Electronic Journal on Networks and Distributed Processing, No. 2, September 1995, pp. 57 – 73.
Manuscript received april 1995.

Utilisation de la méthode des traces pour la définition formelle d'un protocole de communication

Abdellatif Obaid, Michal Iglewski et Benoit Desrosiers

Département d'informatique, Université du Québec à Hull
C.P. 1250, Succ. B., Hull (Québec), Canada
e-mail: {obaid, iglewski, desrosier}@uqah.uquebec.ca

RÉSUMÉ — Cet article présente un exemple d'application de la méthode des traces dans le domaine des protocoles de communication. L'étude porte sur une version simplifiée d'un protocole de liaisons de données inspiré de HDLC (High-level Data Link Control). Une spécification formelle du protocole et certains problèmes rencontrés, font l'objet d'une discussion.
MOT-CLÉS : *Méthode des traces, spécification formelle, protocoles, HDLC.*

ABSTRACT — In this paper we present an example of application of the trace assertion method in the area of communication protocols. The example we chose is a simplified version of the HDLC (High-level Data Link Control) protocol. A formal description of this protocol as well as the discussion of encountered problems are given.
KEY WORDS: *Trace assertion method, formal specification, protocols, HDLC.*

1. INTRODUCTION

Dans cet article on présente une spécification d'un protocole de la couche de liaisons de données de l'OSI. Ce protocole inspiré de HDLC [ANSI 88] définit les procédures à suivre pour l'échange d'informations entre deux usagers reliés par une liaison de données. Plusieurs spécifications formelles de ce protocole ou de certaines versions simplifiées ont déjà été publiées. Citons, par exemple, celles faites en LOTOS [ISO 88] ou Object-Z [Duke 91]. Dans cet article nous proposons la spécification d'une version simplifiée de ce protocole en utilisant la méthode des traces.

La méthode des traces, présentée pour la première fois à la fin des années 1970 peut être utilisée pour fournir une description formelle des automates de style "boite noire" [Bartussek 85]. Dans [Hoffman 85] cette méthode avec plusieurs modifications est utilisée pour spécifier un protocole simple de liaisons de données. Notre article est basé sur la version révisée et plus complète de la méthode des traces [Parnas 89, Iglewski 93]. La section 1.2 contient une courte description de la méthode.

 © 1995 TASC

Le concept de traces selon un modèle un peu différent (une séquence de valeurs d'entrée/sortie) a fait l'objet de plusieurs études. Dans [Gallouzi 90] et [Gallouzi 91] les traces sont utilisées pour spécifier des propriétés de spécifications LOTOS. Dans [Hoare 85] les traces sont utilisées comme base de description de systèmes (non-déterministes) communicants.

L'approche que nous utilisons consiste en une décomposition modulaire du protocole. Cette décomposition permet d'isoler certaines particularités du protocole de façon à en simplifier la spécification. De ce fait, cette spécification est orientée protocole par opposition à une approche orientée service que nous introduisons à la section 5. Les modules *Transaction*, *Frame* et Copy (voir section 1.4) définissent le fonctionnement du protocole en spécifiant les interactions possibles entre les différentes composantes du réseau. Nous avons simplifié la définition du protocole en omettant de spécifier le traitement des erreurs de transmission; seules les pertes complètes d'information sont spécifiées.

1.1. Terminologie utilisée dans ce document

La communication entre deux usagers est effectuée grâce à deux canaux unidirectionnels: le *canal d'émission* et le *canal d'acquittement*. Ces canaux sont non fiables, i.e. l'information qui y transite peut être perdue. Le modèle de service basé sur ces entités est donné à la figure 1.

Une série de messages transmis d'un usager à l'autre constitue ce que nous appelons une *transaction*. Une transaction peut être représentée comme étant une structure composée d'une suite de messages pouvant être émis. Un message faisant partie d'une transaction a un identificateur permettant de le différencier des autres. Le couple message-identificateur forme une *trame*.

1.2. Introduction à la méthode des traces

La méthode des traces est articulée autour du concept de module. Un module est un ensemble de sous-programmes appelés *programmes d'accès* du module qui sont visibles de l'extérieur du module et qui utilisent une structure de données cachée à l'intérieur de ce module. Un module peut être vu comme une implémentation d'un ou plusieurs automates à états finis appelés *objets*. La communication entre les objets implémentés par le module et le monde extérieur est accomplie via:

- un vecteur de variables externes que l'objet observe (*variables d'entrée*),
- un vecteur de variables qui sont contrôlées par l'objet et qui peuvent être observées de l'extérieur (*variables de sortie*), et
- des programmes d'accès du module qui peuvent être utilisés par d'autres modules pour fournir de l'information à l'objet ou recevoir de l'information de celui-ci.

Les valeurs des variables d'entrée peuvent être observées à chaque moment par le module. Pour les observer, celui-ci doit utiliser des dispositifs matériels ou des programmes d'accès propres à ces variables, fournis de l'extérieur. Les observations réelles sont soumises à des contraintes temporelles déterminées par l'environnement.

Un changement d'état d'un objet peut être causé seulement par un événement externe, i.e. une invocation d'un programme d'accès ou un changement dans la valeur d'une ou plusieurs variables d'entrée.

Une trace est une histoire complète (finie) du comportement observable de l'objet. Elle contient tous les événements affectant l'objet et toutes les sorties produites par l'objet. L'ensemble de traces est divisé par une relation d'équivalence (définie ci-dessous) en nombre fini de classes d'équivalence. Chaque classe d'équivalence est représentée par un de ses éléments appelé *trace canonique*. Intuitivement, les traces canoniques correspondent aux états du module. L'ensemble de traces canoniques est caractérisé par un prédicat appelé *canonical*.

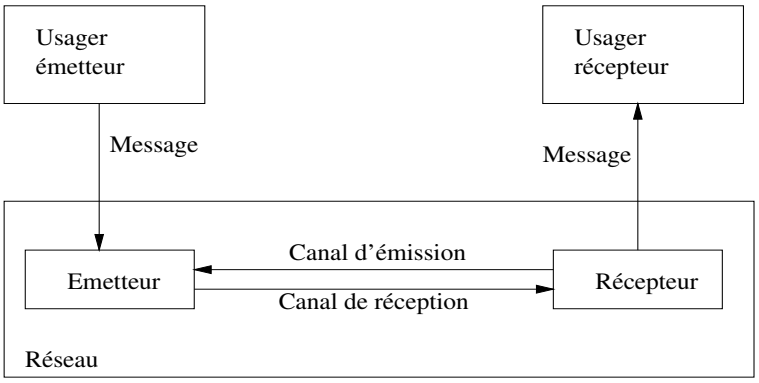

Figure 1. *Modèle de service*

Une description complète d'un module (dans le style de boite noire) avec la méthode des traces est composée:

- d'une fonction *ext* dont le domaine est l'ensemble de tous les couples (trace canonique, événement) et dont l'image est un ensemble de traces canoniques. $ext(T1, e) = T2$ si la trace étendue $T1.e$ est équivalente à la trace canonique $T2$. Cette fonction, appelée *fonction d'extension*, définit la relation d'équivalence sur les traces;
- d'une relation binaire dont le domaine est l'ensemble de toutes les traces canoniques et dont l'image est un ensemble de vecteurs formés de valeurs des variables de sortie, et
- d'une relation binaire dont le domaine est un ensemble de couples (trace canonique, invocation) et dont l'image est un ensemble de vecteurs contenant une valeur pour chaque argument de sortie de l'invocation.

Pour des raisons de lisibilité de la description, le domaine de la fonction d'extension est partitionné et donc présenté sous forme de plusieurs fonctions chacune correspondant à un programme d'accès ou à un événement de variable d'entrée. À titre d'exemple, la fonction d'extension dans le module *Transaction* utilisé dans la section 2.1, est définie séparément pour chaque programme d'accès. Pour le programme d'accès *CREATE_MSG* la fonction d'extension est spécifiée dans le tableau de la figure 2. Ce tableau doit se lire comme suit: étant donné un objet représenté par le couple (n, T) où n est son nom et T est sa valeur, l'état résultant de l'exécution du programme *CREATE_MSG ((n, T),id,m)* est spécifié par la trace canonique donnée par la colonne *Equivalence* (*id* représente le numéro du message à créer et m est le contenu de ce message). Le texte *%frame already created%* signifie que si un message portant le numéro *id* a déjà été créé, l'appel est illégal et l'état n'est pas changé. Les fonctions *frame_exists* et *send_buffer_full* sont des fonctions auxiliaires définies dans le même module. Pour exprimer le fait que deux traces canoniques qui se distinguent seulement par un nom d'un objet auquel elles s'appliquent sont indentiques, nous remplaçons le nom de cet objet dans les traces canoniques par le paramètre générique *.

1.3. Description sommaire de l'approche

La méthode des traces permet d'utiliser des valeurs d'un module dans un autre module par appels de programmes d'accès avec arguments; elle permet aussi d'"équiper" un module avec des variables d'entrées qui peuvent être observées par le module et qui, pour des situations spécifiques, peuvent influencer l'état du module. Ces deux moyens ne sont pas suffisants pour la description de modules complexes. La trace

$(n,T).CREATE_MSG((n,T),id,m) \Rightarrow$

Condition		**Equivalence**
$frame_exists(T,id)$		$\%frame\ already\ created\%$
$\neg frame_exists(T,id) \wedge$	$\neg send_buffer_full(T)$	$T.A_FRAME(*,id,f,m,time:0)$ $where\ f = Frame:CREATE(*)$
	$send_buffer_full(T)$	$\%send\ buffer\ full\%$

Figure 2. *Exemple de description d'un programme d'accès*

canonique de tels modules peut devenir très difficile à manipuler. La modularisation permet de simplifier la trace en la divisant en plusieurs sections chacune traitant un aspect particulier. Cette approche introduit de nouveaux niveaux d'abstraction impliquant souvent l'introduction de programmes d'accès servant principalement de contenants pour les valeurs sous-jacentes et ne devant pas directement être appelés de l'extérieur du module. L'accès à ces programmes peut être limité par la relation *uses*.

Dans la section qui suit nous présentons un modèle de spécification orientée protocole. La méthode des traces peut aussi être utilisée pour représenter des entités de protocoles de façon isolée. Cette approche que nous avons adoptée dans d'autres études est plus appropriée à la description de services bout-en-bout offerts par cette entité [Bojanowski 94]. Dans la section 5, nous en donnons une brève description.

La méthode des traces est une méthode fonctionnelle dont les éléments de bases sont les modules, leurs programmes d'accès, leurs variables d'entrée et de sortie (avec leurs événements associés) et les traces sur chacun ou sur l'ensemble des objets appartenant à un même module. Par rapport aux autres techniques formelles de description, elle offre les avantages suivants:

- Elle permet de spécifier les exigences de l'usager. En utilisant une structure abstraite (la trace), on peut spécifier le comportement observable de l'extérieur de l'objet considéré.
- L'état d'un objet peut être modifié par des appels des programmes d'accès et des événements d'entrée et le nouvel état est une trace canonique. De cette façon on s'assure au fur et à mesure de l'évolution d'un module donné de ne garder que l'information qui détermine le comportement du module dans le futur. Cette approche est différente des méthodes basées sur les types de données abstraites telles qu'on les trouve sur les systèmes comme AFFIRM [Gerhart 80] par exemple.
- On peut décrire les différentes fonctions de spécification sous forme d'un automate à états finis en utilisant une notation tabulaire). Cette approche est similaire à celle utilisée dans le langage ESTELLE [ISO 88].
- Abstraction faite de la représentation explicite des opérations de parallélisme, la méthode des traces permet de spécifier des comportements de la même façon que LOTOS de base (i.e. Basic LOTOS).

1.4. Choix du modèle

Afin de définir le protocole proposé, nous avons choisi d'étudier les changements apportés à l'état des messages échangés par deux usagers. Ces états correspondent à ce que verrait un observateur ayant une vision globale du système. Il pourrait donc décrire le parcours d'un message, i.e. sa transmission à partir de l'émetteur jusqu'à l'arrivée de l'acquittement. Notre description du protocole se résumera, entre autres, à l'étude des points d'interaction entre les différentes entités du réseau et la définition du comportement lorsqu'un message atteint ces points.

Certains événements ne sont pas localisés à ces points d'accès du service mais plutôt à l'intérieur des composantes du réseau. Des événements tels que la perte d'une information sur un canal peuvent se

$$\left[unMessage_j \,.\, \left[unEtat_{ji}\right]_{i=1}^{maxRetry}\right]_{j=1}^{sendBufferMax}$$

Figure 3. *Déefinition d'une transaction*

produire et seront formalisés. La perte d'une information n'est pas activée par un processus; c'est un événement interne produit spontanément. Cet événement est important dans notre spécification car il permet d'indiquer qu'un message ne pourra plus continuer à circuler.

Suite à ces observations et d'après la définition d'une transaction, nous pouvons déduire que la trace d'une transaction aurait le format défini dans la figure 3 où *maxRetry* représente le nombre maximum de copies d'un même message pouvant être envoyées sur le réseau avant l'abandon de la transmission, $sendBufferMax$ représente la taille du tampon de l'émetteur en nombre de messages, et $unEtat_{ji}$ représente l'état d'une copie i du message j.

Nous distinguerons quatorze états différents pour décrire l'état d'une copie de message (voir module *Copy*), soit:

> envoyée (1), acceptée (2), transmise (3) ou perdue (4), reçue (5) ou rejetée (6), acquittement envoyé (7), acquittement accepté (8), acquittement transmis (9) ou perdu (10), acquittement reçu (11) ou rejeté (12) et de plus, la copie peut être dans l'état time_out (13) ou livrée (14).

De plus, dans le protocole de liaisons de données les messages traversent les liens en suivant l'ordre FIFO. Cette contrainte nous oblige à spécifier l'ordre dans lequel les messages circulent.

La complexité de la trace canonique résultante nous a conduits à décomposer la structure afin de simplifier la spécification. La décomposition se fait en trois modules comme suit:

- Le module *Transaction* gère la succession des différents messages transmis entre les deux usagers.
- Le module *Frame* gère la succession des copies d'un même message (dans notre cas, transmis par le module *Transaction*).
- Le module *Copy* gère l'état dans lequel se trouve chacune des copies des messages (dans notre cas, transmis par le module *Frame*).

2. FONCTIONNALITÉ DE CHACUN DES MODULES

2.1. Le module *Transaction*

La structure définie par ce module s'appelle une transaction. Une transaction est formée d'une série de messages. Le module gère le séquencement des messages qui sont envoyés de l'émetteur au récepteur dans l'ordre FIFO. Il assure une transmission fiable en utilisant une technique d'acquittement positif. Si un acquittement n'est pas parvenu à l'émetteur après un nombre fixe de retransmissions, celui-ci conclura que le message n'a pas été livré.

La transmission d'un message est réalisée par une séquence d'appels de programmes selon l'ordre suivant (pour simplifier la présentation, le couple (n, T) a été remplacé par T):

1. L'usager émetteur crée un message. La création se fait à l'aide du programme *CREATE_MSG (T,id,m)* indiquant que la trame numéro id contenant le message m de la transaction T a été créée. Si l'émetteur n'a pas d'espace pour mémoriser ce message, un message d'erreur est produit.

2. Une fois créé, le message est émis sur le canal d'émission via le programme *SEND(T,id)* et ce dans l'ordre FIFO. On suppose que la capacité du canal d'émission est non bornée.

3. Le canal d'émission accepte le message lui parvenant de l'émetteur via le programme *ACCEPT(T,id)*.

4. Le message est soit transmis au récepteur via le programme *TRANSMIT(T,id)*, soit perdu par le canal d'émission par le programme *LOSE(T,id)*.

5. Chaque message transmis est reçu par le récepteur via le programme *RECEIVE(T,id)*. Ce programme vérifie s'il reste de la place dans le tampon du récepteur. Si oui, il accepte le message, sinon le message est rejeté.

6. Un acquittement est transmis pour une série de messages (principe de la fenêtre coulissante).

7. Le canal d'acquittement accepte l'acquittement via le programme *ACCEPT_ACK(T,id)*.

8. Chaque acquittement émis peut être soit transmis via le programme *TRANS_ACK(T,id)*, soit perdu via le programme *LOSE_ACK(T,id)*.

9. Chaque acquittement transmis est reçu par l'émetteur via le programme *REC_ACK(T,id)*. Ce programme acceptera les acquittements jugés valides et rejettera les autres. Un acquittement est jugé valide s'il n'a pas déjà été reçu.

Deux autres opérations peuvent être effectuées, une par l'émetteur et l'autre par le récepteur:

- La première est la génération d'un événement *TIME_OUT(T,id)* pour chaque message envoyé. Cette opération est générée si l'acquittement n'est pas arrivé après un certain temps.

- La deuxième opération, qui concerne le récepteur, est la livraison via le programme *DELIVER(T,id,m)* d'un message reçu. Chaque message reçu doit être livré à l'usager récepteur. Pour être livré, un message doit arriver au récepteur et tous les messages le précédant doivent avoir été livrés.

Chacune des opérations ci-haut mentionnées correspond à l'appel d'un programme d'accès du module *Transaction*. La figure 4 illustre une séquence d'appels de ces programmes et la figure 5 donne un diagramme temporel d'une séquence d'appels de programme d'accès.

Les opérations doivent se produire dans un ordre prédéfini. L'opération *DELIVER(T,id)* est un peu spéciale par le fait qu'elle peut être appelée à tout moment dès que le message a été reçu. Cette opération se fait en parallèle aux autres opérations étant donné que, mis à part *SEND_ACK(T,id)*, les opérations subséquentes ne sont pas effectuées par le récepteur. L'événement *TIME_OUT(T,id)* peut se produire dès que *SEND(T,id)* a été exécuté jusqu'au moment où *REC_ACK(T,id)* sera exécuté. Le paramètre de spécification *maxDelay* détermine le temps d'attente maximale avant *TIME_OUT*.

En plus de spécifier la séquence des événements possibles, le module *Transaction* s'occupe aussi de la gestion des tampons de l'émetteur et du récepteur. La taille de ces deux tampons est donnée par les paramètres de spécification *sendBufferMax* et *recBufferMax*. Les usagers et les entités qu'ils utilisent sont synchronisés. Si l'usager émetteur essaie de fournir des messages "trop rapidement" à l'émetteur, il sera averti que le tampon de l'émetteur est plein et le message envoyé sera refusé. Pour calculer l'espace utilisé dans le tampon de l'émetteur, on doit compter le nombre total de messages créés jusqu'à maintenant et soustraire ceux qui ont été acquittés. D'autre part, si les messages arrivent trop rapidement au récepteur et que son tampon est plein, alors les messages seront rejetés et un *TIME_OUT* sera émis pour ceux-ci. Les conditions de rejet des messages par le récepteur sont spécifiées dans le programme *RECEIVE*. L'espace utilisé dans le tampon du récepteur se calcule en comptant les messages transmis et en soustrayant ceux qui ont été livrés à l'usager récepteur.

Une trace canonique est une séquence d'appels du programme *A_FRAME*, dont la définition détaillée est donnée dans [Desrosiers 93]. Pour chacun des messages constituant une transaction il y a une occurrence du

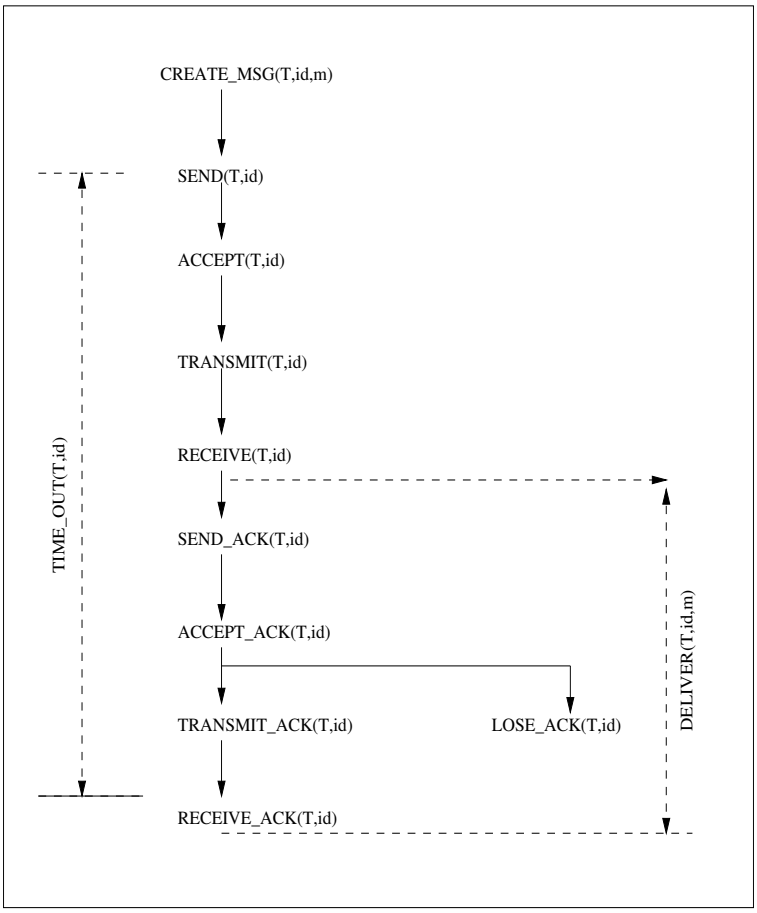

Figure 4. *Transmission d'un message (module* Transaction*)*

programme *A_FRAME(T,id,f,m)* dans la trace canonique. L'identificateur de trame (le deuxième argument de chaque programme d'accès) sert à différencier chacun des messages.

La figure 6 donne un exemple de modifications apportées à la trace canonique suite à une séquence d'appels de programmes d'accès du module *Transaction*. Par exemple, dans la première entrée, l'appel de *CREATE_MSG* résulte en la création d'une copie de *A_FRAME* (i.e. une trame est générée). La condition *ready_to_send(f)* définit l'état initial de la première copie à envoyer. La onzième entrée (l'appel de *SEND(T,id)*) indique qu'une trame ne peut être émise par l'émetteur si cette trame n'a pas déjà été transmise par l'usager émetteur (via le programme *CREATE_MSG*) ou bien si elle est déjà transmise par l'usager émetteur et par l'émetteur mais que l'événement *TIME_OUT* ne s'est pas produit pour cette copie.

Le même message peut être émis plusieurs fois si un *TIME_OUT* se produit. Il y aura alors plusieurs copies du message sur le réseau et chacune de ces copies pourra être dans un état différent (voir les deux derniers appels: *SEND(T,id)* et *SEND(T,id2)*). Le nombre de copies dans une trame est limité par le paramètre de spécification *maxRetry*. On peut aussi noter que l'appel du programme *TIME_OUT(T,n)* a pour conséquence de mettre le message ayant l'identificateur n et tous ceux ayant été envoyés après lui dans l'état *TIME_OUT* (i.e., ils seront ré-émis).

La figure 7 donne le diagramme temporel pour la séquence d'appels présentée à la figure 6.

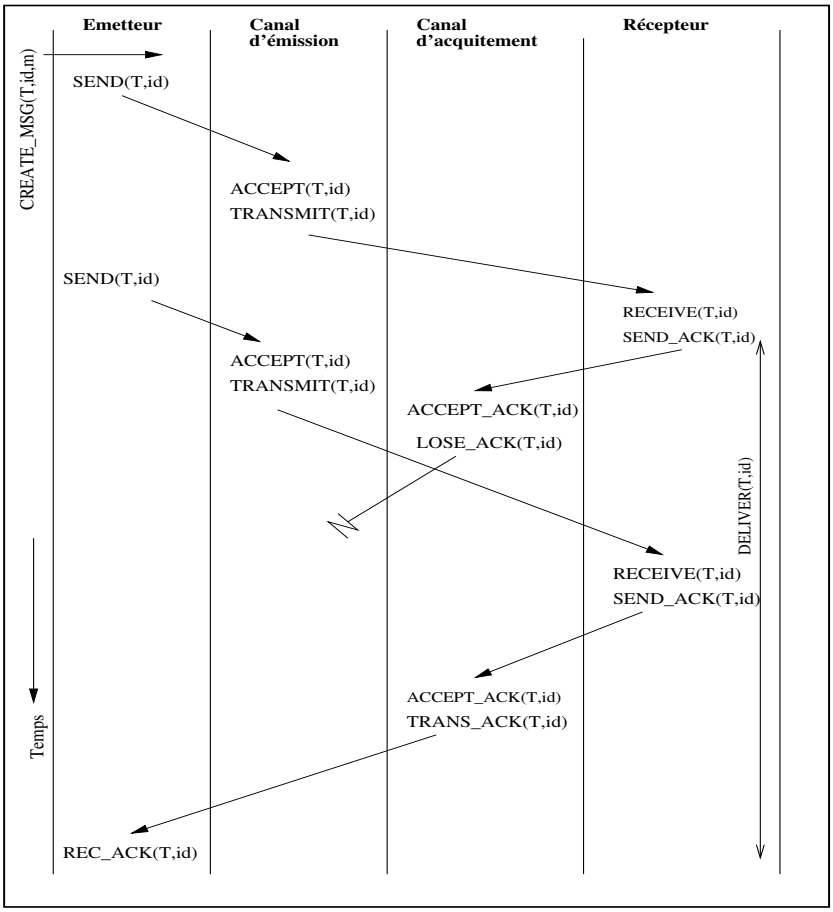

Figure 5. *Décompsition temporelle d'une séquence d'appels*

2.2. Le module *Frame*

Tel que défini dans [ANSI 88], une trame contient un champ pour le numéro de séquence du message et un champ pour les données. Nous avons appelé ce module *Frame* parce que chacune de ses valeurs représente l'état dans lequel se trouve une trame, son numéro de séquence devant être spécifié par le module utilisant *Frame* (en l'occurrence, le module *Transaction*). Chaque trame peut être envoyée plusieurs fois avant de réussir à traverser le réseau. C'est pourquoi ce module utilise le module Copy qui représente l'état plus spécifique de chacune de ces copies.

Lorsqu'un usager envoie un message sur le canal d'émission, il envoie en fait une copie de ce message. Cette copie passe d'une entité du réseau à une autre. Chaque fois que la copie traverse une étape nous disons qu'elle change d'état. Les quatre entités (l'émetteur, le canal d'émission, le récepteur et le canal d'acquittement) sont connectées tel qu'illustré à la figure 8. Il y a quatre connexions dans le réseau et chaque connexion relie deux entités. Lorsqu'on veut transmettre un message d'une entité à une autre, la première doit transmettre le message et la deuxième doit le recevoir, ce qui donne 8 programmes différents. De plus, les deux canaux peuvent perdre l'information et l'émetteur comme le récepteur peuvent rejeter un message.

Chacune des copies envoyées sur le réseau est représentée par une copie du programme *A_COPY(F,fs)* dans la trace canonique (voir Figure 9). L'argument *fs* de ce programme contient l'état de la copie. Le

trace	**trace canonique correspondante**
$CREATE_MSG(T, id, m)$	$A_FRAME(*, id, f, m, t1)$ *where* $Frame : ready_to_send(f)$
$.SEND(T, id)$	$A_FRAME(*, id, f, m, t1)$ *where* $Frame : ready_to_accept(f)$
$.ACCEPT(T, id)$	$A_FRAME(*, id, f, m, t1)$ *where* $Frame : ready_to_transmit(f)$
$.TRANSMIT(T, id)$	$A_FRAME(*, id, f, m, t1)$ *where* $Frame : ready_to_receive(f)$
$.RECEIVE(T, id)$	$A_FRAME(*, id, f, m, t1)$ *where* $Frame : ready_to_send_ack(f)$
$.CREATE_MSG(T, id2, m2)$	$A_FRAME(*, id, f, m, t1).A_FRAME(*, id2, f2, m2, t2)$ *where* $Frame : ready_to_accept(f) \wedge Frame : ready_to_send(f2)$
$.SEND(T, id2)$	$A_FRAME(*, id, f, m, t1).A_FRAME(*, id2, f2, m2, t2)$ *where* $Frame : ready_to_accept(f) \wedge Frame : ready_to_accept(f2)$
$.TRANSMIT(T, id2)$	*%no next to transmit%*
$.DELIVER(T, id, m)$	$A_FRAME(*, id, f, m).A_FRAME(*, id2, f2, m2)$ *where* $Frame : ready_to_send_ack(f) \wedge Frame : has_been_delivered(f) \wedge Frame : ready_to_accept(f2)$
$.SEND_ACK(T, id)$	$A_FRAME(*, id, f, m, t1).A_FRAME(*, id2, f2, m2, t2)$ *where* $Frame : ready_to_accept_ack(f) \wedge Frame : has_been_delivered(f) \wedge Frame : ready_to_accept(f2)$
$.SEND(T, id)$	*%no next to send%*
$.TIME_OUT(T, id)$	$A_FRAME(*, id, f, m, t1).A_FRAME(*, id2, f2, m2, t2)$ *where* $Frame : ready_to_accept_ack(f) \wedge Frame : has_been_delivered(f) \wedge Frame : is_timeout(f) \wedge Frame : ready_to_accept(f2) \wedge Frame : is_timeout(f2)$
$.SEND(T, id)$	$A_FRAME(*, id, f, m, t1).A_FRAME(*, id2, f2, m2, t2)$ *where* $Frame : ready_to_accept_ack(f) \wedge Frame : has_been_delivered(f) \wedge Frame : ready_to_accept(f) \wedge Frame : ready_to_accept(f2)$
$.SEND(T, id2)$	$A_FRAME(*, id, f, m, t1).A_FRAME(*, id2, f2, m2, t2)$ *where* $Frame : ready_to_accept_ack(f) \wedge Frame : has_been_delivered(f) \wedge Frame : ready_to_accept(f) \wedge Frame : ready_to_accept(f2)$

Figure 6. *Changements d'états (module* Transaction*)*

module *Frame* assure qu'une opération est toujours appliquée sur la plus ancienne des copies pouvant effectuer cette opération. Le paramètre de spécification *maxRetry* permet de limiter le nombre de copies d'un même message. Si l'usager émetteur est obligé de faire *maxRetry+1* essais pour transmettre son message, il recevra un message d'erreur lui indiquant qu'il a fait trop d'essais. La figure 9 donne un exemple de modifications apportées à la trace suite à une série d'appels de programme d'accès du module *Frame*.

Les programmes *DELIVER* et *TIME_OUT* ne changent pas l'état de la copie. Ces deux programmes sont conservés au niveau du module *Frame* car ils s'appliquent au message plutôt qu'à une copie particulière de ce message.

2.3. Le module Copy

Ce module gère les états dans lesquels une copie de message peut se trouver. Les programmes sont similaires à ceux du module *Frame* mais ils s'appliquent à une seule copie du message.

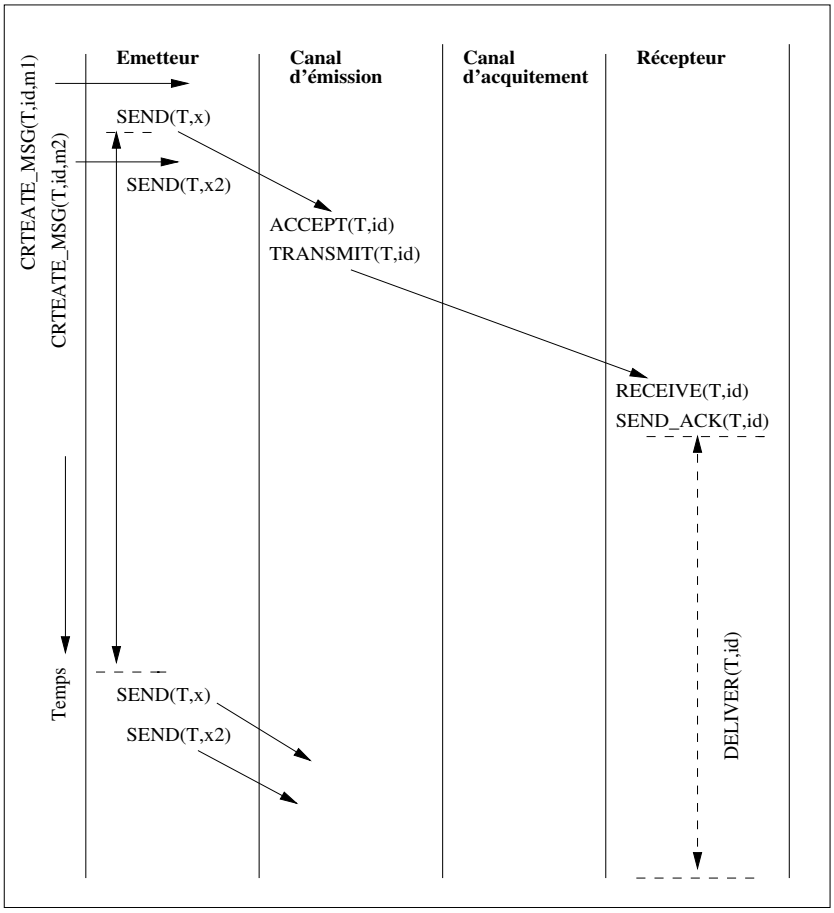

Figure 7. *Diagramme temporel pour l'exemple de la figure 6*

La séquence d'opérations est plus simple dans ce module car les opérations *TIME_OUT* et *DELIVER* n'en font pas partie. En effet, ce n'est pas à une copie en particulier de savoir si elle a été livrée ou si un *TIME_OUT* a été provoqué pour elle (ces deux programmes s'appliquent à une série de copies et ils sont conservés au niveau du module *Frame*). La séquence est une simple suite de transformations qui ont été appliquées à une copie donnée et qui doivent s'effectuer dans un ordre précis.

Il n'y a pas d'opérateur de création comme dans la figure 4. En effet l'opération *SEND(fs)* sert à créer la copie étant donné que celle-ci commence à exister au moment de son envoi sur le canal d'émission.

La figure 10 illustre la séquence d'appels des programmes d'accès et la figure 11 donne un exemple de modifications apportées à la trace canonique du module par une séquence d'appels des programmes d'accès.

Une fois qu'une opération a été exécutée sur une copie, elle ne peut plus l'être à nouveau. De plus, si une copie se perd ou qu'elle est rejetée par le récepteur, aucune opération ne pourra dorénavant être appliquée sur celle-ci (d'autres copies devront être envoyées tant que l'acquittement n'aura pas été accepté par l'émetteur).

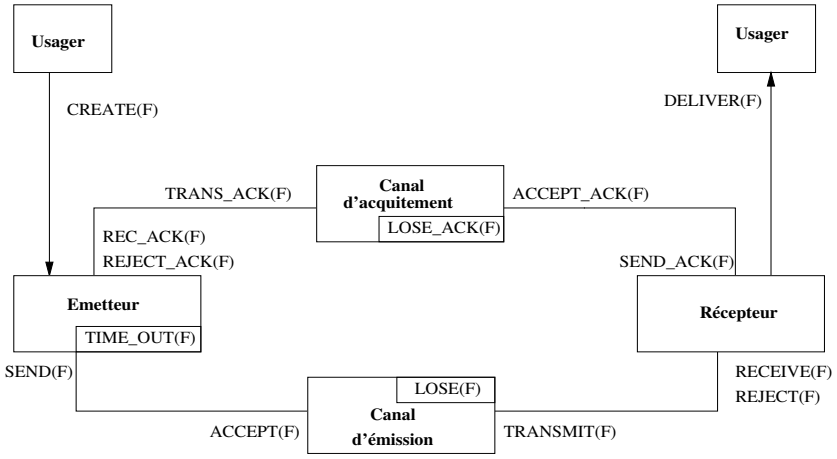

Figure 8. *Localisation des appels des programmes d'accès*

trace	**trace canonique correspondante**
$CREATE(F)$	$CREATE(F)$
$.SEND(F)$	$CREATE(F).A_COPY(F, fs)$ *where* $Copy : ready_to_accept(fs)$
$.TIME_OUT(F)$	$CREATE(F).A_COPY(F, fs).TIME_OUT(F)$ *where* $Copy : ready_to_accept(fs)$
$.RECEIVE(F)$	*%not transmitted%*
$.ACCEPT(F)$	$CREATE(F).A_COPY(F, fs).TIME_OUT(F)$ *where* $Copy : ready_to_transmit(fs)$
$.TRANSMIT(F)$	$CREATE(F).A_COPY(F, fs).TIME_OUT(F)$ *where* $Copy : ready_to_receive(fs)$
$.SEND(F)$	$CREATE(F).A_COPY(F, fs).A_COPY(F, fs2)$ *where* $Copy : ready_to_receive(fs)$ $\wedge\ Copy : ready_to_accept(fs2)$
$.RECEIVE(F)$	$CREATE(F).A_COPY(F, fs).A_COPY(F, fs2)$ *where* $Copy : ready_to_send_ack(fs)$ $\wedge\ Copy : ready_to_accept(fs2)$
$.ACCEPT(F)$	$CREATE(F).A_COPY(F, fs).A_COPY(F, fs2)$ *where* $Copy : ready_to_send_ack(fs)$ $\wedge\ Copy : ready_to_transmit(fs2)$

Figure 9. *Changements d'états (module* Frame*)*

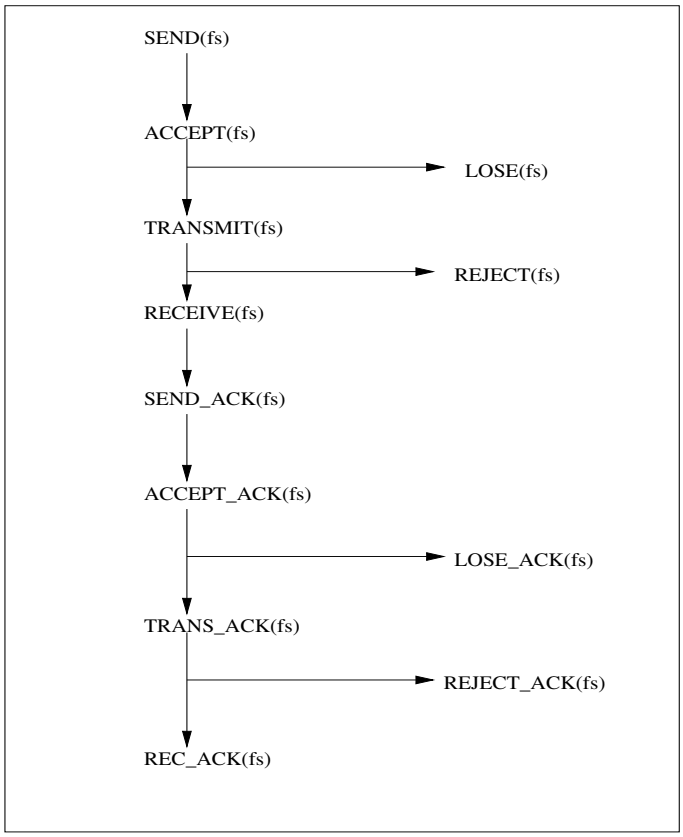

Figure 10. *Transmission d'un message (module Copy)*

trace	**trace canonique correspondante**
$SEND(fs)$	$SEND(fs)$
$.SEND(fs)$	$\%already\ sent\%$
$.ACCEPT(fs)$	$SEND(fs).ACCEPT(fs)$
$.TRANSMIT(fs)$	$SEND(fs).ACCEPT(fs).TRANSMIT(fs)$
$.RECEIVE(fs)$	$SEND(fs).ACCEPT(fs)$ $.TRANSMIT(fs).RECEIVE(fs)$
$.SEND_ACK(fs)$	$SEND(fs).ACCEPT(fs).TRANSMIT(fs)$ $.RECEIVE(fs).SEND_ACK(fs)$
$.LOSE_ACK(fs)$	$SEND(fs).ACCEPT(fs).TRANSMIT(fs)$ $.RECEIVE(fs)$ $.SEND_ACK(fs).LOSE_ACK(fs)$
$.TRANS_ACK(fs)$	$\%not\ ready\ to\ transmit\ ack\%$
$.SEND(fs)$	$\%already\ sent\%$

Figure 11. *Changements d'états (module Copy)*

3. UTILISATION DES MODULES *Transaction*, *Frame* et *Copy*

Ces trois modules représentent l'état des messages d'une communication entre deux usagers. Ils peuvent être utilisés pour modéliser le fonctionnement des quatre entités du réseau.

Le comportement global de ces quatre entités doit correspondre au comportement du module *Transaction*. Chaque entité devra se charger d'une partie des services du protocole:

1. L'usager émetteur doit s'occuper de l'exécution de *CREATE_MSG(T,id,m)*, de *SEND(T,id)* et de *REC_ACK(T,id)*.
2. L'événement *TIME_OUT(T,id)* se produira du côté de l'usager émetteur.
3. Le canal d'émission doit s'occuper de *ACCEPT(T,id)*, de *TRANSMIT(T,id)* et de *LOSE(T,id)*.
4. Le récepteur doit s'occuper de *RECEIVE(T,id)*, de *SEND_ACK(T,id)* ainsi que de *DELIVER(T,id,m)*.
5. Le canal d'acquittement doit s'occuper de *ACCEPT_ACK(T,id)*, de *TRANS_ACK(T,id)* et de *LOSE_ACK(T,id)*.

Pour des raisons mentionnées dans la section 4, les programmes *A_FRAME* et *A_COPY* ont été introduits seulement pour alléger la spécification et ne peuvent pas être utilisés de l'extérieur.

4. DISCUSSION DE LA SPÉCIFICATION ORIENTÉE PROTOCOLE

Nous présentons quelques aspects de notre modèle qui nécessitent des explications particulières.

1. Les programmes *A_FRAME* et *A_COPY* servent à conserver les valeurs des modules utilisés dans les spécifications de *Transaction* et de *Frame* respectivement. Si on prend le cas du programme *A_FRAME*, on peut dire qu'il sert de contenant pour une valeur de type *Frame* et normalement, il ne devrait pas être appelé de l'extérieur du module. Le dernier fait ne peut pas être exprimé dans la méthode des traces. Une autre solution qui fut envisagée, consistait à introduire un seul module et à utiliser des fonctions auxiliaires pour identifier dans une trace des segments correspondant à des informations actuellement représentées par des valeurs des modules *Frame* et Copy. Les inconvénients de cette approche sont l'utilisation d'expressions plus complexes et le fait que les domaines et les images des fonctions auxiliaires devraient être définis comme ensembles de chaînes de caractères et ne seraient pas exprimés en fonction de types de modules définis précédemment.
2. La trace contient l'historique de ce qui s'est passé sur le réseau. Cet historique contient entre autres le nombre de messages transférés. Ce compteur est représenté par le nombre d'occurrences du programme *A_FRAME* dans la trace. En réalité, la trace contient tous les messages transférés, et il serait difficile de définir une condition nécessaire pour éliminer des occurrences de *A_FRAME* lorsqu'elles ne sont plus utilisées. Nous pourrions vérifier dans les programmes *REC_ACK*, *REJECT*, *REJECT_ACK*, *DELIVER*, *LOSE*, et *LOSE_ACK* que la trame est dans un état qui nous permet de l'effacer mais l'opération d'effacer ne trouve aucun parallèle dans la réalité. Ce manque de parallélisme provient du fait que nous représentons une structure répartie à l'aide d'une structure centralisée. Une des solutions envisagées est de doubler certaines informations dans la trace pour qu'on puisse en enlever partiellement.
3. Les programmes *LOSE* et *LOSE_ACK* sont utilisés pour représenter un événement qui s'est produit mais qu'aucun autre programme ne peut appeler. Nous devons tout de même les utiliser car ils permettent d'exprimer le fait que la trame, ou l'acquittement, ne peut plus être transmis. Si nous n'exprimions pas ce fait, l'information pourrait rester indéfiniment sur le réseau, ce qui ne répondrait pas au modèle réel représenté.

4. L'argument *frameId* dans le module *Transaction* sert à différencier chacune des trames. Il indique l'ordre d'apparition des trames dans la trace.

5. On aurait pu utiliser une trace canonique dans laquelle l'appel de *ACCEPT* éliminerait l'appel de *SEND*. Cependant, si la trace *CREATE_MSG. ACCEPT* est canonique alors le programme *ACCEPT* ne peut pas vérifier que le programme *SEND* a effectivement été exécuté, ce qui entraîne un rejet de ce modèle.

6. Selon la description de la méthode, dans le cas de modules multi-objets une copie des variables d'entrée et de sortie existe pour chaque objet du module. Nous avons trouvé que dans certaines applications (par ex. module *Transaction*), l'usage d'une variable partagée par tous les objets du module serait préférable.

5. MODÈLE ORIENTÉ SERVICE

Dans l'approche que nous avons décrite jusqu'à présent, la spécification est orientée protocole. Une autre approche que nous expliquons ici consiste à écrire une spécification orientée service. Le module spécifié correspond alors à une boite noire dont les interfaces (points d'accès de service ou SAP) seront associées à des programmes d'accès ou des variables d'entrée/sortie [Bojanowski 94]. Cette approche est basée alors sur le modèle de la figure 12.

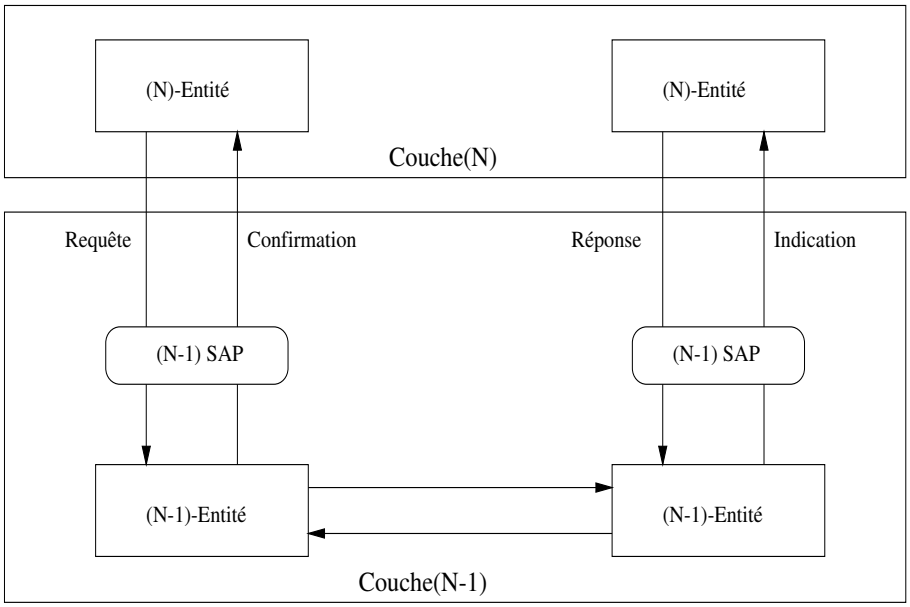

Figure 12. *Points d'accès de service et primitives de service*

Dans ce modèle la spécification peut être structurée soit en utilisant des programmes d'accès ou des variables d'événements (d'entrée ou de sortie). Les variables d'entrée peuvent être utilisées comme paramètres d'un événement (Figure 13). Un événement a lieu au moment où la condition qui lui est associée devient vraie.

Il y a un nouvel événement associé à chaque changement de la valeur de la variable d'entrée. Les nouvelles valeurs et les anciennes valeurs des variables d'entrée pourraient apparaître dans la trace canonique.

Nom de variable	**Type de variable**	**Condition**	**Événement**
Var_1	Type_1	Cond_1(Var_1)	Event_1(Var_1)
Var_2	Type_2	Cond_2(Var_2)	Event_2(Var_2)

Figure 13. *Exemple de déclarations d'événements et leurs variables*

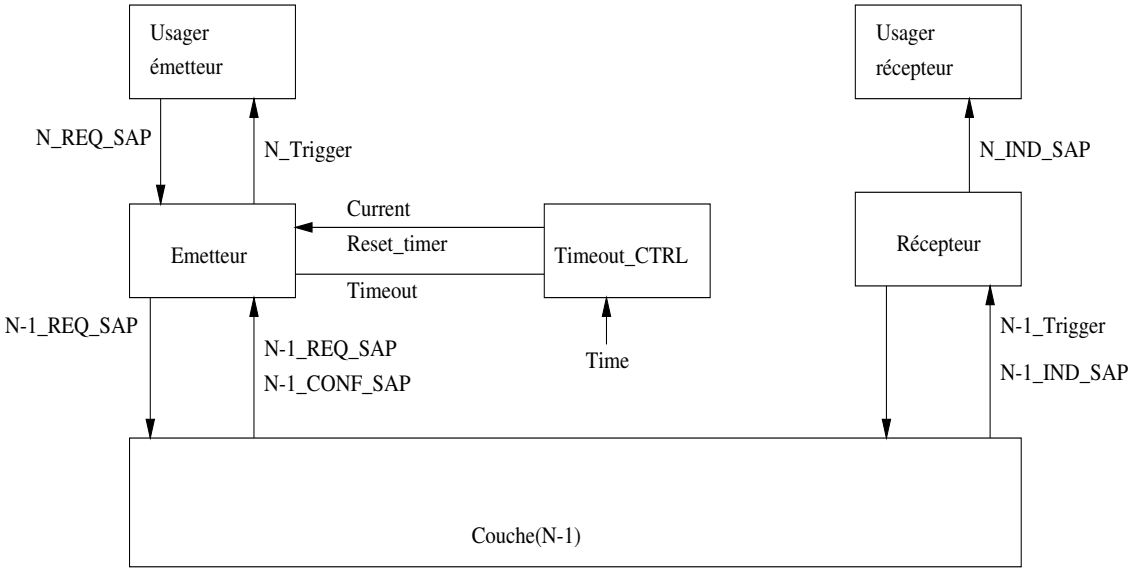

Figure 14. *Dépendances entre les entités*

Dans ce modèle on distingue entre les primitives qui arrivent à une entité et celles qui partent de cette entité d'une part et les requêtes vers la couche N-1 et les réponses de la couche N-1 d'autre part.

Les représentations des primitives qui partent d'une entité et celles qui arrivent à cette entité doivent être différentes: les primitives qui arrivent à une entité sont décrites à l'aide d'événements de variables d'entrée, les primitives qui partent d'une entité sont décrites par des changements des valeurs de variables de sortie. Les variables représentent alors les points d'accès de service (Figure 14).

Dans cet exemple les variables d'entrée sont:

- *N_REQ_SAP*: sa valeur est le couple (trame, numéro de séquence de la trame).
- *N-1_CONF_SAP*: sa valeur est le numéro de séquence de la deuxième trame acquiescée.
- *N-1_Trigger*: ses valeurs changent lorsque la couche N-1 est prête pour une nouvelle primitive de service. Cette variable est utilisée pour la synchronisation entre deux couches adjacentes.

Les variables de sortie sont:

- *N-1_SEQ_SAP*: sa valeur est la dernière unité de données du protocole (PDU) qui a été émise.
- *N-1_Trigger*: sa valeur est le numéro de séquence de la trame attendue dans le point d'accès de service N-1.
- *Reset_timer*: sa valeur est le plus haut numéro de séquence de toutes les trames acquiescées.

Cette approche a été expérimentée et donne une spécification plus modulaire. La notion de variables d'entrée et de sortie combinée avec les programmes d'accès permet d'écrire des spécifications plus courtes et structurées de façon plus consistante avec le modèle de référence de l'OSI.

6. CONCLUSIONS

Jusqu'à présent, la méthode des traces avait une portée limitée dans le domaine des protocoles de communication. Dans cet article nous avons présenté comment elle pouvait être utilisée dans ce domaine. La spécification complète peut être trouvée dans [Desrosiers 93]. À notre connaissance, la spécification contient le premier exemple illustrant l'usage de variables d'entrée dans la spécification de modules. Nous avons montré aussi comment les spécifications de plusieurs modules peuvent être reliées. Notre solution est basée sur un seul module pour l'ensemble des entités du réseau. Une solution alternative est la définition d'un module par entité du réseau. Dans le travail sur le modèle orienté service nous essayons d'établir une méthodologie générale pour la spécification de protocoles de communication. Certains résultats préliminaires sont présentés dans [Bojanowski 94]. Nous pensons que ce sujet comme la méthode entière exige d'autres recherches pour que la méthode puisse être utilisée effectivement dans le domaine des protocoles, du génie logiciel et plus généralement pour la spécification et la validation des systèmes répartis.

Les méthodes existantes de modélisation des protocoles de communication (logique temporelle, systèmes de transitions, algèbre de processus, ...) permettent de spécifier et/ou valider ces protocoles et les système répartis en général. Chacune couvre l'un ou l'autre des aspects de développement des systèmes. L'avantage de la méthode des traces, en plus de la spécification des exigences de l'utilisateur, du comportement des modules et de leurs interfaces permet, lorsque combinée avec une méthode complémentaire ([Iglewski 93], [Iglewski 94]), l'analyse et la modélisation du design interne des modules ainsi que leurs implantations.

7. BIBLIOGRAPHIE

[ANSI 88] ANSI "ANSI Advanced Data Communication Control Procedure (ADCCP)", Datapro research corporation, Delran, NJ, USA; Standards C07-044-301, January 1988.

[Bartussek 85] BARTUSSEK W., PARNAS D.L., "Using Traces to Write Abstract Specifications for Software Modules", In: Gehani, N., McGettrick, A.D. (editors), Software Specification Techniques, AT&T Bell Telephone Laboratories, 1985, pp. 111-130.

[Bojanowski 94] BOJANOWSKI J., IGLEWSKI M., MADEY J., OBAID A., "Functional approach to Protocols Specification", The 14th International IFIP Symposium on Protocol Specification, Testing and Verification, Vancouver, B.C., 7-10 June 1994.

[Desrosiers 93] DESROSIERS B., IGLEWSKI M., OBAID A., "Utilisation de la méthode des traces pour la définition formelle d'un protocole de communication", Rapport de recherche 93/05-6, Université du Québec à Hull, Dép. d'informatique, Hull, Québec, Canada, 1993.

[Duke 91] DUKE R., KING P., ROSE G., SMITH G., "The Object-Z Specification Language, Version 1", Technical Report No. 91-1, The University of Queensland, Software Verification Research Center, Queensland, Australia, May 1991.

[Gallouzi 90] GALLOUZI S., LOGRIPPO L. and OBAID A., "A Hoare-Style Proof System for LOTOS", FORTE'90, The 3rd International Conference on Formal Description Techniques, November 5-8 1990, Madrid, pp. 41-56.

[Gallouzi 91] GALLOUZI S., LOGRIPPO L., OBAID A., "An expressive trace theory for Lotos", 11th IFIP Symp. on Protocol Specification Testing and Verification, Stockholm, 1991.

[Gerhart 80] GERHART S. et al., "An overview of the AFFIRM: A specification and verification System", Proceedings of the IFIP Congress 80, Melbourne, Australia, Oct. 1980, pp. 343-347.

[Hoare 85] HOARE C.A.R., "Communicating Sequential Processes", Englewood Cliffs, NJ: Prentice-Hall International, 1985.

[Hoffman 85] HOFFMAN D.M., "The Trace Specification of Communications Protocols", IEEE *Transactions* on Computers, Vol. C-34, No. 12, December 1985, pp. 1102-1113.

[Iglewski 93] IGLEWSKI M., MADEY J., PARNAS D.L., KELLY P., "Documentation Paradigms", CRL Report No. 270, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.

[Iglewski 94] IGLEWSKI M., KUBICA M. and MADEY J., "Editor for the Tracae Assertion Method", Proc. of the 10th International Conference on CAD/CAM, Robotics and Factories in the Future, 94, Ottawa, Canada, August 1994, pp. 876-881.

[ISO 88] ISO "ISO 88 Guidelines for the application of Estelle, Lotos and SDL". ISO/IEC JTC/SC 21 WG1 OSI-Architecture 1988.

[Parnas 89] PARNAS D.L., WANG Y., "The Trace Assertion Method of Module Interface Specification", Technical Report 89-261, Queen's University, C&IS, Kingston, Ontario, Canada, Oct. 1989, 39 pp.

∞

Biographies

A. Obaid *a un doctorat de 3e cycle en informatique de l'Université de Bordeaux I (France) et un Ph.D. en informatique de l'Université d'Ottawa (Canada). Il est actuellement professeur au Département d'informatique de l'Université du Québec à Hull. Son domaine d'intérêt inclut les techniques formelles de spécification, la conception et la validation des systèmes répartis et le génie logiciel.*

M. Iglewski *a un Ph.D. en sciences techniques de l'Institut d'informatique de l'Académie Polonaise des Sciences. Il est actuellement professeur au Département d'informatique de l'Université du Québec à Hull (Canada). Son domaine d'intérêt inclut le génie logiciel, les environnements de programmation et les langages de programmation.*

B. Desrosiers *a travaillé comme chercheur chez Bell Northern Research à Ottawa (Canada). Il est actuellement chercheur au Département d'informatique de l'Université du Québec à Hull (Canada).*