Spécification en pi-calcul de l'étude de cas relative au contrôle d'accès

Noël Bernard, Yves Dumond

Spécification en π-calcul de l'étude de cas relative au contrôle d'accès

Noël Bernard †, Yves Dumond ‡
†LAMA, ‡LGIS-LLP/CESALP
Université de Savoie, Campus Scientifique
73376 Le Bourget-du-Lac cedex
e-mail: {Noel.Bernard, Yves.Dumond}@univ-savoie.fr

Résumé

Nous traitons dans cet article l'étude de cas relative au contrôle d'accès. Le formalisme choisi pour la spécification est le π-calcul et la μ-logique sert de fondement à l'expression des propriétés souhaitées pour le système. Après avoir brièvement introduit ces formalismes, nous décrivons la spécification de la partie "*Contrôleur*" de l'étude de cas. Celle-ci s'articule autour d'un processus traitant les requêtes provenant du procédé et d'une image temps-réel représentant ce dernier. Les propriétés souhaitées pour le *Contrôleur* sont exprimées en μ-logique, puis nous tentons de les vérifier à l'aide du Mobility Workbench. Nous évaluons ensuite l'adéquation des techniques formelles utilisées vis-à-vis des tâches qu'il était initialement prévu de réaliser et nous envisageons les possibilités d'extension de la spécification proposée.

Mots-clés : Spécifications formelles, vérification de logiciel, model-checking, π-calcul, μ-logique.

1 Introduction

Nous nous intéressons dans cet article à la description dans une algèbre de processus de la partie *Contrôleur* de l'étude de cas relative au système de contrôle d'accès qui a été introduite par Yves Ledru [Ledru 98a]. De fait, ce type d'outils formels destinés à la description et à l'analyse de systèmes concurrents est bien adapté au traitement d'applications à forte composante interactionnelle. Nous prenons donc le parti d'envisager sous cet angle l'étude de cas proposée. Nous décrivons donc en section 2 le π-calcul et la μ-logique qui lui est associée. En section 3, le *Contrôleur* est spécifié en π-calcul. La vérification formelle de cette spécification, sous forme d'un ensemble de formules exprimées en μ-logique, est abordée à la section 4. Une tentative d'expérimentation dans le cadre d'un outil, le Mobility Workbench, fait l'objet de la section 5. Nous évaluons la démarche proposée et évoquons les possibilités d'extension offertes par la spécification en section 6. Nous concluons en section 7 sur le bilan de cette étude de cas.

2 Présentation du π-calcul et de la μ-logique

2.1 Le π-calcul

Les algèbres de processus constituent des cadres adéquats pour la spécification et la vérification de systèmes réactifs. Cette école a vu le jour en 1980 avec le modèle CCS [Milner 80] dont le π-calcul [Milner 92] est un descendant direct. Dans ces modèles, tout terme bien formé dénote un processus. L'abstraction fondamentale est qu'on ne s'intéresse au comportement d'un processus qu'à travers un certain nombre de points d'interaction également appelés "canaux". La synchronisation et la communication sont alors exprimées par des lois de composition internes et externes. Il faut noter que tant en CCS qu'en π-calcul, on ne fait aucune hypothèse sur les

vitesses d'exécution relatives des différents processus, qui sont donc présumés progresser chacun à leur rythme.
Nous utilisons dans cet article le π-calcul polyadique synchrone du premier ordre [Milner 93]. La syntaxe retenue est donc la suivante :

$$P ::= \mathbf{0} \mid \rho.P \mid P \mid P \mid P + P \mid [x = y]P \mid [x \neq y]P \mid (\nu\, \vec{u})P \mid A(\vec{u})$$

où les préfixes ρ ou "actions" sont définis de la manière suivante :

$$\rho ::= \tau \mid \alpha(\vec{u}) \mid \bar{\alpha}\langle\vec{u}\rangle$$

et où l'abstraction de processus doit faire l'objet de déclarations de la forme :

$$A(\vec{u}) \stackrel{def}{=} P$$

$\vec{u}$ dénotant une liste, éventuellement vide, de variables également appelées "noms". Nous donnons à la suite une sémantique informelle du π-calcul :

- Le processus "**0**" ne fait rien, c'est-à-dire qu'il n'interagit pas avec son environnement. Il dénote la terminaison normale d'un processus mais aussi des cas plus pathologiques comme celui d'un processus dont l'évolution interne est bloquée. Enfin, "**0**" étant la seule constante de base de l'algèbre, elle permet la construction, de façon inductive, des autres processus.

- Le processus $\rho.P$ fait intervenir l'opérateur de préfixage : il s'agit d'une loi de composition externe impliquant une action "ρ" et un processus "P". Cet opérateur est le seul qui exprime la séquentialité en π-calcul. Ainsi, $\rho.P$ exécute l'action ρ puis se comporte comme P. Par ailleurs, ρ peut être soit une action interne inobservable, alors notée τ, soit une interaction avec l'environnement : $\alpha(\vec{a})$ ou $\bar{\alpha}\langle\vec{a}\rangle$ qui expriment respectivement la réception et l'émission d'une liste de variables sur le canal α. D'un point de vue strictement temporel, une interaction est semblable à un mécanisme de type rendez-vous tel qu'il a été défini en CSP [Hoare 85] : un processus qui entend exécuter une action ne peut le faire que s'il existe au même moment une contrepartie dans l'environnement. Nous entendons par là une offre d'interaction duale sur le même canal, c'est-à-dire une émission dans le cas d'une réception et réciproquement. Notons que les offres de réception sont représentées par des noms (α, β, γ, ...) et que les offres d'émission font intervenir leurs conjugés ($\bar{\alpha}$, $\bar{\beta}$, $\bar{\gamma}$, ...). On dispose de plus de la propriété suivante:

$$\bar{\bar{\alpha}} = \alpha$$

- $P \mid Q$ dénote un couple de processus qui s'exécutent en parallèle. Chacun d'entre-eux a ainsi la possibilité d'interagir avec l'environnement, mais ils peuvent également interagir l'un avec l'autre. En pareil cas, une synchronisation se produit et si la liste associée n'est pas vide, il y a également une communication unidirectionnelle. Ainsi, dans le cas où :

$$\bar{\alpha}\langle\vec{a_i}\rangle.P \mid \alpha(\vec{b_i}).Q \xrightarrow{\tau} P \mid Q\{\vec{a_i}/\vec{b_i}\}$$

  on substitue à chaque occurrence libre du marqueur de place b_i dans le processus récepteur la variable correspondante a_i. La transition est étiquetée par τ dans la mesure où on considère qu'il s'agit d'une opération interne à l'ensemble des deux processus. Une différence essentielle entre CCS et le π-calcul est que dans le cas du second, les noms échangés peuvent être eux-mêmes des canaux. Ainsi, le terme :

$$\bar{\alpha}\langle\beta\rangle.P \mid \alpha(a).\bar{a}\langle x\rangle.Q$$

  se réduit-il, en l'absence d'autres processus dans l'environnement, à :

$$P \mid \bar{\beta}\langle x\rangle.Q\{\beta/a\}$$

  Cette propriété est appelée *mobilité* : son objectif est d'autoriser la reconfiguration dynamique de la topologie des systèmes.

- Le processus $P + Q$ dénote le choix indéterministe entre les processus P et Q. Ces derniers ne peuvent être exécutés tous les deux. Ils leur est notamment impossible d'interagir l'un avec l'autre. Le choix de celui qui est exécuté est effectué en fonction des offres d'interaction présentes dans l'environnement. En l'absence d'opérateur de matching (voir plus bas), il s'agit donc d'un choix externe, c'est-à-dire effectué par l'environnement. Par exemple, en l'absence d'autres processus dans l'environnement et si α, β et γ sont tous différents, le processus :

  $$\alpha(a_1, a_2).P \ + \ \bar{\beta}\langle b\rangle.Q \ \mid \ \beta(c).R \ + \ \gamma(d).S$$

  ne peut évoluer que vers :

  $$Q \ \mid \ R\{b/c\}$$

  Si plusieurs possibilités d'interaction sont simultanément éligibles, le choix est effectué de façon indéterministe.

- L'opérateur de matching permet de réaliser l'exécution conditionnelle d'un processus. Ainsi, si $a = b$, $[a = b]P$ se comporte comme P, sinon il est bloqué, c'est-à-dire qu'il se comporte comme $\mathbf{0}$. L'opérateur de mismatching a une sémantique symétrique : si $a \neq b$, $[a \neq b]P$ se comporte comme P, sinon il est bloqué. La combinaison de ces opérateurs avec le choix indéterministe permet de réaliser des choix internes équivalents à des énoncés conditionnels.

- $(\nu\ \vec{a})P$ dénote un processus dans lequel les variables de la liste $\vec{a}$ ont une portée limitée à P, c'est-à-dire qu'elles sont inconnues dans les autres processus : on parle alors de variables *privées*. Une des conséquences de cette situation est que P ne peut communiquer avec l'environnement en utilisant ces canaux. Si l'un d'entre-eux est émis sur un autre canal, sa portée est étendue au processus récepteur : ce phénomène est désigné sous le terme de "*scope extrusion*". Les éventuels conflits d'homonymie entre des variables privées exportées et des variables locales sont résolus par le biais d'alpha-conversions. Par exemple :

  $$(\nu\ a)(\bar{\beta}\langle a\rangle.P) \ \mid \ \beta(x).\bar{a}\langle x\rangle.Q$$

  requiert une alpha-conversion avant l'exportation de a de façon à préserver la différence entre les variables a appartenant à chacun des processus définis de part et d'autre de l'opérateur "|". On effectue donc l'alpha-conversion (a' est obligatoirement un nom libre) préalablement à l'émission :

  $$(\nu\ a')(\bar{\beta}\langle a'\rangle.P\{a'/a\}) \ \mid \ \beta(x).\bar{a}\langle x\rangle.Q$$

  puis la réduction du terme avec extension de la portée de a' :

  $$(\nu\ a')(P\{a'/a\} \ \mid \ \bar{a}\langle a'\rangle.Q\{a'/x\})$$

  Une application de ce principe réside dans l'écriture d'un processus générateur de noms libres qui est d'ailleurs utilisé dans l'étude de cas :

  $$Gensym(gs) \stackrel{def}{=} (\nu\ freshname)(\overline{gs}\langle freshname\rangle.Gensym(gs))$$

  Gensym génère ainsi une infinité de noms libres que l'on peut acquérir sur le canal gs.

- La définition d'une abstraction de processus :

  $$A(\vec{u}) \stackrel{def}{=} P$$

  lie un identifiant de processus A et une liste de variables $\vec{u}$ à un comportement de processus P. Par substitution de paramètres effectifs à ces paramètres formels, on crée un processus instance de P. Ce mécanisme de définition d'abstractions autorise de plus la récursivité et permet donc de créer des comportements de processus infinis. Il est également le seul outil dont on dispose en π-calcul pour constituer des structures de données.

2.2 La μ-logique

Tout comme CCS avec la Hennessy-Milner Logic, le π-calcul dispose d'une logique associée : la μ-logique. On peut ainsi soumettre les processus à une vérification sous forme de model-checking. La μ-logique permet ainsi d'exprimer qu'une propriété donnée est vérifiée par un processus donné. La syntaxe utilisée ici est la suivante :

$$\varphi ::= tt \mid ff \mid a = b \mid \neg\,\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \forall x\,\varphi \mid \exists x\,\varphi \mid \Sigma x\,\varphi \mid \nu X\varphi \mid \mu X\varphi$$

Le fait que φ est vérifiée par P est noté :

$$P \models \varphi$$

A chaque action α pouvant être effectuée par un processus spécifié en π-calcul (une émission, une réception, ou encore τ) sont associées deux modalités $[\alpha]$ et $\langle\alpha\rangle$ dont nous donnons à la suite la sémantique. Soit P un processus et φ une propriété, alors :

- $P \models [\alpha]\,\varphi$ signifie que pour toute action $P \xrightarrow{\alpha} P'$ on a : $P' \models \varphi$. C'est donc une modalité apparentée à la *nécessité*.
- $P \models \langle\alpha\rangle\,\varphi$ signifie qu'il existe au moins une action $P \xrightarrow{\alpha} P'$ pour laquelle : $P' \models \varphi$. C'est donc une modalité apparentée à la *possibilité*.

Ainsi, à l'aide de la formule toujours vraie "*tt*" et de la formule toujours fausse "*ff*" on peut exprimer le fait que le processus P peut exécuter l'action α par : $P \models \langle\alpha\rangle\,tt$ ou le fait que P ne peut pas exécuter α par : $P \models [\alpha]\,ff$.
Dans le cas du π-calcul, la situation est rendue plus complexe par le fait que lors d'une émission ou d'une réception des messages sont véhiculés. Ceci amène à introduire des quantificateurs : $\forall x$, $\exists x$ et Σx. Ainsi, un processus de la forme $a(x).P$ vérifiera une propriété :

$$[a]\,\forall x\,\varphi$$

si quel que soit le nom n reçu par ce processus, on a :

$$P\{n/x\} \models \varphi\{n/x\}$$

Comme en logique usuelle, le quantificateur $\forall$ possède un dual logique $\exists$ lié à l'existence de formules négatives ($\exists x\,\varphi \Leftrightarrow \neg\forall x\,\neg\varphi$). Mais il existe de plus un dual à caractère plus informatique, lié à l'émission. Ainsi, un processus $\bar{a}\langle n\rangle.P$ satisfait :

$$[\bar{a}]\,\Sigma x\,\varphi$$

si :

$$P \models \varphi\{n/x\}$$

Enfin, le μ-calcul intègre, et c'est là l'origine de son nom, deux opérateurs de plus petit et de plus grand point fixe sur les formules, respectivement notés : $\mu\,X\,\varphi$ et $\nu\,X\,\varphi$. Ceux-ci permettent de décrire des propriétés globales vérifiées par les processus infinis. Ainsi, le fait qu'un processus P soit capable d'effectuer indéfiniment une action a sera traduit par la formule φ suivante :

$$\varphi \;=\; (\nu\,X)\langle a\rangle X$$

Le fait que le conjugué de cette même action soit susceptible d'être accomplie dans le futur s'écrira quant à lui :

$$\varphi \;=\; (\mu\,X)(\langle\bar{a}\rangle\,tt \;\vee\; \langle -\rangle\,X)$$

où "-" désigne une action quelconque.

3 Spécification en π-calcul

3.1 Architecture globale du système : le *Contrôleur*

Nous réalisons tout d'abord une abstraction sur une partie du système en considérant le *Bus* comme un processus π-calcul, non explicitement décrit et que nous supposerons fiable, encapsulant ce que nous appellerons le "procédé", c'est-à-dire les utilisateurs porteurs de cartes, les bâtiments avec leurs portes, leurs lecteurs de cartes et leurs capteurs. Le *Bus* réagit donc comme un processus aux différentes offres d'interaction, tant en émission : il s'agit alors d'informations provenant des capteurs et transmises au *Contrôleur*, qu'en réception : il s'agit alors d'ordres destinés aux actionneurs. Le *Contrôleur* proprement dit est constitué d'une douzaine de processus et de leurs instances respectives. Ces processus sont répartis en deux catégories dont chacune correspond à une fonctionnalité bien déterminée :

- La fonction de contrôle-commande proprement dite. Ceci concerne les processus *Interface*, *Site*, *Horloge*, *Archives*, ainsi que *Cycle*, *Refus*, et *Autorisation* avec leurs instances respectives.
- La modélisation du procédé, à travers la constitution d'une image intégrant des données à caractère "temps-réel". Ceci concerne les processus *Utilisateur*, *Batiment*, *Porte*, *Lecteur de carte*, *Capteur* et leurs instances respectives, également appelés dans la suite "processus images". Il est à noter que nous avons choisi de faire transiter les ordres destinés aux actionneurs par les processus représentant les entités concernées. Ceci autorise un éventuel filtrage en fonction de l'état du procédé, par exemple en cas d'incendie.

La structure du *Contrôleur* est résumée dans le schéma ci-dessous qui spécifie en outre les flots de communications entre le *Bus* et les processus concernés :

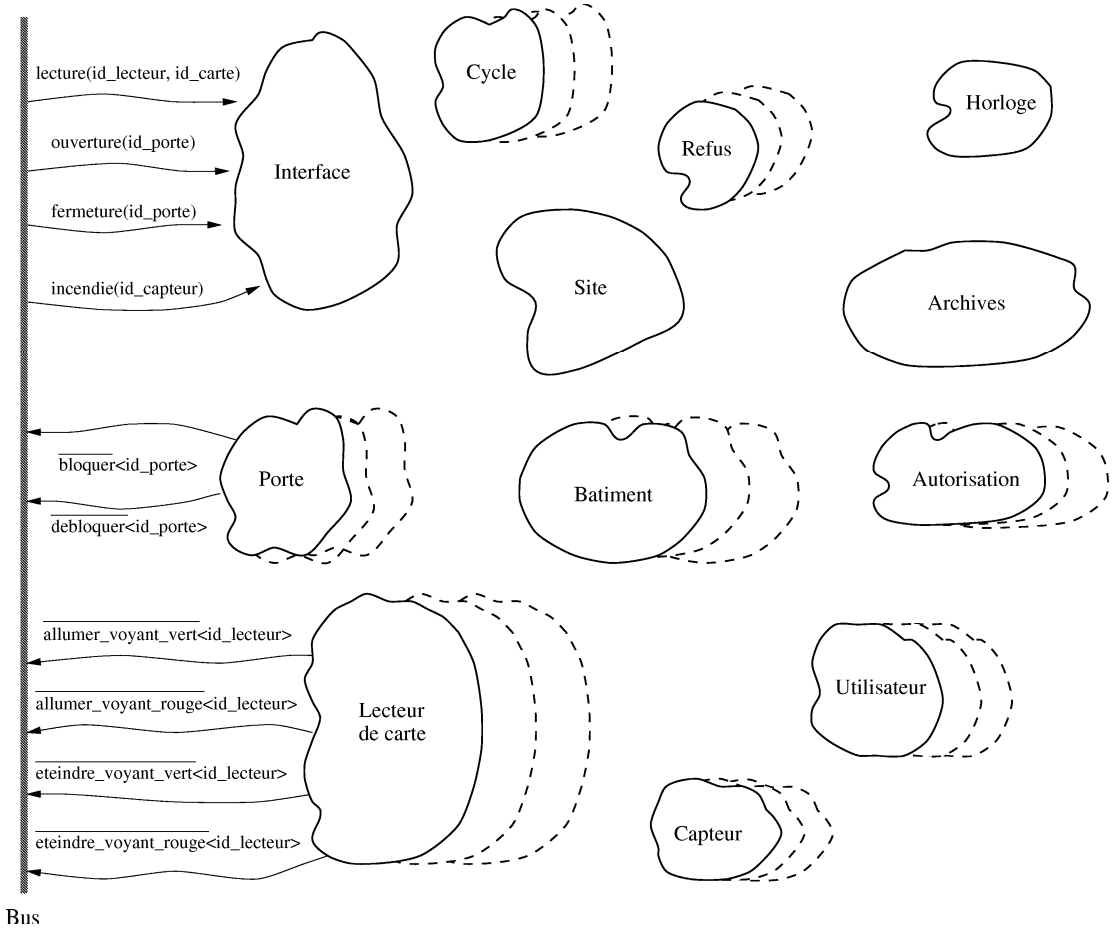

Figure: architecture globale du Contrôleur

3.2 Le processus Interface

Le processus *Interface* constitue le pivot de la partie contrôle-commande du *Contrôleur*. Il n'en existe qu'une seule instance qui interagit directement avec le *Bus* et reçoit à ce titre les données en provenance du procédé. Ces messages sont ceux décrits dans [Ledru 98] :

- `lecture(id_lecteur, id_carte)`
- `ouverture(id_porte)`
- `fermeture(id_porte)`
- `incendie(id_capteur)`

La structure globale du processus *Interface* est donc exprimée sous la forme d'un choix indéterministe entre ces différentes alternatives :

- Dans le cas d'une `lecture`, *Interface* interroge *Site* sur la légalité du franchissement de la porte. Il recueille la réponse ainsi que d'autres informations telles que les identifiants de la porte et du bâtiment concernés, le nom du propriétaire de la carte et le sens de la porte, c'est-à-dire s'il s'agit d'une tentative d'entrée ou de sortie. Un port privé préalablement créé et exporté sert à la transaction. Si le bâtiment concerné est dans un état d'incendie, *Interface* ne donne pas suite, la porte étant débloquée ou en passe de l'être. Si une fraude est détectée (carte perdue ou volée, utilisateur appartenant à un groupe non autorisé, ...), un processus gérant le refus d'accès est activé (allumage du voyant rouge pendant une durée de deux secondes puis extinction). Enfin, dans le cas d'une tentative de franchissement licite, on crée un processus spécifique, sous forme d'une instance de *Cycle*. Celle-ci est identifiée par un nom libre unique dans le système : l'*estampille*. Elle est par ailleurs chargée de gérer le cycle de franchissement d'une porte. L'*estampille* est également communiquée au processus modélisant la porte concernée. Ainsi, on dispose d'un lien entre ce dernier et l'instance de *Cycle* correspondante, ce qui permet de donner ultérieurement les suites appropriées aux messages d'ouverture et de fermeture de la porte.

- Les messages d'`ouverture` et de `fermeture` d'une porte donnent lieu à des traitements assez semblables. Il s'agit dans les deux cas de s'assurer que le bâtiment n'est pas en état d'incendie, faute de quoi aucun traitement n'est entrepris, pour les mêmes raisons que dans le cas du message précédent. Si la situation est normale, des messages sont émis en direction du processus image de la porte, pour signaler respectivement son ouverture et sa fermeture. Ces messages sont ensuite redirigés vers l'instance correspondante de *Cycle*.

- Le message d'alerte `incendie` fait l'objet d'un traitement spécifique : il faut prioritairement s'assurer du fait que le processus représentant le bâtiment prend en compte cette situation. L'appel récursif d'*Interface* n'est donc effectué qu'après qu'un message de synchronisation provenant du processus bâtiment en ait rendu compte. Ceci revient à suspendre les autres activités d'*Interface* c'est-à-dire l'activation de nouveaux cycles ou la transmission de messages d'ouverture et de fermeture de portes et donc à conférer un certain niveau de priorité au traitement de l'alerte incendie.

Au-delà de la description purement fonctionnelle d'*Interface*, il est intéressant de noter les points suivants :

- L'indéterminisme quant à la nature des messages reçus en entrée est exprimé de manière très naturelle grâce à l'opérateur de choix indéterministe.

- Les variables constituant le contexte d'appel du processus ne sont jamais modifiées : on peut donc supposer que les instances successives d'*Interface* sont toutes strictement identiques.

- Les traitements sont accomplis, chaque fois que c'est possible, en parallèle avec l'appel récursif du processus *Interface* : ceci vise à donner à ce dernier un niveau de vivacité aussi élevé que possible.

On donne à la suite la spécification intégrale du processus $Interface$ (les appels récursifs ont été abrégés) :

$$
\begin{array}{l}
\textbf{Interface}(\textbf{site}, \textbf{gs}, \textbf{timer}, \textbf{horl}, \textbf{archiv}, \textbf{acces}, \\
\qquad \textbf{lecture}, \textbf{ouverture}, \textbf{fermeture}, \textbf{incendie}) \stackrel{\textbf{def}}{=} \\
\\
(\quad (lecture(id\_lecteur, id\_carte). \\
\quad gs(reps).\overline{site}\langle id\_lecteur, id\_carte, reps\rangle.reps(id\_porte, nom, bat, sens, cr). \\
\quad (\quad [cr = incendie]\ Interface(...) \\
\quad + \\
\qquad [cr = fraude]\ (\ Interface(...) \\
\qquad\qquad\qquad | \\
\qquad\qquad\qquad Refus(gs, timer, id\_lecteur)\ ) \\
\quad + \\
\qquad [cr = ok]\ (\ Interface(...) \\
\qquad\qquad\qquad | \\
\qquad\qquad\qquad gs(estampille). \\
\qquad\qquad\qquad \overline{id\_porte}\langle creation\_cycle, estampille\rangle. \\
\qquad\qquad\qquad Cycle(estampille, gs, timer, horl, archive, acces, \\
\qquad\qquad\qquad\qquad id\_lecteur, id\_carte, id\_porte, nom, bat, sens)\ )\quad)) \\
+ \\
\quad (ouverture(id\_porte). \\
\quad gs(rep).\overline{id\_porte}\langle etat\_incendie, rep\rangle.rep(cr). \\
\quad (\quad [cr = incendie]\ Interface(...) \\
\quad + \\
\qquad [cr = pas\_incendie]\ (\ Interface(...) \\
\qquad\qquad\qquad\qquad | \\
\qquad\qquad\qquad\qquad \overline{id\_porte}\langle ouverture, nil\rangle.\mathbf{0}\ )\quad)) \\
+ \\
\quad (fermeture(id\_porte). \\
\quad gs(rep).\overline{id\_porte}\langle etat\_incendie, rep\rangle.rep(cr). \\
\quad (\quad [cr = incendie]\ Interface(...) \\
\quad + \\
\qquad [cr = pas\_incendie]\ (\ Interface(...) \\
\qquad\qquad\qquad\qquad | \\
\qquad\qquad\qquad\qquad \overline{id\_porte}\langle fermeture, nil\rangle.\mathbf{0}\ )\quad)) \\
+ \\
\quad (incendie(id\_capteur). \\
\quad gs(rep).\overline{id\_capteur}\langle batiment, rep\rangle.rep(id\_batiment). \\
\quad gs(rep).\overline{id\_batiment}\langle alerte\_incendie, rep, nil, nil\rangle.rep. \\
\quad Interface(...))\quad)
\end{array}
$$

3.3 Le processus Cycle

Le processus $Cycle$ a pour objectif de gérer les cycles de franchissement de portes. Il est créé une instance de $Cycle$ par franchissement. Celle-ci disparaît lorsque le cycle correspondant s'achève. Ce processus se distingue également par le fait que des portions de spécifications ont été ajoutées qui ne relèvent pas de l'aspect strictement opérationnel du processus, mais qui sont en fait destinées à faciliter les preuves.

L'exécution d'une instance de $Cycle$ débute par l'armement d'un timer : par interaction avec l'horloge, on spécifie le fait qu'à échéance d'un délai de trente secondes une offre d'interaction en émission sera effectuée sur le canal δt. Le voyant vert du lecteur concerné est ensuite allumé

et la porte débloquée. Plusieurs scénarii sont alors possibles :

- Le délai arrive à échéance sans que la porte ait été ouverte : celle-ci est alors rebloquée et le voyant vert éteint.
- Avant échéance du délai, la porte est ouverte. Elle est ensuite refermée comme en témoignent les synchronisations effectuées sur le canal *estp*. La localisation (identifiant de bâtiment ou valeur "extérieur") de la carte correspondante est alors mise à jour (ce qui permet de situer son propriétaire), le franchissement de la porte archivé, la porte bloquée et le voyant vert éteint.
- Il est également possible qu'entre les deux synchronisations sur *estp*, une alarme incendie survienne, et que le processus image du bâtiment enregistre celle-ci : les processus images des portes sont alors écrits pour ne plus accepter en pareil cas qu'un nombre limité de requêtes (en priorité les ordres de déblocage), ce qui exclut notamment les synchronisations sur les canaux *estampille* rendant compte des ouvertures et fermetures de portes aux processus instances de $Cycle$. Si cela survient, le processus $Cycle$ en cours ne sera pas en mesure d'aller jusqu'à son terme, mais cela restera sans conséquence pour la gestion de la porte en question et a fortiori pour le reste du système. Il est à noter que ce cas de figure pourrait faire l'objet d'un traitement explicite par armement d'un timer dont l'échéance ferait l'objet d'un choix indéterministe avec la seconde occurrence de *estp* (de même que la première émission sur le canal *acces*). Cette seconde occurrence ne survenant pas, à échéance du timer, un traitement approprié pourrait alors être entrepris.
- En toute rigueur, un autre cas de non-terminaison d'une instance de processus $Cycle$ viendrait à survenir si un utilisateur ouvrait la porte sans jamais la refermer, mais cela est exclu par l'hypothèse des utilisateurs disciplinés.

Il faut noter que les deux émissions sur le canal *acces* ont été ajoutées pour faciliter les preuves. On trouvera à la suite la spécification du processus $Cycle$:

$$
\begin{array}{l}
\mathbf{Cycle}(\mathbf{estp}, \mathbf{gs}, \mathbf{timer}, \mathbf{horl}, \mathbf{archiv}, \mathbf{acces}, \\
\qquad \mathbf{id\_lecteur}, \mathbf{id\_carte}, \mathbf{id\_porte}, \mathbf{nom}, \mathbf{bat}, \mathbf{sens}) \stackrel{\mathbf{def}}{=} \\
\quad gs(\delta t).\overline{timer}\langle \delta t, 30\rangle. \\
\quad \overline{id\_lecteur}\langle allumer\_vvert, nil\rangle.\overline{id\_porte}\langle debloc, nil\rangle. \\
\quad (\quad (\delta t. \\
\qquad \overline{id\_porte}\langle bloc, nil\rangle. \\
\qquad \overline{id\_lecteur}\langle eteindre\_vvert, nil\rangle.\mathbf{0}) \\
\quad + \\
\qquad (estp. \\
\qquad (\ \overline{acces}\langle id\_carte, id\_lecteur, engage\rangle.\mathbf{0} \\
\qquad \ + \\
\qquad \ \ estp. \\
\qquad \ \ gs(h).\overline{horl}\langle h\rangle.h(heure, date). \\
\qquad \ \ \overline{id\_carte}\langle mise\_a\_jour\_localisation, sens, bat\rangle. \\
\qquad \ \ \overline{archiv}\langle heure, date, nom, bat, sens\rangle. \\
\qquad \ \ \overline{id\_porte}\langle bloc, nil\rangle. \\
\qquad \ \ \overline{id\_lecteur}\langle eteindre\_vvert, nil\rangle.\mathbf{0}) \ ) \\
\quad + \\
\qquad \overline{acces}\langle id\_carte, id\_lecteur, en\_attente\rangle.\mathbf{0} \quad) \\
\end{array}
$$

3.4 Rôle des autres processus de la partie contrôle-commande

Le processus $Site$ sert d'intermédiaire entre $Interface$ et le reste du système. Il sous-traite le traitement des autorisations d'accès au bâtiment concerné qui fait à son tour appel à $Autorisation$.

Le rôle du processus *Archives* a été limité ici au stockage des enregistrements horodatés des entrées et des sorties dans les différents bâtiments.

3.5 L'image du procédé

En π-calcul, la notion de processus est la seule abstraction permettant d'envisager la définition de structures de données. Celles-ci sont donc définies, le cas échéant, par le biais de codages comme en λ-calcul. Une telle démarche s'avère en fait fastidieuse. Nous souhaitions limiter autant que possible le recours à cette technique tout en respectant quelques principes généraux :

- Adopter une modélisation favorisant l'autonomie des différents composants et limitée, dans un but de concision, aux besoins ressentis.
- Pas de duplication d'informations variant dans le temps : état d'incendie d'un bâtiment, localisation d'un utilisateur, etc ... de façon à éviter les situations d'incohérence dans l'état du système.

Nous avons donc choisi un mode de représentation qui évoque clairement une méthodologie "par objets". Chaque processus image représentant une personne ou un objet physique est donc l'instance d'un processus caractérisé par un contexte d'appel donné. Celui-ci constitue l'état du processus, en quelque sorte ses "variables d'instances". Par exemple, partant de l'hypothèse que chaque utilisateur disposait d'une seule carte valide à un instant donné, nous avons défini un processus *Utilisateur* caractérisé par un identifiant de carte, le nom du propriétaire de la carte, la liste des groupes d'utilisateurs auxquels appartient le propriétaire, sa localisation (l'identifiant d'un bâtiment ou "exterieur" s'il ne se trouve dans aucun bâtiment), et la validité de la carte :

$$Utilisateur(id\_carte, id\_nom, l\_groupes, localisation, validite)$$

Pour chaque instance des différentes classes de processus image, il existe dans le contexte d'appel un attribut discriminant (dans le cas de l'exemple que nous venons de donner, il s'agit de l'identifiant de carte). Cet attribut sert donc logiquement de canal de communication avec l'environnement. Dans le cadre que nous sommes donné, ces communications correspondent à des requêtes, des appels de méthodes en somme, visant à obtenir du processus qu'il accomplisse certaines actions, qu'il fournisse des informations sur son état interne, ou encore qu'il modifie ce dernier. Le comportement des processus images des entités du procédé est donc purement réactif. De plus, une adresse de continuation, sous forme d'un identifiant de canal, est fournie lors des requêtes et permet de récupérer le résultat des traitements. Le choix de cette représentation nous a par ailleurs contraints à "normaliser" les requêtes destinées à une même classe de processus en imposant un nombre fixe de paramètres. Ceci a conduit à ce que dans la spécification, certains messages contiennent des arguments non significatifs.
Chaque interaction avec un processus se termine par un appel récursif avec un contexte d'appel éventuellement modifié, ce qui permet de rendre compte des évolutions de l'état interne du processus. Enfin, nous avons décidé de particulariser le comportement de certains processus en cas d'incendie : concrètement, certaines requêtes cessent alors d'être traitées. L'exemple le plus typique est celui des processus instances de *Porte* qui n'admettent plus les ordres de blocage de portes dès lors qu'ils appartiennent à un bâtiment en état d'incendie (voir page suivante la description partielle du processus *Porte*).
Le mode de représentation choisi ici nous a permis de représenter tous les types de données nécessaires à l'étude de cas à l'exception des listes d'utilisateurs. Nous avons donc dû nous résoudre à adopter un codage pour les listes directement inspiré de [Milner 93]. Celui-ci met en œuvre un ensemble de cellules sous formes de processus délivrant sur un canal donné l'élément contenu dans la cellule et le canal permettant d'interagir avec la cellule suivante. Le fait qu'un utilisateur soit autorisé à entrer dans un bâtiment se ramène alors à ce que l'intersection entre la liste des groupes d'utilisateurs auxquels appartient l'utilisateur postulant et la liste des groupes d'utilisateurs autorisés à pénétrer dans le bâtiment soit non vide. Le calcul de cette intersection est effectué par un processus idoine que l'on sollicite sur le canal *interlistes*.

On trouvera à la suite une spécification partielle du processus *Porte* :

$$\mathbf{Porte(id\_porte, id\_batiment, sens, estp\_cycle\_en\_cours, etat\_blocage)} \stackrel{\mathbf{def}}{=}$$

$$
\begin{array}{l}
\quad id\_porte(requete, param). \\
\quad gs(rep).\overline{id\_batiment}\langle etat\_incendie, rep, nil, nil\rangle.rep(cr). \\
(\quad [cr = incendie] \\
\qquad (\ [requete = bloc] \\
\qquad\quad Porte(id\_porte, id\_batiment, sens, estp\_cycle\_en\_cours, debloquee) \\
\qquad + \\
\qquad\quad \ldots \\
\qquad\quad -- \ traitement\ des\ autres\ requêtes\ en\ cas\ d'incendie \\
\qquad\quad \ldots \\
\qquad + \\
\qquad\quad [requete = etat\_blocage\_porte] \\
\qquad\quad (\ Porte(id\_porte, id\_batiment, sens, estp\_cycle\_en\_cours, etat\_blocage) \\
\qquad\quad\ | \\
\qquad\quad\ \overline{param}\langle etat\_blocage\rangle.\mathbf{0})\ \ ) \\
+ \\
\quad [cr = pas\_incendie] \\
\qquad (\ [requete = bloc] \\
\qquad\quad \overline{bloquer}\langle id\_porte\rangle. \\
\qquad\quad Porte(id\_porte, id\_batiment, sens, estp\_cycle\_en\_cours, bloquee) \\
\qquad + \\
\qquad\quad [requete = creation\_cycle] \\
\qquad\quad Porte(id\_porte, id\_batiment, sens, param, etat\_blocage) \\
\qquad + \\
\qquad\quad \ldots \\
\qquad\quad -- \ traitement\ des\ autres\ requêtes\ hors\ cas\ d'incendie \\
\qquad\quad \ldots \\
\qquad + \\
\qquad\quad [requete = etat\_blocage\_porte] \\
\qquad\quad (\ Porte(id\_porte, id\_batiment, sens, estp\_cycle\_en\_cours, etat\_blocage) \\
\qquad\quad\ | \\
\qquad\quad\ \overline{param}\langle etat\_blocage\rangle.\mathbf{0})\ \ )\quad\ )
\end{array}
$$

De même que les processus *Porte*, les processus *Utilisateur* répondent à des requêtes telles que *presence* qui indique si l'utilisateur est ou non présent dans un bâtiment donné, *situation* qui donne la localisation de l'utilisateur ou *groupe* qui donne la liste des groupes d'appartenance de l'utilisateur. Les processus lecteurs de cartes *Lecteur* peuvent indiquer le *batiment* et la *porte* auxquels ils appartiennent. De même, les processus *Batiment* sont susceptibles d'indiquer les groupes d'utilisateurs autorisés en accès et le fait qu'ils sont ou non en état d'incendie. Ces quelques indications sont loin de constituer une description complète des processus images du procédé, mais nous avons voulu faire état de l'existence de ces quelques requêtes dans la mesure où elles sont utilisées dans la description des propriétés logiques du système.

3.6 Le système complet

Le système complet est constitué de la mise en parallèle des processus assurant les fonctions de contrôle-commande et de ceux représentant l'image du procédé avec dans les deux cas leurs instances respectives. On applique de plus sur cet ensemble une restriction qui ne laisse comme variables libres que les noms connus à l'extérieur, tels que par exemple les identifiants d'utilisateurs, de bâtiments ou de portes, ainsi que les dix canaux servant aux interactions avec le bus (*lecture, ouverture, ..., eteindre_voyant_rouge*).

4 Expression des propriétés du *Contrôleur* en μ-logique

Les propriétés attendues pour le *Contrôleur* sont exprimées dans [Ledru 98b] par des formules d'une logique temporelle, à partir d'un ensemble de formules de base décrivant l'état instantané des composants et la situation d'un utilisateur. La logique utilisée possède cinq opérateurs du futur et cinq opérateurs du passé. La traduction de ces propriétés attendues dans la μ-logique passe par trois phases : traduction des opérateurs temporels, traduction des formules de base, et enfin traduction des propriétés énoncées.
Concernant les opérateurs temporels, une difficulté vient de ce que la μ-logique est adaptée à la traduction de propriétés futures mais pas de propriétés passées. Mais on remarque que dans la spécification, tous les opérateurs du passé sont précédés par un opérateur du futur. Ceci permet en pratique de n'utiliser dans les formules que les opérateurs $\bigcirc, \Box, \Diamond, \mathcal{U}, \mathcal{W}$: on en verra un exemple avec la formule (21) traitée ci-dessous. Les outils clés pour ces traductions sont les opérateurs de point fixe ν et μ.
Nous utilisons par ailleurs une extension, vis-à-vis de la syntaxe donnée plus haut pour la μ-logique. Il s'agit du symbole "-" qui dénote, au sein d'une modalité, une action quelconque. On peut donc utiliser les expressions : $\langle - \rangle$ et $[-]$ au sein d'une formule.
Les formules de base utilisées dans les spécifications sont des prédicats formés à l'aide de fonctions unaires ou binaires comme $Dans(u, b), Bloque(a)$, etc ... La μ-logique ne connaît comme fonctions de base que les égalités ou inégalités entre deux noms de canaux. Nous avons donc procédé à la traduction de toutes ces formules de base, ce qui a nécessité dans certains cas d'introduire dans les processus que nous écrivions des offres d'interactions sur des canaux de contrôle. La justification de ces adjonctions est que les processus obtenus sont équivalents (au sens de la bisimulation) aux processus initialement écrits dès que les canaux de contrôle sont inhibés par une restriction.
La troisième phase, consistant à traduire les spécifications, se réduit à exploiter les deux premières. Notons que les spécifications utilisaient le connecteur d'implication "$\rightarrow$" absent de la syntaxe de la μ-logique retenue ici. Nous avons donc systématiquement remplacé les énoncés "$\varphi \rightarrow \psi$" par leur équivalent "$\neg\varphi \vee \psi$". De plus, bien que la négation $\neg$ existe dans la μ-logique, nous avons préféré l'éviter en la faisant rentrer par dualité à l'intérieur des formules et en n'utilisant que l'inégalité entre noms : "$a \neq b$".

4.1 Traduction des opérateurs temporels

nous exprimons à la suite en μ-logique les cinq opérateurs temporels relatifs à l'expression de propriétés futures.

- L'opérateur *next* ($\bigcirc\ \varphi$) exprime que la formule φ sera vraie après la prochaine action.

$$[-]\varphi$$

- L'opérateur *Always* ($\Box\ \varphi$) exprime que la formule φ est vraie et le restera indéfiniment.

$$(\nu\, X)(\varphi\ \wedge\ \bigcirc X) \quad \text{ou encore} \quad (\nu\, X)(\varphi\ \wedge\ [-]X)$$

- L'opérateur *Eventually* ($\Diamond\varphi$) exprime que φ finira nécessairement par arriver. On entend par là que φ sera effectivement vérifiée pour toutes les dérivations que peut connaître le système.

$$(\mu\, X)(\varphi\ \vee\ (\langle - \rangle\, tt\ \wedge\ [-]X))$$

- L'opérateur *Until* ($\varphi\ \mathcal{U}\ \psi$) exprime que ψ sera vrai un jour, et en qu'en attendant, φ est vrai.

$$(\mu\, X)(\psi\ \vee\ (\varphi\ \wedge\ \langle - \rangle\, tt\ \wedge\ [-]X))$$

- L'opérateur *Unless* ($\varphi\ \mathcal{W}\ \psi$) exprime que φ est vrai tant que ψ n'est pas vrai, ce qui peut ne jamais se produire. φ peut toutefois continuer d'être vraie lorsque ψ le devient. Notons

que cette propriété est vraie pour toutes les dérivations finies pour lesquelles φ est vraie jusqu'au bout, même si ψ ne devient pas vraie.

$$(\nu\, X)(\psi \ \vee \ (\varphi \ \wedge \ [-]X))$$

Pour ces deux opérateurs, plus rien n'est exigé après que ψ soit devenue vraie.

4.2 Formules de base

Nous décrivons dans ce paragraphe quelques formules simples exposées dans [Ledru 98b]. Celles-ci expriment des propriétés élémentaires qui concernent notamment les utilisateurs.

- j_dans(u, b) : l'utilisateur u est dans le bâtiment b. Nous exprimons qu'à réception de la requête "*presence*" associée à un paramètre identifiant un bâtiment, le processus image de l'utilisateur u émet la valeur *vrai* si l'utilisateur est effectivement présent.

  $[u]\,\forall req\ \forall ib\ \forall reps\ \ (req \neq presence \ \vee \ ib \neq b \ \vee \ \langle\, \overline{reps}\,\rangle\, \Sigma\, cr \ \ cr = vrai)$

- u.enAttente(a) : l'utilisateur u a passé sa carte dans le lecteur a avec pour objectif d'entrer ou de sortir, et attend l'ouverture de la porte.

  $\langle\, \overline{acces}\,\rangle\, \Sigma\, id\_carte\ \Sigma\, id\_lecteur\ \Sigma\, etat\ \ (id\_carte = u \ \wedge \ id\_lecteur = a \ \wedge \ etat = attente)$

- u.engagé(a) : l'utilisateur u a ouvert la porte de lecteur a mais ne l'a pas encore refermée.

  $\langle\, \overline{acces}\,\rangle\, \Sigma\, id\_carte\ \Sigma\, id\_lecteur\ \Sigma\, etat\ \ (id\_carte = u \ \wedge \ id\_lecteur = a \ \wedge \ etat = engage)$

- u.enAccès(a) : le fait que l'utilisateur u soit en accès sur la porte de lecteur a s'exprime comme la disjonction des cas précédents.

  $\langle\, \overline{acces}\,\rangle\, \Sigma\, id\_carte\ \Sigma\, id\_lecteur\ \Sigma\, etat\ \ (id\_carte = u \ \wedge \ id\_lecteur = a)$

- u.dans($a.batiment$) : l'utilisateur u est dans un bâtiment dont a est un lecteur.

  $[u]\,\forall rq\ \forall p1\ \forall p2$
  $(rq \ \neq \ situation \ \vee \ \langle\, \overline{p1}\,\rangle \Sigma\, b$
  $[a]\,\forall req\ \forall param$
  $(req \neq batiment \ \vee \ \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\langle\, \overline{param}\,\rangle\, \Sigma\, id\_bat \ \ b = id\_bat))$

- u.autoriséAccès(a) : l'utilisateur u est autorisé à pénétrer dans le bâtiment dont a est un lecteur, si a est un lecteur associé à une porte de sortie alors u est dedans, si au contraire c'est une porte d'entrée, u est à l'extérieur du bâtiment.

  $[a]\,\forall req\ \forall p$
  $(req \ \neq \ batiment \ \vee \ \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\langle\, \overline{p}\,\rangle\, \Sigma\, bat$ - - bat = a.Batiment
  $(([bat]\,\forall req\ \forall p1\ \forall p2\ \forall p3$
  $(req \ \neq \ groupes \ \vee \ \langle\, \overline{p1}\,\rangle\, \Sigma\, lg$ - - lg est la liste des groupes d'utilisateurs autorisés à accéder à bat
  $[u]\,\forall req\ \forall p1\ \forall p2\ \forall p3$
  $(req \ \neq \ groupes \ \vee \ \langle\, \overline{p1}\,\rangle\, \Sigma\, lgu$ - - lgu est la liste des groupes d'utilisateurs auxquels appartient u
  $[interlistes]\,\forall lgru\ \forall lgr\ \forall rp$
  $(lgru \ \neq \ lgu \ \vee \ lgr \ \neq \ lg \ \vee \ \Diamond\langle\, \overline{rp}\,\rangle\, \Sigma\, cr \ \ cr = non\_vide))))$ - - u.autorisé(bat)
  $\wedge$
  $(([a]\,\forall p\ \forall p1\ (p \ \neq \ porte \ \vee \ \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\,\langle\, \overline{p1}\,\rangle\, \Sigma\, b$
  $[b]\,\forall s\ \forall p2\ (s \ \neq \ sens \ \vee \ \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\,\langle\, \overline{p2}\,\rangle\, \Sigma\, sns \ \ sns = sortie)) \ \wedge \ j\_dans(u, bat))$
  $\vee$
  $([a]\,\forall p\ \forall p1\ (p \ \neq \ porte \ \vee \ \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\,\langle\, \overline{p1}\,\rangle\, \Sigma\, b$
  $[b]\,\forall s\ \forall p2\ (s \ \neq \ sens \ \vee \ \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\,\langle\, \overline{p2}\,\rangle\, \Sigma\, sns \ \ sns = entree)) \ \wedge \ u.dehors)))$

- incendie(b): on vérifie que le bâtiment est bien dans l'état incendie.

  $[b]\,\forall req\ \forall p1\ \forall p2\ \forall p3$
  $(req \ \neq \ etat\_incendie \ \vee \ \langle\, \overline{p1}\,\rangle\, \Sigma\, etat \ \ etat = incendie)$

4.3 Propriétés du système d'accès

Nous exprimons à présent les propriétés attendues du système de contrôle d'accès. Celles-ci s'obtiennent en composant les formules précédentes. Il faut bien sûr adapter, sans en trahir le sens, les propriétés apparaissant dans [Ledru 98b] au contexte particulier du *Contrôleur* que nous avons spécifié. Nous avons toutefois dû renoncer à traduire les formules (9) car la signification du symbole $\bigcirc$ telle que nous l'avons définie est liée à la notion de pas d'exécution du programme, c'est-à-dire d'état suivant dans une dérivation. Au contraire, il nous semble que la notion "d'instant suivant" telle qu'elle est utilisée dans les formules (9) est de nature plus macroscopique, et dès lors difficile à exprimer par la μ-logique.
A titre d'exemple, nous donnons à la suite deux formules en forme déployée, ainsi que trois autres où nous nous limitons à préciser les adaptations nécessaires. Nous pensons que ce sous-ensemble est significatif de la spécification globale.

- La formule (10) "$\Box\,(j\_dans(u,b) \rightarrow u.Autorise(b))$" exprime qu'un usager u, qui se trouve dans un bâtiment b est nécessairement autorisé à y être. D'où la forme déployée :

$$
\begin{array}{l}
(\nu\, X)((\quad [u]\,\forall req\,\forall ib\,\forall reps \\
\qquad (req \neq presence \;\vee\; ib \neq b \;\vee\; \langle\, \overline{reps}\,\rangle\,\Sigma\, cr \;\; cr = faux) \\
\qquad \vee \\
\qquad [b]\,\forall rq\,\forall p1\,\forall p2\,\forall p3 \\
\qquad (rq \neq groupes \;\vee\; \langle\, \overline{p1}\,\rangle\,\Sigma\, lg \\
\qquad [u]\forall g\,\forall reps\,(g \neq groupes \;\vee\; \langle\, \overline{reps}\,\rangle\,\Sigma\, lgu \\
\qquad [interlistes]\,\forall\, lgru\,\forall\, lgr\,\forall rp \\
\qquad (lgru \neq lgu \;\vee\; lgr \neq lg \;\vee\; (\mu\, Y)(\langle\, \overline{rp}\,\rangle\,\Sigma\, cr \;\; cr = non\_vide \;\vee\; (\langle - \rangle tt \;\wedge\; [-]\, Y)))))) \\
\wedge \\
[-]\, X)
\end{array}
$$

- La formule (31) "$\Box\,(incendie(b) \rightarrow \Box\, incendie(b))$" exprime que l'état d'incendie est un état stable du contrôleur. Nous avons dû modifier légèrement cette définition dans la mesure où nous distinguons les états d'incendie pour chacun des bâtiments. D'où la forme déployée :

$$
\begin{array}{l}
(\nu\, X)((\quad [b]\,\forall req\,\forall p1\,\forall p2\,\forall p3 \\
\qquad (req \neq etat\_incendie \;\vee\; \langle\, \overline{p1}\,\rangle\,\Sigma\, etat \;\; etat = pas\_incendie) \\
\qquad \vee \\
\qquad (\nu\, Y)(\;\; [b]\,\forall req\,\forall p1\,\forall p2\,\forall p3 \\
\qquad\qquad (req \neq etat\_incendie \;\vee\; \langle\, \overline{p1}\,\rangle\,\Sigma\, etat \;\; etat = incendie) \;\wedge\; [-]\, Y)) \\
\wedge \\
[-]\, X)
\end{array}
$$

- La formule (17) "$\Box\,((u.souhaite\_entrer(a) \;\wedge\; u.autoriseAcces(a)) \rightarrow \Diamond \neg a.bloque)$" exprime qu'un usager autorisé qui souhaite entrer, finit par en avoir la possibilité. Nous interprétons le fait qu'un usager souhaite entrer par le fait qu'il passe sa carte dans le lecteur associé à la porte d'entrée considérée. Ceci se traduit dans le *Contrôleur* par la réception d'un message de `lecture` de carte :

$$\Box\,([lecture]\,\forall il\,\forall ic\,(il \neq a \;\vee\; ic \neq u \;\vee\; u.non\_autoriseAcces(a) \;\vee\; \Diamond a.debloque))$$

- La formule (21) "$\Box\,((\ominus\, a.bloque \;\wedge\; a.debloque \;\wedge\; time = k) \rightarrow \Diamond((time = k+30 \;\wedge\; a.bloque) \;\vee\; a.ouvert))$" exprime que le *Contrôleur* rebloque la porte trente secondes après le déblocage si celle-ci n'a pas été ouverte entre-temps. Ne disposant pas d'opérateurs temporels du passé, ni de moyen de mesurer le temps, nous modifions sensiblement la formule en nous

plaçant non pas à l'instant où la porte est débloquée par le *Contrôleur*, mais à l'instant précédent, c'est-à-dire celui où le timer est armé :

$$
\begin{array}{l}
[a]\,\forall\, req\;\forall\, param\;(req \neq porte \vee \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle[\overline{param}]\,\Sigma\, b \\
(\Box\;([\overline{timer}]\,\Sigma\,\delta t\,\Sigma\, t\;(t \neq trente \vee \\
\quad [\overline{a}]\,\Sigma\, inst\,\Sigma\, n\;(inst \neq allumer\_vvert \vee \\
\quad [\overline{b}]\,\Sigma\, instr\,\Sigma\, m\;(instr \neq debloc \vee \\
\quad (\langle\,\delta t\,\rangle\,\langle\,\overline{b}\,\rangle\,\Sigma\, instru\,\Sigma\, l\;\; instru = bloc \\
\quad \wedge \\
\quad \langle\tau\rangle\langle\,\overline{acces}\,\rangle\,\Sigma\, ic\,\Sigma\, ip\,\Sigma\, e\;(ic = u \;\wedge\; ip = b \;\wedge\; e = engage)))))))
\end{array}
$$

- - le τ correspond à la synchronisation sur le canal estp dans le processus *Cycle*

- La formule (33) "$\Box\,(incendie \rightarrow \Diamond \neg\, a.bloque)$" exprime qu'en cas d'incendie, la porte de lecteur a finit par être débloquée. Dans le cas de notre spécification, nous disposons de la propriété supplémentaire qui est que la porte demeurera indéfiniment débloquée. Comme nous l'avons déjà mentionné, nous gérons les situations d'incendie bâtiment par bâtiment :

$$
\begin{array}{l}
[a]\,\forall\, req\,\forall\, p\;(req \neq batiment \vee \langle\tau\rangle\langle\tau\rangle\langle\tau\rangle\langle\,\overline{p}\,\rangle\,\Sigma\, bat \\
\Box\,(pas\_incendie(bat) \vee \Diamond\Box\, a.debloque))
\end{array}
$$

5 Expérimentation avec le Mobility Workbench

Les assistants de preuve adaptés au π-calcul et à la μ-logique sont très peu nombreux. Malgré tout, la version du π-calcul que nous avons choisie dans cet article permet de disposer du Mobility Workbench [Victor 95] avec l'extension [Beste 98], qui autorise une exploitation expérimentale de la μ-logique.
Une première caractéristique liée à l'utilisation de ces systèmes est que la forme tant des processus que des formules doit être mise en conformité avec un format interne qui leur est propre. Outre son caractère fastidieux, cette opération rend la spécification résiduelle très peu lisible.
Le Mobility Workbench dans sa version d'origine possède un model checker, dû à Mads Dam qui dispose d'un pouvoir expressif élevé, quasiment identique à celui de la μ-logique, mais au détriment de l'efficacité à l'exécution. L'extension proposée par Beste est basée sur le calcul des séquents et permet dans certains cas d'obtenir des résultats plus rapidement. Toutefois, ceci n'a pu être obtenu que par le biais d'une réduction importante du pouvoir expressif, tant au point de vue du π-calcul : suppression de l'opérateur de matching, que de la μ-logique : pas d'instruction "-" et pas d'abstraction possible sur les actions figurant dans les modalités.
Après avoir tenté de réaliser l'expérimentation, il nous a fallu renoncer à utiliser ces outils pour valider le *Contrôleur*, le système s'effondrant manifestement sous la charge.
Finalement, la μ-logique semblant posséder les caractéristiques requises pour la description des propriétés voulues, il nous est apparu très regrettable de ne pouvoir valider le système spécifié.

6 Evaluation et possibilités d'extension du modèle

Cette étude de cas nous a paru très intéressante et relativement facile à traiter en π-calcul. De fait, de nombreux aspects du système à spécifier ont pu aisément se ramener à des concepts de nature interactionnelle. Toutefois, le π-calcul étant un formalisme à caractère très opérationnel, nous avons sans doute plus spécifié *une* solution pour le *Contrôleur* qu'un composant logiciel doté de propriétés véritablement abstraites.
De ce point de vue, on ne manquera pas de constater ici l'absence de référence à la notion de raffinement, très actuelle dans le domaine des spécifications formelles, tout particulièrement avec le langage B. En fait, il nous apparaît que l'étude de l'intersection entre théorie de la concurrence et théorie du raffinement reste une question très ouverte. Il serait ainsi intéressant d'inverser

la démarche de type model-checking suivie dans cet article, pour essayer de dériver vers une spécification en π-calcul un ensemble de propriétés exprimées dans une logique temporelle.
Concernant la possibilité de générer du code exécutable à partir de la spécification, il semble qu'une solution soit plus clairement et plus directement envisageable. En effet, sur la base d'une spécification en π-calcul asynchrone, il paraît facile de dériver un code compilable écrit dans le langage PICT et d'obtenir ainsi une maquette du système. Certes, l'outil réalisant automatiquement cette transformation n'existe pas à l'heure actuelle mais la faisabilité de cette démarche ne fait guère de doute.

S'agissant du problème des extensions, nous constatons que le niveau de modularité induit par un formalisme exprimant la concurrence a permis de prendre en compte d'emblée un certain nombre de caractéristiques qui n'étaient à l'origine prévues qu'en extension, telles que la gestion des accès et des états d'incendie bâtiment par bâtiment ou l'affiliation d'un utilisateur à plusieurs groupes. Nous pensons pouvoir traiter tout aussi bien des problèmes d'accès liés à un horaire, la spécialisation des portes ou la modification dynamique des droits d'accès, ces questions se ramenant à la définition de nouvelles requêtes dans les processus existants ou à celle de nouveaux processus.
D'autres aspects présents dans le cahier des charges n'ont pas été traités, essentiellement pour trois types de raisons :

- Une impossibilité pure et simple : l'introduction du temps, ici envisagée à travers le processus *Horloge* non décrit, échappe a priori à toute description en π-calcul.
- Une faisabilité théorique, mais une démarche fastidieuse : la notion de code de détection d'erreur pourrait certes théoriquement être spécifiée sur la base d'un codage des entiers et d'un opérateur de division, mais cela ne nous a pas semblé raisonnablement envisageable.
- Une mise en œuvre assez simple mais qui aurait alourdi la spécification sans rien ajouter à la démonstration : on peut à ce niveau citer les problèmes de gestion de comptes-rendus de messages et de condamnation des portes ainsi que le traitement des problèmes de fiabilité du système de communication.

7 Conclusion

Nous pensons avoir démontré dans cet article la capacité du π-calcul et de la μ-logique, du moins en ce qui concerne leur pouvoir expressif, à traiter l'étude de cas du système de contrôle d'accès, même si le résultat de la phase d'expérimentation s'avère décevant. Il faut de ce point de vue constater que les efforts consacrés au développement d'outils destinés au π-calcul (un Phd et un master thesis) apparaissent extrêmement modestes si on les compare à ceux dont ont pu bénéficier d'autres techniques de spécification telles que B ou Lotos.
En tout état de cause, l'utilisation du π-calcul, qui constitue une perspective intéressante du fait qu'il s'agit d'une algèbre de processus mobile, ne saurait être raisonnablement envisagée que dans la mesure où on disposera d'outils associés de qualité.

Références

[Beste 98] Beste F. : The Model Prover - a sequent-calculus based modal μ-calculus model checker tool for finite control π-calculus agents. Master thesis. Uppsala University, Sweden. (1998)
[Hoare 85] Hoare C.A.R.: Communicating Sequential processes. Prentice Hall, (1985)
[Ledru 98a] Ledru Y.: Etude de cas : système de contrôle d'accès. Document interne. LSR-IMAG. (1998)
[Ledru 98b] Ledru Y.: Propriétés attendues du contrôle d'accès. Document interne. LSR-IMAG. (1998)
[Milner 80] Milner R.: A Calculus of Communicating Systems. LNCS 92. Springer-Verlag. (1980)
[Milner 92] Milner R., Parrow J., Walker D.: a Calculus of Mobile processes. Journal of Information and Computation; 100. (1992)
[Milner 93] Milner R.: The polyadic π-calculus : a tutorial, in: F.L. Bauer, W. Brauer and H. Schwichtenberg (editors). Logic and Algebra of Specification. Springer-Verlag. (1993)
[Victor 95] Victor B.: The Mobility Workbench User's Guide, Polyadic version, Internal Report, Department of Computer Systems. Uppsala University, Sweden. (1995)