

# DISPO : Disponibilité de services et composants logiciels sécurisés

ACI Sécurité Informatique

Rapport intermédiaire, avril 2005  
édité par Thomas Jensen

## 1 Résumé des objectifs

L'objectif du projet DISPO est de proposer des méthodes de construction et de validation de logiciels sécurisés. Nous nous focalisons sur des propriétés liées à la *disponibilité* et absence de déni de service d'un logiciel. Les méthodes proposées sont conçues pour intégrer et tirer profit des nouvelles méthodes de structuration de logiciels en composants et aspects. Une attention particulière est portée à la définition d'interfaces de sécurité afin de permettre la vérification d'un assemblage par composition d'interfaces.

Avec la confidentialité et l'intégrité, la **disponibilité** est l'une des trois propriétés de sécurité de base. Cependant, alors qu'il existe plusieurs modèles pour les propriétés de confidentialité et d'intégrité, le concept de disponibilité n'a été que très peu étudié. Liée à ce concept, la notion de **ressource** (de calcul, de stockage, de communication...) joue un rôle central dans le projet. La disponibilité d'une ressource dépend de la politique de sécurité qui gère l'accès à la ressource, de la consommation des processus individuels et de la façon dont plusieurs processus se partagent cette ressource.

Un premier défi du projet a été d'identifier un cadre formel permettant de formaliser une politique de sécurité. Nous nous sommes focalisés sur un cadre fondé sur la logique *déontique* qui permet d'exprimer la notion d'obligation. L'obligation d'effectuer une opération (notamment dans un délai déterminé) permet d'assurer certaines propriétés de disponibilité mais les politiques exprimées dans ce formalisme peuvent ne pas être cohérentes et donc impossible à réaliser. Il est donc nécessaire de pouvoir analyser une telle spécification afin de prouver d'une part sa cohérence et d'autre part son respect par un système donné.

Pour ce deuxième objectif important pour le projet nous cherchons à modéliser et formaliser la propriété de disponibilité d'un système de manière à pouvoir assurer que les mécanismes de protection de logiciel utilisés par ailleurs permettent effectivement de garantir la disponibilité du système. Pour les premiers travaux qui ont visé à obtenir ces assurances, nous avons utilisé des techniques d'analyse statique, de vérification de modèles et de transformation de logiciel visant à assurer des propriétés d'un composant logiciel.

Pour le passage à l'échelle des méthodes conçues, un axe du projet concerne la vérification modulaire et l'assemblage de logiciel par **composants** et par aspects. L'objectif est de concevoir des formalismes permettant de décrire l'architecture d'un logiciel et les interfaces de ses composants afin de faciliter la vérification de propriétés sécuritaires de l'assemblage.

## 2 Résultats nouveaux

### Résumé de résultats intermédiaires

- Un modèle de disponibilité d'un système informatique basé sur les notions déontiques de *droits* et d'*obligation* pouvant s'appliquer à des actions non atomiques.
- Une méthode de vérification de la *cohérence* d'une politique de sécurité.
- Un modèle de composants fondé sur les systèmes de transitions symboliques.
- Une technique de *fusion* de composants permettant d'obtenir un programme séquentiel à partir d'un réseau de composants.
- Des analyse statiques permettant d'identifier des consommations de ressources non bornées, pouvant conduire à la non-disponibilité.

### 2.1 Modèles et politiques de disponibilité

Nous nous sommes intéressés aux occurrences de dénis de service liés à des conflits d'accès à des ressources. Dans ce cas particulier, des situations d'interblocage ou de famine peuvent s'interpréter comme des dénis de service. Les réflexions sur le cadre logique permettant d'exprimer formellement une politique de disponibilité se sont appuyées sur l'étude du problème des philosophes, qui constitue le paradigme général des problèmes de conflit d'accès à des ressources.

Son analyse nous a conduit à identifier la notion d'obligation comme clé de voûte de ce type de politique. Le cadre logique adéquat doit ainsi nécessairement intégrer des aspects déontiques. La notion d'obligation s'accompagne très souvent d'une notion de délai permettant d'exprimer l'intervalle temporel de réalisation de l'obligation. Cela conduit à intégrer dans ce même cadre logique, des aspects temporels.

Un modèle formel de sécurité basé sur une logique déontique et temporelle a été ainsi développé [4]. Il propose un formalisme homogène permettant l'expression de privilèges simples ou conditionnels associés à des actions non atomiques. Les privilèges sont de type permission, interdiction, obligation, obligation avec délai. La propagation de ces privilèges, lorsqu'on décompose une action non atomique en actions atomiques, est analysée.

Pour la vérification formelle de politiques de disponibilité exprimées dans une logique déontique nous avons proposé une sémantique, basée sur des structures de Kripke [2], pour un tel langage. La principale difficulté réside dans l'imbrication des opérateurs temporels et déontiques. Savoir si une politique - exprimée dans ce langage - est consistante revient à décider de la satisfiabilité de la formule correspondante. Pour illustrer ces aspects, nous avons considéré une étude de cas simple exprimée à l'aide d'automates temporisés [1]. Nous avons utilisé UPPAAL comme un outil de modélisation. Une première phase a donc consisté à traduire manuellement les opérateurs déontiques en logique temporelle. Nous avons été ainsi amené à définir des règles de traduction pour un fragment de la logique déontique.

### 2.2 Architecture à composants et aspects intégrant la sécurité

Les travaux sur les architectures de logiciels utilisent des modèles de composants où la communication entre composants varie en complexité. Nous avons conçu une technique pour *fusionner* des composants organisés en un réseau de Kahn et communicants par des canaux FIFO. Dans un deuxième travail nous avons étudié des réseaux de composants où les protocoles

de communication entre composants sont décrits par des systèmes de transitions symbolique, permettant une communication plus variée.

Nous avons développé une méthode de composition de composants (la *fusion de composants*) qui transforme automatiquement un réseau de composants en un unique programme séquentiel [5]. La fusion prend en entrée un réseau de composants et un aspect d'ordonnement. Le réseau est formalisé par des réseaux de Kahn (désignés par KPNs pour « Kahn Process Networks »), un modèle de parallélisme à la fois simple et expressif. Dans ce modèle, un composants est un programme séquentiel qui effectue des lectures et des écritures sur des ports de communication. Ces ports sont liés par des canaux FIFO. L'aspect d'ordonnement permet à l'utilisateur de guider et contrôler la stratégie d'ordonnement. L'aspect se présente sous la forme de contraintes qui spécifient le sous-ensemble des exécutions permises. La sémantique opérationnelle des KPNs et l'aspect d'ordonnement sont formalisés par des systèmes de transitions étiquetées. Le programme obtenu rend les mêmes résultats que le KPN modulo les blocages artificiels (« deadlocks ») éventuellement introduits par les contraintes (auquel cas, l'utilisateur est averti). Le fusion diminue le surcoût introduit par l'assemblage de composants en permettant la suppression des changements de contexte, le remplacement des communications internes par des affectations à des variables locales et l'utilisation d'optimisations standard sur le programme séquentiel résultant.

Nous avons également considéré les modèles de composants avec protocoles explicites. Dans un premier temps, nous avons proposé d'enrichir les modèles usuels de composants avec des protocoles explicites basés sur des systèmes de transitions symboliques (STS) [10]. Cette notion facilite la spécification des composants et permet de faire un lien avec le monde de la vérification tout en fournissant un support pour l'implémentation des composants. Pour ce qui est de la disponibilité, elle permet de modéliser séquences d'accès aux services et aux ressources mais aussi d'intégrer des données permettant par exemple de quantifier les ressources.

À un niveau plus opérationnel, nous avons considéré, à la suite du travail de DEA de Sebastian Pavel [9], la possibilité d'ajouter des protocoles explicites au langage de composants ArchJava. Ce choix nous est toutefois rapidement apparu comme inutilement contraignant au niveau de l'implémentation et limitant la généralité de notre approche. Nous sommes revenus à un modèle très simple : un composant atomique est fabriqué à partir d'une implémentation des services en Java et un STS, un composant structuré est obtenu par une composition parallèle de ses sous-composants et, éventuellement, application d'un nouvel STS. Ce modèle ainsi qu'un premier prototype sont décrits dans [8]. Chaque STS est traduit en un objet actif « contrôleur » qui maintient l'état logique du protocole et gère les interactions du composant en accord avec cet état. D'une certaine manière, ce contrôleur peut être vu comme l'implémentation d'un aspect d'interaction du composant.

### 2.3 Assurer des propriétés de disponibilité

Nous avons conçu des analyses permettant de vérifier le caractère borné d'une part de la consommation de ressources à l'intérieur d'un composants et d'autre part de la communication (taille de boîte à lettres) entre composants.

Au niveau de la vérification « intra-composant » nous avons conçu un algorithme pour analyser la consommation de ressources (mémoire, bande passante...) d'un programme afin d'assurer des propriétés liées à la disponibilité et le déni de service. L'analyse est formalisée dans le cadre d'un langage intermédiaire comme le byte code Java et se focalise sur la consom-

mation de mémoire mais elle s'étend facilement à d'autres types de langages procéduraux et à d'autres types de ressources. L'analyse vérifie pour une instruction critique donnée qu'un programme n'exécute cette instruction qu'un nombre borné de fois. La base de l'algorithme est un algorithme de détection de boucles dans des graphes de flot de contrôle inter-procéduraux. L'algorithme d'analyse est destiné à être utilisé pendant le développement d'applettes ou comme une partie d'un vérifieur de byte code amélioré pouvant être installé sur un système embarqué. Nous avons donc visé à développer un algorithme de faible complexité. L'algorithme est exprimé comme une analyse statique basée sur des contraintes. Son lien fort avec l'interprétation abstraite a permis de formaliser l'algorithme à l'aide de l'assistant de preuve Coq et, de ce fait, extraire un analyseur certifié à l'aide du mécanisme d'extraction de Coq [3].

Au niveau de l'architecture de composants, l'utilisation de communications asynchrones implique que le caractère borné de ces systèmes n'est pas assuré *a priori* et nécessite donc des vérifications statiques. Nous avons travaillé sur deux types de modélisation des boîtes à lettres nécessaires à la communication asynchrone : une modélisation sous la forme d'un dictionnaire des appels de services (associant à chaque service le nombre de requêtes en attente) et une modélisation plus fine sous la forme d'une file d'attente représentant explicitement toutes les requêtes. Le caractère borné des dictionnaires est décidable mais celui des files d'attente ne l'est pas, toutefois le caractère borné des dictionnaires est une condition suffisante pour celui des files d'attente. Ces résultats ont été publiés dans [12, 6]. Nous nous sommes également intéressés à la preuve avec la vérification de modèles et un prouveur général, PVS en l'occurrence. L'idée que nous avons abordée dans [7] est la possibilité d'utiliser les résultats d'une première analyse du caractère borné des boîtes à lettres pour simplifier le modèle avant vérification. Un travail complémentaire, décrit dans [11], propose une logique temporelle, extension de CTL\* avec des données, pour les STS.

## 3 Perspectives

### 3.1 Modèles et politiques de disponibilité

La première partie du projet a permis d'élaborer un formalisme fondé sur la notion d'obligation et la logique déontique. Dans la deuxième partie nous pensons valider ce formalisme par des études de cas et de l'intégrer avec d'autres formalismes. Nous avons identifié deux activités à court/moyen terme :

- L'application du modèle déontique à des cas de déni de service provenant du monde des réseaux, notamment le « Syn Flooding ».
- Le cas d'étude (la solution de Chandy et Misra au problème des philosophes <sup>[1]</sup>) nous a montré l'intérêt d'intégrer des notions déontiques dans une logique dynamique. Une direction possible serait d'exprimer explicitement le temps au sein de la logique dynamique, et d'y intégrer la notion d'obligation.

### 3.2 Aspects et composants

Nos réflexions sur l'utilisation des STS pour exprimer des propriétés de disponibilité nous amènent à penser qu'un grand nombre de situations pourrait être modélisé à l'aide

---

[1] K.M. Chandy and J.Misra. The drinking philosophers problem. In *ACM TOPLAS*, Vol. 6, No. 4, pp.632-646, October 1984.

d'*automates avec compteurs*, représentant l'existence de ressources en quantité limitée (ce qui peut inclure le temps). Suivant les propriétés des fonctions décrivant l'évolution des compteurs pour chaque transition, il peut être possible de calculer les configurations atteignables et donc de garantir certaines propriétés comme l'impossibilité d'épuiser une ressource critique. Voir par exemple les travaux de Comon et Jurski [2]. Nous allons également étudier la conception d'un langage d'aspects basé sur la notion d'automates avec compteurs.

Nous comptons faire évoluer notre modèle de composants avec protocoles explicites en un véritable langage, à la fois plus expressif (par exemple, introduction des notions de restriction et de renommage, possibilité de considérer des transitions comme des exécutions atomiques ou des événements, prise en compte de la distribution) et plus efficace (gestion des notifications, fusion de l'implémentation des composants et des contrôleurs).

Nous nous intéressons aussi aux situations où en l'absence de politique de disponibilité, on aboutit à des dénis de services dus à des conflits d'intérêts non régulés entre les différents acteurs. La théorie des jeux fournit alors un cadre naturel pour analyser ce type de problème. Ceci nous conduit à identifier des formes plus ou moins restrictives de la propriété de disponibilité, déterminées par les possibilités respectives des différents acteurs pour rendre la satisfaction de la disponibilité, inévitable ou impossible. Cette analyse permet de caractériser le rôle particulier de régulateur de conflits dévolu à la politique de disponibilité. Cette régulation des conflits peut alors être mise en œuvre au moyen de contrats entre les différents acteurs. Notre approche s'appuie sur l'exemple de l'allocateur de ressources ainsi que sur le problème des philosophes. Ce travail est en cours de finalisation.

A plus long terme et de façon plus spéculative, nous envisageons d'étudier l'interprétation sous forme d'aspects d'une propriété de disponibilité exprimée dans le modèle Nomad développé dans [4].

### 3.3 Vérification

La vérification du respect d'une politique de disponibilité déontique par un système reste à définir. Ceci implique à la fois la définition de la notion de validité et la mise en œuvre d'une méthode de vérification associée.

Une propriété fondamentale de disponibilité est la garantie que toute requête reçoit une réponse. Cette garantie peut s'exprimer par une formule de la logique du second ordre monadique, interprétée dans l'ensemble des séquences de services. On peut alors s'appuyer sur les résultats de Rabin [3] (équivalence de la définissabilité logique et de la reconnaissance par automates d'arbres - où un arbre code ici un ensemble de séquences) pour produire une procédure de décision.

La vérification compositionnelle demeure un objectif central du projet. Nous allons étudier comment l'analyse de la consommation de mémoire décrite en Section 2.3 peut s'appliquer à des fragments de programme afin d'inférer des interfaces permettant une analyse compositionnelle de composants logiciels.

---

[2] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In A. Hu and M. Vardi, editors, *Proc. Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279, Vancouver, 1998. Springer-Verlag.

[3] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. of Amer. Math. Soc.*, 141:1–35, 1969.

## Références

- [1] Julien Brunel. Deontic Logic for the Specification of System Availability. In *6th school on MOdeling and VErifying parallel Processes MOVEP'04 (student papers)*, Bruxelles, pages 40–45, 13-17 décembre 2004.
- [2] Julien Brunel. Logique déontique pour la disponibilité. Rapport de stage DEA PS 2003-2004, IRIT, Université Paul Sabatier, Toulouse, 2004.
- [3] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In *Proc. Formal Methods (FM'05)*, 2005.
- [4] F.Cuppens and N.Cuppens-Boulaïhia. Nomad : a security model with non atomic actions and deadlines. In *Proc. 18th IEEE Computer Security Foundations Workshop*, juin 2005.
- [5] P. Fradet and S. Hong Tuan Ha. Network fusion. In Wei-Ngan Chin, editor, *Programming Languages and Systems : Second Asian Symposium, APLAS 2004*, volume Springer LNCS vol. 3302, pages 21–40, 2004.
- [6] Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking asynchronously communicating components using Symbolic Transition Systems. In R. Meersman and Z. Tari, editors, *On The Move to Meaningful Internet Systems 2004 : CoopIS, DOA, and ODBASE*, number 3291 in Lecture Notes in Computer Science, pages 1502–1519. Springer-Verlag, 2004.
- [7] Jacques Noyé, Sebastian Pavel, and Jean-Claude Royer. A PVS experiment with asynchronous communicating components. In *17th Workshop on Algebraic Development Techniques*, Barcelona, Spain, March 2004.
- [8] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.
- [9] Sebastian Pavel, Jacques Noyé, and Jean-Claude Royer. Dynamic configuration of software product lines in ArchJava. In Robert L. Nord, editor, *Software Product Lines : Third International Conference, SPLC 2004*, number 3154 in Lecture Notes in Computer Science, pages 90–109. Software Engineering Institute, Springer-Verlag, August 2004.
- [10] Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. Formal methods for component description, coordination and adaptation. In *Proceedings of the ECOOP Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04)*, Oslo, Norway, June 2004.
- [11] Jean-Claude Royer. A framework for the GAT temporal logic. In *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, pages 275–280, Nice, France, July 2004. ISCA.
- [12] Jean-Claude Royer and Michael Xu. Analysing Mailboxes of Asynchronous Communicating Components. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *On The Move to Meaningful Internet Systems 2003 : Coopis, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1421–1438. Springer-Verlag, 2003.