NORME INTERNATIONALE

ISO/CEI 10746-4

Première édition
1998-12-15

AMENDEMENT 1
2001-12-15

Technologies de l'information — Traitement réparti ouvert — Modèle de référence: Sémantique architecturale

AMENDEMENT 1: Formalisation de traitement

Information technology — Open Distributed Processing — Reference Model: Architectural semantics

AMENDMENT 1: Computational formalization

Numéro de référence
ISO/CEI 10746-4:1998/Amd.1:2001(F)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax. + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Version française parue en 2002

Imprimé en Suisse

TABLE DES MATIÈRES

Avant-propos

L'ISO (Organisation internationale de normalisation) et la CEI (Commission électrotechnique internationale) forment le système spécialisé de la normalisation mondiale. Les organismes nationaux membres de l'ISO ou de la CEI participent au développement de Normes internationales par l'intermédiaire des comités techniques créés par l'organisation concernée afin de s'occuper des domaines particuliers de l'activité technique. Les comités techniques de l'ISO et de la CEI collaborent dans des domaines d'intérêt commun. D'autres organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'ISO et la CEI participent également aux travaux. Dans le domaine des technologies de l'information, l'ISO et la CEI ont créé un comité technique mixte, l'ISO/CEI JTC 1.

Les Normes internationales sont rédigées conformément aux règles données dans les Directives ISO/CEI, Partie 3.

La tâche principale du comité technique mixte est d'élaborer les Normes internationales. Les projets de Normes internationales adoptés par le comité technique mixte sont soumis aux organismes nationaux pour vote. Leur publication comme Normes internationales requiert l'approbation de 75 % au moins des organismes nationaux votants.

L'attention est appelée sur le fait que certains des éléments du présent Amendement peuvent faire l'objet de droits de propriété intellectuelle ou de droits analogues. L'ISO et la CEI ne sauraient être tenues pour responsables de ne pas avoir identifié de tels droits de propriété et averti de leur existence.

L'Amendement 1 à la Norme internationale ISO/CEI 10746-4:1998 a été élaboré par le comité technique mixte ISO/CEI JTC 1, *Technologies de l'information*, sous-comité SC 7, *Ingénierie du logiciel*, en collaboration avec l'UIT-T. Le texte identique est publié en tant que Rec. UIT-T X.904/Amd.1.

NORME INTERNATIONALE

RECOMMANDATION UIT-T

TECHNOLOGIES DE L'INFORMATION – TRAITEMENT RÉPARTI OUVERT – MODÈLE DE RÉFÉRENCE: SÉMANTIQUE ARCHITECTURALE

AMENDEMENT 1

Formalisation de traitement

1) Avant-propos

Remplacer le 1er paragraphe de l'avant-propos

La présente Recommandation | Norme internationale fait partie intégrante du modèle de référence du traitement réparti ouvert (ODP, *open distributed processing*). Elle contient une formalisation des concepts de modélisation ODP définis dans les articles 8 et 9 de la Rec. UIT-T X.902 | ISO/CEI 10746-2. La formalisation est obtenue par l'interprétation de chaque concept en fonction des constructions des différentes techniques de description formelle normalisées.

par:

La présente Recommandation | Norme internationale fait partie intégrante du modèle de référence du traitement réparti ouvert (ODP, *open distributed processing*). Elle contient une formalisation des concepts de modélisation ODP définis dans les articles 8 et 9 de la Rec. UIT-T X.902 | ISO/CEI 10746-2 et dans l'article 7 (Language de traitement) de la Rec. UIT-T X.903 | ISO/CEI 10746-3. La formalisation est obtenue par l'interprétation de chaque concept en fonction des constructions des différentes techniques de description formelle normalisées.

2) Article 0 – Introduction

Remplacer la 4e énumération sous "Le modèle RM-ODP se compose"

– de la Rec. UIT-T X.904 | ISO/CEI 10746-4: **Sémantique architecturale:** elle contient une formalisation des concepts de modélisation ODP définis dans les articles 8 et 9 de la Rec. UIT-T X.902 | ISO/CEI 10746-2 et une formalisation des langages de point de vue définis dans la Rec. UIT-T X.903 | ISO/CEI 10746-3. La formalisation est obtenue par l'interprétation de chaque concept en fonction des constructions des différentes techniques de description formelle normalisées. Ce texte est normatif.

par

– de la Rec. UIT-T X.904 | ISO/CEI 10746-4: **Sémantique architecturale:** elle contient une formalisation des concepts de modélisation ODP définis dans les articles 8 et 9 de la Rec. UIT-T X.902 | ISO/CEI 10746-2 et une formalisation des langages de point de vue traitement définis dans la Rec. UIT-T X.903 | ISO/CEI 10746-3. La formalisation est obtenue par l'interprétation de chaque concept en fonction des constructions des différentes techniques de description formelle normalisées. Ce texte est normatif.

Remplacer le 4e paragraphe:

La présente Recommandation | Norme internationale a pour objet de fournir une sémantique architecturale pour les systèmes ODP, ce qui se traduit par une interprétation des concepts de modélisation de base et de spécification définis dans la Rec. UIT-T X.902 | ISO/CEI 10746-2 et des langages de point de vue définis dans la Rec. UIT-T X.903 | ISO/CEI 10746-3; elle utilise les diverses caractéristiques de différents langages de spécification formelle. Une sémantique architecturale est élaborée pour quatre différents langages de spécification formelle: LOTOS, ESTELLE, SDL et Z, ce qui conduit à une formalisation de l'architecture des systèmes ODP. Un processus d'élaboration itérative et de retour a permis d'améliorer la cohérence des Rec. UIT-T X.902 | ISO/CEI 10746-2 et UIT-T X.903 | ISO/CEI 10746-3.

par

La présente Recommandation | Norme internationale a pour objet de fournir une sémantique architecturale pour les systèmes ODP, ce qui se traduit par une interprétation des concepts de modélisation de base et de spécification définis dans la Rec. UIT-T X.902 | ISO/CEI 10746-2 et des langages de point de vue traitement définis dans la Rec. UIT-T X.903 | ISO/CEI 10746-3; elle utilise les diverses caractéristiques de différents langages de spécification formelle. Une sémantique architecturale est élaborée pour quatre différents langages de spécification formelle: LOTOS, ESTELLE, SDL et Z, ce qui conduit à une formalisation de l'architecture des systèmes ODP. Un processus d'élaboration itérative et de retour a permis d'améliorer la cohérence des Rec. UIT-T X.902 | ISO/CEI 10746-2 et UIT-T X.903 | ISO/CEI 10746-3.

3) Article 1 – Domaine d'application

Ajouter le paragraphe suivant à la fin du Domaine d'application

L'Annexe A indique une manière selon laquelle le langage de point de vue traitement de la Rec. UIT-T X.903 | ISO/CEI 10746-3 peut-être représenté dans les langages formels LOTOS, SDL, Z et Estelle. La présente Recommandation | Norme internationale fait également appel aux concepts définis dans la Rec. UIT-T X.902 | ISO/CEI 10746-2.

4) Article 2 – Références normatives

Changer la date de publication pour la Recommandation UIT-T Z.100 de (1993) en (1999).

ISO/CEI 13568:

Ajouter la référence suivante:

Z Notation, ISO/CEI JTC 1 SC 22 GT 19, Advanced Working Draft 2.C, 13 juillet 1999.

5) Paragraphe 3.2 – Termes définis dans la Recommandation UIT-T Z.100

Remplacer la liste des termes par la suivante:

active, adding, all, alternative, and, any, as, atleast, axioms, block, call, channel, comment, connect, connection, constant, constants, create, dcl, decision, default, else, endalternative, endblock, endchannel, endconnection, enddecision, endgenerator, endnewtype, endoperator, endpackage, endprocedure, endprocess, endrefinement, endselect, endservice, endstate, endsubstructure, endsyntype, endsystem, env, error, export, exported, external, fi, finalized, for, fpar, from, gate, generator, if, import, imported, in, inherits, input, interface, join, literal, literals, map, mod, nameclass, newtype, nextstate, nodelay, noequality, none, not, now, offspring, operator, operators, or, ordering, out, output, package, parent, priority, procedure, process, provided, redefined, referenced, refinement, rem, remote, reset, return, returns, revealed, reverse, save, select, self, sender, service, set, signal, signallist, signalroute, signalset, spelling, start, state, stop, struct, substructure, synonym, syntype, system, task, then, this, timer, to, type, use, via, view, viewed, virtual, with, xor.

6) Paragraphe 3.3 – Termes définis dans "The Z Base Standard"

Changer le titre du paragraphe par:

3.3 – Termes définis dans "The Z Notation"

Remplacer la liste des termes par la suivante:

affinement de données, affinement d'opération, théorie des schémas, composition de schéma, description axiomatique, occultation, remplacement, schéma (opération, état, encadrement), séquence, type.

7) Annexe A

Ajouter une nouvelle Annexe A comme suit:

Annexe A

Formalisation de traitement

A.1 Formalisation du langage de point de vue traitement en LOTOS

A.1.1 Concepts

La formalisation du langage de traitement en LOTOS utilise les concepts définis dans la formalisation des règles de modélisation et de structuration de base indiquées aux articles 8 et 9 de la Rec. UIT-T X.902 | ISO/CEI 10746–2.

Structures élémentaires associées aux interfaces opération et signal

Pour formaliser le langage de traitement en LOTOS, il est nécessaire d'introduire certaines structures élémentaires. Celles-ci comportent des paramètres susceptibles d'être associés à certaines interfaces de traitement ainsi qu'un modèle d'information de base susceptible d'être utilisé dans un flux.

Pour formaliser des paramètres il est nécessaire d'adopter deux concepts, à savoir celui de noms d'objets et celui de types d'objets. Les noms sont simplement des étiquettes. Comme nous allons le voir, du point de vue traitement, ces étiquettes doivent faire l'objet de contrôles, visant par exemple à en vérifier l'égalité, au moment où les interfaces sont créées. En règle générale, nous pouvons représenter les noms comme suit:

```
type Name is Boolean
        sorts Name
        opns    newName: -> Name
                anotherName: Name -> Name
                _eq_, _ne_: Name, Name -> Bool
endtype (* Name *)
```


Par souci de concision, nous faisons l'impasse sur les équations, qui ne sont pas censées poser de problème particulier. Il est possible d'être ici plus normatif, par exemple en utilisant des chaînes de caractères de la bibliothèque LOTOS. La seule chose qui nous intéresse en ce qui concerne les noms est que nous puissions en déterminer l'égalité ou l'inégalité.

Comme indiqué dans la présente Recommandation | Norme internationale, un type selon la terminologie ODP ne peut pas être interprété directement dans la partie "algèbre de processus" du langage LOTOS. Il est toutefois possible de modéliser des types à l'aide de la partie "*Act One*" du LOTOS. Bien qu'elle ait été expressément conçue pour représenter des types, la partie *Act One* est malheureusement limitée dans ses modalités de vérification des types et des relations entre les types. Par exemple, il est impossible de vérifier le sous-typage ou l'équivalence, voire l'isomorphisme entre les types, du fait que l'égalité de types en *Act One* est fondée sur l'équivalence de noms de sortes. A l'appui de notre raisonnement, nous introduisons ici une notion élémentaire de types qui nous permet de tester l'égalité, l'inégalité et le sous-typage.

```
type AnyType is  Boolean
      sorts AnyType
      opns newType: -> AnyType
           anotherType: AnyType -> AnyType
           _eq_,_isSubtype_: AnyType, AnyType -> Bool
endtype (* AnyType *)
```

Un paramètre est une relation entre un nom et sa représentation de type sous-jacente. Un paramètre peut donc être représenté comme suit:

```
type Param is Name, AnyType
        sorts Param
        opns newParam: Name, AnyType -> Param
             _eq_,_ne_,_isSubtype_: Param, Param -> Bool
endtype (* Param *)
```

Comme précédemment, nous devons procéder à des contrôles de l'égalité ou de l'inégalité des paramètres ainsi que dans le cas où un paramètre est un sous-type d'un autre paramètre. Deux paramètres ont une relation de sous-typage lorsque leurs types ont une relation de sous-typage. Il est par ailleurs utile pour nous d'introduire des séquences de ces paramètres.

```
type PList is String actualizedby Param
        using sortnames PList for String Param for Element Bool for FBool
        opns _isSubtype_: PList, PList -> Bool
endtype (* PList *)
```

Nous utilisons ici le type *String* de la bibliothèque LOTOS, actualisé avec le type *Param* défini précédemment. En outre, nous incluons ici une opération *isSubtype* permettant de vérifier si une séquence de paramètres est un sous-type d'une autre. Une liste de paramètres est un sous-type d'une autre liste lorsque tous les paramètres qu'elle contient sont des sous-types de ceux qui figurent dans la première liste. En outre, les paramètres devraient occuper la même position dans leurs listes respectives. Il convient de noter que ces paramètres sont susceptibles de contenir des références aux interfaces utilisées pour limiter les interactions qui peuvent se produire. Alors qu'il est tout à fait possible de modéliser une interface dans l'algèbre de processus, il est impossible de modéliser une référence à cette interface dans l'algèbre de processus qui, pour ainsi dire, s'approprie la fonctionnalité de cette interface. Pour surmonter cette difficulté, nous modélisons les références d'interface en *Act One*. Etant donné qu'une référence d'interface reproduit, entre autres choses, la signature de l'interface, nous fournissons un modèle de signatures en *Act One* pour les opérations. Les opérations se composent d'un nom, d'une séquence d'entrées et éventuellement d'une séquence de sorties. Pour simplifier les choses, nous ne nous soucions pas ici de savoir si l'opération est en notation infixée, préfixée ou suffixée. Cela peut-être représenté par le fragment LOTOS suivant:

```
type Op is Name, PList
        sorts Op
        opns makeOp: Name, PList -> Op
             makeOp: Name, PList, PList -> Op
             getName: Op -> Name
             getInps: Op -> PList
             getOuts: Op -> PList
             _eq_: Op, Op -> Bool
        eqns forall op1,op2: Op, n: Name; pl1, pl2: PList
ofsort Name     getName(makeOp(n,pl1,pl2)) = n;
ofsort PList    getInps(makeOp(n,pl1)) = pl1;
                        getInps(makeOp(n,pl1,pl2)) = pl1;
                        getOuts(makeOp(n,pl1)) = <>;
                        getOuts(makeOp(n,pl1,pl2)) = pl2;
ofsort Bool     op1 eq op2 = ((getName(op1) eq getName(op2)) and
                              (getInps(op1) isSubtype getInps(op2)) and
                              (getOuts(op2) isSubtype getOuts(op1)));
endtype (* Op *)
```


Le fait de disposer d'une méthode permettant de déterminer si deux opérations sont identiques ramène le problème du sous-typage entre types abstraits de données à une comparaison d'ensembles, dont les éléments sont les opérations créées. Ainsi, un serveur est un sous-type d'un autre serveur s'il prend en charge toutes les opérations de cet autre serveur. A noter ici que nous modélisons deux formes d'opérations: celle dont on n'attend pas de résultats et celle dont on en attend. En outre, nous introduisons des ensembles de ces opérations.

```
type OpSet is Set actualizedby Op
        using sortnames OpSet for Set Op for Element Bool for FBool
endtype (* OpSet *)
```

Cela étant, une référence d'interface peut être représentée par le fragment LOTOS suivant:

```
type IRef is OpSet
        sorts IRef
        opns makeIRef     : OpSet -> IRef
             NULL         : -> IRef
             getOps       : IRef -> OpSet
             _eq_         : IRef, IRef -> Bool
        eqns forall o: OpSet; ir1, ir2: IRef
             ofsort OpSet  getOps(makeIRef(o)) = o;
             ofsort Bool   ir1 eq ir2 = getOps(ir1) eq getOps(ir2) ;
endtype (* IRef *)
```

Nous constatons ici que l'égalité des références d'interface est fondée uniquement sur les opérations contenues dans la référence considérée. Elle pourrait parfaitement être étendue à d'autres aspects, tels que l'emplacement de l'interface ou les contraintes qui en restreignent l'utilisation. En outre, nous introduisons des ensembles de ces références d'interface.

```
type IRefSet is Set actualizedby IRef
        using sortnames IRefSet for Set IRef for Element Bool for FBool
endtype (* IRefSet *)
```

Structures élémentaires associées aux interfaces flux

Le langage de point de vue traitement de la Recommandation UIT-T X.903 | ISO/CEI 10746–3 examine en outre les interfaces relatives au flux continu de données, par exemple multimédias. Ces interfaces sont appelées *interfaces flux*. Les interfaces flux contiennent des ensembles finis de flux. Ces flux peuvent provenir de l'interface (flux producteur) ou se diriger vers l'interface (flux consommateur). Chaque flux est modélisé au moyen d'un modèle d'action. Chaque modèle d'action contient le nom du flux, le type du flux ainsi qu'une indication de causalité pour ce flux.

Le point de vue traitement s'éloigne du contenu du flux d'informations proprement dit. Nous retenons ici un principe générique du flux d'informations selon lequel ce flux est représenté par une séquence d'éléments de flux. Un élément de flux peut être considéré comme un élément particulier du flux d'informations. Nous constatons ici que les flux sont considérés du point de vue traitement comme des actions continues. Dans le modèle que nous présentons ici, les flux sont représentés comme des séquences d'événements temporels discrets. D'un côté cela nous permet de remédier aux problèmes de synchronisation des flux d'informations, même si cela se traduit par la perte du caractère continu des flux.

Chaque élément d'un flux d'information peut être considéré comme une unité composée de données (qui peuvent être comprimées) et que nous représentons par *Données*. Ce modèle peut indiquer comment les informations ont été comprimées, quelles informations ont été comprimées, etc. En tant que tel, ce modèle n'est pas examiné plus avant ici. De plus, les éléments de flux contiennent un champ d'indication horaire servant à modéliser l'heure à laquelle l'élément de flux considéré a été envoyé ou reçu. En outre, il arrive souvent, dans le cas de flux multimédias, que des éléments de flux de synchronisation soient nécessaires, par exemple pour synchroniser le son avec l'image. C'est pourquoi nous associons un nom donné (*Name*) à chaque élément de flux. Ce nom peut ensuite servir à sélectionner un élément de flux donné, selon les besoins. Il en découle que nous pouvons modéliser un élément de flux comme suit:

```
type FlowElement is Name, NaturalNumber, Data, Param
sorts FlowElement
opns makeFlowElement: Data, Nat, Name -> FlowElement
     nullFlowElement  : -> FlowElement
     getData    : FlowElement -> Data
     getTime    : FlowElement -> Nat
     getName    : FlowElement -> Name
     toParam    : FlowElement -> Param
     setTime    : Nat, FlowElement -> FlowElement
eqns forall d: Data, s,t: Nat, n: Name
         ofsort Data      getData(makeFlowElement(d,t,n)) = d;
         ofsort Nat       getTime(makeFlowElement(d,t,n)) = t;
         ofsort Name      getName(makeFlowElement(d,t,n)) = n;
         ofsort FlowElement       setTime(s,makeFlowElement(d,t,n)) = makeFlowElement(d,s,n);
endtype (* FlowElement *)
```


Il convient de noter ici que nous modélisons le temps sous la forme d'un nombre naturel. Toutefois, les valeurs de modélisation retenues pourraient très bien aussi être exprimées en temps réel (continu) ou en intervalles de temps. Pour simplifier les choses, nous nous bornons ici à représenter le temps discret sous la forme d'un nombre naturel. Par ailleurs, nous introduisons une opération permettant de convertir un élément de flux en un paramètre. Pour simplifier les choses, nous faisons l'impasse sur les équations associées. En outre, nous introduisons des séquences de ces éléments de flux:

```
type FlowElementSeq is FlowElement
        sorts FlowElementSeq
        opns makeFlowElementSeq: -> FlowElementSeq
             addFlowElement: FlowElement, FlowElementSeq -> FlowElementSeq
             remFlowElement: FlowElement, FlowElementSeq -> FlowElementSeq
             getFlowElement: Name, FlowElementSeq -> FlowElement
             timeDiff: FlowElement, FlowElement -> Nat
        eqns forall f1, f2: FlowElement, fs: FlowElementSeq, n1,n2: Name
        ofsort FlowElementSeq
        getTime(f1) le getTime(f2) =>
                addFlowElement(f1,addFlowElement(f2,makeFlowElementSeq)) =
                                        addFlowElement(f2,makeFlowElementSeq);
        ofsort FlowElement
                getFlowElement(n1,makeFlowElementSeq) = nullFlowElement;
                n1 ne n2 =>
                        getFlowElement(n1,addFlowElement(makeFlowElement(d,t,n2),fs)) =
                                getFlowElement(n1,fs);
                n1 eq n2 =>
                        getFlowElement(n1,addFlowElement(makeFlowElement(d,t,n2),fs)) =
                                makeFlowElement(d,t,n2);
endtype (* FlowElementSeq *)
```

Par souci de concision, nous ne reproduisons pas toutes les équations. Les éléments de flux sont ajoutés à la séquence pour autant qu'ils présentent des valeurs de champ d'indication horaire croissantes. Une opération permettant de traverser une séquence d'éléments de flux pour trouver un élément de flux nommé est prévue. En outre, nous introduisons une opération permettant de calculer les différences horaires entre les champs d'indication horaire de deux éléments de flux. Le recours à cette opération permet de préciser, par exemple, que tous les éléments de flux d'une séquence sont séparés par des champs d'indication horaire égaux. Nous sommes alors en présence d'un flux isochrone. En outre, nous introduisons des ensembles de ces séquences d'éléments de flux:

```
type FlowElementSeqSet is Set actualizedby FlowElementSeq
        using sortnames FlowElementSeqSet for Set FlowElementSeq for Element Bool for FBool
endtype (* FlowElementSeqSet *)
```

A.1.1.1 Signal

Aucun élément propre au LOTOS ne permet de faire la distinction entre un signal, un flux et une opération. Il peut arriver, toutefois, qu'un style de LOTOS permette de faire la distinction entre des signaux, des flux et des opérations. Par exemple, tous les signaux pourraient avoir des formats analogues pour leurs offres d'événement. Un exemple d'un format possible côté serveur d'un signal est représenté dans le fragment LOTOS suivant:

```
<g> ?<sigName: Name> !<myRef> ?<inArgs: PList>;
```

Ici et dans le reste du paragraphe A.1, nous adoptons la notation selon laquelle <*X*> tient lieu de paramètre fictif pour un *X*, c'est-à-dire que *g*, *sigName*, *myRef* et *inArgs* tiennent lieu respectivement de paramètres fictifs pour la porte, le nom du signal, la référence d'interface associée au serveur offrant ce signal et les paramètres associés au signal.

Un exemple de format possible côté client d'un signal est représenté dans le fragment LOTOS suivant:

```
<g> !<sigName> !<SomeIRef> !<inArgs>;
```

Ici le côté client du signal contient une porte (*g*), une étiquette pour le nom du signal (*sigName*), une référence désignant l'objet auquel le signal est envoyé (*SomeIRef*) et les paramètres associés au signal (*inArgs*). Nous verrons au § A.1.1.11 comment ces offres d'événement peuvent être utilisées pour créer des signatures d'interface signal.

A.1.1.2 Opération

Occurrence d'une interrogation ou d'une annonce.

A.1.1.3 Annonce

Interaction consistant en une seule invocation. Pour les raisons indiquées au § A.1.1.1, on ne peut utiliser qu'une convention de modélisation informelle pour modéliser les annonces. Un exemple de cette règle pour le côté client d'une annonce pourrait être représenté comme suit:

```
<g> !<invName> !<SomeIRef> !<inArgs>;
```

Le côté serveur d'une annonce pourrait être représenté comme suit:

```
<g> ?<invName: Name> !<myRef>  ?<inArgs: PList>;
```

Les structures de données sont ici analogues à celles qui sont décrites au § A.1.1.1. Nous verrons au § A.1.1.12 comment ces offres d'événement peuvent être utilisées pour créer des parties de signature d'interface opération.

A.1.1.4 Interrogation

Invocation entre un client et un serveur, suivie d'une ou plusieurs terminaisons entre ce serveur et ce client. Toutefois, pour les raisons indiquées au § A.1.1.1, on ne peut utiliser qu'une convention de modélisation informelle pour modéliser les interrogations. Un exemple de cette règle pour le côté client d'une interrogation pourrait être représenté comme suit:

```
<g> !<invName> !<SomeIRef> !<inArgs> !<outArgs>;
( <g> ?<termName:Name> !<myRef> ?<outArgs: PList>;  (* ... other behaviour *)
  [ ]  (* ... other terminations *))
```

Ici *termName* représente les noms de terminaison et *outArgs* représente les paramètres de sortie. Le côté serveur d'une interrogation pourrait être représenté comme suit:

```
<g> ?<invName: Name> !<myRef>  ?<inArgs: PList> ?<outArgs: PList>;
( <g> !<termName> !<SomeIRef>  !<outArgs>;  (* ... other behaviour *)
  [ ] (* ... other terminations *))
```

Les autres structures de données considérées ici sont analogues à celles qui sont décrites au § A.1.1.1. Nous verrons au § A.1.1.12 comment ces offres d'événement peuvent être utilisées pour créer des parties de signatures d'interface opération.

A.1.1.5 Flux

Abstraction d'une séquence d'interaction, entre un objet producteur et un objet consommateur donnant lieu à l'acheminement d'informations. Pour les raisons indiquées au § A.1.1.1, les flux ne peuvent être représentés qu'en LOTOS à l'aide de conventions de modélisation informelle. Les flux sont souvent soumis à des exigences temporelles strictes. Une manière d'appliquer cette règle à la production de flux consisterait par exemple à recourir à un processus paramétré par une séquence de structures de données à envoyer, par exemple des éléments de flux qui peuvent être datés au moment où ils sont envoyés. Une manière simple de modéliser cette règle en LOTOS serait par exemple la suivante:

```
process ProduceAction[ g, ...](... toSend: FlowElementSeq, tnow: Nat, rate: Nat ...):noexit:=
        g !<flowName> !<SomeIRef> !<SetTime(tnow+rate,head(toSend))>;
        (*... other behaviour and recurse with FlowElement removed from toSend *)
endproc (* ProduceAction *)
```

Ici les éléments de flux sont envoyés ensemble avec l'indication de l'heure (locale) du moment considéré et de la cadence à laquelle les éléments de flux doivent être produits.

La consommation des éléments de flux est généralement soumise à des exigences différentes. La nécessité de surveiller continuellement les champs d'indication horaire des flux d'information entrants revêt une importance particulière. La consommation d'un flux d'information peut être représentée de manière simple comme suit:

```
process ConsumeAction[ g,...](myRef: IRef, recFlowElements: FlowElementSeq, tnow, rate: Nat...) :noexit:=
        g ?<flowName: Name> !myRef ?<inFlowElement: FlowElement>;
        (* check temporal requirements of inFlowElement are satisfied then *)
        (* display FlowElement and recurse with time incremented *)
        (* or recurse with FlowElement added to received FlowElements and time incremented *)
endproc (* ConsumeAction *)
```

A.1.1.6 Interface signal

Comme il n'existe aucun moyen direct, en LOTOS, d'établir une distinction formelle entre un signal et tout autre événement LOTOS, on ne peut déterminer qu'une interface donnée est une interface signal que de manière informelle en modélisant les événements LOTOS utilisés pour représenter les signaux différemment de tout autre événement. Un exemple de la manière dont une signature d'interface signal pourrait être modélisée en LOTOS est donné au § A.1.1.11.

A.1.1.7 Interface opération

Comme il n'existe aucun moyen direct, en LOTOS, d'établir une distinction formelle entre une opération et tout autre événement LOTOS, on ne peut déterminer qu'une interface donnée est une interface opération que de manière informelle en modélisant les événements LOTOS utilisés pour représenter les opérations différemment de tout autre événement. Un exemple de la manière dont une signature d'interface opération pourrait être modélisée en LOTOS est donné au § A.1.1.12.

A.1.1.8 Interface flux

Comme il n'existe aucun moyen direct, en LOTOS, d'établir une distinction formelle entre un flux et tout autre événement LOTOS, on ne peut déterminer qu'une interface donnée est une interface de flux que de manière informelle en modélisant les événements LOTOS utilisés pour représenter les flux différemment de tout autre événement. Un exemple de la manière dont une signature d'interface flux pourrait être modélisée en LOTOS est donné au § A.1.1.13.

A.1.1.9 Modèle d'objet de traitement

En LOTOS, un modèle d'objet de traitement est représenté par une définition de processus à laquelle est associé un ensemble de modèles d'interface de traitement que l'objet peut instancier et par une spécification de comportement, c'est-à-dire une expression de comportement qui n'est pas composée d'événements modélisés sous forme de signatures de signal, de signatures de flux ou de signatures d'opération. Par ailleurs, une forme quelconque de contrat d'environnement devrait être modélisée dans le cadre de la définition de processus. Toutefois, le LOTOS n'est pas doté de toutes les fonctions nécessaires à la modélisation intégrale de contrats d'environnement. Il est possible de modéliser certaines fonctions d'un contrat d'environnement à l'aide d'un type de donnée *Act One*. Ce type doit être indiqué sous la forme d'un paramètre formel dans la liste des paramètres de valeur de la définition de processus.

A.1.1.10 Modèle d'interface de traitement

Modèle d'interface signal, modèle d'interface flux ou modèle d'interface opération.

A.1.1.11 Signature d'interface signal

En LOTOS, une signature d'interface signal est représentée par une définition de processus, de telle sorte que toutes les offres d'événement qui doivent être synchronisées avec l'environnement pour se produire soient modélisées sous forme de signatures de signal. L'occurrence de ces offres d'événement se traduit par l'établissement d'une communication unidirectionnelle entre un objet initiateur et un objet répondeur. Sur le plan structurel, une signature de signal est analogue à une invocation d'annonce (ou de terminaison associée à une interrogation), c'est-à-dire qu'elle se compose d'un nom (pour le signal), d'une séquence de paramètres associés au signal et d'une indication de causalité. Comme tous les événements en LOTOS sont atomiques, il n'existe aucune distinction intrinsèque entre les événements modélisés sous forme d'annonces et les événements modélisés sous forme de signaux.

Les signatures d'interface signal diffèrent toutefois des signatures d'interface opération en ceci qu'elles ne requièrent pas qu'une causalité soit conférée à toute l'interface. Les signatures d'interface signal peuvent au contraire contenir des signaux avec des causalités d'initiateur ou de répondeur. Il s'ensuit que nous pouvons modéliser une signature d'interface signal en LOTOS comme suit:

```
process SignalIntSig[ g... ](myRef: IRef, known: IRefs...):noexit:=
 g !<sigName> !<SomeIRef> !<pl>;                  ...(* other behaviour *)
   [ ]... (* other initiating actions *)
 [ ]
 g ?<sigName: Name> !myRef ?<inArgs: PList>;
          ([ not(makeOp(sigName,inArgs) IsIn getOps(myRef))] -> ...(* unsuccessful behaviour *)
          [ ]
          [ makeOp(sigName,inArgs) IsIn getOps(myRef) ] ->    ...(* successful behaviour *) )
 [ ]... (* other responding actions *)
endproc (* SignalIntSig *)
```

Nous constatons ici qu'une interface signal se compose d'un ensemble d'offres d'événement. Ces offres d'événement peuvent modéliser des signaux sortants, pour celles d'entre elles qui comportent un point d'exclamation (**!**) précédant le nom du signal et la liste de paramètres, ou des signaux entrants, pour celles d'entre elles qui comportent un point d'interrogation (**?**) précédant le nom du signal et la liste de paramètres. Dans le cas de signaux entrants, il est possible de vérifier que le signal entrant est bien un des signaux qui sont attendus, c'est-à-dire qu'il fait partie de l'ensemble de signaux autorisés associés à cette référence d'interface.

NOTE – Ce fragment de spécification requiert que le processus soit instancié avec au moins une porte qui corresponde au point d'interaction auquel l'interface se situe. Le processus devrait aussi être instancié avec un ensemble de références d'interface et sa propre référence d'interface. Nous constatons ici qu'il est impossible d'écrire des prédicats sur les signaux envoyés. Il faudrait pour cela un niveau d'autorité que nous n'avons pas, garantissant par exemple que *SomeIRef* soit une des références d'interfaces figurant dans l'ensemble de références d'interface connues associé au processus. Il est toutefois possible de soumettre les signaux entrants à des contrôles, le but étant de vérifier que le signal entrant est bien un des signaux associés à cette référence d'interface. Nous notons en outre que nous avons utilisé ici l'opérateur choix pour modéliser la composition des différents signaux. Il est tout à fait possible d'utiliser ici plusieurs autres opérateurs de composition, par exemple l'opérateur d'entrelacement. L'utilisation de la composition avec entrelacement permet la réception de plusieurs signaux entrants avant qu'un signal de réponse ne soit envoyé. Les interfaces étant généralement matérialisées sous une forme quelconque, c'est-à-dire qu'elles offrent des opérations qui peuvent être invoquées plusieurs fois, les commentaires représentant d'autres comportements sont susceptibles de contenir des instanciations de processus récursif. L'utilisation de l'opérateur de choix nous offre une forme de blocage des signaux, c'est-à-dire qu'en cas d'arrivée d'un signal nous devrons d'abord y répondre avant de pouvoir accepter un nouveau signal. Des arguments analogues s'appliquent à tous les autres processus représentant des signatures d'interface de traitement.

A.1.1.12 Signature d'interface opération

En LOTOS, une signature d'interface opération est représentée par une définition de processus, de telle sorte que toutes les offres d'événement qui doivent être synchronisées avec l'environnement pour se produire soient modélisées sous la forme d'une partie de signatures d'opération, en d'autres termes qu'elles représentent toutes des parties d'annonce ou d'interrogation. Nous pouvons modéliser une signature d'interface opération pour un client selon la définition de processus suivante:

```
process OpIntSigClient[ g... ](myRef: IRef, known: IRefs, ...):noexit:=
  g !<invName> !<SomeIRef> !<inArgs>;               ...(* other behaviour *)
   [ ]... (* other announcements *)
  [ ]
  g !<invName> !<SomeIRef> !<inArgs> !<outArgs>;        ...(* other behaviour *)
          (g ?<termName: Name> !myRef ?<outArgs: PList>;
            [ not(makeOp(termName,outArgs) IsIn getOps(myRef))] -> ...(* return error message *)
            [ ]
            [ makeOp(termName,outArgs) IsIn getOps(myRef)] ->    ...(* other behaviour *)
          [ ] ... (* other terminations *))
   [ ] ... (* other interrogations *)
endproc (* OpIntSigClient *)
```

Nous constatons ici qu'une signature d'interface client se compose d'un ensemble d'offres d'événement. Ces offres d'événement peuvent modéliser soit des invocations (annonce ou interrogation) sortantes, pour celles de ces offres qui comportent un point d'exclamation (**!**) précédant le nom de l'invocation et la liste de paramètres, soit des terminaisons entrantes, pour les offres d'événement qui comportent un point d'interrogation (**?**) précédant le nom de la terminaison et la liste de paramètres. Dans le cas de terminaisons entrantes, il est possible de vérifier que la terminaison entrante est bien une des terminaisons prévues, c'est-à-dire qu'elle fait partie de la série de terminaisons autorisées et associées à cette référence d'interface.

La Note du § A.1.1.11 s'applique également aux signatures d'interface opération, sous réserve d'en modifier dûment le texte (remplacement de l'expression "signal entrant" par "invocation", par exemple).

Les signatures d'interfaces opération pour les serveurs peuvent être représentées en LOTOS comme suit:

```
process OpIntSigServer[ g... ](myRef: IRef, known: IRefs, ...):noexit:=
  g ?<invName: Name> !myRef ?<inArgs: PList>;
   ([ not(makeOp(invName,inArgs) IsIn getOps(myRef))] ->        ...(* ignore/other behaviour *)
    [ ]
    [ makeOp(invName,inArgs) IsIn getOps(myRef) ] ->            ...(* other behaviour *)
    [ ]... (* other announcements *))
 [ ]
  g ?<invName: Name> !myRef ?<inArgs:PList> ?<outArgs:PList>;    ...(* other behaviour *)
   ([ not(makeOp(invName,inArgs,outArgs) IsIn getOps(myRef))] -> ...(* return error message *)
    [ ]
    [ makeOp(invName,inArgs,outArgs) IsIn getOps(myRef) ] ->    ...(* other behaviour *)
     g !<termName> !<SomeIref> !resList ;                       ...(* other behaviour *)
     [ ] ... (* other terminations *))
  [ ] ... (* other interrogations *))
endproc (* OpIntSigServer *)
```

Comme les signatures d'interface client, une signature d'interface serveur a une série de références d'interface connues et sa propre référence. Cette dernière référence d'interface sert à vérifier que les invocations d'annonce ou d'interrogation que le serveur reçoit sont bien celles qui étaient attendues, c'est-à-dire qu'elles faisaient partie de l'ensemble d'opérations associé à cette référence d'interface. En cas d'impossibilité de recevoir ces invocations, pour cause par exemple d'inexactitude des paramètres ou d'indisponibilité de l'opération demandée, des comportements de traitement d'erreurs sont mis en œuvre. Dans le cas d'annonces, cette situation pourrait donner lieu à un appel récursif sans que cela modifie la liste des paramètres formels. Il est également possible de recourir ici à un mécanisme de garde qui empêche dès le départ l'événement de se produire. Nous ne le faisons pas car cela pourrait aboutir à des impasses indésirables dans la spécification. Dans le cas d'interrogations, cela se traduirait par le renvoi d'une forme quelconque de message d'erreur.

Comme dans le cas des interfaces opération client, il est possible d'exiger que les messages reçus soient ceux qui étaient attendus. Il est toutefois impossible d'assortir les messages envoyés d'instructions. D'aucuns diront que cette restriction n'est pas nécessairement une mauvaise chose du fait que, pour autant que chaque processus traite de la même manière les messages reçus, les messages envoyés ne devraient pas donner lieu à des impasses imputables au fait, par exemple, que leur format n'est pas compris.

A.1.1.13 Signature d'interface flux

En LOTOS, une signature d'interface flux est représentée par une définition de processus, de telle sorte que toutes les offres d'événement qui doivent être synchronisées avec l'environnement pour se produire soient modélisées sous forme de flux producteurs ou consommateurs. Une telle signature peut être représentée en LOTOS comme suit:

```
process StreamIntSig[ g... ](myRef: IRef, known: IRefSet, fss: FlowElementSeqSet...):noexit:=
          ConsumeAction[ g...](myRef, known, recFlowElements...) [ ]... (* other consume actions *)
           [ ]
           ProduceAction[ g...](myRef, known, FlowElementstoSend, ...) [ ]... (* other produce actions *)
endproc (* StreamIntSig *)
```

Comme dans le cas des interfaces signal, la notion de causalité est appliquée aux modèles d'action individuels dans la signature d'interface flux. Une telle signature contient des ensembles de flux consommateurs ou producteurs d'actions. Chaque signature de flux est représentée par un processus. Ces processus contiennent la référence de l'interface flux à laquelle ils sont associés, un ensemble de références d'interface représentant les références d'interface connues de cette interface ainsi qu'une séquence d'éléments de flux à envoyer (dans le cas de flux producteurs) ou à recevoir (dans le cas de flux consommateurs). Par souci de concision, nous n'indiquons pas ici comment les ensembles de séquences d'éléments de flux qui sont transmis à une signature d'interface flux sont attribués aux flux producteurs dans cette interface. Lorsqu'ils sont instanciés, tous les flux consommateurs sont, naturellement, vides.

La Note au § A.1.1.11 s'applique aussi aux signatures d'interface flux, sous réserve d'en modifier dûment le texte (remplacement de "signal entrant" par "flux consommateur", par exemple).