)$ mène à des transitions qui vérifient la relation r et
  - si pour chaque séquence S_i $(i = 1..n)$, l'exécution de e mène à des transitions qui vérifient la relation r implique que l'exécution de S_i $(i = 1..n)$ dans l'environnement e se termine et que le dernier état vérifie le prédicat p_i $(i = 1..n)$

  alors: si l'exécution de e mène à des transitions qui vérifient la relation r alors l'exécution parallèle des séquences S_i $(i = 1..n)$ dans l'environnement e se termine et le dernier état vérifie la conjonction des prédicats p_i $(i = 1..n)$.

3.4 Les propriétés vérifiables

Les propriétés vérifiables à l'aide de cette logique sont:

- les propriétés de **surêté**: il s'agit de vérifier qu'une propriété est toujours vraie, par exemple l'absence d'interblocage dans l'application (voir l'invariant de la file d'attente dans la section 4.2.2).
- les propriétés de **vivacité**: il s'agit de vérifier qu'une propriété désirée sera vérifiée un jour par le système (i.e, que le système progresse) ou que le système termine par exemple.

4 Une méthode pour contraindre les interférences

La première méthode de développement concerne les applications concurrentes où les risques d'interférence ne sont pas très importants et peuvent être évités [Jon93a]. Les données de l'application sont décomposées en parties isolées les unes des autres, ce qui permet de décomposer et ensuite de paralléliser les traitements tout en évitant les interférences. Or, si on veut prouver l'absence de deadlock dans l'application, il faut montrer qu'il n'y a pas de cycle dans les structures de données. En modélisant les données par une structure inductive, on peut utiliser une induction structurelle, ce qui facilite les preuves d'absence de deadlock.

Dans l'orienté objet, la structure de données inductive sera décrite par une classe. Les objets de la classe constitueront ainsi les parties isolées de la structure inductive.

Le développement d'une application se fait par raffinement et se déroule en quatre étapes. À chaque étape, on prouve que le programme obtenu satisfait la spécification initiale. Les premières étapes supposent une exécution séquentielle de l'application. La concurrence est introduite seulement à la dernière étape du développement.

4.1 La spécification

La spécification du système est donnée en termes de classes. Les méthodes de chaque classe sont spécifiées par l'intermédiaire de pre/post-conditions.

L'exemple qui est étudié dans cette section concerne un programme qui donne une implémentation parallèle d'une file d'attente avec priorité. La méthode *add* insère son argument de façon à ce que la file reste ordonnée de manière croissante. La méthode *rem* supprime l'élément de tête de la file. Enfin, la méthode *test* vérifie si son argument est présent dans la file. Plusieurs utilisateurs, s'exécutant en parallèle, peuvent invoquer les méthodes offertes par la file. L'implémentation de la file doit donc gérer cette concurrence.

Dans ce qui suit, on va d'abord donner une spécification séquentielle de la file d'attente. Ensuite, on va introduire les structures de données permettant d'éviter les interférences entre les utilisateurs de la file. Enfin, en évitant toujours les interférences, on va introduire la concurrence afin d'obtenir un code parallèle implémentant la file d'attente.

Dans la spécification ci-dessous, la file avec priorité est modélisée par un multi-ensemble d'entiers. La notation $\overline{b}$ signifie le contenu du multi-ensemble b avant l'exécution de la méthode; b son contenu après l'exécution de la méthode. On utilise le type *bag* (multi-ensemble) et les opérateurs associés qui sont fournis par le langage de spécification inspiré de VDM [Jon90]. Nous allons décrire le type multi-ensemble d'entiers:

- $\mathcal{N}$-*bag*: multi-ensemble d'entiers naturels (i.e, un entier peut avoir plusieurs occurrences dans le multi-ensemble),
- $\{\}$: dénote le multi-ensemble vide,
- $\cup$: dénote l'union de deux multi-ensembles,

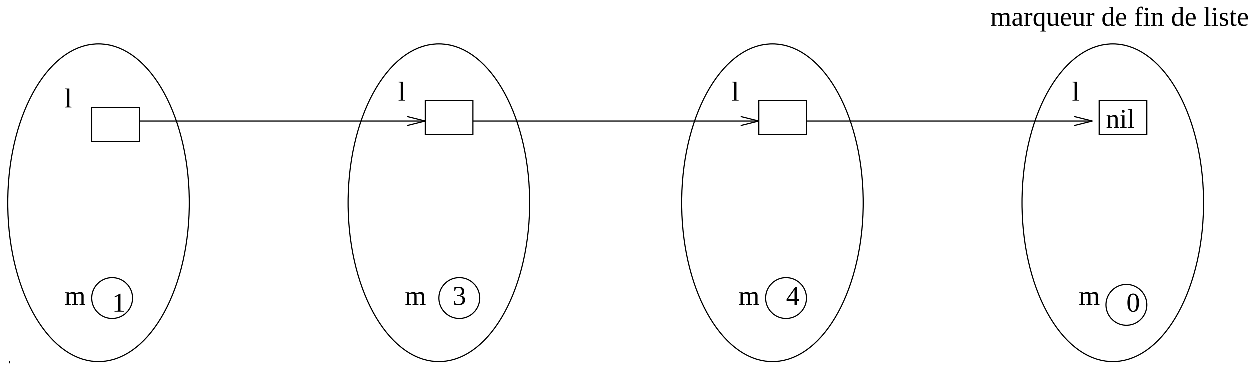

FIG. 1 - *Liste chaînée des instances de Priq*

- $min(b)$: retourne un élément quelconque du multi-ensemble b,
- b - $\{r\}$: retourne le multi-ensemble b privé de l'élément r,
- $e \in b$: retourne vrai si e est un élément du multi-ensemble b.

$Priq\ Class$
  $vars\ b$: $\mathcal{N}$-$bag \leftarrow \{\}$
  $add(e$: $\mathcal{N})$ $method$
    $post\ b$ = $\overline{b} \cup \{e\}$
  $rem(\ )$ $method\ r$: $\mathcal{N}$
    $pre\ b \neq \{\}$
    $post\ r$ = $min(\overline{b})$ and b = $\overline{b}$ - $\{r\}$
  $test(e$: $\mathcal{N})$ $method\ r$: $\mathcal{B}$
    $post\ r$ = $(e \in b)$

4.2 Le choix des structures de données

4.2.1 Les structures de données

Dans cette étape, on cherche d'abord une structure de données concrète pour représenter la structure de données abstraite de la spécification.

Pour l'exemple traité, chaque élément de la file d'attente (i.e, appartenant au multi-ensemble) est mémorisé dans une instance de la classe $Queue$ décrite dans cette sous-section. Les objets (i.e, instances) sont organisés en **liste chaînée**. L'objet de tête se comporte comme un serveur contenant toute la file. Mais en fait, chaque objet contient un seul élément de la file (m) et une référence à l'objet suivant de la liste chaînée (l). Le dernier objet contient la valeur 0 et n'a pas de successeur; il marque ainsi la fin de la file. Cette configuration est illustrée par la figure 1.

4.2.2 L'invariant du programme

Une fois qu'on a choisi la structure de données concrète, on définit l'invariant du programme qui dépend de cette structure et qui exprime la propriété de sûreté du système, à savoir l'absence d'interblocage.

Comme l'invariant dépend de plusieurs instances de la classe, il est nécessaire d'introduire un état global (i.e, l'environnement). Cet état global est modélisé par une fonction Σ qui associe à chaque instance créée une référence:

Σ: $Inst \rightarrow Ref$.

Avant d'écrire l'invariant pour l'exemple traité, on a besoin de définir certaines fonctions intermédiaires. Soit σ de type Σ:

- *is-linked-list*: prend en arguments une référence (p), un pointeur (l) et l'état global (σ) et vérifie si p représente bien la référence de tête d'une liste chaînée d'objets liés entre eux par le pointeur l.
- *extract-seq*: prend en arguments une référence de liste chaînée d'objets (p), le pointeur vers l'objet suivant (l), le contenu d'un objet (m) et l'état global (σ) et retourne une séquence formée des éléments (m) des objets de la liste chaînée,
- *reach*: prend en arguments une référence de liste chaînée d'objets (p), le pointeur qui lie les objets (l) et l'état global (σ) et retourne toutes les références accessibles par cette référence de tête et
- *is-ascending*: prend en argument une séquence d'entiers et vérifie si les éléments de la séquence sont ordonnés de façon croissante.

Maintenant, on peut écrire l'invariant:

inv: $Ref \rightarrow \Sigma \rightarrow \mathcal{B}$

Soit p de type Ref et σ de type Σ:

$$
\begin{aligned}
inv(p)(\sigma) = \; & is\text{-}linked\text{-}list(p, l)(\sigma) \; and \\
& is\text{-}ascending(extract\text{-}seq(p, l, m)(\sigma)) \; and \\
& \forall \; r \in reach(p, l)(\sigma) \\
& \quad l(\sigma(r)) = nil \Leftrightarrow m(\sigma(r)) = 0
\end{aligned}
$$

L'invariant ainsi défini implique qu'il ne peut y avoir de cycle dans la chaîne des références des objets car les objets sont ordonnés de manière croissante et l'objet contenant 0 marque la fin de la liste chaînée et ne peut pas avoir de successeur dans la file. Sinon, cela introduirait un cycle et les invocations des méthodes *add*, *rem* et *test* de la classe *Queue* provoqueraient un **deadlock**.

Noter que l'invariant est vrai uniquement entre les invocations des méthodes et non pendant l'exécution d'une méthode car les méthodes sont définies à l'aide des pre/post-conditions et sont donc atomiques.

4.2.3 La nouvelle spécification

Maintenant, on peut spécifier la file avec priorité en utilisant la représentation concrète. Comme les références des objets sont de type *Private* (i.e, elles ne peuvent être connues par d'autres objets), chaque objet de la file est protégé de tout danger d'interférence.

Les pre/post-conditions dans la nouvelle spécification signifient:

- méthode *add*: si on ajoute un élément e dans la file d'attente, alors si on extrait la séquence des éléments de la file, il existe un indice i tel que le contenu de la séquence à cet indice est égal à e. De plus, si on retire cet élément de la file, on retrouve la file à l'état précédent.
- méthode *rem*: la file d'attente doit être non vide et si on supprime l'élément de tête alors, la nouvelle file devient l'ancienne file amputée de son élément de tête.
- méthode *test*: l'élément e est dans la file si et seulement si, si on extrait la séquence des éléments de la file, alors il existe un indice i tel que le contenu de la file à cet indice est égal à e.

La spécification utilise le type *seq* (séquence) ainsi que les opérateurs associés que nous allons décrire:

- $del(s, i)$: retourne la séquence s privée de l'élément d'indice i,
- [ ]: dénote la séquence vide,
- $hd(s)$: retourne l'élément de tête de la équence s,
- $tl(s)$: retourne la séquence s privée de son élément de tête,
- $inds(s)$: retourne l'ensemble formé des indices de la séquence s,
- $bagof(s)$: retourne un multi-ensemble formé des éléments de la séquence.

$Queue\ Class$
$\quad vars\ m\colon \mathcal{N} \leftarrow 0;$
$\qquad\quad l\colon PrivateRef(Queue) \leftarrow nil$
$\quad add(e\colon \mathcal{N})\ method$
$\qquad\quad post\ let\ \overline{s} = extract\text{-}seq(self, l, m)(\overline{\sigma})\ in$
$\qquad\qquad let\ s = extract\text{-}seq(self, l, m)(\sigma)\ in$
$\qquad\qquad \exists\ i \in inds(s),\ s(i) = e\ and\ del(s, i) = \overline{s}$
$\quad rem(\ )\ method\ r\colon \mathcal{N}$
$\qquad\quad pre\ extract\text{-}seq(self, l, m)(\sigma) \neq [\ ]$
$\qquad\quad post\ let\ \overline{s} = extract\text{-}seq(self, l, m)(\overline{\sigma})\ in$
$\qquad\qquad let\ s = extract\text{-}seq(self, l, m)(\sigma)\ in$
$\qquad\qquad r = hd(\overline{s})\ and\ s = tl(\overline{s})$
$\quad test(e\colon \mathcal{N})\ method\ r\colon \mathcal{B}$
$\qquad\quad post\ let\ s = extract\text{-}seq(self, l, m)(\sigma)\ in$
$\qquad\qquad r = \exists\ i \in inds(s),\ s(i) = e$

4.2.4 La relation entre les représentations abstraite et concrète

Enfin, on va établir la correspondance entre la représentation concrète de la file (la liste chaînée) et sa représentation abstraite (le multi-ensemble). Cette correspondance servira à prouver l'équivalence entre la nouvelle spécification et celle de l'étape précédente.

$retr\colon\ Ref \rightarrow \Sigma \rightarrow \mathcal{N}\text{-}bag$

$retr(p)(\sigma) = bagof(extract\text{-}seq(p, l, m)(\sigma))$

La fonction *retr* prend en arguments une référence de liste chaînée d'objets (p) et l'état global (σ) et retourne le multi-ensemble formé par les éléments de la liste chaînée.

Maintenant, pour établir l'équivalence entre les deux spécifications, il faut montrer que partant d'une file abstraite et d'une file concrète équivalentes, et en appliquant une méthode quelconque parmi les méthodes offertes par la classe *Queue*, on conserve toujours l'équivalence entre les deux représentations concrète et abstraite (i.e, le multi-ensemble constitué des éléments de la liste chaînée obtenue par l'application de la méthode concrète est équivalent au multi-ensemble obtenu par application de la méthode abstraite). Il s'agit donc d'établir une équivalence observationnelle entre les deux spécifications.

4.3 La décomposition des opérations

Cette étape du développement consiste à écrire le code qui satisfait la spécification. Les pre/post-conditions exprimées précédemment en fonction de l'état précédent et de l'état courant des variables sont décomposées en plusieurs instructions.

La décomposition est faite de manière à obtenir des méthodes récursives afin de pouvoir utiliser une induction structurelle sur les appels des méthodes lors des preuves de l'équivalence du code obtenu et de la spécification initiale. Par exemple, dans la définition de la méthode *add*, on ramène toujours l'insertion d'un élément dans la file d'attente à une insertion en tête de file. Ainsi, pour insérer l'élément 3 dans une file d'attente contenant les éléments 2, 4 et 5, comme 3 doit être inséré entre 2 et 4, on remplace 4 par 3 et on insère 4 en tête de la file restante commençant par l'élément 5.

La solution classique d'insertion par modification de pointeur (i.e, pour l'exemple, le suivant de 2 devient 3 et le suivant de 3 est l'ancien suivant de 2 qui est 4) n'est pas envisagée car elle nécessite qu'un élément connaisse son suivant et le suivant de son suivant. La modification de pointeur suppose donc que l'objet courant (i.e, 2) connaisse à la fois la référence de l'objet créé (i.e, 3) qui devient son successeur et la référence de son ancien successeur (i.e, 4). Cela ne peut être réalisé car le pointeur vers l'élément suivant dans la définition de la classe *Queue* est de type *Private*; il ne peut être connu que par un seul objet.

$Queue\ \ Class$
$\quad vars\ m\colon\ \mathcal{N} \leftarrow 0;$
$\qquad\qquad l\colon\ PrivateRef(Queue) \leftarrow nil$

```
add(e: 𝒩) method
  if l = nil then (m ← e; l ← new Queue) -- insertion en tête
  elif m ≺ e then l.add(e) -- progresser dans la file
  else (l.add(m); m ← e) -- insertion au milieu de la file
        -- e remplace m et m est inseré dans le reste de la file
  fi
  return
rem( ) method r: 𝒩
  t: 𝒩
  t ← m -- élément de tête mis dans t, si file vide on retourne 0
  if l ≠ nil then m ← l.rem() -- décaler le reste de la file
       if m = 0 then l ← nil -- MAJ de l si fin de file
       fi
  fi
  return t
test(e: 𝒩) method r: ℬ
  if (l = nil or e ≺ m) then return false -- e non trouvé
  elif e = m then return true -- e est l'élément courant
  else return l.test(e) -- progresser dans la file
  fi
```

La justification de cette décomposition (i.e, que le programme obtenu satisfait la spécification) se fait par induction. En effet, la structure de données inductive (liste chaînée d'objets) permet d'appliquer une **induction structurelle** sur les appels récursifs des méthodes *add*, *rem* et *test*.

4.4 Introduction de la concurrence

Rappelons qu'une seule méthode est active dans un objet à un moment donné. Donc, dans le code précédent, les méthodes *add*, *rem* et *test* bloquent l'appelant dans un rendez-vous jusqu'à ce qu'elles se terminent. Or, comme les méthodes *add*, *rem* et *test* progressent dans la file, un appel de méthode en tête de liste chaînée (file d'attente) ne se termine que lorsque tous les appels récursifs se terminent. Ce transfert de contrôle est illustré par la figure 2.

4.4.1 Les règles de transformation

L'idée de base est d'identifier les instructions qui peuvent s'exécuter en parallèle sans qu'une différence de comportement du système soit observable. **On suppose que toutes les méthodes se terminent**.

La concurrence peut être introduite en appliquant l'une des deux équivalences suivantes. La première équivalence permet de relâcher l'invoquant d'une méthode dès que cela est possible et de continuer le traitement de la méthode. La deuxième équivalence permet de déléguer la réponse à l'invocation d'une méthode d'un objet à un autre objet; l'objet invoqué

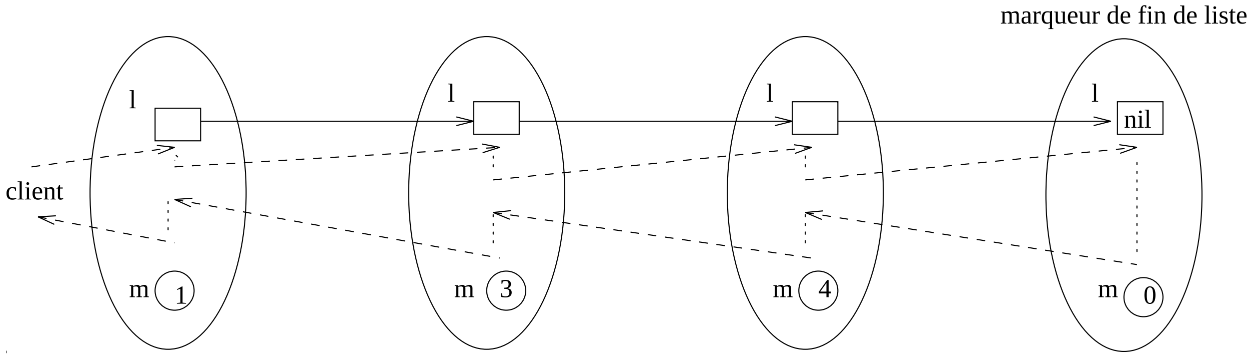

FIG. 2 - *Transfert de contrôle des méthodes (cas séquentiel)*

peut ainsi accepter d'autres invocations de ses méthodes qui vont s'exécuter en parallèle avec la première invocation.

1. [**S; return e** $\equiv$ **return e; S**] avec:
   - e n'est pas modifiée par S,
   - S ne contient pas d'instruction *return* ou *delegate*,
   - S se termine toujours et
   - chaque méthode invoquée dans S est associée à un objet accédé par une référence de type *Private* (ce qui assure que l'exécution de S ne provoquera aucun danger d'interférence et que l'équivalence sera préservée).
2. [**return l.m(x)** $\equiv$ **delegate l.m(x)**] avec:
   - l est une référence de type *Private* et
   - $l.m(x)$ se termine (i.e, l'appel de la méthode m par l'objet référencé par l se termine).

4.4.2 Application à la file d'attente

La méthode *add* bloque son invoquant dans un rendez-vous jusqu'à ce qu'elle arrive à l'endroit de la liste chaînée où elle doit insérer son argument et que l'instruction *return* ait été exécutée dans chaque objet de la liste chaînée ayant été invoqué (figure 3).

Si on applique la première équivalence, l'instruction *return* sera placée au début de la méthode *add* et l'invoquant sera relâché du rendez-vous juste après l'appel et pourra continuer ses calculs en parallèle avec l'activité de *add*. Or, comme *add* progresse dans la liste, les objets ayant été parcourus pourront à leur tour accepter d'autres invocations de méthodes (figure 4).

La validité de cette transformation (équivalence observationnelle des 2 codes de la méthode *add*) est assurée par le fait que la liste chaînée est contrôlée par des références de type

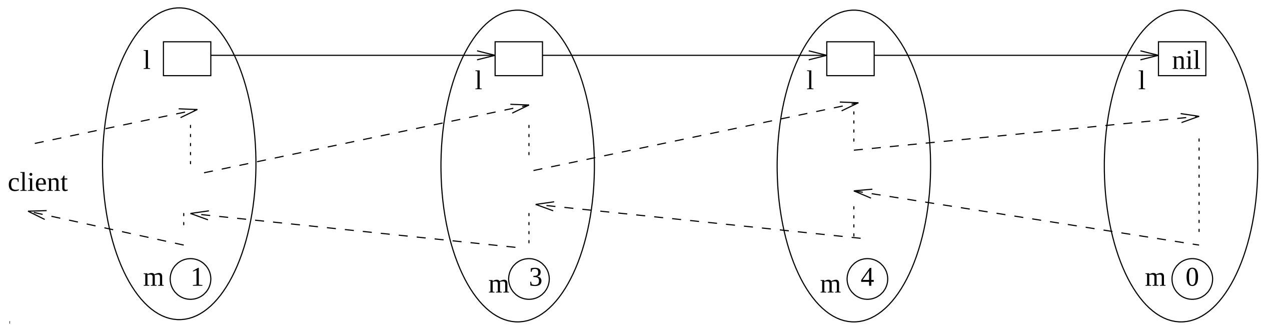

FIG. 3 - *Comportement de add (cas séquentiel)*

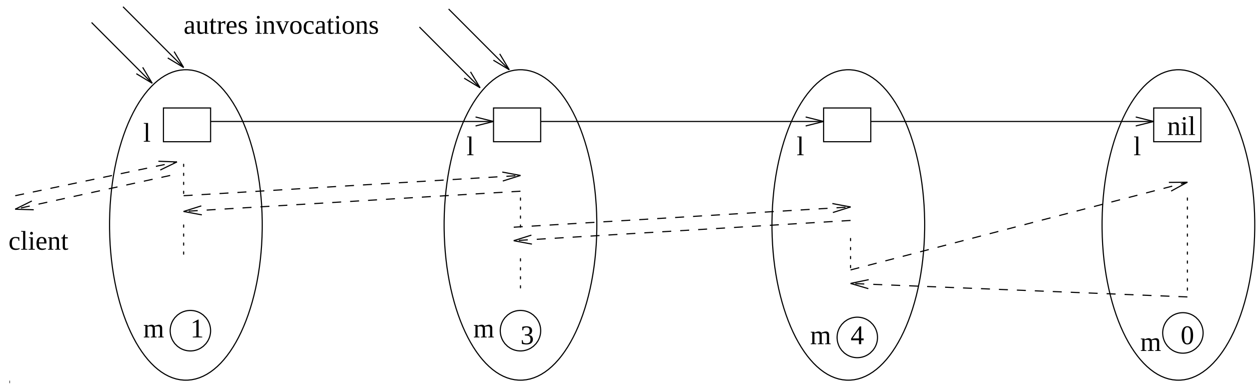

FIG. 4 - *Comportement de add (cas parallèle)*

Private (aucune autre file d'exécution ne peut interférer avec la file) et par le fait qu'une seule méthode est active dans un objet à un moment donné. Si on applique les deux codes pour insérer un élément dans la file d'attente, le résultat observé est exactement le même.

Cette transformation serait fausse si on avait des références de type *Shared* car ces références permettraient aux objets d'interférer entre eux et le comportement du code après le déplacement du *return* ne correspondrait pas à celui de la version séquentielle. En effet, deux objets clients qui partagent la référence d'un objet serveur peuvent interférer entre eux par les invocations de méthodes qu'ils peuvent faire à ce serveur (la référence *Shared* simule une variable partagée).

La première équivalence peut être appliquée également à la méthode *rem*. Cependant, il n'est pas possible d'appliquer cette équivalence à la méthode *test* car l'invoquant doit être bloqué jusqu'à ce que un résultat soit retourné (figure 5). Toutefois, il est possible d'éviter un verouillage complet de la liste pendant l'exécution de la méthode *test* en appliquant la deuxième équivalence. Si chaque objet transfère la responsabilité de réponse à l'invoquant (instruction **delegate**) à l'objet qui le suit dans la liste, il serait possible pour les objets de

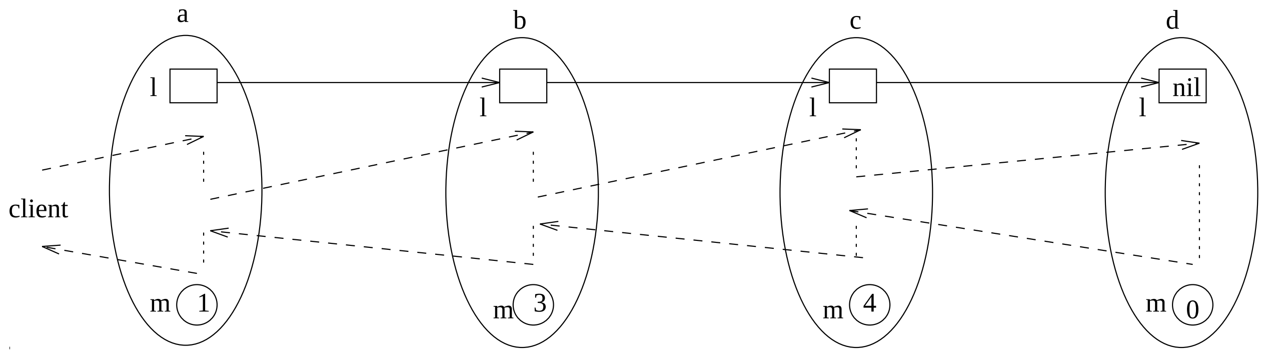

FIG. 5 - *Comportement de test (cas séquentiel)*

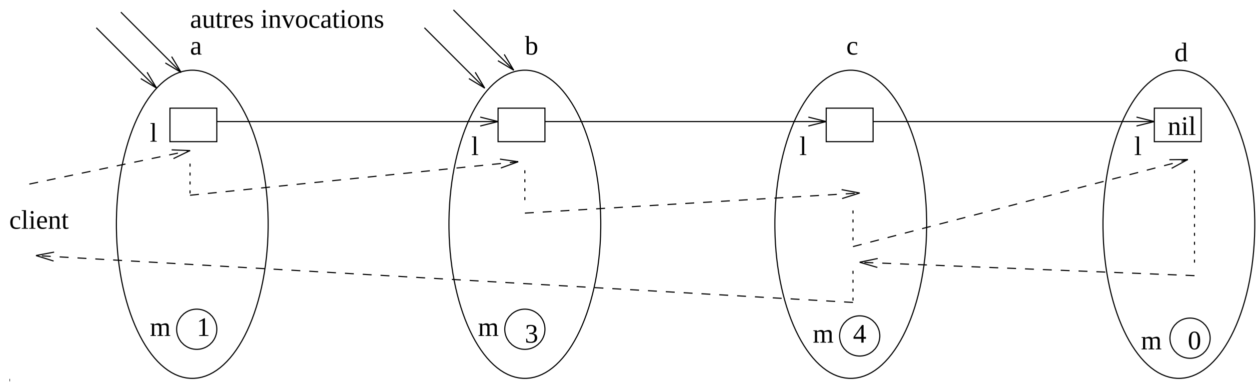

FIG. 6 - *Comportement de test (cas parallèle)*

la liste ayant été testés d'accepter d'autres invocations de méthodes qui vont s'exécuter en parallèle avec la première invocation. Cette dernière situation est illustrée par la figure 6.

Dans la figure 6, si on invoque $test(4)$, l'objet 'a' transfère la responsabilité de réponse à l'objet 'b'. Ce dernier à son tour transfère la responsabilité de réponse à l'objet 'c' qui enfin peut retourner le résultat ($true$) à l'invoquant puisqu'il contient la valeur 4.

Après cette transformation, il est possible que les invocations de la méthode $test$ se terminent dans un ordre différent de celui où elles ont été acceptées. Par exemple, si $test(4)$ et $test(1)$ ont été invoquées par deux objets distincts, $test(4)$ peut être acceptée en premier mais la réponse peut parvenir d'abord à l'invoquant de $test(1)$. Cette modification de comportement ne peut être vue que par un observateur qui voit tout, y compris les actions internes de la méthode $test$. Mais, aucun programme $\pi o \beta \lambda$ ne peut la détecter. Les deux codes de la méthode $test$ sont donc observationnellement équivalents.

Ainsi, en appliquant les deux équivalences au code précédent (i.e, la première équivalence aux méthodes *add* et *rem*; la deuxième équivalence à la méthode *test*), on obtient le code suivant qui satisfait la spécification initiale et dont l'exécution est parallèle (plusieurs objets peuvent évoluer en parallèle):

```
Queue Class
  vars m: 𝒩 ← 0;
         l: PrivateRef(Queue) ← nil
  add(e: 𝒩) method
    return
    if l = nil then (m ← e; l ← new Queue)
    elif m ≺ e then l.add(e)
    else (l.add(m); m ← e)
    fi
  rem( ) method r: 𝒩
    return m
    if l ≠ nil then m ← l.rem()
          if m = 0 then l ← nil
          fi
    fi
  test(e: 𝒩) method r: ℬ
    if (l = nil or e ≺ m) then return false
    elif e = m then return true
    else delegate l.test(e)
    fi
```

Un autre exemple implémentant une table de symbôles par un arbre binaire peut être consulté dans [Jon93a].

5 Une méthode pour raisonner sur les interférences

Cette deuxième méthode de développement est destinée aux systèmes complexes où les risques d'interférence sont plus importants et ne peuvent être évités parce que le partage de données est nécessaire. Les graphes d'objets résultants sont complexes: DAG, ... Cliff Jones montre comment utiliser les rely/garantee-conditions pour spécifier et raisonner sur les interférences d'une manière qui permet de faire des développements compositionnels de programmes concurrents [Jon93b].

Le développement de ces applications suivra les mêmes phases que celles de la méthode précédente. Mais, les preuves seront plus compliquées à faire puisque d'une part, le développeur sera obligé d'analyser l'entrelacement des exécutions des objets. D'autre part, le graphe des objets ne permettra plus d'appliquer une induction structurelle.

5.1 La spécification

L'exemple étudié dans cette section est une implémentation concurrente de l'algorithme du crible d'Eratosthène qui calcule les entiers premiers inférieurs à un entier donné. Sa justification a été utilisée souvent dans la littérature pour illustrer différentes manières de raisonner sur la concurrence.

Dans l'algorithme du crible d'Eratosthène séquentiel, on considère la séquence des entiers inférieurs à un entier donné (n). Ensuite, on traite un par un les éléments de la séquence en éliminant tous ses multiples de la séquence. L'ensemble des éléments restants à la fin de l'algorithme constitue les nombres premiers inférieurs à n.

Dans l'implémentation parallèle de l'algorithme, on voudrait exécuter en parallèle les traitements correspondants aux différents entiers (i.e, les suppressions de leurs multiples). Mais, le problème qui se pose ici est que deux entiers voudraient supprimer un élément faisant partie de leurs multiples communs. Par exemple, si on veut calculer les entiers premiers inférieurs à 10, lors du traitement des entiers 2 et 3, on voudrait supprimer l'élément 6. Il faut donc gérer l'accès en exclusion mutuelle aux éléments de la séquence.

La spécification est écrite avec la notation VDM. En effet, dans VDM, chaque méthode possède une liste de variables lues dénotées par rd et une liste de variables modifiées dénotées par wr. Par défaut, les variables modifiées ne sont pas marquées wr.

Dans la spécification ci-dessous, les entiers premiers sont représentés par un ensemble. La méthode *new* construit l'ensemble des entiers premiers inférieurs à un entier donné n. La méthode *test* vérifie si un entier est premier (i.e, s'il appartient à l'ensemble). On suppose que la fonction *is-prime* est définie et qu'elle vérifie si un entier est premier.

$Primes\ Class$
$\quad vars\ max\colon \mathcal{N};\ sieve\colon \mathcal{N}\text{-}set$
$\quad new(n\colon \mathcal{N})\ method\ r\colon ref(Primes)$
$\quad\quad post\ r = self\ and\ max = n\ and\ sieve = \{2 <= i <= max \mid is\text{-}prime(i)\}$

$test(n\colon \mathcal{N})$ *method* $r\colon \mathcal{B}$
  pre $2 <= n <= max$
  post $r = (n \in sieve)$

Les instances de la classe *Primes* peuvent être créées par invocation de la méthode *new* (i.e, *new* $Primes(n)$) qui retourne une référence; appelons cette référence p.

Si la précondition de la méthode *test* est respectée, chaque processus connaissant la référence p peut invoquer la méthode *test* (i.e, $p.test(i)$) et obtenir en retour un booléen qui indique si l'entier i est premier ou pas.

Même si plusieurs files d'exécution concurrentes sont possibles, une seule méthode est active à un moment donné dans chaque instance de la classe *Primes*. Évidemment, plusieurs instances de la classe *Primes* peuvent être créées.

5.2 Le choix des structures de données

On peut représenter l'ensemble des entiers premiers (*sieve*) de la spécification abstraite par un tableau de booléens représentant sa fonction caractérisque. Cependant, un tableau n'offre aucune possibilité de distribution de l'algorithme car lors du traitement d'un élément i, on a besoin du tableau entier pour supprimer ses multiples (i.e, les éléments $mult(i)$). De plus, deux entiers distincts i et j voudraient tous les deux supprimer du tableau un de leurs multiples communs. Le tableau constitue donc une variable partagée par tous les processus $mult(i)$.

Une solution de conception consisterait à mettre chaque booléen du tableau dans une instance séparée d'une nouvelle classe *El*. Chaque instance offre une méthode pour supprimer l'élément qu'elle désigne de l'ensemble. Ces instances séparées fourniront alors un parallélisme potentiel car plusieurs processus peuvent s'exécuter maintenant en parallèle pour supprimer leurs multiples. De plus, si deux processus invoquent la méthode d'un même objet, une seule invocation sera considérée à la fois.

La fonction *map* associe à chaque entier une instance de la classe *El*:

$map\colon \mathcal{N} \rightarrow ref(El)$ **(1)**

El Class
  vars $b\colon \mathcal{B} \leftarrow true$
  test( ) *method* $r\colon \mathcal{B}$
    rd b
    return b
  del( ) *method*
    $b \leftarrow false$
    return

Les instances de la classe El sont initialisées à $true$ quand elles sont créées à l'aide de la méthode new. Seule la méthode del est disponible en modification restreignant ainsi les possibilités d'interférence. cette idée intuitive peut être formalisée par la formule suivante où $p.b$ signifie le contenu de la variable b après l'exécution de la méthode $test$ et $\overline{p.b}$ signifie son contenu avant l'exécution de la méthode $test$. La formule exprime que l'invocation de la méthode $test$ ne modifie pas la valeur de la variable b.

$p \in ref(El) \Rightarrow p.test()\ links\ (p.b \Leftrightarrow \overline{p.b})$

Cette formule peut être réécrite en utilisant l'opérateur $maintains$ en:

$p \in ref(El) \Rightarrow p.test()\ maintains\ p.b$

De manière similaire, la formule suivante exprime que l'invocation de la méthode del fait passer la valeur de la variable b à $false$:

$p \in ref(El) \Rightarrow p.del()\ confirms\ not(p.b)$ **(2)**

Comme la classe El contient uniquement les deux méthodes $test$ et del, un concepteur peut se baser sur un environnement e qui satisfait la formule suivante:

$e\ confirms\ not(p.b)$ **(3)**

Ainsi, grâçe à la formule **(2)**, on peut déduire:

$p \in ref(El) \Rightarrow p.del()\ fin\ not(p.b)$ **(4)**

On mémorise la fonction map de l'équation (1) dans la variable v d'une instance d'une nouvelle classe $Vector$ qui associe via sa méthode lu une référence d'une instance de El à chaque indice du vecteur.

Comme les références des instances de El doivent être retournées comme résultat de la méthode lu, elles doivent être de type $Shared$. Contrairement aux références de type $Private$, ces références peuvent être copiées.

Vector Class
  $vars\ max\colon \mathcal{N};\ v\colon \mathcal{N} \rightarrow SharedRef(El)$
  $new(n\colon \mathcal{N})\ method\ r\colon ref(Vector)$
    $post\ r = self\ and\ max = n\ and\ is\text{-}one\text{-}one(v)\ and$
        $\forall\ i \in \{2,..,max\},\ b(\sigma(v(i)))$
  $lu(n\colon \mathcal{N})\ method\ r\colon ref(El)$
    $rd\ max,\ v$
    $pre\ 2 <= n <= max$
    $post\ r = v(n)$

La condition que v doit être une injection, exprimée par la fonction *is-one-one*, assure qu'il y a une et une seule instance de la classe El par indice (i.e, par entier).

Maintenant, on va écrire la spécification des méthodes de la classe *Primes* en utilisant la nouvelle représentation. Avant cela, on va établir la correspondance entre la représentation concrète (*Vector*) et la représentation abstraite ($\mathcal{N}$-*set*).

On définit d'abord la fonction intermédiaire *rmap* qui prend en arguments une fonction qui associe à chaque entier la référence d'une instance de la classe El (rm) et l'état global (σ) et retourne une fonction qui associe à chaque entier la valeur booléenne contenue dans l'instance de la classe El qui lui est associée. La définiton de *rmap* utilise l'opérateur *dom*, qui appliqué à une fonction, retourne son domaine.

$rmap\colon\ (\mathcal{N} \rightarrow ref(El)) \rightarrow \Sigma \rightarrow (\mathcal{N} \rightarrow \mathcal{B})$

$rmap(rm)(\sigma) = \{i \mapsto b(\sigma(rm(i))) \mid i \in dom(rm)\}$

La fonction *retr* prend en arguments la référence d'une instance de la classe *Vector* (sr) et l'état global (σ) et retourne l'ensemble des indices du vecteur dont l'instance de la classe El associée contient la valeur booléenne vraie. Pour ce faire, elle fait appel à la fonction *rmap* qui construit la fonction associant à chaque indice la valeur booléenne contenue dans l'instance de la classe El qui lui est associée.

$retr\colon\ ref(Vector) \rightarrow \Sigma \rightarrow \mathcal{N}\text{-}set$

$retr(sr)(\sigma) = let\ m = rmap(v(\sigma(sr)))(\sigma)\ in\ \{i \in dom(m) \mid m(i)\}$

La nouvelle spécification de la classe *Primes* est la suivante:

$Primes\ Class$
  $vars\ max\colon \mathcal{N};\ sr\colon SharedRef(Vector)$
  $new(n\colon \mathcal{N})\ method\ r\colon ref(Primes)$
    $post\ r = self\ and\ max = n\ and\ retr(sr)(\sigma) = \{i \in \{2,..,max\} \mid is\text{-}prime(i)\}$
  $test(n\colon \mathcal{N})\ method\ r\colon \mathcal{B}$
    $return\ (sr.lu(n)).test()$

5.3 La décomposition des opérations

5.3.1 Décomposition de la classe Vector

La post-condition de la méthode *new* de la classe *Vector* peut être satisfaite si *new El* est invoquée pour initialiser chaque $v(i)$ ($i = 2..max$). Cela peut être réalisé grâçe à l'instruction *while* mais il est possible également d'utiliser l'instruction parallèle $\|$ de $\pi o\beta\lambda$ qui crée plusieurs files d'exécution concurrentes indépendantes. Comme chaque $v(i)$ est indépendant, aucune interférence n'est possible.

$Vector\ Class$

$vars\ max\colon \mathcal{N};\ v\colon \mathcal{N} \rightarrow SharedRef(El)$
$new(n\colon \mathcal{N})\ method\ r\colon ref(Vector)$
  $max \leftarrow n$
  $\|_{i \in \{2,..,max\}}\ v(i) \leftarrow new\ El$
  $return\ self$
$lu(n\colon \mathcal{N})\ method\ r\colon ref(El)$
  $pre\ 2 <= n <= max$
  $post\ r = v(n)$

5.3.2 Décomposition de la classe Primes

La méthode *new* de la classe *Primes* doit créer une instance de la classe *Vector* qui met par défaut la valeur *true* dans la variable b de chaque instance *El* pointée par chaque indice du vecteur. Elle doit faire en sorte que chaque entier non premier soit supprimé (la variable b pointée par l'indice correspondant du vecteur v est mise à $false$). Cette suppression peut être implémentée par deux boucles imbriquées: une boucle pour traiter chaque indice i du vecteur et une deuxième boucle pour supprimer les multiples de i dans le vecteur. Une telle approche séquentielle ne poserait aucun problème d'interférence.

La décision de conception qui est prise ici est d'utiliser des instances parallèles d'un processus *Rem*. Chaque instance $Rem(i, sr)$ supprime les entiers qui sont multiples de i de l'instance sr de la classe *Vector*.

Sachant que sr est partagée par les instances parallèles de la classe *Rem*, le graphe des objets construit est un DAG (Directed Acyclic Graph). Comme chaque indice du vecteur sr ne fait référence qu'à une seule instance de la classe *El* qui offre la méthode *del*, il est facile de voir qu'il n'y a aucun cycle dans les appels de la méthode *del* car la structure de données obtenue est linéaire.

Maintenant on va attaquer le problème de l'interférence. Le concepteur de la classe *Primes* doit concevoir et justifier la méthode *new* en termes de la spécification de la classe *Rem*, différant ainsi l'implémentation de la classe *Primes*. Afin de faciliter la compréhension de la spécification de la classe *Rem*, on va considérer d'abord le cas où aucune interférence n'est possible.

Soit i un entier inférieur à max et soit $mult(i)$ l'ensemble des mutilples de i. Une post-condition de la méthode *new* peut être la suivante: après le traitement de l'élément i, on a supprimé du vecteur sr exactement l'ensemble des éléments $mult(i)$:

$$retr(sr)(\overline{\sigma}) - retr(sr)(\sigma) = mult(i)$$

Mais, ceci est faux même en cas d'absence d'interférence, car un entier c non premier appartenant à $mult(i)$, que la i^{me} instance de *Rem* doit supprimer, pourrait être absent au moment de la suppression car il a déjà été supprimé par une invocation antérieure de *Rem* pour un indice j inférieur à i tel que c est aussi multiple de j.

Une post-condition correcte, en absence d'interférence, serait de dire qu'après le traitement d'un entier i, on a supprimé les multiples de i qui étaient encore présents dans le vecteur sr:

$$retr(sr)(\overline{\sigma}) \text{ - } retr(sr)(\sigma) = mult(i) \cap retr(sr)(\overline{\sigma}) \quad \textbf{(5)}$$

Si les instances de Rem s'exécutent en parallèle, l'interférence peut se produire et il est possible que cela ne supprime pas des éléments qui sont multiples de i dans l'équation **(5)**. Cela suggère qu'il faut reconsidérer les actions de $Rem(i, sr)$ en introduisant une contrainte dynamique:

$$new\ Rem(i, sr)\ links\ (retr(sr)(\overline{\sigma}) \text{ - } retr(sr)(\sigma) \subseteq mult(i)) \quad \textbf{(6)}$$

En utilisant $\|$-$links$, on peut conclure:

$$\|_i\ new\ Rem(i, sr)\ links\ (retr(sr)(\overline{\sigma}) \text{ - } retr(sr)(\sigma) \subseteq \bigcup_i mult(i)) \quad \textbf{(7)}$$

Mais, ceci n'est pas suffisant pour le concepteur de la classe $Primes$ car il est nécessaire de montrer que suffisamment d'éléments sont supprimés. En se référant à l'équation **(5)**, ce qui manque est une contrainte qui exprime que tous les multiples de i ont été supprimés:

$$new\ Rem(i, sr)/e\ fin\ (retr(sr)(\sigma) \cap mult(i) = \{\ \})$$

Mais, le concepteur de Rem sera incapable de construire une implémentation qui remplit cette condition à moins qu'une hypothèse ne soit donnée assurant que:

$$\begin{aligned} &e\ links\ (retr(sr)(\sigma) \subseteq retr(sr)(\overline{\sigma})) \Rightarrow \\ &\qquad new\ Rem(i, sr)/e\ fin\ (retr(sr)(\sigma) \cap mult(i) = \{\ \}) \quad \textbf{(8)} \end{aligned}$$

Comme l'environnement de $Primes$ ne peut référencer sr, il est possible d'utiliser la règle $\|$-fin pour conclure qu'à la fin du calcul, on ne garde que les entiers qui sont premiers:

$$new\ Primes\ fin\ (retr(sr)(\sigma) \cap \bigcup_i mult(i) = \{\ \})$$

La nouvelle spécification de la classe $Primes$ utilise l'opérateur rng, qui appliqué à une fonction, retourne l'ensemble image de la fonction.

$$\begin{aligned}
&Primes\ Class \\
&\quad vars\ max\colon \mathcal{N};\ sr\colon SharedRef(Vector) \\
&\quad new(n\colon \mathcal{N})\ method\ r\colon ref(Primes) \\
&\qquad max \leftarrow n \\
&\qquad sr \leftarrow new\ Vector(max) \\
&\qquad \{let\ m = rmap(v(\sigma(sr))(\sigma))\ in\ rng(m) = \{true\}\} \\
&\qquad \|_{i \in \{2,..,\sqrt{max}\}}\ new\ Rem(i, sr) \\
&\qquad \{let\ m = rmap(v(\sigma(sr))(\sigma))\ in \\
&\qquad\qquad \forall\ i \in \{2,..,max\},\ m(i) \Leftrightarrow is\text{-}prime(i)\} \\
&\qquad return\ self \\
&\quad test(n\colon \mathcal{N})\ method\ r\colon \mathcal{B} \\
&\qquad return\ (sr.lu(n)).test()
\end{aligned}$$

5.3.3 Décomposition de la classe Rem

Maintenant, on va écrire le code qui satifait la spécification de la classe *Rem* (voir les équations **(6)** et **(8)**).

Grâçe à l'équation **(2)** et en appliquant l'opérateur $\|$-*links*, on obtient:

$$\|_{m \in \{2,..,Ent(max/i)\}} \ (sr.lu(i*m)).del() \ \ links \ \ (retr(sr)(\overline{\sigma}) \ \text{-} \ retr(sr)(\sigma) \subseteq mult(i)) \ \ (9)$$

On peut remarquer que l'équation **(6)** est une conséquence de l'équation (9).

On peut encore appliquer l'opérateur $\|$-fin à la post-condition de l'équation **(8)**).

$Rem \ \ Class$
$\quad new(i\!: \mathcal{N}, \ sr\!: \ ref(Vector)) \ \ method$
$\qquad \|_{m \in \{2,..,Ent(max/i)\}} \ (sr.lu(i*m)).del()$
$\qquad \{let \ \ m \ = \ rmap(v(\sigma(sr))(\sigma)) \ \ in \ \ \forall \ \ c \ \in \ mult(i), \ \ not(m(c))\}$

5.4 Introduction de la concurrence

On peut appliquer la première règle de transformation à la méthode *del* de la classe *El*. Ainsi, l'instruction *return* est positionnée au début de la méthode *del*.

$El \ \ Class$
$\quad vars \ \ b\!: \ \mathcal{B} \leftarrow \ true$
$\quad test(\ ) \ \ method \ \ r\!: \ \mathcal{B}$
$\qquad return \ \ b$
$\quad del(:\ ) \ \ method$
$\qquad return$
$\qquad b \ \leftarrow \ \ false$

De même, la première règle de transformation peut être appliquée pour déplacer l'instruction *return* de la méthode *new* de la classe *Vector* et la mettre juste après la première instruction ($max \leftarrow n$).

$Vector \ \ Class$
$\quad vars \ \ max\!: \ \mathcal{N}; \ \ v\!: \ \mathcal{N} \ \rightarrow \ SharedRef(El)$
$\quad new(n\!: \ \mathcal{N}) \ \ method \ \ r\!: \ \ ref(Vector)$
$\qquad max \ \leftarrow \ \ n$
$\qquad return \ \ self$
$\qquad \|_{i \in \{2,..,max\}} \ v(i) \ \leftarrow \ \ new \ \ El$
$\quad lu(n\!: \ \mathcal{N}) \ \ method \ \ r\!: \ \ ref(El)$
$\qquad pre \ \ 2 \ <= \ n \ <= \ max$
$\qquad post \ \ r \ = \ v(n)$

6 Sémantique opérationnelle de $\pi\o\beta\lambda$

Dans un premier temps, une sémantique opérationnelle structurée (SOS) du langage de conception $\pi\o\beta\lambda$ a été définie en termes d'un système de transitions à deux niveaux: les transitions locales et les transitions globales. Cette sémantique permet de justifier les transformations et les preuves de programmes.

6.1 Les transitions locales

Ce premier niveau concerne les transitions internes à un objet. Ces transitions s'appuient sur un état contenant les valeurs des variables de l'objet. Cet état est modélisé par une fonction qui associe des valeurs aux variables de l'objet:

Ω: $Id \rightarrow Val$.

Les valeurs de base considérées sont les entiers naturels et les booléens. Les valeurs dans Oid correspondent aux références des objets créés:

$Val ::= \mathcal{B} \mid \mathcal{N} \mid Oid$.

Soit $Stmt$ l'ensemble des instructions de $\pi\o\beta\lambda$ (voir la syntaxe de $\pi\o\beta\lambda$ dans la section 2.2). Les transitions internes à un objet sont notées $\xrightarrow{s}$ et définissent une relation dans $(Stmt^* \times \Omega)$:

$\xrightarrow{s}$: $Stmt^* \times \Omega \rightarrow Stmt^* \times \Omega$.

Ainsi, si w_1 et w_2 sont des états (i.e, de type Ω) et si s_1 et s_2 sont des suites d'instructions (i.e, de type $Stmt^*$), alors la transition locale $(s_1, w_1) \xrightarrow{s} (s_2, w_2)$ signifie que l'exécution d'une action de la suite d'instructions s_1 dans l'état w_1 mène à l'exécution de la suite d'instructions s_2 dans l'état w_2.

Comme exemples, nous allons voir la sémantique de l'instruction d'affectation et la sémantique de l'instruction if-then-else.

Soit ω un état (i.e, de type Ω). Et soient l, S_t et S_f des suites d'instructions (i.e, de type $Stmt^*$). Dans ce qui suit, l'opérateur ';' dénote la séquence, l'opérateur '$\mapsto$' dénote la substitution d'une variable par une expression et l'opérateur '$[\![\ ]\!]$' dénote l'évaluation. Enfin, l'opérateur '$\dagger$' dénote la composition de substitutions.

6.1.1 Sémantique de l'affectation

$$\frac{}{((x \leftarrow e\, ;\, l)\ ,\, \omega) \xrightarrow{s} (l, \omega\ \{x \mapsto [\![e]\!]\ \omega\})}$$

L'exécution de l'instruction d'affectation $(x \leftarrow e)$ provoque la substitution dans l'état ω de la valeur de x par la valuation de l'expression e dans l'état ω. Toutes les autres variables apparaissant dans ω restent inchangées.

6.1.2 Sémantique de l'instruction If-then-else

$$\frac{[\![e]\!]\,\omega = true}{(((if\ e\ then\ S_t\ else\ S_f)\,;\,l),\omega) \xrightarrow{s} (S_t\,;\,l,\omega)}$$

$$\frac{[\![e]\!]\,\omega = false}{(((if\ e\ then\ S_t\ else\ S_f)\,;\,l),\omega) \xrightarrow{s} (S_f\,;\,l,\omega)}$$

Selon que la valuation de l'expression e dans l'état ω est $true$ ou $false$, on exécute respectivement la suite S_t ou la suite S_f.

6.2 Les transitions globales

Ce deuxième niveau concerne les transitions globales à plusieurs objets. Ces transitions ont besoin d'informations sur tous les objets qui ont été créés et qui constituent l'environnement du système. Cet environnement est modélisé par la structure de données **Omap** qui associe à chaque objet une structure de données $Oinfo$ contenant les informations caractérisant son état.

$Omap$ = $Oid \rightarrow Oinfo$

$Oinfo$::= cn: Id (identificateur de la classe de l'objet)
$status$: $Status$ (état d'exécution de l'objet)
$rest$: $Stmt^*$ (instructions a exécuter par la méthode active)
$state$: Ω (variables de l'objet et leurs valeurs)
$client$: Oid (référence de l'invoquant pour le retour du résultat)

$Status$ = Act (actif)
| $Quies$ (dormant)
| $Wait$ (en attente suite à l'appel d'une méthode)

$Wait$::= Id (variable qui reçoit le résultat de l'invocation)

Par exemple, si O est de type $Omap$ alors la structure $O(\alpha) = (c, Act, S, w, \beta)$ signifie qu'un objet de la classe c de nom α est à l'état actif, il est prêt à exécuter la suite d'instructions S dans l'état w et il est invoqué par un objet de référence β.

On a également besoin de la définition des classes. La structure de données **Cmap** associe à chaque classe une structure de données $Cdef$ contenant sa définition.

$Cmap$ = $Id \rightarrow Cdef$

$Cdef$::= vs : $Id \rightarrow Type$ (variables d'instance et leur type)
$init$: $Id \rightarrow Val$ (valeurs initiales des variables)
mm : $Id \rightarrow Mdef$ (méthodes de la classe)

Une structure de données $Mdef$ est associée à chaque méthode et fournit sa définition.

$Mdef$::= r: $Type$ (type du résultat retourné par la méthode)
pl: $(Id \times Type)$ (liste des paramètres de la méthode)
bo: $Stmt^*$ (corps de la méthode)

Les transitions globales sont marquées $\xrightarrow{g}$ et définissent une relation sur le couple ($Cmap$,$Omap$). Cette relation traduit l'évolution des objets du système par les modifications de l'environnement $Omap$.

$\xrightarrow{g}$: $Cmap \times Omap \rightarrow Cmap \times Omap \times Omap$

La sémantique du système entier est donnée par les transitions globales qui font évoluer la structure $Omap$. À l'état initial, $Omap$ contient une collection d'objets dont l'un est initialement actif; tous les autres objets sont à l'état dormant. Pour tous les objets, le champ *client* de la structure $Oinfo$ est initialisé à *nil*: aucun objet n'a encore été invoqué par un autre objet.

Comme exemples, nous allons voir d'abord comment une transition locale est perçue au niveau global. Ensuite, nous allons décrire la sémantique de l'instruction *new*, la sémantique de l'invocation d'une méthode et enfin la sémantique de l'instruction *return*.

Soient α et β deux références de deux objets distincts. Et soient C de type $Cmap$ un ensemble de classes et O de type $Omap$ une collection d'objets des classes définies dans C.

Les objets dans O doivent être obligatoirement des instances des classes définies dans C. Cette cohérence entre la définition des classes dans la structure C et la définition des objets dans la structure O est notée $C \vdash O$. Initialement, $Omap$ est cohérent avec $Cmap$.

6.2.1 Perception d'une transition locale au niveau global

La règle suivante montre comment une transition locale ($\xrightarrow{s}$) est perçue au niveau global ($\xrightarrow{g}$).

$$\frac{O(\alpha) = (c, Act, l, \omega, k)\ ,\ (l, \omega) \xrightarrow{s} (l_1, \omega_1)}{C \vdash O \xrightarrow{g} O\ \{\alpha \mapsto (c, Act, l_1, \omega_1, k)\}}$$

La transition locale (l, ω) vers (l_1, ω_1) dans l'objet α provoque la substitution de l et ω respectivement par l_1 et ω_1 dans la structure $Oinfo$ de l'objet α dans la structure O.

6.2.2 Sémantique de l'instruction new

$$\frac{O(\alpha) = (c_\alpha, Act, (v \leftarrow new\ c;\, l), \omega, k)\ ,\ \beta \notin dom(O)}{C \vdash O \xrightarrow{g} O\ \{\alpha \mapsto (c_\alpha, Act, l, \omega\ \{v \mapsto \beta\}, k)\}\ \dagger\ \{\beta \mapsto (c, Quies, [\,], init(C(c)), nil)\}}$$

La règle de réduction qui donne la sémantique de l'instruction *new* introduit une nouvelle référence β et crée une nouvelle entrée dans la structure O pour cette référence. L'objet créé

β est mis à l'état dormant ($Quies$), il n'a aucune instruction à exécuter au départ ([ ]) et ses variables d'instance sont initialisées à leurs valeurs par défault ($init(C(c))$). Initialement, cet objet n'est invoqué par aucun autre objet (le champ $client$ de la structure $Oinfo$ est mis à nil).

Quant à l'objet α qui a créé l'instance β de la classe c, il ajoute à son environnement un couple composé de la variable v de l'instruction d'affectation et de la référence β retournée par l'instruction new.

6.2.3 Sémantique de l'invocation d'une méthode

Cet exemple et celui qui suit illustrent deux transitions qui produisent un changement de l'état des objets.

$$\frac{O(\alpha) = (c_\alpha, Act, (r \leftarrow v.m()\,;\, l_\alpha), \omega_\alpha, k),\ \omega_\alpha(v) = \beta,\ O(\beta) = (c_\beta, Quies, [\,], \omega_\beta, nil)}{C \vdash O \xrightarrow{g} O\ \{\alpha \mapsto (c_\alpha, Wait, (r \leftarrow v.m()\,;\, l_\alpha), \omega_\alpha, k)\}}$$

$$\dagger\ \{\beta \mapsto (c_\beta, Act, body(mm(C(c_\beta))(m)), \omega_\beta, \alpha)\}$$

L'invocation de la méthode m par l'objet actif α provoque son passage à l'état $Wait$; tandis que l'invoqué (β) qui était à l'état dormant passe à l'état actif pour exécuter le corps de la méthode m. On récupère dans la structure C le corps de la méthode m de la classe c_β: $body(mm(C(c_\beta))(m))$. La référence α de l'appelant est mémorisée dans le champ $client$ de la structure $Oinfo$ associée à β pour lui retourner le résultat de la méthode m.

6.2.4 Sémantique de l'instruction return

$$\frac{O(\alpha) = (c_\alpha, Wait, (r \leftarrow v.m()\,;\, l_\alpha), \omega_\alpha, k)\ ,\ O(\beta) = (c_\beta, Act, (return(e)\,;\, l_\beta), \omega_\beta, \alpha)}{C \vdash O \xrightarrow{g} O\ \{\alpha \mapsto (c_\alpha, Act, l_\alpha, \omega_\alpha\ \{r \mapsto [\![e]\!]\omega_\beta\}, k)\}\ \dagger\ \{\beta \mapsto (c_\beta, Act, l_\beta, \omega_\beta, nil)\}}$$

L'exécution de l'instruction $return$ par β provoque le déblocage de α qui repasse à l'état actif, la variable r qui devait recevoir le résultat est substituée dans l'environnement de α par la valeur retournée par β (i.e, par la valuation de e dans l'environnement de β). Le champ $client$ de la structure $Oinfo$ associée à β est remis à nil; mais l'objet β reste à l'état actif s'il a d'autres instructions à exécuter en parallèle avec l'exécution de l'objet α.

À partir de la SOS, la concurrence du langage $\pi\o\beta\lambda$ est maintenant claire car plusieurs objets peuvent évoluer en parallèle à un moment donné. Mais, l'ordre dans lequel ils évoluent n'est pas deterministe.

Enfin, la validité des deux règles de transformation utilisées pendant le développement (le déplacement de l'instruction $return$ et l'utilisation de l'instruction $delegate$) a été prouvée en se basant sur la sémantique SOS de $\pi\o\beta\lambda$ et en utilisant un argument de confluence. Les preuves complètes sont décrites dans [HJ96].

7 Sémantique π-calcul de $\pi\emptyset\beta\lambda$

Des difficultés ont été rencontrées lors de la définition de la sémantique SOS, elles sont expliquées dans la section 8.4. À cause de cette raison, une deuxième sémantique du langage de conception $\pi\emptyset\beta\lambda$ a été donnée dans le π-calcul polyadique de premier ordre.

Le π-calcul [Mil89] est un calcul de processus où les processus communiquent par envoi/réception de messages via des canaux synchrones. C'est un calcul de processus mobiles car la communication peut modifier les liaisons entre les processus contrairement à CCS [Mil89b], CSP [Hoa85] et les réseaux de Pétri [Rei83] où cette mobilité ne peut être exprimée directement.

Dans le π-calcul monoadique, les processus peuvent émettre ou recevoir un seul message à la fois; tandis que dans le π-calcul polyadique [Mil91], plusieurs messages peuvent être émis ou reçus en même temps sur le même canal. Le π-calcul polyadique se code de manière simple dans le π-calcul monoadique.

7.1 Syntaxe et sémantique informelle du π-calcul polyadique

Soient $x, y, z, u, v, ... \in \mathcal{N}$ (ensemble des noms).

On dénote par $\tilde{y}$ le n-uplet $y_1\ y_2\ ...\ y_n$.

Soient $A, B, ...$ des processus qui peuvent être les suivants:

- 0, un processus qui ne fait rien.
- $\overline{y}\widetilde{x}.P$, un processus qui envoie le n-uplet $\widetilde{x}$ sur le canal y, ensuite se comporte comme P. $\overline{y}\widetilde{x}$ est appelé **préfixe négatif**.
- $y(\widetilde{x}).P$, un processus qui reçoit un n-uplet sur le canal y, ensuite se comporte comme P dans lequel les variables x_1,...,x_n sont remplacées par les projections du n-uplet reçu. $y(\widetilde{x})$ est appelé **préfixe positif**.
- $\tau.P$, un processus qui fait une transition silencieuse τ et se comporte ensuite comme P.
- $P + Q$, un processus qui se comporte comme P ou comme Q.
- $P \mid Q$, un processus représentant la composition parallèle de P et Q. Les processus P et Q peuvent évoluer séparement. De plus, les communications entre P et Q peuvent se produire si l'un des processus émet sur un canal et l'autre reçoit sur le même canal. Ces communications se traduisent à l'extérieur par des transitions silencieuses τ.
- $(\nu\ x)P$, un processus qui agit comme P mais le nom x est privé à P; x ne peut être utilisé comme canal dans les communications avec l'environnement du processus (i.e, avec les autres processus). Le nom x est donc lié dans P (i.e, x est un nom local dont la portée est P).

- $[x = y]P$, si x et y sont identiques, le processus se comporte comme P, sinon il ne fait rien.

- $!\,P$, un processus qui exécute indéfiniment les actions de P.

- $A(y_1, ..., y_n)$ est un processus si A est un identificateur de processus d'arité n défini par une équation de la forme $P = A(x_1, ..., x_n)$, où les noms x_1, ..., x_n sont des noms distincts (représentant des paramètres) et sont les seuls noms libres dans P. Le processus $A(y_1, ..., y_n)$ se comporte comme P dans lequel chaque x_i a été substitué par y_i $(i = 1..n)$. Les x_i peuvent être considérés comme les paramètres formels de A; tandis que les y_i sont les paramètres d'appel dans $A(y_1, ..., y_n)$.

  Ainsi, il est possible de coder une fonction ayant un nombre fini de paramètres par un processus dans le π-calcul. De plus, les identificateurs de processus fournissent la récursion car l'équation de définition de A peut contenir A.

On écrira $(\nu\ x_1...x_n)P$ au lieu de $(\nu\ x_1)...(\nu\ x_n)P$. Et on introduit aussi un nouveau préfixe:

$\overline{x}(\tilde{y}).P = (\nu\ \tilde{y})\overline{x}\tilde{y}.P$ (on introduit un nouveau n-uplet de noms $\tilde{y}$ et on émet $\tilde{y}$ sur le canal x).

La syntaxe complète du π-calcul polyadique peut être consultée dans [Mil91].

7.2 Le codage des booléens

Le codage est fait de manière analogue au codage des booléens dans le λ-calcul où la valeur $True$ est codée par la fonction $\lambda x\ y.x$ et la valeur $False$ est codée par la fonction $\lambda x\ y.y$.

- La valeur $True$ est codée par le processus $[\![True]\!](b_t)$ qui attend deux noms de canaux t et f sur le canal de nom b_t, émet un signal sur le premier canal t et retourne à son état initial (ci-dessous, $\overline{t}$ désigne l'émission d'un signal sur le canal t, c'est une émission sans paramètre). On dira que la valeur booléenne $True$ est pointée par le canal de nom b_t.

  $[\![True]\!](b_t) \equiv b_t(t\ f).\overline{t}.[\![True]\!](b_t)$

  On peut faire l'analogie avec la fonction $\lambda x\ y.x$ qui appliquée à 2 valeurs a et b $((\lambda x\ y.x)a\ b)$ rend la valeur a. L'application dans le π-calcul est la composition parallèle de processus, un exemple est illustré plus loin.

- La valeur $False$ est codée par le processus $[\![False]\!](b_f)$ qui attend deux noms de canaux t et f sur le canal de nom b_f, émet un signal sur le deuxième canal f et retourne à son état initial. De même, on dira que la valeur booléenne $False$ est pointée par le canal de nom b_f.

  $[\![False]\!](b_f) \equiv b_f(t\ f).\overline{f}.[\![False]\!](b_f)$

Maintenant, on peut coder des fonctions sur les booléens.

- La fonction *copy_bool* ci-dessous copie la valeur booléenne pointée par un canal y (i.e, $True$ ou $False$) dans un autre canal z. Elle introduit deux nouveaux noms u et v et envoie ces deux noms sur le canal y. Si elle reçoit un signal sur le canal u, elle crée un processus $[\![True]\!](z)$. Sinon, elle crée un processus $[\![False]\!](z)$. L'envoi des deux canaux u et v sur le canal y et la réception d'un signal sur l'un de ces deux canaux correspond à l'évaluation de y.

  $$\mathit{copy\_bool}(y, z) \;=\; (\nu\ u\ v)\ \overline{y}\ u\ v.(u().[\![True]\!](z)\ +\ v().[\![False]\!](z))$$

  Supposons que le canal y pointe la valeur $True$, on a alors un processus de la forme $y(t\ f).\overline{t}$. Si on veut copier la valeur pointée par y dans un autre canal z, on doit composer ce processus avec le processus correspondant à la fonction *copy_bool* appliquée aux canaux y et z. On obtient alors:

  $[\![True]\!](y) \mid \mathit{copy\_bool}(y, z)$

  $\equiv y(t\ f).\overline{t}.[\![True]\!](y) \mid (\nu\ u\ v)\ \overline{y}\ u\ v.(u().[\![True]\!](z) + v().[\![False]\!](z))$

  Les noms u et v ne sont pas libres dans la partie gauche de la composition parallèle. On peut alors étendre leur portée à la composition parallèle.

  $\equiv (\nu\ u\ v)\ (y(t\ f).\overline{t}.[\![True]\!](y) \mid \overline{y}\ u\ v.(u().[\![True]\!](z) + v().[\![False]\!](z)))$

  On peut réaliser la communication le long du canal y, ce qui permet de remplacer les noms t et f respectivement par u et v dans la partie gauche de la composition. Cette communication est traduite par une transition τ.

  $\rightarrow \tau.(\nu\ u\ v)\ (\overline{u}.[\![True]\!](y) \mid (u().[\![True]\!](z) + v().[\![False]\!](z)))$

  Une deuxième communication est possible le long du canal u.

  $\rightarrow \tau.\tau.(\nu\ u\ v)\ ([\![True]\!](y) \mid [\![True]\!](z))$

  Or u et v n'apparaissent plus dans la composition parallèle, on supprime alors la restriction sur u et v.

  $\rightarrow \tau.\tau.([\![True]\!](y) \mid [\![True]\!](z))$

  On remarque donc que le résultat de $\mathit{copy\_bool}(y, z)$ est un processus qui pointe la valeur $True$ par le canal z. La valeur pointée par le canal y est inchangée.

- La fonction and évalue l'expression ($c\ and\ d$), où c et d sont deux expressions booléennes. On évalue d'abord c et d en parallèle. Leurs valeurs sont pointées respectivement par les canaux l_1 et l_2 (deux nouveaux noms). Sur l_1, on attend le nom b_1 qui localise la valeur de c. Ensuite, on émet deux nouveaux noms u et v sur b_1. Si u est activé (l'évaluation de c retourne $true$), on attend sur l_2 le nom b_2 qui localise la valeur de d et on copie la valeur pointée par b_2 dans le canal résultat l. Sinon, on copie la valeur $false$ dans le canal l.

  $[\![c\ and\ d]\!]\ l$ = $(\nu\ l_1\ l_2)([\![c]\!](l_1)$ `(évaluation de` c`, le résultat est dans` l_1`)`

$| \; [\![d]\!](l_2)$ `(évaluation de` d`, le résultat est dans` l_2`)`

$| \; l_1(b_1).(\nu \;\; u \;\; v)(\overline{b_1} \;\; u \;\; v.(u().l_2(b_2).copy\_bool(b_2, l) \; + \; v().[\![False]\!](l))))$

`(selon la valeur de` c`, on envoie la valeur de` d `ou` $false$ `sur` l`)`

On peut coder aussi des instructions dans le π-calcul. Ainsi, l'instruction $If \; e \; then \; P$ $else \; Q$, e étant une expression booléenne, peut être codée par le processus:

$$(\nu \;\; b)([\![e]\!] \;\; b \mid (\nu \;\; u \;\; v)(\overline{b} \;\; u \;\; v.(u().P \; + \; v().Q)))$$

On évalue d'abord l'expression e, le résultat est pointé par un nouveau canal b. Parallèlement, on envoie deux nouveaux noms u et v sur b. Si on reçoit un signal sur u (ce qui veut dire que e vaut $true$), on exécute P sinon on exécute Q.

Ce codage s'appuie sur le fait qu'il y a un nombre fini de valeurs booléennes (un canal est réservé pour chaque valeur), ce qui permet d'utiliser la somme finie de processus pour traiter les différentes valeurs possibles.

Pour plus de détails, la sémantique d'autres instructions simples telles que la séquence, while, ... peut être consultée dans [Jon92].

7.3 Le codage des entiers naturels

Les entiers naturels sont infinis, on ne peut les coder de la même manière que les booléens (i.e, en utilisant une somme finie de processus). On va donc les coder inductivement par le zéro et le successeur (par analogie aux entiers de Church).

- L'entier zéro est codé par le processus $[\![0]\!](l)$ qui attend deux noms de canaux z (pour zéro) et s (pour successeur) sur le canal de nom l et émet un signal sur le premier canal z. On dira que l'entier zéro est pointé par le canal l.

  $[\![0]\!](l) = l(z \; s).\overline{z}$

- L'entier $succ(n)$ est codé par le processus $[\![succ(n)]\!](l)$. Ce processus est la composition parallèle d'un processus qui définit n ($[\![n]\!](l')$), n est pointé par un nouveau nom l'. L'autre processus attend sur l deux noms de canaux z et s et envoie l' sur le canal s. Autrement-dit, on envoie sur le canal s le canal l' qui pointe le prédécesseur n.

  $[\![succ(n)]\!](l) = (\nu \; l')(l(z \; s).\overline{s} \; l' \mid [\![n]\!](l'))$

Avec cette représentation, l'entier 1 est codé par le processus $(\nu \; l')(l(z \; s).\overline{s} \; l' \mid l'(z \; s).\overline{z})$.

En utilisant cette définition inductive, on peut définir les opérateurs arithmétiques sur les entiers naturels. Par exemple, la fonction $copy$ ci-après effectue la copie d'un entier pointé par un canal l dans un autre canal m.

$$copy(l, m) \; = \; (\nu \;\; u \;\; v)\overline{l} \;\; u \;\; v.(u().[\![0]\!](m) \; + \; v(l').(\nu \;\; m')(m(z \;\; s).\overline{s} \;\; m' \mid copy(l', m')))$$

La fonction *copy* introduit deux nouveaux noms u et v et envoie ces deux noms sur le canal l. Si elle reçoit un signal sur le canal u (la valeur pointée par le canal l est zéro), elle crée un processus $[\![0]\!](m)$ (i.e, copie zéro dans m). Sinon, elle reçoit un nom l' sur le canal v (il s'agit d'un entier $succ(n)$, n étant pointé par le canal l'). Dans ce cas, elle introduit un nouveau nom m' dans lequel elle copie la valeur pointée par l'. Parallèlement, elle attend deux noms z et s sur le canal m et envoie m' sur s. Autrement-dit, $succ(n)$ est maintenant pointé par le canal m sachant que n est pointé par le canal m' (définition inductive de la copie).

La fonction *add* effectue l'addition de deux entiers pointés respectivement par les canaux m et n, le résultat est pointé par le canal res.

$$\begin{aligned} add(m,n,res) \texttt{ = } (\nu \;\; u \;\; v)\overline{m} \;\; u \;\; v.(&u().copy(n,res) \\ &+ \; v(m').(\nu \;\; res')(res(z \;\; s).\overline{s} \;\; res' \mid add(m',n,res'))) \end{aligned}$$

La fonction *add* introduit deux nouveaux noms u et v et envoie ces deux noms sur le canal m. Si elle reçoit un signal sur le canal u (la valeur pointée par le canal m est zéro), elle effectue la copie de l'entier pointé par le canal n dans le canal res. Sinon, elle reçoit un nom m' sur le canal v (il s'agit d'un entier $succ(a)$, a étant pointé par le canal m'). Dans ce cas, elle introduit un nouveau nom res' qui pointera le résultat (b) de l'addition des entiers pointés par m' et n. Parallèlement, elle attend deux noms z et s sur le canal res et émet le canal res' sur s. Autrement-dit, $succ(b)$ est la valeur pointée par le canal res sachant que b est pointé par le canal res'.

7.4 Sémantique des classes sans variables

Considérons la définition de la classe $\pi\o\beta\lambda$ suivante:

C *class*
  $m1(x)$ *method*
    return
  $m2(\;)$ *method* r: *ref*
    return self

La création d'instances multiples de la classe C est modélisée par l'itération pour fournir une ressource non bornée. Pour chaque instance I_C créée, on introduit un nouveau nom u qui est émis sur un canal c pour signaler la création de l'instance.

$[\![C]\!]$ `= !` I_C `(définition de la classe` C`: itération du processus` I_C`)`

I_C `=` $(\nu \;\; u)$ $(\overline{c} \;\; u.B_u)$ `(création des instances de` C`)`

Une fois l'instance u créée, elle émet sur le canal u deux nouveaux noms α_1 et α_2 pour signaler qu'elle est prête à recevoir les appels des méthodes $m1$ et $m2$ respectivement sur ces deux canaux.

B_u `=` $(\nu \;\; \alpha_1 \;\; \alpha_2)(\overline{u} \;\; \alpha_1 \;\; \alpha_2.M_u)$

$M_u = (\alpha_1(w_1\ \ x).\overline{w_1}.B_u\ +\ \alpha_2(w_2).\overline{w_2}\ \ u.B_u)$

Le premier terme de la somme dans M_u désigne la réception sur α_1 d'un appel de la méthode $m1$ avec un nom de canal w_1 pour le retour du résultat et un argument x. La méthode $m1$ ne fait que signaler sa terminaison en activant le canal w_1 (elle ne retourne pas de résultat). Ensuite, elle se remet en attente d'un autre appel de méthode.

Le second terme désigne la réception sur α_2 d'un appel de la méthode $m2$ avec seulement un nom de canal w_2 pour le retour du résultat ($m2$ n'a pas de paramètre). La méthode $m2$ retourne la référence de l'objet courant contenue dans u en l'envoyant sur le canal w_2. Ensuite, elle se remet en attente d'un autre appel de méthode.

Donc, à chaque nom α_i d'une méthode m_i (réservé à la réception des invocations de la méthode), est associé un nom de terminaison w_i qui est utilisé pour signaler la fin du rendez-vous et pour contenir le résultat retourné par la méthode.

Côté appelant, l'invocation de la méthode $m1$ doit se faire ainsi:

$u(\alpha_1\ \ \alpha_2).(\nu\ \ w_1)(\overline{\alpha_1}\ \ w_1\ \ e.w_1())$

On attend sur le canal u les deux noms α_1 et α_2 (i.e, l'instance u est prête à recevoir des appels sur ces deux canaux). Ensuite, on introduit un nouveau nom w_1 qui va contenir le résultat d'appel de la méthode $m1$ puis on envoie sur α_1 l'argument e et w_1. Comme $m1$ ne retoune aucun résultat, aucun message n'est attendu sur w_1 (on attend uniquement le signal de terminaison de $m1$).

L'invocation de la méthode m2 doit se faire ainsi:

$u(\alpha_1\ \ \alpha_2).(\nu\ \ w_2)(\overline{\alpha_2}\ \ w_2.w_2(u'))$

On attend sur le canal u les deux noms α_1 et α_2. Ensuite, on introduit un nouveau nom w_2 qui va contenir le résultat d'appel de la méthode $m2$. Aucun argument n'est émis sur α_2 car la méthode $m2$ ne possède pas de paramètre; on envoie uniquement w_2 sur α_2. Par contre, l'invoquant attend la référence de l'objet courant u' sur le canal w_2.

Ce codage illustre bien le fait que les objets s'exécutent en parallèle puisque chaque objet créé possède des canaux privés α_i pour recevoir et traiter indépendamment les invocations de ses méthodes m_i (voir les définitions de I_C et de B_u). Il montre aussi qu'une seule méthode est active dans un objet à un moment donné car une seule réception sur l'un des canaux α_i est traitée à la fois. Ceci est exprimé par la somme de processus dans la définition de M_u.

7.5 Sémantique des classes avec variables

L'exemple précédent concernait une classe sans variables d'instance. Nous allons voir les adaptations à faire quand des variables d'instance sont définies dans la classe, notamment dans le cas où ces variables sont des références.

7.5.1 Variables d'instance booléennes

$Bit\ Class$
  $vars\ v\colon\ \mathcal{B} \leftarrow false$
  $write(x\colon\ \mathcal{B})\ method$
    $v \leftarrow x$
    $return$
  $read(\ )\ method\ r\colon\ \mathcal{B}\ return\ v$

Cette classe est modélisée comme dans l'exemple précédent; par contre, chaque variable d'instance v est représentée par un processus somme V qui s'exécute en parallèle avec l'instance et qui va gérer l'accès à la variable en lecture et en écriture. Le processus V est activé en initialisant le contenu de la variable v à $false$.

En effet, comme tout se fait par communication dans le π-calcul, la variable d'instance est traitée elle aussi comme une classe simple (sans variables) pour laquelle une seule instance est créée. Cette classe particulière a deux méthodes: une méthode d'accès au contenu de la variable (a_v) et une méthode de modification de son contenu (s_v) dont l'interface est simple car elles ne sont invoquées que par une seule instance.

$[\![Bit]\!]$ = $!\ IBit$

$IBit$ = $(\nu\ u)(\overline{c}\ u.(\nu\ s_v\ a_v)(B_u\ |\ V(b_f)))$

B_u = $(\nu\ \alpha_w\ \alpha_r)(\overline{u}\ \alpha_w\ \alpha_r.M_u)$

M_u = $(\alpha_w(w_w\ x).\overline{s_v}\ x.\overline{w_w}.B_u\ +\ \alpha_r(w_r).a_v(y).\overline{w_r}\ y.B_u)$

La réception d'un appel de la méthode $write$ sur le canal α_w avec un nom de terminaison w_w pour le retour de l'appel et un argument x déclenche d'abord l'appel de la méthode s_v en émettant l'argument x sur le canal s_v pour modifier le contenu de v. Ensuite, la terminaison est signalée sur w_w. Enfin, la méthode $write$ se remet en attente d'un autre appel de méthode.

La réception d'un appel de la méthode $read$ sur le canal α_r avec seulement un nom de terminaison w_r pour le retour de l'appel déclenche d'abord l'appel de la méthode a_v pour lire le contenu de v. Ceci est traduit par la réception du message y sur le canal a_v. Ensuite, y est retourné comme résultat sur le canal w_r. Enfin, la méthode $read$ se remet en attente d'un autre appel de méthode.

$V(y)$ = $(\overline{a_v}\ y.V(y)\ +\ s_v(z).V(z))$

Le processus $V(y)$ est la somme de deux processus: soit le contenu y de la variable v est émis sur le canal a_v, soit la valeur de z reçue sur le canal s_v est affectée à la variable v. Dans les deux cas, le processus revient à son état initial. Plus généralement, l'accès au contenu d'une variable v et la modification de son contenu sont codés de la façon suivante:

$[\![v]\!]\ l$ = $a_v(x).\overline{l}\ x$ (lecture du contenu de la variable v, la valeur x reçue dans a_v est retournée comme résultat dans le canal l)

$[\![v \leftarrow e]\!]$ = $(\nu\ l)([\![e]\!]l\ |\ l(b).\overline{s_v}b)$ (affectation de la valuation de e à la variable v en l'émettant sur le canal s_v)

7.5.2 Variables de type référence

Les variables d'instance de type référence mémorisent des noms d'objets tout comme les variables vues dans le paragraphe précédent contiennent des valeurs booléennes. Le principe de codage est donc le même.

La valeur *nil* indique que la référence n'a pas encore été initialisée. Si une référence contenant la valeur *nil* est utilisée alors ce cas d'erreur peut être traité en utilisant un nom privé dédié à cette erreur. L'utilisation de ce nom privé provoquera la réduction à l'echec. Des mécanismes de traitement d'exception peuvent être programmés dans des langages plus riches que $\pi\o\beta\lambda$.

Pour plus de détails concernant la sémantique π-calcul de $\pi\o\beta\lambda$, voir [Jon92], [Jon93c] et [Jon95].

Enfin la validité des règles de transformation de programmes a été démontrée en se basant sur la sémantique π-calcul de $\pi\o\beta\lambda$. Les preuves sont décrites dans [PW96].

8 Conclusions et perspectives

8.1 Le langage de conception

Le langage $\pi\o\beta\lambda$ n'offre pas:

- l'héritage car il est très difficile de traiter l'héritage en présence de la concurrence,
- l'appel local de méthode: une méthode m_1 ne peut invoquer une autre méthode m_2 définie dans la même classe; en effet, la méthode m_2 peut être invoquée de l'extérieur par un autre objet, ce qui complique le contrôle des interférences,
- le déclenchement conditionnel de méthode: cette notion sert à déclencher une méthode uniquement quand une condition est vérifiée sur l'état des variables de l'objet, ce qui permet d'imposer certaines contraintes à l'environnement qui invoque la méthode afin de lui garantir une certaine propriété,
- le traitement des exceptions: en cas d'erreurs, il faut lever des exceptions; aucun mécanisme de traitement d'exception n'est fourni pour l'instant par le langage.

Il est prévu d'étendre le langage avec ces différents aspects. Il est également prévu d'implémenter les communications entre les objets par le biais de canaux synchrones puisque les invocations de méthodes se font par rendez-vous. Enfin, il est envisagé de contraindre l'ordre dans lequel les méthodes d'une classe peuvent être invoquées, ce qui permettra de mieux contrôler les interactions entre les objets et de limiter les interférences.

8.2 La logique de preuves

Dans la spécification, il n'est pas possible d'avoir une vue abstraite d'un objet où les variables sont invisibles puisque les méthodes sont spécifiées en termes de pre/post-conditions et/ou de rely/garantee-conditions. Et pour exprimer les pre/post-conditions ou les rely/garantee-conditions on a besoin d'utiliser les variables.

8.3 Les méthodes de développement

Cliff Jones propose deux méthodes de développement compositionnelles qui procèdent par transformation de la spécification:

1. la première méthode est destinée aux systèmes concurrents où les risques d'interférence sont très limités et peuvent donc être évités en structurant les données de l'application et en utilisant les concepts orientés-objet. On peut alors modéliser les objets de l'application par une structure de données inductive (liste chaînée, arbre binaire, ...) de façon à éviter le partage de données. Ainsi, grâçe aux propriétés de chaque objet, on peut obtenir les propriétés du système en appliquant une induction structurelle sur la structure inductive obtenue.

   Cependant, cette méthode ne peut être appliquée qu'à des systèmes qui sont très peu concurrents.

2. la deuxième méthode est destinée aux systèmes complexes où les risques d'interférence sont plus importants et ne peuvent être évités parce que le partage de données est nécessaire. Les graphes d'objets résultants sont complexes: DAG, ... Cliff Jones montre comment utiliser les rely/garantee-conditions pour spécifier et raisonner sur l'interférence dans une manière qui permet de faire des développements compositionnels de programmes concurrents.

   Mais dans ce cas, les preuves sont plus difficiles à faire car d'une part, le développeur est confronté au problème classique de recherche des invariants et lemmes intermédiaires et d'autre part, il est obligé d'analyser l'entrelacement entre les objets qui interfèrent.

Dans les deux méthodes, lors de l'application de la première règle de transformation qui consiste à déplacer l'instruction *return* dans une méthode, il est possible que l'exécution de la méthode qui doit se poursuivre échoue. Dans ce cas, les exceptions ne peuvent pas être levées lors de l'exécution de la méthode puisqu'on ne peut pas avertir l'invoquant qui a déjà été relâché.

De même, lors de la délégation du retour du résultat de l'appel d'une méthode (deuxième règle de transformation), si une erreur se produit, il faut être capable de remonter l'erreur à la méthode responsable de la délégation, motamment dans le cas où plusieurs délégations en cascades ont eu lieu.

Aucun mécanisme ne permet de traiter ces problèmes actuellement dans $\pi\o\beta\lambda$. Toutefois, il est prévu d'implémenter les exceptions dans une nouvelle version du langage.

8.4 La sémantique SOS de $\pi\o\beta\lambda$

1. Points positifs

   - La traduction SOS est naturelle et est facile à comprendre.
   - Dans la SOS, on peut connaître les interférences qui ne peuvent pas se produire connaissant les objets qui sont prêts à évoluer à un moment donné. Par contre, on ne sait rien sur les interférences qui peuvent se produire à cause de l'indéterminisme.
   - La stratégie de preuves utilisée pour prouver les deux équivalences appliquées lors des transformations de programmes est naturelle; elle consiste à faire une induction sur les pas de réduction de la SOS [Jon96b].

2. Points négatifs

   - Les preuves des deux équivalences citées précédemment ont été basées sur la définition SOS du langage $\pi\o\beta\lambda$. La première reproche concernant l'utilisation de la SOS réside dans le fait qu'il n'y a pas d'algèbre naturelle pour cette définition qui permettrait de parler d'équivalence entre processus.

     Cette difficulté a été surmontée par l'introduction de certains lemmes nécessaires aux preuves et en utilisant un argument de confluence. Mais de nombreux points, tels que l'équité entre les processus d'un système, restent à étudier de manière très approfondie.

- La SOS force à définir la sémantique à un niveau de granularité bas alors qu'on spécifie à un niveau d'abstraction plus haut. Il est alors impossible, dans les spécifications, d'avoir une vue abstraite d'un objet où les variables sont invisibles.
- Enfin, il est difficile de prouver qu'un code ayant un niveau de granularité plus haut a le même comportement qu'un autre code ayant un niveau de granularité bas lors des transformations de programmes.

8.5 La sémantique π-calcul de $\pi\o\beta\lambda$

1. Points positifs

   - Dans le π-calcul, toute entité est identifiée par un nom ce qui facilite la manipulation des objets.
   - La traduction des instructions $\pi\o\beta\lambda$ dans le π-calcul est immédiate ce qui n'était pas le cas pour la SOS puisqu'il a fallu introduire les structures de données $Cmap$, $Omap$, un état global, ...
   - les lois algébriques du π-calcul offrent l'équivalence observationnelle entre processus qui est utilisée pour démontrer la validité des règles de transformation.

2. Points négatifs

   - Les preuves des équivalences appliquées lors des transformations de programmes (i.e, le déplacement de l'instruction $return$ et l'utilisation de l'instruction $delegate$) ont été faites pour des exemples spécifiques où les équivalences sont préservées. Les preuves de ces deux équivalences doivent être fournies dans un cadre général. Seules des preuves informelles ont été données dans [Jon93c] et [Jon94]. Les preuves complètement formelles sont difficiles à faire à cause de l'indéterminisme dans $\pi\o\beta\lambda$.

     Cliff Jones s'inspire actuellement des résultats généraux obtenus dans la SOS pour fournir ces preuves dans le π-calcul.
   - On pourrait aussi reprocher à la sémantique π-calcul de $\pi\o\beta\lambda$ de donner une sémantique de bas niveau. Mais, la différence par rapport à la SOS, c'est que le π-calcul possède une algèbre qui permet de tester l'équivalence entre les processus (la bisimulation).

Références

[Ame89] Issues in the design of a parallel object-oriented language. Pierre America, Formal aspects of computing, 1(4), 1989.

[CM88] Parallel program design: A foundation, K.M.Chandy and J.Misra, Addison-Wesley,1988.

[Hoa85] Communicating Sequential Processes, C.A.R Hoare, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HJ96] Non-interference properties of a concurrent object-based language: proofs based on an operational semantics, S.J.Hodges and C.B.Jones, 1996.

[Jon81] Development methods for computer programs including a notion of interference. C.B.Jones, PhD thesis, Oxford University, Juin 81.

[Jon90] Systematic Software Development using VDM. C.B.Jones, Prentice Hall International, second edition, 1990.

[Jon91a] Interference resumed, C.B.Jones in P.Bailes, editor, Engineering Safe Software, pages 31-56, Australian Computer Society, 1991.

[Jon92] An Object-Based design method for concurrent programs, C.B.Jones, Technical report UMCS-92-12-1, University of Manchester.

[Jon93a] Constraining interference in an object-based design method, C.B.Jones, TAPSOFT'93, LNCS 668.

[Jon93b] Reasoning about interference in an object-based design method, C.B.Jones, FME'93, LNCS 670.

[Jon93c] A π-calculus semantics for an object-based design notation, C.B.Jones, CONCUR'93, LNCS 715.

[Jon94] Process Algebra arguments about an object-based design notation, C.B.Jones, in A.W.Roscoe, editor, Festschrift for Tony Hoare, Prentice-Hall, 94.

[Jon95] Fixing the semantics of some concurrent object-oriented concepts, C.B.Jones, MFPS, Tulane University, New-Orleans, Mars 95.

[Jon95b] Granularity and the development of concurrent programs, Extended Abstract, Elsevier Science B.V, 1995.

[Jon96a] Accommodating interference in the formal design of concurrent object-based programs, C.B.Jones, Formal Methods in System Design, 8(2), pages 105-122, Mars 96.

[Jon96b] Some practical problems and their influence on semantics, C.B.Jones, ESOP'96, LNCS 1058.

[Lam91] The Temporal Logic of Actions, L.Lamport, Technical report 79, Digital, SRC, 1991.

[Mil89] A calculus of mobile processes, Part 1 and Part2, R.Milner, J.Parrow, D.Walker, LFCS, technical report ECS-LFCS-89-85 (86), June 1989.

[Mil89b] Communication and Concurrency, R.Milner, Prentice-Hall, 1989.

[Mil91] The polyadic π-calculus: a tutorial, R.Milner, LFCS, technical report ECS-LFCS-91-180, October 1991.

[PW96] On Transformations of Concurrent Object-Based Programs, A.Philippou, D.Walker, CONCUR'96, LNCS 1119, 1996.

[Rei83] Petri Nets, W.Reisig, EATCS Mongraphs on Theoretical Computer Science, Vol 4, Springer, Berlin, 1983.

[Sto90] Development of parallel programs based on shared data-structures, K.Stolen, PhD thesis, Manchester University, UMCS-91-1-1, 1990.