Le langage de programmation **SR**

IFT630

Gabriel Girard

Chapitre 1

Introduction

SR est un langage de programmation qui permet de faire des programmes séquentiels, parallèles et répartis.

Les trois éléments de base du langage sont :

— les ressources locales (resource)
— les ressources globales (globals)
— les operations (proc, op ou procedure)

1.1 Les ressources locales (par la suite appelées ressource)

Un programme SR contient une ou plusieurs ressources. Chaque ressource est un modèle à partir duquel on peut créer des instances (variables ou processus) dynamiquement.

On peut déterminer comment et où une ressource s'exécute (le processus associé).

Chaque programme possède une ressource principale qui est créée dynamiquement lorsque l'exécution du programme SR débute.

Une ressource correspond grossièrement à un module ou une classe dans d'autres langages tels Modula ou C++. Elle possède une partie spécification (**spec**) et une partie implantation (**body**). La partie spécification est automatiquement exportée. Cela signifie que d'autres ressources pourront utilisées les fonctions ou variables nommées dans la spécification. Une fonction **import** est utilisée pour obtenir l'accès aux objets qui sont exportés.

L'implantation d'une ressource (**body**) contient la déclaration des données, les énoncés d'initialisation (≡ au constructeur de C++) qui sont exécutés lors de la création de la ressource , les procédures, les processus et le code final (≡ au destructeur en C++) qui est exécuté lors de la destruction .

1.2 Les ressources globales (global)

Une ressource globale (`global`) est une collection d'objets partagés par des ressources. C'est une ressource sans paramètre pour laquelle il existe une seule et unique instance. Une seule instance d'une ressource globale est créée pour chaque "programme" qui s'exécute. Elle est créée de façon implicite la première fois que la ressource globale est importé.

Une ressource globale est définie de façon similaire à une ressource.

1.3 Les opérations

Une opération est une généralisation d'une procédure. Elle peut être appelée de façon synchrone (call - appel de procédure) ou asynchrone (send - envoie de messages).

Une opération peut être fournie de deux façons :

— par une procédure : `proc` ;
— par une entrée : un énoncé `in`.

Cela permet de supporter : RPC, Rendez-vous, Communication synchrone et asynchrone.

1.4 Utilisation

Un programme écrit dans le langage SR doit se terminer avec le suffixe "`sr`". Pour compiler un programme SR (ex. test.sr), on utilise la commande d'appel au compilateur **sr** (ex. `sr -o test test.sr`). Pour plus de détails sur les options vous pouvez faire `man sr`. Vous trouverez aussi en annexe des informations complémentaires sur les outils fournis avec le langage.

Chapitre 2

Éléments syntaxiques de base et exemples

SR est un langage procédural ressemblant à Pascal, Modula et C. Il contient certains concepts similaires :

— Un programme SR contient une ou plusieurs ressources. Ces ressources sont similaires aux modules de Modula-2 et contiennent du code, des procédures et des processus.

— Un programme comportant une seule ressource démarre automatiquement cette ressource peu importe son nom. Elle est considérée comme la ressource principale. Un programme comportant plusieurs ressources démarre toujours la ressource **main** en premier. Dans ce cas la ressource nommée `main` est considérée comme la ressource principale.

— On retrouve le concept de bloc dans SR (comme dans Pascal, modula, C, C++, ...).

Un bloc comprend des déclarations et des énoncés. Dans un bloc on retrouve des ressources, des ressources globales, des procédures ou des énoncés.

On peut utiliser les **begin-end** pour délimiter les blocs. Si on ne les utilise pas, les énoncés du langage servent de délimiteur de bloc.

— Les commentaires débutent par «#» et terminent à la fin de la ligne. Les commentaires peuvent aussi être mis entre /* et */.

— Un énoncé termine par un « ;» ou la fin de la ligne.

— Le langage SR utilise le concept d'affectation et d'expression. Un énoncé d'affectation peut prendre les formes suivantes :

var := *expr* (affectation)
var := *var* (affectation)
var :=: *var* (échange)

Une expression est une suite d'opérations appliquées sur des variables. Nous introduirons bientôt les variables, les types et les opérateurs que l'on peut utiliser.

— Les entrées/sorties :

read(var1, var2, ...)
write(var1, var2, ...)
printf(format, var1, var2, ...)
scanf(format,var1, var2, ...)

Les fonctions `printf` et `scanf` sont similaires à celles du langage C. Vous trouverez en annexe plus d'informations sur les entrées/sorties.

Le programme 1 montre un exemple simple de programme SR comprenant une seule ressource.

Programme 1 : Premier programme simple en SR

```
# programme qui ecrit allo
resource hello()
  write("Bonjour a tous")
end
```

2.1 Variables, constantes et types

Le langage SR comme la plupart des langages procéduraux possède le concept de variables et de constantes typées. On définit des variables ou des constantes de la façon suivante :

— `var` *def_var, def_var ...*

où

def_var : : *var_nom* : *type* := *expression*
: : *var_nom1, var_nom2, ...* : *type*

var_nom : : *var_id*
: : *var_id dimensions*

Le programme 2 montre des exemples de déclaration de variables en SR.

Programme 2 : Exemples de déclaration de variables

```
var i,j,k : int
var front:int:=0, fin :=0;
var ligne : string[80]=`` ''
var a[10]:int /* var a : [10] int */
var mat[10,10]: boolean /* var mat : [10,10] boolean */
```

— `const` *def_init, def_init, ...*

où

def_init : : const_nom : *type* := *expression*

const_nom : : const_id
: : const_id dimensions

Le programme 3 montre des exemples de déclaration de constantes en SR.

Programme 3 : Exemples de déclaration de constantes

```
const N:integer:=100, TO:int:= 2*1000
const X :=500
const msg : string[17] := "bonjour le monde"
const msg1 := "allo, comment ca va"
```

Les types de bases du langage sont :

— Les **booléens (boolean)** qui ne peuvent prendre que les valeurs vraie ou fausse . Les opérateurs sur les booléens sont `and` , `or`, `xor` et `not`. Les opérateurs de relation (utilisable sur tous les types de données) qui permettent de retourner des valeurs booléennes sont $<, >, =, !=, <=$ et $>=$.

— Les **entiers (int)** sur lesquelles sont définis les opérations $+, -$, $**$ (exposant), $*$, /, % (reste), `mod`, $<<, >>$. On peut aussi, comme dans le C, utiliser des compressions d'opérateurs. Ainsi on peut avoir les opérations $++$, $--$, + :=, - :=, * :=, / := et ** :=.

— Les **réels (real)** dont la valeur est de la forme

partie_entière.partie_fractionnaire exposant

Exemple : 3.45e10

Les opérations valides sont les mêmes que sur les entiers.

— Les **caractères (char)** qui ressemblent au char de C.

— Les **chaînes de caractères (string)** sur lesquelles on peut faire l'opération || (concaténation). Ainsi, l'énoncé `string[20]` défini une chaîne de 20 caractères.

— Les **pointeurs (ptr)** sont aussi supportés par SR. Les pointeurs sont utilisés en conjonction avec l'opérateur *addresse_de* (@) et l'opérateur (^) qui permet de retrouver la valeur pointée par le pointeur.

Le programme 4 montre des exemples de déclaration de pointeurs en SR.

Programme 4 : Exemples de déclaration de pointeurs

```
var p : ptr int
var pp : ptr ptr int
var ptrc : ptr char
```

Dans les exemples précédents, plusieurs variables (aucun vecteur) sont initialisées lors de leur déclaration. Il est possible en SR d'initialiser des vecteurs lors de leur déclaration. Pour l'initialisation de vecteurs on utilise le format suivant (dans cet exemple la partie déclaration est omise, on retrouve seulement la partie initialisation) :

(élément, élément, ...)

élément : : expression
: : [répétition] expression

Le programme 5 montre des exemples de déclaration de vecteurs en SR.

Programme 5 : Exemples de déclaration de vecteurs

```
var x[3] : int := (0, 0, 0)
var x[3] : int := ([3] 0)

var y[6] : int := (1, 2, 0, 0, 0, 6)
var y[6] : int := (1, 2, [3] 0, 6)
```

2.2 Les énoncés séquentiels de base du langage

SR possède aussi un ensemble d'énoncés similaire aux autres langages. On retrouve des énoncés de selection et d'itération (la séquence étant implicite).

1. L'énoncé de sélection est le `if`. Il combine à la fois le `if` et le `switch` de C++. Il a le format suivant :

if *commande_gardée* [] *commande_gardée* [] .... **fi**

commande_gardée : : expression → *bloc*
[] else → *bloc*

Un bloc peut contenir des énoncés, des expressions, des déclarations et des *importations*. Comme nous l'avons déjà mentionné il peut débuter par un `begin` et terminer par un `end`. Cependant, cela alourdit souvent le programme inutilement. Il n'est donc pas recommendé de les utiliser à moins que cela ne soit nécessaire.

Le programme 6 montre des exemples d'utilisation de l'énoncé «if» en SR.

Programme 6 : Exemples d'utilisation de l'énoncé «if»

```
   if x<0 -> x := -x fi
---------
   if x<y -> min := x
   [] y<x -> min := y
   [] x=y -> min = 0
   fi
---------
   if x<=y -> _________
   [] else -> _________
   fi
```

2. La principale commande d'itération est le `do`. Cet énoncé a le format suivant :

```
do commande_gardée
[] commande_gardée
    ⋮
od
```

Un énoncé `do` peut contenir un `else`. Si une garde est vraie, on boucle. Si toutes les gardes sont fausses et qu'il n'y a pas de `else`, on sort de la boucle. Si on inclus un `else` dans un `do`, la seule façon de sortir de la boucle est un `exit`.

Le programme 7 montre des exemples d'utilisation de l'énoncé «do» en SR.

Programme 7 : Exemples d'utilisation de l'énoncé «do»

```
do
   x<y -> y := y-x
[]
   y<x -> x := x-y
od
gcd := x
```

3. L'autre énoncé d'itération est un **for** (appelé `fa` (*Forall*)). Son format est :

> `fa` *quantité*, *quantité* → bloc `af`
>
> *quantité* : : `var` := *init* `to` *final* [by *step*] [st *expression*]
> *quantité* : : `var` := *init* `downto` *final* [by *step*] [st *expression*]

Le programme 8 montre des exemples d'utilisation de l'énoncé «fa» en SR.

Programme 8 : Exemples d'utilisation de l'énoncé «fa»

```
fa i:=1 to 10 -> write(i) af
---------
fa i:='a' to 'z' -> write(i) af
---------
fa i:=1 to 3, j:=i-1 downto 0 -> write(i,j) af
```

Deux variations du `fa` méritent quelques explications. Le `by` permet de changer le niveau d'incrément ou de décrément. Ainsi dans l'exemple qui suit on fait une boucle de 0 à 10 par incrément de 2 (0, 2, 4, ...).

Le programme 9 montre des exemples d'utilisation de l'énoncé «fa» en SR avec l'option «by».

Programme 9 : Exemples d'utilisation de l'énoncé «fa»

```
fa i:=0 to 10 by 2 -> write(i) af
```

Le `st` (such that) permet de valider une condition avant d'exécuter les énoncés associés au `fa`. Cela revient à mettre un `if` dans la boucle. Dans l'exemple qui suit on fait une boucle qui parcourt tous les éléments d'un vecteur et incrémente `cpt` seulement si la valeur contenu dans l'élément du vecteur est 0. On compte toutes les entrées du vecteur qui contiennent 0.

Le programme 10 montre des exemples d'utilisation de l'énoncé «fa» en SR avec l'option «st».

Programme 10 : Exemples d'utilisation de l'énoncé «fa»

```
cpt := 0
fa i:=0 to max st v[i] = 0 -> cpt++ af
```

4. **exit** : termine une boucle.
5. **next** : retour au contrôle de la boucle.
6. **skip** : ne fait rien.
7. **stop** : arrête l'exécution du programme.

2.3 Définition de types

Finalement, comme la plupart des langage, SR fournit la possibilité de créer de nouveaux types. Cela peut se faire par l'utilisation de ressources (nous le verrons plus tard) ou bien de la façon suivante :

```
type nom_type = def_type

def_type : : type_id
         : : dimension type_id

dimension : : [range, range, ....]

range : : expr :expr
      : : expr
```

Le programme 11 montre des exemples déclaration de type en SR.

Programme 11 : Exemples de déclaration de types

```
type age = integer
type point = [2] real
type note = [0:100] integer
type adresse = [3] string[40]
type pint = ptr int
type pchar = ptr char
type pr = ptr rec(i1,i2:int)
```

La création de nouveaux types utilise des énoncés pour regrouper ou identifier des données. On retrouve donc :

— Les types énumérés qui s'utilisent de la façon suivante :

```
enum (id1, id2, ...)
```

Le programme 11 montre des exemples déclaration de type énuméré en SR.

Programme 12 : Exemples de déclaration de type énuméré

```
type couleur = enum(rouge, bleu, jaune)
```

— Les enregistrements qui ont la forme suivante :

```
rec( def_champs1 ; def_champs2 ; ...)
```

Le programme 13 montre des exemples déclaration d'enregistrements en SR.

Programme 13 : Exemples de déclaration de types

```
type dimension = rec(taille, poids:real)
type personne = rec(nom:string[20]; note:int)
type info = rec(p:personne; n:dimension; emploi:boolean)
```

— Les unions ont le même format que les enregistrements. On remplace seulement le mot réservé `rec` par `union`.

Le langage SR est un langage dit **fortement typé**.

L'équivalence de type en SR est donc strictement structurelle. Un objet peut être affecté à un autre s'il est structurellement équivalent. Deux objets sont structurellement équivalents s'ils ont le même nombre de dimensions, le même nombre d'éléments dans chaque dimension, et si chaque paire d'éléments correspondant sont de même type. Pour un enregistrement, ils doivent avoir le même nombre de champs et les champs doivent être de même type.

Il existe des opérateurs de conversion.

2.4 Exemples de programme SR

Voici quelques exemples de programmes SR. Ils contiennent des éléments syntaxiques que nous n'avons pas encore abordés, telles la déclaration de procédure et de processus ainsi que la lecture des arguments d'un programme, mais il donne un bon aperçu de l'aspect général d'un programme SR.

Le programme 14 calcule le factoriel d'un nombre. Pour se faire, il définit une procédure (fonction) qui reçoit un entier en paramètre et qui retourne un entier.

Programme 14 : Programme qui calcule le factoriel

```
# programme qui calcule le factoriel
resource factorial()
  procedure fact(v: int) returns r: int
    # v is assumed to be positive
    if v = 1 -> r := 1
    [] v > 1 -> r := v*fact(v-1)
    fi
  end
  var n: int
  writes("enter a positive integer: "); read(n)
  write("the factorial of", n, "is", fact(n))
end
```

Le programme 15 multiplie deux matrices. Pour ce faire, il démarre autant de processus

que le nombre de produits scalaires à effectuer. On remarque dans cet exemple l'énoncé `process` qui permet de déclarer des processus et l'énoncé `getarg` qui permet de lire les arguments passés sur la ligne de commande du programme.

Programme 15 : Programme qui multiplie des matrices en parallele

```
# programme qui multiplie des matrices en parallele
resource matrix()
  # read command line arguments and open files
  var n: int; getarg(1, n)
  var namea: string[20]; getarg(2, namea)
  var nameb: string[20]; getarg(3, nameb)
  var filea := open(namea, READ)
  var fileb := open(nameb, READ)
  # declare and initialize matrices a, b, and c
  var a[1:n,1:n], b[1:n,1:n]: real
  var c[1:n,1:n]: real := ([n] ([n] 0.0) )
  fa i := 1 to n, j := 1 to n ->
      read(filea, a[i,j]); read(fileb, b[i,j])
  af

  # compute n**2 inner products in parallel
  process compute(i := 1 to n, j := 1 to n)
    fa k := 1 to n ->
        c[i,j] := c[i,j] + a[i,k]*b[k,j]
    af
  end

  final    # print c, one row per line
    fa i := 1 to n ->
      fa j := 1 to n -> writes(c[i,j], " ") af
      write()    # force new line
    af
  end
end matrix
```

Notons les éléments suivants au sujet de ces exemples.

— Contrairement à la plupart des langages, SR supporte la programmation parallèle, Ainsi, le mot réservé **process** démarre des processus lorsqu'une instance de la ressource est créée.

— Une ressource possède du code initial qui est exécuté lors de la création d'une instance de la ressource. Ce code est éparpillé dans le corps de la ressource.

— Une ressource possède un code final qui est exécuté lorsque la ressource est détruite.

2.5 Exemple de programmation parallèle

Le langage SR supporte la programmation parallèle. Il permet de créer des processus et de les faire communiquer soit par des variables globales ou par des messages. Le langage SR permet de créer des ressources globales grâce à l'énoncé **global** qui devront être utilisées dans une section critique si accédés par plusieurs processus. Le programme 16 comporte une ressource globale et utilise la communication par message.

Programme 16 : Programme parallèle avec communication par messages

```
global CS
  op CSenter(id: int) {call}, # doit être appelé
     CSexit()                 # peut être appelé (call ou send)
body CS
  process arbitrator
    do true ->
      in CSenter(id) by id ->
          write("user", id, "in its CS at", age())
      ni
      receive CSexit()
  od
  end
end

resource main()
  import CS
  var numusers, rounds: int
  getarg(1, numusers); getarg(2, rounds)

  process user(i := 1 to numusers)
    fa j := 1 to rounds ->
      call CSenter(i)
      nap(int(random()*100))
      send CSexit()
      nap(int(random()*1000))
    af
  end
end
```

Dans cet exemple, on utilise deux primitives de communication :

— **in** et **call** qui permettent d'effectuer un rendez-vous.

— **send - receive** qui permettent de faire de la communication asynchrone.

La fonction **age()** donne la nombre de millisecondes d'exécution accumulées par le programme. La fonction **nap(s)** permet au programme de *dormir* pendant `s` millisecondes. La fonction **random(x)** retourne un nombre réel compris entre 0 et x.

Chapitre 3

Définition de procédures

Définition de procédure

```
procedure nom_proc (paramètres_formels)
        bloc
end
```

Définition d'une fonction :

```
procedure nom_proc (paramètres_formels) returns nom_var : type
        bloc
end
```

Les paramètres formels sont définis de façon similaire aux variables. Les définitions de paramètres sont séparés par des « ;».

Les programmes 17, 18 et 19 montrent comment utiliser les procédures.

Programme 17 : Programme déclarant une procédure

```
procedure X(a,b : int; c:char)
```

Programme 18 : Programme déclarant une fonction

```
# Fonction qui calcule et retourne le factoriel.
# Par défaut, les deux paramètres formels sont des entiers
proc fact(n) returns prod
    prod := 1
    fa i := 1 to n -> prod *:= i af
end
```

Programme 19 : Programme utilisant les procédures

```
# Ce programme contient une procedure qui calcule et imprime
# le factoriel d'un nombre
resource main()
  procedure factorial(x: int)
      var fact := 1
      fa i := 1 to x -> fact *:= i af
      write(x, "factorial is", fact)
  end
  # Trouve le factoriel de tous les nombres entrés
  var x: int
  do true ->
      writes("Entrez un nombre > 0 ")
      writes("(ou = 0 pour terminer): ")
      read(x)
      if x = 0 -> exit
      [] x < 0 -> write("Le nombre doit être >= 0")
      [] x > 0 -> call factorial(x)
      fi
  od
end
```

Par défaut, les paramètres effectifs sont toujours passés par valeur. Ainsi dans les exemples précédents, les paramètres effectifs sont passés par valeur.

Cependant, SR fournit plusieurs façons de passer des paramètres :

— par valeur (défaut) : `val`

— par résultat : `res`. Ce sont des paramètres de sortie seulement.

— par variable : `var`. Ce sont des paramètres d'entrée et de sortie.

— par référence : `ref`. On reçoit un pointeur vers le paramètre effectif.

Un appel de procédure se fait grâce à l'énoncé `call`. L'énoncé `call` peut cependant être omis.

`call` *nom_proc* (*paramètres_effectifs*)

ou

nom_proc (*paramètres_effectifs*)

Une fonction peut arrêter son exécution et retourner une valeur grâce à l'énoncé `return` (à ne pas confondre avec l'énoncé `returns` qui permet de spécifier qu'une procédure retourne une valeur lors de sa déclaration).

Le programme 20 montre comment utiliser les procédures.

Programme 20 : Programme qui utilise des procédures

```
# retourne vrai si x est dans la liste
# type :  node is rec(value: int; ...; nxt: ptr node)
procedure search(x: int) returns found: bool
    var p: ptr node
    p := head
    do p != null ->
        if p^.value = x -> found := true; return fi
        p := p^.nxt
    od
    found := false
end
```

3.1 Les énoncés `proc` et `op`

Une procédure respecte les concepts de SR. Elle doit posséder une spécification et une implantation. La spécification d'une procédure se fait grâce à l'énoncé `op`. Elle sert à exporter ou à déclarer des procécures. L'implantation d'une procédure se fait grâce à l'énoncé `proc`.

Ils s'utilisent de la façon suivante :

> `op` *def_op, def_op, ...*
> *def_op* : : *id* (*paramètres*) [`returns` *var : type*]
>
> `proc` *id* (*paramètres*) [`returns` *var : type*]

Le programme 21 montre comment utiliser les «proc» et les «op».

Programme 21 : Programme qui implante un tri

```
resource main()
  op sort(var a[1:*]: int)

  var x[1:20], y[2:30], z['a':'z']: int
  call sort(x)
  call sort(y)
  call sort(z)

  proc sort(a)
      fa i := lb(a) to ub(a)-1,
         j := i+1 to ub(a) st a[i] > a[j] ->
             a[i] :=: a[j]
      af
  end
end
```

Dans cet exemple, on utilise l'énoncé `op` afin de déclarer et utiliser la fonction avant de l'implanter.

3.2 Différence entre `proc` et `procedure`

L'énoncé `procedure` est une abbréviation des énoncés `op` et `proc`. C'est suffisant pour les cas simples, mais il ne possède pas la souplesse de `proc` et `op`. Avec l'énoncé `procédure`, le tri doit être implanté comme le montre le programme 22.

Programme 22 : Programme qui implante un tri

```
resource main()
  procedure sort(var a[1:*]: int)
      fa i := lb(a) to ub(a)-1,
         j := i+1 to ub(a) st a[i] > a[j] ->
             a[i] :=: a[j]
      af
  end

  var x[1:20], y[2:30], z['a':'z']: int
  call sort(x)
  call sort(y)
  call sort(z)
end
```

3.3 Les pouvoirs (capabilities)

Un pouvoir est un pointeur vers une fonction ou une opération. Ils peuvent être affectés à des variables, passés en paramètres, ...

On définit un pouvoir de la façon suivante :

`cap` *spec_op* ou *id_op*

Soit une opération `d` définie de la façon suivante : `op d(x:int)`. Le programme 23 montre comment déclarer un pouvoir vers cette opération.

Programme 23 : Exemples utilisant les pouvoirs

```
var X : cap d

X := d      # X pointe vers l'operation d

X(...)      #  appel a la procedure d
```

Chapitre 4

Ressource et ressource globale

Les ressources créent des modèles à partir desquels on peut instancier des objets. Les ressources globales permettent de définir un object pour lequel une seule instance sera créée et partagée par tous.

Un **ressource** se définit de la façon suivante :

```
resource nom_ressource
    imports
    déclarations de constantes, variables, types à exporter
body nom_ressource(paramètres)
    imports
    déclarations, énoncés (code initial), procédures
    code final
end [nom_ressource]
```

Le programme 24 montre comment utiliser les ressources pour implanter une pile.

Quelques remarques :

1. Pour créer une instance d'une ressource, on utilise l'énoncé `create`. Cet énoncé retourne un pouvoir vers la ressource. Ainsi, pour définir une instance d'une ressource, on doit définir une variable de type pouvoir (de pile) et lui affecter la valeur retournée par `create`.
2. Pour utiliser les types exportés par une ressource, on ajoute toujours son nom en préfixe. Ainsi `stack.result` réfère au type `result` définit dans la ressource `stack`.
3. Pour utiliser les variables ou procédures d'une ressource, on préfixe le nom de la variable ou de la procédure par le nom du pouvoir pointant vers la ressource. Dans notre

Programme 24 : Programmes qui implante une pile

```
#  Ce code définit la ressource pile
#
resource Stack
  type result = enum(OK, OVERFLOW, UNDERFLOW)
  op push(item: int) returns r: result
  op pop(res item: int) returns r: result
body Stack(size: int)
  var store[1:size]: int, top: int := 0
  proc push(item) returns r
    if top < size -> store[++top] := item; r := OK
    [] top = size -> r := OVERFLOW
    fi
  end
  proc pop(item) returns r
    if top > 0 -> item := store[top--]; r := OK
    [] top = 0 -> r := UNDERFLOW
    fi
  end
end Stack
#
# Ce programme utilise la ressource pile
#
resource Stack_User()
  import Stack
  var x: Stack.result
  var s1, s2: cap Stack
  var y: int
  s1 := create Stack(10); s2 := create Stack(20)
   ...
  s1.push(4); s1.push(37); s2.push(98)
  if x := s1.pop(y) != OK -> ... fi
  if x := s2.pop(y) != OK -> ... fi
   ...
end
```

exemple, `s1` et `s2` sont deux pouvoirs vers deux piles dictintes. Pour empiler sur `s1` on doit faire `s1.push(1)`.

4. Il est important de noter que le mot réservé `resource` ne prend qu'un seul `s`.

Si on définit une ressource sans rien exporter, on peut abréger la définition d'une ressource de la façon suivante :

```
resource foo( ... )
    code
end foo
```

4.1 Compilation séparée

La langage SR permet de compiler séparément la spécification et l'implantation d'une ressource. Le programme 25 montre comment déclarer les différentes parties d'un programme SR en vue de la compilation séparée.

Programme 25 : Exemple de compilation séparée

```
# Dans un fichier
resource X
  ...
body X(...) separate

# Dans un second fichier
body X(...)
   ...
end X
```

4.2 Énoncés concernant les ressources

Import

L'énoncé `import` permet d'importer les spécifications d'autres ressources. On l'utilise dans la spécification ou l'implantation (body) d'une ressource. Il s'utilise de la façon suivante :

```
import nom_ress1, nom_ress2, ...
```

Create et Destroy

On peut créer et détruire dynamiquement des instances de ressources. Les pouvoirs permettent de faire la distinction entre les différentes instances d'une même ressources ou de plusieurs ressources.

La création se fait de la manière suivante :

```
var nom_pouv : cap nom_ressource
x := create nom_ressource
```

On peut ensuite utiliser les variables ou les opérations grâce à ce pouvoir. On a vu un exemple d'utilisation dans l'exemple de pile.

La destruction se fait grâce à la commande `destroy` :

```
destroy nom_pouvoir
```

4.3 Code initial et final

Le code initial d'une ressource est exécutée lors de la création d'une instance. Ce code est inséré au même niveau que les procédures. Tout code qui ne se trouve pas dans une procédure est considéré comme du code initial.

Le code final apparaît dans un bloc spécifique :

```
final
    code
end
```

Une ressources peut donc avoir le format suivant :

```
resource X( ...)
    déclarations
    code initial
    procedure ... end
    déclarations        # pour la ressource
    code initial        # pour la ressource
    procedure ... end
    final ... end
end
```

4.4 Les ressources globales

La définition et l'implantation d'une ressource globale sont très similaires à ceux d'une ressource. On change seulement le terme `resource` par `global`.

Une ressource globale peut donc être déclarée de trois façons distinctes :

1. Sans aucune spécification :

```
global nom_global
    importation
    déclarations
end
```

2. Avec spécification et implantation :

```
global nom_global
    importation
    déclarations
body nom_global
    importation
    déclarations
    énoncé
    procedure
    code final
end
```

3. En utilisant la compilation séparée :

```
global nom_global
    imports
    déclarations
body nom_global separate

body nom_global
    importations
    déclarations
    énoncé
    procedure
    code final
end
```

Chapitre 5

Concurrence

La particularité du langage SR est qu'il supporte la concurrence. Il fournit donc des énoncés pour déclarer des processus, démarrer des processus, faire communiquer des processus et synchroniser des processus.

5.1 Déclaration de processus

On peut déclarer des processus à l'intérieur d'une ressource. Ceux-ci sont démarrés aussitôt qu'une instance d'une ressource les contenant est créée. On déclare des processus de la façon suivante :

```
resource X
        ...
   body
        ...
        process nom(quantité, quantité,...)
          bloc
        end
   end
```

Les quantités sont spécifiées exactement selon le même format que pour l'énoncé `fa`. Ils servent à créer le nombre de processus spécifié par la «quantité».

Les programmes 26 et 27 montre deux exemples d'utilisation de la concurrence en SR. Le premier exemple déclare deux processus. Le second exemple effectue une multiplication de matrices en parallèle. On y crée (`i × j`) processus, i.e. un pour calculer chaque élément

de la matrice résultante.

Programme 26 : Programme avec deux processus

```
resource foo()
  var x := 0
  process p1
      x +:= 3
  end
  var a[10]: int
  fa i := 1 to 10 -> a[i] := i af
  process p2
      x +:= 4
  end
end
```

Programme 27 : Programme qui implante une multiplication de matrices

```
resource mult()
  const N := 20
  var a[N,N], b[N,N], c[N,N]: real
  # read in some initial values for a and b
   ...
  # multiply a and b in parallel
  # place result in matrix c
  process multiply(i := 1 to N, j := 1 to N)
      var inner_prod := 0.0
      fa k := 1 to N ->
          inner_prod +:= a[i,k]*b[k,j]
      af
      c[i,j] := inner_prod
  end
  final
      # output values from c
       ...
  end
end
```

5.2 Énoncés concurrents

SR permet aussi d'exécuter une série d'énoncés en parallèle. Cela se fait grâce à l'énoncé `co`.

`co` *énoncé* || *énoncé* || ... `oc`

Le programme 28 montre un exemple d'utilisation du «co». L'énoncé `co` peut aussi s'utiliser d'une façon similaire à un `fa` comme le montre les programmes 29, 30 et 31.

Programme 28 : Programme qui implante un tri rapide en parallèle

```
# Ce programme implante un tri rapide (quicksort) parallèle
#
resource quick()
  op sort(var a[1:*]: int)
  var n: int
  # read in n
  getarg(1,n)
  var a[1:n]: int
  # read in data to be sorted
  fa i := 1 to n -> read(a[i]) af
  write("input:"); fa i := 1 to n -> write(a[i]) af
  sort(a)
  write("sorted:"); fa i := 1 to n -> write(a[i]) af

  proc sort(a)
    if ub(a) <= 1 -> write("small array", ub(a)); return fi
    fa i := 1 to ub(a) -> writes(a[i], " ") af; write()
    var pivot := a[1]
    var lx := 2, rx := ub(a)
    do lx <= rx ->
        if a[lx] <= pivot -> lx++
        [] a[lx] >  pivot -> a[lx] :=: a[rx]; rx--
        fi
    od
    a[rx] :=: a[1]
    co sort(a[1:rx-1]) // sort(a[lx:ub(a)]) oc
  end
end quick
```

Programme 29 : Programme avec «co»

```
# Ce programme demarre autant de processus qu'il y a d'elements
# differents de 0 dan le vecteur "a".
# Il appelle une procedure dont la valeur retournee determine si
# on execute le code qui suit.
cpt := 0
co (i := 1 to n st a[i] != 0) p(i) -> cpt++; write(cpt,i)
oc
```

Programme 30 : Programme avec «co»

```
resource partial_sums()
  op save(i: int), update(i: int)
  var n := 20 /* default */, d := 1
  const N := 200
  var sum[N], old[N]: int
  getarg(1,n)
  if n > N -> write(stderr,"n exceeds", N); stop fi
  fa i := 1 to n -> sum[i] := i af
  do d < n ->
      co (i := 1 to n) save(i) oc
      co (i := 1 to n) update(i) oc
      d := 2*d
  od
  fa i := 1 to n -> write(i, sum[i]) af
  proc save(i)
      old[i] := sum[i]
  end
  proc update(i)
      if i-d >= 1 -> sum[i] +:= old[i-d] fi
  end
end
```

Programme 31 : Programme avec «co»

```
# On veut lire sur un fichier a copie multiples
# on lance la lecture sur toutes les copies
# on conserve seulement la premiere reponse

co (i:= 1 to 4)
    fichier.read(arguments) -> lequel := i; exit
oc
```

5.3 Synchronisation

5.3.1 Les sémaphores

Avec SR, on peut déclarer des variables de type sémaphore et les utiliser pour la synchronisation lors d'accès à des variables partagées.

Définition :

> `sem` *définition, définition, ...*
> *définition : : nom dimension := expression*

Le programme 32 montre comment déclarer des sémaphores.

Programme 32 : Exemples de déclaration de sémaphores

```
sem mutex := 1

sem mutex[N] := (1, [N-1] 0)
```

Les opérations sur les variables de type sémaphore sont l'opération `p` et l'opération `v`.

> `p(`*nom*[*indice*]`)`
> `v(`*nom*[*indice*]`)`

Les programmes 33, 34, 35 et 36 montrent comment utiliser des sémaphores.

Programme 33 : Programme utilisant les sémaphores

```
resource CS()
  const N := 20    # number of processes
  var x := 0       # shared variable
  sem mutex := 1   # mutual exclusion for x
  process p(i := 1 to N)
    # non-critical section
      ...
    # critical section
    P(mutex)   # enter critical section
    x := x+1
    V(mutex)   # exit critical section
    # non-critical section
      ...
  end
end
```

Programme 34 : Programme utilisant les sémaphores

```
resource CS_Ordered()
  const N := 20  # nombre de processus
  sem mutex[N] := (1, [N-1] 0)
  process p(i := 1 to N)
    #  section non critique
      ...
    # section critique
    P(mutex[i])             # début section critique
     ...                    # section critique
    V(mutex[(i mod N)+1])   # finsection critique
    # section non critique
      ...
  end
end
```

Programme 35 : Programme utilisant les sémaphores

```
resource barriere()
  const N := 20  # nombre de processus
  sem done := 0, continue[N] := ([N] 0)
  # déclaration des variables partagées par les travailleurs
  process travailleur(i := 1 to N)
    do true ->
      # code pour la tâche i
      V(done)
      P(continue[i])
    od
  end
  process coordonnateur
    do true ->
      fa w := 1 to N -> P(done) af
      fa w := 1 to N -> V(continue[w]) af
    od
  end
end
```

Programme 36 : Programme utilisant les sémaphores

```
global barriere
  const N := 20  # nombre de processus
  sem continue[N] := ([N] 0)
  sem done := 0
body barriere
  process c
    do true ->
      fa w := 1 to N -> P(done) af
      fa w := 1 to N -> V(continue[w]) af
    od
  end
end

resource travailleurs()
  import barrier
  # déclaration des variables partagées par les travailleurs
  process travailleur(i := 1 to N)
    do true ->
      # code pour la tâche i
      V(done)
      P(continue[i])
    od
  end
end
```

5.3.2 Communication par messages asynchrones

Le langage SR fournit aussi des énoncés permettant de faire communiquer des processus par l'envoi et la réception de messages.

Pour expédier un message, on fait :

```
send id_op(...)
```

Pour recevoir un message, on fait :

```
receive id_op indice(var, var, ...)
```

Il est à noter que l'identificateur **id_op** est défini grâce à l'énoncé `op`. Cependant, dans ce cas, plutôt que de définir une procédure, il sert à définir un port de communication.

Les programmes 37, 38, 39, 40 et 37 montrent comment utiliser les messages. Dans le programme 41, on remarque que l'on défini une procédure comme "port". Lorsque l'on expédie un message avec l'énoncé `send`, un nouveau processus est automatiquement créé pour répondre à la demande. Dans cet exemple, un processus serveur est créé pour chaque client.

Programme 37 : Programme utilisant la communication par messages

```
resource stream_merge()
  const EOS := high(int)  # end of stream marker
  op stream1(x: int), stream2(x: int)
  process one
     ...
    send stream1(y)
     ...
    send stream1(EOS)
  end
  process two
     ...
    send stream2(y)
     ...
    send stream2(EOS)
  end
  process merge
    var v1, v2: int
    receive stream1(v1); receive stream2(v2)
    do v1 < EOS or v2 < EOS ->
      if v1 <= v2 -> write(v1); receive stream1(v1)
      [] v2 <= v1 -> write(v2); receive stream2(v2)
      fi
    od
    write(EOS)
  end
end
```

Programme 38 : Programme utilisant la communication par messages

```
resource cs1()
  op request(...), results(...)
  process client
     ...
    send request(...)
    # possibly perform some other work
    receive results(...)
     ...
  end
  process server
    do true ->
      receive request(...)
      # handle request
      send results(...)
    od
  end
end
```

Programme 39 : Programme utilisant la communication par messages

```
resource cs2()
  const N := 20  # number of client processes
  op request(id: int; ...), results[N](...)
  process client(i := 1 to N)
     ...
    send request(i, ...)
    # possibly perform some other work
    receive results[i](...)
     ...
  end
  process server
    do true ->
      receive request(id, ...)
      # handle request
      send results[id](...)
    od
  end
end
```

Programme 40 : Programme utilisant la communication par messages

```
resource cs3()
  optype results_type = (...)
  op request(results_cap: cap results_type; ...)
  const N := 20  # number of client processes
  process client(i := 1 to N)
    op results: results_type
     ...
    send request(results,...)
    # possibly perform some other work
    receive results(...)
     ...
  end
  process server
    do true ->
      var results_cap: cap results_type
      receive request(results_cap, ...)
      # handle request
      send results_cap(...)
    od
  end
end
```

Programme 41 : Programme utilisant la communication par messages

```
resource cs4()
  optype results_type = (...)
  op request(results_cap: cap results_type; ...)
  const N := 20  # number of client processes
  process client(i := 1 to N)
    op results: results_type
     ...
    send request(results,...)
    # possibly perform some other work
    receive results(...)
     ...
  end
  proc request(results_cap, ...)
    # handle request
    send results_cap(...)
  end
end
```

5.3.3 Communication par Rendez-vous et RPC

Le mécanisme de rendez-vous se fait grâce à l'utilisation de l'énoncé `in` combiné avec un appel de procédure. L'énoncé `in` s'utilise comme suit :

```
in nom_cmd
[] nom_cmd
[] ...
ni

nom_cmd : : opération(paramètres) st sync by sched → bloc
        : : opération(paramètres) returns résultat st sync
                                  by sched → bloc
        : : else → bloc
```

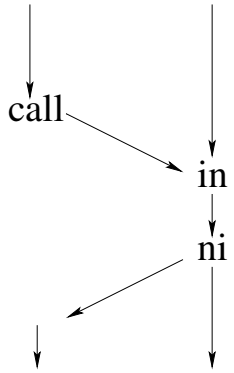
La partie *opération(paramètres)* est identique à une définition de procédure. L'option `st` permet de définir une condition d'exécution (sync). L'option `by` permet de définir des priorités lors de l'exécution. Une telle définition est équivalente à inclure plusieurs `receive` ou à définir plusieurs procédures différentes.

Un processus exécutant un "`in`" est bloqué jusqu'à ce qu'une des opérations soit appelée. La présence d'une `else` permet à un processus de ne pas se bloquer.

Pour communiquer avec une opération définie par un énoncé `in`, on peut utiliser soit un énoncé `call` ou un énoncé `send`.

Call - in

L'utilisation d'un call avec un `in` permet d'implanter des RPC ou des rendez-vous selon leur terminologie.



Les programmes 42, 43, 44 et 45 montrent comment utiliser «Call-in».

Programme 42 : Programme utilisant les rendez-vous

```
resource main()
  op f(x: int), g(u: real) returns v: real
  process p1
     ...
    call f(y)
     ...
  end
  process p2
     ...
    w := g(3.8)
     ...
  end
  process q
     ...
    in f(x) -> z +:= x
    [] g(u) returns v -> v := u*u-9.3
    ni
     ...
  end
end
```

Send - in

Un `send` utilisé avec une opération définie par un `in` est équivalent à un `send` utilisé avec un `receive`. En fait, un receive est une abbréviation simplifiée d'un `in`.

Programme 43 : Programme utilisant les rendez-vous

```
resource main()
  op f(x: int)
  process p
     ...
    call f(y)
     ...
  end
  process q
     ...
    in f(x) -> z +:= x ni
     ...
  end
end
```

Programme 44 : Programme utilisant les rendez-vous

```
resource bounded_buffer
  op deposit(item: int)
  op fetch() returns item: int
body bounded_buffer(size: int)
  var buf[0:size-1]: int
  var count := 0, front := 0, rear := 0
  process worker
    do true ->
      in deposit(item) st count < size ->
          buf[rear] := item
          rear := (rear+1) % size
          count++
      [] fetch() returns item st count > 0 ->
          item := buf[front]
          front := (front+1) % size
          count--
      ni
    od
  end
end
```

Programme 45 : Programme utilisant les rendez-vous

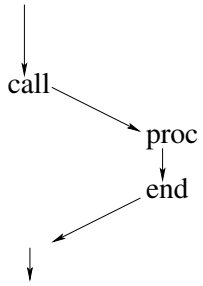
```
resource main()
  op swap(var x: int)
  process p1
     ...
    call swap(y)
     ...
  end
  process p2
     ...
    call swap(z)
     ...
  end
  process q
     ...
    in swap(x1) -> in swap(x2) -> x1 :=: x2 ni ni
     ...
  end
end
```

5.3.4 Résumé des communications

En combinant les énoncés `in`, `proc`, `send` et `call`, le langage SR offre plusieurs types de communication différente.

Call - proc

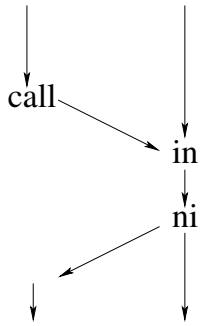
Cette combinaison est un appel de procédure normal. Cependant, il est possible que la procédure s'exécute sur un autre site. Cela devient alors un appel de procédure éloigné (RPC). La séquence d'exécution est la suivante :



Call - in

L'énoncé `call` combiné avec un énoncé `in` effectue aussi un appel de procédure éloigné (RPC). Cependant, le schéma d'exécution est légèrement différent du précédent. Dans la

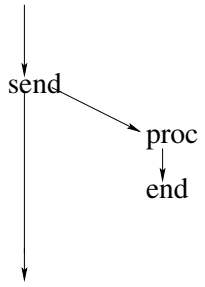
terminologie de SR, cela s'apelle un **rendez-vous**. Cela est aussi similaire aux **rendez-vous** de ADA. Voici un schéma d'exécution pour ces deux énoncés.



Send - proc

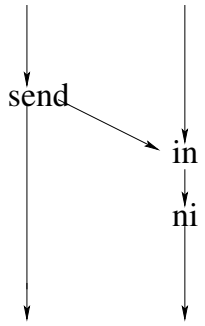
Cette combinaison permet de démarrer un nouveau processus. En effet, faire un `send` vers une procédure, démarre un nouveau processus exécutant cette procédure.

Voici un schéma d'exécution dans cette situation :



Send - in

Cette conbinaison est équivalente à la combinaison `send - receive`.



5.4 Autres techniques de synchronisation

SR ne fournit directement aucune autre technique de synchronisation tels les moniteurs, les régions critiques conditionnelles ou les expressions de chemins. Cependant, avec le compilateur SR, il vient trois pré-processeurs qui permettent de simuler les moniteurs, les régions critiques conditionnelles et le langage CSP (les rendez-vous tel que nous l'avons défini).

5.4.1 Les moniteurs

Utilisation

Le pré-processeur qui simule les moniteurs est `m2sr`. Il traduit le programme utilisant les moniteurs en un programme SR qui doit ensuite être compilé. L'extension pour un fichier contenant un programme utilisant les moniteurs est "`.m`". Un fichier `essai.m` compilé avec `m2sr` génère un fichier contenant un programme SR appelé `essai.sr`.

Syntaxe

Les moniteurs de SR possèdent une syntaxe similaire aux ressources. Il possède une partie spécification et une implantation. La syntaxe d'un moniteur est la suivante :

```
_monitor nom_moniteur
        imports
        constantes, variables, types et opérations à exporter
_body nom_moniteur(paramètres)
        imports
        déclarations, énoncés (code initial), procédures
        code final
_monitor_end [nom_moniteur]
```

Les déclarations de constantes, de variables, de types et d'opérations se font avec les énoncés de SR. Les déclarations de procédures possèdent une syntaxe spéciale :

```
_proc (nom_proc(paramètres) returns nom_var : type)
        bloc
_proc_end
```

Un nouveau type de données est supporté avec les moniteurs. Ce sont les **variables conditions**. le programme 46 montre comment définir des variables de type condition.
Les opérations sur les variables de types conditions sont les suivantes :

— **_wait(var_cond)**

Programme 46 : Programme qui multiplie des matrices en parallele

```
_condvar(X)          /* une variable de type condition */
_condvar1(X,2)       /* un vecteur contenant deux variables */
                     /* de types conditions : X[1] et X[2]  */

_condvar2(X, 2, 2)   /* une matrice de 2X2 contenant des         */
                     /* variables de types conditions x[1,1] ... */
```

- **__signal(var_cond)** : cette fonction peut avoir plusieurs comportements selon l'option utilisée à la compilation. Les options sont :
  - **SC** (Signal and Continue) : c'est l'option par défaut. Le processus qui émet le signal continue son exécution. Le processus débloqué continuera son exécution plus tard.
  - **SW** (Signal and Wait) : le processus émettant le signal est bloqué au profit du processus qu'il vient de libérer. L'émetteur du signal devra ré-acquérir le moniteur afin de poursuivre son exécution.
  - **SX** (Signal and Exit) : l'émetteur du signal est automatiquement sorti du moniteur (retour au programme appelant).
  - **SU** (Signal and Urgent wait) : c'est une variante du SW. L'émetteur du signal est assuré de ré-acquérir le moniteur avant tout nouvel appel.
- **__print(var_cond)** : cette fonction imprime le nombre de processus en attente sur la variable condition.
- **__empty(var_cond)** : cette fonction retourne faux s'il n'y a aucun processus en attente sur la variable condition.
- **__signal_all(var_cond)** : cette fonction s'utilise seulement avec l'option SC. Elle libère tous les processus en attente sur la variable condition.
- **__minrank(var_cond)** : cette fonction retourne la valeur entière de la priorité associée au processus en tête de la file d'attente.
- **__pri_wait(var_cond, prio)** : cette fontion permet d'attendre sur une variable condition selon une certaine priorité basée sur la valeur de **prio**. Le premier processus libéré lors d'un signal sera celui avec la plus petite valeur de **prio**.

Lorsqu'on utilise les moniteurs, on ne doit pas :

- utiliser les commentaires commençant par #.
- utiliser les autres outils de synchronisation.
- utiliser des noms commençant par "m_".
- utiliser de «return» dans une `_proc`.

Voici des exemples de programmes utilisant les moniteurs :

Programme 47 : Exemple d'utilisation des moniteurs

```
 /* utiliser seulement cette forme de commentaires dans les
moniteurs */
 _monitor(bounded_buffer)
     op deposit(data:int), fetch() returns data:int
 _body(bounded_buffer)
     const N := 4
     var buf[0:N-1]: int
     var front := 0, rear := 0, count := 0
     _condvar(not_full); _condvar(not_empty)
     _proc( deposit(data) )
         if count = N -> _print(not_full); _print(not_empty)
                         _wait(not_full)  fi
         _print(not_full); _print(not_empty)
         buf[rear] := data
         rear := (rear+1) % N
         count := count+1
         _signal(not_empty)
     _proc_end
     _proc( fetch() returns result )
         if count = 0 -> _print(not_full); _print(not_empty)
                         _wait(not_empty)  fi
         _print(not_full); _print(not_empty)
         result := buf[front]
         front := (front+1) % N
         count := count-1
         _signal(not_full)
     _proc_end
 _monitor_end
```

Programme 48 : Exemple d'utilisation des moniteurs

```
 #programme principal utilisant le moniteur
 resource main()
     import bounded_buffer
     process prod(i:=1 to 6)
         bounded_buffer.deposit(i)
     end
     process cons(i:=1 to 6)
         write( bounded_buffer.fetch() )  # consommation
     end
 end
```

Programme 49 : Exemple d'utilisation des moniteurs

```
    _monitor(foo)
        op enter(who:int), xexit()
    _body(foo)
        var free := true
        _condvar(not_busy)
        _proc(enter(who))
            if not free -> _pri_wait(not_busy,who) fi
            free := false
        _proc_end
        _proc(xexit())
            if _empty(not_busy) -> write("not_busy empty")
            [] else -> write("not_busy minrank is", _minrank(
   not_busy))
           fi
            _print(not_busy)
            free := true
            _signal(not_busy)
       _proc_end
    _monitor_end
```

Programme 50 : Exemple d'utilisation des moniteurs

```
# note: very nondeterministic due to the naps, and output is nondet
   too.
# gives different output when run with sc than with sw or su
# demonstrates effect of different signal discipline
# (of course, could be just scheduler, though; but it is not.)
# (output from sw and su is the same)
resource main()
    import foo
    process p(i:=1 to 10)
        write( i, "before" )
  var k: int
  nap(10-i)
  if i > 5 -> k := 10-i [] else -> k := i+3; k := 99 /*testing*/  fi
        foo.enter(k)
        write( i, "in" )
        nap(2)
        write( i, "leaves" )
        foo.xexit()
        write( i, "after" )
    end
end
```

Programme 51 : Exemple d'utilisation des moniteurs

```
_monitor(foo)
    op enter(), xexit()
_body(foo)
    var free := true
    _condvar(not_busy)
    _proc(enter())
        if not free -> _wait(not_busy) fi
        free := false
    _proc_end
    _proc(xexit())
        free := true
        _signal(not_busy)
    _proc_end
_monitor_end
```

Programme 52 : Exemple d'utilisation des moniteurs

```
# gives different output when run with sc than with sw, su, or sx
# demonstrates effect of different signal discipline
# (of course, could be just scheduler, though; but it is not.)
# (output from sw and su is the same)
resource main()
    import foo
    process p(i:=1 to 10)
        write( i, "before" )
        foo.enter()
        write( i, "in" )
        nap(1)
        write( i, "leaves" )
        foo.xexit()
        write( i, "after" )
    end
end
```

Programme 53 : Exemple d'utilisation des moniteurs

```
/* CSCAN disk scheduler */
_monitor(cscan)
    op request(cyli: int), release()
_body(cscan)
    _condvar1(scan, 0:1 )
    var c := 0, n := 1, pos := -1

    _proc( request(cyli) )
        if pos = -1 ->
            pos := cyli
        [] pos != -1  and cyli > pos ->
             _pri_wait(scan[c],cyli)
        [] pos != -1  and cyli <= pos ->
             _pri_wait(scan[n],cyli)
        fi
    _proc_end

    _proc( release() )
      write("on release, current scan; next scan:")
  _print(scan[c]);  _print(scan[n])
        if  not  _empty(scan[c]) ->
            pos :=  _minrank(scan[c])
        []  _empty(scan[c])  and  not  _empty(scan[n]) ->
            c :=: n
            pos :=  _minrank(scan[c])
        []  _empty(scan[c])  and  _empty(scan[n]) ->
            pos := -1
        fi
        _signal(scan[c])
    _proc_end
_monitor_end

resource main()
    import cscan
    process p(i:=1 to 10)
        write( i, "before" )
  var k: int
     fa h := 1 to 2 ->
  nap(10-i)
  if i > 5 -> k := 10+i [] i = 5 -> k := 12 [] else -> k := 10-i  fi
        cscan.request(k)
        write( i, "got it" )
        nap(20)
        write( i, "done" )
        cscan.release()
        write( i, "after" )
    af
    end
end
```

Programme 54 : Exemple d'utilisation des moniteurs

```
resource main()
    import repro, cscan
    process p1
        var me := 1
        repro.entry(me)
        cscan.request(40)
        repro.entry(me)
        write(me, "releasing after", 40)
        cscan.release()
        repro.entry(me)
        cscan.request(20)
        repro.entry(me)
        write(me, "releasing after", 20)
        cscan.release()
    end
    process p2
        var me := 2
        repro.entry(me)
        cscan.request(35)
        repro.entry(me)
        write(me, "releasing after", 35)
        cscan.release()
        repro.entry(me)
        cscan.request(60)
        repro.entry(me)
        write(me, "releasing after", 60)
        cscan.release()
        repro.entry(me)
        cscan.request(10)
        repro.entry(me)
        write(me, "releasing after", 10)
        cscan.release()
        repro.entry(me)
        cscan.request(80)
        repro.entry(me)
        write(me, "releasing after", 80)
        cscan.release()
    end
    process p3
        var me := 3
        repro.entry(me)
        cscan.request(30)
        repro.entry(me)
        write(me, "releasing after", 30)
        cscan.release()
        repro.entry(me)
        cscan.request(50)
        repro.entry(me)
        write(me, "releasing after", 50)
        cscan.release()
        repro.entry(me)
        cscan.request(5)
        repro.entry(me)
        write(me, "releasing after", 5)
        cscan.release()
    end
end
```

Programme 55 : Exemple d'utilisation des moniteurs

```
/* CSCAN disk scheduler problem
 * like other cscan test, but forces reproducibility.
 * tests: nested monitor calls (exclusion in first not released)
 * _pri_wait, _minrank, _empty, _print,
 * arrays of condition variables,
 * two monitors in a file.
 */
_monitor(repro)
    op entry(pid: int), increment()
_body(repro)
    /* for reproducibility:
     * order[turn] is pid of process allowed to enter monitor next.
     * hangout[pid] is private condition variable for process pid
     * on which it waits until its turn.
     * assume we know in advance number of processes.
     */
    var turn := 1
    /* output will be the same as the releases below. */
    var order[19] := ( 1, /* requests 40 */
                       1, /* releases */
                       3, /* requests 30 */
                       2, /* requests 35 */
                       1, /* requests 20 */
                       3, /* releases */
                       3, /* requests 50 */
                       2, /* releases */
                       2, /* requests 60 */
                       3, /* releases */
                       2, /* releases */
                       2, /* requests 10 */
                       3, /* requests 5 */
                       1, /* releases */
                       3, /* releases */
                       2, /* releases */
                       2, /* requests 80 */
                       2, /* releases */
                       1) /* extra to simplify termination */
                          /* 1, 2, or 3 will work. */

    _condvar1(hangout, 3 )

    _proc( entry(pid) )
        if order[turn] != pid -> _wait(hangout[pid]) fi
    _proc_end

    _proc( increment() )
        turn++
        _signal(hangout[order[turn]])
    _proc_end
_monitor_end

_monitor(cscan)
    op request(cyli: int), release()
_body(cscan)
    import repro
    _condvar1(scan, 0:1 )
    var c := 0, n := 1, pos := -1
```

Programme 56 : Exemple d'utilisation des moniteurs

```
resource main()
    import repro, cscan
    process p1
        var me := 1
        repro.entry(me)
        cscan.request(40)
        repro.entry(me)
        write(me, "releasing after", 40)
        cscan.release()
        repro.entry(me)
        cscan.request(20)
        repro.entry(me)
        write(me, "releasing after", 20)
        cscan.release()
    end
    process p2
        var me := 2
        repro.entry(me)
        cscan.request(35)
        repro.entry(me)
        write(me, "releasing after", 35)
        cscan.release()
        repro.entry(me)
        cscan.request(60)
        repro.entry(me)
        write(me, "releasing after", 60)
        cscan.release()
        repro.entry(me)
        cscan.request(10)
        repro.entry(me)
        write(me, "releasing after", 10)
        cscan.release()
        repro.entry(me)
        cscan.request(80)
        repro.entry(me)
        write(me, "releasing after", 80)
        cscan.release()
    end
    process p3
        var me := 3
        repro.entry(me)
        cscan.request(30)
        repro.entry(me)
        write(me, "releasing after", 30)
        cscan.release()
        repro.entry(me)
        cscan.request(50)
        repro.entry(me)
        write(me, "releasing after", 50)
        cscan.release()
        repro.entry(me)
        cscan.request(5)
        repro.entry(me)
        write(me, "releasing after", 5)
        cscan.release()
    end
end
```

Programme 57 : Exemple d'utilisation des moniteurs

```
/* interval timer using a covering condition */
_monitor(timer)
  op delay(interval : int), tick()
_body(timer)

  var tod := 0
  _condvar(check)

  _proc( delay(interval) )
    var wake_time : int
    wake_time := tod + interval
    do wake_time > tod -> _wait(check) od
  _proc_end

  _proc( tick() )
    tod++
    _signal_all(check)
  _proc_end

_monitor_end

/* results will almost differ between executions.
 * uses signal_all (so only works with sc).
 */
resource test()
  import timer
  var done := 0

  write("simulation started at", age())

  process user(i := 1 to 4)
    fa j := 1 to 10 ->
      write("process", i, "sleeping at", age(), "for", 100*(i+j) )
      timer.delay(10*(i+j))
      write("process", i, "awake at", age())
    af
    done++
  end
  process clock
    do true ->
      nap(10)
      timer.tick()
      if done = 4 -> exit fi
    od
    write("simulation ended at", age())
  end
end
```

5.4.2 Les régions critiques conditionnelles

Les régions critiques conditionnelles sont définies de façon similaire aux ressources. Pour définir une ressource à utiliser dans une région critique conditionnelle, on utilise le mot réservé "`_resource`". On regroupe dans une telle ressource toutes les variables à utiliser exclusivement dans la région critique conditionnelle. Le format de déclaration est le suivant :

```
_resource(nom_région)
    déclaration des variables
_resource_end(nom_région)
```

Les variables définies dans une telle ressource ne peuvent être utilisées que dans une région critique conditionnelle.

Lorsqu'un région est définie, on peut l'accéder dans un processus grâce à l'énoncé "`région`". Cet énoncé a le format suivant :

```
_région(nom_région, condition)
    énoncés
_region_end(nom_région)
```

Les énoncés peuvent être des opérations sur des variables locales ou des variables définies dans la région spécifiée. La condition est un test sur une ou des variables définies avec la région.

Exemples

Programme 58 : Programme qui utilise les RCC

```
\_resource(mutex)
   var xfree := true
\_resource\_end(mutex)

resource foo()
  process p(i := 1 to 10)
    nap(10-i)
    \_region(mutex,xfree)
      write(i); nap(1)
      xfree := false
    \_region\_end(mutex)
    nap(2)
    \_region(mutex,true)
      write(i, "done"); nap(1)
      xfree := true
    \_region\_end(mutex)
  end
end
```

Programme 59 : Programme qui utilise les RCC

```
/* same as a.ccr in ../1 but vary timing
 * still not that interesting ...
 */
\_resource(mutex)
   var xfree := true
\_resource\_end(mutex)

resource foo()
  process p(i := 1 to 10)
    if i%2 = 0 -> nap(20*i) fi
    \_region(mutex,xfree)
      write(i); nap(10+i)
      xfree := false
    \_region\_end(mutex)
    \_region(mutex,true)
      write(i, "done"); nap(i)
      xfree := true
    \_region\_end(mutex)
  end
end
```

Programme 60 : Programme qui utilise les RCC

```
\_resource(mutex)
   var xfree := true
\_resource\_end(mutex)
\_resource(mutex2)
   var xfree := true
\_resource\_end(mutex2)

resource foo()
  process p(i := 1 to 10)
    nap(10-i)
    \_region(mutex,xfree)
      write(i); nap(1)
      xfree := false
    \_region\_end(mutex)
    nap(2)
    \_region(mutex,true)
      write(i, "done"); nap(1)
      xfree := true
    \_region\_end(mutex)
  end
  process p2(i := 1 to 10)
    nap(10-i)
    \_region(mutex2,xfree)
      write(i); nap(1)
      xfree := false
    \_region\_end(mutex2)
    nap(2)
    \_region(mutex2,true)
      write(i, "dones"); nap(1)
      xfree := true
    \_region\_end(mutex2)
  end
end
```

Programme 61 : Programme qui utilise les RCC

```
/** this should fail during SR compilation.
 ** see comment below.
 **/
\_resource(mutex)
   var xfree := true
\_resource\_end(mutex)
\_resource(mutex2)
   var xfree := true
\_resource\_end(mutex2)

resource foo()
  process p(i := 1 to 10)
    nap(10-i)
    \_region(mutex,xfree)
      write(i); nap(1)
      xfree := false
    \_region\_end(mutex)
    nap(2)
    \_region(mutex,true)
      write(i, "done"); nap(1)
      xfree := true
    \_region\_end(mutex)
  end
  process p2(i := 1 to 10)
    nap(10-i)
    \_region(mutex2,xfree)
      write(i); nap(1)
      xfree := false
    \_region\_end(mutex2)
    nap(2)
/* test that can't access xfree here. */
    xfree := true
    \_region(mutex2,true)
      write(i, "dones"); nap(1)
      xfree := true
    \_region\_end(mutex2)
  end
end
```

Programme 62 : Programme qui utilise les RCC

```
\_resource(bb)
   const N := 10
   var count := 0, front := 1, rear := 1
   var buf[N]: int
\_resource\_end(bb)

resource foo()
  process depositer(i := 1 to 10)
    nap(10-i)
    \_region(bb,count < N)
      write("depositer",i); nap(1)
      buf[rear] := i*100; rear := rear%N + 1; count++
    \_region\_end(bb)
  end
  process fetcher(i := 1 to 10)
    nap(4-i)
    \_region(bb,count>0)
      write("fetcher",i); write("fetch",buf[front]); nap(1)
      front := front%N + 1; count--
    \_region\_end(bb)
  end
end
```

5.4.3 Émulation de CSP

CSP est un langage implantant le concept de communication de messages par rendez-vous (aucun emmagasinage de messages). On retrouve dans l'émulation de CSP le concept de programme (_program) et de processus. Les processus possèdent une partie spécification(_specs, _specs_end, _process_spec, _process_spec1, _process_spec2) et une partie implantation (_process_body, _process_body1, _process_body2).

Le format d'un programme CSP est le suivant :

```
_program(nom_pgm)
      constantes, variables, types
_specs
      spécification de processus
_specs_end
      implantation des processus
_program_end [nom_moniteur]
```

La spécification d'un processus est la suivante :

```
_process_spec(nom_pcs)
    déclarations de portes
```

Les portes servent à la communication par messages. Elles sont déclarées de la façon suivante :

```
_port_spec(nom_pcs, nom_porte, (paramètres))
```

L'implantation d'un processus a le format suivant :

```
_process_body(nom_pcs)
    constantes, variables, types, énoncés SR
    énoncés de communications
_process_end
```

Pour la communication par messages, il y a les énoncés de réception (`_stnt_i` et `_guard_i`) et les énoncés d'envoie (`_stmt_o` et `_guard_o`) de messages. Ils ont le format suivant :

```
_stmt_i(nom_pcs, nom_porte,(paramètres)
_guard_i(expression booléenne, nom_pcs, nom_porte,(paramètres)

_stmt_o(nom_pcs, nom_porte,(paramètres )
_guard_o(expression booléenne, nom_pcs, nom_porte, (paramètres) )
```

Il est possible de définir des processus qui recoivent des paramètres. Ces paramètres servent seulement à spécifier le nombre de processus à démarrer.Dans ce cas, on utilise :

— `_process_spec1` et `_process_body1` pour définir un processus recevant un seul paramètre.

— `_process_spec2` et `_process_body2` pour définir un processus recevant deux paramètres.

— ...

Les paramètres sont spécifiés par paire `(i, j)` où j-i donne le nombre de processus à créer et les valeurs de i à j représentent les index des processus. Ainsi,

Programme 63 : Programme qui multiplie des matrices en parallele

```
\_process\_spec1(A, 1,2)
```

permet de démarrer deux processus numérotés 1 et 2. L'énoncé

Programme 64 : Programme qui multiplie des matrices en parallele

```
\_process\_spec2(A, 1,2,1,2)
```

permet de démarrer quatre processus indexés (1,1), (1,2), (2,1) et (2,2). La même syntaxe s'applique à l'implantation (`_process_body1` et `_process_body2`).

Voici quelques exemples de programmes CSP :

Programme 65 : programme qui utilise CSP

```
\_program(a)
  \_specs
    \_process\_spec(A)
      /* don't put spaces around op or process names */
      \_port(A,o1, (x: int))

    \_process\_spec(B)
      \_port(B,o2, (x: int) )
  \_specs\_end

  \_process\_body(A)
    var x: int
    write("start of A", A, B)
    \_stmt\_i(B,o1,(x))
    write("in middle of A")
    \_stmt\_o(B,o2,(17))
    write(x)
  \_process\_end

  \_process\_body(B)
    var x: int
    write("start of B", B, A)
    \_stmt\_o(A,o1,(3))
    write("in middle of B")
    \_stmt\_i(A,o2,(x))
    write(x)
  \_process\_end

\_program\_end
```

Programme 66 : programme qui utilise CSP

```
\_program(b)

  \_specs
    \_process\_spec2(A, 1,2, 1,3)
      \_port(A,o1, (x: int))

    \_process\_spec(B)
      \_port(B,o2, (x: int) )
  \_specs\_end

/* this is used only for debugging csp2sr. */
  \_dump\_pidx

  \_process\_body2(A, i, j)
    var x: int
    write("start of A", i, j)
    \_stmt\_i(B,o1,(x))
    write("in middle of A")
    \_stmt\_o(B,o2,(17))
    write(x)
  \_process\_end

  \_process\_body(B)
    var x: int
    write("start of B")
    \_stmt\_o(A[2][2],o1,(3))
    write("in middle of B")
    \_stmt\_i(A[2][2],o2,(x))
    write(x)
  \_process\_end

\_program\_end
```

Programme 67 : programme qui utilise CSP

```
\_program(c)

  \_specs
    \_process\_spec2(A, 1,4, 1,4)
      \_port(A,o1, (x,y: int))
  \_specs\_end

  \_process\_body2(A, i, j)
    var x := -1, y := -2
    write("start of A", i, j)
    if i < j ->
      \_stmt\_i(A[j][i],o1,(x,y))
      \_stmt\_o(A[j][i],o1,(i,j))
    [] i > j ->
      \_stmt\_o(A[j][i],o1,(i,j))
      \_stmt\_i(A[j][i],o1,(x,y))
    fi
    write(i,j,"got",x,y)
  \_process\_end

\_program\_end
```

Programme 68 : programme qui utilise CSP

```
\_program(d)
  const N := 100
  \_specs
    \_process\_spec1(A, 1,N)
      \_port(A,goo, ())

    \_process\_spec(B)
      \_port(B,foo, (x, i:int))
  \_specs\_end

  \_process\_body1(A, i)
    write("start of A", i)
    \_stmt\_o(B,foo,(-i,i))
    \_stmt\_i(B,goo,())
  \_process\_end

  \_process\_body(B)
    write("start of B")
    fa i := 1 to N-2 ->
      var x, which: int
      \_stmt\_iq1(i, 1,N, A[i],foo, (x, which))
      write(x)
      \_stmt\_o(A[which],goo, ())
    af
  \_process\_end

\_program\_end
```

Programme 69 : programme qui utilise CSP

```
\_program(e)
  \_specs
    \_process\_spec(A)
      \_port(A,foo, (x:int))
    \_process\_spec(B)
  \_specs\_end

  \_process\_body(A)
    write("start of A")
    fa k := 1 to 4 ->
      var x: int
      \_if
        \_guard\_i(true,B,foo, (x)) ->
          write("got", x)
      \_fi
    af
  \_process\_end

  \_process\_body(B)
    write("start of B")
    fa k := 1 to 4 ->
      \_stmt\_o(A,foo, (k*100))
    af
  \_process\_end

\_program\_end
```

Programme 70 : programme qui utilise CSP

```
\_program(i)

  \_specs
    \_process\_spec(X)
  \_specs\_end

  \_process\_body(X)
    write("start of X")
    var c := 'a'
    \_do
      \_guard(c<='z') -> writes(c); c++
    \_od
    write()
  \_process\_end

\_program\_end
```

Programme 71 : programme qui utilise CSP

```
\_program(f)
  \_specs
    \_process\_spec(A)
    \_process\_spec(B)
      \_port(A,goo, (x:int))
  \_specs\_end

  \_process\_body(A)
    write("start of A")
    fa k := 1 to 2 ->
      var x := 11
      \_if
        \_guard\_o(k>0,B,goo, (k*100)) -> write("A sent", k*100)
      \_fi
      write(x)
    af
  \_process\_end

  \_process\_body(B)
    write("start of B")
    fa k := 1 to 2 ->
      var x := 22
      \_if
        \_guard\_i(true,A,goo, (x)) -> write("B got", x)
      \_fi
      write(x)
    af
  \_process\_end

\_program\_end
```

Programme 72 : programme qui utilise CSP

```
\_program(f)
  \_specs
    \_process\_spec(A)
      \_port(A,foo, (x:int))
    \_process\_spec(B)
      \_port(A,goo, (x:int))
  \_specs\_end

  \_process\_body(A)
    write("start of A")
    fa k := 1 to 4 ->
      var x: int
      \_if
        \_guard\_i(k mod 2 = 0,B,foo, (x)) -> write("A got", x)
        \_guard\_o(k mod 2 = 1,B,goo, (k*100)) -> write("A sent", k
   *100)
      \_fi
    af
  \_process\_end

  \_process\_body(B)
    write("start of B")
    fa k := 1 to 4 ->
      var x: int
      \_if
        \_guard\_i(k mod 2 = 1,A,goo, (x)) -> write("B got", x)
        \_guard\_o(k mod 2 = 0,A,foo, (k*1000)) -> write("B sent", k
   *1000)
      \_fi
    af
  \_process\_end

\_program\_end
```

Programme 73 : programme qui utilise CSP

```
/* dining philosophers */
\_program(dp)
  const N := 5, rounds := 4
  \_specs

    \_process\_spec1(waiter,0,N-1)
      \_port(waiter,need, ())
      \_port(waiter,hungry, ())
      \_port(waiter,full, ())

    \_process\_spec1(phil,0,N-1)
      \_port(phil,eat, ())
  \_specs\_end

  \_process\_body1(waiter,i)
    var peating := false, phungry := false
    var haveL, haveR: bool
    if i = 0   -> haveL := haveR := true
    [] i = N-1 -> haveL := haveR := false
    [] else    -> haveL := false; haveR := true
    fi
    var dirtyL := false, dirtyR := false
    const L := (i-1) mod N
    const R := (i+1) mod N
    \_do
      \_guard\_i(true, phil[i],hungry, ()) ->
        phungry := true
      \_guard(phungry & haveL & haveR) ->
        phungry := false
        peating := dirtyL := dirtyR := true
        \_stmt\_o(phil[i],eat, ())
      \_guard\_o(phungry & not haveL, waiter[L],need, ()) ->
        haveL := true
      \_guard\_o(phungry & not haveR, waiter[R],need, ()) ->
        haveR := true
      \_guard\_i(haveL & not peating & dirtyL, waiter[L],need, ())
   ->
        haveL := dirtyL := false
      \_guard\_i(haveR & not peating & dirtyR, waiter[R],need, ())
   ->
        haveR := dirtyR := false
      \_guard\_i(true, phil[i],full, ()) ->
        peating := false
    \_od
  \_process\_end

  \_process\_body1(phil,i)
    fa k := 1 to rounds ->
      \_stmt\_o(waiter[i],hungry, ())
      \_stmt\_i(waiter[i],eat, ())
      write(i, "eating", k)
      \_stmt\_o(waiter[i],full, ())
    af
  \_process\_end
\_program\_end
```

Programme 74 : programme qui utilise CSP

```
/* parallel sort */
\_program(sort)
  \_specs
    \_process\_spec(P1)
      \_port(P1,pass1, (x: int))
      \_port(P1,print, ())
    \_process\_spec(P2)
      \_port(P2,pass2, (x: int))
  \_specs\_end

  const N := 10
  \_process\_body(P1)
    var a[1:N/2]: int
    fa i := 1 to N by 2 -> a[i/2+1] := i af
    fa k := 1 to N/2 -> write("P1", a[k]) af
    \_stmt\_i(P2,print, ()) /* to sync printing. */
    const largest := N/2; var got: int
    \_stmt\_o(P2,pass2, (a[largest])); \_stmt\_i(P2,pass1, (got))
    do a[largest] > got ->
      var k := 1
      do a[k] < got -> k++ od /* k is at most largest-1 */
      fa j :=  largest-1 downto k ->
        a[j+1] := a[j]
      af
      a[k] := got
      \_stmt\_o(P2,pass2, (a[largest])); \_stmt\_i(P2,pass1, (got))
    od
    fa k := 1 to N/2 -> write("P1", a[k]) af
    \_stmt\_i(P2,print, ()) /* to sync printing. */
  \_process\_end

  \_process\_body(P2)
    var a[1:N/2]: int
    fa i := 1 to N by 2 -> a[i/2+1] := N/2+i-2 af
    \_stmt\_o(P1,print, ()) /* to sync printing. */
    fa k := 1 to N/2 -> write("P2", a[k]) af
    const smallest := 1; var got: int
    \_stmt\_i(P1,pass2, (got)); \_stmt\_o(P1,pass1, (a[smallest]))
    do a[smallest] < got ->
      var k := N/2
      do a[k] > got -> k-- od /* k is at least smallest+1 */
      fa j :=  smallest+1 to k ->
        a[j-1] := a[j]
      af
      a[k] := got
      \_stmt\_i(P1,pass2, (got)); \_stmt\_o(P1,pass1, (a[smallest]))
    od
    \_stmt\_o(P1,print, ()) /* to sync printing. */
    fa k := 1 to N/2 -> write("P2", a[k]) af
  \_process\_end
\_program\_end
```