Modèlisation et Preuve de Protocoles Temps-Réel : une Approche Basée Objet [1]

Arnaud Bailly[*] — Elie Najm[*] — Jean-Bernard Stefani[**] — Laurent Leboucher[**]

[*] *École Nationale Supérieure des Télécommunications, 46 rue Barrault, 75634 Paris Cedex 13 {abailly | najm}@infres.enst.fr, Tel : 01 45 81 63 74, Fax : 01 45 81 31 19*

[**] *France Télécom - Centre National des Études en Télécommunications {laurent.leboucher | jean-bernard.stefani}@cnet.francetelecom.fr*

RÉSUMÉ. *Nous proposons de spécifier et de vérifier statiquement des protocoles temps réel à l'aide de méthode basées objet. Nous considèrons des systèmes* ouverts, *où l'ensemble des entités d'une configuration ne sont pas connues lors de la vérification. Nous présentons un langage dans lequel les interactions entre entités réparties sont spécifiées à l'aide d'interfaces comportementales temporisées, qui peuvent évoluer en fonction du temps et des signaux traités. Grâce à elles, des échanges à caractère périodique ou apériodique peuvent être décrits de manière simple et concise, puis vérifiés.Nous proposons donc une méthode de vérification, qui procède en deux étapes. Tout d'abord la compatibilité entre interfaces temporisées communicantes est vérifiée. Ensuite, nous vérifions qu'une entité émet ou reçoit uniquement lorsque ses interfaces le lui permettent. Cela suffit à garantir la sûreté (safety) d'exécution de telles configurations. La méthode obtenue est compositionnelle.*

ABSTRACT. *We propose to specify and statically verify protocols with real-time orientiation using an object-based method. We consider* open *systems, where the entire composition of commmunicating entities present in a configuration is not known. We introduce a langage where interactions between objects are described through timed behavioral interfaces, that can evolve according to message processing and time progress. We are able to specify both periodic and aperiodic interaction patterns in a simple, synthetic and provable fashion. Accordingly, we propose a proof method made of two steps. In the first one, the compatibility of communicating timed interfaces is checked. Then, we verify that an object sends or receives messages only when it is allowed to by its interfaces. We show that this method is sufficient to allow the compositional verification of safety properties on given object configurations.*

MOTS-CLÉS : *Méthodes Formelles, Temps Réel, Typage, ODP, Multimedia*

KEYWORDS: *Formal Methods, Real-time, Typing, ODP, Multimedia*

1. Ces recherches sont patiellement financées par FT/CNET et le projet RNRT MARVEL

[e] soumission à *cfip'00*, le 22 mars 2000.

1. Introduction

Nous nous intéressons à certaines applications récemment apparues dans le monde de l'informatique et des télécommunications, citons : la téléconférence, la télévision numérique, les systèmes de contôle répartis (eg la domotique) et les réseaux actifs. Toutes ces applications font apparaître le besoin de produire du code valide possèdant un caractère temps réel. Des illustrations concrètes peuvent être trouvées dans les protocoles de niveau application comme RTSP [16], et dans les réseaux actifs [17].Notre but est de proposer des moyens efficaces de spécification et de validation pour les protocoles qu'elles utilisent, afin d'obtenir des *exécutables* certifiés.

Il existe de nombreuses propositions permettant la validation de modèles ou de code temporisés : aussi bien dans le test, le model checking [12, 13, 3], que le theorem proving [2]. Cependant, toutes ne répondent que partiellement ou de manière peu adaptée au problème posé par les applications que nous ciblons. En effet, ces applications requièrent le déplacement, l'ajout, l'échange, ou le retrait de composants logiciels lors de l'exécution. On désigne usuellement de tels systèmes configurations comme *ouvertes*. Dans ces conditions, le test et le model checking ne sont pas compétents, puisqu'ils ne s'appliquent qu'à des configurations connues entièrement (ie fermées). Le theorem proving possède lui un caractère compositionnel, ce qui lui permet de répondre à ce besoin. Toutefois, la preuve de propriétés très simples peut se révéler difficile, et une telle preuve est de plus rarement complètement automatisée. Nous allons donc nous inspirer du theorem proving pour vérifier nos spécifications, en tentant de pallier ces deux inconvénients.

Par ailleurs, manipuler des configurations ouvertes fait apparaître de nouvelles perspectives concernant la notation à utiliser pour de telles modèlisations. En particulier, on veut pouvoir vérifier par typage la *reconfiguration* d'un ensemble d'objets. Un exemple de reconfiguration est le passage d'un flot audio-video de 25 à 10 images par secondes : il faut en établir la cohérence (la source et la cible évoluent vers des comportements compatibles) et le caractère synchrone (pas d'interruption du flot). De plus, il est important de pouvoir *paramètrer* ces evolutions, afin d'atteindre des états différents en fonction des contraintes existantes dans l'environnement de la configuration. Par environnement, nous entendons par exemple l'utilisateur, qui peut exiger une certaine qualité de service à un moment donné, puis relâcher ses exigences en pleine exécution. La notation utilisée doit donc pouvoir décrire de tels comportements de manière simple et vérifiable.

Nous proposons donc comme notation un calcul d'acteurs, appelé $\mathcal{A}$*rtOC*, qui introduit des *interfaces comportementales temporisées*. Une interface comportementale [14, 6, 15] définit à un instant donné quel sont les opérations disponibles pour l'objet qui la possède. Cet ensemble peut évoluer lorsqu'une opération est déclenchée, l'interface offrant alors un nouvel ensemble d'opérations. Ceci peut être représenté par un automate, appelé *type comportemental* de l'interface. Nous étendons un type, en l'assimilant maintenant à un automate temporisé [12]. L'ensemble des opérations disponibles peut donc également évoluer en fonction de l'avancement du temps. Nous

donnons une syntaxe qui permet d'exprimer facilement des échanges de messages périodiques et apériodiques. Pour valider une configuration, nous vérifions d'abord la compatibilité des interfaces qui échangent des messages, puis la conformité des objets avec les interfaces qu'ils possèdent.

L'article est organisé comme suit. Tout d'abord, nous donnons quelques considérations supplémentaires sur les travaux existants qui concernent la modèlisation et la vérification de systèmes ouverts (Section 2). Ensuite, nous détaillons un exemple de type en Section 3.1 avant de donner la syntaxe formelle d'écriture des types en Section 3.2. La Section 4 est la section la plus importante en taille, car elle décrit comment vérifier la compatibilité des interfaces. Le langage *ArtOC* est alors décrit dans la Section 5, comprenant les sémantiques statiques et opérationnelles. La Section 7 présente un court exemple avant nos conclusion.

2. Langage et Taxonomie des Types

Nous considèrons un monde d'acteurs évoluant en concurrence et échangeant des messages à travers des interfaces différenciées. Un acteur peut possèder plusieurs interfaces. Ces interfaces ont un rôle, qui désigne leur capacité à émettre ou à recevoir des signaux. Une interface privée est appelée *source* si elle émet un flot et *cible* si elle reçoit un flot. Une interface publique possède le rôle *client* en émission, ou *serveur* en réception. Une interface possèdant un certain rôle peut être émise dans le contenu d'un message, en tant que paramètre. Elle confie alors au destinataire une *capacité* à émettre ou à recevoir. Il existe toutefois des limitations sur ces envois, que nous détaillerons lors de l'examen du langage (Section 5).

Les interfaces que nous utilisons se également divisent entre interfaces *publiques* et *privées*, comme introduit par [6]. Si une interface publique peut interagir avec n'importe-quelle autre interface publique, les interfaces privées sont appariées (une source et une cible), et les échanges de messages sont internes aux couples source-cible. Ceci est une condition nécessaire et suffisante pour pouvoir déterminer les évolutions de types d'interfaces privées au cours du temps et de leurs échanges.

Il est à noter que lorsque deux interfaces temporisées doivent échanger des messages, elles doivent être en phase (comme les ondes), donc recevoir des conditions d'initialisation appropriées. Pour assurer cette étape de synchronisation, nous imposons aux paires d'interfaces privées d'être créées par le même objet, puis que le rôle cible soit envoyée à un objet potentiellement distant. Les interfaces d'un couple sont immuables tant que leur créateur les possède toutes deux, et l'interface cible commence à évoluer dès réception du message dans lequel elle est incluse, alors que l'interface source évolue dès l'envoi de ce message. Comme montré en Section 6.2, ceci suffit à assurer une synchronisation satisfaisante (et vérifiable) des interfaces temporisées.

3. Types pour Interfaces

3.1. *Un Petit Exemple de Type*

Les interfaces privées possèdent deux embouts, appelés *stream* et *control*. L'embout stream (flot) est dédié à l'envoi (rôle source) ou à la réception (rôle cible) de signaux temporisés, périodiques ou apériodiques.L'embout de contôle permet aux interfaces d'échanger des messages relatifs à l'état de la liaison, comme par exemple son arrêt ou un changement de fréquence d'émission et de réception des flots. Le sens de circulation des messages de contrôle peut s'inverser après chaque échange, d'un commun accord entre la source et la cible.

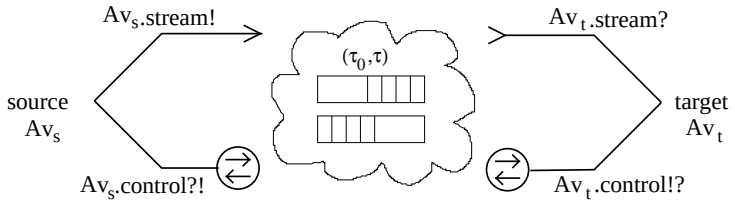

Figure 3.1

Sur la Figure 3.1, apparaissent les caractéristiques du réseau que nous considèreons dans notre modèle. Chaque message passe entre τ_0 et $\tau_0 + \tau$ unités de temps dans le réseau. Les canaux sont FIFO, et les pertes de messages ne sont pas pour l'instant tolérées. Chaque type d'interface est paramètré par τ et τ_0. On obtient ainsi des types valables pour un cetrain nombre de valeurs de ces paramètres. Les valeurs de τ_0 et τ étant connues au moment de la création d'une interface, il appartient au système de typage de n'autoriser que les instanciations correctes d'interfaces.

Nous décrivons maintenant un type comportemental temporisé AV_t permettant la réception cadencée d'audio et de video. Comme tous les types, AV_t décrit à la fois le comportement des parties flot et contrôle d'une interface instanciée sur lui. L'unité de temps est la milliseconde. La fréquence d'émission de trames video est 25 Hz, alors que 10 trames audio sont émises par seconde. Ici, les seuls signaux de contrôle sont *start* et *stop.*

Dans la Table 1, x_1 et x_2 sont les états possibles de la partie flot du type, tandis que x_3, x_4, x_5, x_6 et x_7 sont les états possibles de la partie contrôle. Chaque partie possède ses propres horloges : c_1 et c_2 pour la partie flot, c_3 pour la partie contôle. Les horloges progessent toutes de manière homogène. Quand un type est instancié, l'état initial de l'interface créée est composé des équations de plus faible indice de chaque partie (ici x_1 et x_3), et toutes les horloges sont mises à zéro.

Donc initialement, AV_t autorise l'envoi (noté !) du message de contrôle $start()$, alors qu'aucun message de flot ne peut être reçu (état (x_1, x_3)). $start$ doit être envoyé avant 5 secondes, écrit "c_3 in $[0, 5000]$ $!start$;". Alors la partie de contrôle évolue vers l'état x_4, et c_3 est remise à zéro : "control.$x_4(c_3 = 0)$". L'état du type

Type target $AV_t(\tau_0, \tau) :=$		
stream$[c_1, c_2]$:	x_1	$= \emptyset$
	x_2	$= c_1$ in $[40 - \tau, 40]$ $video()$ until 40 ; stream.$x_2(c_1 = 0)$
		$+ c_2$ in $[100 - \tau, 100]$ $audio()$ until 100 ; stream.$x_2(c_2 = 0)$
control$[c_3]$:	x_3	$= c_3$ in $[0, 5000]$ $!start()$; control.$x_4(c_3 = 0)$
	x_4	$= c_3$ in $[0, 5000]$ $?start\_ack()$; (stream.x_2, control.x_5)$(c_3 = 0)$
	x_5	$= c_3$ in $[5000, \infty]$ $!stop()$; control.$x_6(c_3 = 0)$
	x_6	$= c_3$ in $[0, 1000]$ $?stop\_ack()$; (stream.x_1, control.x_7)$(c_3 = 0)$
	x_7	$= \emptyset$

TAB. 1. *Un Petit Exemple de Type*

est maintenant (stream.x_1, control.x_4), et le message de contrôle $start\_ack()$ doit être reçu (noté ?) avant 5 secondes. Lors de la réception, le type évolue vers l'état (stream.x_2, control.x_5). Vous remarquez ici que la réception d'un message de contrôle autorise l'évolution des deux parties de l'interface. Cette capacité donne tout son intérêt à la partie de contrôle, et ce n'est qu'avec elle que des reconfigurations peuvent avoir lieu. Néanmoins, lors d'une telle transition, il faut que l'évolution de la partie de flot soit déterministe. En effet, on ne connait pas les valeurs des horloges de flot au moment de la réception d'un message de contrôle. Donc, toutes les horloges de la partie de flot doivent recevoir une valeur connue, et si cela n'est pas le cas, elles sont implicitement remises à zéro. C'est le cas ici de x_1 et x_2 lors de la réception de $start\_ack()$.

Après la réception de $start\_ack()$, la réception cadencée de signaux $audio()$ et $video()$ peut commencer. C'est l'état x_2 qui est utilisé tant que durent ces réceptions. Supposons pour simplifier que le paramètre τ vaut 5 millisecondes (ce qui est maintenant décrit restera valable pour d'autre valeurs de τ) . Le signal $video()$ peut alors être reçu quand l'horloge c_2 vaut entre 35 et 40 ms. Lorsque $video()$ est reçu, ce qui est garanti par des propriétés que nous détaillerons plus tard, la transition accomplie laisse la partie flot du type dans l'état x_2. Afin de donner un caractère périodique à cette opération, elle utilise la clause *until*, qui modifie les valeurs attribuées aux horloges lors de la transition. Ainsi, "until 40" et $(c_1 = 0)$ étant spécifiés pour cette transition, l'horloge c_1 n'est pas remise à zéro mais reçoit la valeur telle qu'elle vaudra zéro à la fin de la période de 40 ms. De la sorte, la fin de la période courante a bien lieu 40 ms après son début, et une nouvelle période peut commencer. De manière identique, le signal $audio()$ est reçu suivant une période de 100 ms, car après une transition l'horloge c_2 ne retrouve la valeur 0 qu'à la fin de la période. Concurremment aux réceptions d'audio et de video, le signal $stop()$ est disponible dans la partie de contrôle lorsque c_3 est supérieure à 5 secondes. Après que $stop()$ a été envoyé, audio et video continuent d'être reçus, jusqu'à la réception de $stop\_ack()$ qui interdit toute autre opération en passant à l'état (x_1, x_7).

3.2. *Syntaxe des Types*

ρ	::=	! | ?
Assignment	::=	$c = c'$ | $c = \nu$ | Assignment, Assignment
ec	::=	c in $[\tau_{min}, \tau_{max}]$! $m(\tilde{v})$; control.x'(Assignment)
	|	c in $[\tau_{min}, \tau_{max}]$? $m(\tilde{v})$; control.x'(Assignment)
	|	c in $[\tau_{min}, \tau_{max}]$ ρ $m(\tilde{v})$; (control.x', stream.x'')(Assignment)
es	::=	c in $[\tau_{min}, \tau_{max}]$ $m(\tilde{v})$; stream.x''(Assignment)
	|	c in $[\tau_{min}, \tau_{max}]$ $m(\tilde{v})$ until τ_{max} ; stream.x''(Assignment)
E_c	::=	$x = ec$ | $E_c + ec$ | E_c E_c.
E_s	::=	$x = es$ | $E_s + es$ | E_s E_s
$Private\_Role$	::=	source | target
$Public\_Role$	::=	client | server
Type_Decl	::=	Type $Private\_Role$ T $(\tau_0, \tau) :=$ stream$[c_1 \ldots c_k] : E_s,$ control$[c_{k+1} \ldots c_n] : E_c$
	|	Type $Public\_Role$ T $:= \Sigma m(\tilde{v})$

TAB. 2. *Syntaxe des Types d'Interface*

Un type comportemental s'écrit $T :=$ stream$[\tilde{c}] : E,$ control$[\tilde{c'}] : E'$, où T est un identificateur de type, $\tilde{c}$ et $\tilde{c'}$ sont des ensembles d'horloges, et E et E' des ensembles d'équations (voir Table 2). Cette écriture est préfixée par le mot-clé de déclaration "Type" et du rôle du type, *source* ou *target*. Le rôle d'un type s'applique à toutes les opérations de sa partie flot.

E et E' définissent des ensembles d'équations $(x_1 = e_1), \ldots (x_k = e_k)$, chaque x_i n'apparaissant qu'une et une seule fois dans la partie gauche d'une équation. Chaque équation déclare une opération (émission ou réception) associée à une contrainte temporelle. Une contrainte comporte une référence d'horloge et un intervalle, écrit : c in $[\tau_{min}, \tau_{max}]$. L'intervalle donne la fenêtre de disponibilité de l'opération.

Pour les opérations de flot, deux constructions syntaxiques sont possibles (voir les équations es dans la Table 2). L'une décrit des signaux apériodiques, et l'autre (utilisant *until*) des signaux périodiques. Dans les deux cas, un nom d'équation de E est donné comme prochain état. La valeur assignée à une horloge peut être une constante ν ou la valeur d'une horloge (y compris elle-même). Les valeurs des horloges référencées dans la partie droite sont déterminées avant toute assignation de la transition courante.

La partie de contrôle ne peut effectuer de transition utilisant la clause *until*. Cela n'est pas nécessaire, car les signaux de contrôles sont par nature non périodiques. Cependant deux cas syntaxiques sont requis, car une transition peut aussi modifier

l'état des deux parties du type. C'est le cas uniquement des envois de message de contrôle pour des types sources et des réceptions de messages de contrôle pour des types cibles. L'évolution de la partie flot doit être déterministe (voir l'exemple).

Enfin, nous présentons également des types uniformes pour les interfaces publiques. La déclaration d'un type public est préfixée par le mot-clé "Type", et le rôle du type (*client* ou *server*). La déclaration elle-même s'écrit $T := \Sigma m(\tilde{v})$, où la deuxième partie de l'équation est un ensemble d'opérations non temporisées.

3.2.1. *Paramètrage des Types*

Nous avons jusqu'à présent considéré les types temporisés comme paramètrés uniquement par les caractéristiques du réseau. Nous pouvons généraliser cette proposition en autorisant des valeurs de paramètres de messages reçus par la partie contrôle à entrer dans l'état du type en tant que constante. Ces constantes peuvent être des valeurs temporelles ou entières.

Un exemple d'utilisation de cette capacité peut être l'introduction d'un signal $?reconfigure(\langle audio_p \rangle, \langle video_p \rangle)$ dans la partie contrôle d'un type semblable à AV_t. Les angles $\langle\rangle$ ont pour effet d'associer les identificateurs $audio_p$ et $video_p$ aux valeurs reçues et de les ajouter à l'état du type. Ces identificateurs peuvent ensuite être utilisés dans la partie flot de l'interface, comme désignant les nouvelles périodes d'émission de signaux $audio()$ et $video()$. Dans le cas où $reconfigure$ est utilisée plusieur fois, les anciennes valeurs d'$audio_p$ et $video_p$ sont remplacées par les nouvelles. Un identificateur ne peut toutefois pas être retiré de l'environnement d'un type une fois qu'il a été défini. Des valeurs par défaut des identificateurs peuvent être données dans l'entête du type (avant les équations de la partie de flot). Les paramètres d'une opération n'entrent pas dans l'état du type si les angles $\langle\rangle$ ne sont pas utilisés.

3.3. *Sémantique des Types*

Nous avons retenu une sémantique identique pour les types source et cibles. Elle engendre toutefois des contraintes bien différentes sur les objets possèdant des interfaces sources et ceux possèdant des interfaces cibles.

Principe 3.1 *Principe de Typage*

A l'intérieur d'une interface temporisée, la fenêtre de disponibilité ne doit jamais se fermer sans qu'une opération ne soit déclenchée sur cette interface.

Cela signifie qu'un objet possèdant une interface source est contraint par celle-ci d'envoyer des messages dans certains délais. Un objet possèdant une interface cible doit lui être capable de recevoir tous les messages qui peuvent éventuellement arriver.

4. Compatibilité des Types

Quand un type d'interface source T_s est compatible avec un type d'interface cible T_t, des interfaces instanciées sur ces types communiquent sans se désynchroniser.

Nous commençons par décrire cette relation sous la forme d'un système imaginaire comprenant les types source et cible reliés par deux canaux symétriques. Nous considèrons (condition de compatibilité) que le type cible commence à évoluer entre τ_0 et $\tau_0 + \tau$ unités de temps après l'initialisation du type source. Pour que les types soient compatibles, il faut que lorsque l'un deux peut envoyer un message, l'autre soit prêt à le recevoir entre τ_0 et $\tau_0 + \tau$ unités de temps.

Les interfaces adoptent le comportement décrit par leur type. Il incombe donc au calcul de faire respecter cet écart entre les temps d'initialisation des interfaces. Comme mentionné précédemment, nous imposons pour l'instant le scénario dans lequel les rôles d'interfaces sources et cibles sont créés par un même acteur. Lorsque l'interface cible est envoyée sur le réseau, l'interface source est activée (ses horloges commencent à évoluer). L'interface cible est activée lors de sa réception.

4.1. *Comportement d'un Canal de Communication*

Un canal est une file d'attente FIFO sans pertes. Un canal $\sigma[(m_1, \tau_1), \ldots, (m_k, \tau_k)]$ qui contient les messages $m_1 \ldots m_k$ depuis $\tau_1, \ldots, \tau_k$ unités de temps avec $\tau_1 \leq \tau_2 \leq \ldots \leq \tau_k$, a le choix parmi trois transitions différentes :

- $\sigma[M] \xrightarrow{!m(\tilde{v})} \sigma[m(\tilde{v}), M]$ est une émission de message avec les paramètres $\tilde{v}$, ajoutant $m(\tilde{v})$ à σ,
- $\sigma[M, m(\tilde{v})] \xrightarrow{?m(\tilde{v})} \sigma[M]$ est une réception de message soustrayant $m(\tilde{v})$ de σ, et
- $\sigma[(m_1, \tau_1), \ldots, (m_k, \tau_k)] \xrightarrow{\tau_x} \sigma[(m_1, \tau_1 + \tau_x), \ldots, (m_k, \tau_k + \tau_x)]$ est une transition laissant passer le temps qui, pour chaque message de σ, met à jour son temps passé dans le canal.

4.2. *Comportement d'un Type*

Trois transitions sont également possibles pour un type dans notre système. L'envoi de message est noté $T \xrightarrow{!m(\tilde{v})} T'$, une réception $T \xrightarrow{?m(\tilde{v})} T'$, et une avancée du temps $T \xrightarrow{\tau} T'$. Nous notons ici T' le nouvel état de T après la transition (T' n'est pas un type d'interface, seulement un état).

Definition *Relation de Compatibilité des Types*

Considèrons deux types T_s et T_t, respectivement source et cible de messages. La relation de compatibilité, notée $C_{\tau_0,\tau}(\sigma_{st}[M_s],\sigma_{ts}[M_t])$, tient compte de l'état courant des buffers reliant les types. Nous utilisons la même convention que celle appliquée

aux transitions de types, c'est-à-dire nous nommons $\sigma[M']$ l'état du canal $\sigma[M]$ après évolution. Le contenu exact de M' peut être trouvé en Section 4.1.
Nous dirons que deux types (T_s, T_t) appartiennent à $C_{\tau,\tau_0}(\sigma_{st}[M_{st}], \sigma_{ts}[M_{ts}])$ si et seulement si :

- $(T_s \xrightarrow{!m_s(\tilde{v})} T'_s) \Rightarrow (\sigma_{st}[M_{st}] \xrightarrow{!m(\tilde{v})} \sigma_{st}[M'_{st}] \wedge (T'_s, T_t) \in C_{\tau,\tau_0}(\sigma_{st}[M'_{st}], \sigma_{ts}[M_{ts}]))$
- $(T_t \xrightarrow{!m_c(\tilde{v})} T'_t) \Rightarrow (\sigma_{ts}[M_{ts}] \xrightarrow{!m(\tilde{v})} \sigma_{ts}[M'_{ts}] \wedge (T_s, T'_t) \in C_{\tau,\tau_0}(\sigma_{st}[M_{st}], \sigma_{ts}[M'_{ts}]))$
- $((\sigma_{st}[M_{st}] \xrightarrow{?m_s(\tilde{v})} (\sigma_{st}[M'_{st}])) \Rightarrow (T_t \xrightarrow{?m_s(\tilde{v})} T'_t \wedge (T_s, T'_t) \in C_{\tau,\tau_0}(\sigma_{st}[M'_{st}], \sigma_{ts}[M_{ts}])))$
- $((\sigma_{ts}[M_{ts}] \xrightarrow{?m_c(\tilde{v})} (\sigma_{ts}[M'_{ts}])) \Rightarrow (T_s \xrightarrow{?m_s(\tilde{v})} T'_s \wedge (T'_s, T_t) \in C_{\tau,\tau_0}(\sigma_{st}[M_{st}], \sigma_{ts}[M'_{ts}])))$
- $(T_s \xrightarrow{\tau_x} T'_s \wedge \sigma_{st}[M_{st}] \xrightarrow{\tau_x} \sigma_{st}[M'_{st}] \wedge \sigma_{ts}[M_{ts}] \xrightarrow{\tau_x} \sigma_{ts}[M'_{ts}]) \Rightarrow (T_t \xrightarrow{\tau_x} T'_t \wedge (T'_s, T'_t) \in C_{\tau,\tau_0}(\sigma_{st}[M'_{st}], \sigma_{ts}[M'_{ts}]))$
- $((T_c \xrightarrow{\tau_x} T'_c \wedge \sigma_{st}[M'_{st}] \xrightarrow{\tau_x} \sigma_{st}[M'_{st}] \wedge \sigma_{ts}[M_{ts}] \xrightarrow{\tau_x} \sigma_{ts}[M'_{ts}]) \Rightarrow (T_s \xrightarrow{\tau_x} T'_s \wedge (T'_s, T'_t) \in C_{\tau,\tau_0}(\sigma_{st}[M'_{st}], \sigma_{ts}[M'_{ts}])))$

Si $(T_s, T_t) \in C_{\tau,\tau_0}(\emptyset, \emptyset)$, nous dirons T_s compatible avec T_t, et nous noterons $T_s \rightsquigarrow_{\tau,\tau_0} T_t$.

On peut distinguer trois parties dans cette définition, chacune constituée d'une paire d'équations. Le premier couple impose que, lorsqu'un message est envoyé, il soit placé sur le canal correspondant et que les types obtenus après évolution soient compatibles. La seconde partie décrit que lorsqu'un message peut être délivré alors il peut être reçu, et que les types obtenus doivent être compatibles. Enfin, le dernier couple exprime l'obligation d'homogénéité de l'évolution du temps.

Proposition *L*a relation de compatibilité des types est décidable.

4.3. *Décidabilité de la Relation de Compatibilité*

Nous décrivons maintenant formellement un algorithme résolvant le problème de décision de compatibilité entre deux types d'interfaces. Il nous semble important qu'apparaisse l'intérêt fondamental d'un tel algorithme. Nous pourrions appliquer dans notre cas les méthodes classiques de therorem proving ou de model checking, mais elles se révèleraient inefficaces car inadaptées aux spécificités du système étudié. Le model checking souffre ainsi du problème d'explosion de la taille du graphe des marquages accessibles, auquel notre algorithme est insensible. Le theorem proving lui n'est en général que partiellement automatisable, alors que nous devons pouvoir vérifier la compatibilité d'interfaces (donc de types) lors de l'exécution.

4.3.1. *Description Informelle*

Le principe de l'algorithme repose sur les cinq observations suivantes :

- Une transition qui ne comporte pas de clause *until*, lorsqu'elle assigne à une horloge c une valeur fixe, influence toute transition future contrainte par la valeur de c, jusqu'à ce que c soit modifiée à nouveau par une transition ultérieure.

- L'intervalle de temps dans lequel peut se produire une telle transition est donné par le type, se référent à la valeur d'une horloge donnée c.
- Un intervalle utilisant la valeur de c comme référence dépend de la transition précédante ayant amené l'affectation de c à une valeur fixe.
- Quand une horloge c est modifiée lors d'une transition utilisant la clause *until*, la valeur de c dépend toujours d'une transition plus ancienne que la transition actuelle.
- L'instant de déclenchement d'une transition non-*until* due à une réception de message est lié à la transition d'envoi du message : il arrive entre τ_0 et $\tau_0 + \tau$ unités de temps après celle-ci.

L'exécution de l'algorithme peut être représenté par une machine à états, chaque transition consistant en la vérification des contraintes d'envoi et de réception d'un message. Á partir d'un état, il existe une transition pour chaque message dont l'envoi est possible à partir de l'état courant. Cela assure l'évaluation de tous les états possibles du système. Si un état est rencontré où un message pouvant être émis peut ne pas être reçu, l'algorithme s'interrompt avec un résultat négatif. Dans le cas contraire, la rencontre d'un état déjà typé correctement permet l'arrêt d'une branche d'évaluation (point fixe) avec un résultat positif partiel. Un point fixe se définit aisément en comparant les positions relatives des types. Si toutes les étapes mènent à un point fixe, les types sont déclarés compatibles.

Des considérations précédentes, nous déduisons les règles suivantes permettant l'évaluation des horloges et des contraintes sur lesquelles reposent les envois et réceptions de messages :

- Chaque transition possible est nommée, et son nom n est associé à au moins un quintuplet $n = (n', c, \tau_{min}, \tau_{max}, Assignments)$. L'horloge c est celle dont la valeur sert de base à l'autorisation de la transition présente, avec la contrainte $[\tau_{min}, \tau_{max}]$. n' est le nom de la transition précédente durant laquelle l'horloge c a été assignée à une valeur fixe. $Assignments$ sont les modifications apportées aux horloges pas la transition courante.
- Un envoi de message est associé à un quintuplet.
- Une réception de message est associée à un ou deux quintuplet. L'un deux relie la réception à la transition précédente du même type, et l'autre à l'envoi du message correspondant. Dans ce dernier cas, le nom de l'horloge, qui n'est pas utile, est toujours c_{-1}.
- L'origine des temps est nommée n_0.
- On peut toujours appuyer la comparaison des intervalles d'envoi et de réception possibles pour un message sur une transition antérieure, en évaluant le temps écoulé depuis lors pour chaque interface.

L'état initial comprend les quintuplets $n_0 = (n_0, c_{-1}, 0, 0, (c_0 = 0, \ldots, c_n = 0))$ et $n_1 = (n_0, c_{-1}, \tau_0, \tau_0 + \tau, (c_0 = 0, \ldots, c_n = 0))$ (transitions initiales de la source et de la cible), ainsi que les deux interfaces dans leur état respectif de création. La procédure d'inférence précédente est ensuite appliquée, en créant une nouvelle étape de vérification concurrente chaque fois qu'un message peut être envoyé.

4.3.2. *Description Formelle*

Nous considèrons deux types T_s et T_t, écrits selon la syntaxe donnée dans la Table 2. Le type T_s a s_c équations de contrôle et s_s équations de flot, alors que T_t a t_c équations de contrôle et t_s équations de flot. Chaque équation i d'un état x s'écrit : c_i in $[\tau_{min_i}, \tau_{max_i}]$ m_i $\langle until\ \tau_{max_i} \rangle$; A_i. La clause "*until*" est optionnelle, et A_i est l'ensemble d'affectations d'horloges.

Nous décrivons formellement l'algorithme à l'aide de deux règles de réécritures (équations (1),(2)). L'environnement Γ contient les états courants des types ainsi que l'ensemble des quintuplets associés aux transitions déjà effectuées. La première règle décrit l'état initial du système, et la seconde la procédure d'inférence.

$$\frac{-}{\Gamma \ \vdash \ \begin{array}{l} n_0 = (n_0, c_{-1}, 0, 0, (c_0 = 0, \ldots, c_n = 0)), \\ n_1 = (n_0, c_{-1}, \tau_0, \tau_0 + \tau, (c_0 = 0, \ldots, c_n = 0)) \end{array}} \tag{1}$$

$$\frac{\forall\, !m_i \in X, \text{possible}(\Gamma, X, i) \Rightarrow \left\{ \begin{array}{l} \text{valid}(\Gamma, X, i) \wedge \\ \text{newenv}(\Gamma, X, i) \ \vdash \ \{I_s(y_s, y'_s); I_t(y_t, y'_t)\} \end{array} \right.}{\Gamma \ \vdash \ \{I_s(x_s, x'_s); I_t(x_t, x'_t)\}} \tag{2}$$

L'équation (2) déclare que les configurations (x_s, x'_s) et (x_t, x'_t) sont compatibles si un message pouvant être émis par la partie X d'une interface sera assurément reçue par son vis-à-vis et que les états obtenus ensuite sont compatibles.

Le prédicat *valid* (Equation (3)) requiert que le vis-à-vis du type X effectuant l'envoi possède une opération correspondante m_j telle que les contraintes de l'envoi et de la réception se recouvrent assurément.

$$\text{valid}(\Gamma, X, i) = \left\{ \begin{array}{l} \exists\, j \text{ such that } (\Gamma \ \vdash \ \text{peer}(X).m_j \ \wedge \\ (\exists\, n_k \leq min(last\_reset(\Gamma, X, i), last\_reset(\Gamma, peer(X), j)) \text{ such that} \\ (\text{val}(\Gamma, X.c_i, n_k)).max + X.\tau_{min_i} \geq \\ \ \text{val}(\Gamma, \text{peer}(X).c_j, n_k).min + \text{peer}(X).\tau_{min_j}) \wedge \\ (\text{val}(\Gamma, X.c_i, n_k).min + X.\tau_{max_i} \leq \\ \ \text{val}(\Gamma, \text{peer}(X).c_j, n_k).max + \text{peer}(X).\tau_{max_j})) \end{array} \right. \tag{3}$$

$$\text{peer}(X) = \left\{ \begin{array}{l} x_t \text{ if } X = x_s \\ x'_t \text{ if } X = x'_s \\ x'_s \text{ if } X = x'_t \end{array} \right. \tag{4}$$

La fonction *val* retourne un intervalle représentant les valeur possibles d'une horloge au moment de la transition, le décompte du temps écoulé débutant après la transition n. La description formelle de ces deux fonctions est toutefois trop longue pour figurer ici, et nous n'en donnerons que le principe.

Depuis la dernière modification de l'horloge c à évaluer, un certain nombre de transitions ont été effectuées, chacune basée sur une horloge c_i et appliquant les affectations A_i. Si c est utilisée comme référence lors d'une transition, alors sa valeur est

connue jusqu'à cette dernière. Dans tous les autres cas, on utilisera les horloges c_i pour connaitre le temps écoulé durant chaque transition. Dans ces conditions, on construit un graphe orienté sans cycle possèdant des arcs valués par le non-déterminisme apporté par chaque transition. Sur ce graphe (lors de sa construction même), on applique un algorithme de recherche de chemin de coût minimum. Ceci est fait jusq'à obtenir la précision la plus importante possible concernant la valeur de c.

L'environnement obtenu après évolution des types est donné par la commande *newenv*. Les noms des transitions les plus récentes de X et de son ci-à-vis sont respectivement n_p et n_q. De nouveaux noms sont créés, et associés aux tuples appropriés. Les fonctions *high* et *low* donnent les estimations hautes et basses du temps écoulé lors de la transition.

$$\text{newenv}(\Gamma, X, i) = \begin{cases} \Gamma, n_{p+1} = (n_p, \text{low}(\Gamma, X.c_i, X.\tau_{min_i}), \\ \text{high}(\Gamma, X.c_i, X.\tau_{max_i}), X.A_i), \\ n_{q+1} = (n_q, \text{low}(\Gamma, \text{peer}(X).c_j, \text{peer}(X).\tau_{min_j}), \\ \text{high}(\Gamma, \text{peer}(X).c_j, \text{peer}(X).\tau_{max_j}), \text{peer}(X).A_j) \\ n_{q+1} = (n_{p+1}, c_{-1}, \tau_0, \tau_0 + \tau, \text{peer}(X).A_j) \end{cases} \quad (5)$$

5. *ArtOC*

ArtOC est un langage d'acteurs temporisés avec localités. Les localités servent uniquement pour l'instant à déterminer les contraintes réseaux entre les objets distants. Une localité contient des comportements évoluant en concurrence. Un comportement peut être un envoi de message, une réception, un délai, une instanciation d'objet concurrent, une création d'interface, ou l'adoption d'un comportement nommé (le *become* des acteurs). Toutes ces opérations sont courantes dans les formalismes existants. Nous affectons cependant une signification particulière à l'envoi de message contenant des interfaces cibles (comme déjà mentionné) afin de permettre l'initialisation ad-hoc des interfaces temporisées. Dans le reste de cette section, en parallèle avec la description informelle du langage, nous donnerons les éléments essentiels qui rendent le typage possible.

Une localité est définie comme suit :

$$\text{loc } LocId \text{ with } Dec \text{ in } B,$$

où $LocId$ est un nom de localité, B un comportement et Dec un ensemble de déclarations (interfaces publiques et comportements nommés). Les localités sont composées en utilisant l'opérateur "|".
La création d'une interface u de type T est écrite

$$\text{new } u : T \text{ in } B.$$

Deux interfaces privées doivent porter le même nom pour être en mesure de communiquer. Une interface privée source est inactive jusqu'à l'envoi d'une cible de même

nom sur le réseau, et une cible jusqu'à sa réception par un process comme paramètre d'un message.
L'envoi de message est syntaxiquement similaire à celui de CSP :

$$!u_c.m(\tilde{v}) > B$$

envoie le message m avec les arguments v sur l'interface u_c. Les arguments peuvent être des interfaces, de entiers ou des valeurs temporelles. Les interfaces publiques peuvent être envoyées librement. Lors de l'envoi, si l'interface est une cible, elle est perdue par l'émetteur. Les interfaces cibles suivent ce comportement, mais elles ne peuvent plus être envoyées une fois actives.
La réception sur une interface se fait en décrivant les comportements déclenchés pour tous les messages souhaités :

$$?u_c[m_1(\tilde{v}_1) = B_1, \ldots, m_n(\tilde{v}_n) = B_n].$$

On peut être prêt à recevoir en parallèle sur plusieures interfaces, en attendant au maximum τ unités de temps (τ peut être également nul ou infini) :

$$(\tau, @waiting\_time, transmission\_delay) \sum_{i=1}^{n} Recep_i > B.$$

Le nom *waiting_time* définit une variable qui reçoit le temps attendu, et *transmission_delay* le temps passé par le message dans le réseau. Le comportement B est adopté si aucun message n'est reçu dans le temps imparti.
L'opération de délai permet la référence aux variables temporelles dans son argument, en utilisant une algèbre à deux opérateurs (+ et -) :

$$\text{delay } (\nu) > B.$$

L'opération *create* créée un nouvel objet concurrent, et lui passe les interfaces (obligatoirement publiques ou inactives) $\tilde{u}$ avant d'exécuter B :

$$\text{create } A[\tilde{u}] > B.$$

Enfin, l'instanciation provoque l'adoption par un objet du comportement nommé A :

$$A[\tilde{u}]$$

se comporte comme A.

6. Sémantique Opérationnelle et Typage

Il ne nous est pas possible de donner ici les règles de sémantique opérationnelle et de typage, pour des limitations d'ordre spatial.

6.1. *Sémantique Opérationnelle*

Les points essentiels de cette sémantique concernent le temps et l'évolution des configurations lors de l'envoi de messages.

Le temps évolue uniformément dans toutes les localités, pour tous les objets et tous les messages, de manière similaire à [11, 7, 8]. Il progresse toujours du montant minimum autorisé par l'ensemble des parties. Seules les actions laissant explicitement passer le temps ont une durée non nulle. Le temps diverge nécessairement car le langage interdit les boucles ne comportant pas d'opérations faisant s'écouler le temps (voir Table 8). Toute synchronisation manquée fait évoluer le système vers l'état particulier *Error,* où plus aucune opération n'est possible.

Quand un message est émis, il est placé en parallèle avec les localités. Les communications locales sont représentées par des messages au temps de transmission nul. Quand un message a passé τ_0 unités de temps dans un canal, il peut devenir *disponible* à n'importe-quel instant jusqu'à $\tau_0 + \tau$. Une fois disponible, il doit être reçu immédiatement sinon la configuration évolue vers *Error*. Cela est également le cas si aucune interface de la localité cible ne possède l'opération disponible correspondante.

6.2. *Sémantique Statique*

L'algorithme de typage des objets constitue la deuxième partie du système de vérification. Une fois vérifiée la compatibilité des types d'interfaces utilisés, l'algorithme de typage doit lui vérifier l'instanciation correcte de ces types ainsi que la conformité des comportements des objets avec les interfaces qu'ils possèdent. Les contraintes sur la création et l'envoi d'interfaces temporisées ont déjà été exposée, et sont statiquement typables. On s'attache donc maintenant à décrire le dernier aspect, dont le principe est assez simple.

De manière similaire à l'algorithme précédent, on calcule les évolutions du temps possibles pour chaque objet, et on estime les valeurs des horloges des interfaces qu'ils possèdent. Si une inadéquation existe entre un comportement et une interface, alors l'objet est mal typé.

Il n'existe que deux instructions laissant passer le temps. Des deux, seule la réception de message induit du non-déterminisme : on ne connait pas le temps écoulé lors de la réception avant l'exécution du programme spécifié. Ce temps est donc désigné par une variable temporelle, dont la valeur est incluse dans un intervalle $[\tau_{min}, \tau_{max}]$. La valeur d'une horloge qui n'est pas modifiée lors d'une réception de message dépend du temps écoulé lors de cette réception. Par conséquent, chaque horloge possède une liste de variables temporelles dont sa valeur dépend. Ainsi calcule-t-on toutes les valeurs possibles d'une horloges : on somme les bornes inférieures des intervalles des variables référencées d'une part, et des bornes supérieures d'autre part. Lorsqu'une horloge passe à une valeur fixe lors d'une transition, sa liste de dépendances devient

vide. Enfin, les opérations de délai ne font que modifier la liste des horloges en fonction des variables temporelles qu'elles utilisent, sans changer le principe de calcul.

7. Un Exemple

On présente maintenant un type source d'audio et de video compatible avec l'exemple de la Section 3.1, tant que $\tau \leq 40$.

Type $\triangle! \, AV_s :=$

stream$[c_1, c_2]$:	x_1	$= \emptyset$
	x_2	$= c_1$ in $[40, 40]$ $video()$ until 40 ; stream.$x_2(c_1 = 0)$
		$+ c_2$ in $[100, 100]$ $audio()$ until 100 ; stream.$x_2(c_2 = 0)$
control$[c_3]$:	x_3	$= c_3$ in $[0, 5000]$ $?start()$; control.$x_4(c_3 = 0)$
	x_4	$= c_3$ in $[0, 4995]$ $!start\_ack()$; (stream.x_2, control.x_5)$(c_3 = 0)$
	x_5	$= c_3$ in $[4995, \infty]$ $?stop()$; control.$x_6(c_3 = 0)$
	x_6	$= c_3$ in $[0, 995]$ $!stop\_ack()$; (stream.x_1, control.x_7)$(c_3 = 0)$
	x_7	$= \emptyset$

8. Conclusion

Nous avons montré comment vérifier la sûreté de protocoles temps-réel. Les applications sont nombreuse, et nous pensons répondre de manière plus judicieuse aux exigences présentées en termes de simplicité de spécification et d'efficacité de vérification. Nos travaux trouvent leur place comme poursuite et extension des nombreux travaux existants sur le typage comportemental explicite. Par ailleurs, nos travaux (le formalisme que nous avons défini) entre dans le cadre architectural d'ODP [5]. D'un point de vue génie logiciel, le typage explicite (la spécification de types par le programmeur) prend toute sa valeur pour des configurations temporisées. Il rend le travail de vérification plus cohérent, et pousse l'utilisateur à la rédaction de systèmes plus raisonnés (sans abandonner la souplesse). Les interfaces telles que présentées ici peuvent par exemple être vues comme exprimant des offres et des exigences de QoS. On a alors un système reconfigurable exprimé en ces termes, dont la cohérence entre offres et exigences a été vérifié statiquement.

Nous prévoyons d'étendre ces travaux de différentes manières. Tout d'abord, il semble intéressant de nous tourner vers la préservation par des implémentations de propriétés prouvées sur leurs modèles temps-réel. En effet, c'est ce caractère temporisé qui rend les propriétés prouvées sur un modèle difficile à conserver lors de l'implémentation. Une utilisation judicieuse de la réflexivité et des résultats de typage *lors de l'exécution* pourrait rencontrer des succès probants en ce sens. Ensuite, nous souhaiterions modèliser la gestion des fautes dans nos modèles, via le langage ou bien encore des constructions sémantiques. Les fautes peuvent être de deux natures : les erreurs du réseau (perte, délais excessifs) et les erreurs des acteurs (manquement au contrat). Là encore les informations de typages pourraient être utilisées à l'exécution par un calcul d'observateurs afin de décider des mesures à prendre lors d'un com-

portement inconvenant. À cette occasion un typage "relâché" pourrait être créé, qui permettrait la garantie de la sûreté sans pour autant imposer les contraintes les plus strictes sur le modèle.

9. Bibliographie

B. Nielsen and G.A. Agha, *Towards Reusable Real-Time Objects*, Annals of Software Engineering, 7 : 257-282, 1999.

J. Hooman, *Compositional Verification of Real-Time Applications*, Proceedings Compositionality - The Significant Difference (COMPOS '97), LNCS 1536, Springer-Verlag, pp. 276-300, 1998.

J.-P. Courtiat and R.C. de Oliveira, *A Reachability Analysis of RT-LOTOS Specifications*, In Proc. FORTE'95, Montreal, Canada, October 17-20, 1995, Chapman & Hall.

N. Bjørner, Z. Manna, H. B. Sipma and T. E. Uribe, *Deductive Verification of Real-Time Systems Using STeP*, Proc. of ARTS'97, vol. 1231 of LNCS, pp. 22-43, Springer-Verlag, May 1997.

Open Distributed Processing Reference Model, parts 1,2,3,4, ISO/IEC IS 10746-1..4 or ITU-T X901..4, 1995.

E. Najm, A. Nimour and J-B Stefani, *Infinite Types for Distributed Objects Interfaces*, Proc. of IFIP conf. FMOODS'99, Kluwer, Feb 1999.

L. Léonard and G. Leduc, *A formal definition of time in LOTOS*, Formal Aspects of Computing,(1998) 10 : 248-266.

C. J. Fidge and J. J. Zic, *An expressive real-time CCS*, In Proc. Second Australasian Conference on Parallel and Real-Time Systems, pages 365-372, Fremantle, September 1995. Longer version available as SVRC TR 94-27.

K.G. Larsen, P. Petterson, W. Yi, *Compositional and Symbolic Model Checking of Real-Time Systems*, Appears in 16th IEEE Real-Time Systems Symposium, RTSS '95 Proceedings, 1995.

M. Abadi and L. Lamport, *Conjoining specifications*, ACM Transactions on Programming Languages and Systems, 17(3) :507-534, May 1995.

M. Abadi and L. Lamport, *An old-fashioned recipe for real-time*, ACM Transactions on Programming Languages and Systems, 16(5) :1543-1571, September 1994.

R. Alur and D. Dill, *Automata for Modelling Real-Time Systems*, Theoretical Computer Science, 126(2) :183-236, April 1994.

T.A. Henzinger, Z. Nicollin, J. Sifakis and S. Yovine, *Symbolic Model Checking for Realt-time Systems*, in Logic in Computer Science, 1992.

O. Nierstrasz, *Regular Types for Active Objects*, Object-Oriented Software Composition, Prentice Hall, 1995.

A. Ravavra and V.T. Vasconcelos, *Behavioral Types for a Calculus of Concurrent Objects*, Euro-Par'97, Springer-Verlag, 1997.

H. Schulzrinne, A. Rao and L. Lanphier, *Real Time Streaming Protocol (RTSP)*, IETF draft 09, 1998, http ://www.cs.columbia.edu/ hgs/rtsp/.

I. Wakeman, A. Jeffrey, R. Graves, T. Owen, *Designing a Programming Language for Active Networks*, submitted to Hipparch special issue of Network and ISDN Systems, http ://www.cogs.susx.ac.uk/projects/safetynet/.

abbreviation	**meaning**
ρ	interface role
μ	interface mode
B	behavior expression
A	behavior name
P	formal parameter for named behavior
p	effective parameter for named behavior
T	interface type
u	interface name
a	immediate action
t	lasting action
τ	clock value
v	method parameter (formal or effective)
$VarId$	time variable Id
$m, m_1, \ldots, m_n$	method names
L	location definition
C	parallel configuration of objects

$$
\begin{array}{rcl}
\rho & ::= & client \mid server \\
v & ::= & u : \mathrm{T} \\
Recep & ::= & ?u[m_1(\tilde{v}_1) = B_1, \ldots, m_n(\tilde{v}_n) = B_n] \\
B & ::= & a > B \mid t > B \\
 & \mid & \text{new } u : T \text{ in } B \\
 & \mid & A[\tilde{u}] \\
 & \mid & 0 \\
a & ::= & !\, u.m(\tilde{u}) \\
 & \mid & \text{create } A[\tilde{u}] \\
t & ::= & (\tau, @VarId, VarId) \sum_{i=1}^{n} Recep_i \\
 & \mid & \text{delay } (\tau) \\
E & ::= & B \\
 & \mid & B|B \\
Dec & ::= & A[\tilde{v}] = t > B \\
 & \mid & \langle \rho u : T \rangle \\
 & \mid & Dec, Dec \\
L & ::= & \text{loc } LocId \text{ with } Dec \text{ in } E \\
C & ::= & L \\
 & \mid & C|C
\end{array}
$$