Langage de production : description cohérente du court terme (deuxième partie)

Jean Hilger

INRIA

UNITÉ DE RECHERCHE
INRIA-LORRAINE

Rapports de Recherche

N° 1039

Programme 5

LANGAGE DE PRODUCTION : DESCRIPTION COHERENTE DU COURT TERME

(DEUXIEME PARTIE)

Jean HILGER

Mai 1989

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Langage de Production:

description cohérente du court terme

(DEUXIEME PARTIE)

Production Language:
coherent description of the short term

(SECOND PART)

Jean HILGER

projet SAGEP

INRIA-LORRAINE
Technopôle de NANCY-BRABOIS
Campus Scientifique
Bd des Aiguillettes, BP 239
54506 VANDOEUVRE-LES-NANCY

* Ce travail a été partiellement financé par le Groupement Scientifique Mathématiques appliquées, Informatique et Gestion associant la FNEGE, l'INRIA et le CNRS (groupe de travail LAPON).

Abstract

In this paper we present a description and command language for hierachical management of discrete production systems. The language is designed to guarantee coherency of any description and command it provides.

Looking at the short term level of production system management as a information system we specify algebraically data types it contains and show properties of our specification. We give an implementation of our specification with an executable specification language and discuss the syntactic interfaces with the specification.

KEY-WORDS: PRODUCTION MANAGEMENT, PRODUCTION LANGUAGE, INFORMATION SYSTEM, ABSTRACT DATA TYPES, GRAMMAR, SPECIFICATION, ALGEBRAIC SPECIFICATION, THEOREM PROVING, REWRITING, INITIAL ALGEBRA, FREE ALGEBRA, CONSISTENCY, COMPLETNESS.

Résumé

Nous présentons dans ce papier un langage de description et de commande pour la gestion hiérarchisée de systèmes de production discrets. Le langage que nous developpons garantit la cohérence des descriptions faites.

Nous considérons le niveau de gestion du court terme d'un système de production comme un système d'information, nous spécifions algébriquement les types de données qu'il contient et nous montrons les propriétés de notre spécification. Nous donnons une implémentation de notre spécification dans un langage de spécification exécutable et nous discutons des interfaces syntaxiques avec la spécification.

MOTS-CLE: GESTION DE PRODUCTION, LANGAGE DE PRODUCTION, SYSTEME D'INFORMATION, TYPES ABSTRAITS, GRAMMAIRE, SPECIFICATION, DEMONSTRATION AUTOMATIQUE, REECRITURE, ALGEBRE INITIALE, ALGEBRE LIBRE, CONSISTANCE, COMPLETUDE.

INTRODUCTION

La nature du langage de production

De multiples approches sont proposées pour appréhender les problèmes de gestion d'un système de production. Chacune d'elles présente ses propres concepts de structuration du système et utilise des hypothèses sur le modèle de gestion. La structure du modèle, les problèmes à résoudre et les solutions calculées sont exprimés dans le langage spécifique à l'approche choisie. La diversité des langages spécifiques soulève un problème d'unification du discours de la gestion de la production. Nos recherches ont pour but la conception d'un langage unifié pour la formulation des problèmes de gestion. Dans la suite, nous appelons ce langage : langage de production.

Nous définissons le **langage de production** comme un **langage de description et de commande cohérent pour une gestion hiérarchisée de la production.** Dans cette approche nous considérons le système de production comme un système d'information, c'est-à-dire un ensemble structuré d'informations concernant la gestion de la production.

Le langage de production est hiérarchisé en référence à un modèle de **gestion hiérarchisée** de la production. Une gestion hiérarchisée réduit la complexité des problèmes de gestion à résoudre. Chaque niveau hiérarchique possède un modèle dont on calcule le contrôle optimal. Les niveaux de la hiérarchie sont liés entre eux : le contrôle optimal à un niveau donné fournit les contraintes à appliquer à la recherche d'un contrôle optimal au niveau hiérarchique immédiatement inférieur, s'il existe. Sinon le contrôle optimal trouvé s'applique à l'atelier.

Le langage de production est un langage de **description** en ce sens qu'il fait intervenir dans ses phrases les noms des types d'information intéressant un niveau hiérarchique donné et exprime les liens spécifiés entre types.

Le langage de production est un langage de **commande** en ce sens qu'il permet de formuler des décisions résultant de processus variés.

Le langage de production permet des descriptions et commandes **cohérentes** en ce sens qu'elles peuvent être prouvées valides à partir d'une spécification consistante des types d'information.

Les applications

Les applications possibles du langage de production se situent à la fois dans le domaine de la recherche et de l'industrie.

Le langage se présente comme un **moyen d'unification** de modèles de gestion. Une formulation unifiée des problèmes de gestion nous paraît indispensable pour favoriser un consensus sur la nature des problèmes.

La spécification du système d'information oblige à établir un **noyau consistant de connaissances sur le système.** Ce noyau est minimal pour l'ensemble des systèmes de production discrets et peut être complété pour représenter les systèmes de production réels.

Le langage de production est également un **langage algorithmique** car il permet de décrire par exemple une méthode d'ordonnancement d'un atelier. Ce langage utilise des types de données et des opérateurs spécifiés et permet la vérification sémantique des algorithmes écrits.

Du point de vue des applications industrielles, le langage de production est un **outil de rédaction du cahier des charges** d'un système de gestion nouveau. Il peut également être utilisé comme **langage d'interface** pour des bases de données production.

Le papier...

... fait suite à une publication [HILG88] qui présente une première méthode de conception d'un langage de description des systèmes de production du point de vue de la gestion. Les phrases de ce langage sont générées par une grammaire context-free qui est dérivée d'un schéma entités-associations [CHEN76] du système d'information du court terme. Cette méthode permet de générer facilement un ensemble de phrases non redondantes qui expriment les liens entre informations représentées dans le schéma des données. Cependant la méthode est limitée à la définition syntaxique d'un langage. La sémantique des contraintes sur les entités et leurs associations doit être programmée dans un analyseur sématique qui s'exécute en séquence avec l'analyseur lexical-syntaxique. De plus la méthode ne prévoit pas la spécification d'opérations de calcul propre à l'univers du court terme.

Dans ce papier, nous présentons une approche reposant sur la spécification algébrique du système d'information du court terme qui nous permet de tenir compte des insuffisances de l'approche précédente.

En introduisant les spécifications algébriques nous nous donnons des bases mathématiques pour la définition syntaxique et sémantique d'un langage. Les premiers travaux sur les spécifications algébriques sont dûs à [LISK74], [LISK75] et au groupe ADJ [GOGU76] et [GOGU77a]. Ils établissent une méthode de spécification formelle de types de données en vue de la construction de programmes corrects. Liés aux techniques de réécriture [KNUTH70] pour la preuve automatique de propriétés d'une spécification axiomatique, ils fournissent des bases solides pour la conception d'un langage de production. Dans ce papier nous présentons une synthèse des résultats des domaines ci-dessus appliqués à la conception d'un langage de production pour la gestion du court terme.

Dans une première section, nous posons les fondements théoriques, qui nous permettent de construire une spécification algébrique du court terme. Nous définissons les propriétés souhaitées pour une spécification et nous montrons le calcul de cohérence sémantique de descriptions avec une implémentation d'un langage du court terme.

Dans une deuxième section, nous présentons une méthode de preuve de consistance d'une spécification équationnelle et de démonstration automatique de propriétés de types d'information. Nous expliquons également le principe d'un enrichissement consistant d'une spécification qui permet d'assurer l'ouverture requise du langage.

Dans une troisième section nous définissons une grammaire de la spécification du court terme. Cette grammaire permet de définir un interface avec un langage utilisateur quelconque.

Finalement nous évaluons les résultats obtenus dans une conclusion et nous indiquons la suite des travaux à mener.

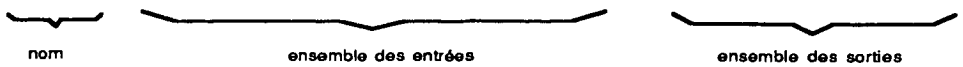
0. Préliminaires

Avant d'entrer dans la présentation formelle du langage de production, nous voulons donner au lecteur un aperçu rapide des problèmes posés par la construction du langage à partir d'un exemple simple.

Dans le système de gestion que nous voulons décrire nous rencontrons des opérations. Une opération transforme des références (matières premières ou en-cours) en entrée de l'opération en des références (en-cours ou produits finis) en sortie de l'opération.

Soit 'ope-a' le nom d'une opération qui utilise 3 unités de la référence 'ref-a' et 9 unités de la référence 'ref-b' et les transforme en 13 unités de la référence 'ref-c'. Cette description peut s'écrire :

OP[ope-a /{ << ref-a ; 3 >> | <<ref-b ; 9 >> | {} }/{ <<ref-c ; 13 >> | {} }]



Plusieurs types d'information interviennent dans cette description :

- les chaînes de caractères, qui identifient l'opération ('ope-a') et les références ('ref-a', ...)
- les entiers, qui designent les quantités (3, 9, 13),
- les couples qui associent une référence et un entier (<< ref-a ; 3 >>, ...) ,
- les ensembles d'entrée ou de sortie dont les éléments sont des couples
- et finalement l'opération qui réunit une chaîne de caractères et deux ensembles.

Lors de la description d'une opération, on aimerait vérifier un certain nombre de contraintes :

[1] les quantités associées aux références sont toujours positives ;
[2] une même référence n'intervient pas plus d'une fois dans un ensemble de couples ;
[3] les ensembles des entrées et des sorties ne sont pas vides ;
[4] comme l'opération est une transformation de références, une référence en entrée de l'opération ne doit pas intervenir dans l'ensemble des sorties.

L'idée que nous poursuivons dans ces pages est de ***spécifier algébriquement*** les types d'information du système de gestion et leurs contraintes (cf. secton I). A chaque type d'information nous donnons un nom et nous définissons des opérateurs qui sont spécifiés par des équations.

Par exemple :

Appelons le type ensemble *Ens*, le type de ses éléments *Elt* et le type des booléens *Booléen*.
{} est un opérateur constant qui repésente l'ensemble vide ({} : --> *Ens*).
L'opérateur ! ajoute un élément à un ensemble (_ ! _ : *Elt Ens* --> *Ens*).
L'opérateur ∈ teste l'appartenance d'un élément à un ensemble (_ ∈ _ : *Elt Ens* --> *Booléen*).

L'action de ces opérateurs est spécifiée par des équations dans lequelles X et Y représentent des

éléments (*Elt*), S un ensemble (*Ens*) et vrai et faux les constantes booléennes.

X n'appartient pas à l'ensemble vide :

$X \in \{\} = \text{faux}.$

De façon récursive nous définissons l'appartenance à un ensemble non vide :

$X \in (Y \mid S) = \text{si } (X = Y) \text{ alors vrai sinon } X \in S.$

On ajoute X à un ensemble si X n'appartient pas à l'ensemble :

$X \mid S = \text{si } X \in S \text{ alors } S \text{ sinon } X\ S.$

Dans notre phrase, on observe que :

<< ref-a ; 3 >> | << ref-b ; 9 >> | {} (1)

et **<< ref-c ; 13 >> | {}** (2)

repésentent des expressions de calcul formées avec des opérateurs de *Ens* et où les éléments sont des couples.

Le type d'information *Opération* est composé d'une chaîne de caractères (*Chaine*) et de deux ensembles (*Ens*). L'opérateur qui construit ce type peut s'écrire :

OP [_ /{ _ }/{ _ }] : *Chaine Ens Ens --> Opération.*

On reconnaît dans la structure de cet opérateur la structure de notre phrase.

La sémantique des contraintes auxquelles nous soumettons les descriptions d'opérations peut aisément s'écrire sous forme d'équations.

Par exemple, la contrainte [3] sur OP[c /{ S_1 }/{ S_2 }] exprime que $S_1 \neq \{\}$ et $S_2 \neq \{\}$. Ceci implique que

OP[ope-b /{ << ref-a ; 3 >> | << ref-b ; 9 >> | {} }/{ {} }]

n'est pas une description cohérente avec les contraintes. Cette ***cohérence*** est ***calculable*** par application des équations aux expressions d'opérateurs que sont les phrases (cf. sous-section I.3.).

Donnons nous deux opérateurs qui sélectionnent l'ensemble des entrées et l'ensemble des sorties d'une opération dans le type *Opération* :

entrée _ : *Opération* --> *Ens*
et sortie _ : *Opération* --> *Ens* .

Les équations qui définissent l'action des sélecteurs sont :

entrée(OP[c /{ S_1 }/{ S_2 }]) = S_1
et sortie(OP[c /{ S_1 }/{ S_2 }]) = S_2 .

On voit que les opérateurs entrée et sortie appliqués à notre phrase selectionnent les expressions (1) et (2), respectivement.

Nous pouvons vérifier la contrainte [3] en appliquant un opérateur à réponse booléenne à notre phrase :

correct3 _ : *Opération* --> *Booléen.*

L'action de cet opérateur est définie par une équation où O représente un type *Opération:*

correct3 O = (entrée O ≠ {}) et (sortie O ≠ {}).

Nous pouvons alors calculer :

correct3(OP[ope-b /{ << ref-a ; 3 >> | << ref-b ; 9 >> | {} }/{ {} }])
= ((<< ref-a ; 3 >> | << ref-b ; 9 >> | {}) ≠ {}) et ({} ≠ {})
= vrai et faux
= faux

Bien sûr, pour être en droit d'effectuer ces calcul nous devons spécifier tous les types d'information (*Booléen, Chaine*, ...) et leurs opérateurs (et, ≠, ...) qui interviennent dans les expressions.

Calculer la cohérence d'une description d'opération pour les quatre contraintes de notre sémantique revient à appliquer à une phrase un opérateur correct _ : *Opération* --> *Booléen* défini par

correct O = correct1 O et correct2 O et correct3 O et correct4 O

où correct1, correct2 , correct3, correct4 sont les évaluations booléennes des contraintes [1], [2], [3] et [4].

Cependant le calcul de cohérence n'est significatif que lorsque la ***spécification*** elle-même est ***consistante***, ou encore si, au cours des écritures d'équations, on n'a pas ajouté une équation qui contredit les précédentes. Pour vérifier cela on prouve que l'équation vrai = faux n'est pas un théorème de la théorie construite sur les équations de la spécification. (cf sous-section I.2 et section II)

Supposons maintenant que nous ayons su définir une spécification consistante et que nous puissions calculer la cohérence des phrases qui décrivent des opérations. Nous complétons ce langage afin de pouvoir décrire des gammes. Une gamme se définit comme une liste ordonnée d'opérations. En partant de la spécification précédente nous ajoutons le type liste ordonnée, le type gamme ainsi que les contraintes qui portent sur les gammes.

Par exemple, nous voulons décrire des gammes pour lesquelles

[5] la $(i+1)^{ème}$ opération de la liste des opérations d'une gamme possède en entrée au moins une référence qui est sortie de la $i^{ème}$ opération.

La phrase suivante décrit une gamme appelée 'gam-b' qui est une liste de trois opérations 'ope-c', 'ope-d' et 'ope-e'.

```
GA[gam-b // $
  ~ OP[ ope-c /{<< ref-a ; 2 >> | << ref-b ; 3 >> | {} }/{ << ref-c ; 3 >> | {} }]
  ~ OP[ ope-d /{<< ref-d ; 1 >> | << ref-c ; 3>> | << ref-i ; 1 >> | {} }
              /{ << ref-e ; 1 >> | << ref-h ; 2 >> | {} }                    ]
  ~ OP[ ope-e /{<< ref-c ; 10 >> | << ref-e ; 1 >> | {} }{ << ref-g ; 4 >> | {} }]
  ]
```

Lors de la construction progressive d'une spécification par de nouveaux types d'information et leurs contraintes il est indispensable de s'assurer que les anciennes spécifications ne sont pas altérées. En d'autres termes, il faut prouver la ***consistance de l'extension d'une spécification*** consistante. Pour cela on verifie que la théorie de la nouvelle spécification restreinte aux types d'information et opérateurs connus par l'ancienne spécification ne comporte pas un théorème vrai = faux (cf. section II.2.).

Cependant, même si la spécification obtenue est consistante, il est utile de vérifier la signification de notre spécification par la ***preuve des propriétés implicites*** des types d'information spécifiés.

Par exemple en combinant les contraintes [3], [4] et [5] nous trouvons une propriété des

gammes qui n'a pas été explicitée dans la formulation de nos contraintes : une gamme ne peut avoir dans sa liste deux opérations consécutives qui sont identiques.

Le but de ce papier est non seulement d'établir les preuves de consistance et les calculs de cohérence que nous requérons, mais encore de les rendre automatiques et exécutables dans un contexte réel. Pour cela nous empruntons aux domaines à la fois mathématiques et informatiques des spécifications algébriques et de la réécriture.

Le langage que nous obtenons est un langage de expressions cohérentes d'une spécification consistante. La structure syntaxique des opérateurs de cette spécification peut être redéfinie sous la forme classique de règles de production d'une grammaire. La grammaire du langage des expressions facilite la définition d'un interface standard avec un langage propre à un utilisateur industriel. Dans le contexte d'une application industrielle, l'utilisateur, qui rédige le cahier des charges d'un système de gestion, veut disposer d'un langage facile à écrire et à comprendre. Ce langage utilisateur peut être un langage graphique qui se sert d'icônes pour représenter les types d'information du système, ou encore un ensemble de formulaires normalisés. Le langage des expressions d'opérateurs que nous proposons devient alors trop complexe. Pour permettre à l'utilisateur de rédiger dans le langage qui lui convient, nous devons disposer d'un traducteur de phrases du langage de l'utilisateur dans des phrases du langage de la spécification. Sans préjuger de la forme du langage de l'utilisateur nous pouvons définir les caractéristiques d'un interface qui effectue cette traduction.

I. Spécification algébrique du système d'information

Le système d'information de la gestion du court terme est spécifié algébriquement (I.1.). Cette spécification doit être consistante (non contradictoire) et suffisamment complète pour décrire toutes les propriétés du système (I.2.) à l'aide des informations qui le définissent. Les phrases du langage doivent être cohérentes avec la spécification du système qu'elles décrivent ou commandent (I.3.).

Les notions de base de la spécification algébrique sont présentées avec un formalisme unifié et accompagné d'exemples illustratifs. Nous nous référons pour ce travail de synthèse à [ERIG85], [GOGU76], [GOGU77a], [GOGU79b], [GOGU86], [THIER85], [KOUN85b], [LISK75], [THAT77], [BERG82], [DERN79], [BROY82], [BROY85], [TOMP80], [BURS79], [BURS82], [REIC87], [BEIE86], [WIRS83] et [ZILL74].

I.1. Spécification algébrique et sémantique

Le système d'information est composé de types d'information. Un type d'information est spécifié algébriquement par la donnée de son nom et de la syntaxe des opérateurs qui créent ou modifient le type, sélectionnent ou calculent des informations du type (I.1.1.). L'action des opérateurs est spécifiée par un ensemble d'équations qui sont des égalités entre expressions formées par les opérateurs sur le type (I.1.2.). Une spécification est une algèbre. La sémantique de la spécification est définie par une classe d'algèbres de même structure (I.1.3.). Dans la suite nous appelons les types d'information des sortes.

I.1.1. Signature, algèbre et termes

Définition 1 Une **signature** $\Sigma = (S, OP)$ est composée d'un ensemble S dont les éléments $s \in S$ sont appelés des **sortes** et une famille $OP = < OP_{w,s} \mid w \in S^*, s \in S >$ d'ensemble $OP_{w,s}$ de symboles d'opérations. S^*représente l'ensemble des suites finies de sortes. $N \in OP_{w,s}$ est un opérateur d'arité w, de sorte s et de profil <w, s>. On écrit $N : s_1 \dots s_n \text{ --> } s$ avec $w = s_1 \dots s_n$. Si N est d'arité 0 alors w est le mot vide λ et on dit que $N \in OP_{\lambda,s}$ est une constante. ❑

Une signature des entiers naturels $\Sigma_{nat} = (S, OP)$ est composée de S = {nat} et OP = {0: -->nat, +1: nat -->nat, +: nat nat -->nat}. Une signature des booléens $\Sigma_{bool} = (S, OP)$

est composée de S = {bool} et OP = { true : --> bool, false : --> bool, not : bool --> bool, and : bool bool --> bool }.

Définition 2 On appelle **Σ-algèbre A** une algèbre $A = (S_A, OP_A)$ de signature Σ. Elle est donnée par une famille $S_A = <A_s \mid s \in S>$ d'ensembles A_s appelés supports et une famille $OP_A = <(N_A) \mid N \in OP_{w,s}, s \in S>$ d'ensembles (N_A) de fonctions $N_A: A_w \text{ --> } A_s$ pour $N \in OP_{w,s}$ et $s \in S$. ❑

L'algèbre BOOL = (𝔹, true, false, not, and) où 𝔹 = {true, false} est le support et true, false, not et and sont les opérations de BOOL. L'algèbre NAT = (ℕ, 0, +1, +) où ℕ est le support. 0, +1 et + sont les opérations de NAT.

Définition 3 Soit $X = <X_s \mid s \in S>$ une famille d'ensembles X_s disjoints de **variables** $x \in X_s$ de sorte $s \in S$. Un **Σ-terme** est une séquence de symboles de Σ et de variables de X. L'ensemble $T_{OP,s}(X)$ des **Σ-termes** de sorte $s \in S$ est défini par

(i) $X_s \cup OP_{\lambda,s} \subset T_{OP,s}(X)$ et

(ii) $N(t_1, ..., t_n) \in T_{OP,s}(X)$ pour $N \in OP_{w,s}$ avec $N: s_1 ... s_n \text{ --> } s$ et $t_1 \in T_{OP,s1}(X), ..., t_n \in T_{OP,sn}(X)$.

On note $T_{OP}(X) = \cup_{s \in S} T_{OP,s}(X)$ l'ensemble des Σ-termes. Des Σ-termes sans variables sont appelés des **Σ-termes clos** et on note $T_{OP,s} = T_{OP,s}(\phi)$ l'ensemble des Σ-termes clos de sorte s. L'ensemble des Σ-termes clos est noté $T_{OP} = \cup_{s \in S} T_{OP,s}$. ❑

Soit Σ = (S, OP) une signature avec S = {nat} et OP = {0 : --> nat ; succ : nat --> nat} et l'ensemble des variables $X_{nat} = \{n\}$. Les ensembles de Σ -termes sont définis par

$T_{nat} = \{succ^k(0) / k \geq 0\}$ et $T_{nat}(X) = T_{nat} \cup \{succ^k(n) / k \geq 0\}$ avec $succ^0(t) = t$, $succ^{k+1}(t) = succ(succ^k(t))$ et $X = \{n\}$.

Définition 4 La Σ-algèbre dont T_{OP} est le support est appelée **algèbre des termes** que l'on note T_Σ. Elle est définie par une famille $<T_{OP,s} \mid s \in S>$ d'ensembles de Σ-termes clos de sorte s et une famille $<(T_{OP,N}) \mid N \in OP_{w,s}, w \in S^*$ et $s \in S>$ d'ensembles $(T_{OP,N})$ d'opérateurs $T_{OP,N}: t_1, ..., t_n \text{ --> } t_{n+1}$ avec $t_i \in T_{OP,si}$ pour i = 1, ..., n+1. **L'algèbre libre des termes** est définie par $T_\Sigma(X) = (<T_{OP,s}(X) \mid s \in S>, <(T_{OP,N}(X)) \mid N \in OP_{w,s}, w \in S^*$ et $s \in S>)$. ❑

Remarquons que l'ensemble des termes clos T_{OP} est le support d'une algèbre dite initiale, l'ensemble des termes non clos $T_{OP}(X)$ est le support d'une algèbre dite libre.

I.1.2. Evaluation, spécification et algèbre d'une spécification

On peut évaluer des Σ-termes dans une algèbre (c'est-à-dire faire correspondre à un Σ-terme $t \in T_{OP,s}(X)$ un élément du support A_s de l'algèbre). L'évaluations de termes permet la vérification de la validité d'équations (formées par des termes) dans une algèbre.

Définition 5 Soit T_{OP} l'ensemble des Σ-termes clos et A une Σ-algèbre.

L'**évaluation** eval : T_{OP} --> A est définie récursivement par

(i) $eval(N) = N_A$ pour $N \in OP_{\lambda,s}$ et

(ii) $eval(N(t_1, ..., t_n)) = N_A(eval(t_1), ..., eval(t_n))$ pour $N(t_1, ..., t_n) \in T_{OP}$ et $N \in OP_{w,s}$.

Soit $T_{OP}(X)$ l'ensemble des Σ-termes, A une Σ-algèbre et X un ensemble de variables. L'**affectation** aff : X --> A avec $aff(x) \in A_s$ pour $x \in X_s$ et $s \in S$ associe une valeur d'un support à une variable. L'affectation étendue à des termes aff@ : $T_{OP}(X)$ --> A est définie récursivement par

(i) $aff@(x) = aff(x)$ pour $x \in X$ et $aff@(N) = N_A$ pour $N \in OP_{\lambda,s}$

(ii) $aff@(N(t_1, ..., t_n)) = N_A(aff@(t_1), ..., aff@(t_n))$ pour $N(t_1, ..., t_n) \in T_{OP}(X)$ et $N \in OP_{w,s}$.

L'affectation s'applique à des Σ-termes alors que l'évaluation s'applique uniquement à des Σ-termes clos. De plus, si $X = \phi$ alors aff@ = eval . On notera dans la suite aff@ par aff. ❑

Nous allons définir des équations, la validité d'équations dans les algèbres, des spécifications d'algèbres et l'algèbre d'une spécification, pour finalement aboutir à la notion de sémantique d'une algèbre.

Une spécification équationnelle est un ensemble d'équations qui définissent les opérateurs d'une signature Σ et par conséquent la structure sur laquelle ils agissent. Par exemple, la spécification d'une structure de pile est donné par la définition des opérateurs de création d'une pile vide, d'ajout et de retrait d'un élément de la pile. D'autres exemples de spécification de la logique booléenne (sorte bool) ou de l'addition d'entiers naturels (sorte nat) sont donnés ci-après.

Définition 6 Une **spécification équationnelle** SPEC = (S, OP, E) est constituée à partir d'une signature Σ = (S, OP) et d'un ensemble E d'équations. Une **équation** $e \in E$ est une paire de Σ-termes e = (g, d) avec $g, d \in T_{OP}(X)$. On dit qu'une équation est **valide** dans une Σ-algèbre A si pour toute affectation aff : $T_{OP}(X)$ --> A on verifie aff(g) = aff(d). L'ensemble TH(E) des équations valides pour E est appelé **théorie équationnelle**. On dit aussi qu'une Σ-algèbre A est un **modèle** si A valide toutes les équations de E. ❑

Les spécification équationnelles des sortes bool (SPECB = (S_B, OP_B, E_B)) et nat (SPECN=(S_N, OP_N, E_N)) s'écrivent :

sorte bool
opérateurs

false	:	--> bool	/* constante false	*/
true	:	--> bool	/* constante true	*/
not	: bool	--> bool	/* négation logique	*/
and	: bool bool	--> bool	/* conjonction logique	*/
equal	: bool bool	--> bool	/* égalité de booléens donne un booléen	*/

var x : bool /* x est une variable booléenne */
équations

(1) not(true) = false
(2) not(not(x)) = x
(3) and(x, true) = x
(4) and(x, false) = false
(5) equal(false, x) = not x
(6) equal(true, x) = x

sorte nat
opérateurs

0	:	--> nat	/* constante 0	*/
succ	: nat	--> nat	/* le successeur d'un entier	*/
add	: nat nat	--> nat	/* l'addition de deux entiers	*/

var x, y : nat /* x et y sont des variables entiers naturels */
équations

(1) add(x, 0) = x
(2) add(succ(x),y) = succ(add(x, y))

On peut évaluer les Σ-termes clos de la spécification de la sorte nat dans l'algèbre NAT = ($\mathbb{N}$, 0, +1, +).

On fixe eval(0) = 0, eval(succ(t)) = eval(t)+1, eval(add(t_1, t_2)) = eval(t_1)+ eval(t_2) et on évalue : eval(add(succ(0),succ(0))) = eval(succ(0))+eval (succ(0)) = (eval(0)+1) + (eval(0)+1) = (0+1) + (0+1) = 1+1= 2.

Soit X= {n, m} un ensemble de variables et aff(n) = 2 et aff(m) = 5 des affectations on obtient : aff(add(succ(n),m)) = aff(succ(n)+ aff(m)) = (aff(n)+1)+aff(m) = (2+1)+5 = 8.

Définition 7 Une algèbre A d'une spécification équationnelle SPEC = (S, OP, E) est une algèbre de signature Σ qui satisfait e pour tout e $\in$ E. On dit que A est une **SPEC-algèbre.** ❑

La sorte Nat est une algèbre NAT = ($\mathbb{N}$, 0, +1, +) car pour toute affectation aff : X --> $\mathbb{N}$ avec X = {n, m} on a : aff(add(n,0)) = aff(n) et aff(add(n, succ(m))) = aff(succ(add(n, m))).
Par définition de aff on obtient en effet :

aff(add(n, 0)) = aff(n) + aff(0) = aff(n) + 0 = aff(n) et

aff(add(n, succ(m)) = aff(n) + aff(succ(m)) = aff(n) + aff(m) + 1 = aff(succ(add(n, m)))

Définition 8 Soit E un ensemble d'équations. On dit que E **vérifie** une équation g = d et on note E ⇒ g = d si et seulement si g = d est valide dans tout modèle de E. E verifie un ensemble d'équations E' et on note E ⇒ E' si et seulement si E vérifie toutes les équations de E'. L'ensemble des équations vérifiées par E est appelé **théorie inductive** de E. ❑

La notion de théorie inductive fait référence aux preuves par récurrence et la notion de théorie équationnelle se réfère aux remplacements d'égaux par égaux.

I.1.3. Initialité, congruence et types abstraits

Nous allons définir dans cette sous-section un morphisme unique entre Σ-algèbres qui établit un préordre dans la classe des Σ-algèbres et donne l'algèbre minimale pour cette relation, unique à un isomorphisme près, que l'on appelle algèbre initiale. Nous définissons une relation de congruence qui nous permet de construire l'algèbre initiale et la classe des Σ-algèbres qui ont même structure que l'algèbre initiale : le type abstrait.

Définition 9 Soit $A = (S_A, OP_A)$ et $B = (S_B, OP_B)$ des Σ-algèbres. Un **Σ-homomorphisme** f : A --> B est une famille de fonctions $f_s : A_s \dashrightarrow B_s$ pour $s \in S$ tel que pour chaque symbole constant N : --> s dans $OP_{\lambda,s}$ et $s \in S$ on a : $f_s(N_A) = N_B$ et pour chaque symbole d'opération $N : s_1 \dots s_n \dashrightarrow s$ dans $OP_{w,s}$ et $s \in S$ et pour $a_i \in A_{si}$, pour i = 1, ..., n on a : $f_s(N_A(a_1, \dots, a_n)) = N_B(f_{s1}(a_1), \dots, f_{sn}(a_n))$. Un Σ-homomorphisme est un **Σ-isomorphisme** si et seulement si toutes les fonctions f_s sont bijectives. Si deux algèbres A et B sont isomorphes alors on note A ≈ B. ❑

Définition 10 Une Σ-algèbre A est initiale si et seulement si pour toute Σ-algèbre B il existe un Σ-homomorphisme unique de A vers B. ❑

Proposition 1 T_Σ est une algèbre initiale. Deux Σ-algèbres initiales sont isomorphes et chaque Σ-algèbre isomorphe à une Σ-algèbre initiale est initiale. ◆

Une relation de congruence est une relation d'équivalence qui permet de constituer des classes d'équivalence de termes. Ces classes d'équivalence de termes constituent le support d'une algèbre construite à partir de la relation de congruence : l'algèbre quotient. Les opérateurs de cette algèbre sont les fonctions définies sur les classes d'équivalence. Par construction l'algèbre quotient est initiale dans sa classe.

Définition 11 Une **Σ-congruence sur** une Σ-algèbre A est definie par une famille $\equiv = <\equiv_s | s \in S>$ de relations d'équivalence $\equiv_s$ sur A_s. Pour $a_i, b_i \in A_{si}$, $i = 1, ..., n$ et pour chaque $N \in OP_{w,s}$ avec $w = s_1 ... s_n$ on a : $a_i \equiv_{si} b_i$, $i = 1, ..., n$ implique $N_A(a_1, ..., a_n) \equiv_s N_A(b_1, ..., b_n)$. Soit A une Σ-algèbre avec des supports A_s pour $s \in S$ et E un ensemble d'équations. Nous appelons $\equiv_E$ une **Σ-congruence générée par E sur A.** On définit $\equiv_E$ par la famille

$\equiv_E = <\equiv_{E,s} | s \in S>$ et $t \equiv_{E,s} t'$ si $t = h(l)$ et $t' = h(r)$, pour $h : T_{OP,s}(X) \dashrightarrow A_s$ et $(l, r) \in E$.

Soit A une Σ-algèbre et R une relation de congruence sur A. L'**algèbre quotient** $Q := A_{/R}$, aussi appelée la factorisation de A par R, est définie par :

$Q = (<Q_s | s \in S>, <(N_Q) | N \in OP_{w,s}, w \in S^*, s \in S>)$ avec

1) pour tout $s \in S$ on a un ensemble de base $Q_s := \{[a] / a \in A_s\}$ pour tout $s \in S$ où [a] désigne une classe de congruence définie par $[a] = \{ b \in A_s / (a,b) \in R \}$

2) $N_Q := [N_A]$ pour tout symbole $N : \dashrightarrow s$ de $OP_{\lambda,s}$ et $s \in S$

3) $N_Q := Q_{s1} ... Q_{sn} \dashrightarrow Q_s$ pour chaque opération $N : s_1 ... s_n \dashrightarrow s$ de $OP_{w,s}$

définie par $N_Q([a_1], ..., [a_1]) = [N_A(a_1, ..., a_n)]$ et $a_i \in A_{si}$ pour $i = 1, ..., n$. ❑

Proposition 2 $T_{\Sigma/E}$ est une SPEC-algèbre initiale. $T_{\Sigma/E}$ est initiale dans sa classe et valide E. ◆

Appelons $SPECN = (S_N, OP_N, E_N)$ la spécification algébrique de la sorte nat. L'algèbre quotient obtenue par la congruence générée par E_N sur SPECN est $T_{SPECN/EN} = (Q_{nat}, 0_Q, succ_Q, add_Q)$ avec $Q_{nat} = \{[succ^n(0)] / n \geq 0\}$, $0_Q = [0]$,

$succ_Q([succ^n(0)]) = [succ^{n+1}(0)]$, $n \geq 0$,

$add_Q([succ^n(0)],[succ^m(0)]) = [succ^{n+m}(0)]$, $n,m \geq 0$.

Les éléments du support Q_{nat} de l'algèbre $T_{SPECN/EN}$ sont les classes d'équivalence sur les Σ-termes clos T_{OP} et les opérations 0_Q, $succ_Q$ et add_Q sont définis par des fonctions sur des classes d'équivalence. L'algèbre $T_{SPECN/EN} = (Q_{nat}, 0_Q, succ_Q, add_Q)$ est isomorphe aux entiers naturels NAT = ($\mathbb{N}$, 0, +1, +) puisqu'il existe f : $T_{SPECN/EN} \approx$ Nat donnée par $f([succ^n(0)]) = n$ pour $n \in \mathbb{N}$.

Le support Q_{bool} de l'algèbre quotient de la spécification SPECB comporte deux éléments [true] et [false] qui représentent les classes d'équivalence des expressions booléennes.

Définition 12 Soit SPEC = (S, OP, E) une spécification de signature Σ = (S, OP). Le **type abstrait** est la classe des Σ-algèbres validant E et isomorphes à l'algèbre initiale $T_{\Sigma/E}$. ❑

Le type abstrait définit une **sémantique** abstraite et unique à un isomorphisme près. Il est représenté par une algèbre quotient d'une relation de congruence. Cette algèbre est minimale à l'égard de l'existence des éléments du support, des opérations et des propriétés spécifiées.

I.2. Propriétés d'une spécification

Une spécification est consistante si les équations ne sont pas contradictoires ; elle est complète si on a introduit suffisamment d'équations pour décrire toutes les propriétés des sortes qu'elle comporte [WING82]. Si l'on connaît, avant d'écrire la spécification équationnelle, une sémantique voulue de la spécification (par exemple la sémantique de la logique booléenne), alors il suffit de vérifier que l'algèbre initiale de la spécification obtenue est isomorphe à la sémantique voulue. Si on ne dispose pas d'une sémantique de référence la vérification devient indécidable. Nous montrerons dans la section II une procédure de décision reposant sur l'algorithme de KNUTH-BENDIX [KNUT70] qui permet de vérifier la consistance d'une spécification dans une majorité de cas et nous donnons les conditions de complétude suffisante.

Ces propriétés ont été étudiés dans [GUTT78], [GOGU80], [HUET80b], [GOGU82], [HUET82], [KAPU86], [KIRC84], [LAZR87], [MUSS80a], [KOUA85a], [THIE84] [GOGU8284], [MOLL88] et [WIRS82].

La construction d'une spécification d'un système d'information se fait par enrichissements et extensions à partir d'une spécification primitive. Nous définissons la consistance et la complétude suffisante des spécifications construites.

Définition 13 Une spécification SPEC = (S, OP, E) est une **extension** d'une spécification primitive $SPEC_0 = (S_0, OP_0, E_0)$ si et seulement si $S_0 \subset S$, $OP_0 \subset OP$ et $E \rightarrow E_0$. Un **enrichissement** est une extension pour laquelle $S_0 = S$. Si SPEC' est un enrichissement de SPEC alors SPEC est appelé une **restriction** de SPEC'. ❑

L'enrichissement de la spécification $SPECB = (S_B, OP_B, E_B)$ par un opérateur d'implication et de disjonction logique OP'_B = {impl : bool bool --> bool, or : bool bool --> bool} et leur définition équationnelle E'_B = {(true impl b = b), (false impl b = true), (true or b = true), (false or b = b)} donne une spécification $SPECB' = (S_B, OP_B \cup OP'_B, E_B \cup E'_B)$. On dit que SPECB' est un enrichissement consistant de SPECB si la relation de congruence générée par $E_B \cup E'_B$ définit les mêmes classes d'équivalence que celles obtenues par E_B. Autrement dit l'ajout de la définition des opérateurs d'implication et de disjonction logique doit conserver les classes d'équivalence distinctes [true] et [false].

Définition 14 Une **spécification hiérarchique** $SPECH = (S \cup \{T\}, OP \cup OP_T, E \cup E_T)$ est définie comme une **extension** d'une **spécification primitive** SPEC = (S, OP, E) appelée son **environnement** par le triplet $(\{T\}, OP_T, E_T)$ qui est la spécification de la sorte T. ❑

Une spécification hiérarchique est consistante vis-à-vis de son environnement si la relation d'équivalence sur les termes des sortes de son environnement n'est pas modifiée.

Définition 15 Une spécification SPEC = (S, OP, E), extension de la spécification $SPEC_0 = (S_0, OP_0, E_0)$, est **consistante** par rapport à $SPEC_0$ si et seulement si la restriction de $\equiv_E$ aux Σ_0-termes coïncide avec $\equiv_{E_0}$, alors on a pour tout $t_o, t'_o \in T_{OPo}$, $t_o \equiv_E t'_o \Rightarrow t_o \equiv_{E_0} t'_o$. ❑

L'extension de la spécification $SPECB = (S_B, OP_B, E_B)$ par $SPECN = (S_N, OP_N, E_N)$ est consistante car la spécification étendue est l'union de deux spécifications disjointes. Par contre si on étend SPECB par $SPECN' = (S'_N, OP'_N, E'_N)$ où

$S'_N = S_N \cup \{bool\}$

$OP'_N = OP_N \cup \{ \leq : nat\ nat \text{ --> } bool, eqn : nat\ nat \text{ --> } bool \}$

$E'_N = E_N \cup \{ (0 \leq n = true), (succ(n) \leq n = false), (succ(m) \leq succ(n) = m \leq n),$
$(eqn(m, n) = (m \leq n \text{ and } n \leq m))\}$

l'union n'est plus disjointe. On vérifie que la relation d'équivalence sur l'ensemble des Σ_B-termes n'est pas modifiée.

Une spécification hiérarchique est suffisamment complète vis-à-vis de son environnement si tout terme de sorte appartenant à l'environnement est équivalent à un terme de l'environnement.

Définition 16 Une spécification SPEC = (S, OP, E), extension de la spécification $SPEC_0 = (S_0, OP_0, E_0)$ est **suffisamment complète** vis-à-vis de SPEC si et seulement si pour tout $t \in T_{OP,so}$ il existe $t_o \in T_{OPo,so}$ tel que $t_o \equiv_E t$. ❑

Définition 17 Soit $SPEC_0 = (S_0, OP_0, E_0)$ une spécification primitive. Si une spécification enrichie SPEC = (S_0, OP, E) et $OP_0 \subset OP$ et $E_0 \subset E$ est **consistante et suffisamment complète** par rapport à $SPEC_0$ alors $T_{SPEC/E} \approx T_{SPECo/Eo}$. ❑

Supposons que SPECB soit la spécification primitive. L'algèbre quotient $T_{SPECB/EB}$ définit une sémantique initiale qui est celle de la logique booléenne. Une spécification enrichie est suffisamment complète et consistante par rapport à SPECB si son algèbre quotient conserve la même sémantique.

La spécification d'un système d'information complexe comporte un certain nombre de structures d'information paramétrées [THIER85], [EHRI79] comme par exemple la structure d'ensemble ou de suite ordonnée. La structure paramétrée est définie indépendamment des éléments qui la composent. Ces éléments quelconques sont des paramètres formels auxquels on substitue des paramètres effectifs. La sorte paramètre effectif peut ainsi devenir au gré des substitutions une suite de caractères (le paramètre effectif est la sorte "caractère") ou une suite d'opérations qui constituent la gamme d'un produit (le paramètre effectif est de sorte "opération").

La propriété demandée à une spécification paramétrée est la protection du paramètre effectif, c'est-à-dire la consistance et la suffisante complétude de la spécification paramétrée, vis-à-vis de son paramètre effectif.

Définition 18 Soient SPEC = (S, OP, E), une spécification appélée **environnement** et $SPECP = (S_P, OP_P, E_P)$ une spécification appelée **paramètre formel** pour SPEC. On appelle **spécification paramétrée** par S_P la spécification définie par SPECTP = SPEC + SPECP + $(\{TP(P)\}, OP_{TP}, E_{TP})$ telle qu'il existe $c : s_1 \ldots s_i \ldots s_n \dashrightarrow s_{TP}$, $c \in OP_{TP}$ et $s_i \in S_P$. ❑

Une spécification environnement SPEC = (S, OP, E) est l'extension de la spécification SPECB' par SPECN' (cf exemples des définitions 13 et 15).

Une spécification paramétrée (liste(P), OP_{liste}, E_{liste}) est donnée par

sorte liste(P)
opérateurs

$	:	--> liste(P)	/* liste vide	*/
~	: liste(P) P	--> liste(P)	/* ajout d'un élément P à la liste	*/
dernier	: liste(P)	--> P	/* dernier élément d'une liste	*/
reste	: liste(P)	--> liste(P)	/* liste moins son dernier élément	*/

var l : liste(P), p : P
équations

(1) dernier($) = indefini
(2) dernier(l ~ p) = p
(3) reste(l ~ p) = l
(4) reste($) = $

La spécification paramètre formel SPECP = ((P), OP_P, E_P) est donnée par

sorte P
opérateurs

indefini	:	-->P	/* constante indefinie	*/
egal	: P P	-->bool	/* égalité	*/

var p p' p" : P
équations

(1) egal(p, p) = true
(2) egal(p, p') = egal(p', p)
(3) egal(p, p') and egal(p', p") impl egal(p, p") = true

SPECTP = SPEC + ((P), OP_P, E_P) + ({liste(P)}, OP_{liste}, E_{liste})

Définition 19 Une spécification SPECF = (S_F, OP_F, E_F) sur l'environnement SPEC est un **paramètre effectif admissible** de SPECTP s'il existe un morphisme de passage de paramètres noté i : SPEC + SPECP --> SPEC + SPECF qui est un morphisme d'identité sur SPEC.

La spécification SPECTF obtenue par instanciation de SPECP par le paramètre admissible SPECE s'écrit : SPECTF = SPEC + SPECF + ({$TP(S_F)$}, OP_{TF}, E_{TF})

$OP_{TF} = OP_{TP}[i(P) \text{ --> } P$, pour tout $P \in S_P]$

$E_{TF} = E_{TP}[X_{iP} \text{ --> } X_P]$ [i(N) --> N, pour tout $N \in OP_P$] ❑

SPECN' est un paramètre effectif admissible pour SPECTP.

Une spécification paramétrée est une extension de la spécification paramètre ; elle doit être consistante et suffisamment complète vis-à-vis de la spécification paramètre.

I.3. Spécification du système d'information, cohérence d'une description et calculs algorithmiques

Nous donnons dans cette section une implémentation d'une spécification du système de gestion du court terme (I.3.1) et nous montrons le calcul de la cohérence d'un texte descriptif ou de commande avec la spécification (I.3.2).Nous indiquons également comment nous pouvons étendre notre approche pour obtenir un langage algorithmique dédié à la gestion de production (I.3.3.).

I.3.1. Spécification du système d'information de la gestion du court terme et implémentation

Une spécification algébrique des sortes ne peut se faire de façon indépendante pour chaque sorte. Certaines sortes sont composées de sortes plus élémentaires. Afin de se donner une méthode progressive de spécification des sortes du système d'information, nous avons delimité quatre univers d'informations. Chaque univers correspond à une collection de types de plus en plus complète à l'égard de la gestion du court terme.

L'univers I donne des sortes de base : les booléens, les entiers, des identificateurs de référence (produits finis, en-cours ou de matières premières), d'opérations, de gammes, la sorte ensemble et liste ordonnée. Nous définissons dans cet univers la gamme comme une liste ordonnée d'opérations qui, elles-mêmes, sont constituées d'un ensemble de références en entrée et en sortie de l'opération.

L'univers II donne la définition des stocks, des unités de transport, des machines, de leurs outils et organes d'assistance. Nous définissons également la sorte individu caractérisée par des

compétences et des périodes de présence.

L'univers III donne la définition d'une sorte paramétrée définissant les opérations de montage, de démontage et les réglages, ainsi que la sorte EXECUTION qui contient l'ensemble des données nécessaire pour l'exécution d'une opération sur un type de machine.

L'univers IV donne la définition des distances et des transports entre machines et stocks. La sorte EX_GAMME spécifie l'ensemble des données pour l'exécution de la gamme.

Nous montrons une implémentation de la spécification de l'univers I avec OBJ3, les spécifications des univers II, III, IV sont ajoutées en annexe.

OBJ3[1] est un langage de spécification exécutable basé sur les algèbres ordo-sortées[2] et la logique équationnelle. Les principes d'OBJ3 sont développés dans [FUTA85], [GNAE87], [GOGU78b], [GOGU79a], [GOGU86a], [KIRC85a], [KIRC85b]. Nous ne donnons ici que les explications indispensables pour la lecture des spécifications écrites dans OBJ3.

OBJ3 est un langage modulaire. Nous utilisons trois types de modules:

- la spécification (définition 6) définie par
  obj ***NOM*** **is** ***CORPS*** **jbo,**
- la spécification paramétrée (déf. 18) définie par
  obj ***NOM [param :: NOM-THEORIE]*** **is** ***CORPS*** **jbo,**
- la spécification du paramètre formel (définition 18 et 19) ou théorie définie par
  th ***NOM-THEORIE*** **is** ***CORPS*** **endth .**

NOM et ***NOM-THEORIE*** sont des noms alphanumeriques de module.

CORPS contient la déclaration
du nom de la sorte spécifiée (**sort**),
des sous-sortes (**subsorts**) ou ordre d'inclusion de sortes,
des modules importés (**protecting**)
des opérateurs (**op** N : w -> s),
des variables (**var**) et
des équations (**eq** terme = terme) de la spécification.

---

[1] D'autres logiciels ayant des possibilités semblables sont décrits dans [MUSS80b] pour AFFIRM, [GOGU80a] et [BURS81] pour CLEAR, [SMOL87b] pour TEL, [GOGU85b] pour EQLOG et [BERT88] pour CEC.

[2] Les algèbres ordo-sortées ([SMOL87a], [SMOL87c], [KIRK85a], [KIRK85b], [GOGU85a]) acceptent une relation d'ordre sur les sortes que l'on peut interpréter comme un ordre d'inclusion. Par exemple, les entiers naturels sont une sous-sorte (subsort) des entiers. Les opérateurs d'addition, de multiplication (...) sont alors dits surchargés parce qu'ils s'appliquent à la sorte et à la sous-sorte.

Les trois premiers modules sont contenus dans OBJ3. Nous les reproduisons ici parce que les modules de notre spécification les importent et utilisent leurs opérateurs et équations. La sorte Universal designe toutes les sortes. La théorie TRIV est une théorie sur un paramètre formel quelconque. La spécification BOOL comporte la spécification de l'égalite et de l'inégalité entre termes réduits de sorte Universal. On n'indique que la syntaxe des opérateurs ; la spécification elle-même est implantée dans un module LISP.

```
obj UNIVERSAL is
 sort Universal .
jbo
```

```
th TRIV is
 sort Elt .
endth
```

```
obj BOOL is
 protecting UNIVERSAL .
 sort Bool .

 op true : -> Bool .
 op false : -> Bool .
 op _and_ : Bool Bool -> Bool .
 op _or_ : Bool Bool -> Bool .
 op not_ : Bool -> Bool .
 op not_ : Bool -> Bool .
 op if_then_else_fi : Bool Universal Universal -> Bool .
 op _==_ : Universal Universal -> Bool .        ** égalite entre termes reduits
 op _=/=_ : Universal Universal -> Bool .       ** inégalite entre termes reduits

 vars X_U Y_U : Universal .
 vars A B : Bool .

 eq if true then X_U else Y_U = X_U .
 eq if false then X_U else Y_U = Y_U .
 eq X_U == Y_U =
 eq X_U =/= Y_U =
 eq not A = (A == false).
 eq A and B = if A == true then B else false fi .
 eq A or B = if A == false then B else true fi .
jbo
```

Le module NAT contient la spécification des entiers naturels. Il importe le module BOOL pour la définition de l'opérateur relationnel <.

```
obj NAT is
 protecting BOOL .
 sort Nat .
 op 0 : -> Nat .
 op s_ : Nat -> Nat .
 op _+_ : Nat Nat -> Nat .
 op _<_ : Nat Nat -> Bool .

 vars x y : Nat .

 eq x + 0 = x .
 eq s(x) + y = s(x + y) .
 eq x < x = false .
 eq x < 0 = false .
 eq s(x) < x = false .
 eq x < s(x) = true .
jbo
```

Les identificateurs de références (Idref), d'opération (Idop) et de gammes (Idgam) sont des opérateurs d'arité 0.

```
obj IDREF is
 sort Idref .
 op ref-a : -> Idref .
 op ref-b : -> Idref .
 op ref-c : -> Idref .
 op ref-d : -> Idref .
 op ref-e : -> Idref .
 op ref-f : -> Idref .
 op ref-g : -> Idref .
 op ref-h : -> Idref .
 op ref-i : -> Idref .
 op ref-j : -> Idref .
 op ref-k : -> Idref .
 op ref-l : -> Idref .
jbo
```

```
obj IDOPE is
 sort Idope .
 op op-a : -> Idope .
 op op-b : -> Idope .
 op op-c : -> Idope .
 op op-d : -> Idope .
 op op-e : -> Idope .
 op op-f : -> Idope .
 op op-g : -> Idope .
 op op-h : -> Idope .
 op op-i : -> Idope .
 op op-j : -> Idope .
 op op-k : -> Idope .
 op op-l : -> Idope .
jbo
```

```
obj IDGAM is
 sort Idgam .
 op gam-a : -> Idgam .
 op gam-b : -> Idgam .
 op gam-c : -> Idgam .
 op gam-c : -> Idgam .
 op gam-e : -> Idgam .
 op gam-f : -> Idgam .
jbo
```

Une référence (Ref) est un couple formé d'un identificateur et d'une quantité.

```
obj REF is
  protecting IDREF .
  protecting NAT .

  sort Ref .

  op <<_;_>> : Idref Nat -> Ref .
  op nom_ : Ref -> Idref .
  op qté_ : Ref -> Nat .

  var I : Nat .
  var Id : Idref .

  eq  nom<<Id ; I>> = Id .
  eq  qté<<Id ; I>> = Id .
jbo
```

La sorte ensemble (Set) est une sorte paramétrée par X, un paramètre dont la théorie TRIV est la spécification d'un paramètre formel qui ne comporte pas d'opérateurs ni d'équations propres.

```
obj SET[X :: TRIV] is
  protecting BOOL .

  sort Set .

  op { } : ->Set .
  op _in_ : Elt Set ->Bool .
  op adj__ : Elt Set -> Set .
  op __ : Elt Set -> Set .

  vars X Y : Elt .
  var S: Set .

  eq  X in { } = false .
  eq  X in adj Y S = if X == Y then true else X in S fi .
  eq  adj X S = if X in S then S else X S fi .
jbo
```

Une opération (Ope) est composée d'un identificateur et de deux ensembles de référence : l'ensemble des référence en entrée d'une opération et l'ensemble des références en sortie de l'opération.

```
obj OPERATION is
 protecting NAT .
 protecting IDOPE .
 protecting SET[REF] .

 sort Ope .

 op OP[_/{_}/{_}] : Idope Set Set -> Ope .
 op  nom-ope_ : Ope -> Idope .
 op  entree_ : Ope -> Set .
 op  sortie_ : Ope -> Set .
 op _|_ : Ref Set ->Set .
 op _dans_ : Idref Set -> Bool .
 op _utilise_ : Ope Idref -> Bool .
 op _fabrique_ : Ope Idref -> Bool .

 var O : Ope .  var Id : Idref .
 var Q : Nat .  var C : Idope .
 vars R R' : Ref .  vars in out S : Set .

 eq  entree OP[C/{in}/{out}] = in .
 eq  sortie OP[C/{in}/{out}] = out .
 eq  nom-ope OP[C/{in}/{out}] = C .
 eq  O utilise Id = Id dans entree O .
 eq  O fabrique Id = Id dans sortie O .
 eq  Id dans (adj R S) = if Id == nom R then true else Id dans S fi .
 eq  Id dans {} = false .
 eq  R' | (adj R S ) =  if(qté R < 1)
                          then S
                          else  if nom R' == nom R
                                  then adj << nom R ; ((qté R') + (qté R)) >> S
                                  else  adj R ( R' | S)
                          fi
                       fi .
 eq  R | {} = adj R {} .
jbo
```

Une liste ordonnée (Liste) est une sorte paramétrée. La théorie sur le paramètre ne comporte ni opérateurs ni d'équations.

```
obj LISTE[X :: TRIV] is
 protecting BOOL .
 protecting NAT .

 sorts Liste-non-vide Liste .
 subsorts Liste-non-vide < Liste .

 op $ : -> Liste .
 op _~_ : Liste Elt -> Liste-non-vide .
 op reste_ : Liste-non-vide -> Liste .
 op dernier_ : Liste-non-vide -> Elt .
 op card_ : Liste -> Nat .

 var L : Liste . var Lnv : Liste-non-vide . var E : Elt .

 eq card $ = 0 .
 eq card L ~ E = s card L .
 eq dernier(L ~E) = E .
 eq reste (L ~ E) = L .
jbo
```

Une gamme (Gam) est composée d'un identificateur de gamme et d'une liste ordonnée d'opérations.

```
obj GAMME is
 protecting IDGAM .
 protecting LISTE[OPERATION] .

 op GA[_//_] : Idgam Liste -> Gam .
 op nomgam_ : Gam -> Idgam .
 op nbreop_ : Gam -> Nat .
 op listop_ : Gam -> Liste ;
 op _est-rattache-a_ : Idref Gam -> Bool .
 var G : Gam . var I : Nat . var Id : Idref . var C : Idgam . var L : Liste .

 eq listop GA[ C // L ] = L .
 eq nomgam GA[ C // L ] = C .
 eq nbreop GA[ C // L ] = card L .
 eq Id est-rattache-a G = dernier listop G fabrique Id .
jbo
```

On peut remarquer que est-rattache-a est un opérateur qui à partir d'une gamme et d'un identificateur calcule si une référence est sortie de la dernière opération de la gamme.

I.3.2. Cohérence d'un texte d'une description

Nous spécifions dans cette sous-section une sorte Texte qui est une suite de sortes Phrase. Les phrases sont des formules construites avec les opérateurs sur la sorte Gam et ses composantes. Chaque phrase (ou chaque description d'une gamme) doit vérifier des contraintes. Ces contraintes sont spécifiées dans le module PHRASE par des opérateurs à réponse booléenne.

[1] Pour chaque opération d'une gamme, une référence entrée d'une opération ne peut être sortie de cette même opération (op corr-lien_ : Ope -> Bool .).

[2] Pour chaque opération d'une gamme, ni l'ensemble des références en entrée de l'opération, ni l'ensemble des références en sortie de l'opération ne sont vides (op corr-vide_ : Ope -> Bool .).

[3] La suite des opérations d'une gamme est telle qu'il existe au moins une référence sortie d'une opération qui est également entrée de l'opération suivante. Autrement dit, les opérations consécutives d'une gamme ne sont pas déconnectées (op corr-lien-gam_ : Gam -> Bool .).

Chaque Phrase vérifie ces trois contraintes (op dire_ : Phrase -> Bool).

```
obj PHRASE is
 protecting GAMME ;

 sort Phrase .
 subsorts Gam < Phrase .

 op _intersect_ : Set Set -> Set .
 op corr-lien_ : Ope -> Bool .
 op corr-vide_ : Ope -> Bool .
 op corr-list-ope_ : Liste -> Bool .
 op corr-lien-gam_ : Liste -> Bool .
 op dire-gamme_ : Phrase -> Bool .
 op dire_ : Phrase -> Bool .

 var Ph : Phrase . var G : Gam .
 var O : Ope . var L : Liste .
 var Ido : Idope . var R : Ref .
 vars S Si So : Set .

 eq S intersect {} = {} .
 eq {} intersect S = {} .
 eq (adj R Si) intersect So = if (nom R) dans So) then R ! (Si intersect So) else Si intersect So fi .
 eq corr-vide OP[Id /{Si}/{So}] = (Si =/={}) and (So =/= {}) .
 eq corr-lien OP[Id /{Si}/{So}] = (Si intersect So) == {} .
 eq corr-list-ope L = if (L == $)
                        then true
                        else corr-vide dernier L and corr-lien dernier L and corr-list-ope reste L
                      fi .
 eq corr-lien-gam L ~ O =if (L == $)
                           then true
                           else (sortie dernier L intersect entree O) =/= {} and corr-lien-gam L
                         fi .
 eq dire-gamme G = corr-lien-gam listop G and corr-list-ope listop G . .
 eq dire Ph = dire-gamme Ph .
jbo
```

Un texte est une suite de phrases. La cohérence d'un texte est la vérification de toutes les contraintes sur chacune des phrases (cf. verifie Text -> Bool).

```
obj LAPON is
  protecting PHRASE .

  sort Text .

  op fin : -> Text .
  op _&_ : Phrase Text -> Text  .
  op verifie_ : Text -> Bool .

  var Ph : Phrase .
  var Tx : Text .

  eq verifie (Ph & Tx) = dire Ph and verifie Tx .
  eq verifie fin = true .
jbo
```

Calculer la cohérence d'un texte de descriptions revient à calculer le résultat booléen de l'opérateur verifie sur une opérande de sorte Text. Pour ce calcul applique les équations de la spécification de la gauche vers la droite. Ce mode de calcul s'appelle la réduction.

Dans l'exemple ci-dessous on vérifie la cohérence d'un texte qui décrit trois gammes (gam-a, gam-b et gam-c).

```
reduce in LAPON as : verifie(
    GA[gam-a // $
          ~ OP[ ope-a /{<< ref-a ; 3 >> | <<ref-b ; 9>> | {} }/{ <<ref-c ; 13>> | {} }]
          ~ OP[ ope-b /{<< ref-c ; 10 >> | <<ref-e ; 5>> | {} }/{ <<ref-g ; 18>> | {} }]]
    ) & (
    GA[gam-b // $
          ~ OP[ ope-c /{<< ref-a ; 2 >> | << ref-b ; 3 >> | {} }/{ << ref-c ; 3 >> | {} }]
          ~ OP[ ope-d /{<< ref-d ; 1 >> | << ref-c ; 3 >> | <<ref-i ; 1 >> | {} }
                                                /{ << ref-e ; 1 >> | << ref-h ; 2 >> | {} }]
          ~ OP[ ope-e /{<< ref-c ; 10 >> | << ref-e ; 1 >> | {} }{ << ref-g ; 4 >> | {} }]]
    ) & (
    GA[gam-c // $
          ~ OP[ ope-f /{<< ref-a ; 1 >> | << ref-e ; 1 >> | {} }/{ << ref-g ; 1 >> | {} }]
          ~ OP[ ope-g /{<< ref-b ; 10 >> | << ref-d ; 1 >> | {} }/{ << ref-a ; 18 >> | {} }]]
    ) & fin .

reduction result Bool : true .
```

Le texte est cohérent.

Les trois exemples suivants illustrent l'incohérence d'un texte au sens des contraintes [1], [2] et [3], respectivement.

```
reduce in LAPON as : verifie(
    GA[gam-d // $
        ~ OP[ ope-h /{<< ref-a ; 13 >> | <<ref-b ; 1>> | {} }/{ <<ref-b ; 3>> | {} }]
        ~ OP[ ope-i /{<< ref-b ; 10 >> | <<ref-e ; 1>> | {} }/{ <<ref-g ; 18>> | {} }]]
    ) & fin .

reduction result Bool : false .
```

```
reduce in LAPON as : verifie(
    GA[gam-e // $
        ~ OP[ ope-j /{<< ref-a ; 1 >> | <<ref-b ; 1>> | {} }/ {} }]
        ~ OP[ ope-k /{<< ref-d ; 10 >> | <<ref-e ; 1>> | {} }/{ <<ref-f ; 9>> | {} }]]
    ) & fin .

reduction result Bool : false .
```

```
reduce in LAPON as : verifie(
    GA[gam-f // $
        ~ OP[ ope-j /{<< ref-a ; 1 >> | << ref-b ; 1 >> | {} }/ { << ref-c ; 3 >> | {} }]
        ~ OP[ ope-k /{<< ref-d ; 1 >> | << ref-e ; 1 >> | {} }/{ << ref-f ; 9 >> | {} }]]
        ~ OP[ ope-l /{<< ref-f ; 9 >> | << ref-h ; 1 >> | {} }/{ << ref-i ; 18 >> | {} }]]
    ) & fin .

reduction result Bool : false .
```

I.3.3. Vers un langage algorithmique exécutable

De la même façon que nous avons calculé le résultat de l'opérateur verifie pour nous assurer de la cohérence d'une description, nous pouvons calculer le résultat de n'importe quel opérateur spécifié.

L'opérateur nbreop G calcule le nombre d'opérations qui composent la gamme G. Son résultat est un entier.

On appliquera d'abord l'équation qui spécifie que le nombre d'opérations d'une gamme est égal au cardinal de la liste de ses opérations :

nbreop GA[C // L] = card L (module GAMME).

On applique ensuite récursivement les équations qui calculent le cardinal d'une liste.

card $ = 0 et
card L ~ E = s card L (module LISTE).

On obtient la formule s(s(s(0))) (module NAT) qui représente le troisième successeur de 0, soit l'entier 3.

```
reduce in LAPON as : nbreop (
    GA[gam-f // $
            ~ OP[ ope-j /{<< ref-a ; 1 >> | << ref-b ; 1 >> | {} }/ { << ref-c ; 3 >> | {} }]
            ~ OP[ ope-k /{<< ref-d ; 1 >> | << ref-e ; 1 >> | {} }/{ << ref-f ; 9 >> | {} }]]
            ~ OP[ ope-l /{<< ref-f ; 9 >> | << ref-h ; 1 >> | {} }/{ << ref-i ; 18 >> | {} }]]
    ) & fin .

reduction result Bool : 3 .
```

Prenons un autre exemple. L'opérateur Id est-rattache-a G est un prédicat qui vérifie si G est une gamme qui produit la référence R. Pour cela on applique à nouveau les équations qui spécifient l'opérateur.

```
reduce in LAPON as : ref-i est-rattache-a (
    GA[gam-f // $
            ~ OP[ ope-j /{<< ref-a ; 1 >> | <<ref-b ; 1>> | {} }/ { <<ref-c ; 3>> | {} }]
            ~ OP[ ope-k /{<< ref-d ; 1 >> | <<ref-e ; 1>> | {} }/{ <<ref-f ; 9>> | {} }]]
            ~ OP[ ope-l /{<< ref-f ; 9 >> | <<ref-h ; 1>> | {} }/{ <<ref-i ; 18>> | {} }]]
    ) & fin .

reduction result Bool : true .
```

On voit qu'il suffit de spécifier des opérateurs de sommation de durées opératoires d'opérations, de substituabilité d'opérations et de gammes, de recouvrement d'opérations ... pour réunir les éléments d'un algorithme d'ordonnancement. L'algorithme écrit avec ces opérateurs est exécutable c'est-à-dire qu'il suffit d'appliquer les équations qui spécifient les opérateurs.

Le langage des opérateurs que l'on obtient est un langage algorithmique pour des problèmes de gestion. Les algorithmes sont indépendants de toute représentation en machine et répondent à la sémantique du système de production spécifié.

II. Preuve de consistance d'une spécification et construction consistante

Nous savons calculer la cohérence d'un texte par rapport à la sémantique des contraintes d'une spécification. Cependant ce calcul n'a de sens que si ces contraintes sont non contradictoires ou consistantes. Dans cette section nous montrons à la fois une procédure de preuve de consistance d'une spécification (II.1.) et une méthode de construction consistante d'une spécification (II.2.).

II.1. Spécification consistance et système de réécriture

La procédure de complétion de Kunth-Bendix ([KNUT70], [HUET80b], [HUET81], [JOUA84], [JOUA86c]) transforme un ensemble d'équations en un ensemble d'équations orientées ou règles. Le système de réécriture permet de réécrire un terme de façon unique par un nombre fini d'applications de règles en une forme dite normale ou irréductible. L'idée d'utiliser cette procédure de complétion pour prouver ou refuter des théorèmes dans l'algèbre initiale (théorème inductif) a été proposée par [MUSS80a]. Pour prouver qu'une équation est valide dans l'algèbre initiale d'une spécification on vérifie que l'adjonction de l'équation à l'ensemble E des équations de la spécification ne modifie pas la congruence générée par E. Ce procédé est appelé induction sans induction.

L'auteur impose une restriction aux spécifications : toute sorte possède un prédicat d'égalité complètement spécifié sur les valeurs de la sorte. Pratiquement il faut spécifier l'égalité des booléens, des entiers naturels, des listes (...). Ainsi l'égalité de deux termes peut être déduite de la théorie équationnelle (définition 6) d'une spécification.

L'approche de [GOGU80] reprend l'idée de [MUSS80a]. [GOGU80] exige la spécification d'un prédicat d'égalité sur les booléens et pour tout Σ-terme clos une expression booléenne qui l'évalue. Si $t = t'$ est un théorème que l'on veut prouver à partir de E, il suffit de vérifier que true = false (égalité entre les constantes booléennes) ne peut être déduite de la théorie équationnelle de $E \cup \{t = t'\}$. Afin de rendre automatique cette preuve on vérifie si le système de réécriture obtenu à partir $E \cup \{t = t'\}$ contient une règle true -> false ou false -> true.

Des approches plus récentes [HUET82], [KIRC84], [DERS85a], [JOUA86a], [JOUA86b], [FRIB85], [PAUL84], [KOUA85] et [LAZR87] s'affranchissent de cette contrainte du prédicat d'égalité en introduisant une caractérisation des opérateurs de la signature. Nous renvoyons à l'état de l'art dans [KOUA85] et [LAZR87] et nous retenons ici [MUSS80a] et [GOGU80] que nous préférons pour la simplicité de son principe.

Une spécification qui remplit les conditions de [GOGU80] est appelée s-observable[1].

Définition 20 Une spécification SPEC = (S, OP, E) est dite **s-observable** pour une sorte s si et seulement si

(c1) S contient la sorte bool qui possède les constantes true et false et un symbole d'opération == de profil <bool bool, bool> où == est l'opérateur d'égalité de termes clos booléens dans le sens où

$$t, t' \in T_{OP,bool},\ (t == t') = true \text{ si } t \equiv_E t' \text{ et } (t == t') = false \text{ si } not(t \equiv_E t'),$$

(c2) pour tout $t, t' \in T_{OP,s}$, $s \in S$ avec not $(t \equiv_E t')$ il existe une expression u de sorte bool comportant une seule variable de sorte s tel que $(u(t) == u(t')) \equiv_{E,bool} = false$.

Nous appelons SPEC **observable** si et seulement si SPEC est s-observable pour tout $s \in S$ et nous appelons SPEC **consistant** si not(true $\equiv_E$ false). ❑

Un enrichissement consistant d'une spécification établit une relation entre deux algèbres.

Theorème 1 ([GOGU80])

Soit SPEC = (S, OP, E) une spécification consistante et E' un ensemble de Σ-équations tel que si E' contient une équation de sorte s alors SPEC est s-observable.

Soit SPEC' = (S, OP, $E \cup E'$). On a de façon équivalente :

(i) chaque équation de E' est valide dans $T_{SPEC/E}$,

(ii) $T_{SPEC/E} = T_{SPEC'/E'}$,

(iii) SPEC' est consistant. ■

Preuve ([GOGU80])

(i) et (ii) sont équivalents car si $e \in E'$ est valide dans $T_{SPEC/E}$ alors la congruence générée par e sur T_{OP} est contenue dans la congruence générée par E sur SPEC.

(i) $\Rightarrow$ (iii) Si SPEC' est inconsistant alors SPEC l'est également ce qui contredit l'hypothèse.

(iii) $\Rightarrow$ (i) Supposons que SPEC' est consistant et qu'il existe $e \in E'$, e = (g, d), qui n'est pas valide dans $T_{SPEC/E}$. Alors il existe une substitution h tel que not(h(g) $\equiv_E$ h(d)). Par la condition (c2) il existe une expression u de sorte bool tel que $u(h(g)) == u(h(d)) \equiv_E$ false et $\equiv_E \subset \equiv_{E \cup E'}$ implique $u(h(g)) == u(h(d)) \equiv_{E \cup E'}$ false. Or puisque $(g, d) \in E \cup E'$, $u(h(g)) \equiv_{E \cup E'}$ u(h(d)) et ainsi $u(h(g)) == u(h(d)) \equiv_{E'}$ true par la condition (c1) ce qui contredit la consistance de SPEC'. D'où il n'existe pas de substitution h et e est valide dans $T_{SPEC/E \cup E'}$.

---

[1] L'expression s-observable traduit 's-taut' utilisé dans [GOGU80].

Pour vérifier qu'un théorème est valide dans une l'agèbre initiale, sans expliciter un raisonnement par récurrence, on utilise le principe de l'induction sans induction. Ce principe est basé sur l'idée d'utiliser des techniques purement équationnelles, en l'occurrence la réécriture, pour tester automatiquement la validité d'une équation. La réécriture de termes se définit comme une relation sur un ensemble de termes. Nous déterminons quelques propriétés de cette relation avant de définir le système de réécriture qui représente une spécification.

Définition 21 Soit T un ensemble et R une relation sur T. aRb signifie que a peut être **réécrit** en b et on note a -> b. R^+, R^*, $\equiv_R$ représentant respectivement la **fermeture transitive, transitive-réflexive et transitive-réflexive-symétrique de R** que l'on note a $-^+\!\!>$ b, a $-^*\!\!>$ b et a $\equiv$ b. $\equiv$ est une **relation d'équivalence.** t est dit de forme **R-réduit,** et on note t↓, s'il n'existe pas de t' tel que tRt'. R est à **terminaison unique** si chaque classe d'équivalence contient au plus une forme réduite. R est **noetherien** si et seulement si il n'existe pas de séquence infinie t_1 -> t_2 -> ... t_n -> ... . R est **globalement fini** si et seulement si $\{b \mid aR^*b\}$ est fini pour chaque a dans T. R est **confluent** si et seulement si aR^*b et aR^*c implique qu'il existe bR^*d cR^*d. R est **localement confluent** si et seulement si aRb et aRc implique qu'il existe bR^*d cR^*d. ❑

Proposition 3 ([GOGU80]) Si R est confluent alors R est noetherien. De plus si t $\equiv_R$ t' et t' est R-réduit alors tR^*t'. Si R est confluent et noetherien alors chaque classe d'équivalence construite sur $\equiv_R$ contient un seul élément qui est une forme R-réduite. ◆

Définition 22 Un **système de réécriture** multi-sorte (T, R) est composé d'une famille T = $\langle T_s \mid s \in S\rangle$ d'ensembles T_s de sorte s et d'une famille R = $\langle R_s \mid s \in S\rangle$ de relations R_s de sorte s. On écrit a -> b si aR_sb pour s donné. R est confluent et noetherien si et seulement si chaque R_s l'est. ❑

Définition 23 Soit Σ = (S, OP) une signature et E une famille de Σ-équations. Pour chaque équation e = (g, d), e $\in$ E_s, on définit une relation R_e sur $T_{OP,s}$ par : (t, t') $\in$ R_e si et seulement si t = h(g) et t'= h(d) pour h : X -> T_{OP} une substitution de termes clos à des variables dans g et d. Soit Q_s = $\langle R_e \mid e \in E_s\rangle$ et Q = $\langle Q_s \mid s \in S\rangle$. On définit R = $\langle R_s \mid s \in S\rangle$ avec tR_st' si et seulement si il existe t_1 et t_2 tels que $t_1Q_st_2$ et t = $h_1(u)$ et t' = $h_2(u)$ avec h_1 qui substitue t_1 à x dans u et h_2 qui substitue t_2 à x dans u et u $\in$ $T_{OP,s}(\{x\})$ (c'est-à-dire u est un terme qui possède

une seule variable x). Dans ce cas (T_{OP}, R) est un **Σ-système de réécriture défini par une spécification** SPEC = (S, OP, E). On dira que E est confluent, noetherien (...) sur SPEC si R l'est sur T_{OP}. ❑

La propriété de confluence est indécidable pour un système de réécriture quelconque. Elle est décidable pour des systèmes noetheriens. La propriété de terminaison (noetherien) est indécidable pour des systèmes de réécriture quelconques. Les théorèmes suivants (Newmann et Knuth-Bendix) [HUET80b] fournissent une procédure de décision de la confluence d'un système de réécriture noetherien.

Théorème de Newman Soit R un système noetherien. R est confluent si et seulement si R est localement confluent. ■
Cette confluence locale est décidable pour des systèmes finis de réécriture.

Définition 24 Soit u, v $\in$ $T_{OP}(X)$. u est une **instance** de v s'il existe σ: X --> $T_{OP}(X)$ tel que v = σ(u). On dit que u et v sont **unifiables** s'il existe une substitution σ: X --> $T_{OP}(X)$ tel que $\sigma(u) \equiv \sigma(v)$. Un **unificateur** est un ensemble de telles substitutions. Soit u -> v et u' -> v' deux règles de R. u et u' sont renommés de façon à ne plus avoir de variables communes. Soit p une occurence de u tel que $u_{/p}$ et u' sont unifiables avec un unificateur minimal σ. <σ[p<--v'],σ(v)> est une **paire critique**. ❑

Soient f(x, g(x, h(y))) -> k(x, y) et g(a,z) -> l(z) deux règles. {x <-- a, z <-- h(y) } est un unificateur minimal. <f(a, l(h(y))), k(a,y)> est une paire critique.

Théorème de Knuth-Bendix R est un système localement confluent si et seulement si u↓ = v↓ pour toute paire critique <u, v> de R. ■

Si la confluence est décidable pour des systèmes de réécriture noethériens, la terminaison reste une propriété indécidable. Les algorithmes utilisés pour décider de la terminaison d'un système de réécriture reposent en général sur la structure des règles. Une telle structure est, par exemple, la taille des termes d'une règle où chaque instance du membre droit doit être strictement plus petite que l'instance correspondante du membre gauche. Nous n'entrons pas dans le détail des structures de règles permettant des réécritures finies et nous renvoyons à [HUET80b], [JOUA86b], [DERS85a], [DERS85b], [DERS85b], [DERS85b], [BACH86], [BUCH85], [JOUA83], [CHER87], [LESC87] et [DERS85c].

Si le système de réécriture noetherien est confluent selon les théorèmes de Newman et Knuth-Bendix alors il est complet ou canonique, ou encore il est la forme canonique d'une

théorie équationnelle.

Les systèmes de réécriture canoniques respectant les théorèmes de Newman et Knuth-Bendix fournissent une procédure de décision pour la validité d'équations dans une variété équationnelle d'un ensemble de base E.

Définition 25 Soit E un ensemble fini d'équations qui sont les axiomes d'une théorie. S'il existe un **système de réécriture R canonique** tel que

(1) pour tout (u, v) $\in$ E on a : u$\downarrow$ = v$\downarrow$ et

(2) pour tout u -> v $\in$ R on a : u $\equiv_E$ v,

alors u est une forme canonique de u pour la théorie équationnelle générée par $\equiv_E$ dans le sens où pour tout u, v $\in T_{OP}(X)$ on a : u $\equiv_E$ v si et seulement si u$\downarrow$ = v$\downarrow$. ❑

L'algorithme de completion de Knuth-Bendix K-B(E) tente de transformer un ensemble E d'équations de base en un système R de réécriture compte tenu des conditions (1) et (2) ci - dessus.

K-B(E)

0 : $i = 0 ; E_i = E ; R_i = \phi$;

1 : Si $E_i = \phi$ alors $R = R_i$; *aller en 4 ;*

sinon choisir g = d dans E_i ; calculer g$\downarrow$ et d$\downarrow$ à partir de R_i ;

Si g$\downarrow$ = d$\downarrow$ alors $E_{i+1} = E_i - \{g = d\}$;

$R_{i+1} = R_i ; i = i+1;$

aller en 1 ;

sinon *aller en 2 ;*

2 : Choisir un ordre de réduction

a) g$\downarrow$ > d$\downarrow$ alors u = g$\downarrow$ et v = d$\downarrow$

b) d$\downarrow$ > g$\downarrow$ alors v = g$\downarrow$ et u = d$\downarrow$

Si ni a) ni b) ne s'applique alors *aller en 5 ;*

sinon $\mathcal{E}_i = \{u' = v' / u' \text{ -> } v' \in R_i$ et u' ou v' contient une instance de u en tant que sous-terme$\}$;

$\mathcal{R}_i = R_i - \mathcal{E}_i \cup \{u \text{ -> } v\}$; *aller en 3 ;*

3 : Si $\mathcal{R}_i$ n'est pas noetherien alors *aller en 5 ;*

sinon $R_{i+1} = \mathcal{R}_i$;

$E_{i+1} = (E_i - \{g = d\}) \cup \mathcal{E}_i \cup \{p = q / <p, q> \text{ est paire critique de } R_{i+1}\}$;

$i = i+1$; *aller en 1 ;*

4 : *ARRET AVEC SUCCES ;*

5 : *ARRET AVEC ECHEC ;*

Remarque : > désigne un ordre de réduction tel que celui de la taille des termes qui assure, s'il est applicable, que le système de réécriture se termine.

L'algorithme de Knuth - Bendix est correct :

(i) A chaque itération i $\equiv_{Ei \cup Ri}$ est égal à $\equiv_E$

(ii) Chaque R_i est noetherien et chaque $\{u \rightarrow v\} \in R_i$ est tel que v est en forme normale pour R_i est v en forme normale pour $R_i - \{u \rightarrow v\}$

(iii) Si l'algorithme s'arrête avec succès R est localement confluent et ainsi les conditions (1) et (2) sont satisfaites.

Proposition 4 ([GOGU80])

Soit E une famille de Σ-équations, l'algorithme K-B(E), s'il s'arrête, donne un ensemble R de Σ-règles tel que :

(kb1) R est consistant si et seulement si E est consistant ;
(kb2) R est localement confluent ;

De plus, même si l'algorithme ne s'arrête pas,

(kb3) une équation e peut être déduite de E si et seulement si elle peut être déduite de R_i ;

(kb4) E est inconsistant si et seulement si il existe R_i tel que true -> false ou false -> true. ◆

Nous pouvons conclure avec le théorème de [GOGU80] que l'algorithme qui remplit les conditions kb1 à kb4 permet à la fois de décider de la consistance d'une spécification et de prouver des propriétés des sortes spécifiées.

Theorème 2 ([GOGU80])

Soit SPEC = (S, OP, E) une spécification consistante et soit E' une famille de Σ-équations tel que si E' contient une équation de sorte s alors SPEC est s-observable. Soit K-B() l'algorithme qui satisfait les conditions (kb1) - (kb4) et K-B($E \cup E'$) donne R.
Alors
(1) si R existe, est noetherien et consistant, alors chaque équation dans E' est vraie dans $T_{SPEC/E}$;
(2) il existe une itération i tel que R_i contient true -> false ou false -> true si et seulement si une équation de E' n'est pas vraie dans $T_{SPEC/E}$. ■

A titre d'exemple nous nous donnons une spécification SPEC = (S, OP, E) qui spécifie les booléens (bool) et les entiers naturels (nat). Elle est s-observable (cf. définition 20), car la sorte bool possède un opérateur booléen eq : bool bool -> bool d'égalité d'expressions booléennes (c1) et la sorte nat comporte un observateur booléen iszero : nat -> bool (c2) qui permet de distinguer les entiers naturels réprésentés par des formules $s^n(x)$, $n \geq 0$..

sorte bool
opérateurs
false : --> bool
true : --> bool
not : bool --> bool
and : bool bool --> bool
equal : bool bool --> bool
var x : bool
équations
(1) not(true) = false
(2) not(false) = true
(3) not(not(x)) = x
(4) and(x, true) = x
(5) and(x, false) = false
(6) or(x, false) = x
(7) or(x, true) = true
(8) equal(false, false) = true
(9) equal(true, true) = true
(10) equal(true, false) = true
(11) equal(false, true) = true

sorte nat
opérateurs
0 : --> nat
s : nat --> nat
add : nat nat --> nat
iszero : nat --> bool /* teste si un entier est egal à 0 ou non */
var x, y : nat
équations
(1) add(0, x) = x
(2) add(x, s(y)) = s(add(x, y))
(3) iszero(0) = true
(4) iszero(s(x)) = false

Le système de réécriture[1] R_{SPEC} que nous obtenons avec K-B(E) est :

R_{SPEC} = {

1. not(not(x)) -> x,
2. and(x, true) -> x,
3. and(x, false) -> false,
4. or(x, false) -> x,
5. or(x, true) -> true,
6. equal(true, false) -> false,
7. equal(false, true) -> false,
8. equal(true, true) -> false,
9. equal(false, false) -> false,
10. not(true) -> false,
11. not(false) -> true,
12. add(0, x) -> x,
13. add(x, s(y),) -> s(add(x, y)),
14. iszero(0) -> true,
15. iszero(s(x)) -> false }

La spécification SPEC est consistante, car true -> false ou false -> true n'apparaît pas parmi les règles.

En ajoutant E'={s(x) = 0} à SPEC nous obtenons une spécification SPEC'=(S, OP,E∪E'). Nous déroulons l'algorithme de complétion K-B(E ∪ E') pour vérifier si SPEC' est consistant. Afin de ne pas refaire tous les calculs, nous supposons qu'à l'itération n-1 nous avons obtenu R_{n-1} qui correspond au système de réécriture R_{SPEC} obtenu par K-B(E).

A l'étape 1 de l'itération n de l'algorithme de complétion nous avons :

1 : $E_n = E'$ et $R_n = R_{SPEC}$.

Nous choisissons l'équation s(x) = 0 contenue dans E_n.

Soit g = s(x) et d = 0 on a pour R_n les formes normales g↓ = s(x) et d↓ = 0.

Les formes normales ne sont pas égales et nous passons à l'étape 2.

---

[1] Pour effectuer le calcul de complétion de Knuth-Bendix sous Reve2.7 [LAZR87] pour notre spécification multi-sorte, nous avons dû transformer les équations pour les adapter à une complétion mono-sorte. Nous ne donnons pas ici les équations transformées, car elles pourraient prêter à confusion.

2 : Nous orientons s(x) -> 0, car la taille du membre droit de l'équation est plus petite que la taille du membre gauche.

Les règles 13. add(x, s(y),) -> s(add(x, y)),
et 15. iszero(s(x)) -> false contiennent une instance de s(x) en tant que sous-terme.

On constitue l'ensemble $\mathcal{E}_n$ = {add(x, 0) = s(add(x, y)), iszero(0) -> false}.

On constitue le nouveau système de réécriture $\mathcal{R}_n$ à partir de l'ancien R_n en lui enlevant les règles qui correpondent aux équations de $\mathcal{E}_n$ et en ajoutant la nouvelle règle s(x) -> 0.

3 : $\mathcal{R}_n$ est noethérien, car pour chacune de ses règles le membre droit est plus petit que le membre gauche. Ainsi les réécritures par application des règles se terminent.

$R_{n+1} = \mathcal{R}_n$

$E_{n+1} = \mathcal{E}_n$, car E_n est vide et on n'obtient pas de paires critiques pour R_{n+1}.

1 : R_{n+1} est composé des règles 1 à 12, 14 et s(x) -> 0.

E_{n+1} = {add(x, 0) = s(add(x, y)), iszero(0) = false}.

On choisit l'équation iszero (0) = false et on calcule les formes normales pour R_{n+1}.
On obtient true = false. Cette équation est orientée en une règle **true -> false** ou **false -> true** que l'on rajoute à l'étape 2 à $\mathcal{R}_{n+1}$ puis à l'étape 3 à R_{n+2}.

D'où la spécification SPEC' sur E ∪ E' est inconsistante, car R_{n+2} contient une règle true-> false.

D'après le théorème 2, l'équation s(x) = 0 n'est pas valide dans l'algèbre initiale $T_{SPEC/E}$ de la spécification SPEC = (S, OP, E). En effet, les éléments du support de $T_{SPEC/E}$ constituent un ensemble de classes d'équivalence Q_{SPEC} = {[true], [false]} ∪ {[s^n(0)] / n ≥ 0} (cf. définition 11). $iszero_Q$ est une fonction sur les classes d'équivalence définie par $iszero_Q$([s^n(0)]) = [false], pour n ≥ 0 et $iszero_Q$([s^n(0)]) = [true], pour n = 0.

Par l'adjonction de E' = {s(x) = 0} la nouvelle relation de congruence générée par E ∪ E' confond les classes d'équivalence représentant les entiers naturels en identifiant chaque successeur de 0 à 0. Avec l'opérateur iszero nous répercutons cette destruction de la sémantique initiale des entiers naturels dans la sémantique des booléens et on obtient [true] = [false].

II.2. Construction consistante d'une spécification

Dans l'introduction à ce papier nous avons noté que nous recherchons un langage ouvert basé sur une spécification qui forme un noyau consistant et minimal. Ce noyau représente la spécification d'informations qui existent dans chaque système de gestion en production discontinue. Afin de pouvoir adapter le langage et donc sa spécification à un système de production particulier, nous devons enrichir la spécification avec des types d'information nouveaux et de nouvelles contraintes sur les types et leurs liens. Dans cette sous-section nous montrons comment nous pouvons construire une spécification consistante d'un système de gestion particulier à partir d'un noyau de spécifications consistantes.

L'algèbre initiale représente la sémantique de la spécification. Elle est caractérisée par des supports (classes d'équivalence) et des applications (fonctions sur les classes d'équivalence) interprétant les opérations. En construisant une spécification d'une manière hiérarchique, il faut s'assurer que les nouvelles équations ne changent ni les supports ni les applications préalablement définies. En termes algébriques, on veut que l'algèbre initiale de départ soit isomorphe à la restriction de la nouvelle algèbre aux anciennes sortes et opérations. Autrement dit, les classes d'équivalence engendrées respectivement par l'algèbre de départ et la nouvelle algèbre réduite aux anciennes sortes et opérations doivent être identiques.

Définition 26 Soit $\Sigma = (S, OP)$ et $\Sigma' = (S', OP')$ deux signatures avec $S \subseteq S'$ et $OP \subseteq OP'$. Si B est une Σ'-algèbre, on note $U_{\Sigma}(B)$ la Σ-algèbre obtenue à partir de B en oubliant toutes les sortes et opérations qui sont dans Σ' et qui ne sont pas dans Σ.
On dit que la Σ'-agèbre A' est une **extrusion** de la Σ-algèbre A si et seulement si $A \subseteq U_{\Sigma}(A')$.
A' est une extension de A si $A = U_{\Sigma}(A')$. ❑

Sous certaines hypothèses on verifie que l'algèbre initiale d'une spécification construite à partir d'une spécification de base est une extrusion de l'abgèbre initiale de la base.

Théorème 3 ([GOGU80])

Soit SPEC = (S, OP, E) une spécification consistante. Soit S' une ensemble de sortes disjoint de S et OP' un ensemble d'opérations $(S \cup S')$-sortés disjoint de OP et soit E' un ensemble d'équations de sortes $(S \cup S')$. Soit $SPEC' = (S \cup S', OP \cup OP', E \cup E')$ une spécification consistante et observable alors $T_{SPEC'/E \cup E'}$ est une extrusion de $T_{SPEC/E}$. ■

Le théorème suivant montre que l'on peut à nouveau se servir de l'algorithme de completion K-B() pour vérifier la consistance de la spécification construite.

Théorème 4 ([GOGU80])

Sous les hypothèses du théorème précédent et K-B() un algorithme qui satisfait aux conditions kb1 à kb4 et K-B(E ∪ E') donne R :

(i) si R est noetherien, consistant et observable alors chaque équation valide dans $T_{SPEC/E}$ est valide dans $T_{SPEC'/E\cup E'}$, $T_{SPEC'/E\cup E'}$ est une **extrusion consistante** de $T_{SPEC/E}$.

(ii) une équation de E' n'est pas valide dans $T_{SPEC/E}$ si et seulement s'il existe i tel que R_i contient la règle true -> false ou false -> true. $T_{SPEC'/E\cup E'}$ est une **extrusion consistante** de $T_{SPEC/E}$ s'il n'existe pas un i tel que R_i contient true -> false ou false -> true. ■

Nous illustrons ce résultat en reprenant l'exemple de la section précedente. Appelons SPECX = ($S_b \cup S_n$, $OP_b \cup OP_n$, $E_b \cup E_n$) la spécification SPEC = (S, OP, E) de l'exemple de la section précédente. SPECX est construite à partir de la spécification de la sorte bool, SPECB = (S_b, OP_b, E_b) et de la spécification de la sorte nat, SPECN = (S_n, OP_n, E_n). D'après le théorème 4, $T_{SPECX/Eb\cup En}$ est une extrusion consistante de $T_{SPECB/Eb}$, car R_{SPECX} (= R_{SPEC} de la section précédente) ne contient pas de règle true -> false ou false -> true.

III. Grammaire d'une spécification et interface avec un langage utilisateur

Le langage des formules construites à partir des opérateurs d'une spécification peut être très complexe. Pour les besoins d'une description de la gestion d'un système de production il est nécessaire de disposer d'un langage simple et normalisé : les descriptions doivent être facile à lire et à rédiger. Un langage qui remplit ces conditions est appelé dans la suite langage de production utilisateur (LPU). Le langage de production construit sur la syntaxe des opérateurs d'une spécification est appelé langage de spécification (LPS). La séparation du problème de design d'une spécification et de son langage (consistance et calcul de cohérence) du problème de design d'un langage qui correspond aux besoins de l'utlisateur possède un double avantage :

• la définition du LPU peut être différée. En effet le LPU[1] sera le résultat d'études de normalisation qui obéissent à des critères ergonomiques et de faisabilité informatique. De telles études doivent se dérouler sous l'égide d'une instance de normalisation ;

• les modifications syntaxiques du LPU voulues par l'utilisateur ou dues à des problèmes d'implémentation ne mettent pas en cause la consistance de la spécification, mais seulement la définition de l'interface avec le LPS.

Nous définissons dans cette section la grammaire du LPS et nous étudions son interfaçage avec un LPU.

III.1. Grammaire d'une spécification

Une grammaire est une définition syntaxique formelle d'un langage [AHOU72], [AHOU73], [DAVI87], [WIRT76], [SALO73], [SALO85], [GALL84]. Nous définissons dans cette sous-section une grammaire du langage LPS à partir de la signature de la spécification. [GOGU77b] et [GOGU78a] étudient une grammaire pour une spécification multi-sorte non-paramétrée ; [MUSS80a] et [GUTT78] définissent un langage d'une spécification sans expliciter sa grammaire.

Soit $G = (\Theta, \Omega, \Pi)$ une grammaire avec
• Θ ensemble de symboles terminaux
• Ω ensemble de symboles non terminaux
• Π ensemble de règles de production de la forme $\alpha \text{ --> } \beta$, où $\alpha \in \Omega$ et $\beta \in (\Theta \cup \Omega)^*$.

---

[1]Le LPU peut être un langage graphique qui se sert d'icônes pour représenter machines et stocks, de flèches pour indiquer le passage de produits sur les machines (...) . Le LPU peut également se présenter sous forme de textes formatés d'une rédaction normalisée de cahier des charges.

Soit $\Sigma = (S, OP)$ une signature (cf. définition 1). On pose $\Omega = S$ et $\Theta = \{N \in OP\} \cup \{(,)\}$.

Pour chaque opérateur $N \in OP_{w,s}$ on écrit une règle de production s --> v

avec $v = u_1\ s_1\ u_2 \ldots\ u_k\ s_k\ u_{k+1} \ldots\ u_n\ s_n\ u_{n+1}$, $u_i \in \Theta^*$, $s_i \in \Omega$, $v \in (\Theta \cup \Omega)^*$ et

$w = s_1 \ldots s_n$.

Prenons l'exemple d'une signature $\Sigma = (S, OP)$ des exemples des définitions 13 et 15.

sorte bool				**sorte** nat			
opérateurs				**opérateurs**			
false	:		---> bool	0	:	--->	nat
true	:		---> bool	succ(_)	: nat	--->	nat
not(_)	: bool		---> bool	add(__)	: nat nat	--->	nat
and	: bool bool		---> bool	_≤_	: nat nat	--->	bool
impl	: bool bool		---> bool	eqn(__)	: nat nat	--->	bool

La grammaire $G(\Sigma) = \{ \Theta, \Omega, \Pi)$ est donnée par :

- $\Theta = \{$ (,), false, true, not, and, 0, succ, +, ≤ $\}$
- $\Omega = \{$ bool, nat $\}$
- $\Pi = \{$ bool --> true, bool --> false, bool --> not (bool), bool --> bool and bool, bool --> bool impl bool, nat --> 0, nat --> succ (nat), nat --> add(nat, nat), bool --> nat ≤ nat, bool --> eqn(nat nat) $\}$

Une spécification SPECTF obtenue par instantiation de $SPECP = (S_P, OP_P, E_P)$ avec un paramètre effectif admissible $SPECF = (S_F, OP_F, E_F)$ dans un environnement $SPEC = (S, OP, E)$ se note $SPECTF = SPEC + SPECF + (TP(S_F), OP_{TF}, E_{TF})$ (cf. définition 18 et 19).
On pose $\Omega_{TF} = S \cup TP(S_F)$ et $\Theta_{TF} = \{N \in OP\} \cup \{N \in OP_{TF}\} \cup \{(,)\}$.

Pour chaque opérateur $N \in OP_{TFw,s}$ on écrit une règle de production s --> v où $s \in TP(S_F)$

avec $v = u_1\ s_1\ u_2 \ldots\ u_k\ s_k\ u_{k+1} \ldots\ u_n\ s_n\ u_{n+1}$, $u_i \in \Theta^*$, $s_i \in \Omega$, $v \in (\Theta \cup \Omega)^*$ et

$w = s_1 \ldots s_n$.

Prenons l'exemple de la définitions 18 où Σ = (S, OP) est la signature de spécification de l'environnement et Σ_{liste} = (liste(nat), OP_{liste}) la signature de la spécification paramétrée liste instanciée par nat.

sorte	liste(P)			**sorte**	P	
opérateurs				**opérateurs**		
$	:		--> liste(P)	indefini :		-->P
_~_	:	liste(P)	P --> liste(P)	egal(__):	P	P -->bool
dernier_	:	liste(P)	--> P			
reste_	:	liste(P)	--> liste(P)			

La grammaire $G_{TF}(\Sigma_{TF})$ = { Θ_{TF}, Ω_{TF}, Π_{TF}) est donnée par :

- $\Theta_{TF} = \Theta \cup$ { \$, ~, dernier, reste, indefini, egal }
- $\Omega_{TF} = \Omega \cup$ { liste(nat) }.
- $\Pi_{TF} = \Pi \cup$ { liste(nat) --> \$, liste(nat) --> liste(nat) ~ nat, nat --> dernier liste(nat), liste(nat) --> reste liste(nat), nat --> indefini, bool --> egal(nat,nat) }

Si nous construisons un langage sur les listes d'entiers naturels, nous ajoutons à Ω_{TF} une variable de départ ω ($\Omega'_{TF} = \Omega_{TF} \cup \{\omega\}$) et à Π_{TF} la règle de début de dérivation ω --> liste(nat) ($\Pi'_{TF} = \Pi_{TF} \cup \{\omega$ --> liste(nat)$\}$), nous obtenons une grammaire context-free $\mathcal{G}_{TF}$ au sens de Chomsky définie par $\mathcal{G}_{TF} = (\Theta_{TF}, \Omega'_{TF}, \Pi'_{TF}, \omega)$ (cf aussi [HILG88]).

La formule \$ ~ succ(0) ~ add(succ(0), 0) ~ succ(succ(0)) appartient à $L(\mathcal{G}_{TF})$, le langage défini par $\mathcal{G}_{TF}$.

Pour la spécification de l'univers I de la gestion du court terme (I.3.) nous obtenons la grammaire $\mathcal{G}_{Univers\ I}$. Nous choisissons de construire un langage sur la sorte Universal, qui désigne tous les termes clos. Toute déclaration d'un ordre d'inclusion de sortes dans une implémentation OBJ3 est traduite de la façon suivante :

subsort Gam < Phrase donne lieu à la règle Phrase --> Gam.

$\mathcal{G}_{\text{Univers I}} = (\ \Theta_{\text{Univers I}},\ \Omega'_{\text{Univers I}},\ \Pi'_{\text{Univers I}},\ \omega)$

$\Theta_{\text{Univers I}}$ = { ref-a, ref-b, ref-c, ref-d, ref-e, ref-f, ref-g, ref-h , ref-i, ref-j, ref-k, ref-l, op-a, op-b, op-c , op-d , op-e , op-f , op-g, op-h , op-i, op-j, op-k, op-l, gam-a, gam-b, gam-c, gam-d, gam-e, gam-f, if, then, else, fi, &, [, //,], $, ~, listop, reste, operation, numero-de, OP[, /{, }/{, }], dernier, numero, de, intersect, entree, sortie, |, ajout , {}, #, adj, nom, nom-ope, nomgam, << , ;, >>, 0, s , +, nbreop , card, qté, verifie, dire, dire-gamme, dire-operation, corr-gam, corr-lien-gam, corr-list-ope, corr-vide, corr-lien, corr-ope, est-rattache-a, dans, utilise, fabrique, in, <, true, false, and, or not, ==, =/= }

$\Omega'_{\text{Univers I}}$ = { ω, Universal, Text, Phrase, Gam, Liste[Ope], Ope, Set[Ref], Idref, Idope, Idgam, Ref, Nat, Bool }

$\Pi'_{\text{Univers I}}$ = {

ω -> Universal,

Universal --> Text | Phrase | Gam | Liste[Ope] | Ope | Set[Ref] | Ref | Idref | Idope | Idgam | Nat | Bool | if Bool then Universal else Universal fi,

Text --> fin | Phrase & Text,

Phrase --> Gam,

Gam --> GA[Idgam // Liste[Ope]],

Liste[Ope] -->listop Gam | $ | Liste-non-vide[Ope] | Liste[Ope] ~ Ope | reste Liste-non-vide[Ope],

Ope --> operation Nat numero-de Gam | OP[Idope /{ Set[Ref] }/{ Set[Ref] }] | dernier Liste-non-vide[Ope] | numero Nat de Liste[Ope],

Set[Ref] --> Set[Ref] intersect Set[Ref] | entree Ope | sortie Ope | Ref | Set[Ref] | Ref ajout Set[Ref] | {} | Ref # Set[Ref] | adj Ref Set[Ref]

Idref --> nom Ref | ref-a | ref-b | ref-c | ref-d | ref-e | ref-f | ref-g | ref-h | ref-i | ref-j | ref-k | ref-l,

Idope -> nom-ope Ope | op-a | op-b | op-c | op-d | op-e | op-f | op-g | op-h | op-i | op-j | op-k | op-l,

Idgam --> nomgam Gam | gam-a | gam-b | gam-c | gam-d | gam-e | gam-f,

Ref --> << Idref ; Nat >>,

Nat --> 0 | s Nat | Nat + Nat | nbreop Gam | card Liste[Ope] | qté Ref |

Bool --> verifie Text | dire Phrase | dire-gamme Gam | dire-operation Ope | corr-gam Liste | corr-lien-gam Liste | corr-list-ope Liste | corr-vide Ope | corr-lien Ope | corr-ope Ope | Idref est-rattache-a Gam | Ref dans Set[Ref] | Ope utilise Idref | Ope fabrique Idref | Ref in Set[Ref] | Nat < Nat | true | false | Bool and Bool | Bool or Bool | not Bool | Universal == Universal | Universal =/= Universal }

III.2. Interfaçage de la spécification avec un langage utilisateur

Dans cette sous-section nous posons, sans pour autant le résoudre, le problème d'interfaçage du langage LPS avec le langage LPU.

Nous supposons ici que le LPU est défini par une grammaire context-free $\mathcal{G}_U = (\Theta_U, \Omega'_U, \Pi'_U, \omega_U)$ avec LPU = $L(\mathcal{G}_U)$. Le langage LPS est défini par la grammaire $\mathcal{G}_S = (\Theta_S, \Omega'_S, \Pi'_S, \omega_S)$ avec LPS = $L(\mathcal{G}_S)$.

Soit T un traducteur qui définit une relation T : LPU --> LPS, qui associe toute phrase lu $\in$ LPU à une phrase ls $\in$ LPS. Remarquons que le mot vide ε appartient aux deux langages et que $T(\varepsilon) = \varepsilon$.

Soit $K \subset$ LPU un ensemble de phrases $k \in K$ tel que $T(k) = \varepsilon$. K représente un sous-ensemble de phrases du langage LPU qui n'ont pas de phrases correspondantes dans la spécification (par exemple les formules introductives de texte, les descriptions non prévues par la spécification).

Soit $R \subset$ LPS un ensemble de phrases $r \in R$ tel que $T^{-1}(r) = \varepsilon$. R désigne un sous-ensemble de phrases de la spécification qui n'ont pas de phrase utilisateur qui les traduit (par exemple, les formules de spécifications construites sur des opérateurs cachés).

De plus on peut admettre la redondance dans le langage utilisateur dans le sens où T : LPU \ K --> LPS\ R est une injection.

Le problème d'un traducteur décrit ci-dessus peut être résolu en utilisant les techniques classiques des traductions dirigées par la syntaxe ([AHOU72], [AHOU73], [ABRA73] et [WEIN73)] les automates et les machines à piles.

Conclusion

Nous avons présenté dans ce rapport de recherches à deux parties deux approches de conception d'un langage de description et de commande de systèmes de gestion de production.

La première approche utilise une méthode de représentation d'un système d'information pour obtenir la grammaire du langage qui décrit la gestion du court terme.

La deuxième approche spécifie algébriquement le système d'information et introduit ainsi une sémantique mathématique du système spécifié. Nous savons prouver la consistance de la spécification et de ses extensions. Le langage que nous avons obtenu est un ensemble de phrases formées avec les opérateurs de la spécification. La cohérence des descriptions que l'on rédige avec ce langage peut être calculée. Nous avons posé le problème d'une utilisation industrielle du langage et nous avons étudié l'interfaçage de notre langage avec un langage utilisateur quelconque.

Plusieurs types de travaux doivent être menés pour réaliser les objectifs énoncés pour le langage de production.

• Le langage de production du niveau du court terme doit être implanté avec un langage de spécification qui accepte des spécifications de taille industrielle et permet des complétions ordo-sortées [GNAE87a], conditionnelles [BERT88], [KAPL80] et admet des opérateurs associatifs - commutatifs [JOUA84].

• Afin de compléter notre langage nous devons spécifier les différents niveaux hiérarchiques de la gestion et définir les fonctions d'agrégations et de décomposition de l'information entre niveaux hiérarchiques.

• La description du suivi de la production et de sa commande oblige à introduire le temps dans nos spécifications. Les logiques modales [KROG87], [AUDU87], [AUDU84] et leurs méthodes de preuves nous semblent appropriées.

REFERENCES BIBLIOGRAPHIQUES

[ABRA73] Abrahamson, H., "Theory and Application of a Bottom-up Syntax Directed Translation",ACM Monographs, Academic Press, Montreal, Canada, 1973, p. 159.

[AHOU73] Aho, A., Ullman, J., "The Theory of Parsing, Translation and Compiling", vol. 2, Series in Automatic Computation, Prentice Hall, Englewood Cliffs, 1974.

[AUDU87] Audureau, E., Farinas del Cerro, L., "Théorie de la programmation et logique temporelle, première édition", Technique et Science Informatiques, vol. 6, no. 6, 1987, pp. 527-540.

[AUDU84] Audereau, E., Farinas del Cerro, L., "Programming Theory and Temporal Logic. Part one : validation of sequential algorithms", Technology ansd Science of Informatics, 1987.

[BARH70] Bar-Hillel, Y., "Aspects of Language", The Magnes Press, The Hebrew University, Jerusalem, 1970. p. 381.

[BACH86] Bachmair, L., Dershowitz, N., Hsiang, J., "Orderings for Equationnal Proofs", IEEE, 1986, pp. 346-357.

[BEIE86] Beierle, Ch., Vob, A., "On Implementation of Loose Abstract Data Type Specification and their Vertical Composition",1986, pp. 245-259.

[BERG82] Bergstra, J.A., Tucker, J.V., "Initial and Final Algebra Semantics for Data Type Specifications : Two Characterisation Theorems", Adelfing Informatica Amsterdam, IW 142/80, 1980, pp.1-36.

[BERT88] Bertling, H., Ganzinger, H., Schaeffers, R., "CEC - A system for Conditional Equational Completion. User manuel (Version 1.4)", University of Dortmund, Fachbereich Informatik, Dortmund, 1988.

[BROY82] Broy, M., Wirsing, M.,"Partial Abstract Data Types", Acta Informatica 3, 1982, pp.47-64.

[BROY85] Broy, M., "Structured Algebraic Specification of Backus' Functionnal Programming Language", Technique et Science Informatiques, vol. 5, 1985, pp. 447-458.

[BROY87] Broy, M., "Specification and Top-Down Design of Distributed Systems", Journal of Computer and System Sciences, vol. 34, no. 2/3, april/june 1987, pp. 236-265.

[BUCH85] Buchberger, B., "History and Basic Features of the Critical Pair /Completion Approach", Proceedings of the 1st Conference on Rewriting techniques and Applications, Lecture Notes in Computer Science, Dijon, vol. 202 , 1985, pp. 1-45.

[BURS77] Burstall, R. M., Goguen, J.,A. "Putting theories together to make Specifications", Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977, pp. 1045-1058.

[BURS81] Burstall, R. M., Goguen, J.,A. "An informal Introduction to Specification using CLEAR", Correctness Problem in Computer Science (Boyer &Moore eds.), 1981, pp. 185-213.

[BURS82] Burstall, R. M., Goguen, J.,A. "Algebras, Theories and Freeness : An Introduction for Computer Scientists", Proceedings 1981 Markoktoberdorf NATO Summer Scholl, Reidel, 1982, pp. 329-348.

[CHEN76] Chen, P., "The Entity-Relationship Model-Towards a Unified View of Data", ACM Transactions on Database Systems Vol. 1, No. 1, 1976, pp. 9-36.

[CHER87] Cherifa, A. B., "Termination of Rewriting System by Polynomial Interpretation and its Implementations", Rapports de Recherche INRIA, no. 677, juin 1987.

[CHOP85] Choppy, C., Johnen, C., "Petrireve : Proving Petri Net Propreties with Rewriting Techniques", Proceedings of the 1st Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 202 , Dijon, 1985, pp. 271-286.

[COMO88] Comon, H., "Equationnal Problems and Disunification", Rapports de Recherche INRIA, no. 804, septembre 1988.

[DAVI87] Davis, M.D., Weyuker, E.L., "Computability, Complexity and Languages, Fundamantals of theoretical Computer Science", Computer Science and applied Mathematics, Academic Press, Orlando, FA, 1987, p. 425.

[DERN79] Derniame, J.C., Finance, J.P., "Types Abstraits de Données : Spécification, Utilisation et Réalisation", Cours de l'Ecole d'été de l'AFCET. Monastir, 79.E.57, 1979.

[DERS85a] Dershowitz, N., "Computing with Rewrite Systems", Information and Control, no. 65, 1985, pp. 122-157.

[DERS85b] Dershowitz, N., "Termination", Proceedings of the 1st Conference on Rewriting techniques and Applications, Lecture Notes in Computer Science, vol. 202, Dijon, mai 1985,

pp.180-224.

[DERS85c] Dershowitz, N., "Termination of Rewriting", Journal of Symbolic Computation, no 3, 1987, pp. 69-116.

[DOSC82] Dosch, W., Mascari, G., Wirsing, M., "On the Algebraic Specification of Absract Data Types", Proceedings of the 8th International Conference of Very Large Database, Mexico City, 1982.

[EHRI72] Ehrich, H., "On the theory of Specification, Implementation and Parametrization of Abstrct Data Types", Forschungsbericht, Nr 82, Universität Dortmund, Abteilung fur Informatik, 1972.

[EHRI85] Ehrig, H.-D., Mahr, B., "Fundamentals of Algebraic Specification", Springer Verlag, 1985, p. 321.

[EHRI86] Ehrig, H.-D., Buntrock, J., "Towards an algebraic Semantics of the ISO specification Language", Draft Version, Technische Universität Berlin, 1986.

[FRIB85] Frieburg, L., "A Strong Restriction of the Inductive Completion Procedure", Lecture Notes in Computer Science, 226, 1985, pp.105-115.

[FUTA85] Futatsugi, F., Goguen, J.A., Jouannaud, J.P., Meseguer, J., "Principles of OBJ2", Proceedings of the 12thACM Symposium on Principles of Programming Languages Conference, 1985, pp. 1-20.

[GALL84] Gallaire, H., "Techniques de Compilation", Cepadues Editions, Toulouse, 1984, pp. 75-116.

[GNAE87a] Gnaedig, I., Kirchner, C., Kirchner, H., "Equationnal Completion in Order-Sorted Algebras Extended Abstract", Rapport de Recherche CRIN, Université Nancy I, 1987, p. 20.

[GNAE87b] Gnaedig, I., "Knuth-Bendix Procedure and Non Deterministic Behaviour - An Example", Rapports de Recherche INRIA, no. 733, 1987.

[GOGU76] Goguen, J.A., Thatcher, J.W., Wagner, E.G., "An Initial Algebra Approach to the Specification Correctness and Implentation of Abstract Data Types", IBM Research Report , RC 6487, 1976, p. 109.

[GOGU77a] Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B., "Initial algebra Semantics and the Continious Algebras", Journal of the Association of Computing Machinery,

Vol. 24, No 1, 1977,pp. 68-95..

[GOGU77b] Goguen, J.A., "Abstract Errors for Abstract Data Types", Working Conference on Formal Decription of Programming Concepts IFIP, 1977, pp. 491-525.

[GOGU78a] Goguen, J.A., "Some Ideas in Algebraic Semantics", Proceedings of the 4th IBM-Japon Symposion on Mathematical Formulations of Computer Science, Kobe, Japon, 1978.

[GOGU78b] Goguen, J.A., Tardo, J., "An Introduction to OBJ : A Language for Writing and Testing Formal Algebraic Program Specifications", Specification of Reliable Software IEEE, 1978, pp. 170-189.

[GOGU79a] Goguen, J.A., Tardo, J., "An Introduction to OBJ-T", Specification of Reliable Software IEEE, 1979, pp. 170-190.

[GOGU79b] Goguen, J.A., Tardo, J., Williamson, N., Zamfir, M., "A Practical Method for Testing Algebraic Specifications", UCLA, Computer Science Quaterly, vol. 7, no. 1, 1979, pp. 59-80.

[GOGU80] Goguen, J.A., "How to prove algebraic inductive Hypothesis without Induction, with Application to Correctness of Data Type Implementation", Proceeding of the 5th CADE, 1980.

[GOGU82] Goguen, J.A., Meseguer, J., "Compleetness of Many-Sorted Equational Logic", SRI, TR CSL-135, Sigpaln Notes, vol. 16, no. 7, 1982, pp. 24-32.

[GOGU84] Goguen, J.A., Burstall, R.M., "Introducing Institutions", Proceedings Logics of Programming Workshop, Springer LNCS, 164, Conergie-Mellow, 1984, pp. 221-256.

[GOGU84] Goguen, J.A., Burstall, R.M., "Some Fundamental Algebraic Tools for Semantics of Computation", Theoretical Computer Science, vol. 31, 1984, pp. 175-209.

[GOGU85a] Goguen, J.,A. Jouannaud, J.P., Meseguer, J., "Operational Semantics for order-sorted Algebras", Proceedings of the 12th ICALP, 1985.

[GOGU85b] Goguen, J.A., Meseguer, J., "Eqlog ", to appear in Functionnal and Logic Programming, ed DeGroot and Lindstrom, Prentice Hall, 1985.

[GOGU86c] Goguen, J.A., "Reusing and Implementing Software components", IEEE, 1986, pp. 16-32.

[GOGU86b] Goguen, J.A., "Thoughts on Specification, Design and Verification", Sigsoft SEN, vol. 5, no. 3, pp. 29-33.

[GOGU86c] Goguen, J.A., "A Study in the Foundation of Programming Methologdy : Specification, Institutions, Charters and Parchment", Proceedings Workshop on Categoru Theory and Computer Programming, Springer LNCS 240, Guildford, 1986, pp. 313-333.

[GUTT78] Guttag, H., Horning, J.J., "The Algebraic Specification of Abstract Data Types", Acta Informatica vol. 10, no.1, 1978, pp. 27-52.

[HENK77] Henkin, L., "The Logic of Equality", Mathematical Monthly, oct. 1977, pp. 597-612.

[HILG88] Hilger, J., "Langage de Production : description cohérente du court terme (première partie)", Rapports de Recherche INRIA, no 912 , 1988, p. 30.

[HUET80a] Huet, G., "Confluent Reductions", Journal of the Association for Computing Machinery, vol. 27, no 4, oct. 1980, pp. 791-821.

[HUET80b] Huet, G., Oppen, D., "Equations and Rewrite Rules : A Survey", Open Languages : Perspectives and Open Problems, éd. Book R., Academic Press, 1980, pp. 349-405.

[HUET81] Huet, G., "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm", 21st Symposium on Foundation of Computer Science, 1980, pp. 96-107.

[HUET82] Huet, G., Hullot, J.M., "Proofs by Induction in Equational Theories without Constructors", Journal of the Association for Computing Machinery, 25(2), 1982, pp. 239-266.

[JOUA84] Jouanaud, J.-P., Kirchner, H., "Completion of a Set of Rules Modulo a Set of Equations", Proceedings of the 11th Conference of Principles Programming Languages, 1984, pp. 83-92.

[JOUA83] Jouanaud, J.-P., Kirchner, C., Kirchner, H., "Incremental Construction of Unification Algorithms in Equational Theories", Lecture Notes in Computre Science, 1983, pp. 361-373.

[JOUA86a] Jouanaud, J.-P., "Automatic Proofs by Induction in Equationnal Theories Without Constructors", IEEE Symposium on Logic in Computer Science, Cambridge, juin 1986, pp. 358-366.

[JOUA86b] Jouanaud, J.-P., Kounalis, E. "Proofs by Induction in Equationnal Theories

Without Constructors", Proceedings of POPL, 1986.

[JOUA86c] Jouanaud, J.-P., Lescanne, P., "La Réécriture", Technique et Science Informatique, vol. 6, no. 6, 1986, pp. 433-452.

[JONE85] Jones, Sestort, Sondergaard, "An Experiment in Evaluation : The generation of a Compilor Generator", Proceedings of the 1st Conference on Rewriting techniques and Applications, Lecture Notes in Computer Science, Dijon, vol. 202 , mai 1985, pp.124-140.

[KAPL80] Kaplan, S., "Fair Conditionnal Term Rewriting Systems : Unification, Termination and Confluence",Theoretical Computer Science 56, 1988, pp. 37-57.

[KAPU86] Kapur, D., Musser, D.R., "Inductive Reasonning with Incomplete Specifications", IEEE Symposium on Logic in Computer Science, Cambridge, juin 1986, pp. 367-377.

[KIRC84] Kirchner, H., "A General Inductive Completion Algorithm and Application to Abstract Data Types ", Proceedings of the 7th International Conference on Automated Deduction, Lecture Notes in Computer Science, Napa Valley, California, USA, 1984, pp. 284-302.

[KIRC85a] Kirchner, C., Kirchner, H., Meseguer, J., "Operationnal Semantics of OBJ3", Rapport de Recherche CRIN, 87-R-087, Université Nancy I, 1987, p.19.

[KIRC85b] Kirchner, C., Kirchner, H., Megrelis, A., "OBJ for OBJ", Rapport de Recherche CRIN, 87-R-085, Université Nancy I, 1987, p.16.

[KIRC87] Kirchner, C., Lescanne, P., "Solving Disequations", Rapports de Recherche INRIA, no. 686, juin 1987.

[KIRC87] Kirchner, C., "Order-Sorted Equational Unification", Rapports de Recherche INRIA, no. 954 déc 1988.

[KNUT70] Knuth, D., Bendix, P., "Simple Word Problem in Universal Algebras", Computational Problems in Abstract Algebras, Pergamon Press, 1970.

[KOUN85a] Kounalis, E., "Completness in Data Type Specifications", Proceedings of the Eurocal'85 Conference, Springer Verlag, 1985, pp. 514-519.

[KOUN85b] Kounalis, E., "Validation des spécification algébriques par complétion", Thèse, Nancy I, 1985, p. 195.

[KRIE88] Krieger, M., Harvey, R.A., Lachance, J.P., "Process Activity Language "PAL" for Planning and Coordonnation of FMS", Proceedings Symposium on Manufacturing Application Languages, Winnipeg Manitoba, 1988, pp. 127-135.

[KROG87] Kröger, F., "Abstract Modules : Combining Algebraic and temporal Logic Specification means", Technique et Science Informatique, vol. 6, no. 6, 1987, pp. 559-573.

[LAZR87] Lazarek, A., Lescanne , P., Thiel, J.-J., "Proving Inductive Equalities Algorithms and Implementation", Rapports de Recherche INRIA, no. 682, juin 1987.

[LESC87] Lescanne, P., Heuillard, T., Dauchet, M., Tison, S., "Decidability of the Confluence of Ground Term Rewriting System", Rapports de Recherche INRIA, no. 675, juin 1987.

[LEVY80] Levy, N., Piganiol, A., Souquières, J., "Specifying with SACSO", Rapports de Recherche INRIA, no. 683, juin 1987.

[LISK74] Liskov, B., Zilles, S., "Programming with abstract data types", Proceedings of the ACM Sigplan Conference on very high level language, Sigplan Notices, 1974, PP. 50-59.

[LISK75] Liskov, B., Zilles, S., "An Introduction to Formal Specifications of Data Abstractions", Current Trends in Programming Methology, 1, éd. R.T. YEH , Prentice Hall, Englewood Cliffs, 1975.

[MANN74] Manna, Z., "Mathematical Theory of Computation", Computer Science Series, McGraw Hill, New York, NY, 1974.

[MART82] Martelli, A., Montanari, U., "An Efficient Unification Algorithm", ACM Transactions on Programming Languages and Systems, vol. 4, no. 2 , 1982, pp. 258-282.

[MESG82] Meseguer, J., Goguen, J.A.,"Initiality, Induction and Computability, in Application of Algebra to Language Definition and Compilation", M. Nivat and Reynolds J. (éds), Cambridge University Press, 1982.

[MESG82] Meseguer, J., Goguen, J.A.,"Initiality, Induction and Computability, in Application of Algebra to Language Definition and Compilation", M. Nivat and Reynolds J. (éds), Cambridge University Press, 1982.

[MOHA88] Mohan, C.K., Srivas, M.K.,"Conditional Specifications with Inequational Assumptions", Lecture Notes in Computer Science : 1st Ineternational Workshop, vol 308, Orsay, 1988, pp. 161-178.

[MUSS80b] Musser, D.L., "Abstract Data Type Specification in the AFFIRM System", IEEE Transaction on Software Engeneering, vol. 6, no1, Las Vegas, 1980, pp. 24-32.

[MOLL88] Möller, B., "Algebraic Specification with High-Order Operators", Proceedings IFIP TC Working Conference on Program Specification, Bad Tölz, avril 1986, Amsterdam North-Holland.

[MOLL88] Möller, B., Tarledi, A., Wirsing, M., "Algebraic Specification with Built-in Domain Constructions", Proceeding CAAP '88, pp. 1-17.

[PART88] Partsch, H., "Algebraic Requirement Definitions : a Case Study", Technique et Science Informatiques, vol. 1, 1986, pp. 21-36.

[PAUL88] Paul, E., "Proof by Induction in Equational Theories with Relations between Constructors", Proceedings of the 9th Colloquium on Trees in Algebra and Programming, Cambridge, University Press, Bordeaux, 1984, pp. 210-225.

[REIC87] Reichel, H., "Initial Computability, algebraïc Specifications and Partial Algebras", International Series of Monographs in Computer Science, Oxford-Claredon, 1987.

[REMY85] Remy, J., Zhang, H., "Contextual Rewriting", Proceedings of the 1st Conference on Rewriting techniques and Applications, Lecture Notes in Computer Science, vol. 202 , Dijon, 1985.

[REMY87] Remy, J.-L., "How to Characterize the Language of Ground Normal Forms", Rapports de Recherche INRIA, no. 676, juin 1987.

[SALO73] Salomaa, A., "Formal Languages", ACM Monographs Series, Academic Press, New York, NY, 1973.

[SALO85] Salomaa, A., "Computation and Automata", Cambridge University Press, Encyclopedia of Mathematics and its Applications vol. 25, 1985, p. 282.

[SMOL87a] Smoltka, G., Nutt, W., Goguen, J.A., Meseguer, J., "Order-Sorted Equationnal Computation", SEKI Report , SR-87-14, Universität Kaiserslautern, 1987, p.82.

[SMOL87b] Smoltka, G., "TEL Report and User Manual", SEKI Report, SR-87-11, Universität Kaiserslautern, 1987, p.107.

[SMOL87c] Smoltka, G., Ait-Kaci, H., "Inheritance Hierachies : Semantics and Unifications", MCC Technical Report Number AI-057-87, 1987, p. 28.

[TOMP80] Tompa, F.W., "A Practical Example of Specification of Abstract Data Types", Acta Informatica 13, 1980, pp. 205-224.

[THAT77] Thatcher, J.W., Wagner, E.G., Wright, J.B., "Specification of Abstract Data Types using conditional axioms", IBM Research Report RC-6214, pp. 1-17.

[THIE84] Thiel, J.J., "Stop Loosing Sleep over Incomplete Data Type Specifications", Proceedings of the 11th ACM Conference on Principles of Programming Languages, Salt Lake City, USA, 1984, pp. 76-82.

[THIER85] Thiery, O., "Langage de Spécification de Systèmes d'Information. Logiciel d'Aide à la Spécification de Systèmes d'Information", Thèse, Nancy I, 1985, p. 386.

[WEIN73] Weingarten, F.W.,"Translation of Computer Languages", Holden-Day Computer and Information Sciences, San Francisco, California, 1973, p.180.

[WING84] Wing, J.M.,"Helping Specifiers Evaluate their Specifications", Proceedings Congress CGL2,pp. 71-77.

[WIRS82] Wirsing, M., Broy, M., "An Analysis of Semantics of Models for Algebraic Specifications", Theoretical Foundations of Programming Methology, éd. Dordrech-Reidel, 1982, pp. 351-412.

[WIRS83] Wirsing, M., et al. "On Hierachies of Abstract Data Types", Acta Informatica 20, 1983, pp.1-33.

[WIRT76] Wirth, N.,"Algorithms + Data Structures = Programs", Series in Automatic Computation, Prentice Hall, Englewood Cliffs,1976,p.366.

[ZAVE74] Zave, P., "An Operationnal Approach to Requirement Specification for Embedded Systems", IEEE Transactions on Software Engeneering, vol. SE-8, no. 3, 1982, pp. 250-269.

[ZHAN85] Zhang, H., Remy, J.-L.,"Proceedings onf the 1st International Conference on Rewriting Techniques, Lecture Notes in Computer Science, vol. 202, Dijon, 1985, pp. 46-62.

[ZILL74] Zilles, S.N., "Algebraic Specifications of Data Types", Project Report 11, MIT, 1974, pp.28-52.

Annexe

Nous avons distingué dans la section I.3.1. quatre univers de types d'information qui, ensemble, composent le système d'information de la gestion du court terme et nous avons donné l'implémentation de la spécification de d'univers I. Dans cette annexe nous proposons une implémentation des univers II, III et IV sans pour autant spécifier toutes les contraintes sur les types d'information qu'il contiennent.

Univers II

L'univers II comporte la spécification des identificateurs

de stocks (module ID-STOCK),
de machines (module ID-MACHINE),
d'unités de transport (module ID-TRANSPORT),
de leurs outils (module ID-OUTIL) et
de leurs organes d'assistance (module ID-ORAGNE),
de competences (module ID-COMPETENCE),
d'individus (module ID-INDIVIDU),
des types de machines (module ID-TYPE-MACHINE),
des types d'unités de transport (module ID-TYPE-TRANSPORT),
des types d'outils (module ID-TYPE-OUTIL) et
des types d'organes d'assistance (module ID-TYPE-ORAGNE)

et la spécification de

- la sorte Tmachine (module TMACHINE) des types de machines, qui est composée d'un identificateur de type de machine et d'un ensemble d'identificateurs de machines du même type,

- la sorte TTransport (module TTRANSPORT) des types d'unités de transport, qui est composée d'un identificateur de type d'unités de transport et d'un ensemble d'identificateurs d'unités de transport du même type,

- la sorte Toutil (module TOUTIL) des types d'outils, qui est composée d'un identificateur de type d'outil et d'un ensemble d'identificateurs d'outils du même type,

• la sorte Torgane (module TORGANE) des types d'organes d'assistance, qui est composée d'un identificateur de type d'organes d'assistance et d'un ensemble d'identificateurs d'organes d'assistance du même type,

• la sorte Refstock (module REF-EN-STOCK), qui donne des lieux de stockage possibles de références, qui est composée d'un identificateur de référence et d'un ensemble d'identificateurs de stocks,

• la sorte Periode (module PERIODE), qui définit une période comme un couple d'entiers dont le premier entier indique le début de période et le second la fin de période,

• la sorte Individu (module INDIVIDU) qui caractérise un individu par un identificateur de d'individu, par l'ensemble de ses périodes de présence et par l'ensemble de ses compétences.

Univers III

L'univers III comporte la spécification

des identificateurs d'opérations de réglage (module ID-REGLAGE).

et la spécification de

• la sorte Opmach (module OPMACH), qui donne le nom de la machine sur laquelle une opération peut être exécutée. Elle se compose d'un identificateur d'opération et d'un identificateur de machine,

• la sorte paramétrée Modere (module MO-DE-RE), qui définit les opérations de réglage, de montage et de démontage d'outils et d'organes d'assistance nécessaires pour l'exécution d'une opération sur une machine, ainsi que les compétences et la durée qu'elle requiert. Elle se compose d'une sorte Opmach, d'un paramètre (type d'outil, type d'organe d'assistance ou identificateur d'operation de réglage), d'un ensemble d'identificateurs de compétences requises et d'une durée,

• la sorte Execution (module EXECUTION), qui définit l'ensemble des données nécessaires pour l'exécution d'une opération sur une machine. Elle est composée de la sorte Opmach, qui associe une opération à un type de machine, de cinq ensembles de

sortes Modere qui définissent l'ensemble des opérations de montage de types d'outils, l'ensemble des opérations de démontage de types d'outils, l'ensemble des opérations de montage de types d'organes d'assistance, l'ensemble des opérations de démontage de types d'organes d'assistance et l'ensemble des opérations de réglage, ainsi que la durée totale de l'opération.

Univers IV

L'univers IV comporte la spécification de

• la sorte Endpoint (module END-POINTS), qui définit les points extrêmes de distances que couvrent les unités de transport dans l'atelier. Un point extrême est un stock ou une machine,

• la sorte Distance (module DISTANCE), qui définit les distances entre points extrêmes,

• la sorte Passager (module PASSAGER), qui définit les objets qui peuvent être convoyés par les unités de transport. Ces objets sont des références, des outils ou des organes d'assistance,

• la sorte Transporter (module TRANSPORTER), qui définit l'opération de transport d'un objet d'un point extrême vers un autre par un type d'unité de transport, ainsi que la quantité à transporter,

• la sorte Exgam (module EX-GAMME), qui donne l'ensemble des éléments nécessaires pour l'exécution d'une gamme. Elle est composée de la description d'une gamme (sorte Gam de l'univers I), d'une liste de sortes Execution (description des données nécessaires pour l'exécution des différentes opérations de la gamme), d'une liste des opérations de transport qui s'effectuent avant et après chaque opération de la gamme.

```
obj ID-STOCK is

 sort Idstock .

 *** les identificateurs de stocks sont des operateurs d'arite 0
 op stock-a : -> Idstock .
 op stock-b : -> Idstock .
 op stock-c : -> Idstock .
 op stock-d : -> Idstock .
 op stock-e : -> Idstock .
 op stock-f : -> Idstock .
 op stock-g : -> Idstock .
 op stock-h : -> Idstock .
 op stock-i : -> Idstock .
 op stock-j : -> Idstock .
 op stock-k : -> Idstock .
 op stock-l : -> Idstock .
jbo


obj ID-MACHINE is

 sort Idmach .

 *** les identificateurs de machines sont des operateurs d'arite 0
 op mach-a : -> Idmach .
 op mach-b : -> Idmach .
 op mach-c : -> Idmach .
 op mach-d : -> Idmach .
 op mach-e : -> Idmach .
 op mach-f : -> Idmach .
 op mach-g : -> Idmach .
 op mach-h : -> Idmach .
 op mach-i : -> Idmach .
 op mach-j : -> Idmach .
 op mach-k : -> Idmach .
 op mach-l : -> Idmach .
jbo


obj ID-TRANSPORT is

 sort Idtrans .

 *** les identificateurs d'unites de transport sont des operateurs d'arite 0
 op trans-a : -> Idtrans .
 op trans-b : -> Idtrans .
 op trans-c : -> Idtrans .
 op trans-d : -> Idtrans .
 op trans-e : -> Idtrans .
 op trans-f : -> Idtrans .
 op trans-g : -> Idtrans .
 op trans-h : -> Idtrans .
 op trans-i : -> Idtrans .
 op trans-j : -> Idtrans .
 op trans-k : -> Idtrans .
 op trans-l : -> Idtrans .
jbo
```

```
obj ID-OUTIL is

 sort Idoutil .

 *** les identificateurs des outils sont des operateurs d'arite 0
 op outil-a : -> Idoutil .
 op outil-b : -> Idoutil .
 op outil-c : -> Idoutil .
 op outil-d : -> Idoutil .
 op outil-e : -> Idoutil .
 op outil-f : -> Idoutil .
 op outil-g : -> Idoutil .
 op outil-h : -> Idoutil .
 op outil-i : -> Idoutil .
 op outil-j : -> Idoutil .
 op outil-k : -> Idoutil .
 op outil-l : -> Idoutil .
jbo


obj ID-ORGANE is

 sort Idorg .

 *** les identificateurs des organes d'assistance sont des operateurs d'arite 0
 op org-a : -> Idorg .
 op org-b : -> Idorg .
 op org-c : -> Idorg .
 op org-d : -> Idorg .
 op org-e : -> Idorg .
 op org-f : -> Idorg .
 op org-g : -> Idorg .
 op org-h : -> Idorg .
 op org-i : -> Idorg .
 op org-j : -> Idorg .
 op org-k : -> Idorg .
 op org-l : -> Idorg .
jbo


obj ID-COMPETENCE is

 sort Idcomp .

 *** les identificateurs de competences sont des operateurs d'arite 0
 op comp-a : -> Idcomp .
 op comp-b : -> Idcomp .
 op comp-c : -> Idcomp .
 op comp-d : -> Idcomp .
 op comp-e : -> Idcomp .
 op comp-f : -> Idcomp .
 op comp-g : -> Idcomp .
 op comp-h : -> Idcomp .
 op comp-i : -> Idcomp .
 op comp-j : -> Idcomp .
 op comp-k : -> Idcomp .
 op comp-l : -> Idcomp .
jbo


obj ID-INDIVIDU is

 sort Idind .
```

```
 *** les identificateurs d'individus sont des operateurs d'arite 0
 op ind-a : -> Idind .
 op ind-b : -> Idind .
 op ind-c : -> Idind .
 op ind-d : -> Idind .
 op ind-e : -> Idind .
 op ind-f : -> Idind .
 op ind-g : -> Idind .
 op ind-h : -> Idind .
 op ind-i : -> Idind .
 op ind-j : -> Idind .
 op ind-k : -> Idind .
 op ind-l : -> Idind .
jbo


obj ID-TYPE-MACHINE is

 sort Idtmach .

 *** les identificateurs de types de machines sont des operateurs d'arite 0
 op tmach-a : -> Idtmach .
 op tmach-b : -> Idtmach .
 op tmach-c : -> Idtmach .
 op tmach-d : -> Idtmach .
 op tmach-e : -> Idtmach .
 op tmach-f : -> Idtmach .
 op tmach-g : -> Idtmach .
 op tmach-h : -> Idtmach .
 op tmach-i : -> Idtmach .
 op tmach-j : -> Idtmach .
 op tmach-k : -> Idtmach .
 op tmach-l : -> Idtmach .
jbo



obj ID-TYPES-TRANSPORT is

 sort Idttrans .

 *** les identificateurs des types d'unites de transport sont des operateurs d'arite 0
 op ttrans-a : -> Idttrans .
 op ttrans-b : -> Idttrans .
 op ttrans-c : -> Idttrans .
 op ttrans-d : -> Idttrans .
 op ttrans-e : -> Idttrans .
 op ttrans-f : -> Idttrans .
 op ttrans-g : -> Idttrans .
 op ttrans-h : -> Idttrans .
 op ttrans-i : -> Idttrans .
 op ttrans-j : -> Idttrans .
 op ttrans-k : -> Idttrans .
 op ttrans-l : -> Idttrans .
jbo



obj ID-TYPE-OUTIL is

 sort Idtoutil .

 *** les identificateurs des types d'outils sont des operateurs d'arite 0
 op toutil-a : -> Idtoutil .
 op toutil-b : -> Idtoutil .
```

```
  op toutil-c : -> Idtoutil .
  op toutil-d : -> Idtoutil .
  op toutil-e : -> Idtoutil .
  op toutil-f : -> Idtoutil .
  op toutil-g : -> Idtoutil .
  op toutil-h : -> Idtoutil .
  op toutil-i : -> Idtoutil .
  op toutil-j : -> Idtoutil .
  op toutil-k : -> Idtoutil .
  op toutil-l : -> Idtoutil .
jbo


obj ID-TYPE-ORGANE is

  sort Idtorg .

  *** les identificateurs des types d'organes d'assistance sont des operateurs d'arite 0
  op torg-a : -> Idtorg .
  op torg-b : -> Idtorg .
  op torg-c : -> Idtorg .
  op torg-d : -> Idtorg .
  op torg-e : -> Idtorg .
  op torg-f : -> Idtorg .
  op torg-g : -> Idtorg .
  op torg-h : -> Idtorg .
  op torg-i : -> Idtorg .
  op torg-j : -> Idtorg .
  op torg-k : -> Idtorg .
  op torg-l : -> Idtorg .
jbo



obj TMACHINE is

  protecting BOOL .
  protecting SET[ID-MACHINE] *(sort Set to Setm) .
  protecting ID-TYPE-MACHINE .

  sort Tmachine .

  op TYPEMACHINE[_//MACHINES{_}] : Idtmach Setm -> Tmachine .
  op machines_ : Tmachine -> Setm .
  op nomtmach_ : Tmachine -> Idtmach .
  op typer-machine__ : Idmach Setm -> Setm .
  op est-type__ : Tmachine Idmach -> Bool .

  var Tm : Tmachine .
  var Idm : Idmach .
  var Idtm : Idtmach .
  var Sm : Setm .

  eq nomtmach TYPEMACHINE[ Idtm //MACHINES{ Sm }] = Idtm .
  eq machines TYPEMACHINE[ Idtm //MACHINES{ Sm }] = Sm .
  eq typer-machine Idm machines Tm = Idm # machines Tm .
  eq est-type Tm Idm = Idm in machines Tm .
jbo



obj TTRANSPORT is

  protecting BOOL .
```

```
 protecting SET[ID-TRANSPORT] *(sort Set to Settr) .
 protecting ID-TYPES-TRANSPORT .

 sort TTransport .

 op TYPETRANSPORT[_//TRANSPORTS{_}] : Idttrans Settr -> TTransport .
 op transports_ : TTransport -> Settr .
 op nomttransp_ : TTransport -> Idttrans .
 op typer-machine__ : Idtrans Settr -> Settr .
 op est-type__ : TTransport Idtrans -> Bool .

 var TTr : TTransport .
 var Idt : Idtrans .
 var Idttr : Idttrans .
 var Sttr : Settr .

 eq nomttransp TYPETRANSPORT[ Idttr //TRANSPORTS{ Sttr }] = Idttr .
 eq transports TYPETRANSPORT[ Idttr //TRANSPORTS{ Sttr }] = Sttr .
 eq typer-machine Idt transports TTr = Idt # transports TTr .
 eq est-type TTr Idt = Idt in transports TTr .
jbo




obj REF-EN-STOCK is

 protecting BOOL .
 protecting SET[ID-STOCK] *(sort Set to Sets) .
 protecting IDREF .

 sort Refstock .

 op STOCKAGE[_//_] : Idref Sets -> Refstock .
 op stocker__ : Idstock Sets -> Sets .
 op nomref_ : Refstock -> Idref .
 op situe_ : Refstock -> Sets .
 op _est-situe-en_:  Idstock Refstock -> Bool .

 var RS : Refstock .
 var Ir : Idref .
 var Is : Idstock .
 var S : Sets .

 eq  nomref STOCKAGE[ Ir // S ] = Ir .
 eq  situe STOCKAGE[ Ir // S ] = S .
 eq  stocker Is situe RS = Is # (situe RS) .
 eq  Is est-situe-en RS = Is in situe RS .
jbo




obj TORGANE is

 protecting BOOL .
 protecting SET[ID-ORGANE] *(sort Set to Setr) .
 protecting SET[ID-STOCK] *(sort Set to Sets) .
 protecting ID-TYPE-ORGANE .

 sort Torgane .
```

```
  op TYPEORGANE[_//ORGANES{_}/STOCKS{_}] : Idtorg Setr Sets -> Torgane .
  op organes_ : Torgane -> Setr .
  op stocks_ : Torgane -> Sets .
  op nomtoutil_ : Torgane -> Idtorg .
  op situer-toutil__ : Idstock Sets -> Sets .
  op typer-outil__ : Idorg Setr -> Setr .
  op est-situe__ : Torgane Idstock -> Bool .
  op est-type__ : Torgane Idorg -> Bool .

  var Tg : Torgane .
  var Ids : Idstock .
  var Idg : Idorg .
  var Idtg : Idtorg .
  var Ss : Sets .
  var Sr : Setr .

  eq nomtoutil TYPEORGANE[ Idtg //ORGANES{ Sr }/STOCKS{ Ss }] = Idtg .
  eq organes TYPEORGANE[ Idtg //ORGANES{ Sr }/STOCKS{ Ss }] = Sr .
  eq stocks TYPEORGANE[ Idtg //ORGANES{ Sr }/STOCKS{ Ss }] = Ss .
  eq situer-toutil Ids stocks Tg = Ids # stocks Tg .
  eq typer-outil Idg organes Tg = Idg # organes Tg .
  eq est-situe Tg Ids = Ids in stocks Tg .
  eq est-type Tg Idg = Idg in organes Tg .
jbo



obj TOUTIL is

  protecting BOOL .
  protecting SET[ID-OUTIL] *(sort Set to Seto) .
  protecting SET[ID-STOCK] *(sort Set to Sets) .
  protecting ID-TYPE-OUTIL .

  sort Toutil .

  op TYPEOUTIL[_//OUTILS{_}/STOCKS{_}] : Idtoutil Seto Sets -> Toutil .
  op outils_ : Toutil -> Seto .
  op stocks_ : Toutil -> Sets .
  op nomtoutil_ : Toutil -> Idtoutil .
  op situer-toutil__ : Idstock Sets -> Sets .
  op typer-outil__ : Idoutil Seto -> Seto .
  op est-situe__ : Toutil Idstock -> Bool .
  op est-type__ : Toutil Idoutil -> Bool .

  var To : Toutil .
  var Ids : Idstock .
  var Ido : Idoutil .
  var Idto : Idtoutil .
  var Ss : Sets .
  var So : Seto .

  eq nomtoutil TYPEOUTIL[ Idto //OUTILS{ So }/STOCKS{ Ss }] = Idto .
  eq outils TYPEOUTIL[ Idto //OUTILS{ So }/STOCKS{ Ss }] = So .
  eq stocks TYPEOUTIL[ Idto //OUTILS{ So }/STOCKS{ Ss }] = Ss .
  eq situer-toutil Ids stocks To = Ids # stocks To .
  eq typer-outil Ido outils To = Ido # outils To .
  eq est-situe To Ids = Ids in stocks To .
  eq est-type To Ido = Ido in outils To .
jbo



obj PERIODE is
  protecting BOOL .
  protecting INT .
```

```
  sort Periode .

  op "_-_" : Int Int -> Periode .
  op valide_ : Periode -> Bool .
  op debut_ : Periode -> Int .
  op fin_ : Periode -> Int .
  op union__ : Periode Periode -> Periode .
  op inter__ : Periode Periode -> Periode .
  op recouvrement__ : Periode Periode -> Bool .
  op _est-contenu-dans_ : Periode Periode -> Bool .
  op egal-periode__ : Periode Periode -> Bool .

  vars p1 p2 p : Periode .
  vars td tf : Int .

  eq valide p = debut p < fin p .
  eq debut " td - tf " = td .
  eq fin " td - tf " = tf .
  eq recouvrement p1 p2 = if debut p1 > fin p2 or debut p2 > fin p1 then false else true fi .
  eq union p1 p2 =          if debut p1 > debut p2
                              then if fin p1 > fin p2
                                     then " debut p2 - fin p1 "
                                     else " debut p2 - fin p2 " fi
                              else if fin p1 > fin p2
                                     then " debut p1 - fin p1 "
                                     else " debut p1 - fin p2 " fi
                            fi .
  eq inter p1 p2 = if recouvrement p1 p2
                              then    if debut p1 > debut p2
                                        then if fin p1 > fin p2
                                               then " debut p1 - fin p2 "
                                               else " debut p1 - fin p2 " fi
                                        else if fin p1 > fin p2
                                               then " debut p2 - fin p2 "
                                               else " debut p2 - fin p2 " fi
                                        fi
                              else " 0 - 0 " fi .
  eq egal-periode p1 p2 = debut p1 == debut p2 and fin p1 == fin p2 .
  eq p1 est-contenu-dans p2 = if ((debut p1 > debut p2) or (debut p1 == debut p2))
                                        and ((fin p1 < fin p2) or (fin p1 == fin p2))
                                        then true
                                        else false
                               fi .
jbo


obj INDIVIDU is
  protecting SET[ID-COMPETENCE] *(sort Set to Setc) .
  protecting SET[PERIODE] *(sort Set to Setp) .
  protecting ID-INDIVIDU .

  sort Indiv .

  op INDIVIDU[_//COMPETENCE{_}/PRESENCE{_}] : Idind Setc Setp -> Indiv .
  op nomind_ : Indiv -> Idind .
  op competences_ : Indiv -> Setc .
  op presences_ : Indiv -> Setp .
  op est-competent-individu_pour_ : Indiv Idcomp -> Bool .
  op est-present-individu_pendant_ : Indiv Periode -> Bool .
  op _|c|_ : Idcomp Setc -> Setc .
  op _|p|_ : Periode Setp -> Setp .
  op _danspres_ : Periode Setp -> Bool .

  var Ind : Indiv .
  var Scomp : Setc .
  var Spres : Setp .
  var Idi : Idind .
```

```
  var Idc : Idcomp .
  vars p1 p2 p : Periode .

  eq nomind INDIVIDU[ Idi //COMPETENCE{ Scomp }/PRESENCE{ Spres }] = Idi .
  eq competences INDIVIDU[ Idi //COMPETENCE{ Scomp }/PRESENCE{ Spres }] = Scomp .
  eq presences INDIVIDU[ Idi //COMPETENCE{ Scomp }/PRESENCE{ Spres }] = Spres .
  eq p |p| presences Ind = p # presences Ind .
  eq Idc |c| competences Ind = Idc # competences Ind .
  eq p1 danspres (adj p2 presences Ind) =
        if p1 est-contenu-dans p2 then true else p1 danspres presences Ind fi .
  eq est-present-individu Ind pendant p = p danspres presences Ind .
  eq est-competent-individu Ind pour Idc = Idc in competences Ind .
jbo
```

Univers III

```
obj OPMACH is

  protecting IDOPE .
  protecting ID-TYPE-MACHINE .

  sort Opmach .

  op [EXEC_SUR_] : Idope Idtmach -> Opmach .
  op ope_ : Opmach -> Idope .
  op emach_ : Opmach -> Idtmach .

  var Ido : Idope .
  var Idtm : Idtmach .

  eq ope [EXEC Ido SUR Idtm ] = Ido .
  eq emach [EXEC Ido SUR Idtm ] = Idtm .
jbo
```

```
obj MO-DE-RE[X :: TRIV] is

  protecting SET[ID-COMPETENCE] *(sort Set to Setc) .
  protecting OPMACH .
  protecting INT .
  protecting ID-MACHINE .

  sort Modere .

  op MODERE[OPMACH_||CONCERNE_||NECESSITE_||DUREE_] : Opmach Elt Setc Int -> Modere .
  op quel-opmach_ : Modere -> Opmach .
  op quelle-ope_ : Modere -> Idope .
  op quelle-machine_ : Modere -> Idtmach .
  op quel-mdr_ : Modere -> Elt .
  op quel-comp_ : Modere -> Setc .
  op quelle-duree_ : Modere -> Int .
  op nec-comp __ : Modere Idcomp -> Bool .

  var Mdr : Modere .
  var Opm : Opmach .
  var D : Int .
  var x-mdr : Elt .
  var Io : Idope .
  var Ic : Idcomp .
```

```
 var Sc : Setc .

 eq quel-opmach MODERE[OPMACH Opm ||CONCERNE x-mdr ||NECESSITE Sc ||DUREE D ] = Opm .
 eq quelle-ope Mdr = ope quel-opmach Mdr .
 eq quelle-machine Mdr = emach quel-opmach Mdr .
 eq quel-mdr MODERE[OPMACH Opm ||CONCERNE x-mdr ||NECESSITE Sc ||DUREE D ] = x-mdr .
 eq quelle-duree MODERE[OPMACH Opm ||CONCERNE x-mdr ||NECESSITE Sc ||DUREE D ] = D .
 eq quel-comp MODERE[OPMACH Opm ||CONCERNE x-mdr ||NECESSITE Sc ||DUREE D ] = Sc .
 eq nec-comp Mdr Ic = Ic in quel-comp Mdr .
jbo



obj ID-REGLAGE is

 sort Idreg .

 *** les identificateurs de reglages sont des operateurs d'arite 0
 op reg-a : -> Idreg .
 op reg-b : -> Idreg .
 op reg-c : -> Idreg .
 op reg-d : -> Idreg .
 op reg-e : -> Idreg .
 op reg-f : -> Idreg .
 op reg-g : -> Idreg .
 op reg-h : -> Idreg .
 op reg-i : -> Idreg .
 op reg-j : -> Idreg .
 op reg-k : -> Idreg .
 op reg-l : -> Idreg .
jbo



obj EXECUTION is
 protecting BOOL .
 protecting INT .
 protecting MO-DE-RE[ID-REGLAGE] *(sort Set to Setreg) .
 protecting MO-DE-RE[TOUTIL] *(sort Set to Setmoto) .
 protecting MO-DE-RE[TOUTIL] *(sort Set to Setdeto) .
 protecting MO-DE-RE[TORGANE] *(sort Set to Setmotg) .
 protecting MO-DE-RE[TORGANE] *(sort Set to Setdetg) .
 protecting OPMACH .
 protecting ID-COMPETENCE .


 sort Execut .

 op EXECUTION[OPMACH_||SETMOTO_||SETDETO_||SETMOTG_||SETDETG_||SETREG_||DUREE_] :
                Opmach Setmoto Setdeto Setmotg Setdetg Setreg Int -> Execut .
 op quel-opm_ : Execut -> Opmach .
 op quel-moto_ : Execut -> Setmoto .
 op quel-deto_ : Execut -> Setdeto .
 op quel-motg_ : Execut -> Setmotg .
 op quel-detg_ : Execut -> Setdetg .
 op quel-reg_ : Execut -> Setreg .
 op duree_ : Execut -> Int .

 var E : Execut .
 var Opm : Opmach .
 var Smoto : Setmoto .
 var Sdeto : Setdeto .
 var Smotg : Setmotg .
 var Sdetg : Setdetg .
 var Sreg : Setreg .
 var D : Int .
```

```
 eq quel-opm EXECUTION  [
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = Opm .
 eq quel-moto EXECUTION[
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = Smoto .
 eq quel-deto EXECUTION[
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = Sdeto .
 eq quel-motg EXECUTION[
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = Smotg .
 eq quel-detg EXECUTION[
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = Sdetg .
 eq quel-reg EXECUTION   [
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = Sreg .
 eq duree EXECUTION      [
        OPMACH Opm ||SETMOTO Smoto ||SETDETO Sdeto ||SETMOTG Smotg ||SETDETG Sdetg ||SETREG Sreg ||DUREE D
                        ] = D .

jbo
```

Univers IV

```
obj END-POINTS is
 protecting ID-MACHINE .
 protecting ID-STOCK .

 sort Endpoint .

 subsorts Idstock < Endpoint .
 subsorts Idmach < Endpoint .

 op E : Idstock -> Endpoint .
 op E : Idmach -> Endpoint .
jbo


obj DISTANCE is
 protecting BOOL .
 protecting INT .
 protecting END-POINTS .

 sort Distance .

 op DISTANCE[entre_et_est_] : Endpoint Endpoint Int -> Distance .
 op dep_ :  Distance -> Endpoint .
 op arr_ :  Distance -> Endpoint .
 op dist_ :  Distance -> Int .

 vars x1 x2 : Endpoint .
 var d : Int .

 eq dep DISTANCE[entre x1 et x2 est d ] = x1 .
```

```
 eq arr DISTANCE[entre x1 et x2 est d ] = x2 .
 eq dist DISTANCE[entre x1 et x2 est d ] = d .
jbo


obj PASSAGER is

 protecting BOOL .
 protecting ID-OUTIL .
 protecting ID-ORGANE .
 protecting IDREF .

 sort Passager .

 subsorts Idoutil < Passager .
 subsorts Idref < Passager .
 subsorts Idorg < Passager .

 op P_ : Idoutil -> Passager .
 op P_ : Idref -> Passager .
 op P_ : Idorg -> Passager .
jbo



obj TRANSPORTER is

 protecting BOOL .
 protecting INT .
 protecting END-POINTS .
 protecting TTRANSPORT .
 protecting PASSAGER .

 sort Transporter .

 op TRANSPORTER[un_de_vers_par_en-quantite_] : Passager Endpoint Endpoint TTransport Int -> Transporter .
 op quoi_ : Transporter -> Passager .
 op dou_ : Transporter -> Endpoint .
 op versou_ : Transporter -> Endpoint .
 op par_ : Transporter -> TTransport .
 op qte_ : Transporter -> Int .

 var Tter : Transporter .
 var p : Passager .
 vars e1 e2 : Endpoint .
 var Tt : TTransport .
 var Q : Int .

 eq quoi TRANSPORTER[un p de e1 vers e2 par Tt en-quantite Q ] = p .
 eq dou TRANSPORTER[un p de e1 vers e2 par Tt en-quantite Q ] = e1 .
 eq versou TRANSPORTER[un p de e1 vers e2 par Tt en-quantite Q ] = e2 .
 eq par TRANSPORTER[un p de e1 vers e2 par Tt en-quantite Q ] = Tt .
 eq qte TRANSPORTER[un p de e1 vers e2 par Tt en-quantite Q ] = Q .
jbo



obj EX-GAMME is
 protecting OPMACH .
 protecting LISTE[EXECUTION] *(sort Liste to Listex).
 protecting LISTE[TRANSPORTER] *(sort Liste to Listta).
 protecting LISTE[TRANSPORTER] *(sort Liste to Listtp).
 protecting GAMME .

 sort Exgam .
```

```
op EXGAM[GAMME_||EXECUTIONS_||TRANSPORTS-AVANT_||TRANSPORT-APRES_] : Gam Listex Listta Listtp -> Exgam .
op gam_ : Exgam -> Gam .
op exec_ : Exgam -> Listex .
op trans-avant_ : Exgam -> Listta .
op trans-apres_ : Exgam -> Listtp .

var Le : Listex .
var La : Listta .
var Lp : Listtp .
var G : Gam .
var Ex : Exgam .

eq gam EXGAM[GAMME G ||EXECUTIONS Le ||TRANSPORTS-AVANT La ||TRANSPORT-APRES Lp ] = G .
eq exec EXGAM[GAMME G ||EXECUTIONS Le ||TRANSPORTS-AVANT La ||TRANSPORT-APRES Lp ] = Le .
eq trans-avant EXGAM[GAMME G ||EXECUTIONS Le ||TRANSPORTS-AVANT La ||TRANSPORT-APRES Lp ] = La .
eq trans-apres EXGAM[GAMME G ||EXECUTIONS Le ||TRANSPORTS-AVANT La ||TRANSPORT-APRES Lp ] = Lp .

jbo
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique