

## Un modèle de composant avec protocole symbolique

Sebastian Pavel, Jacques Noyé, Jean-Claude Royer  
OBASCO École des Mines de Nantes - INRIA, LINA  
4, rue Alfred Kastler, 44307 Nantes cedex 3, France  
{Sebastian.Pavel, Jacques.Noye, Jean-Claude.Royer}@emn.fr

### Mots-clefs – Keywords

CBSE, protocole explicite, système de transitions symboliques, contrôleur, canaux de communication

CBSE, Behavioural IDL, Explicit Protocols, Symbolic Transition Systems, Controllers, Channels

### Résumé – Abstract

Les composants proposent une approche du développement logiciel reposant sur des assemblages d'entités réutilisables. Un aspect important est la détection et adaptation des incompatibilités lors de la composition. Notre travail suggère d'utiliser des descriptions des interactions entre composants basées sur une notion de systèmes de transitions symboliques (STS). Nous décrivons dans ce résumé les principes de l'implémentation d'un modèle de composants qui prend en compte les STS. Elle est basée sur une notion de contrôleur qui encapsule le STS et contrôle son comportement. Les communications entre les composants sont réalisées par l'intermédiaire de canaux de communications.

Component-Based Software Engineering (CBSE) has now emerged as a discipline for system development. Important issues in CBSE such as composition incompatibility detection and (dynamic) adaptation can only be addressed with the help of formal component models with Behavioural Interface Description Languages (BIDLs) and explicit protocols. The issue is then to fill the gap between such high-level models and implementation. This paper suggests to do so by relying on Symbolic Transition Systems (STSs). It describes a component model with explicit symbolic protocols based on STSs, and its implementation principles. This implementation is based on controllers that encapsulate protocols and channels devoted to (possibly remote) communications between components.

## 1 Introduction

Les composants proposent une approche du développement basée sur l'assemblage d'entités réutilisables. Il existe actuellement plusieurs travaux dans le domaine académique comme dans le monde industriel, on peut penser à des exemples comme Olan [BAKR95], Fractal [CBS02] ou encore aux EJB [DeM03]. Les langages d'architectures [MT00] proposent aussi des approches très voisines dans l'esprit mais souvent à un niveau plus spécification que programmation. Nous nous intéressons ici aux modèles ou langages de composants permettant la définition de composants hiérarchiques. Une difficulté importante est de pouvoir assurer certaines propriétés des assemblages. Il ne s'agit pas seulement de propriétés de typage mais par exemple d'assurer l'absence de blocage au sens dynamique du terme ou encore la vivacité de certaines actions. Par exemple dans le contexte du projet DISPO [DIS] nous voulons pouvoir assurer que les composants vérifient des propriétés de disponibilité. Pour aborder ce problème nous proposons un modèle de composants disposant, en plus de la notion habituelle d'interface, de protocoles. Il existe déjà quelques langages de spécifications formelles d'architectures proposant de telles constructions, par exemple WRIGHT [All97]. Par ailleurs, des langages de programmation comme SOFA [KT02] ont utilisé les expressions régulières dans ce but. Notre approche est plutôt d'utiliser des systèmes de transitions symboliques (STS [CMS02]), c'est-à-dire des machines à états finis avec des étiquettes autorisant des variables et des gardes. Ce formalisme a le principal avantage d'être lisible et d'éviter les problèmes d'explosion du nombre d'états et de transitions des machines à états et transitions étiquetées. Nous présentons dans ce document les principes d'implémentation de ce modèle en Java. Ceux-ci reposent sur l'utilisation de contrôleurs qui se chargent de gérer les interactions de l'implémentation du composant avec son environnement et cela conformément au protocole défini par le STS. Des canaux de communications permettent de représenter les communications entre composants et d'offrir des possibilités intéressantes de dynamique.

## 2 Un modèle de composant basé sur les STS

Comme d'autres modèles ou langages de composants notre approche considère des composants munis d'un ensemble de services accessibles par des ports de communication. Ces services sont en fait regroupés dans une ou des *interfaces*. Les communications peuvent être en mode synchrone avec retour de valeur ou asynchrone. Nous proposons dans ce modèle une description de la dynamique des composants par l'utilisation de machines à états et transitions. Ces machines, appelé systèmes de transitions symboliques, décrivent la dynamique du composant. Elles autorisent l'utilisation de variables pour les communications mais aussi de gardes pour contrôler le déclenchement des transitions.

Nous allons décrire brièvement dans ce qui suit les éléments du modèle et les principes de son implémentation en Java. Notre cible est Java mais notre approche propose un modèle formel pour des besoins de vérifications qui peut être décliné vers d'autres langages concrets. Des détails complémentaires peuvent être trouvés dans [PPNR05] notamment une sémantique des STS et des éléments relatifs à la vérification.

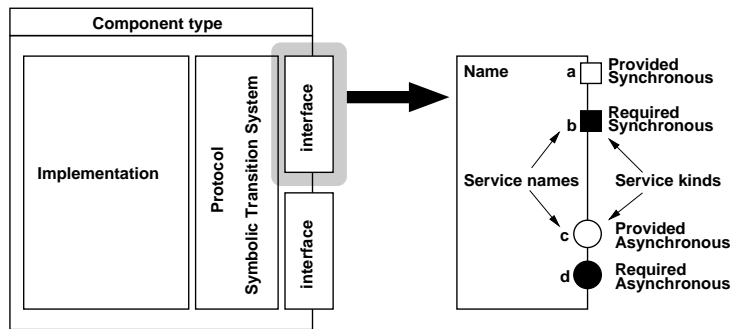


FIG. 1 – Représentation graphique des composants

## 2.1 Composant primitif

La partie visible ou publique d'un composant primitif est un nom, des interfaces nommées et un protocole. Ce composant primitif encapsule soit un type de données soit une autre application (éventuellement un autre composant) appelé son *implémentation*. Les services des interfaces correspondent à des fonctions ou des actions de la partie implémentation du composant. Les composants interagissent par l'intermédiaire d'interfaces nommées chacune correspondant à un *port* de communication. Une interface est un regroupement de services du composant. Un service est la donnée d'un nom unique dans l'interface et d'un type correspondant aux valeurs reçues ou émises par le service. Un service peut être *fourni* ou *requis* et fonctionner de manière *synchrone* ou *asynchrone*.

Les STS [CMS02] ont été développés à l'origine comme une solution au problème d'explosion du nombre d'état et de transition dans les algèbres de processus avec passage de valeurs. Nos STS sont une généralisation de cette idée. Ils associent un état symbolique et un système de transition avec une implémentation [Roy03]. La description abstraite de la partie implémentation peut être donnée en utilisant des spécifications algébriques, des formalismes orientés modèle ou encore des classes Java. Les STS peuvent être comparés aux *Statecharts* mais leur sémantique formelle est plus simple et plus stricte. Le STS va donner les ordonnancements autorisés des services requis et fournis du composant. Il n'y a qu'un STS pour l'ensemble des interfaces ou services du composant.

La partie implémentation du composant est une application encapsulée. Une hypothèse est que cette partie (une application Java dans notre cas) ne fournit pas seulement des services mais aussi l'ensemble des gardes qui sont utilisées dans le STS.

Au niveau syntaxique, nous avons défini des notations graphiques pour l'interface statique (Fig. 1) et pour le STS associé (Fig. 2). La figure 2 décrit le protocole STS associé à un composant appelé *Company* et qui représente une compagnie proposant des vols d'avion. Les services requis sont préfixés par *!* et les services fournis par *?*. Le signe  $\wedge$  dénote un message asynchrone. La syntaxe textuelle dépend de la cible du langage considéré, Java ici (voir [PPNR05] pour plus de détails).

Au niveau sémantique, plusieurs propositions ont été étudiées, voir par exemple [CPR00, MPR04]. La sémantique d'un composant avec STS peut être exprimée comme un graphe de configurations (voir [MPR04, PPNR05]). Un tel graphe est un système de transitions étiquetées avec des valeurs associées aux états. Ainsi les notions habituelles de traces, d'arbre d'atteignabilité, etc. sont transposables dans ce contexte et ouvrent la voie vers

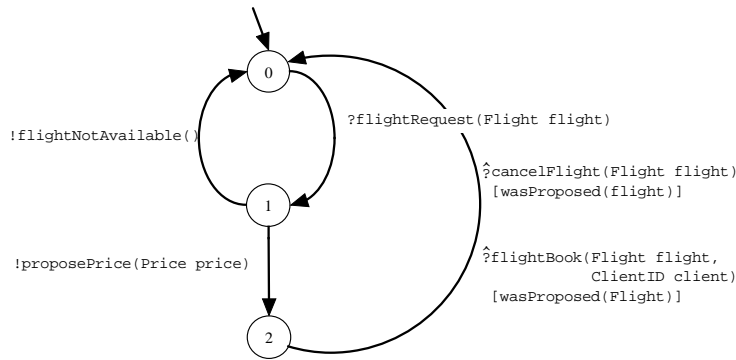


FIG. 2 – Représentation graphique d'un STS

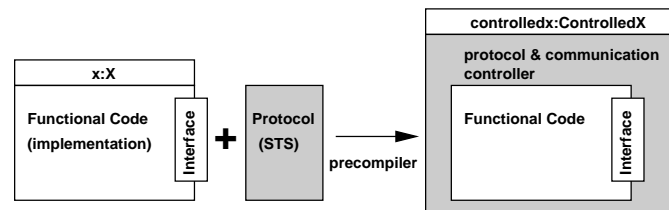


FIG. 3 – Principe de l'implémentation

des vérifications.

## 2.2 Composant composite

Un composant composite (ou hiérarchique) est un assemblage de composants (les sous-composants) et d'un ensemble de connexions entre les ports de ceux-ci. Les communications de base sont point à point et orientées mais des modes plus évolués (broadcast, ...) sont possibles. La correspondance entre deux interfaces liées est faite par un appariement des noms des services avec éventuellement renommage. Les services en correspondance doivent être compatibles au niveau du type et posséder le même mode de communication. Un composite est défini par une interface, les connexions entre ses sous-composants et les connexions entre ses sous-composants et son interface. Un composant composite peut être décrit sous une forme graphique ou sous une forme textuelle. La sémantique d'un composite est décrite par l'adaptation du produit synchronisé aux STS [CPR00, Roy03].

## 3 L'implémentation du modèle

L'idée générale de l'implémentation est donnée par la figure 3. Le principe est d'attacher un protocole à une application existante (l'implémentation). Après cette encapsulation le résultat est un composant primitif dont le STS contrôle la partie implémentation. Le composant fournit au plus les services définis par l'implémentation et inclut un mécanisme pour contrôler et imposer le protocole choisi. Le composant peut communiquer dans une architecture soit par une connexion locale ou par l'intermédiaire de RMI (si ses correspondants sont distants). L'implémentation des connexions d'une architecture est réalisée par l'utilisation de canaux de communication.

### 3.1 Implémentation du protocole

Pour adjoindre un protocole à une implémentation nous utilisons une entité active complémentaire (un *thread* Java) qui joue le rôle de contrôleur pour le composant. Le but de ce contrôleur est : (1) d'intercepter les messages reçus ou émis par l'implémentation et (2) de décider quand ces messages sont autorisés ou interdits. L'implémentation est réalisée en utilisant le principe de l'état logique (*logical state pattern* [GHJV95]). Ce patron est implémenté en deux étapes. Tout d'abord, les états possibles du protocole sont déclarés comme des variables privées du contrôleur. Ensuite, les actions à exécuter quand l'implémentation reçoit un message sont définies dans la méthode `run` du contrôleur. Les actions à réaliser quand l'implémentation doit envoyer un message sont définies dans les méthodes privées implémentant les opérations requises.

### 3.2 Canaux de communications

Pour la connexion des composants, nous réutilisons le principe des canaux de communications [ASdBB02]. Un canal représente une connexion anonyme entre deux composants. Il est orienté de la source vers le destinataire, ainsi il faut généralement deux canaux par connexion, un dans chaque sens possible de communication. Les canaux sont créés dans l'espace de visibilité du composite évitant ainsi des problèmes d'exportation hors de celui-ci. L'intérêt des canaux est qu'ils permettent facilement de prendre en compte le mode de communication synchrone ou asynchrone, la mobilité, des conditions, des priorités, etc. En plus de son rôle de coordination, un canal peut être utilisé pour faire des adaptations (d'interfaces et/ou protocoles), des connexions plus complexes que de simples connexions point à point (broadcast par exemple). Finalement ils uniformisent les communications (distantes ou locales) et autorisent la création et la connexion dynamique.

## 4 Conclusion

Dans ce résumé nous avons présenté l'introduction de protocoles explicites dans un modèle de composants hiérarchiques. Nous préconisons l'utilisation des systèmes de transitions symboliques pour des raisons de lisibilité et d'abstraction. Le modèle développé supporte des interfaces multiples hétérogènes et des communications synchrones et asynchrones. Une implémentation en Java de ce modèle a été prototypée. Elle repose sur la définition d'un mécanisme de contrôleur réalisant la partie protocole du composant. Ce contrôleur intercepte les messages reçus ou émis et déclenche le service correspondant uniquement si les conditions stipulées dans le STS sont réalisées. La communication entre les composants d'une architecture est réalisée par des canaux de communication. Ces canaux offrent beaucoup de souplesse et de puissance d'expression, ils autorisent de nombreux modes de communications et la mobilité.

## Références

- [All97] R. J. Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon University, 1997.
- [ASdBB02] F. Arbab, J. V. Guillen Scholten, F.S. de Boer, and M. M. Bonsangue. A channel-based coordination model for components. Technical report, Centrum voor Wiskunde en Informatica, 2002.
- [BAKR95] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-based programming and Application Management with Olan. In J. Briot, J. Geib, and A. Yonezawa, editors, *Object-Based Parallel And Distributed Computation*, volume 1107 of *LNCS*, pages 290–309. Springer-Verlag, Berlin, 1995.
- [CBS02] T. Coupaye, E. Bruneton, and J.-B. Stefani. The Fractal composition framework. Technical report, The ObjectWeb Group, 2002.
- [CMS02] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1) :55–61, 2002.
- [CPR00] C. Choppy, P. Poizat, and J.-C. Royer. A global semantics for views. In T. Rus, editor, *International Conference, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.
- [DeM03] L.G. DeMichiel. *Enterprise JavaBeans<sup>TM</sup> Specification*. SUN Microsystems, November 2003. Version 2.1, Final Release.
- [DIS] DISPO home page. <http://www.irisa.fr/lande/jensen/dispo.html>.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison Wesley, 1995.
- [KT02] T. Kalibera and P. Tuma. Distributed Component System Based on Architecture Description : The SOFA Experience. In R. Meersman, Z. Tari, and al., editors, *CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 981–994. Springer-Verlag, 2002.
- [MPR04] O. Maréchal, P. Poizat, and J.-C. Royer. Checking asynchronously communicating components using symbolic transition systems. In R. Meersman, Z. Tari, and al., editors, *CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer-Verlag, 2004.
- [MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE - Transactions on Software Engineering*, 26(1) :70–93, 2000.
- [PPNR05] S. Pavel, P. Poizat, J. Noyé, and J.-C. Royer. A formal component model with explicit symbolic protocols and its Java implementation. Technical report, École des Mines de Nantes, January 2005.
- [Roy03] J.-C. Royer. The GAT approach to specify mixed systems. *Informatica*, 27(1) :89–103, 2003.