RenPar'2000
Besançon, 19 juin - 22 juin 2000

Analyse Statique pour la Sûreté des Applications Distribuées

Matthieu Martel et Marc Gengler

Laboratoire d'Informatique de Marseille
Parc Scientifique et Technologique de Luminy
163 Avenue de Luminy - Case 901 F
13288 Marseille Cedex 9, France
[Matthieu.Martel,Marc.Gengler]@esil.univ-mrs.fr

Résumé
Dans cet article nous décrivons le principe de fonctionnement d'une analyse de flot de contrôle pour des langages à base de communications synchrones. Cette analyse utilise des automates d'états finis afin de construire une approximation de la topologie des communications réalisées par les programmes. En comparaison avec d'autres analyses de flot de contrôle, les informations sur la topologie des communications permettent d'améliorer sensiblement la précision des valeurs abstraites attachées aux points de réception. Nous montrons comment cette analyse permet de vérifier le bon fonctionnement d'un mécanisme d'allocation de circuits virtuels similaire à celui utilisé dans ATM. La précision apportée par le calcul de la topologie des communications de l'application s'avère nécessaire pour l'obtention de ces résultats.

Mots-clés : Analyse de flot de contrôle, ATM, Concurrent ML, Automates Produits.

---

1. Introduction

Les analyses statiques sont largement utilisées pour établir des propriétés générales satisfaites par des programmes, quel que soit leur contexte d'exécution [8]. Parmi les applications les plus répandues figurent les optimisations à la compilation et les preuves de correction de programmes. Cependant, ces analyses ont le plus souvent été conçues pour des langages séquentiels et, avec le développement des systèmes distribués, apparaissent de nouvelles problématiques ainsi que de nouvelles classes d'applications.

Récemment, de nombreux travaux de recherche ont été consacrés à l'utilisation d'analyses statiques dans le but d'établir des propriétés liées à la *sûreté* et à la *sécurité* des systèmes distribués, c.à.d. au bon fonctionnement opérationnel des applications (absence d'erreurs) et au respect de l'intégrité des données (problèmes de confidentialité par exemple) [2, 3, 4, 7].

D'un point de vue technique, diverses méthodes d'analyse statique ont été développées, parmi lesquelles figurent les *analyses de flot de contrôle* (CFA) [2, 5, 6]. Ces analyses ont pour but de calculer pour chaque point p d'un programme un sur-ensemble des points susceptibles d'être exécutés immédiatement après p. La *précision* d'une analyse dépend de la qualité du sur-ensemble obtenu. Cependant les propriétés qu'il est possible de prouver à l'aide d'une analyse de flot de contrôle dépendent sensiblement de la précision de cette dernière.

Dans cet article, nous décrivons le principe de fonctionnement d'une CFA construisant une approximation de la topologie des communications réalisées par les programmes afin d'accroître la qualité de ses résultats [5]. Ce calcul permet d'attacher aux points de reception un sur-ensemble plus précis des valeurs pouvant être transmises lors des communications. Tout d'abord, nous ordonnons les communications réalisées par chaque processus en construisant pour chacun d'eux un automate d'états finis en fonction de leur code. Ensuite, nous calculons une approximation conservatrice des interactions entre processus en nous fondant sur un automate réduit dont la taille est polynômiale en celle du programme à analyser.

Cette analyse a été implémentée et nous décrivons son fonctionnement en prenant pour exemple le code d'un mécanisme d'allocation de circuits virtuels, similaire à celui utilisé dans le réseau ATM [1, 11] pour la création dynamique de liens à haut débit. Nous montrons comment les résultats fournis par l'analyse peuvent être utilisés our garantir le bon fonctionnement de notre implémentation de ce mécanisme. La

précision apportée par le calcul de l'approximation de la topologie des communications réalisées par le programme est nécessaire pour obtenir ces résultats.

2. Analyse de flot de contrôle

Dans cette Section, nous décrivons le principe de fonctionnement de l'analyse de flot de contrôle utilisée. La Section 2.1 décrit les principaux problèmes rencontrés dans l'analyse de programmes concurrents et la Section 2.2 présente les techniques utilisées.

2.1. Description générale

Une analyse de flot de contrôle détermine l'ensemble des points d'un programme susceptibles de succéder à un point donné dans l'ordre imposé par l'exécution. Si ce problème est simple dans le cas des langages séquentiels dépourvus de procédures, il en est autrement pour des langages plus évolués.

De nombreux travaux ont été consacrés à l'analyse du flot de contrôle de langages fonctionnels séquentiels du premier ordre ou d'ordre supérieur [8, 10]. Dans ce cadre, l'analyse consiste à déterminer l'ensemble des fonctions susceptibles d'être appelées à un point donné.

Cette tâche se trouve compliquée par l'adjonction de primitives de gestion du parallélisme (pour les communications ou la création de nouveaux processus par exemple). En effet, ces primitives définissent de nouvelles structures de contrôle que les analyses doivent traiter au mieux afin de conserver leur précision. Dans un langage fonctionnel d'ordre supérieur, il est par ailleurs possible de communiquer des noms de canaux et des fonctions (ces dernières peuvent contenir d'autres communications). Cette forme de mobilité du code complique à son tour le calcul statique du flot de contrôle.

Considérons par exemple le programme suivant écrit en Concurrent ML [9] composé de deux processus p_1 et p_2. Chaque instruction est annoté par une étiquette unique. p_1 envoie une fonction sur le canal α et reçoit ensuite, toujours sur α, une fonction dont il calcule la valeur en 1. p_2 recoit une fonction sur le canal α qu'il nomme f et renvoit sur α une fonction g ou une fonction h, selon la valeur de f en 0.

p_1 : $\texttt{let}^1$ foo = $\texttt{send}^2$ α ($\texttt{fun}$ x $\rightarrow$ x)^3
    $\texttt{in}$ (($\texttt{receive}^4$ α) 1^5)^6

p_2 : $\texttt{let}^7$ f = $\texttt{receive}^8$ α
    $\texttt{in}$ $\texttt{if}^9$ (f 0^{10})^{11} $=^{12}$ 0^{13} $\texttt{then}$
        $\texttt{send}^{14}$ α g
    $\texttt{else}$
        $\texttt{send}^{15}$ α h

Un résultat possible pour une analyse de flot de contrôle de ce programme est décrit par le tableau suivant qui associe à chaque point ℓ du programme l'ensemble $\mathcal{C}(\ell)$ des étiquettes des fonctions en lesquelles l'expression d'étiquette ℓ peut s'évaluer. $\mathcal{C}(l)$ est appelé *valeur abstraite* associée au point ℓ. $\mathcal{C}(\ell) = \varnothing$ pour les simples valeurs. Ces étiquettes n'apparaissent pas dans le tableau. $\mathcal{E}(id)$ décrit la valeur abstraite de l'identificateur id dans l'environnement de l'analyse, c.à.d. l'ensemble des étiquettes des fonctions pouvant être affectée à id.

Etiquette	2	3	4	7	8	9	12	14	15
Valeur	$\{3\}$	$\{3\}$	$\mathcal{E}(g) \cup \mathcal{E}(h)$	$\mathcal{E}(g) \cup \mathcal{E}(h)$	$\{3\}$	$\mathcal{C}(14) \cup \mathcal{C}(15)$	$\mathcal{E}(=)$	$\mathcal{E}(g)$	$\mathcal{E}(h)$

$\mathcal{C}(3) = \{3\}$ indique que seule la fonction étiquetée 3 peut apparaître au point 3. $\mathcal{C}(2) = \{3\}$ car une instruction `send` s'évalue en la valeur envoyée. La valeur reçue en 8 est celle qui a été envoyée en 2, on a donc $\mathcal{C}(8) = \mathcal{C}(3) = \{3\}$. Une analyse statique n'exécutant pas les calculs présents dans le programme, la condition du `if` ne peut être évaluée. La seule approximation possible consiste alors à associer au point 9 l'union des résultats possibles pour les deux alternatives, $\mathcal{C}(9) = \mathcal{C}(14) \cup \mathcal{C}(15)$. Pour les mêmes raisons, il n'est pas possible de déterminer laquelle des émissions présentes dans la conditionnelle aura lieu. Ainsi, la valeur reçue en 4 est parmi celles envoyées en 14 et en 15, c.à.d. $\mathcal{C}(4) = \mathcal{E}(g) \cup \mathcal{E}(h)$.

2.2. Principe de fonctionnement

Pour les langages à parallélisme explicite, la précision d'une analyse de flot de contrôle dépend sensiblement des approximations réalisées aux points de réception.

Une première méthode consiste à collecter dans un environnement $\mathcal{K}(\alpha)$ l'ensemble des valeurs abstraites potentiellement émises sur le canal α et à affecter le résultat à tous les points de réception susceptibles d'utiliser ce canal [2, 6]. Ceci est résumé par les équations suivantes.

$$(\texttt{send}^{\ell_0}\ e_1^{\ell_1}\ e_2^{\ell_2} \in \text{prg}) \ \Rightarrow\ \forall \alpha \in \mathcal{C}(\ell_1), \mathcal{C}(\ell_2) \subseteq \mathcal{K}(\alpha) \qquad (1)$$

$$(\texttt{receive}^{\ell_0}\ e^{\ell_1} \in \text{prg}) \ \Rightarrow\ \mathcal{C}(\ell_0) = \bigcup_{\alpha \in \mathcal{C}(\ell_1)} \mathcal{K}(\alpha) \qquad (2)$$

Appliquée au programme de la Section 2.1, cette méthode produit le résultat donné dans la tableau ci-dessous, avec $\mathcal{K}(\alpha) = \{3\} \cup \mathcal{E}(g) \cup \mathcal{E}(h)$.

Etiquette	2	3	4	7	8	9	12	14	15
Valeur	$\{3\}$	$\{3\}$	$\mathcal{K}(\alpha)$	$\mathcal{E}(g) \cup \mathcal{E}(h)$	$\mathcal{K}(\alpha)$	$\mathcal{E}(g) \cup \mathcal{E}(h)$	$\mathcal{E}(=)$	$\mathcal{E}(g)$	$\mathcal{E}(h)$

La valeur abstraite associée aux points de réception 4 et 8 est l'union des valeurs abstraites émises aux points 2, 14 et 15. Ce résultat est moins précis que celui présenté au cours de la Section 2.1. La raison en est que cette analyse ne tient pas compte du fait que la réception étiquetée 8 (resp. 4) peut seulement communiquer avec l'emission d'étiquette 2 (resp. 14 et 15).

Ceci met en évidence que, dans le cas des langages à parallélisme explicite, une analyse de flot de contrôle doit, pour être précise, utiliser une approximation fine de la topologie des communications réalisées par le programme. Dans ce but, nous avons développé une telle analyse [5] dont le principe de fonctionnement peut se résumer en deux points.

Ordonner les points de communication de chaque processus. La topologie des communications d'un programme distribué dépend de la façon dont les points de communication sont ordonnés sur chaque processus. Ceci est lié au flot de contrôle sur les parties séquentielles du programme mais dépend aussi des communications précédentes. Dans un langage d'ordre supérieur, des fonctions contenant des communications peuvent être transmises et appliquées dans le processus qui les reçoit. Nous traitons ce point de la manière suivante. Nous construisons un automate d'états finis $\mathcal{A}$ qui indique l'ordre de succession des points de communication. A chaque sous-expression du programme correspond un automate et les transitions entre différents automates indiquent l'ordre d'évaluation des expressions associées. Les transitions qui correspondent à des pas de réduction séquentiels sont étiquetées ε et une ℓ-transition indique l'occurrence d'un point de communication d'étiquette ℓ. Le résultat est une approximation du schéma de communication de chaque processus.

Déterminer les interactions entre processus. Connaissant la façon dont les points de communication sont ordonnés sur chaque processus, nous devons déterminer une approximation de leurs interactions. Une première manière d'aborder ce problème consiste à construire le produit des automates définis dans la phase précédente. A partir d'une collection $\mathcal{C} = (\mathcal{A}_p)_{p \in \text{Proc(prg)}}$ d'automates d'états finis décrivant les différents processus du programme, nous montrons que l'automate produit $\mathrm{A}_{\otimes}^{\natural}(\mathcal{C})$ décrit une approximation correcte des communications que le programme peut réaliser. Cependant, d'un coté la taille de $\mathrm{A}_{\otimes}^{\natural}(\mathcal{C})$ peut être exponentielle en la taille du programme, tandis que de l'autre, il renferme plus d'informations que véritablement nécessaire pour les besoins de l'analyse. Aussi, nous introduisons un *automate réduit* $\mathrm{A}_{\otimes}^{\sharp}(\mathcal{C})$, de taille polynomiale en la taille du programme à analyser, et nous prouvons que $\mathrm{A}_{\otimes}^{\sharp}(\mathcal{C})$ est une approximation correcte de $\mathrm{A}_{\otimes}^{\natural}(\mathcal{C})$. Le résultat est un sur-ensemble des communications possibles entre processus.

Intuitivement, les seules informations nécessaires pour les besoins de l'analyse sont les paires d'emetteurs et de récepteurs susceptibles d'interagir ensemble. Par exemple, nous souhaitons connaître l'ensemble des points d'émission pouvant communiquer avec un point de réception particulier, indépendamment de toute notion d'ordre, c.à.d. de la place de cette communication dans une trace de l'exécution du programme. L'automate produit $\mathrm{A}_{\otimes}^{\sharp}(\mathcal{C})$ contient toutes les séquences de communication possibles, ce qui est trop précis pour nos besoins.

Nous construisons un automate permettant de déterminer les points du programme pouvant être actifs après qu'une communication donnée ait eu lieu. Soient ℓ_s et ℓ_r deux points qui correspondent aux deux composantes d'une communication c. Un état q_c représente c dans $\mathrm{A}_{\otimes}^{\sharp}(\mathcal{C})$. On attache à cet état l'ensemble des points $\mathcal{P}(q_c)$ pouvant être actifs après exécution de c. Une seconde communication c' entre deux

nouveaux points ℓ'_s et ℓ'_r est autorisée si $\ell'_s \in \mathcal{P}(q_c)$ et $\ell'_r \in \mathcal{P}(q_c)$. Dans ce cas nous ajoutons une c-transition allant de q_c à $q_{c'}$ et $\mathcal{P}(q_{c'})$ est mis à jour. Ainsi, si deux communications différentes c_1 et c_2 peuvent suivre c, deux c-transitions relieront q_c à q_{c_1} et q_{c_2}. De cette manière, $\mathrm{A}^{\sharp}_{\otimes}(\mathcal{C})$ indique que c_1 et c_2 sont deux communications possibles sans en préciser l'ordre. La taille de l'automate produit réduit ainsi obtenu est $O(n^3)$ où n est la taille du programme à analyser.

Cette analyse a été définie pour un sous-ensemble de Concurrent ML [9] comprenant, outre les primitives de communication, la création dynamique de nouveaux canaux et de processus. La preuve de correction se présente sous la forme d'une propriété de *subject reduction*. Appliquée au programme donné précedemment, cette analyse produit les résultats donnés par le tableau de la Section 2.1.

3. Application à la vérification d'un mécanisme d'allocation de circuits virtuels

Dans cette Section, nous examinons les résultats fournis par une implémentation de l'analyse de flot de contrôle décrite au cours de la Section 2.2. Nous montrons comment cette CFA peut être utilisée afin de vérifier la sûreté d'une implémentation d'un mécanisme d'allocation de circuits virtuels similaire à celui utilisé dans ATM [1, 11].

3.1. Allocation de circuits virtuels

La création d'un circuit virtuel se déroule selon le schéma de la Figure 1. Les différents nœuds du réseau sont reliés à leurs voisins par des canaux de contrôle. Ces canaux sont utilisés afin de propager un message de demande de création de circuit jusqu'au destinataire. Ce dernier établit un lien avec son voisin qui procède de même avec son autre voisin, jusqu'à ce que le nœud source soit atteint (toujours en utilisant les canaux de contrôle). Chaque nœud intermédiaire se charge ensuite de propager les données arrivant sur son lien d'entrée vers son lien de sortie.

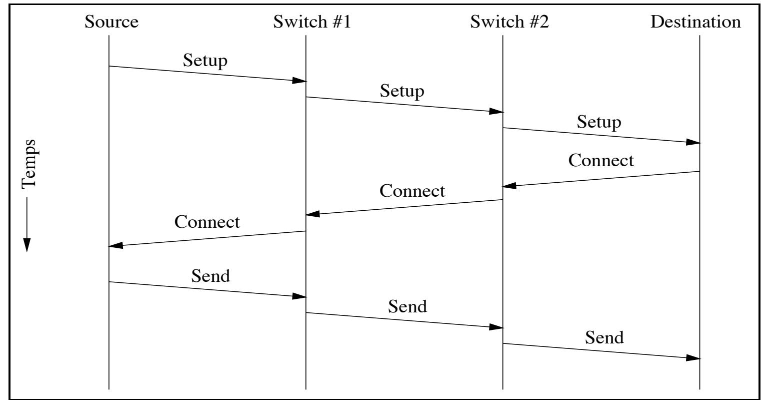

FIG. 1 – *Principe de fonctionnement du mécanisme de création de circuit virtuel.*

Une implémentation complète de ce mécanisme a été réalisée en Concurrent ML, le langage traité par l'analyse. Pour des raisons de clarté, nous en donnons une version simplifiée, écrite dans un pseudo-langage impératif (voir Figure 2). Par ailleurs, un seul nœud intermédiaire est présent entre la source et la destination.

Les trois processus p_1, p_2 et p_3 sont supposés s'exécuter sur des noeuds différents du réseau. #ctrl_12 (resp. #ctrl_23) est un canal de contrôle bidirectionnel reliant les nœuds correspondant à p_1 et p_2 (resp. p_2 et p_3). De plus, comme p_1 crée deux circuits différents a et b, les morceaux de code donnés pour les processus p_2 sont exécutés deux fois. Dans notre implémentation, nous utilisons une analyse de flot de contrôle très simple pour les parties séquentielles du programme (0-CFA [8]). Ceci nous oblige à dupliquer ces morceaux de code afin de ne pas perdre en précision. Cependant ces duplications pourraient être évitées par l'utilisation d'une analyse plus élaborée (k-CFA). Ainsi, dans la Figure 2, les deux étiquettes annotant certaines intructions font références aux étiquettes uniques de deux copies du morceaux de code en question.

p_1, p_2 et p_3 fonctionnent comme suit. p_1 demande successivement la création des deux circuits. La fonction `connect` envoie un message d'initialisation sur le canal de contrôle `#ctrl_12` et affecte le nom du canal à utiliser pour a à la variable `virt_ch_a` lorsque celui-ci est reçu sur `#ctrl_12`. L'opération est répétée pour la création de b. Enfin, des données sont transmises sur `virt_ch_a` et `virt_ch_b`.

p_2 correspond au nœud situé entre la source et la destination. Lorsqu'une demande d'allocation de circuit est reçue sur `#ctrl_12`, elle est transmise à la destination via le canal de contrôle `#ctrl_23`. p_2 reçoit ensuite le nom du canal de sortie à utiliser, qu'il affecte à la variable `ch_out` et crée un nouveau canal `ch_in` (instruction `channel()`) qu'il transmet au processus source. Enfin, un nouveau processus est

Processus p_1 (source)	Processus p_2 (switch)	Processus p_3 (destination)

<pre>function connect(dest) =
{ send #ctrl_12 init(dest);
  ch ← receive^{7,19} #ctrl_12;
  return ch;
}
function main()
{ virt_ch_a ← connect(3)^{13};
  virt_ch_b ← connect(3)^{25};
  send virt_ch_a data_a^{30};
  send virt_ch_b data_b^{35};
}</pre>
	<pre>...
	ctrl_msg ← receive^{51,84} #ctrl_12;
	send #ctrl_23 ctrl_msg;
	ch_out ← receive^{56,89} #ctrl_23;
	ch_in ← channel()^{58,91};
	send #ctrl_12 ch_in;
	fork (while true do
	      send ch_out
	           (receive^{62,95} ch_in)
	      );
	...</pre>
		<pre>...
		ctrl_msg ← receive #ctrl_23;
		ch_dest ← channel()^{124,129};
		send #ctrl_23 ch_in;
		receive^{132,134} ch_in;
		...</pre>

FIG. 2 – *Implémentation du mécanisme de création de circuits virtuels. La procédure connect ainsi que les morceaux de code des processus p_2 et p_3 sont exécutés deux fois.*

créé qui retransmet sur `ch_out` les valeurs reçues sur `ch_in`. L'opération est répétée pour b.

Enfin p_3 correspond au nœud destination. A l'arrivée des messages de contrôle, p_3 crée un nouveau canal qui est affecté à la variable `ch_in` et transmise à p_2 via le canal de contrôle `#ctrl_23`. Une fois les circuits créés, p_3 reçoit des données sur les canaux correspondant à a et b.

3.2. Analyse

Nous avons utilisé l'analyse décrite à la Section 2.2 afin de vérifier le bon fonctionnement d'une implémentation complète du programme décrit à la Figure 2. Nous présentons ici les résultats obtenus qui ont été transposés au programme de la Figure 2. Les tableaux ci-dessous donnent les valeurs abstraites obtenues pour les principaux points du programme ainsi que la valeur abstraite associée à certaines variables.

Comme indiqué précédemment, les morceaux de code correspondant à p_2 et p_3 sont exécutés deux fois. Dans le tableau donnant les valeurs abstraites des variables, nous indiquons à laquelles des deux exécutions il est fait référence par (a) et (b).

Etiquette	7	13	19	25	30	35	51	56	58
Valeur	{58}	{13}	{91}	{25}	{30}	{35}	{13}	{124}	{58}

Etiquette	62	84	89	91	95	124	129	132	134
Valeur	{30}	{25}	{129}	{91}	{35}	{124}	{129}	{30}	{35}

Variable	#ctrl_12	#ctrl_23	virt_ch_a	virt_ch_b	ctrl_msg (a)	ctrl_msg (b)
Valeur	{1}	{2}	{58}	{91}	{13}	{25}

Variable	ch_in (a)	ch_in (b)	ch_out (a)	ch_out (b)	ch_dest (a)	ch_dest (b)
Valeur	{58}	{91}	{124}	{129}	{124}	{129}

La principale observation concerne les valeurs abstraites reçues par p_3 aux points 132 et 134. Ces valeurs, {30} et {35}, sont celles émises par p_1 sur les circuits a et b. Ceci valide le mécanisme d'allocation des circuits. Les valeurs reçues sur a (resp. b) sont celles qui ont été envoyées sur ce circuit et seulement celles-ci. En outre, cela nous assure que le processus p_2 n'intervertit pas les valeurs reçues sur ses liens d'entrée et retransmises sur ses liens de sortie.

Concernant la création du circuit a, nous pouvons vérifier que que les variables `virt_ch_a` et `ch_dest` (a) qui sont liées aux deux extrémités du circuit ont pour valeurs abstraites {58} et {124}, ce qui correspond aux canaux créés au points d'étiquettes 58 et 124. Le nouveau processus créé par p_2 retransmet sur le canal créé en 124 les valeurs arrivant sur le canal créé en 58 (variables `ch_in` (a) et `ch_out` (a)). La transmission des données de la source vers la destination est donc assurée. De plus, il serait possible de vérifier que les canaux de valeurs abstraites {58} et {124} ne sont pas utilisés en d'autres points du programme. Ainsi, les données envoyées sur le circuit virtuel a sont uniquement transmises au destinataire. Les mêmes constatations peuvent être faites pour le circuit b.

Enfin, notons que pour obtenir ces résultats, il est nécessaire d'utiliser une analyse construisant une approximation de la topologie des communications réalisées par le programme. En effet, les mêmes canaux

de contrôle `#ctrl_12` et `#ctrl_23` sont utilisés pour la création des deux circuits virtuels. En outre, l'analyse décrite par les Equations (1) et (2) serait insuffisante. Par exemple, nous obtiendrions `virt_ch_a` = `virt_ch_a` = {58,91}, ce qui ne permet pas de vérifier qu'aucune confusion n'est faite entre les valeurs envoyées sur les deux circuits.

4. Conclusion

Dans cet article nous avons vu comment une analyse de flot de contrôle peut être utilisée afin de vérifier des propriétés de sûreté pour des systèmes distribués. La précision de l'analyse dépend sensiblement de l'approximation réalisée aux points de réceptions.

Bodei et al. ont montré comment utiliser une telle analyse afin de vérifier des propriétés liées à la sécurité des applications distribuées [2]. Cependant, leur analyse demeure imprécise dans la gestion des valeurs transmises lors des communications.

Pour notre part, outre le mécanisme d'allocation de circuits virtuels, nous avons utilisé l'analyse décrite à la Section 2.2 afin de vérifier le bon fonctionnement d'autres applications. Concernant la sûreté, nous avons traîté le cas d'une implémentation d'un multiplexeur/démultiplexeur. Des valeurs sont envoyés sur différents canaux d'entrée (virtuels) du multiplexeur qui les retransmet sur un seul canal (physique). Le démultiplexeur reçoit l'ensemble des valeurs sur ce même canal et les transmet sur différents canaux (virtuels). Nous montrons que, dans notre implémentation, les valeurs envoyées sur les différents canaux à la sortie du démultiplexeur correspondent à celles envoyées sur les canaux correspondant à l'entrée du multiplexeur.

Concernant la sécurité, nous avons vérifié le bon fonctionnement d'une application distribuée de vente aux enchères. Ce système est composé d'un serveur et de plusieurs clients qui transmettent des informations publiques (enchères) et privées (nom, numéro de carte bleue) au serveur. Chaque client est relié au serveur par un seul canal de communication. Le serveur diffuse les données publiques aux autres clients et conserve les informations privées. Nous montrons que, dans notre implémentation, seules les données publiques sont effectivement diffusées.

Tant pour ces deux derniers exemples que pour celui de la Section 3, ces résultats ne peuvent pas être obtenus par les analyses précédemment développées qui se fondent sur les Equations (1) et (2).

Bibliographie

1. Antony Alles. ATM Internetworking. Technical report, 1995. CISCO Systems Inc.
2. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *Concur'98*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 1998.
3. R. De Nicola, G. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects, Lectures Notes in Computer Science, Springer*, volume 1603, 1999.
4. Marc Gengler and Matthieu Martel. Self-applicable partial evaluation for the pi-calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 36–46, 1997.
5. Matthieu Martel and Marc Gengler. Static analysis of the communication topology of concurrent programs. Technical report, RR 354, Laboratoire d'Informatique de Marseille, 2000.
6. Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'97*, pages 332–345. ACM, 1997.
7. Flemming Nielson and Hanne Riis Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency*, Monograph in Computer Science, pages 185–251. Springer, 1999.
8. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.
9. John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
10. Olin Shivers. *Control Flow Analysis of Higher Order Languages.* PhD thesis, Carnegie Mellon University, Scholl of Computer Science, 1991. Technical Report CMU-CS-91-145.
11. Andrew S. Tanenbaum. *Computer Networks, Third Edition.* Prentice Hall, 1996.