Concurrent Object Oriented Petri Nets: CO-OPN

Didier Buchs

Plan

Introduction

Elements lexicaux

Syntaxe et sémantique statique

Sémantique

Langage CO-OPN
Concurrent Object-Oriented Petri Nets

Langage de spécification pour systèmes concurrents complexes.

CO-OPN =
Structures de données + Petri Nets + objets

- Structures de données = Algebraic Abstract Data Types
- Concurrence = Algebraic Petri Nets
- Modularité = Notion d‘objet et de classes
- Dynamicité = Notions de références, créations d’objets
- Hiérarchies = Synchronisation entre objets
- Contextes = unités de répartition + informations de migration

Propriétés:

- Sémantique bien définie et de la vraie concurrence (True Concurrency)
- Equivalences, Raffinements
- Vérification par sélection de tests

Outils de support au développement (CoopnTool/SANDS))

- Vérificateur syntaxique
- Outil de simulation
- Générateur de tests
- Outils de présentations graphiques et textuelles

Mécanismes de structuration et de construction

- Généricité et enrichissement
- Classes et instances
- Algèbres de références (OID)

Modules CO-OPN

Structure des unités de modélisations (sections):

- une en-tête, qui contient le genre et le nom du module et les informations liées à la généricité et l'héritage,
- une interface, décrivant les éléments visibles du module. Cette interface inclus les informations disponibles pour les clients de ce module.
- un corps, qui décrit les composants internes du module par exemple les axiomes définissant le comportement des opérations.

```
[OBJECT¦CLASS¦ADT|CONTEXT¦MORPHISM]  M;
;; En - tête
INTERFACE
...;;Interface
BODY
...;;Corps
END M;
```

Element Lexicaux

Les symboles suivants sont réservés:

<ADT> <END> <CLASS> <OBJECT> <CONTEXT> <MORPHISM> <GENERIC> <PARAMETER> <ABSTRACT> <AS> <INHERIT> <INTERFACE> <BODY> <MORPHISM> <RENAME> <REDEFINE> <UNDEFINE> <USE> <SORT> <TYPE> <SUBSORT> <SUBTYPE> <GENERATOR> <OPERATION> <OBJECT> <GATE> <METHOD> <CREATION> <DESTRUCTION> <TRANSITION> <PLACE> <INITIAL> <AXIOM> <THEOREM> <WHERE> <IN> <WITH> <BORROW> <GIVE> <LEND> <TAKE> <SELF> <CREATE> <DESTROY>

ils correspondent aux mots indifféremments majuscules ou minuscules et singulier ou pluriels

Le symbole <WORD> représente tous les mots non réservé

Les symboles suivants sont également réservés:

<;> <(> <)> <:> <,> <->> <=>> <::> <_>

;; est le début de commentaire finissant à la fin de ligne

(: text :) est le commentaire multi-ligne

Les autres symboles utilisés dans les expressions CO-OPN (synchro et conditions) tel que = + & / . etc. ne sont pas réservés mais sont prédéfinis (cf. méta types).

Les symboles réservés ne sont pas utilisables dans les définitions de symboles de l'utilisateur alors que les symboles prédéfinis sont réutilisable par surcharge.

Syntaxe et sémantique statique

Syntaxe très souple, car la notion d'expression est très libre.

Structure des modules très systématique.

Langage déclaratif (conjonction de propriétés).

Sémantique statique autorisant la surcharge et basée sur le typage fort des expressions

Puissant mécanisme de paramétrisation, d'héritage et d'instanciation.

Elements syntaxiques de base

Un fichier peut contenir plusieurs modules

Un module est composé de sections, chaque sections sont composées de champs. Les champs sont eux-mêmes composés de liste de définitions suivants le même principe général.

```
[OBJECT¦CLASS¦ADT¦CONTEXT¦MORPHISM]  M;
;; En - tête
INTERFACE
   USE ;; ...Champs
   OPERATIONS ...
   METHODS ...
BODY
   USE ...
   OPERATIONS ...
   METHODS ...
   AXIOMS ...
END M;
```

Déclarations

Les déclarations en CO-OPN suivent un principe de base, ou chaque déclaration à la forme:

noms : *définition* ;

Les déclarations peuvent être répétées. Dans certains cas la définition peut être vide et une liste de noms peuvent être définis. Nous avons donc la forme générale suivante:

(nom1, nom2, ... , nomn [: définition] ;)*

```
Operations
    _ | _          : sequence, sequence -> sequence;
    # _            : sequence -> natural;
    empty? _, ordered _ : sequence -> boolean;
```

Il est également possible que la liste de noms soit omise dans certains cas (**Axioms**, **Morphisms**) .

Module COOPN

coopnModule

coopnModule ::= modifier
((<ADT> Header (inheritSection)*
[adtInterfaceSection (inheritSection)*]
[adtBodySection (inheritSection)*]
<END> moduleName <;>) |
(<CLASS> Header (inheritSection)*
[classInterfaceSection (inheritSection)*]
[classBodySection (inheritSection)*]
<END> moduleName <;>) |
(<OBJECT> Header (inheritSection)*
[objectInterfaceSection (inheritSection)*]
[objectBodySection (inheritSection)*]
<END> moduleName <;>) |
(<CONTEXT> Header (inheritSection)*
[contextInterfaceSection (inheritSection)*]
[contextBodySection (inheritSection)*]
<END> moduleName <;>) |
(<MORPHISM> Header (inheritSection)*
<END> moduleName <;>))

modifier

modifier ::= [(<GENERIC> | <PARAMETER>)] [<ABSTRACT>]

Instanciation et Généricité

Header

Header ::= simpleModuleName [<(> moduleNameList <)>]
[<AS> moduleName] <;> ((morphismField | renameField))*

L'instanciation est définie par le <AS> suivit du module et de ses paramètres formels. Un renommage permet de produire un nommage correct des les entités désirées.

```
Adt String As OrderedList(Characters);

Rename
    list -> string;
End String;
```

Un morphisme peut etre réutilisé lors d'une instanciation comme dans l'héritage ou le 'use' (ex: `Sequence(Elem):Lifo`)

Un morphisme peut provenir de n'importe quels modules, les modules de morphisme ne définissent que des morphismes.

Modules paramètres

```
Parameter Adt Elem2;
Interface
  Sort elem2;
End Elem2;

Parameter Adt ComparableElem As Elem;
Inherit EquivalenceRelation;
    Rename theSort -> elem;
End ComparableElem;
```

Module génériques

```
Generic Abstract Adt Sequence(Elem);
Interface
Use Elem;Naturals;Booleans; Sort sequence;
Generators
    []    : -> sequence; _ ‘ _ : elem, sequence -> sequence;
Operations
    _ | _         : sequence, sequence -> sequence;
    # _           : sequence -> natural;
    empty? _      : sequence -> boolean;
Body
Axioms
    [] | sequenceVar2 = sequenceVar2;
    (elemVar‘seqVar1) | seqVar2 =elemVar‘(seqVar1 | seqVar2);
    # [] = 0;
    # (elemVar1 ‘ seqVar1) = succ(# seqVar1);
    empty? [] = true;
    empty? (elemVar1 ‘ seqVar1) = false;
Where
    seqVar1, seqVar2 : sequence;elemVar : elem;
End Sequence;
```

Module génériques

```
Generic Abstract Adt LifoStructure(Elem);
Interface
    Use Elem;Naturals;
;; Actually defined when inherited
    Sort lifo;
    Generators
     [] : -> lifo;
     _ ‘ _ : elem, lifo -> lifo;
     Operations ;; Specific elements
     push _ to _ : elem, lifo -> lifo;
     top of _ : lifo -> elem;
     pop from _ : lifo -> lifo;
Body
    Axioms
     push elemVar1 to lifoVar1 = elemVar1 ‘ lifoVar1;

     top of (elemVar1 ‘ lifoVar1) = elemVar1;
     pop from (elemVar1 ‘ lifoVar1) = lifoVar1;
    Where lifoVar1 : lifo; elemVar1 : elem;
End LifoStructure;
```

Morphismes et héritage

inheritSection

inheritSection::=<INHERIT>moduleName<;>((renameField|redefineField|undefineField))*

Chaque module peut induire un morphisme d'instanciation (ex: Lifo) qui associe des entités du paramètre formel avec le paramètre effectif.

Un morphisme peut etre réutilisé lors d'une instanciation comme dans l'héritage ou le 'use' (ex: `Sequence(Elem):Lifo`)

L'héritage est un mécanisme syntaxique faisant intervenir des déclarations de modification du module importé:

- renameField Pour le renommage d'entités du module hérité,
- redefineField Pour la redéfinition des propriétés d'entités du module hérité,
- undefineField Pour la suppression d'entités du module hérité,

Syntaxe des morphismes

morphismField

morphismField ::= <MORPHISM> ((mappingBloc <;>))+

renameField

renameField::=<RENAME> ((mappingBloc <;>))+

redefineField

redefineField ::= <REDEFINE> (referenceBloc <;>)+

undefineField

undefineField ::= <UNDEFINE> (referenceBloc <;>)+

mappingBloc

mappingBloc ::= [nameList <:>] name [<IN> (moduleName | <(> moduleName <)>)] <->> name [<IN> (moduleName | <(> moduleName <)>)]

referenceBloc

referenceBloc ::= [nameList <:>] name [<IN> (moduleName | <(> moduleName <)>)]

Utillisation des morphismes dans les instanciations ‘internes’ : `Sequence(Elem):Lifo`,

```
Generic Adt Lifo(Elem);
Rename
    sequence -> lifo;
Inherit Sequence(Elem):Lifo;

Inherit LifoStructure(Elem);

End Lifo;
```

Interfaces et Corps pour ADT

adtInterfaceSection

adtInterfaceSection ::= <INTERFACE> ((useField | sortField | subsortField | subtypeField | generatorField | operationField))+

adtBodySection

adtBodySection::=<BODY>((useField|sortField|subsortField|subtypeField|generatorField | operationField | adtAxiomField | adtTheoremField | whereField))+

```
Abstract Adt Commutativity;

Interface
    Sort theSort;
    Operation _ theOp _ : theSort, theSort -> theSort;
Body
    Theorem
     Var1 theOp Var2 = Var2 theOp Var1;
    Where
     Var1, Var2 : theSort;
End Commutativity;
```

ADT: Sous-sortes

Signature avec une relation d'ordre sur les sortes: $\Sigma = (S,<,OP)$

Des propriétés doivent être valides sur les signatures afin de garantir une plus petite sorte pour chaque terme.

L'inclusion des sortes correspond à l'inclusion des ensembles de valeurs des algèbres, les fonctions doivent être semblable sur les sous domaines.

Une terme appartient a plusieur sortes

Les sous-sortes permettent de modéliser la partialité.

Exemple

```
Adt Naturals;
Interface
    Sort nz-natural, natural;
    Subsort nz-natural < natural;
Operations
    0:-> natural;
    succ _ : natural -> nz-natural;
    pred _ : nz-natural -> natural;
    _ + _ : natural natural -> natural;
    _ * _ : natural natural -> natural;
    _ div _ : natural nz-natural -> natural;
```

les termes suivants sont de type natural et également nz-natural:

succ(0), succ(succ(0))

les termes suivants sont de type natural uniquement:

pred(succ(succ(0))), (succ(0))*succ(succ(0))

Termes avec variables

Définition: Termes avec variables

Soit $\Sigma = (S,<,OP)$ une signature ,X un S-ensemble de nom de variables, $T_\Sigma(X)$ l'ensemble des termes fermés est le plus petit S-ensemble tel que:

- $\forall\ x \in X_s,\ x \in T_{\Sigma,s}(X)$
- $\forall\ op{:}s_1\ s_2\ ...\ s_n \rightarrow s'$ et $\forall\ (t_1\ t_2\ ...\ t_n) \in T_{\Sigma,s_1}(X) \times T_{\Sigma,s_2}(X) \times ... \times T_{\Sigma,s_n}(X)$ alors $op(t_1\ t_2\ ...\ t_n) \in T_{\Sigma,s}(X)$ pour $s' < s$

Si une signature S est régulière alors tout terme a une plus petite sorte: LS: $T_\Sigma(X)$ -> S.

Axiomes

Une S-equation est une paire t,t' de termes tq. LS(t) et LS(t') sont dans le même composant connecté de (S,<). i.e. il existe une sorte s t.q. LS(t)<s et LS(t')<s.

Exemple: pred(succ(succ(x))) = succ(x)

pour natural:

pred(succ(x)) = x

0 + x = x;
((succ(x)) + y) = succ(x + y);

0 * x   = 0;
((succ(x)) * y) = ((x * y) + y);

Where x,y: natural;

Exemple: Modélisation de la partialité

```
Adt Lists;
Interface
    Sort ne-list, list;
    Subsort ne-list < list;
Operations
    []:-> list;
    add _ to _ : natural list -> ne-list;
    head _ : ne-list -> natural;
    tail _ : ne-list -> list;
Axioms
    head(add n to l) = n;

    tail(add n to l) = l;

Where n : natural; l: list;
```

head et tail sont des fonctions totales sur les listes non-vides, représentant la partialité sur les listes.

Réécriture avec sorte ordonnée:

Vue abstraite: Récriture = règles + mécanisme d'application

mécanisme d'application:

- filtrage

  l ~> r filtre le terme t pour s

  <=> existe substitution s t.q. sl et t = c[sl] ou c[_] est un contexte de terme

  Exemple: succ(pred (succ(succ(0))))

  par pred succ n ~> n et c[_] = succ (_) et s = {n = succ(0)}. n: natural, succ(0): nz-natural;

  sr = succ(0)

- substitution

  si l ~> r filtre le terme t pour s alors on substitue sl par sr:

  t ~> c[sr]

Exemple: succ(pred (succ(succ(0)))) ~> succ(succ(0))

Interfaces et Corps pour Classes

classInterfaceSection

classInterfaceSection ::= <INTERFACE> ((useField | typeField | subsortField | subtypeField | objectField | classGateField | classMethodField | creationField | destructionField))+

classBodySection

classBodySection ::= <BODY> ((useField | typeField | subsortField | subtypeField | objectField | classGateField | classMethodField | creationField | destructionField | transitionField | placeField | initialField | classAxiomField | classTheoremField | whereField))+

```
(::Example of recursive accumulator.:)
Class Accumulator;
Interface
    Use(::Use the natural numbers.:) Naturals;
    Type (::Computer type.:) accumulator;
    Methods
(::Start a computation. : @paramthe number to inject in the
computer. :)
     start _ : natural;
(::Get the result. : @paramthe result. :)
       result _ : natural;
```

```
Body
    Transition tau;
    Places
        i _ : natural;
        r _ : natural;
    Axioms
(::Inject a zero. Terminal case of the recursion.:)
    start 0:: -> r (0);
(::Inject a positive number.:)
    start succ (n):: -> i (succ n);
(::Computes with a positive number.:)
    o = Self =>
        tau With (o . start (n)) . . (o . result (f))::
        i (succ n) -> r ((succ n) + f);
(::Retrieve the result.:)
    result f:: r (f) ->;
    Where
        n : natural;
        f : natural;
        o : accumulator;
End Accumulator;
```

Interfaces et Corps pour Objets

objectInterfaceSection

objectInterfaceSection::= <INTERFACE> ((useField | subsortField | subtypeField | classGateField | classMethodField))+

objectBodySection

objectBodySection ::= <BODY> ((useField | subsortField | subtypeField | classGateField | classMethodField | transitionField | placeField | initialField | classAxiomField | classTheoremField | whereField))+

La notion d'objet est un sucre syntaxique dans COOPN, à chaque objet il est associé implicitement une classe de même nom.

Interfaces et Corps pour Contextes

contextInterfaceSection

contextInterfaceSection::= <INTERFACE> ((useField | subsortField | subtypeField | contextGateField | contextMethodField))+

contextBodySection

contextBodySection ::= <BODY> ((useContextField | subsortField | subtypeField | contextGateField|contextMethodField|objectField|contextAxiomField|contextTheoremField | whereField))+

Utilisation de spécifications

useField

useField ::= <USE> (useBloc <;>)+

useContextField

useContextField ::= <USE> [<CONTEXT>] (useBloc <;>)+

useBloc

useBloc ::= [nameList <:>] moduleNameList

La politique d'importation par le use est celle de la fermeture transitive des dépendances des interfaces.

Typage

sortField

sortField ::= <SORT> (nameBloc <;>)+

typeField

typeField ::= <TYPE> (nameBloc <;>)+

subsortField

subsortField ::= <SUBSORT> (subtypeBloc <;>)+

subtypeField

subtypeField ::= <SUBTYPE> (subtypeBloc <;>)+

nameBloc

nameBloc ::= nameList

subtypeBloc

subtypeBloc ::= [nameList <:>] expr (<->> expr)+

Les sortes utilisables dans les définitions de profil d'opérations, de places, de méthodes ... sont les sortes définies par les champs : <SORT> et <TYPE>.

Implicitement la sorte OBJECT est super type de toutes les sortes définies dans <TYPE> elle n'a pas d'opérations associée.

Operateurs pour ADT

generatorField

generatorField ::= <GENERATOR> (dblExprNameBloc <;>)+

operationField

operationField ::= <OPERATION> (dblExprNameBloc <;>)+

dblExprNameBloc

dblExprNameBloc ::= nameList <:> [[expr] <->>] expr

Objets statiques

objectField

objectField ::= <OBJECT> (exprNameBloc <;>)+

Il s'agit des objets qui existent dans l'état initial et ne peuvent être détruits par une méthode de destruction.

Operateurs d'instances pour Classes, Contextes et Objets

classGateField

classGateField ::= <GATE> (optExprNameBloc <;>)+

optExprNameBloc

optExprNameBloc ::= nameList [<:> expr]

classMethodField

classMethodField ::= <METHOD> (optExprNameBloc <;>)+

contextGateField

contextGateField ::= <GATE> (migrExprNameBloc <;>)+

contextMethodField

contextMethodField ::= <METHOD> (migrExprNameBloc <;>)+

migrExprNameBloc

migrExprNameBloc ::= nameList [<:> migrExpr]

migration

migration ::= (<BORROW> | <GIVE> | <LEND> | <TAKE>)

migrExpr

migrExpr ::= [migration] term ((<,> [migration] | migration) term)*

transitionField

transitionField ::= <TRANSITION> (nameBloc <;>)+

Operateurs de classes pour modules de Classes

creationField

creationField ::= <CREATION> (optExprNameBloc <;>)+

destructionField

destructionField ::= <DESTRUCTION> (optExprNameBloc <;>)+

Etats pour Classes et Objets

placeField

placeField ::= <PLACE> (exprNameBloc <;>)+

initialField

initialField ::= <INITIAL> (exprOptNameBloc <;>)+

exprNameBloc

exprNameBloc ::= nameList <:> expr

exprOptNameBloc

exprOptNameBloc ::= [nameList <:>] expr

Propriétés des Adt

adtAxiomField

adtAxiomField ::= <AXIOM> (adtFormulaBloc <;>)+

adtTheoremField

adtTheoremField ::= <THEOREM> (adtFormulaBloc <;>)+

adtFormulaBloc

adtFormulaBloc ::= [nameList <:>] [[expr] <**=>**>] expr

Propriétés des Classes

classAxiomField

classAxiomField ::= <AXIOM> (classFormulaBloc <;>)+

classTheoremField

classTheoremField ::= <THEOREM> (classFormulaBloc <;>)+

classFormulaBloc

classFormulaBloc ::= [nameList <:>] [[expr] <**=>**>] expr [<WITH> expr] <::> [[expr] <=>>] [[expr] <:>] [expr] <**->**> [expr]

Propriétés des Contextes

contextAxiomField

contextAxiomField ::= <AXIOM> (contextFormulaBloc <;>)+

contextTheoremField

contextTheoremField ::= <THEOREM> (contextFormulaBloc <;>)+

contextFormulaBloc

contextFormulaBloc ::= [nameList <:>] [[expr] <**=>**>] expr <WITH> expr

Variables

whereField

whereField ::= <WHERE> (exprNameBloc <;>)+

exprNameBloc

exprNameBloc ::= nameList <:> expr

Sémantique statique: Expression de synchro

synchronisation de type : META-Synchro

```
SORT META-Synchro;
Operators
_ & _,_ // _, _ + _:META-Synchro, META-Synchro -> META-Synchro;
_ . _ : Object Event -> META-Synchro;
```

Ces opérateurs permettent de décrire les contraintes de synchronisation des classes, objet et contextes.

Sémantique statique: Expression booléenne

description des conditions: META-Boolean

```
SORT META-Boolean;
Operators
_ & _, _ + _:META-Boolean, META-Boolean -> META-Boolean;
! _ : META-Boolean -> META-Boolean;
_ = _: Term , Term -> META-Boolean;

isany _ : Term -> META-Boolean;
isa _ : Term -> META-Boolean;
```

Une condition est un méta-booléen, un axiome est également un méta booléen (=> est l'implication logique).

Les prédicats de typage permettent de vérifier l'appartenance à un type (isa) ou a un type et ses sous-type (isany).

Expressions et termes

expr

expr ::= term (<,> term)*

term

term ::= (factor)+

factor

factor ::= (<WORD> [<IN> (simpleModuleName | <(> moduleName <)>)] | <SELF> | <(> <OBJECT> <)> | <CREATE> | <DESTROY> | <(> expr <)>)

Sémantique statique: Expressions de références

le type Object est prédéfinis de telle manière que, ∀ type,
type < Object

L'auto référence est de type Object: Self:Object

Remarque:

la définition de Self implique que la définition récursive:

compute(x) with self.compute(x)

est incorrectement typée, et doit être exprimée par:

o = self => compute(x) with o.compute(x)

Noms et Listes de noms

nameList

nameList ::= name (<,> name)*

moduleNameList

moduleNameList ::= moduleName (<,> moduleName)*

name

name ::= (<_>)* <WORD> (<_> | <WORD>)*

simpleModuleName

simpleModuleName ::= <WORD>

paramModuleNameList

paramModuleNameList ::= (moduleName [<,> moduleName)*]

moduleName

moduleName ::= simpleModuleName[<(> paramModuleNameList <)> [<:> simpleModuleName]]

Sémantique

Système de transitions = relation labellée entre états

- Etats = ensemble dynamique des états des objets de la spécification

  Contexte = ensemble d'objets

  Objet = produit cartésien de places

  place = multi-ensemble de valeurs algébriques

  _ + _: état état -> état, est la somme symbolique des états
- Label = nom des événements requis et fournis avec leurs paramètres

Construction par application de règles d'inférences (sur les modèles de la partie algébrique) (non décrit dans ce qui suit)

Règles sémantiques CLASS, MONO, SYNC

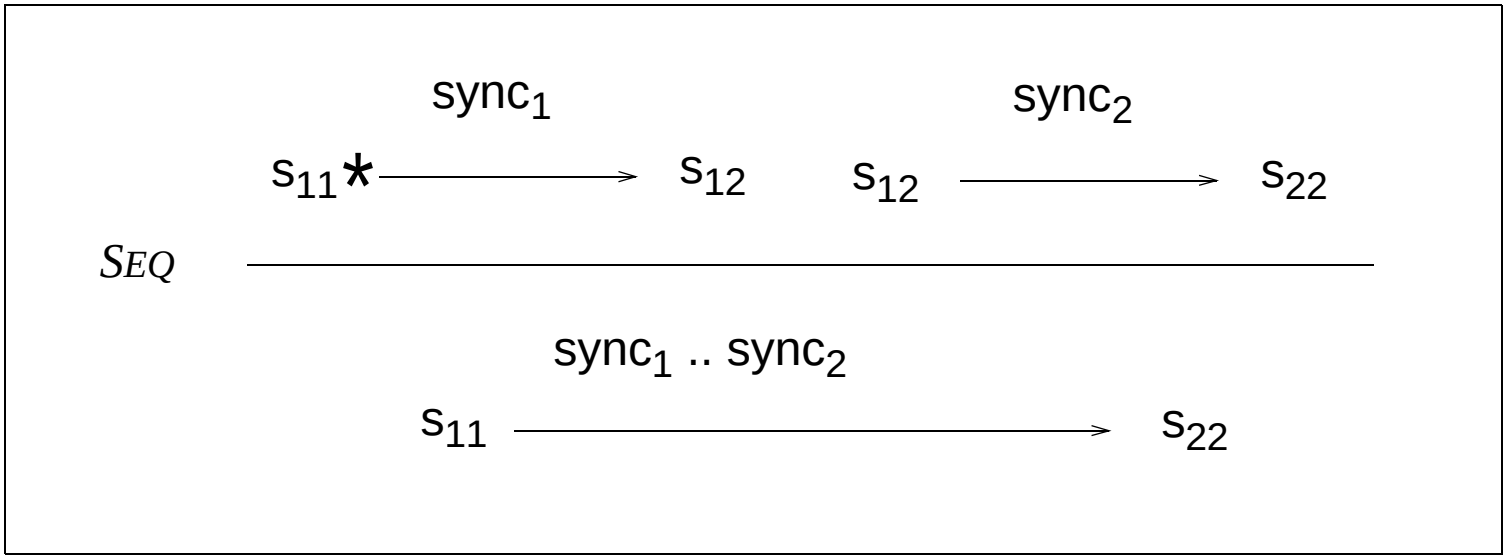
$$\textit{CLASS}\quad \frac{e :: pre_e \rightarrow post_e \in Ax}{pre_e \xrightarrow{e} post_e}$$

$$\textit{MONO}\quad \frac{s' \xrightarrow{e} s''}{s'+s \xrightarrow{e} s''+s}$$

$$\textit{SYNC}\quad \frac{r \xrightarrow{e\ \textbf{With}\ synchro} r' \qquad s \xrightarrow{synchro} s'}{r+s \xrightarrow{e} r'+s'}$$

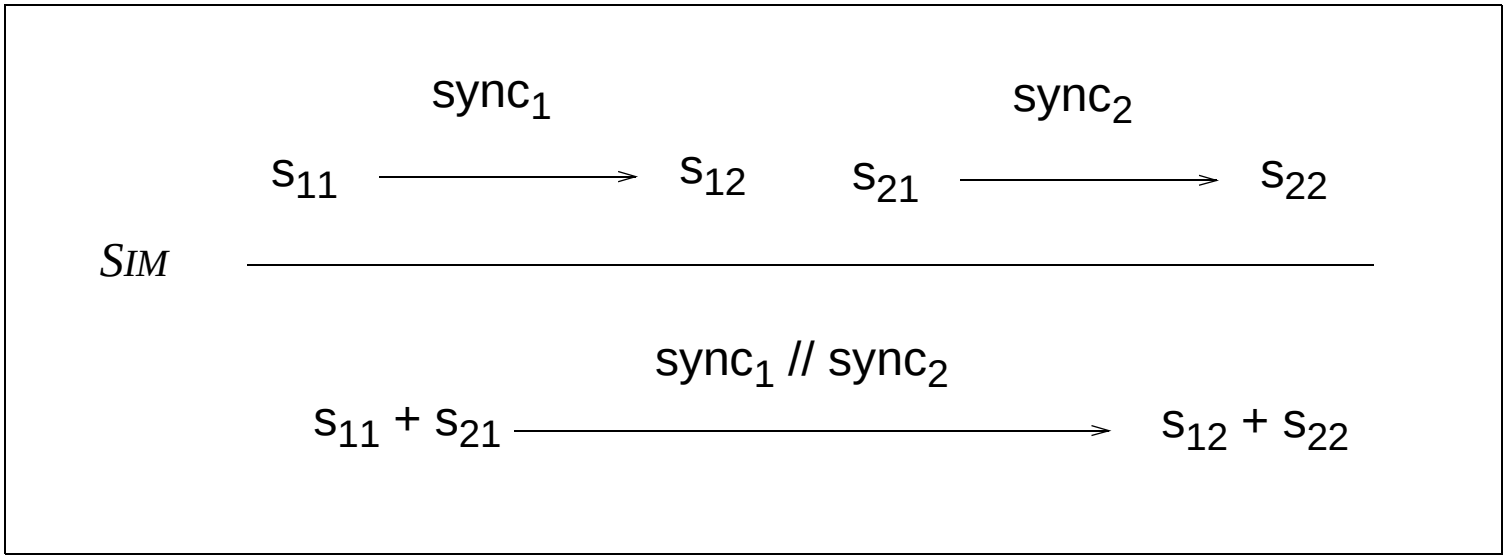
Règles sémantiques SEQ

Eexpressions de synchronisation -Synchronisation séquentielle-



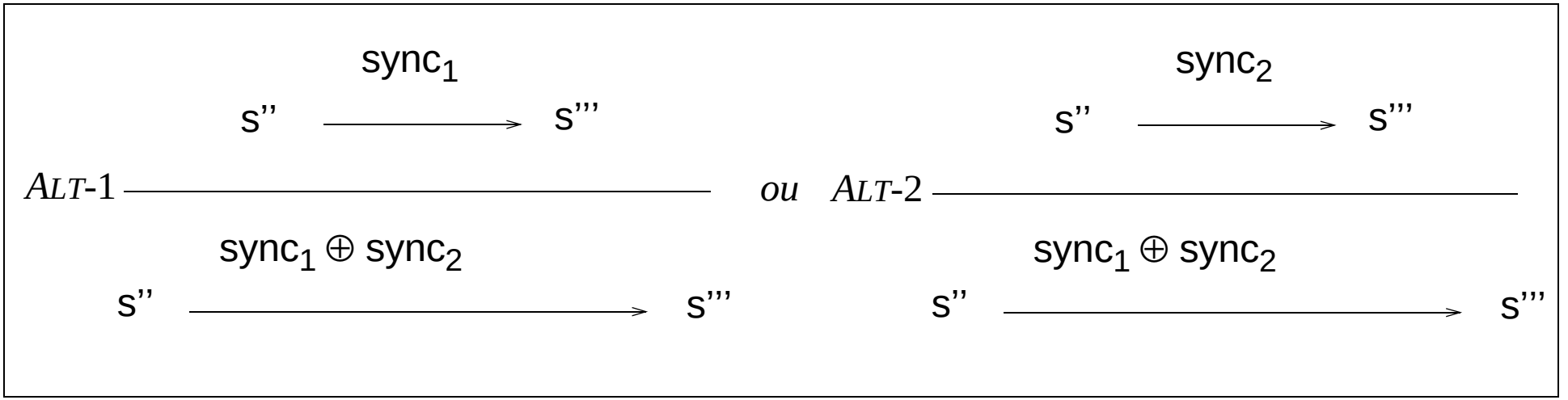
Règles sémantiques SIM

Eexpressions de synchronisation -Synchronisation simultanée-



Règles sémantiques ALT

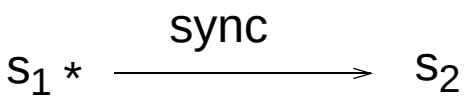
Eexpressions de synchronisation -Synchronisation séquentielle *alternée-*



Construction de la sémantique

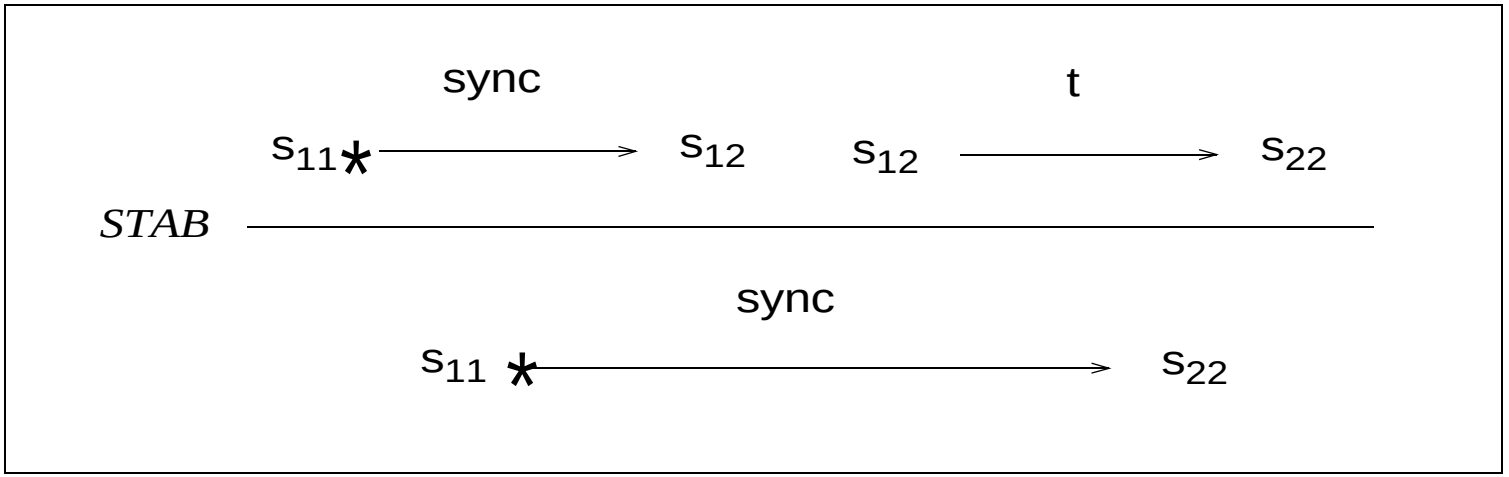
Stabilisation

Calcul des évolutions maximales des transitions internes
=> relation de transition stabilisée.



Règles sémantiques STAB

Stabilisation-



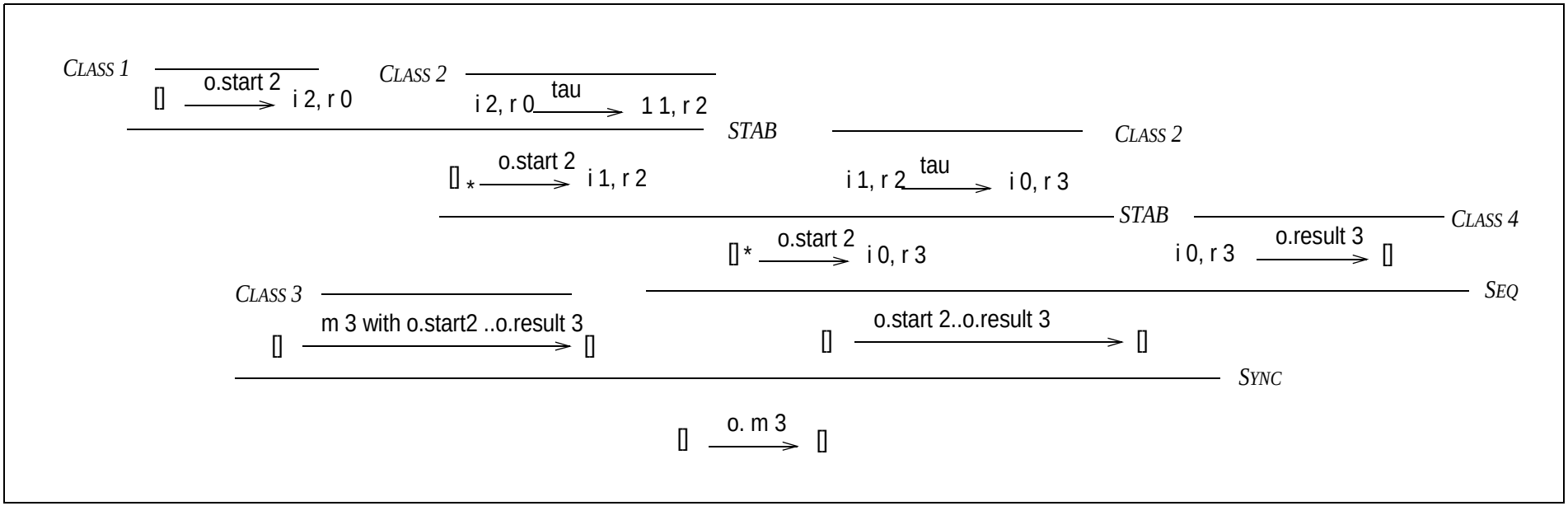
Observation

Suppression des événements n’apparaissant pas dans les définitions d’interfaces.

La sémantique est obtenue par déduction sur les algèbres des TAA (au sens de la déduction des TAA) de tous les comportements et stabilisation des transitions internes. La sémantique finale est celle composée d’événements observables.

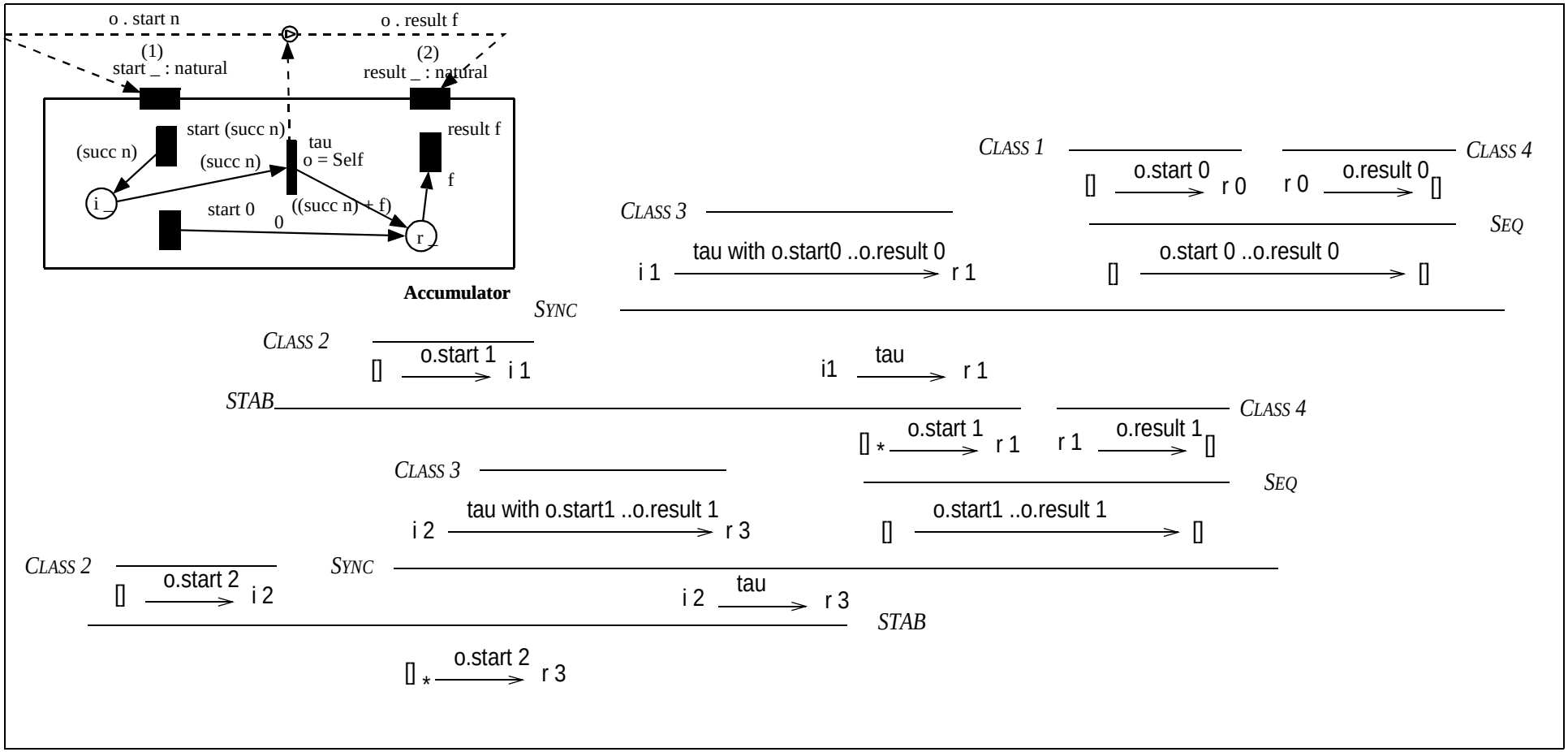
exemple de déduction (somme de valeurs):

ax1: start n:: -> i (n), r 0;
ax2: tau ::i (succ n), r f-> i (n), r (f + succ n);
ax3: m f **With** (o . start (n)) . . (o . result (f)):: ->;
ax4: result f:: i 0,r (f) ->;



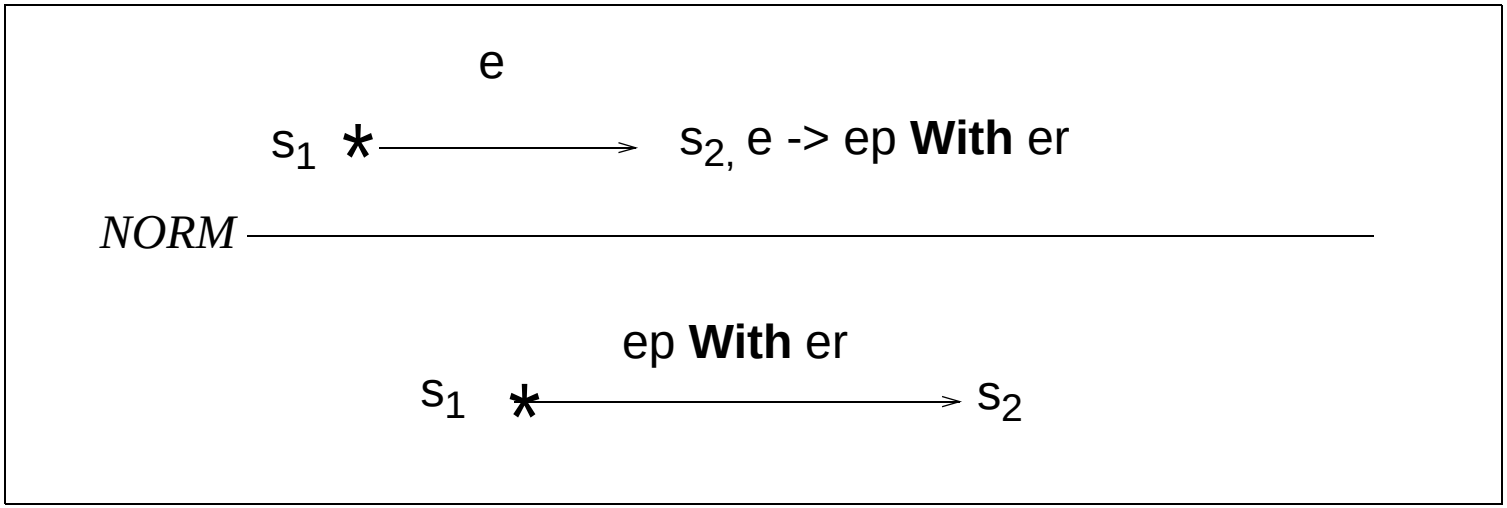
exemple de déduction récursive (accumulator):

ax1: start 0:: -> r (0);
ax2: start succ (n):: -> i (succ n);
ax3: o = **Self** =>tau **With** (o . start (n)) . . (o . result (f))::i (succ n) -> r ((succ n) + f);
ax4: result f:: r (f) ->;



Les classes comme composants

L'adjonction des gates généralise le principe d'événements requis (e With e'). Ceux-ci peuvent devenir membre du résultat des déductions ce qui modifie légèrement les règles précédentes basées sur le principe d'élimination des événements requis. En particulier une relation de normalisation (-> définiis avec -ε>) baséee sur une réduction des expressions de synchronization doit être utilisée.



Règles sémantiques de réduction des synchronizations

Réduction des expressions de synchronisation sous forme normales (term **With** term).

REDUC-OP

$$\frac{e \rightarrow ep \ \textbf{With} \ er \ , \ e' \rightarrow ep' \ \textbf{With} \ er', \ op \in \{//,..,+\}}{(e) \ op \ (e') \rightarrow (ep \ op \ ep') \ \textbf{With} \ (er \ op \ er')}$$

REDUC-WITH

$$\frac{e \rightarrow ep \ \textbf{With} \ er \ , \ e' \rightarrow ep' \ \textbf{With} \ er'}{(e) \ \textbf{With} \ (e') \rightarrow (ep \ // \ er') \ \textbf{With} \ (er \ // \ ep')}$$

Cas de bases

REDUC-METHOD

$$\frac{m \in (M \cup \{creation, destruction, \tau\})}{o.m \rightarrow o.m \textbf{ With } \varepsilon}$$

REDUC-GATE

$$\frac{g \in G}{o.g \rightarrow \varepsilon \textbf{ With } o.g}$$

REDUC-E

$$\frac{op \in \{//,..,+\},\ e \xrightarrow{\varepsilon} er,\ e' \xrightarrow{\varepsilon} er'}{e \textbf{ With } e' \rightarrow er \textbf{ With } er'}$$

REDUC-RE

$$\frac{op \in \{//,..,+\},\ e \xrightarrow{\varepsilon} e'}{e\ op\ \varepsilon \xrightarrow{\varepsilon} e'}$$

REDUC-LE

$$\frac{op \in \{//,..,+\},\ e \xrightarrow{\varepsilon} e'}{\varepsilon\ op\ e \xrightarrow{\varepsilon} e'}$$

REDUC-IDE

$$\frac{m \in M \cup G \cup \{\tau\}}{m \xrightarrow{\varepsilon} m}$$

Sémantique

Sémantique d'un système réactif ouvert

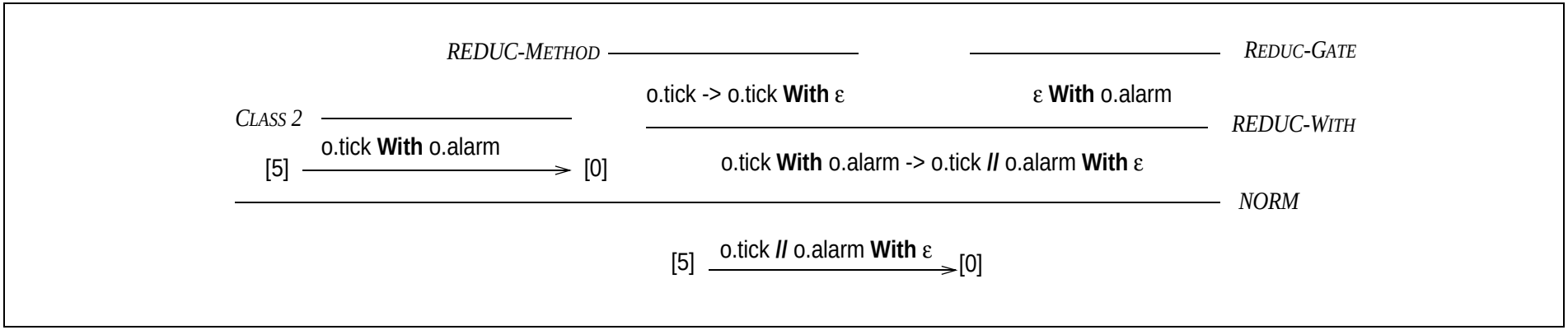
Il s'agit de la sémantique construite avec les règles précédentes. Les événements déduits sont des expressions de synchronizations de la forme (e With e'), ce qui indique que des services tels que e' ne sont pas encore connus, mais participent à l'élaboration de comportement de e.

Sémantique d'un système réactif fermé

Dans la sémantique d'un système fermé. Les événements tels que e With e' ne sont pas intéressant, par contre un évenement m // g, indique qu'une émission synchrone d'une sortie g (un gate) est associée à l'événement en entrée m.

exemple de déduction (événement émis conséquent d'un appel de méthode):

Method
tick;
Gate
alarm;
Axioms
ax1: (n <= 5) = true => tick:: count n -> count (succ n);
ax2: (n <= 5) = false=> tick **With** alarm:: count n -> count 0;



Contextes

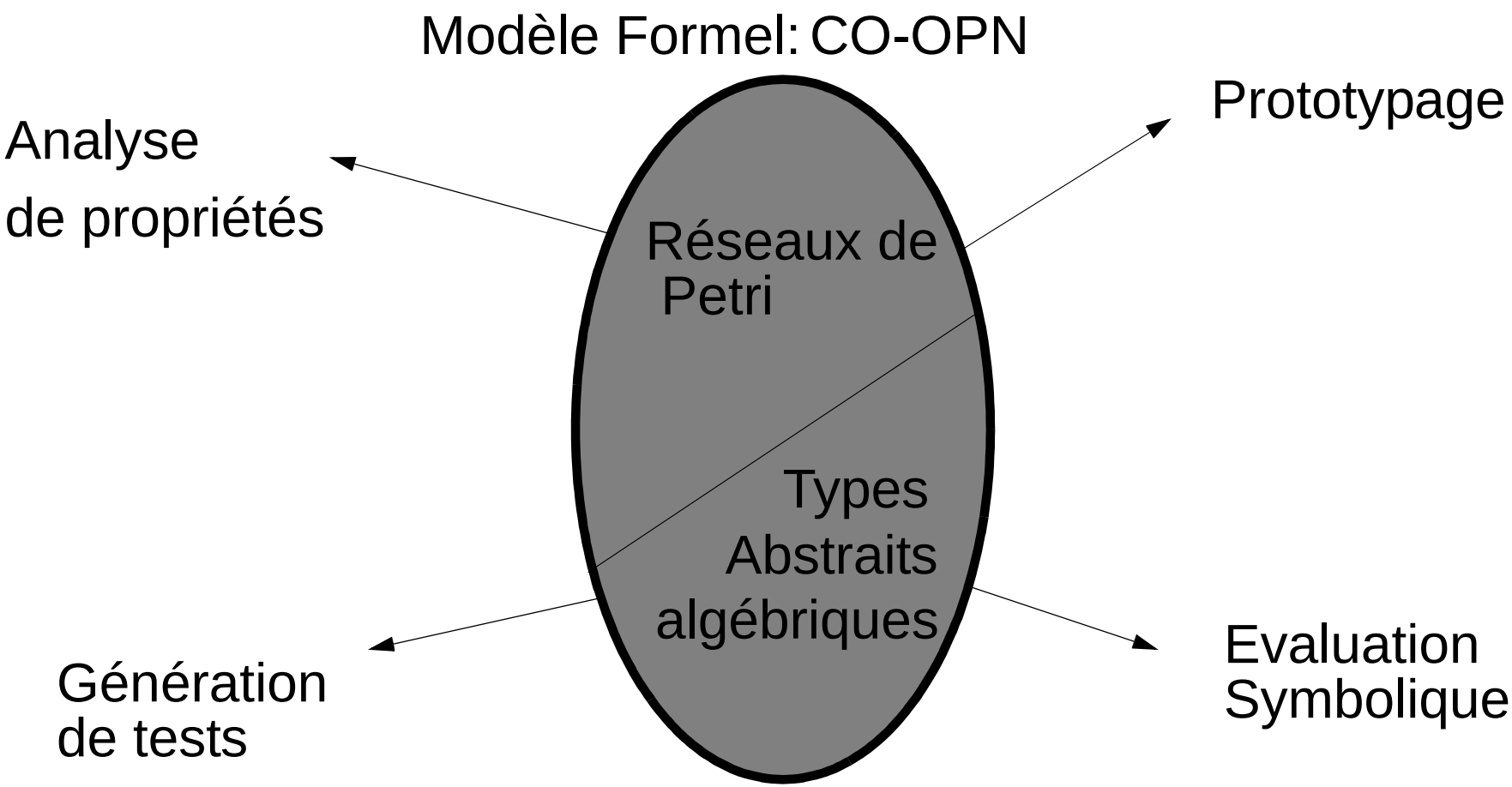
Les contextes décrivent l'aspect physique du système à modéliser c'est à dire la localisation des instances physiques.

Des règles supplémentaires règlent le problème de la migration de ces objets physiques.

les paramètres des méthodes contiennent des attributs décrivant le sens de la migration des objets. Les mots clés TAKE et GIVE sont utilisés pour cette fonctionnalité.

Un contexte ne décrit pas d'état propre, mais seulement un conteneur d'objets communicants.

Techniques Associées:



Vérification de propriété sur une modélisation CO-OPN (réseaux algébriques):

- Propriétés attendues similaires au réseaux places transitions
- La puissance calculatoire des réseaux algébriques rend difficile le calcul des propriétés
  => techniques symboliques, simulations (cf. réécriture)
- Notion de squelette = réseau avec jetons anonymes
  <=> comportement dans l'algèbre triviale
- Certains résultats valides dans le squelette transposables dans tous les modèles

  par ex.:
  blocage dans le squelette =>
  blocage dans tous les modèles

Implémentation de CO-OPN (réseaux algébriques):

- Un programme implémentant un réseau doit avoir le même ensemble d'événements observables.

- Un programme est une implémentation correcte d'un réseau algébrique <=> il a le 'même' comportement observable. (par la notion de bisimulation)

- La compositionalité de l'implémentation entre partie réseau et partie algébrique n'est possible que pour des modèles algébriques observationnellement équivalent par rapport aux termes générés par le réseau.