Systèmes de gestion de ressources et aspects de disponibilité

Pascal Fradet[†] — **Stéphane Hong Tuan Ha**[‡]

[†] *INRIA Rhône-Alpes, 655, av. de l'Europe, 38330 Montbonnot, France*
Pascal.Fradet@inria.fr

[‡] *IRISA/INRIA Rennes, Campus de Beaulieu, 35042 Rennes, France*
Stephane.Hong_Tuan_Ha@irisa.fr

RÉSUMÉ. *Dans cet article, nous nous intéressons aux propriétés de* disponibilité *et à la prévention des dénis de service. Nous proposons un langage d'aspects dédié permettant de prévenir les dénis de service liés à la gestion des ressources. Nos aspects spécifient des politiques de disponibilité en imposant des limites de temps d'allocation des ressources. Dans notre langage, un aspect peut être vu comme une propriété temporelle sur les traces d'exécution. Les programmes et les aspects sont modélisés par des automates temporisés et le tissage par un produit d'automates. L'avantage de cette approche formelle est double : d'une part, l'utilisateur garde la maîtrise de l'impact sémantique du tissage et, d'autre part, il peut utiliser des outils de model-checking pour optimiser le tissage et vérifier des propriétés de disponibilité.*

ABSTRACT. *In this paper, we focus on availability properties and the prevention of denial of services. We propose a domain-specific aspect language to prevent the denials of service caused by resource management. Our aspects specify availability policies by enforcing time limits in the allocation of resources. In our language, an aspect can be seen as a formal temporal property on execution traces. Programs and aspects are specified as timed automata and the weaving process as an automata product. The benefit of this formal approach is two-fold: the user keeps the semantic impact of weaving under control and (s)he can use a model-checker to optimize the woven program and verify availability properties.*

MOTS-CLÉS : *Gestion de ressources, disponibilité, aspect, tissage, vérification, denis de service*

KEYWORDS: *Resource management, availability, aspect, weaving, verification, denial of service*

Ce travail a été effectué dans le cadre et avec le soutien de l'action concertée incitative DISPO.

1re soumission à *L'Objet*, le 5 avril 2006

1. Introduction

Avec la confidentialité et l'intégrité, la *disponibilité* est l'une des trois grandes classes de propriétés de sécurité. La disponibilité assure que les demandes faites par des sujets autorisés sont toujours traitées ; on dit qu'il n'y a pas de *dénis de service*. Dans cet article, nous étudions la gestion des ressources en isolation (*i.e.,* séparée de la fonctionnalité de base) et la prévention des dénis de service (*i.e.,* la disponibilité) comme des aspects.

Nous proposons un langage d'aspects dédié permettant de prévenir les dénis de service liés à la gestion des ressources (famines, interblocages). Nos aspects spécifient des politiques de disponibilité en imposant des limites de temps d'allocation des ressources. Par exemple, on peut vouloir imposer qu'un service n'accapare pas la ressource `R` plus de 10 secondes ou qu'il alloue impérativement la ressource `R1` avant la ressource `R2`. A notre connaissance, ce travail est le premier à proposer une approche orientée aspect pour la gestion et la disponibilité des ressources.

Dans notre langage, un aspect peut être vu comme une propriété temporelle sur les traces d'exécution. Les programmes et les aspects sont modélisés par des *automates temporisés* (Alur *et al.*, 1994, T.A. Henzinger *et al.*, 1992). L'automate correspondant à un aspect décrit l'ensemble des traces temporelles autorisées. Ceci permet de voir le tissage comme un *produit d'automates* (*i.e.,* une intersection des traces d'exécution) qui restreint le comportement du programme aux traces autorisées par l'aspect.

L'avantage principal de cette approche formelle est double :

– elle permet de garder la maîtrise de l'impact sémantique du tissage et de raisonner sur les programmes tissés,

– elle permet d'utiliser des outils automatiques de model-checking (par ex. UPPAAL (Larsen *et al.*, 1997, Bengtsson *et al.*, 2004)) pour optimiser le tissage et vérifier que les aspects imposent les propriétés de disponibilité attendues.

L'article est organisé comme suit. La section 2 présente rapidement le cadre : les systèmes et les problèmes de disponibilité considérés, notre approche et un exemple utilisé au fil de l'article pour illustrer les différentes étapes. En section 3, nous rappelons les caractéristiques du formalisme central à cette étude : les automates temporisés. Les sections 4 et 5 présentent la syntaxe et la sémantique des services et les aspects de disponibilité, respectivement. La section 6, cœur technique de l'article, décrit l'abstraction des services en automates temporisés, la sémantique des aspects en terme d'automates temporisés, le tissage comme un produit, des optimisations et vérifications sur l'automate résultant. La concrétisation de cet automate en un code source utilisant des commandes temporelles (minuteurs, attentes et interruptions) est esquissée en section 7. Nous discutons rapidement des travaux connexes et des extensions possibles en conclusion (Section 8).

2. Cadre général

Nous commençons par présenter les systèmes et les problèmes de disponibilité considérés. Nous présentons ensuite les grandes lignes de notre approche et l'exemple nous servant à illustrer les principales étapes.

2.1. *Systèmes et disponibilité*

Les systèmes considérés sont structurés selon trois couches : les *utilisateurs*, les *services* et les *ressources* (Figure 1).

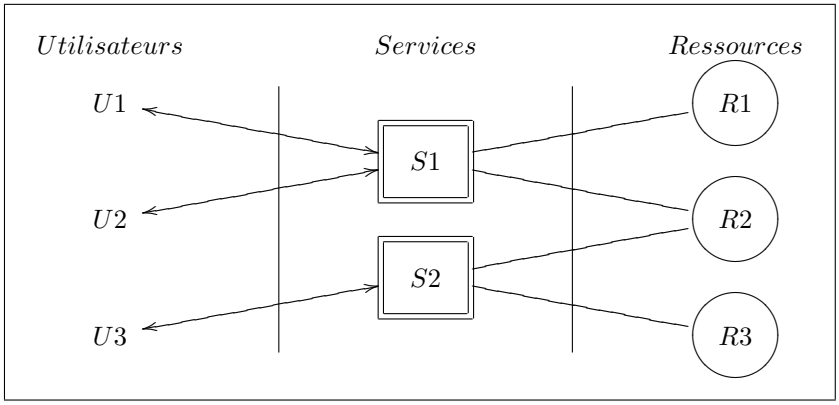

Figure 1. *Modélisation en 3 couches*

Les utilisateurs envoient des requêtes aux services et attendent la réponse à leur requête. Les services traitent séquentiellement les demandes des utilisateurs. Les requêtes des utilisateurs à un service sont mémorisées (par ex. dans une file FIFO) ; le traitement d'une requête par un service entraîne du calcul et le plus souvent des accès à des ressources. Les ressources sont des entités (logiques ou physiques) qui sont partagées entre les différents services. On peut citer comme exemples de ressources, des fichiers, une imprimante, un processeur ou encore le gestionnaire de connexions réseau.

Notre modèle correspond à une architecture client-serveur. Ce type d'architecture est couramment utilisé et se retrouve, par exemple, sur la majorité des serveurs sur le web et applications distribuées. Nous faisons l'hypothèse d'un nombre fixé et connu de services et de ressources. Cette hypothèse, simplificatrice mais raisonnable, correspond, par exemple, au fonctionnement d'un serveur web (Banga *et al.*, 1999).

Chaque service peut être vu comme une boucle sans fin de traitement de requêtes : la requête d'un utilisateur est lue, le traitement correspondant est réalisé, le résultat est retourné à l'utilisateur, et ainsi de suite. Notre but étant la gestion des ressources et

la prévention des dénis de service, nous ne décrivons pas les utilisateurs (qui peuvent être des processus quelconques) ni leur gestion par les services. Nous nous focalisons sur les interactions entre les services et les ressources.

Les problèmes de disponibilité auxquels nous nous intéressons proviennent de la concurrence entre les services pour accéder aux ressources. Il s'agit, par exemple, de famine quand un service ne réussit pas à accéder à une ressource ou d'interblocage quand tous les services sont bloqués en attente des ressources possédées par un autre service. Ces problèmes sont d'ordre logiciel et peuvent être prévenus par une gestion des ressources adaptée. Clairement, des problèmes de dénis de service peuvent également résulter de fautes matérielles qui doivent être prévenues par des techniques relevant de la tolérance aux fautes (voir par ex. (Laprie *et al.,* 1992, Rushby, 1994)).

Yu et Gligor (Yu *et al.,* 1990) ont étudié en détail le problème de dénis de service liés à la gestion des ressources. Ils montrent que pour vérifier une propriété de disponibilité il est nécessaire de connaître les ressources mais aussi de contraindre le comportement des services (par des *user agreements*).

2.2. *Approche*

Notre système de gestion des ressources est composé de deux parties sur le modèle de Yu et Gligor. La première partie consiste en la spécification des ressources et leur traitement des requêtes. La seconde partie décrit les contraintes que les services doivent respecter dans leur utilisation des ressources. Nous introduisons dans ce but des *aspects de disponibilité* qui sont tissés sur les services. L'originalité de ce type d'aspects est de d'intégrer des notions temporelles pour pouvoir, par exemple, limiter le temps d'allocation d'une ressource à un service ou interdire des reallocations de ressources trop rapprochées (Section 5).

La figure 2 récapitule la structure de notre système de gestion de ressources. Celui-ci est constitué de la spécification des différentes ressources et des aspects de disponibilité. Les ressources sont spécifiés en terme d'automates UPPAAL suffisamment précis pour être traduits en programmes. Le lecteur peut se reporter à (Fradet *et al.,* 2005) qui donne quelques exemples de spécification de ressources (*e.g.,* à accès exclusif, partageables).

Dans cet article, nous nous concentrons sur la partie orientée aspect du cadre. Les contraintes de gestion des ressources sont spécifiées à l'aide d'un aspect par service. Chacun de ces aspects est indépendant et définit une propriété locale qui sera tissée sur le service. Ces aspects correspondent aux *user agreements* de Yu et Gligor. Nous avons fait le choix de ne pas considérer d'aspects globaux qui contraignent certains services en fonction du comportement d'autres. Les aspects globaux sont certes plus expressifs et plus précis mais ils ne sont pas facilement tissables sur les codes des différents services. Ils sont plus naturellement mis en œuvre par un moniteur observant l'exécution du système complet. Nous nous concentrons ici sur les aspects locaux qui

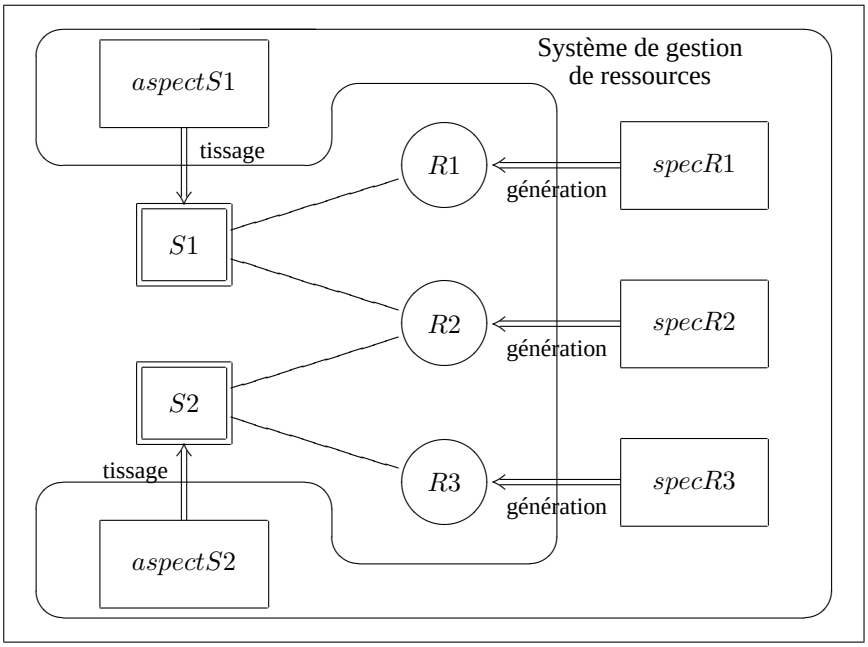

Figure 2. *Vision globale du système avec gestion des ressources séparée*

sont suffisamment expressifs pour prévenir les dénis de service et dont l'implémentation peut être optimisée en les tissant statiquement.

Le formalisme sous-jacent à toute l'approche est celui des automates temporisés et l'opération centrale de tissage est le produit d'automates temporisés. Le cœur technique de notre approche est composé des phases suivantes :

– Le service est abstrait en un automate temporisé sur-approximant ses traces d'exécution et son comportement temporel (Section 6.1).

– Si les aspects sont spécifiés dans une syntaxe textuelle, leur sémantique est donnée par un automate temporisé (Section 6.2).

– L'aspect est intégré au service en faisant le produit des deux automates. Le produit effectuant une intersection des traces d'exécution, l'automate résultant respectera le comportement du programme *et* les contraintes de l'aspect (Section 6.3).

– Afin d'optimiser l'automate obtenu, des informations sur les durées d'exécution des instructions du service sont prises en compte en utilisant une nouvelle fois le produit d'automates (Section 6.4).

– A cette étape, il est possible de montrer automatiquement (par ex. avec UPPAAL) que le système possède les propriétés de disponibilité attendues (Section 6.5).

– La dernière étape consiste à concrétiser l'automate (optimisé et vérifié) en code source instrumenté avec des commandes temporelles (minuteurs, attentes et interruptions) (Section 7).

2.3. *Exemple de système*

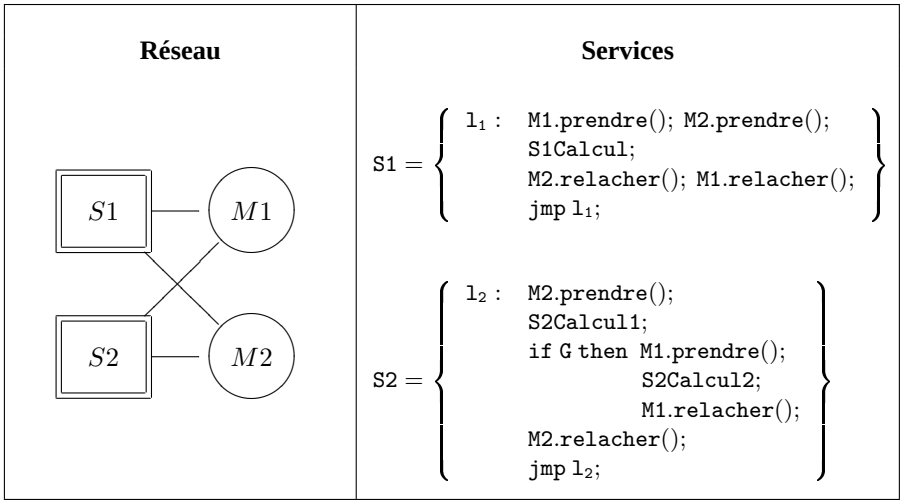

Figure 3. *Exemple de système à deux services et deux ressources*

L'exemple de la figure 3 nous sert à illustrer les différentes étapes (abstraction, tissage, vérification, concrétisation). Ce petit système est composé de deux ressources (`M1` et `M2`) et de deux services (`S1` et `S2`) de type boucle sans fin. Les deux ressources sont supposées être en accès exclusif.

Le service `S1` commence par prendre la ressource `M1` puis `M2` (`M1.prendre()` ; `M2.prendre()` ;). Ensuite il effectue le calcul `S1Calcul` (qui dure entre 2 et 10 secondes), relâche les ressources `M2` puis `M1` et réitère. Le service `S2` modélise un service potentiellement dangereux. Il commence par prendre la ressource `M2` (`M2.prendre()`). Ensuite il effectue le calcul `S2Calcul1` qui dure plus de 1 seconde (et peut ne pas terminer). Si la garde `G` est vraie, il prend `M1`, effectue le calcul `S2Calcul2` (qui dure entre 3 et 20 sec.) et relâche `M1`. Il termine en relâchant `M2` et réitère.

Deux problèmes de disponibilité liés aux ressources peuvent apparaître dans ce système :

– Un dénis de service de type famine peut se produire si `S2Calcul1` ne termine pas. Dans ce cas, le service `S2` ne relâche pas la ressource `M2` ce qui peut bloquer le service `S1`.

– Un interblocage peut également se produire : le service `S1` possède la ressource `M1` et attend la ressource `M2` alors que le service `S2` possède la ressource `M2` et attend la ressource `M1`.

3. Les automates temporisés

Nous présentons dans cette section la syntaxe et la sémantique des automates temporisé, formalisme utilisé pour modéliser les programmes, les aspects et le tissage. Les automates temporisés permettent de modéliser des problèmes, de vérifier des propriétés, où le temps est explicite. Nous décrivons les *Timed Safety Automata* (Larsen *et al.*, 1997, Alur, 1999) qui est un modèle d'automate temporisé couramment utilisé.

3.1. *Syntaxe*

Nous notons $\mathcal{H}$ un ensemble de variables à valeurs réelles servant à représenter les horloges. Une *contrainte d'horloge* C est de la forme

$$C ::= x \odot k \mid x - y \odot k \quad \text{avec } x, y \in \mathcal{H},\ \ k \in \mathbb{N} \text{ et } \odot \in \{\leq, <, =, >, \geq\}$$

Les transitions des automates temporisés sont gardés par un ensemble de contraintes d'horloges qui doit être interprété comme la conjonction des différentes contraintes. Nous notons 2^C l'ensemble des gardes possibles (*i.e.,* les ensembles des contraintes d'horloges).

Un automate temporisé A est un n-uplet $(Q, q_0, H, \Sigma, \delta, I)$ où :

– Q est un ensemble fini d'états ;

– $q_0 \in Q$ est l'état initial ;

– H est un ensemble fini d'horloges ($H \in \mathcal{H}$) ;

– Σ est un alphabet fini dénotant les actions de l'automate ;

– $\delta \subseteq Q \times 2^C \times \Sigma \times 2^H \times Q$ est la relation de transition ;

– $I : Q \rightarrow 2^C$ est une fonction associant une garde (un invariant) aux états.

Une transition $(q, g, a, r, q') \in \delta$ spécifie que l'on peut passer de l'état q à l'état q', en effectuant l'action a et en remettant l'ensemble d'horloges r à zéro ($r \in H$) si la garde g est vraie. Le sous-ensemble des horloges r est appelé une *remise à zéro (raz)*. Nous restreignons les invariants d'état à des conjonctions de contraintes de la forme $x \leq k$ ou $x < k$ avec k un entier naturel.

Nous dénotons par le même symbole . à la fois la garde vide (*i.e.,* $\emptyset$ ou $\mathbf{true}$), la raz vide (*i.e.,* $\emptyset$) et l'action vide, notée plus traditionnellement ϵ ($\epsilon \notin \Sigma$). Nous utilisons également la notation $q \xrightarrow{g,a,r} q'$ pour les transitions ; par exemple, $q \xrightarrow{\cdots} q'$ modélise la transition spontanée.

3.2. *Sémantique*

La sémantique opérationnelle d'un automate temporisé $(Q, q_0, H, \Sigma, \delta, I)$ est donnée par un système de transition où un état (q, u) est composé de l'état courant $q \in Q$ de l'automate temporisé et la fonction $u : H \to \mathbb{R}$ associe les horloges à leur valeur courante. L'état initial est composé de l'état initial de l'automate temporisé et de la fonction associant 0 à toutes les horloges.

Pour définir la relation de transition, nous utilisons les notations suivantes :

– $u \in g$ signifie que les valeurs des horloges défini par u vérifient la garde g ;

– $u + d$ signifie que l'on incrémente toutes les horloges par d ;

– $u[r \mapsto 0]$ signifie que les horloges de l'ensemble r sont remises à 0.

Les transitions sont :

– soit des transitions représentant un écoulement du temps

$$(q, u) \to (q, u + d) \text{ si } \forall d' : O \leq d' \leq d \Rightarrow u + d' \in I(q)$$

– soit des transitions représentant l'exécution d'une action

$$(q, u) \to (q', u') \text{ s'il existe } q \xrightarrow{g,a,r} q' \in \delta \text{ tel que } u \in g, u' \in I(q'), u' = u[r \mapsto 0]$$

Un écoulement du temps est possible uniquement s'il respecte l'invariant de l'état courant. Une transition de l'automate peut être effectuée si et seulement si sa garde et l'invariant du nouvel état est vérifiée. La sémantique de l'automate temporisé est donnée par l'ensemble des traces de ce système de transition associé.

3.3. *Produit d'automates temporisés*

Le produit de deux automates temporisés $X = (Q_x, x_0, H_x, \Sigma, \to_x, I_x)$ et $Y = (Q_y, y_0, H_y, \Sigma, \to_y, I_y)$ avec le même ensemble d'actions et des horloges disjointes, est l'automate $X \otimes Y = (Q_x \times Q_y, (x_0, y_0), H_x \cup H_y, \Sigma, \to, I)$ avec :

$$I(x, y) = I_x(x) \cup I_y(y)$$

$$\textsc{Action} \quad \frac{x_1 \xrightarrow{g_x,a,r_x}_x x_2 \qquad y_1 \xrightarrow{g_y,a,r_y}_y y_2}{(x_1, y_1) \xrightarrow{g_x \cup g_y, a, r_x \cup r_y} (x_2, y_2)}$$

$$\textsc{Vide}_1 \quad \frac{x_1 \xrightarrow{g_x,.,r_x}_x x_2}{(x_1, y) \xrightarrow{g_x,.,r_x} (x_2, y)} \qquad \textsc{Vide}_2 \quad \frac{y_1 \xrightarrow{g_y,.,r_y}_y y_2}{(x, y_1) \xrightarrow{g_y,.,r_y} (x, y_2)}$$

L'ensemble d'états de l'automate produit est le produit cartésien des deux ensembles d'états des automates arguments. L'état initial est constitué des états initiaux des deux automates. L'invariant d'un état du produit est la conjonction des invariants des deux états constituants.

Le produit calcule un nouvel automate temporisé qui reconnaît l'intersection des traces de deux automates temporisés. La règle ACTION modélise le cas où une action est effectuée par les 2 automates. La garde de cette transition est la conjonction des gardes temporelles et l'ensemble des horloges à remettre à zéro est l'union des deux raz. Les règles VIDE_1 et VIDE_2 modélisent le cas où un des deux automates effectue une action vide. Dans ce cas chacun peut évoluer indépendamment de l'autre.

Les traces d'exécution de l'automate produit $X \otimes Y$ est l'intersection des traces des deux automates X et Y.

4. Les services

Le langage que nous utilisons est simple mais suffisamment expressif. Son avantage principal est que les programmes sources restent proches de leur graphe de flot de contrôle et donc de leur représentation en terme d'automate. Un langage de plus haut niveau pourrait être considéré en ajoutant une étape préliminaire d'analyse de flot de contrôle.

4.1. *Syntaxe*

Un service est défini par un ensemble d'*instructions* $\{I_1, \ldots, I_n\}$ de la forme

$$I \quad ::= \quad l_1 : c \rightsquigarrow l_2 \quad | \quad l_1 : g \rightsquigarrow l_2 \; ; \; l_3$$

où l_1, l_2 et l_3 sont des *étiquettes*, c une *commande* (*e.g.,* une affectation) et g un *test* (*i.e.,* une expression booléenne). Par la suite, on utilise le terme *action* pour désigner indifféremment une commande ou un test. Intuitivement, si le point de programme courant est l_1 et le service S contient l'instruction :

– $l_1 : c \rightsquigarrow l_2$ alors la commande c est exécutée et le point de programme courant devient l_2,

– $l_1 : g \rightsquigarrow l_2 \; ; \; l_3$ si le test g est vrai alors le point de programme courant devient l_2 sinon il passe à l_3.

Dans cet article, nous considérons seulement des services composés d'instructions dont l'étiquette gauche est unique. Cette restriction syntaxique assure la séquentialité et le déterminisme des services (pourvu que les commandes le soient).

Un service se représente par une boucle infinie de traitement qui attend une requête d'un utilisateur, effectue un traitement/calcul pour cette requête, renvoie à l'utilisateur la réponse à sa requête, puis traite une nouvelle requête. Un service commence par

l'instruction l_0 : `getUser`$() \rightsquigarrow l_1$ qui attend et prend une nouvelle requête d'utilisateur et se termine par l_i : `endUser`$() \rightsquigarrow l_0$ qui renvoie les résultats à l'utilisateur et retourne à l_0 pour traiter une nouvelle requête.

Par exemple, le service `S1` de la figure 3 s'écrit dans cette syntaxe :

$$\texttt{S1} = \left\{ \begin{array}{lcll} l_0 & : & \texttt{getUser}() & \rightsquigarrow \; l_1 \\ l_1 & : & \texttt{M1.prendre}() & \rightsquigarrow \; l_2 \\ l_2 & : & \texttt{M2.prendre}() & \rightsquigarrow \; l_3 \\ l_3 & : & \texttt{S1.calcul}() & \rightsquigarrow \; l_4 \\ l_4 & : & \texttt{M2.relacher}() & \rightsquigarrow \; l_5 \\ l_5 & : & \texttt{M1.relacher}() & \rightsquigarrow \; l_6 \\ l_6 & : & \texttt{endUser}() & \rightsquigarrow \; l_0 \end{array} \right\}$$

Les commandes `getUser`$()$, `prendre`$()$ sont bloquantes (*e.g.,* s'il n'y a de requête ou si la ressource n'est pas libre) ; la commande `S1.calcul` correspond en fait à une agrégation d'actions élémentaires qui n'effectuent aucune opération de gestion des ressources.

4.2. *Sémantique*

La sémantique d'un service S est exprimée comme un système de transition étiquetée (LTS) $(\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$ où :

– Σ_p est un ensemble (infini) d'états (l, s) avec l une étiquette et s une mémoire,

– (l_0, s_0) est l'état initial,

– $\mathcal{E}_S$ est l'ensemble des actions de S,

– $\longrightarrow_S$ est une fonction de transition sur les états étiquetée par l'action (commande ou test) courante.

La sémantique des commandes c est donnée par la fonction $\mathcal{C}[\![c]\!]$ qui prend la mémoire courante et rend la mémoire modifiée par la commande. La sémantique des tests est donnée par la fonction $\mathcal{G}[\![g]\!]$ qui prend la mémoire courante et rend un booléen.

La fonction de transition est définie par les trois règles suivantes très classiques :

$$\textsc{Comm} \quad \frac{l_1 : c \rightsquigarrow l_2 \in S \quad \mathcal{C}[\![c]\!]s_1 = s_2}{(l_1, s_1) \xrightarrow{c}_S (l_2, s_2)}$$

$$\textsc{Then} \quad \frac{l_1 : g \rightsquigarrow l_2 \; ; \; l_3 \in S \quad \mathcal{G}[\![g]\!]s_1}{(l_1, s_1) \xrightarrow{g}_S (l_2, s_1)} \qquad \textsc{Else} \quad \frac{l_1 : g \rightsquigarrow l_2 \; ; \; l_3 \in S \quad \neg\mathcal{G}[\![g]\!]s_1}{(l_1, s_1) \xrightarrow{g}_S (l_3, s_1)}$$

5. Aspects de disponibilité

Dans le domaine de la disponibilité, on considère souvent des politiques en *temps fini* où l'on assure (par analyse statique ou vérification) que les requêtes des utilisateurs seront finalement traitées «un jour». Ce style de propriété (de type vivacité) n'est pas imposable par un moniteur, par tissage ou instrumentation de code (Schneider, 2000). Seules des propriétés de sûreté peuvent s'imposer de cette façon. Aussi, nous nous intéressons à des politiques de disponibilité de type *temps borné*. Plus précisément, nous souhaitons assurer une réponse en un temps borné (fixé par l'aspect) aux requêtes des services. D'autres propriétés temporelles correspondant à une gestion plus fine des ressources peuvent également être spécifiées. Par exemple, pour imposer un partage plus équitable des ressources, on peut vouloir limiter la fréquence de prise de ressource par un service (*i.e.,* imposer des attentes).

Les aspects de disponibilité ont principalement comme effet d'imposer des durées maximales ou minimales. Ils sont définis dans un langage textuel qui peut être automatiquement traduit en automate. Le formalisme des automates temporisés est clairement bien adapté pour exprimer ce style de propriétés temporelles.

5.1. *Syntaxe*

Notre langage est inspiré des *stateful aspects* introduits dans (Douence *et al.*, 2002) qui prennent en compte l'histoire de l'exécution. La grammaire du langage est décrite en figure 4. Un aspect est composé d'une collection d'équations mutuellement récursives. Une équation est de la forme $a_i = (C \triangleright L); a_j$: l'aspect attend l'évènement C qui, lorsqu'il survient, déclenche l'exécution d'une liste d'instructions L et fait passer le contrôle à l'équation a_j. En général une équation peut comporter des choix comme $(C \triangleright L); a \,\Box\, (C' \triangleright L'); a'$: l'aspect attend les évènements C ou C', le premier arrivant déclenche l'exécution des instructions et équations correspondantes (L et a ou L' et a'). Pour assurer le déterminisme, nous supposons que les choix sont exclusifs[1].

Le filtre d'évènement F, proche des poincuts d'AspectJ (Kiczales *et al.*, 2001) ou des compositions de filtres de (Aksit *et al.*, 1994), est soit un motif, soit une combinaison logique de filtres. Un motif est un terme (éventuellement avec des jokers `*`) filtrant les instructions du programmes. Par exemple, `R.prendre` filtre uniquement la prise de la ressource `R`, `*.prendre` filtre toutes les prises de ressources et `R.*` toutes les opérations sur `R`. Une garde G est une conjonction (représentée par un ensemble) de comparaisons de minuteurs à des entiers.

La liste L dénote une liste d'inserts (*advice* dans la terminologie AspectJ) à exécuter lorsque l'évènement correspondant à été émis. Nos aspects de disponibilité sont limités à cinq types d'inserts :

1. Une autre possibilité serait de considérer que l'opérateur sélectionne le premier choix lorsque les deux sont possibles.

A	::=	$\{a_1 = E_1, \ldots, a_n = E_n\}$	*; équations mutuellement récursives*
E	::=	$E_1 \square E_2$	*; choix*
	|	$((F, G) \triangleright L); a_i$	*; ajoute la liste d'inserts L si l'instruction ; est filtrée par le filtre d'évènement F et la ; garde temporelle G et continue avec a_i*
F	::=	$Motif \mid F_1 \wedge F_2 \mid \neg F$	*; filtre d'évènement de base*
G	::=	$\{\ldots, t \odot k, \ldots\}$	*; garde temporelle* $\odot \in \{\leq, <, >, \geq\}$
L	::=	$\{I; \ldots; I\}$	*; liste d'inserts*
I	::=	*reset* (i, k)	*; programme l'exception i pour un ; déclenchement dans k unités de temps*
	|	*cancel* (i)	*; annule l'exception i*
	|	*start* (t)	*; initialise le minuteur t*
	|	*wait* (t, k)	*; attend jusqu'à ce que t vaille k*
	|	nop	*; opération vide*

La variable k représente un entier, i une interruption et t un minuteur.

Figure 4. *Syntaxe du langage d'aspects*

– *reset*(i, k) consiste à programmer la libération de toutes les ressources allouées et à terminer le service (***i.e.,*** la boucle de traitement courante) au bout de k unités de temps. Nous faisons l'hypothèse qu'un mécanisme transactionnel assure qu'un reset ramène le service dans un état initial sain sans compromettre sa sûreté. Nous modélisons cette action par une transition vers un état puits RESET. Cet état sera traduit par un relâchement de toutes les ressources allouées au service et la réinitialisation de celui-ci au début de sa boucle de traitement des requêtes.

– *cancel*(i) annule l'interruption i.

– *start*(t) initialise le minuteur t.

– *wait*(t, k) modélise une attente jusqu'à ce que $t = k$. Si $t \geq k$ alors l'instruction ne fait rien (*wait*$(t, k) \equiv nop$). Contrairement aux autres, cette action est effectuée avant d'exécuter l'instruction filtrée.

– nop permet de faire évoluer l'aspect sans effectuer d'action.

Nous imposons la restriction syntaxique interdisant de programmer et d'annuler un même exception (*reset*(i, k) et *cancel*(i)) dans la même liste d'inserts.

Nos aspects peuvent uniquement insérer des gardes ou des instructions temporelles qui ne modifient pas l'état du service. L'effet sémantique est uniquement d'interdire

certaines exécutions : soit elles sont coupées par un $reset$, soit leur comportement temporel est modifié par un $wait$. Ces restrictions autorisent à voir les aspects comme des propriétés temporelles et permettent de raisonner sur les programmes tissés.

Pour simplifier l'écriture des aspects, nous omettons de préciser la garde quand il s'agit de $true$ et d'utiliser la notation de liste lorsqu'il n'y a qu'un seul insert. Par exemple, $(true, \texttt{M1.prendre}) \triangleright \{\textit{reset}(i_1, 25)\}$ est notée $\texttt{M1.prendre} \triangleright \textit{reset}(i_1, 25)$.

5.2. *Exemples d'aspects*

Nous illustrons notre langage d'aspects par plusieurs exemples classiques d'utilisation : contrôler la durée d'utilisation des ressources, contrôler la fréquence d'utilisation de ressources, contrôler la durée suivant la fréquence et enfin imposer un ordre d'acquisition des ressources.

Contrôler la durée d'utilisation des ressources

On peut vouloir imposer les deux aspects de disponibilité suivants au service `S1` de la figure 3 :

– Un aspect A_1 qui assure que la ressource `M1` est relâchée avant 25 secondes ;

– Un aspect A_2 qui assure que la ressource `M2` est relâchée avant 35 secondes ;

Ces deux aspects sont spécifiés dans la syntaxe précédente comme suit :

$$A_1 = \left\{ \begin{array}{ll} a_1 = & \texttt{M1.prendre} \triangleright \textit{reset}(i_1, 25);\ a_2 \\ a_2 = & \texttt{M1.relacher} \triangleright \textit{cancel}(i_1);\ a_1 \end{array} \right\}$$

$$A_2 = \left\{ \begin{array}{ll} a_1 = & \texttt{M2.prendre} \triangleright \textit{reset}(i_2, 35);\ a_2 \\ a_2 = & \texttt{M2.relacher} \triangleright \textit{cancel}(i_2);\ a_1 \end{array} \right\}$$

Dès que l'évènement `M1.prendre` (resp. `M1.prendre`) est produit, une interruption (*i.e.,* un reset) est programmée pour se déclencher 25 secondes (resp. 35 secondes) plus tard. Si l'évènement `M1.relacher` (resp. `M2.relacher`) se produit avant, l'interruption est annulée.

Contrôler la fréquence d'utilisation de ressources

Il s'agit ici d'empêcher un service d'accaparer une ressource en la redemandant tout de suite. Cette politique s'applique à des ressources qui sont constamment demandées par plusieurs services. Elle impose de façon simple une répartition plus équitable des ressources.

Pour donner une idée, prenons le cas de deux services S_1 et S_2 qui nécessitent la ressource M pour fonctionner. Le service S_1 essaye de reprendre la ressource immédiatement après l'avoir relâchée alors que S_2 la redemande après 20 secondes. On

peut assurer une plus grande équité entre les deux services en imposant au service S_1 d'attendre au moins 20 secondes entre chaque prise de la ressource. L'aspect suivant impose une telle propriété :

$$\left\{ \begin{array}{lcl} a_1 & = & \texttt{M.prendre} \triangleright start(t);\ a_2 \\ a_2 & = & \texttt{M.prendre} \triangleright \{\mathit{wait}(t, 20); start(t)\};\ a_2 \end{array} \right\}$$

Dès que l'évènement `M.prendre` est produit, un minuteur t est lancé. On force une durée d'au moins 20 secondes avant de permettre l'évènement `M.prendre` suivant. L'attente (*wait*$(t, 20)$) est d'abord effectuée, puis `M.prendre`, enfin le minuteur est remis à zéro et relancé.

Contrôler la durée d'utilisation suivant la fréquence

En cas d'utilisation trop fréquente, au lieu de diminuer la fréquence, une autre possibilité est de limiter le temps d'utilisation de la ressource. Par exemple, on fixe une durée d'utilisation maximale de 10 secondes (*reset*$(i, 10)$) sauf si la ressource était déjà utilisée moins de 20 secondes avant ($t < 20$). Dans ce cas, on impose une durée d'utilisation maximale de 5 secondes (*reset*$(i, 5)$) seulement.

$$\left\{ \begin{array}{lcl} a_1 & = & \texttt{M.prendre} \triangleright \mathit{reset}(i, 10);\ a_2 \\ a_2 & = & \texttt{M.relacher} \triangleright \{\mathit{cancel}(i); \mathit{start}(t)\};\ a_1 \\ a_3 & = & (t < 20, \texttt{M.prendre}) \triangleright \mathit{reset}(i, 5);\ a_2 \\ & \square & (t \geq 20, \texttt{M.prendre}) \triangleright \mathit{reset}(i, 10);\ a_2 \end{array} \right\}$$

Imposer un ordre d'acquisition des ressources

Des aspects spécifiant des propriétés non temporelles peuvent également être décrits dans le langage. Par exemple, on peut vouloir imposer un ordre d'acquisition des ressources pour éviter les interblocages. Par exemple, l'aspect suivant interdit l'acquisition de la ressource `M1` si le service utilise déjà la ressource `M2`. Dans ce cas, le service est coupé immédiatement en utilisant *reset*$(i, 0)$. Il est possible d'utiliser `M1` et `M2` en les acquérant dans cet ordre ou de relâcher `M2` avant d'acquérir `M1`.

$$\left\{ \begin{array}{lcl} a_1 & = & \texttt{M2.prendre} \triangleright \{\};\ a_2 \\ a_2 & = & \texttt{M1.prendre} \triangleright \mathit{reset}(i, 0);\ a_3 \\ & \square & \texttt{M2.relacher} \triangleright \{\};\ a_1 \end{array} \right\}$$

Bien d'autres politiques de disponibilité peuvent se décrire dans le langage. Par exemple, il serait possible d'associer des priorités aux services et de les faire évoluer suivant le respect des contraintes ou, plus généralement, le comportement temporel des services.

6. Tissage

Le tissage d'un aspect repose sur l'opération de produit d'automates temporisés. Un service est représenté par un automate temporisé sur-approximant les traces d'exécution du code source. La sémantique des aspects est donnée également sous forme d'automate temporisé. L'automate représentant l'aspect décrit l'ensemble des traces temporelles permises. Le produit de ces deux automates fait l'intersection des traces d'exécutions et impose les contraintes spécifiées par l'aspect. En pratique, cela revient à couper certaines traces (interruptions et retour vers l'état initial) ou en allonger certaines (attentes dans certains états).

Dans un premier temps, nous décrivons comment les services sont abstraits en automates temporisés. Cette abstraction est le graphe de flot de contrôle du programme sans aucune contrainte temporelle. Dans un second temps, nous donnons la sémantique du langage d'aspect précédent en terme d'automates temporisés. L'étape suivante combine l'automate représentant le service et l'automate de l'aspect en un unique automate temporisé. Cette étape se résume à effectuer l'opération classique de produit. L'automate résultant décrit le comportement du service respectant la propriété spécifiée par l'aspect. La quatrième étape consiste à optimiser cet automate en prenant en compte les durées d'exécution des instructions du service. Cette étape repose sur une fonction de coût encadrant le temps d'exécution de chaque instruction par un intervalle $[\mathit{min}, \mathit{max}]$ qui peut être vu comme une contrainte temporelle interdisant les exécutions où l'instruction prendrait moins de *min* ou plus de *max*. Ces contraintes sont intégrées une nouvelle fois par produit. Elles permettent d'optimiser le tissage (*e.g.,* enlever des attentes ou des minuteurs inutiles).

6.1. *Abstraction des services en automates temporisés*

Nous utilisons une abstraction sur-approximant toutes les traces d'exécution du service et qui ne prend pas en compte les durées d'exécution des instructions. Ceci correspond à la plus grande sur-approximation du point de vue temporel. Un service est représenté par un automate temporisé qui peut se voir comme le graphe de flot de contrôle du service.

Nous décrivons cette abstraction à l'aide de la relation $\mathbf{next}_S(l_1, a, l_2)$ qui indique qu'une instruction de S permet de passer du point de programme l_1 à l_2 en effectuant l'action a. Cette relation est définie par :

$$\mathbf{next}_S(l_1, a, l_2) \;\; ssi \;\; l_1 : a \rightsquigarrow l_2 \in S \;\vee\; l_1 : a \rightsquigarrow l_2 \;;\; l \in S \;\vee\; l_1 : \neg a \rightsquigarrow l \;;\; l_2 \in S$$

Cette relation est une sur-approximation du flot de contrôle puisqu'elle ne prend pas en compte l'évaluation des tests.

Le service $S = (\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$ est *abstrait* par l'automate temporisé

$$S^\sharp = (\Sigma_{S^\sharp}, l_0, \emptyset, \mathcal{E}_{S^\sharp}, \longrightarrow_{S^\sharp}, I_{S^\sharp}) \qquad \text{où}$$

– $\Sigma_{S^\sharp}$, l'ensemble des états abstraits, est constitué de l'ensemble des points de programmes et d'un ensemble d'états intermédiaires. Formellement :

$$\Sigma_{S^\sharp} = \{l, l_a \mid \mathbf{next}_S(l, a, l')\}$$

– l'état initial abstrait est l'étiquette initiale l_0.

– l'ensemble des horloges est vide.

– l'ensemble des actions est celui de S dans lequel chaque action a est découpée en deux «instants» $deb\ a$ et $fin\ a$ et l'instruction spéciale *wait* :

$$\mathcal{E}_{S^\sharp} = \{deb\ a, fin\ a \mid a \in \mathcal{E}_S\} \cup \{\mathit{wait}\}$$

Cette découpe nous permettra de prendre en compte la durée d'exécution de a. L'instruction *wait* ne change pas la mémoire et nous sert à modéliser l'attente dans un état.

– la relation de transition $\longrightarrow_{S^\sharp}$ est définie par :

$$(l, ., deb\ a, ., l_a) \in \longrightarrow_{S^\sharp} \wedge (l_a, ., fin\ a, ., l') \in \longrightarrow_{S^\sharp} \quad ssi \quad \mathbf{next}_S(l, a, l')$$

Chaque action a passant d'un point de programme à un autre est représentée à l'aide de l'état intermédiaire l_a et de deux transitions correspondant aux deux instants $deb\ a$ et $fin\ a$ de l'action sans contrainte temporelle. Pour modéliser le temps pouvant s'écouler entre deux actions, nous ajoutons pour chaque état l une transition *wait* :

$$\forall l \in \Sigma_{S^\sharp}.\ (l, ., \mathit{wait}, ., l) \in \longrightarrow_{S^\sharp}$$

– la fonction $I_{S^\sharp}$ ne place aucune contrainte temporelle sur les états, c'est à dire $\forall l \in \Sigma_{S^\sharp}. I_{S^\sharp}(l) = \emptyset$.

L'absence de contrainte temporelle fait que l'automate représente toutes les durées d'exécution possibles pour chaque action. La figure 3 illustre cette étape en présentant l'abstraction du service `S1` en automate.

L'abstraction ainsi définie est sûre car les traces de l'automate incluent tous les chemins d'exécution du programme source. Formellement :

PROPRIÉTÉ **6.1** **[Sûreté]** *Tout service* $S = (\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$ *et abstraction associée* $S^\sharp = (\Sigma_{S^\sharp}, l_0, \mathcal{E}_{S^\sharp}, \longrightarrow_{S^\sharp}, I_{S^\sharp})$ *sont tels que*

$$(l_1, s_1) \xrightarrow{a}_S (l_2, s_2) \Rightarrow \exists l.\ l_1 \xrightarrow{.,deb\ a,.}_{S^\sharp} l \ \wedge\ l \xrightarrow{.,fin\ a,.}_{S^\sharp} l_2$$

6.2. *Sémantique des aspects sous forme d'automates temporisés*

La sémantique d'aspects est donnée par un automate temporisé. L'aspect spécifie une propriété temporelle et les traces de cet automate sont les traces temporelles autorisées par l'aspect.

La sémantique des aspects est donnée par des automates temporisés de la forme

$$A = (N_a, l_{a0}, H_a, \mathcal{E}_a, \longrightarrow_a, I_e) \qquad \text{où}$$

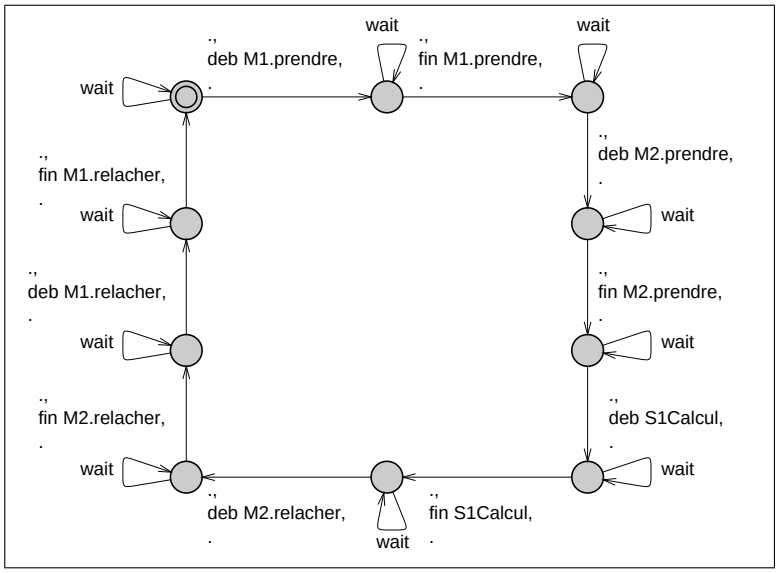

Figure 5. *Abstraction du service* `S1`

– l'ensemble d'états N_a est composé d'un état puits RESET et de couples (q, e) où q représente l'état de l'aspect et e l'environnement d'interruption courant.

– $l_{a0} = (a_0, \{\})$ est l'état initial ;

– H_a est l'ensemble des horloges (interruptions et minuteurs) utilisées dans l'aspect ;

– $\mathcal{E}_a = \mathcal{E}_{S^\sharp}$ définit les mêmes commandes que le service abstrait ;

– I_a la fonction associe à chaque état (q, e) un invariant imposant qu'aucune interruption valide (*i.e.,* définie dans e) n'ait lieu. Cette fonction se définit ainsi

$$I_a(q, e) = \{i \leq e(i) \mid \forall i.e(i) \neq \perp\}$$

La relation $\longrightarrow_a$ est définie sur la syntaxe de l'aspect. Nous utilisons la transition $(q, e) \xrightarrow{sinon}_a (q, e)$ qui précise que si aucune des transitions sortantes de l'état (q, e) ne s'applique alors l'aspect reste dans le même état. Cette notation est du sucre syntaxique qui peut être remplacé dans un deuxième temps par une collection de transitions (dénotant le complémentaire des transitions sortantes).

$$[a_0 = E_0] \quad = \quad (a_0, \{\}) \xrightarrow{sinon}_a (a_0, \{\}) \ \cup \ [E_0]^{(a_0,\{\})}$$

On produit l'automate correspondant à E_0 avec comme état initial $(a_0, \{\})$. Aucune interruption n'est activée et comme tout état, l'état initial a une transition en boucle $sinon$.

$$[E_1 \Box E_2]^{(q,e)} \quad = \quad [E_1]^{(q,e)} \cup [E_2]^{(q,e)}$$

Les transitions correspondant à un choix exclusif sont juste l'union des transitions pour les deux choix.

$$\begin{array}{rcl}[(F,G) \triangleright L; a_i]^{(q,e)} & = & [(F,G) \triangleright L]^{(q,e)}_{(a_i,e')} \ \cup \ [E_i]^{(a_i,e')} \ (\{a_i = E_i\} \in A) \\ & \cup & (a_i, e') \xrightarrow{sinon}_a (a_i, e') \ \cup \ interrupt(a_i, e')\end{array}$$

Le traitement d'une règle $(F,G) \triangleright L$ calcule un nouvel environnement d'interruption (règle suivante) et produit des transitions. L'automate correspondant à la suite de l'aspect est produit à partir de ce nouvel état. Pour tout nouvel état, on génère la transition $sinon$ et les transitions modélisant les interruptions contenue dans l'environnement courant (fonction $interrupt$).

$$\begin{array}{rcl}[(F,G) \triangleright L]^{(q_1,e_1)}_{(q_2,e_2)} & = & \{ (q_1, e_1) \xrightarrow{g, deb(a),.}_a (q', e_1) \\ & & \cup \ (q', e_1) \xrightarrow{conj(e), fin(a), r}_a (q_2, e_2) \\ & & \cup \ interrupt(q', e_1) \mid \ match(a, F) \\ & & \qquad \wedge \ ins(e_1, L) = (g_i, r, e_2) \\ & & \qquad \wedge \ (g = conj(e_1) \cup G \cup g_i)\} \\ & \cup & \{(q_1, e_1) \xrightarrow{t<k \wedge G \wedge conj(e), \textsf{wait},.}_a (q_1, e_1) \mid \forall \ \textsf{wait}(i,k) \in L\}\end{array}$$

Pour chaque action a filtrée par F, deux transitions ($deb\ a$ et $fin\ a$) sont produites à l'aide d'un nouvel état intermédiaire (q', e_1). Les transitions modélisant les interruptions possibles sont ajoutées sur cet état. La fonction ins analyse la liste d'inserts L pour calculer les gardes et les raz des transitions ainsi que le nouvel environnement d'interruptions. Si L contient des *wait*, les transitions d'attentes correspondantes sont ajoutées sur l'état de départ

Les fonctions intermédiaires sont définies ainsi :

– La fonction $interrupt$ prend un état (q,e) et retourne l'ensemble des transitions à ajouter pour modéliser les interruptions possibles dans cet état.

$$interrupt((q,e)) = \{(q,e) \xrightarrow{i \geq e(i),.,.}_a \ \textsc{Reset} \mid e(i) \neq \perp\}$$

En d'autres termes, il y a une interruption à chaque fois qu'un minuteur i dépasse sa valeur de déclenchement mémorisée dans l'environnement e.

– La fonction $match(a, F)$ retourne vraie si l'action a est filtrée par F.

– La fonction ins prend un environnement d'interruptions, une liste d'inserts et calcule la garde correspondant aux *wait* des inserts, le raz correspondant aux $reset$ et $start$ des inserts et, enfin, le nouvel environnement d'interruptions calculé en prenant en compte les $reset$ et $cancel$ des inserts.

$$ins(e, L) = (\{t \geq k \ \mid \ \textsf{wait}(t,k) \in L\}, \quad \{z | reset(z,k) \in L \vee start(z) \in L\}, \quad e')$$

avec $\begin{cases} e'(i) = \perp & \text{si } cancel(i) \in L \\ e'(i) = k & \text{si } reset(i,k) \in L \\ e'(i) = e(i) & \text{sinon} \end{cases}$

– La fonction $conj$ prend un environnement et retourne la garde correspondant au cas où il n'y a pas d'interruptions à déclencher : $conj(e) = \{i < e(i) \mid e(i) \neq \perp\}$

L'automate est produit en dépliant les définitions récursives des aspects. Le processus s'arrête car il y a un nombre fini de définitions récursives $a_i = \ldots$ et d'environnements possibles.

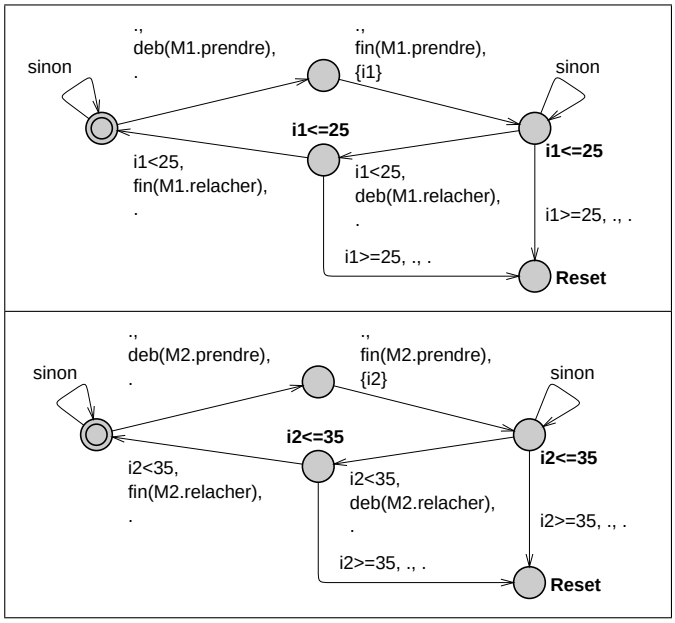

Figure 6. *Automates temporisés des aspects A_1 (en haut) et A_2 (en bas)*

La figure 6 montre les automates obtenus pour les aspects A_1 et A_2 définis précédemment. Intuitivement, le tissage consistera à initialiser une horloge quand le service prend la ressource puis à interrompre le service (en l'emmenant dans l'état RESET) lorsque le minuteur dépasse le quota de temps autorisé (*i.e.,* 25 sec. ou 35 sec.). L'état puits RESET ajouté par les aspects de disponibilité, sera interprété à la concrétisation en ajoutant des transitions relâchant les ressources suivi par une transition qui revient au début de la boucle de traitement des requêtes.

6.3. *Intégration d'un aspect à un service*

Le tissage à proprement parler est juste le produit (comme décrit en section 3.3) des automates représentant le service et l'aspect. L'impact sémantique de ce tissage est de restreindre le comportement du service en sélectionnant les traces autorisées par l'aspect (en coupant ou allongeant les traces interdites par l'aspect).

L'automate de l'aspect décrit les traces autorisées du service et est annoté temporellement (par des initialisations d'horloges, des gardes temporelles, ou des invariants d'états) pour gérer les inserts et les gardes temporelles. Le produit d'automates (*i.e.,* le tissage) effectue l'intersection des traces des automates du service et de l'aspect. Cela revient à modifier l'automate du service en ajoutant les annotations temporelles de l'aspect ou en coupant les traces interdites par l'aspect.

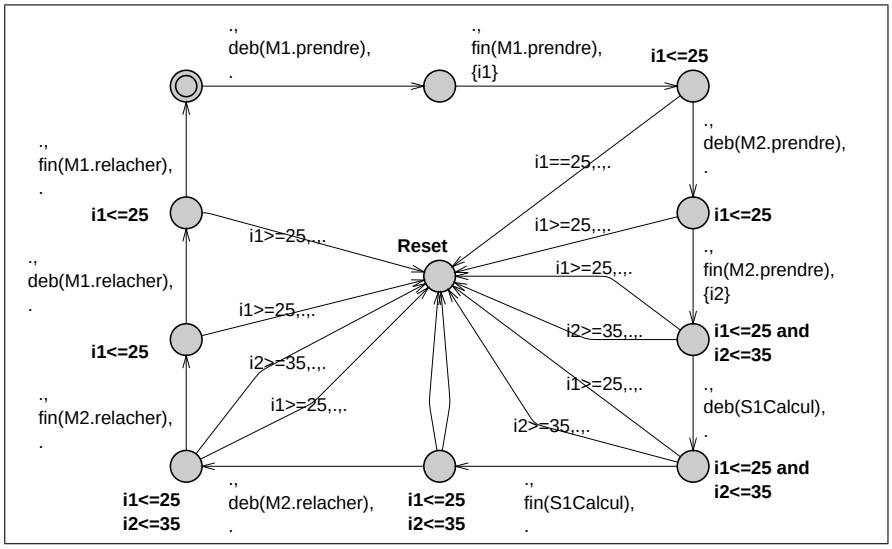

Figure 7. *Produit du service* `S1` *avec les aspects* A_1 *et* A_2

La figure 7 montre le produit de l'abstraction du service `S1` avec les aspects A_1 et A_2. Dans l'automate résultant (*i.e.,* le service tissé), une interruption est programmée après `M1.prendre` et une autre après `M2.prendre`. Si un `M1.relacher` (resp. `M2.relacher`) n'est pas effectué dans les 25 secondes (resp. 35 secondes), l'automate passe dans l'état $Reset$.

Comparé à un tissage plus classique à la AspectJ, le résultat final est similaire : du code (*advice*) est inséré à différents points de jointure du code source. Les approches sont en revanche totalement différentes. Dans AspectJ, la conception et le raisonnement sont principalement de nature syntaxique. Les aspects spécifient des ensembles précis de point de programmes et le code à insérer à ces points. Le programmeur raisonne sur l'impact des aspects en visualisant l'instrumentation attendue et le texte du programme tissé. Dans notre langage dédié, les restrictions sur les inserts permettent de voir l'aspect comme une propriété sur les traces d'exécutions. L'aspect spécifie un ensemble de traces autorisées qui peut être imposé au service par produit. L'automate résultant peut être ensuite optimisé et traduit en code insérant les bonnes instructions aux bons endroits. L'avantage principal de cette approche est évidemment de faire apparaître explicitement l'impact sémantique du tissage. Dans un langage général ne

faisant aucune restriction sur les inserts, il est impossible en général de caractériser précisément l'effet des aspects sur la sémantique du programme.

6.4. *Prise en compte des durées d'exécution et optimisations*

La prise en compte de durées d'exécution repose sur l'utilisation d'une fonction de coût f_{cout} qui prend une instruction (ou un bloc d'instructions) I et retourne un intervalle de temps $[\text{BCET}(I), \text{WCET}(I)]$ où $\text{BCET}(I)$ (resp. $\text{WCET}(I)$) représente la borne inférieure (resp. supérieure) du temps d'exécution de I. Notons que l'on peut toujours concevoir une telle fonction de coût puisque l'approximation triviale $f_{cout}(I) = [0, +\infty]$ est toujours valide. Cependant une fonction de coût plus précise (voir par ex. (Puschner *et al.*, 1989, Li *et al.*, 2005)) peut éviter au tisseur d'insérer des tests. Par exemple, si f_{cout} permet de déterminer statiquement qu'un service relâche toujours une ressource dans les temps voulus par un aspect de disponibilité, aucun test ou instrumentation du service ne sera nécessaire.

Par la suite, nous faisons l'hypothèse que nous possédons une telle fonction de coût et qu'elle rend les résultats suivants pour les actions du service `S1` :

$$\begin{array}{lcllcl} f_{cout}(\texttt{S1calcul}) & = & [2,10] & & & \\ f_{cout}(\texttt{M1.Prendre}()) & = & [0,+\infty] & f_{cout}(\texttt{M2.Prendre}()) & = & [0,+\infty] \\ f_{cout}(\texttt{M1.relacher}()) & = & [0,0] & f_{cout}(\texttt{M2.relacher}()) & = & [0,0] \end{array}$$

La prise en compte de la fonction de coût est effectuée par un produit d'automates temporisés avec l'automate $C = (N_c, c_0, \{k\}, \mathcal{E}_{S\sharp}, \longrightarrow_c, I_c)$ où :

– Pour toute action a telle que $f_{cout}(a) = [min, max]$ on a les transitions

$$c_0 \xrightarrow{.,deb(a),\{k\}}_c q \quad \text{et} \quad q \xrightarrow{k \geq min, fin(a),.}_c c_0 \quad \text{avec } q \text{ un nouvel état}$$

– L'invariant d'état spécifie qu'on ne peut y rester que le maximum du coût de l'action qui nous y a conduit. C'est à dire :

$$I_c(q) = \{k \leq max \mid q \xrightarrow{k \geq min, fin(a),.}_c c_0\}$$

L'horloge k est remise à zéro au début de l'action a. On reste dans l'état intermédiaire au plus tard jusqu'à ce que $k = max$ et on en sort à la fin de l'action au plus tôt quand $k \geq min$.

Une autre information à prendre en compte est que le séquencement est instantané (les ; sont exécutés immédiatement). Ceci est décrit par un produit avec l'automate temporisé à deux états $E = (\{e_0, e_1\}, e_0, \{seq\}, \mathcal{E}_{S\sharp}, \longrightarrow_e, I_e)$ où :

– Chaque début d'action fait passer à l'état e_1 et chaque fin d'action, fait passer à l'état e_0 en remettant à zéro l'horloge dédiée seq. Intuitivement, l'état e_0 représente

les points de jonction des actions (qui ne prennent pas de temps) alors que l'état e_1 représente le corps de l'action (où du temps peut s'écouler).

$$\longrightarrow_e = \left\{ \begin{array}{l} e_0 \xrightarrow{.,deb(a),.}_e e_1 \\ e_1 \xrightarrow{.,fin(a),\{seq\}}_e e_0 \end{array} \middle| deb(a) \in \mathcal{E}_{S^\sharp} \wedge fin(a) \in \mathcal{E}_{S^\sharp} \right\}.$$

– L'invariant de l'état e_0 (représentant le séquencement des actions) assure que le temps ne peut s'écouler dans cet état. Aucune contrainte n'est imposée sur l'état e_1.

$$I_e(e_0) = \{seq \leq 0\} \qquad I_e(e_1) = \emptyset$$

L'automate temporisé obtenu après le produit avec les deux automates précédents est plus précis. Le produit a restreint l'ensemble des traces possibles et a intégré les informations de durées d'exécution disponibles. L'automate peut maintenant être analysé pour enlever les gardes, horloges et invariants d'états inutiles ainsi que les états non atteignables. Cette optimisation permet de diminuer le surcoût introduit par l'aspect. Elle est faite automatiquement dans des outils tels qu'UPPAAL.

La figure 8 montre l'automate obtenu pour le service tissé `S1` de la figure 7 après produit avec les automates C et E et optimisation.

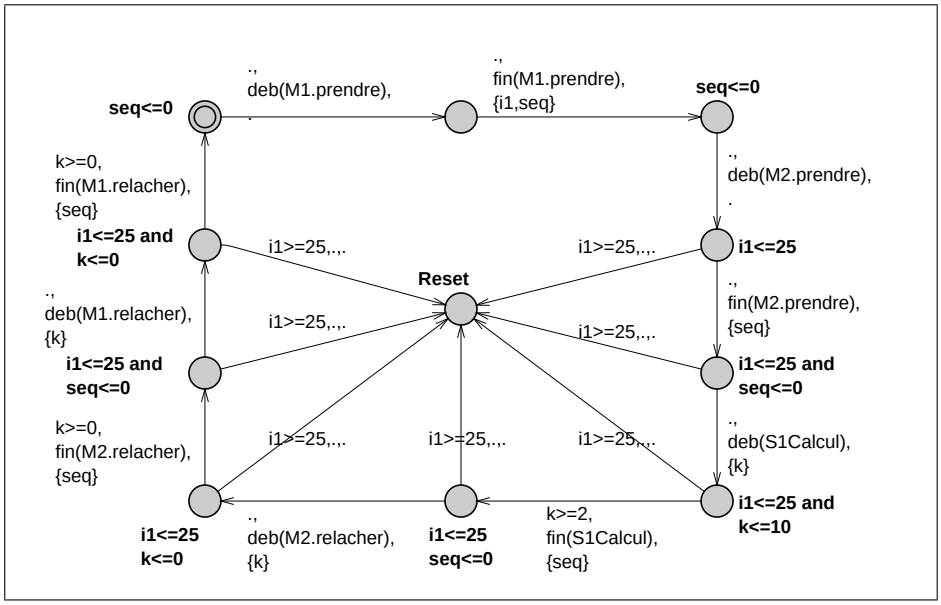

Figure 8. *Automate temporisé du service* `S1` *après tissage et optimisation*

L'aspect A_2 interdit au service de retenir la ressource `M2` plus de 35 sec. Le tissage de cet aspect n'a aucun impact sur le code car l'automate du service indique clairement

que S1calcul (et donc l'utilisation de M2) dure au plus 10 sec. Cette information, donnée initialement par la fonction de coût et intégrée dans l'automate représentant le service, permet d'optimiser le tissage et d'éviter d'introduire un compteur inutile.

6.5. *Vérification*

L'automate obtenu précédemment est une représentation formelle du service tissé. A cette étape, on peut vouloir vérifier que les services tissés garantissent des propriétés de disponibilité. En effet, les aspects ne définissent pas directement une propriété de disponibilité. Ce sont plutôt un ensemble de contraintes temporelles censé impliquer une propriété de disponibilité de plus haut niveau.

Ce type de propriété peut être vérifié par *model-checking* sur le système obtenu après tissage. Cette étape permet également de vérifier que les aspects ne sont pas contradictoires. En effet, certains aspects peuvent entrer en conflit (par ex. un aspect qui prévient les interblocages en augmentant le temps d'allocation d'une ressource peut rentrer en conflit avec un aspect temporel limitant ce même temps d'allocation).

Nous avons utilisé UPPAAL qui permet de représenter les services, d'exécuter (et de mettre au point) les modélisations, pour vérifier des propriétés exprimées en LTL. Nous avons représenté et analysé l'exemple pris dans cet article avec UPPAAL. Nous avons vérifié que le système tissé respectait les propriétés suivantes :

– *le système est bien temporisé et n'arrive pas à des situations d'interblocage*. Dans cet article, nous n'avons pas prévenu les interblocages qui peuvent survenir entre S1 et S2. Néanmoins, ceux-ci sont traités par l'aspect interrompant S2 au bout de 35 secondes.

– *le service* S1 *accomplit un tour de boucle en moins de* 45 *secondes*. Cette propriété s'exprime en introduisant un nouveau compteur dispo mis à zéro au début de la boucle de traitement et à vérifier qu'il n'est, dans aucun état, supérieur à 45 secondes. Cette propriété a été garantie par tissage. En effet, après tissage le service S2 relâche la ressource M2 après au plus 35 sec. et comme S1calcul prend au plus 10 secondes, S1 aura terminé en au plus 45 secondes. Ceci signifie aussi que le service S1 a toujours accès aux ressources et qu'il ne peut pas y avoir de dénis d'accès aux ressources dans le système (contrairement à avant le tissage).

L'analyse de ces propriétés est très rapide (moins d'une seconde). Nous supposons qu'UPPAAL pourra traiter des systèmes de grande taille car il a déjà servi pour analyser des protocoles complexes.

7. Concrétisation

La génération du code à partir d'un automate non temporisé est simple ; nous l'avons formalisée dans (Fradet *et al.*, 2004). La nouveauté apportée par les automates

temporisés réside dans les instructions temporelles (initialisation d'horloge, invariants d'états et gardes temporelles).

Afin de prendre en compte les actions introduites dans le tissage, nous étendons notre langage source avec des commandes temporelles et des comparaisons de minuteurs. Le langage est étendu avec les commandes suivantes :

$$c ::= \textit{start}(t) \mid wait(t,k) \mid \textit{reset}(i,k) \mid \textit{cancel}(i) \mid \ldots$$

où t dénote un identificateur de minuteur, i un identificateur d'interruption et k un entier. Les tests sont étendus par des comparaisons de minuteurs :

$$g ::= t \odot k \mid \ldots \text{ avec } \odot \in \{<, >, \leq, \ldots\}$$

La commande *start*(t) déclenche (initialise à zéro) un minuteur t qui pourra être comparé à des entiers dans des tests. Les minuteurs sont également utilisés pour retarder l'exécution à l'aide de la commande *wait*(t,k) qui attend tant que $t < k$.

La commande *reset*(i,k) programme une interruption i pour se déclencher dans k unités de temps. L'instruction *cancel*(i) déprogramme l'interruption i.

Nous décrivons informellement comment un automate temporisé est traduit dans ce langage enrichi. Tout d'abord, les informations temporelles introduites par les automates C et E sont enlevées car elles ne correspondent à des propriétés non fonctionnelles et non à du code de programmes. La concrétisation suit les règles suivantes :

– Les couples de transitions

$$(q_1 \xrightarrow{.,deb(c),.} q',\ q' \xrightarrow{.,fin(c),.} q_2)$$

correspondent à une commande c et sont traduites par l'instruction $l_{q1} : A \rightsquigarrow l_{q2}$.

– Les quadruplets de transitions

$$(q_1 \xrightarrow{.,deb(g),.} q',\ q' \xrightarrow{.,fin(g),.} q_2,\ q_1 \xrightarrow{.,deb(\neg g),.} q'',\ q'' \xrightarrow{.,fin(\neg g),.} q_3)$$

correspondent à un test g et sont traduites par l'instruction $l_{q1} : g \rightsquigarrow l_{q2}\ ;\ l_{q3}$.

– Une boucle $q \xrightarrow{t<k,\textit{wait},.} q$ est traduite par l'insertion d'une commande *wait*(t,k) avant le point de programme correspondant (l_q).

– La remise à zéro d'un minuteur t sur la transition $q \xrightarrow{g,fin(a),\{t\}} q'$ est traduite par l'insertion d'une commande *start*(t) après le point de programme correspondant à q' ($l_{q'}$).

– La gestion des interruptions est effectuée en insérant la commande *reset*(i,k) à l'initialisation de l'interruption i (*i.e.,* i dans un raz) et en insérant la commande $cancel(i)$ au premier état q où il n'y a plus de transition $q \xrightarrow{i \geq k,.,.,.}$ RESET.

La figure 9 montre le code source du service `S1` obtenu après concrétisation de l'automate de la figure 8. Après l'instruction `M1.prendre()`, une nouvelle interruption `i` est initialisée pour s'exécuter après 25 secondes. Dans le cas où le service prend moins de 25 sec pour finir son traitement, la ressource `M1` est relâchée (`M1.relacher()`) et l'interruption est déprogrammée (*cancel* (i)).

$$\texttt{S1} = \left\{ \begin{array}{lcll} l_0 & : & \texttt{getUser}() & \leadsto\ l_1 \\ l_1 & : & \texttt{M1.prendre}() & \leadsto\ l'_1 \\ l'_1 & : & \textit{reset}(i, 25) & \leadsto\ l_2 \\ l_2 & : & \texttt{M2.prendre}() & \leadsto\ l_3 \\ l_3 & : & \texttt{S1.calcul}() & \leadsto\ l_4 \\ l_4 & : & \texttt{M2.relacher}() & \leadsto\ l_5 \\ l_5 & : & \texttt{M1.relacher}() & \leadsto\ l'_6 \\ l'_6 & : & \textit{cancel}(i) & \leadsto\ l_6 \\ l_6 & : & \texttt{endUser}() & \leadsto\ l_0 \end{array} \right\}$$

Figure 9. *Code du service* `S1` *après tissage*

8. Conclusion

Nous avons présenté un cadre formel permettant d'imposer des propriétés de disponibilité sur un système de services partageant des ressources. Notre contribution tient à la fois dans la proposition d'un langage d'aspect dédié à la prévention des dénis de service et à l'approche qui présente les aspects comme des propriétés temporelles et le tissage comme un produit d'automates. Nous n'avons pas mis en œuvre cette technique mais nous pensons néanmoins qu'elle est réaliste. Les services devraient rester de taille raisonnable puisque le code sans rapport avec la gestion des ressources n'a pas à être pris en compte. D'autre part, on peut maîtriser les coûts des analyses (flot de contrôle, temps d'exécution) en adaptant leurs approximations (*i.e.,* leur précision). Enfin, si le tissage par un produit d'automates naïf peut dans certains cas entraîner une explosion de code, il est facile d'éviter ce problème en instrumentant le code au lieu de le dupliquer (Colcombet *et al.*, 2000).

Cette étude fait partie d'une série de travaux considérant les aspects comme des propriétés sur les traces d'exécution. L'approche commune est de traduire les programmes et les aspects en des automates (plus ou moins expressifs) et d'utiliser l'opération de produit pour le tissage. En plus du cadre, ces différents travaux partagent le souci de contrôler l'impact sémantique du tissage afin de pouvoir raisonner (analyser, vérifier, prouver) sur les programmes orientés aspects.

– Dans (Colcombet *et al.*, 2000), nous avons proposé une technique pour imposer des politiques de sécurité exprimées par des automates. L'approche permet d'imposer par tissage et au téléchargement des propriétés de sécurité à des applets. Le tissage permet d'assurer que l'applet sera stoppée au cas où elle essaye de violer la propriété.

– Dans (Fradet *et al.*, 2004), nous avons utilisé les aspects pour spécifier et imposer des politiques d'ordonnancement à des réseaux de processus communicants. Un aspect d'ordonnancement (décrit par un automate) sélectionne un ensemble d'exécutions autorisées parmi tous les entrelacements possibles. Cette étape permet de fusionner les réseaux en un unique programme séquentiel plus efficace.

– Dans cet article, nous avons généralisé le cadre aux automates temporisés. Nous pouvons à la fois interdire des exécutions ou modifier leur comportement temporel. Cette généralisation nous permet d'exprimer et d'imposer des propriétés sur les temps d'exécution. Le langage d'aspects est assez expressif pour spécifier un grand nombre de politiques de disponibilité.

Yu et Gligor (Yu *et al.*, 1990) ont proposé une méthode pour vérifier qu'un allocateur de ressources reste disponible. Notre cadre peut être vu comme une généralisation de leur travail à des politiques en temps borné. De plus, l'utilisation de la programmation par aspects permet une meilleure séparation des problèmes et, surtout, apporte une instrumentation automatique des programmes par tissage. Millen (Millen, 1994) propose un modèle de moniteur global pour la disponibilité qui repose sur une *Trusted Computing Base*. Notre exemple montre que des politiques locales peuvent aussi être utilisés pour assurer la disponibilité. Les politiques locales de disponibilité proposées ont l'avantage d'être facilement compréhensibles et tissables. Cuppens et Saurel (Cuppens *et al.*, 1999) ont proposé un cadre logique pour exprimer et vérifier des politiques de disponibilité. Le modèle est précis et expressif mais il n'a pas été conçu dans le but de faire respecter ces politiques mais de les vérifier/prouver a posteriori. J-Seal2 (Binder *et al.*, 2001) propose un mécanisme simple et global pour assurer la disponibilité du processeur et de la mémoire. Ils décrivent l'implémentation en terme d'instrumentation des services mais cette technique n'est pas générique et ne s'applique pas à d'autres types de ressources (*e.g.,* les ressources en accès exclusif).

Nous travaillons actuellement à finaliser la formalisation de la concrétisation et la preuve de correction associée. Un autre sujet, que nous n'avons fait qu'effleurer ici, est celui de la prévention des interblocages. Limiter la durée de possession des ressources ou imposer un ordre d'acquisition (*c.f.* section 5.2) permettent de prévenir les interblocages. Pourtant, ces techniques ne sont pas idéales : le système peut se retrouver bloqué à attendre qu'une limite de temps soit atteinte et imposer un ordre d'acquisition peut résulter dans l'interruption systématique de services qui ne pourront plus remplir leur tâche. Une autre solution serait de transformer les services (*e.g.,* avancer l'allocation de certaines ressources) afin qu'ils respectent l'ordre d'acquisition spécifié par l'aspect. Nous n'avons pas encore formalisé cette transformation mais il semble clair que des techniques d'analyse statique seraient utiles pour satisfaire l'ordre d'acquisition tout en retenant le moins longtemps possible les ressources.

9. Bibliographie

Aksit M., Wakita K., Bosch J., Bergmans L., « Abstracting Object Interactions Using Composition Filters », vol. 791, p. 152-184, 1994.

Alur R., « Timed Automata », *Computer Aided Verification*, p. 8-22, 1999.

Alur R., Dill D. L., « A theory of timed automata », *Theoretical Computer Science*, vol. 126, n° 2, p. 183-235, 1994.

Banga G., Druschel P., Mogul J. C., « Resource containers : a new facility for resource management in server systems », *OSDI '99 : Proceedings of the third symposium on Operating systems design and implementation*, USENIX Association, p. 45-58, 1999.

Bengtsson J., Yi W., « Timed Automata : Semantics, Algorithms and Tools », *in* , W. Reisig, , G. Rozenberg (eds), *Concurrency and Petri Nets*, LNCS vol 3098, Springer–Verlag, 2004.

Binder W., Hulaas J. G., Villaz A., « Portable resource control in Java », *OOPSLA '01 : Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, ACM Press, p. 139-155, 2001.

Colcombet T., Fradet P., « Enforcing Trace Properties by Program Transformation », *Symposium on Principles of Programming Languages (POPL'00)*, p. 54-66, 2000.

Cuppens F., Saurel C., « Towards a formalization of availability and denial of service », *Information Systems Technology Panel Symposium on Protecting Nato Information Systems in the 21st century*, 1999.

Douence R., Fradet P., Südholt M., « A framework for the detection and resolution of aspect interactions », *Proc. of Conference on Generative Programming and Component Engineering (GPCE'02)*, Springer-Verlag, LNCS Vol. 2487, 2002.

Fradet P., Hong Tuan Ha, S., « Network Fusion », *Programming Languages and Systems : Second Asian Symposium, (APLAS'04)*, LNCS vol 3302, 2004.

Fradet P., Hong Tuan Ha S., « Systèmes de gestion de ressources et aspects de disponibilité », *2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, Lille, France, September, 2005. (In French).

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., « An Overview of AspectJ », *Lecture Notes in Computer Science*, vol. 2072, p. 327-355, 2001.

Laprie J.-C. et al., *Dependability : Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems, Springer-Verlag, 1992.

Larsen K. G., Pettersson P., Yi W., « UPPAAL in a Nutshell », *International Journal on Software Tools for Technology Transfer*, vol. 1, n° 1-2, p. 134-152, 1997.

Li X., Mitra T., Roychoudhury A., « Modeling control speculation for timing analysis », *Real-Time Syst.*, vol. 29, n° 1, p. 27-58, 2005.

Millen J. K., « A Resource Allocation Model for Denial of Service Protection », *Journal of Computer Security*, 1994.

Puschner P., Koza C., « Calculating the maximum, execution time of real-time programs », *Real-Time Syst.*, vol. 1, n° 2, p. 159-176, 1989.

Rushby J., « Critical System Properties : Survey and Taxonomy », *Reliability Engineering and Systems Safety*, vol. 43, n° 2, p. 189-219, 1994. Research report CSL-93-01.

Schneider F. B., « Enforceable security policies », *ACM Transactions on Information and System Security*, vol. 3, n° 1, p. 1-50, February, 2000.

T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, « Symbolic Model Checking for Real-Time Systems », *7th. Symposium of Logics in Computer Science*, p. 394-406, 1992.

Yu C.-F., Gligor V. D., « A Specification and Verification Method for Preventing Denial of Service », *IEEE Trans. Softw. Eng.*, vol. 16, n° 6, p. 581-592, 1990.