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 considérons 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 nous concentrons sur les dénis de service liés à la gestion de ressources (famines, interblocages). Nos aspects spécifient des limites de temps ou un ordre dans l'allocation des ressources. Ils peuvent être vus comme la spécification d'une politique de disponibilité. L'approche repose sur les automates temporisés pour modéliser les services et les aspects. Ceci permet de voir le tissage comme un produit d'automates et d'utiliser des outils automatiques pour vérifier que les aspects imposent les propriétés de disponibilité attendues.*

ABSTRACT. *In this paper, we consider resource management in isolation (separation of concerns) and the prevention of denial of service (i.e. availability) as aspects. We concentrate on denials of service caused by resource management (starvations, deadlocks). Our aspects specify time limits or orderings in the allocation of resources. They can be seen as the specification of an availability policy. The approach relies on timed automata to specify services and aspects. It allows us to implement weaving as an automata product and to use model-checking tools to verify that aspects enforce the required availability properties.*

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

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

1. Introduction

Avec la confidentialité et l'intégrité, la disponibilité est l'une des trois familles de propriétés de sécurité. Les propriétés de disponibilité assurent 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 décrivons le système de gestion de ressources de façon séparée comme la spécification des ressources et une collection d'aspects de disponibilité. La spécification des ressources sert à générer le code implémentant les ressources (*e.g.*, la gestion des files d'attentes, etc.). Les aspects spécifient des limites de temps ou un ordre dans l'allocation des ressources. Ils peuvent être vus comme la spécification d'une politique de disponibilité prévenant les famines ou/et les interblocages. Les aspects sont tissés sur le code des services accédant aux ressources. Nous utilisons les automates temporisés (Alur *et al.,* 1994, T.A. Henzinger *et al.,* 1992) pour modéliser les services et les aspects. Ceci permet de voir le tissage comme un produit d'automates et d'utiliser des outils automatiques (par ex. UPPAAL (Larsen *et al.,* 1997, Bengtsson *et al.,* 2004)) pour vérifier que les aspects imposent les propriétés de disponibilité attendues.

L'article est organisé comme suit. La section 2 présente rapidement le cadre de l'étude, notre approche et l'exemple utilisé dans toute la suite. La section 3 présente la façon dont les ressources sont spécifiées et deux types de ressources classiques. La section 4 (resp. section 5) présente les services (resp. les aspects) comme des automates temporisés. Nous décrivons le tissage des aspects en section 6 et la vérification des propriétés de disponibilité en section 7. Nous discutons rapidement des travaux connexes et des extensions possibles en conclusion.

Ce travail est effectué dans le cadre de l'ACI Sécurité Informatique DISPO. L'objectif de cette ACI (Action Concertée Incitative) est de concevoir des méthodes de construction et de validation de composants logiciels disponibles. Le travail présenté ici n'est pas complètement finalisé. L'article se contente de présenter les principales idées sur un exemple.

2. Vue d'ensemble

Nous commençons par présenter le cadre de notre étude sur la gestion des ressources et la disponibilité. Nous présentons ensuite les grandes lignes de notre approche et l'exemple nous servant à illustrer les principales étapes.

2.1. *Cadre général*

Nous distinguons trois acteurs structurés en couches : les *utilisateurs*, les *services* et les *ressources* (Figure 1).

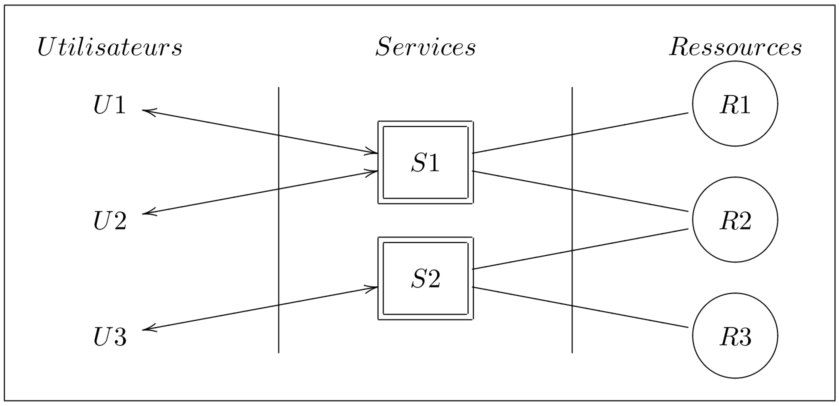


Figure 1. *Modélisation en 3 couches*

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, un fichier, 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.

2.2. *Approche*

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 de 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*). 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 (Section 3). 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. On considère des aspects temporels qui limitent le temps d'allocation d'une ressource à un service, et des aspects d'ordonnancement qui permettent de prévenir ou traiter les interblocages (Section 5).

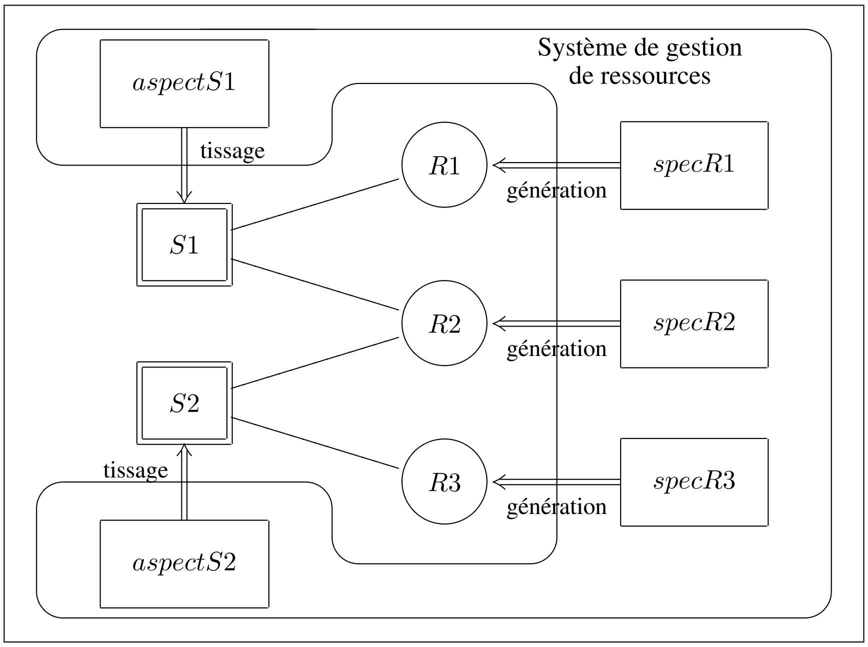


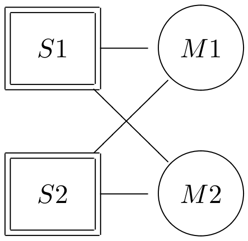
Figure 2. *Vision globale du système avec gestion des ressources séparée*

La figure 2 récapitule la structure de notre système de gestion de ressources. Celui-ci est constitué des aspects de disponibilité et de la spécification des différentes ressources. Les contraintes sur les services sont composées 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 des autres. Les aspects globaux sont plus expressifs et plus précis. Par contre, ils ne sont pas facilement tissables sur le code des 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 sont suffisamment expressifs pour prévenir les dénis de service et dont l'implémentation peut être optimisée en les tissant statiquement (Section 6).

Le formalisme sous-jacent à toute l'approche est celui des automates temporisés. Les ressources, les programmes et les aspects sont décrits par de tels automates. Le tissage repose sur l'opération de produit d'automates temporisés. Ce formalisme nous permet également de disposer d'outils de vérification et de montrer automatiquement que le système complet (ressources + services + aspects) possède les propriétés de disponibilité attendues (Section 7).

2.3. *Exemple de système*

Réseau



Services

$$
\mathtt{S1} = \left\{ \begin{array}{ll} \mathtt{l}_1 : & \mathtt{M1.prendre(); M2.prendre();} \\ & \mathtt{S1Calcul;} \\ & \mathtt{M2.relacher(); M1.relacher();} \\ & \mathtt{jmp\ l}_1\mathtt{;} \end{array} \right\}
$$

$$
\mathtt{S2} = \left\{ \begin{array}{ll} \mathtt{l}_2 : & \mathtt{M2.prendre();} \\ & \mathtt{S2Calcul1;} \\ & \mathtt{if\ G\ then\ M1.prendre();} \\ & \qquad\qquad \mathtt{S2Calcul2;} \\ & \qquad\qquad \mathtt{M1.relacher();} \\ & \mathtt{M2.relacher();} \\ & \mathtt{jmp\ l}_2\mathtt{;} \end{array} \right\}
$$

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

Cet article présente les principales étapes (spécification, tissage, vérification) à l'aide de l'exemple de la figure 3. 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 de type `mutex`, type décrit plus en détail dans la section 3.

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 ressources

La spécification séparée de la gestion des ressources consiste tout d'abord à définir les ressources par :

– une *interface* qui énumère les opérations d'accès à la ressource ;

– un *automate* qui spécifie le comportement de la ressource, en particulier :

- l'évolution de son état quand les services utilisent les opérations d'accès,
- la gestion de la file d'attente des requêtes de services en attente.

Nous décrivons ci-dessous deux exemples de ressources. Nous avons choisi des types très communs (`mutex` et `partageable`) et des spécifications simples et directes. Bien d'autres spécifications ainsi et/ou types de ressources existent.

3.1. *Ressource de type* `mutex`

Une ressource utilisée par un seul utilisateur à la fois est dite de type `mutex`. Elle est utilisée, par exemple, pour avoir une section critique et protéger la lecture ou l'écriture sur des données partagées. Elle est définie par :

– une interface :

- `reqPrendre()` pour demander d'entrer en section critique ;
- `traitPrendre()` quand la ressource accorde l'entrée en section critique ;
- `relacher()` pour ressortir de la section critique.

– un comportement : Les `reqPrendre()` sont accordés selon un ordre FIFO. Les `traitPrendre()` sont effectués dès que la ressource est libre. Les `relacher()` sont effectués sans délai.

Le comportement d'une ressource `mutex` est spécifié précisément par l'automate de la figure 4. Pour spécifier les ressources, comme pour spécifier les aspects de disponibilité (voir Section 5), nous utilisons les automates temporisés d'UPPAAL (Larsen *et al.,* 1997, Bengtsson *et al.,* 2004). Si l'automate est déterministe et complet (tous les cas sont couverts), il est facile de générer du code exécutable d'une telle spécification. Cette approche est plus déclarative qu'un codage direct dans un langage de programmation. Elle reste également dans le même formalisme que la spécification des aspects.

Nous nous contentons de décrire la syntaxe UPPAAL au fil de nos exemples[1]. Dans

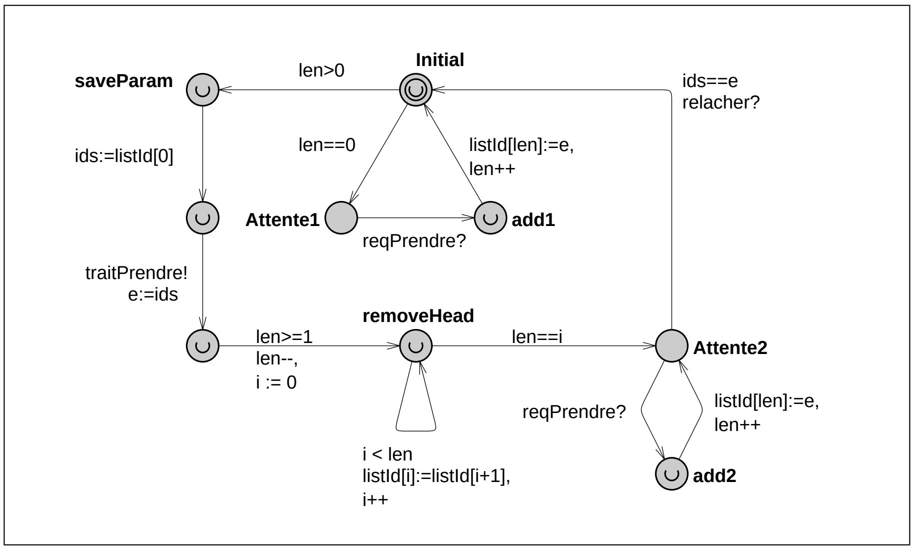


Figure 4. *Spécification de la ressource* `mutex`

la figure 4, l'automate manipule la liste `listld` pour implanter la file FIFO des requêtes. La variable interne `len` a pour valeur le nombre de requêtes stockées dans la liste `listld`. La variable partagée `e` dénote l'identité du service effectuant la requête courante. Dans l'état `Initial`, si la file est vide, il faut attendre une requête `reqPrendre()`, sinon la requête `traitPrendre!` du service correspondant au premier `reqPrendre?` de la file (`listld[0]`) est effectuée (l'identité du service est passé dans `e`). La file FIFO est mise à jour (dans l'état `RemoveHead`) puis de nouvelles requêtes `reqPrendre?` peuvent être enregistrées dans la FIFO. La requête `relacher` correspondant aux requêtes `reqPrendre()` et `traitPrendre()` précédentes est effectuée (encore une fois, la variable partagée `e` est utilisée pour passer l'identité du service entre ressource et service). Les états étiquetés `U` sont dits "urgents" ; il n'est pas possible d'attendre dans ces états. Ils seront traduits par l'opérateur de séquen-

1. Le lecteur pourra trouver une abondante documentation sur UPPAAL (manuel, didacticiel, articles) sur le site `http ://www.uppaal.com/`

cement " ;". Les autres états correspondent à (et seront traduits par) l'attente d'une requête faite par un service.

Le comportement décrit par l'automate est complet et déterministe. Il peut se traduire directement en un code séquentiel. Nous indiquons ci-dessous uniquement les premières instructions.

```
Initial : if len==0 then wait(reqPrendre);
                         listld[len]:=e; len++;
          else ids:=listld[0]; traitPrendre; e:=ids;
               ...
          jmp Initial;
```

Avec ce type de ressource, des problèmes de disponibilité peuvent apparaître :

– si un processus prend une ressource `mutex` et ne la relâche pas,

– plusieurs services accédant à plusieurs ressources `mutex` peuvent donner lieu à un interblocage.

3.2. *Ressource de type* `partageable`

Un raffinement de la ressource `mutex` est la ressource `partageable` qui comporte un ensemble de k parts pouvant être allouées à différents services. Ce type de ressource se rencontre fréquemment dans la littérature (Leiwo *et al.,* 1997) et correspond à de nombreuses ressources réelles (par exemple, l'ensemble des connections réseau). L'interface et le comportement d'une ressource partageable sont :

– interface :

- `reqPrendre(i)` pour demander i parts de la ressource ;
- `traitPrendre(i)` quand la ressource accorde les i parts ;
- `relacher(i)` pour libérer i parts.

– comportement : Les `reqPrendre(i)` sont accordés selon un ordre FIFO. Les `traitPrendre(i)` sont effectués dès que i parts sont libres. Les `relacher(i)` sont effectués sans délai.

Nous ne donnons pas ici l'automate de comportement qui est similaire au précédent. Une ressource partageable pose les mêmes problèmes de disponibilité qu'une ressource `mutex`. En fait, une ressource partageable en k peut être vue comme k ressources `mutex`. De ce fait, des interblocages peuvent apparaître avec une seule ressource partageable.

D'autres types de ressources ou des spécifications plus sophistiquées peuvent se décrire dans ce cadre. Par exemple, il serait possible de prendre en compte une notion de priorité associée aux services. Le traitement des requêtes se ferait alors en fonction de l'ordre d'arrivée *et* de la priorité du service.

4. Abstraction des services en automates temporisés

Les propriétés ou aspects que nous voulons imposer aux services ont une composante temporelle. Afin de tisser ce type d'aspect le plus efficacement possible, il est donc important que le modèle de programmes intègre des informations temporelles (temps de calcul). Nous utilisons les automates temporisés d'UPPAAL comme modèle abstrait de programmes.

Les services sont transformés en automates temporisés en deux étapes. La première étape consiste à abstraire le service en un automate classique. Cette étape est très classique et consiste à construire le graphe de flot de contrôle du programme. Les transitions sont étiquetées des instructions gardées du programme source.

La seconde étape 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 sur notre exemple :

$$
\begin{array}{llllll}
f_{cout}(\texttt{S1calcul}) & = & [2,10] & f_{cout}(\texttt{S2calcul1}) & = & [1,+\infty] \\
f_{cout}(\texttt{S2calcul2}) & = & [3,20] & f_{cout}(\texttt{reqPrendre}()) & = & [0,0] \\
f_{cout}(\texttt{traitPrendre}()) & = & [0,+\infty] & f_{cout}(\texttt{relacher}()) & = & [0,0]
\end{array}
$$

La prise en compte de la fonction de coût pour obtenir un automate temporisé est obtenu par une simple transformation. Soit q un état de l'automate (*i.e.*, du graphe de flot de contrôle), une transition sortante de q étiquetée par I, et $f_{cout}(I) = [b, w]$, on transforme l'automate comme suit :

– l'initialisation d'un nouveau compteur ($t := 0$) est ajoutée à toutes les transitions entrantes de l'état q,

– l'invariant T associé à l'état q devient $T \wedge t \leq w$,

– la garde G de l'instruction I devient $G \wedge t \geq b$.

La nouvelle garde de l'instruction permet d'effectuer la transition dès que l'on a passé au moins b unités de temps dans q. Le nouvel invariant d'état impose de prendre une transition dès qu'on a passé w unités de temps dans l'état q.

Après ces deux étapes de transformation, on obtient les automates temporisés de la figure 5 pour les services `S1` et `S2`. Les instructions `Mi.prendre()` représentent la requête pour obtenir `Mi` et l'attente de l'accord, c'est à dire : `Mi.reqPrendre!;`

`Mi.traitPrendre?`. Les bornes 0 et $+\infty$ n'apportent pas d'information et ne sont pas intégrées dans l'automate. Le calcul `S1calcul` est modélisé à l'aide du compteur c mis à zéro avant d'arriver dans l'état représentant le calcul. La transition sortante est étiquetée par $c \geq 2$: l'automate ne peut prendre cette transition avant d'avoir passé 2 unités de temps (le BCET de `S1calcul`). L'invariant associé à cet état $c \leq 10$ impose de sortir de cet état avant 10 unités de temps (le WCET de `S1calcul`).

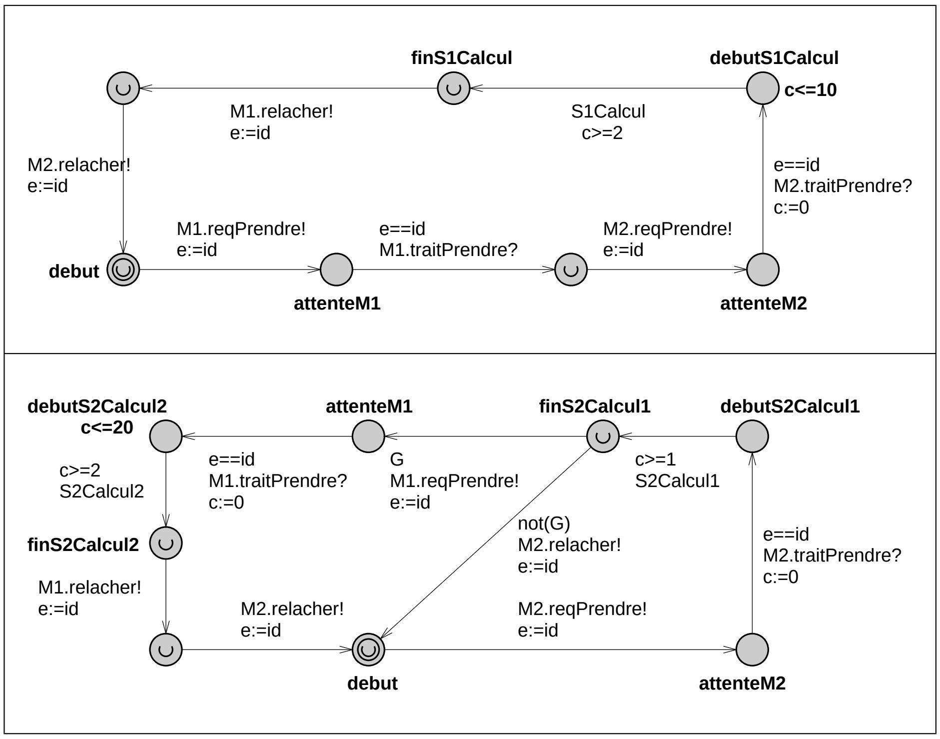


Figure 5. *Abstraction des services S1 (haut) et S2 (bas) par des automates temporisés*

La sémantique des automates temporisés fournit deux types de progression : soit le système progresse en prenant une transition, soit le système progresse avec un avancement du temps. Dans ce second cas, l'avancement du temps doit respecter les invariants sur les états. Les états urgents (étiquetés par un U) codent les états dans lesquels il n'est pas possible d'attendre.

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 services seront finalement traitées “un jour”. Ce style de propriété (de type vivacité) n’est pas imposable par instrumentation de code (Schneider, 2000). Seules des propriétés de sûreté peuvent s’imposer par tissage (ou par un moniteur). 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. En ce qui concerne les interblocages, il existe plusieurs types de solutions : locales ou globales, prévention ou détection.

Aspects temporels

Les aspects de disponibilité qui bornent les temps d’allocation des ressources sont définis par des automates temporisés. Ce formalisme est bien adapté pour exprimer des propriétés de type "temps borné". Il serait possible de concevoir un langage de plus haut niveau qui se compilerait en automates temporisés. Néanmoins, les outils et interfaces d’UPPAAL rend ce formalisme praticable au moins pour de petits exemples. Nos aspects sont ici limités à un seul type d’action (*advice*) qui consiste à forcer la libération de toutes les ressources allouées et terminer le service (*i.e.*, la boucle de traitement courante). 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. D’autres actions sont envisageables (par ex., libérer une ressource et reprendre le service à partir d’un point de reprise).

La figure 6 présente deux aspects de disponibilité que nous souhaitons imposer au service `S2` :

– l’aspect A_1 impose que `M1` soit relâchée avant 25 secondes ;

– l’aspect A_2 impose que `M2` soit relâchée avant 35 secondes ;

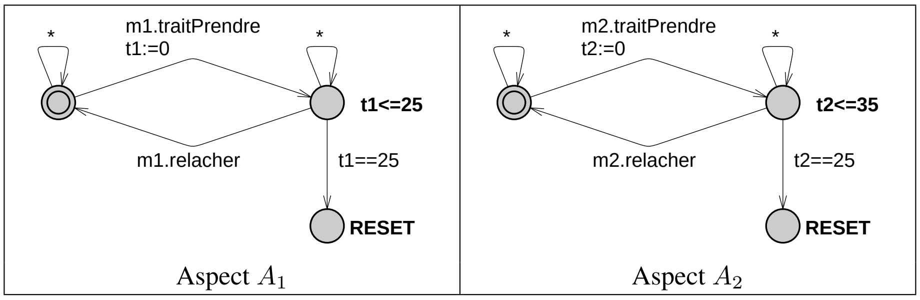


Figure 6. *Exemples d’aspects temporels*

Intuitivement, le tissage consistera à introduire un compteur de temps puis d’interrompre le service (en le faisant relâcher ses ressources) lorsqu’il dépasse son quota de temps (*i.e.*, 25 sec. ou 35 sec.). Nous décrivons cette étape plus précisément en section 6.

Des aspects plus sophistiqués ou d'autres types d'action peuvent se décrire dans ce cadre. Par exemple, les aspects peuvent spécifier des bornes de temps différentes selon l'histoire de l'exécution ou les services. Une politique souvent utilisée est d'empêcher un service qui accapare une ressource de redemander les mêmes ressources tout de suite. Dans le cas où les services ont des priorités, il est naturel de faire baisser la priorité de services ne respectant pas les contraintes de temps.

Aspects d'ordonnancement

Les interblocages sont une autre cause de dénis de service. Les aspects temporelles précédents peuvent résoudre les interblocages en forçant la libération des ressources. Cependant cette solution n'est pas optimale car le système peut se retrouver dans un état bloqué à attendre qu'une limite de temps soit atteinte et une ressource libérée. D'autres solutions pour résoudre les problèmes d'interblocages sont envisageables :

1) Une première solution est de spécifier un bon ordre d'acquisition des ressources par un automate. Un produit de cet aspect avec le service résulte en une coupure de celui-ci lorsqu'il viole cet ordre. Dans notre exemple, cela revient à couper `S2` lorsqu'il alloue `S1` c'est à dire à chaque fois que la garde G est vraie. Cette approche à l'avantage d'utiliser les mêmes outils que les aspects temporels mais peut résulter dans l'interruption systématique de services qui ne pourront plus remplir leur tâche.

2) Une deuxième solution est d'assurer un bon ordre d'acquisition en transformant le service pour qu'il respecte cet ordre en restant le plus proche possible de l'allocation initiale. Pour notre exemple, un tel aspect peut s'écrire

```
S2 : prendre(M1) ; prendre(M2) ; relacher(M2) ; relacher(M1) ;
```

Le service `S2` est transformé en allouant `M1` avant `M2`. Les instructions `M1.prendre() ;` et `M1.relacher()` sont déplacées avant `M2.prendre()` et `M2.relacher()` respectivement. Cette solution à l'avantage de prévenir les interblocages sans interrompre de service. L'inconvénient est que certains services doivent retenir certaines ressources plus longtemps (dans notre exemple, `S2` retient `M1` plus longtemps). Dans certains cas, cela peut rentrer en conflit avec d'autres aspects de disponibilité.

3) Une troisième solution, plus précise, est d'utiliser un mécanisme global de détection des interblocages (Krishnamurthi *et al.,* 1994). A chaque traitement d'une requête `prendre`, on vérifie qu'on n'introduit pas un interblocage (absence de cycle). L'aspect se résume au choix d'actions à effectuer en cas d'interblocage (*e.g.,* quel service interrompre dans les différents cas). Dans cette solution, il n'y a pas de tissage de code. Il s'agit plutôt d'un moniteur global de détection/résolution d'interblocages fonctionnant en parallèle avec le reste du système.

6. Tissage

Le tissage d'un aspect sur un service s'effectue en deux étapes. La première étape combine l'automate représentant le service et l'automate de l'aspect en un unique automate temporisé. L'automate résultant décrit le comportement du service respectant la propriété spécifiée par l'aspect (*e.g.,* s'interrompant en cas de dépassement des bornes de temps). La seconde étape transforme cet automate dans le code source correspondant.

Intégration d'un aspect à un service

Un aspect est intégré à un service par un produit d'automates temporisés. Ce type de produit ressemble au produit d'automate habituel modulo les différences suivantes :

– les invariants des états de l'automate résultat sont la conjonction des invariants d'états des automates initiaux ;

– les gardes des transitions de l'automate résultat sont la conjonction des différentes gardes des automates initiaux ;

– les actions des transitions de l'automate résultat sont la séquence des actions correspondantes des automates initiaux.

L'état puits RESET ajouté par les aspects de disponibilité, est interprété en ajoutant des transitions relâchant les ressources suivi par une transition qui revient au début de la boucle de traitement des requêtes.

L'automate temporisé résultat est ensuite analysé pour enlever les gardes, horloges, et invariants d'états inutiles et les états non atteignables. Cette optimisation est importante : elle permet d'enlever les timers inutiles et de diminuer le surcoût introduit par l'aspect. La figure 7 montre l'automate obtenu pour le service `S2` après tissage des aspect A_1 et A_2. L'aspect A_1 interdit au service de retenir la ressource `M1` plus de 25 sec. Le tissage de cet aspect n'a aucun impact sur le code car l'automate du service indique clairement que `S2calcul2` (et donc l'utilisation de `M1`) dure au plus 20 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.

Génération du code source résultat à partir de l'automate temporisé

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 est la prise en compte des instructions temporelles (initialisation d'horloge, invariants d'états et gardes temporelles). Nous gérons ces traits temporels en utilisant des timers. Ces timers sont similaires à la classe *Timer* de JAVA et sont utilisables par les instructions suivantes :

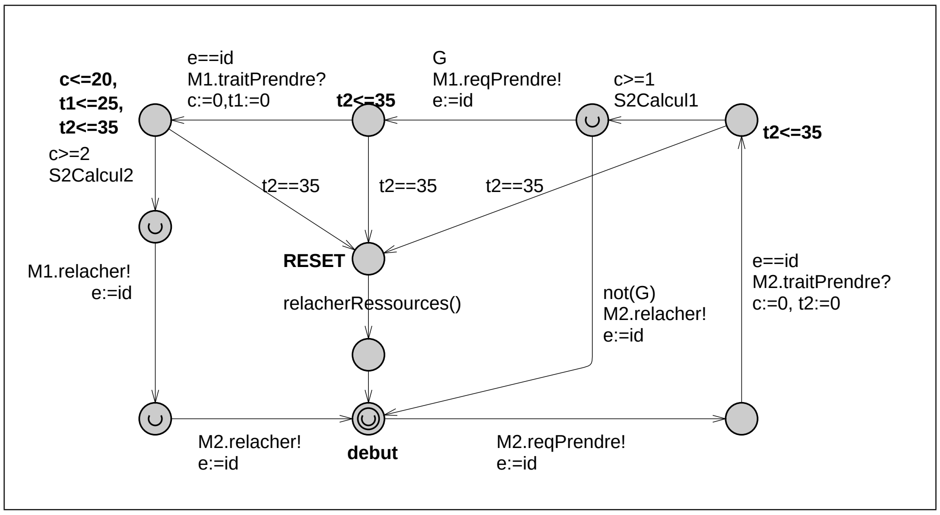


Figure 7. *Automate temporisé du service* `S2` *après tissage des aspects* A_1 *et* A_2

– `new Timer()` pour créer un objet `Timer` ;

– `schedule(t,d)` pour programmer l'exécution de la tâche `t` après le délai `d` ;

– `cancel()` pour décharger les routines programmées.

La figure 8 montre le code source du service `S2` obtenu de l'automate de la figure 7. Après l'instruction `M2.traitPrendre()`, un nouveau timer `tim` est initialisé et la routine `routineReset` est programmée pour s'exécuter après 35 secondes. Dans le cas où le service prend moins de 35 sec pour finir son traitement, la ressource `M2` est relâchée (`M2.relacher()`) et la programmation de `routineReset` est déprogrammée.

Le tissage des aspects d'ordonnancement (*c.f.* Section 5) diffère selon les cas. Pour la première catégorie (ordre d'acquisition imposé par un automate) le tissage est également un produit d'automate. La seconde catégorie nécessite de transformer les services (par ex. 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. La troisième catégorie (détection d'interblocage) est plus naturellement implémentée par un moniteur global que par tissage statique.

$$\texttt{S2} = \left\{ \begin{array}{ll} \texttt{l}_2: & \texttt{M2.prendre}();\\ & \texttt{tim} = \texttt{new Timer}();\ \texttt{tim.schedule}(\texttt{routineReset}, 35);\\ & \texttt{S2Calcul1};\\ & \texttt{if G then M1.prendre}();\\ & \qquad\qquad \texttt{S2Calcul2};\\ & \qquad\qquad \texttt{M1.relacher}();\\ & \texttt{M2.relacher}();\ \texttt{tim.cancel}();\\ & \texttt{jmp l}_2; \end{array} \right\}$$

$$\texttt{routineReset} = \left\{ \begin{array}{l} \texttt{relacherRessources}();\\ \texttt{jmp l}_2; \end{array} \right\}$$

Figure 8. *Code du service* S2 *après tissage*

7. Vérification

Les aspects de disponibilité ne définissent pas directement une propriété de disponibilité. Ce sont plutôt un ensemble de contraintes (temporelles, ordre d'acquisition) censés impliquer une propriété de disponibilité de plus haut niveau. Cette propriété peut être vérifiée sur le système obtenu après tissage par *model-checking*. 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 utilisons UPPAAL qui permet de modéliser les services, d'exécuter (et de mettre au point) les modélisations et de 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 :

– $A[]\ not\ deadlock$: 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.

– $A[]\ S1.dispo\ <=\ 45$: 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

8. Conclusion

Nous avons présenté un cadre permettant d'imposer des propriétés de disponibilité sur un système de services partageant des ressources. Ce cadre utilise les automates temporisés comme formalisme sous-jacent. L'approche est assez flexible pour prendre en compte des types de ressources variés et pour décrire 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 aspect global, simple et compréhensible pour assurer la disponibilité du processeur et de la mémoire. Ils décrivent le tissage sur les services mais cet aspect n'est pas générique et ne s'applique pas à d'autres types de ressources (*e.g.*, les ressources `mutex`). Fradet et Colcombet (Colcombet *et al.,* 2000) ont proposé une technique pour imposer des politiques de sécurité exprimées par des automates. Une partie de notre travail peut se voir comme une généralisation de cette approche à des propriétés de sûreté exprimées par des automates temporisés.

Nous travaillons actuellement à compléter la formalisation des différentes étapes (en particulier, le tissage des aspects d'interblocage). Nous souhaitons aussi définir un langage dédié d'aspects de plus haut niveau qui se compilerait dans des automates temporisés. L'idéal serait un langage où on exprimerait le "quoi" (par ex. `S1` ne doit jamais avoir à attendre plus de x sec. la ressource `M1`) plutôt que le "comment" (*i.e.,* limiter les temps d'utilisation des ressources). Finalement, nous envisageons de traiter un véritable cas d'étude afin de mieux valider l'approche et le choix d'UPPAAL.

9. Bibliographie

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.

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

Krishnamurthi M., Basavatia A., Thallikar S., « Deadlock detection and resolution in simulation models », *WSC '94 : Proceedings of the 26th conference on Winter simulation*, Society for Computer Simulation International, p. 708-715, 1994.

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.

Leiwo J., Zheng Y., « A Method to Implement a Denial of Service Protection Base », *ACISP '97 : Proceedings of the Second Australasian Conference on Information Security and Privacy*, Springer-Verlag, p. 90-101, 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*, IEEE Computer Scienty Press, 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.