

Approche formelle intégrée pour la spécification des architectures dynamiques orientées composants

Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel
University of Sfax
Laboratory LARIS

B.P. 1088, 3018 Sfax, Tunisia

E-mails : Imen.loulou@tunet.tn, {Ahmed@fsegs, Mohamed.Jmaiel@enis}.rnu.tn

Khalil Drira

LAAS-CNRS

7 avenue de Colonel Roche

31007 Toulouse Cedex 4, France

E-mail : Khalil@laas.fr

Abstract

Nous proposons dans cet article une approche formelle intégrée pour la spécification des architectures dynamiques orientées composants. Nous procédons par intégration d'une approche fonctionnelle et d'une approche structurelle basée sur les grammaires de graphes. Nous visons par là, la simplification de la spécification, l'amélioration de la lisibilité et de la compréhension et la prise en charge de la dynamique. En effet, nous utilisons le langage formel Z pour décrire le style architectural qui doit être préservé durant l'évolution d'une architecture, d'une part. D'autre part, nous décrivons la dynamique via des règles de transformation de graphes gardées dont le corps décrit les contraintes structurelles et dont les gardes décrivent principalement les contraintes fonctionnelles du système. Nous exprimons la sémantique avec la notation Z également, obtenant ainsi une approche unifiée qui prend en charge l'aspect statique et l'aspect dynamique. Nous utilisons pour cela l'outil d'analyse Z-EVES pour valider nos spécifications Z.

Keywords: architecture logicielle, style architectural, architecture dynamique, réécriture de graphes, applications orientées composants, spécification formelle.

1 Introduction

Les systèmes logiciels ont par le passé été développés dans deux communautés séparées : Wet et Dry tel que c'est décrit par Goguen [1]. La Communauté Wet est caractérisée

par la mentalité "Hacker", dans laquelle le développeur veut créer le système le plus vite possible, et utilise pour cela des modèles heuristiques de conception basés sur l'expérience antérieure et les anecdotes transmises entre programmeurs. Le Système peut (ou non) être documenté, et sa validation (par rapport à une spécification soft des besoins) peut être très difficile à mener. La deuxième alternative de développement est la communauté Dry dans laquelle uniquement les méthodes formelles sont d'usage permettant ainsi une conception rigoureuse. Néanmoins, les modèles utilisés sont souvent théoriques. L'objectif est donc de faire le lien entre les deux philosophies en termes de leurs principes de conception (heuristique et formelle) [2]. Ceci permettra, d'une part, de raisonner sur des systèmes complexes en les caractérisant à un haut niveau d'abstraction (apport de la communauté Dry). D'autre part, ce lien permettra aux concepteurs d'exploiter des modèles récurrents d'organisation de systèmes. De tels modèles connus sous le nom de styles architecturaux [3] facilitent le processus de conception en recourant à des solutions connues pour certaines classes de problèmes (apport de la communauté Wet). Ils peuvent mener à une réutilisation significative du code, facilitent la communication entre concepteurs et/ou utilisateurs et supportent l'interopérabilité. Dès lors que la conception d'architectures logicielles émerge comme une discipline du Génie Logiciel, cette démarche revêt une importance toute particulière.

En outre, l'industrie et la recherche informatiques, en général, et logicielles en particulier, doivent actuellement affronter des applications logicielles dont la complexité est

sans cesse en croissance. L'évolution oblige les concepteurs des nouvelles applications à faire face au renforcement des exigences applicatives traditionnelles et à l'introduction de nouvelles exigences liées à une réadaptation des applications nécessitant une création et une distribution dynamique des composants et de leurs supports d'interaction. L'adaptabilité des applications à ces changements est une exigence récente qui requiert une configuration dynamique et une évolution fréquente des architectures des nouvelles applications. Ceci constitue depuis quelques années un domaine de recherche et de développement articulé autour de l'étude des architectures logicielles dynamiques.

L'objectif principal est donc de contrôler l'évolution et la reconfiguration de l'architecture en exprimant leur conformité par rapport à un style architectural.

2 Etat de l'art

Notre étude des techniques de description et de reconfiguration des architectures logicielles nous a permis d'identifier deux courants de recherche. Le premier s'intéresse à développer des notations spécialisées appelées "langages de description d'architecture" (ADLs) pour remplacer les diagrammes informels de type Box-and-line. Les ADLs tels que Darwin [4], Rapide [5, 6], Wright [7] et LEDA [8] fournissent en fait un support de modélisation pour aider les concepteurs à structurer le système et à composer les différents éléments. L'ensemble des ADLs se limite à la description de la dynamique prédéfinie, dans le sens où ils s'intéressent uniquement à des systèmes ayant un nombre fini de configurations et qui doivent être connues a priori. Par ailleurs, à l'exception du langage Wright, les approches basées sur les ADLs ne permettent pas de différencier les styles architecturaux et ne sont pas adaptées à la vérification des propriétés décrites de l'architecture. Quant au deuxième courant, il consiste à appliquer des méthodes formelles existantes à la conception architecturale dans un but d'analyse et de vérification. Nous avons constaté que les travaux de ce courant sont pratiquement articulés autour de deux formalismes: les langages fonctionnels et les grammaires de graphes. Dans le contexte des langages fonctionnels, la logique temporelle a été utilisée dans [9] pour son grand pouvoir expressif et surtout pour la possibilité de vérification. Néanmoins, cette logique est un formalisme axiomatique difficile à appréhender qui requiert des experts qualifiés. En outre, la reconfiguration n'est pas bien explicitée avec ce formalisme. Dans [10], les auteurs ont développé des modèles formels avec Z pour les styles filter-pipe, et event system et Gamble et al. [2] ont utilisé Z pour spécifier un style dit à base de règles. Toutefois, ces travaux n'évoquent pas les problèmes liés à l'évolutivité des architectures des applications. Quant au principe du deuxième formalisme, il consiste à utiliser les

graphes pour la représentation des structures, étant donné qu'ils constituent le moyen mathématique le plus naturel et intuitif pour la représentation de situations complexes. Dans ce contexte, [11] propose de représenter l'architecture d'un système logiciel par un graphe et le style architectural via une grammaire de graphe à contexte libre. Ce type de grammaires ne permet pas de décrire certaines propriétés logiques permettant par exemple de raisonner sur le nombre d'instances d'un composant donné. Quant à la reconfiguration, elle est décrite par un ensemble de règles de réécriture dont la description est assez simple et compréhensible, et qui explicitent bien le changement topologique effectué. Néanmoins, ces règles ne permettent pas d'exprimer certaines conditions logiques tel que l'absence d'un lien de communication entre deux entités logicielles.

Dans [12], les auteurs considèrent qu'un style architectural inclut une spécification statique et une spécification dynamique. La partie statique définit l'ensemble des composants et des connecteurs possibles et la manière avec laquelle ces éléments peuvent être connectés ensemble. La partie dynamique spécifie la façon avec laquelle une architecture donnée peut évoluer suite à des reconfigurations planifiées ou à des changements non anticipés de l'environnement. Ils proposent donc de modéliser la partie statique par des digrammes de classe UML et la partie dynamique via des règles de transformation de graphe. Le modèle du style architectural assiste l'architecte pour décider si le style utilisé convient à une application donnée. Il spécifie alors une configuration initiale convenable à l'application selon le style architectural en question. Ensuite, le modèle dynamique du style est utilisé pour raisonner sur les configurations possibles et vérifier si toutes les configurations requises de l'application peuvent être atteintes.

Bettaz et al. [13] ont utilisé la notation Z et les transformations de graphe pour formaliser un style pour les réseaux cellulaires.

Dans ce papier, nous proposons une approche qui profite des avantages du pouvoir expressif des langages fonctionnels. Elle exploite également les privilèges qu'offre les grammaires de graphes, notamment en terme de manipulation d'architectures. Nous décrivons donc le style architectural d'un système logiciel avec la notation Z [14] et les opérations de reconfiguration via des règles de réécriture de graphe. Ces règles sont en fait exprimées via une intégration du langage fonctionnel Z et de la notation Δ [15] (notation inspirée des grammaires de graphes). Cette intégration possède l'avantage d'offrir au développeur une technique de spécification facile à appréhender tout en présentant un haut pouvoir d'expression. Ces règles prennent en considération les contraintes structurelles et fonctionnelles, définies pour le système, dans leurs conditions d'application assurant ainsi sa consistance durant son évolution. Nous décrivons

toute la sémantique en notation Z , obtenant ainsi une approche unifiée. La notation Z est l'un des rares langages formels accepté par l'industrie et les communautés Génie Logiciel et Architecture Logicielle. Nous pouvons ainsi utiliser un démonstrateur de théorèmes qui supporte les spécifications Z tels que $Z/EVES$ [16] et $Z-HOL$ [17].

Ce papier est organisé comme suit : dans la section 2, nous décrivons l'approche proposée et ses principaux éléments. La section 3 présente une étude de cas qui illustre notre approche. Elle explique également le processus de vérification qui doit être appliqué. Dans la section 4, nous donnons quelques remarques de conclusion et les perspectives de ce travail.

3 Approche proposée

Compte tenu de la pertinence des graphes dans la représentation des structures [18] et leur fondement mathématique, nous les avons adoptés pour représenter l'architecture d'un système logiciel de façon similaire aux approches de [11] où les noeuds représentent les composants du système et les arcs décrivent les liens de communication entre ces composants. Un graphe d'architecture représente une configuration donnée d'un système. Ce dernier peut être amené à évoluer en modifiant son architecture. En fait, ces configurations ou architectures représentent des instances du style architectural d'un système. Celui-ci définit les types de composants pouvant intervenir et les types de relations qui peuvent relier ces composants d'une part, ainsi que l'ensemble des propriétés architecturales qui doivent être satisfaites par toutes les configurations appartenant à ce style [7]. Nous donnerons dans ce qui suit comment spécifier un style architectural en Z à partir de la définition générale d'un graphe d'architecture. Nous nous intéresserons plus particulièrement aux graphes orientés, étiquetés et typés. En général, un graphe d'architecture s'écrit: $G = \langle N, L, E, T, f \rangle$ où:

1. $E \subseteq N \times L \times N$, est un ensemble d'arcs ;
2. N est un ensemble de noeuds ;
3. L est un ensemble d'étiquettes ;
4. $T = \{t1, t2, \dots, tn\}$ est un ensemble de types ;
5. $f: N \rightarrow T$ est une fonction qui associe à chaque noeud un seul type.

Partant de cette définition, le méta modèle Z associé est représenté par un schéma, qui contient deux parties: une partie de déclaration et une partie prédicat.

N.B. A ce niveau d'abstraction, nous n'avons pas besoin de considérer la représentation des noeuds de type ti ; nous les introduisons donc comme étant des types basiques de

spécification: $[N_t1, N_t2, \dots, N_tn]$. Ainsi, N_t1 , par exemple, dénote l'ensemble de noeuds de type $t1$.

$$\begin{array}{l} \text{system_name} \\ \hline N_1 : \mathbb{F} N_t1; N_2 : \mathbb{F} N_t2; \dots; N_n : \mathbb{F} N_tn \\ R_1 : N_ti \leftrightarrow l_j \leftrightarrow N_tk \\ R_2 : N_ts \leftrightarrow l_k \leftrightarrow N_tj \\ \dots \\ R_m : N_tu \leftrightarrow l_v \leftrightarrow N_ts \end{array}$$

Avec:

- $N = \bigcup_{i=1}^m N_i$
- $E = \bigcup_{i=1}^m R_i$
- $T = \{t1, t2, \dots, tn\}$
- $L = \{l_1, l_2, \dots, l_n\}$

Prenons par exemple le style architectural Producteur/Consommateur. La définition d'un graphe d'architecture appartenant à ce style est comme suit:

$$G = \langle N, L, E, T, f \rangle$$

où:

- $L = \{pushP, pushS, pullC, pullS\}$;
- $T = \{Producer, Service, Consumer\}$;
- $E = \bigcup_{i=1}^4 E_i$;
- $E_1 \subseteq \{x \in N \mid f(x) = Producer\} \times \{pushP\} \times \{y \in N \mid f(y) = Service\}$;
- $E_2 \subseteq \{x \in N \mid f(x) = Service\} \times \{pushS\} \times \{y \in N \mid f(y) = Consumer\}$;
- $E_3 \subseteq \{x \in N \mid f(x) = Consumer\} \times \{pullC\} \times \{y \in N \mid f(y) = Service\}$;
- $E_4 \subseteq \{x \in N \mid f(x) = Service\} \times \{pullS\} \times \{y \in N \mid f(y) = Producer\}$;

Partant de cette définition, la spécification de ce style en Z est donnée par le schéma suivant:

$$\begin{array}{l} \text{Prod_Cons}[N_Producer, N_Service, N_Consumer]. \\ \hline P : \mathbb{F} N_Producer \\ S : \mathbb{F} N_Service \\ C : \mathbb{F} N_Consumer \\ pushP : N_Producer \leftrightarrow N_Service \\ pushS : N_Service \leftrightarrow N_Consumer \\ pullC : N_Consumer \leftrightarrow N_Service \\ pullS : N_Service \leftrightarrow N_Producer \end{array}$$

Cette spécification constitue un schéma générique pour une architecture logicielle. Ainsi, $N_Producer$, $N_Service$ et $N_Consumer$ peuvent être instanciés avec n'importe quels ensembles de composants. Ce schéma peut être utilisé toutes les fois que nous souhaitons introduire deux objets qui sont reliés de cette manière.

Les quatre opérations fondamentales fournies par la plupart des langages sont la création et la suppression des composants et des connexions[19]. Les grammaires de graphes ont montré leur pertinence pour exprimer ces opérations de reconfiguration notamment dans les travaux décrits dans [19] et [11]. Nous proposons donc de les décrire via des règles de réécriture de graphe. Suite à l'étude effectuée sur les différentes notations graphiques proposées pour la réécriture de graphe (Y [20], X [21], Δ [15]), nous suggérons une nouvelle notation qui profite des avantages de la description visuelle et qui couvre les limites que nous avons soulevées et que nous détaillerons dans la suite. Notre description se compose de deux parties : une partie structurelle et une partie fonctionnelle. La partie structurelle est exprimée par la notation Δ qui est composée de cinq sections comme le montre la figure 1.

- *Rétraction* : le fragment de graphe enlevé durant la réécriture.
- *Insertion* : le fragment qui est créé et connecté dans le graphe hôte durant la réécriture.
- *Contexte requis* : le fragment identifié mais non modifié durant la réécriture.
- *Garde* : exprime des conditions sur les étiquettes qui apparaissent dans le graphe de la règle.
- *Restriction* : le fragment de graphe qui doit être absent pour que la réécriture ait lieu. Si le sous graphe correspondant au contexte et la rétraction peut être étendu pour correspondre à la restriction, alors la règle ne peut pas être appliquée à ce sous graphe.

Une règle r est appliquée à un graphe g selon les étapes suivantes:

1. Un sous graphe isomorphe à $g_l = \text{contexte requis} \cup \text{retraction}$ est identifié dans g .
2. S'il n'existe pas de sous graphe isomorphe ou si le sous graphe isomorphe peut être étendu pour correspondre à g_l plus la Restriction (si elle existe), r ne peut pas être appliquée à g .
3. La garde, si elle existe, est évaluée. si elle est évaluée à faux, r ne peut pas être appliquée à g .

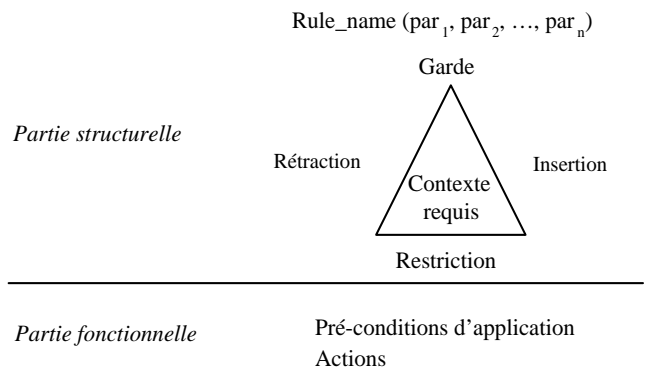


Figure 1. Notation mixte proposée

4. Les éléments du sous graphe isomorphe à la Rétraction sont retirés de g .
5. Un graphe isomorphe à l'Insertion est attaché au graphe hôte g via les arcs décrits entre le Contexte et l'Insertion dans r .

Quant à la partie fonctionnelle, elle exprime des pré-conditions d'application qui ont pour objectif le contrôle de l'évolution conformément aux propriétés du système. Elle exprime également les actions d'ordre fonctionnel à exécuter si la règle s'avère applicable. Nous exprimons cette partie en notation Z qui est basée sur la théorie des ensembles et la logique de prédicat de premier ordre.

L'introduction de la partie fonctionnelle permet de résoudre les limites présentes dans les notations de transformation de graphes notamment leur pouvoir expressif en ce qui concerne les conditions et les actions non structurelles d'application. En effet, ces notations ne peuvent pas exprimer des conditions sur les valeurs des attributs des noeuds du graphe par exemple ou des conditions faisant intervenir la disjonction et toute autre contrainte fonctionnelle. D'autre part, le fait de spécifier tout le système en utilisant les graphes uniquement entraîne dans certains cas l'augmentation de la complexité du graphe de la règle ce qui compliquerait ce graphe de point de vue lisibilité et compréhension, entravant de ce fait l'avantage principal de ces notations. La partie fonctionnelle permet au contraire d'exprimer par une simple expression logique, une condition assez compliquée si exprimée graphiquement. Nous proposons de décrire la sémantique de nos descriptions par le langage Z . Ainsi, chaque règle est traduite par un schéma Z comme suit :

| |
|---|
| $Rule_name$ $par_1?; par_2?; \dots; par_n?$ $\Delta system_name$ |
| <i>Pre – conditions :</i> <i>contraintes fonctionnelles : formules sur les parametres, formules sur les noeuds attribut, etc.</i> <i>contraintes structurelles : formules sur les relations : identification de $g_l = contexte requis \cup retraction$ dans le graphe du systeme</i> <i>Actions :</i> <i>Operations sur les ensembles des noeuds/arcs (insertion/retraction)</i> <i>Actions fonctionnelles</i> |

Où $Par_i?$ représentent les paramètres d'entrée de la règle et $\Delta system_name$ indique que la règle peut changer l'état du système donné par le schéma $system_name$.

L'application d'une règle de transformation s'effectue alors comme suit : si les contraintes fonctionnelles sont vérifiées et si les contraintes structurelles sont également satisfaites (existence du graphe décrit dans $g_l = contexte requis \cup retraction$ et l'absence du graphe décrit dans la partie Restriction) alors les actions peuvent être exécutées (insertion et/ou rétraction des noeuds et/ou arcs et autres actions fonctionnelles).

4 Etude de cas

Pour détailler davantage notre approche, nous présentons dans cette section un exemple simple: le système de contrôle de patients PMS (Patient Monitoring System) qui a été utilisé pour illustrer les travaux de [11] et [22]. Pour représenter l'architecture de communication de ce système, nous avons choisi le style architectural Producteur/Consommateur.

A chaque service de la clinique (pédiatrie, cardiologie, maternité,) est associé un service d'événement pour gérer les communications entre les infirmières et les contrôleurs de lit rattachés au service en question. Chaque infirmière demande des informations relatives à ses patients en envoyant une requête au service d'événement auquel elle est liée. Ce service prend en charge cette demande et la transmet aux contrôleurs de lit des patients concernés. Lorsque l'état d'un patient est jugé anormal, son contrôleur de lit envoie un signal d'alarme au service d'événement auquel il est connecté. Ce service transmet alors ce signal à l'infirmière responsable. L'infirmière joue ainsi le rôle du composant consommateur et le contrôleur de lit celui du composant producteur.

4.1 Spécification du système

En plus des contraintes du style architectural, une application peut avoir des propriétés spécifiques qui doivent être toujours satisfaites durant l'évolution de son architecture. Nous allons prendre quelques propriétés du système PMS telles que:

- Le système doit contenir au maximum 3 services.
- Un service contient au maximum 5 infirmières et 15 patients.
- Un patient doit être toujours affecté à un et un seul service et ce service doit contenir au moins une infirmière pour pouvoir s'occuper de ce patient.
- Une infirmière doit être attachée à un seul service.

La spécification du système doit donc considérer les contraintes du style producteur/Consommateur et les propriétés spécifiques citées. Dans la notation Z , on peut combiner les informations contenues dans deux schémas en incluant un schéma dans la partie déclaration de l'autre. Les déclarations sont fusionnées et les prédicats sont conjoints. La spécification est donc comme suit: $[BED_MONITOR, EV_SER, NURSE]$

| |
|--|
| $PMS1$ $Prod_Cons[BED_MONITOR, EV_SER, NURSE]$ |
|--|

| |
|--|
| PMS $PMS1[PB/P, ES/S, CN/C]$ $NB_CN : EV_SER \rightarrow \mathbb{N}$ $NB_PB : EV_SER \rightarrow \mathbb{N}$ $\#ES \leq 3$ $\forall s : ES \bullet NB_CNs \leq 5 \wedge NB_PBs \leq 15$ $\forall x : PB \bullet$ $\exists_1 s : ES \bullet (x, s) \in pushP \wedge (s, x) \in pullS$ $\wedge NB_CNs \geq 1$ $\forall s, z : ES; y : CN \bullet$ $(y, s) \in pullC \wedge (s, y) \in pushS \wedge (y, z) \in pullC$ $\wedge (z, y) \in pushS \Rightarrow s = z$ |
|--|

Notez que $N_Producer$, $N_Service$ et $N_Consumer$ sont instanciés respectivement par $BED_MONITOR$, EV_SER et $NURSE$ et nous avons renommé également quelques variables du schéma $Prod_Cons[N_Producer, N_Service, N_Consumer]$ pour faciliter leur utilisation; En notation Z , si $Schema$ est un schéma, alors on écrit $Schema[new/old]$.

La fonction NB_CN calcule pour un service donné le nombre d'infirmières connectées et la fonction NB_PB donne pour un service donné le nombre de patients affectés.

Il est donc évident que le graphe de la figure 2 est une instance possible du système *PMS*.

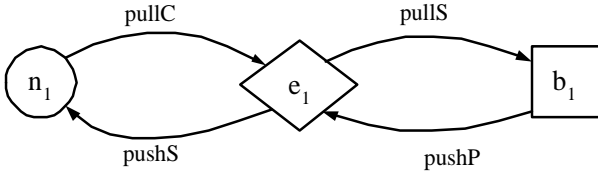


Figure 2. une configuration possible du système *PMS*

4.2 Spécification de l'évolution

Dans cette section, nous présenterons la spécification des différentes règles permettant de faire évoluer notre système *PMS* tout en tenant compte des propriétés décrites.

- *Insertion d'un service d'événement*: cette règle permet d'insérer une instance de composant de type *EV_SER* à condition que le système ne contient pas déjà trois services d'événement comme l'impose le premier prédicat du schéma *PMS*. La représentation syntaxique de cette règle est donnée par la figure 3 et la sémantique par le schéma *insert_ES*.

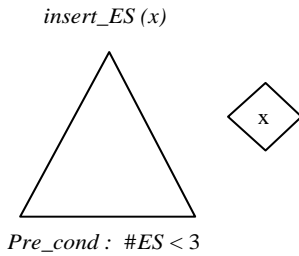
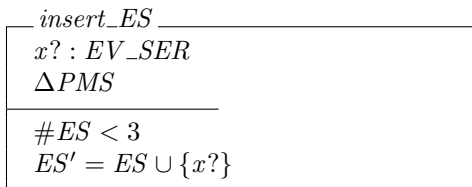


Figure 3. Règle d'insertion d'un service d'événement



Si nous appliquons cette règle au graphe de la figure 2 avec e_2 comme paramètre, nous obtiendrons le graphe donné par la figure 4.

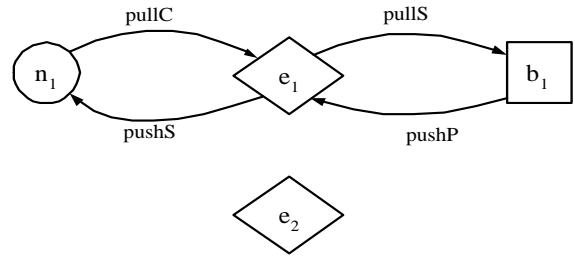


Figure 4. Graphe obtenu suite à l'application de la règle *insert_ES* au graphe de la figure 2

- *Insertion d'un patient*: l'insertion d'un nouveau patient dans le système se traduit par l'insertion d'un nouveau contrôleur de lit. Ainsi, cette règle permet d'insérer une instance de composant de type *BED_MONITOR* et de la lier à un service d'événement sous deux conditions : il faudrait d'abord qu'il y ait au moins une infirmière appartenant à ce service pour pouvoir s'occuper du nouveau patient. En plus, il faudrait vérifier que ce service ne contient pas déjà quinze patients, et ce conformément au deuxième prédicat du schéma *PMS*. La représentation syntaxique de cette règle selon la notation purement graphique est donnée par la figure 5. Cette représentation complique considérablement le graphe et rend difficile la compréhension de la règle. Tandis qu'une simple expression logique nous épargne cette complexité et facilite la tâche de spécification comme le montre la figure 6. De plus, les actions d'ordre fonctionnel peuvent être exprimées. La sémantique est spécifiée par le schéma *insert_PB*.

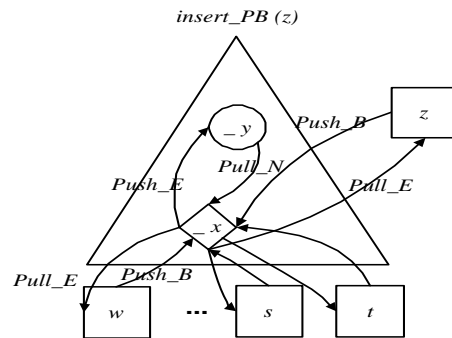


Figure 5. Règle d'insertion d'un patient (notation Δ)

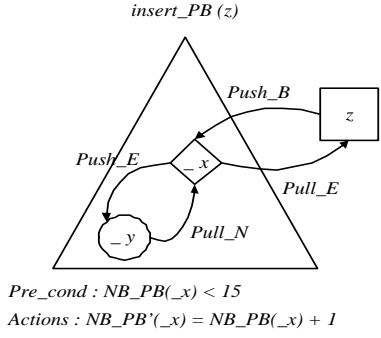
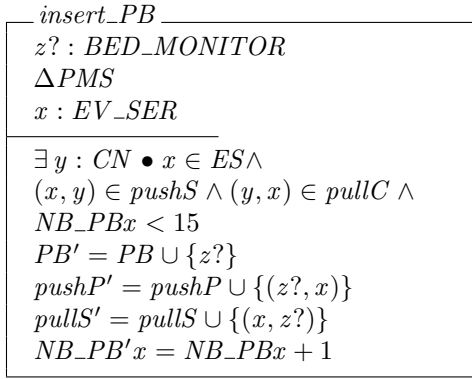


Figure 6. Règle d'insertion d'un patient (notre notation)



L'application de cette règle au graphe de la figure 4 avec b_2 comme paramètre, génère le graphe d'architecture donné par la figure 7.

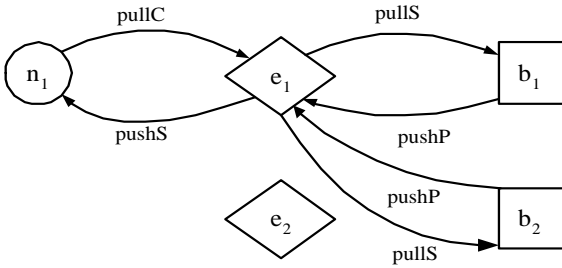


Figure 7. Graphe obtenu suite à l'application de la règle *insert_PB* au graphe de la figure 4

- *Insertion d'une infirmière*: cette règle permet d'insérer une instance de composant de type *NURSE* et de la lier à un service d'événement quelconque (devant exister dans le système) via les liens précisés dans la partie déclaration du schéma *PMS*. Pour appliquer cette règle, on doit vérifier que le service en question ne contient pas déjà cinq infirmières conformément au

deuxième prédicat du schéma *PMS*. La représentation syntaxique est donnée par la figure 8 et la sémantique est exprimée avec le schéma *insert_CN*.

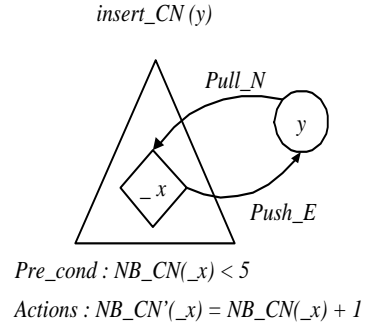
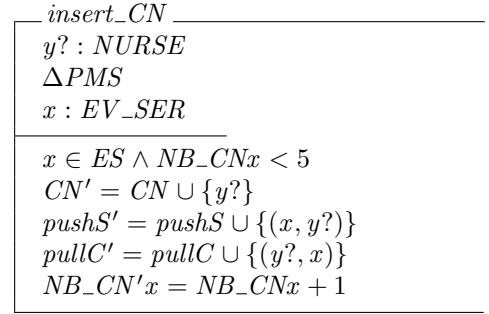


Figure 8. Règle d'insertion d'une infirmière



Si nous appliquons cette règle au graphe de la figure 7 avec n_2 comme paramètre, il y aura deux choix possibles : l'infirmière peut être affectée au service e_1 ou au service e_2 ; le choix est non déterministe. On peut obtenir, par exemple, le graphe donné par la figure 9.

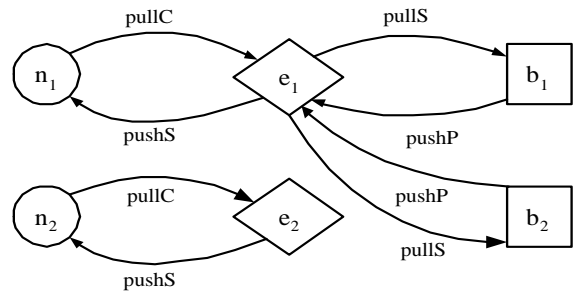


Figure 9. Graphe obtenu suite à l'application de la règle *insert_CN* au graphe de la figure 7

- *Suppression d'une infirmière*: une infirmière peut quitter le système si elle n'a aucun patient en charge. Cette

règle supprime donc un composant de type *NURSE* si et seulement s'il n'est pas connecté à un service contenant des patients et ne contenant pas d'autres infirmières. La syntaxe est donnée par la figure 10 et la sémantique par le schéma *supp_CN*.

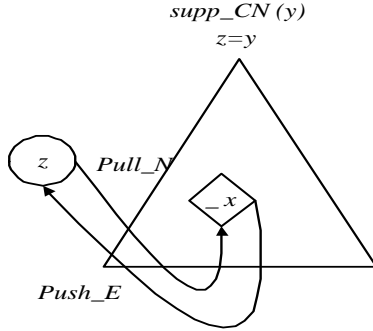


Figure 10. Règle de suppression d'une infirmière

| |
|---|
| $\frac{\text{supp_CN}}{y? : NURSE}$ ΔPMS |
| $y? \in CN$ $\forall x : ES \bullet (y?, x) \notin pullC \wedge (x, y?) \notin pushS$ $CN' = CN \setminus \{y?\}$ |

Si on voudrait par exemple appliquer cette règle au graphe de la figure 9 avec n_1 comme paramètre, nous remarquerons que le deuxième prédicat de la règle n'est pas vérifié ; la règle est par conséquent inapplicable.

- *Suppression d'un patient*: lorsqu'un patient quitte le système, le contrôleur de lit correspondant est déconnecté du service en question et supprimé du système. La syntaxe de cette règle est exprimée par la figure 11 et la sémantique par le schéma *supp_PB*.

| |
|---|
| $\frac{\text{supp_PB}}{z? : BED_MONITOR}$ ΔPMS $x : EV_SER$ |
| $z? \in PB$ $x \in ES \wedge (x, z?) \in pullS \wedge (z?, x) \in pushP$ $PB' = PB \setminus \{z?\}$ $pushP' = pushP \setminus \{(z?, x)\}$ $pullS' = pullS \setminus \{(x, z?)\}$ $NB_PB'x = NB_PBx - 1$ |

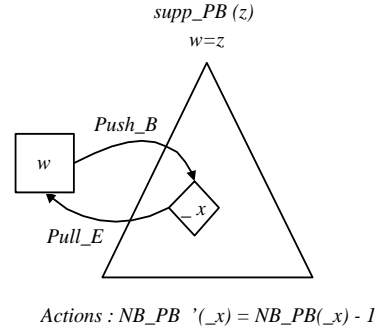


Figure 11. Règle de suppression d'un patient

- *Déconnexion d'une infirmière*: une infirmière peut quitter son service (sans pour autant être supprimée du système) seulement si ce service ne contient pas de patients ou alors il contient d'autres infirmières qui pourraient s'occuper des patients. Ainsi, cette règle ne permet de déconnecter un composant de type *NURSE* que si l'une des deux conditions citées est satisfaite. La représentation syntaxique de cette règle selon notre notation est donnée par la figure 12. Nous pouvons relever dans ce cas précis une autre limite de la notation purement graphique : ce qui est exprimé simplement par l'opérateur de disjonction ne peut pas être exprimé graphiquement.

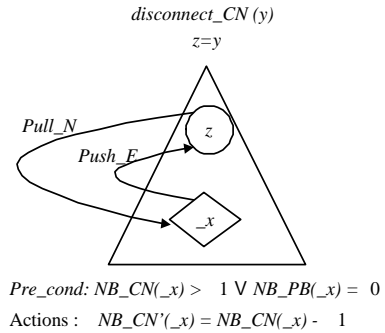


Figure 12. Règle de déconnexion d'une infirmière

Nous exprimons la sémantique avec le schéma *disconnect_CN*.

| |
|---|
| $\frac{\text{disconnect_CN}}{y? : NURSE}$ ΔPMS $x : EV_SER$ |
| $y? \in CN \wedge x \in ES \wedge$ $(x, y?) \in pushS \wedge (y?, x) \in pullC$ $NB_CNx > 1 \vee NB_PBx = 0$ $pushS' = pushS \setminus \{(x, y?)\}$ $pullC' = pullC \setminus \{(y?, x)\}$ $NB_CN'x = NB_CNx - 1$ |

- *Connexion d'une infirmière:* Il faudrait d'abord vérifier que l'infirmière est libre, autrement dit, elle n'est pas attachée à un service. En plus, elle ne pourrait être connectée à un service que si ce dernier ne contient pas déjà cinq infirmières conformément au deuxième prédicat du système. La représentation syntaxique selon notre notation est donnée par la figure 13 ; notons qu'il s'agit ici de la même remarque de complexité faite pour la règle 5. La sémantique est spécifiée par le schéma *connect_CN*.

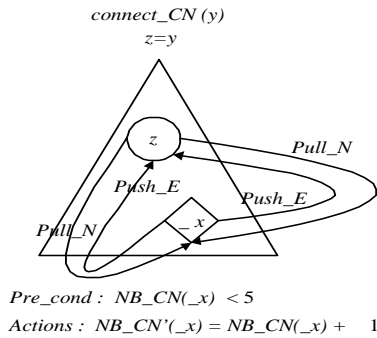


Figure 13. Règle de déconnexion d'une infirmière

| |
|--|
| $\frac{\text{connect_CN}}{y? : NURSE}$ ΔPMS $x : EV_SER$ |
| $y? \in CN$ $\forall z : ES \bullet (y?, z) \notin pullC \wedge (z, y?) \notin pushS$ $x \in ES \wedge NB_CNx < 5$ $pushS' = pushS \cup \{(x, y?)\}$ $pullC' = pullC \cup \{(y?, x)\}$ $NB_CN'x = NB_CNx + 1$ |

En conclusion, nous pouvons remarquer la pertinence des grammaires de graphe en ce qui concerne la façon avec laquelle l'architecture est manipulée.

4.3 Vérification des propriétés

Dans cette section, nous allons montrer que la spécification proposée préserve la consistance du système et assure sa sécurité durant son évolution. Pour ce faire, nous devons prouver le théorème suivant:

$$\forall c, c' \in Conf, \forall r \in Rules. (c \text{ sat } P) \wedge (c \xrightarrow{r} c') \Rightarrow c' \text{ sat } P$$

Informellement, ce théorème peut être traduit comme suit: c étant une configuration satisfaisant les propriétés requises P , et r étant une règle pouvant être appliquée sur c , alors la nouvelle configuration c' obtenue suite à l'application de r sur c , satisfait à son tour les dites propriétés. Pour cela il suffit d'identifier pour chaque propriété les règles susceptibles de générer des architectures qui violent les propriétés correspondantes.

Prenons par exemple la propriété qui indique qu'un service contient au maximum cinq infirmières. Les seules règles pouvant éventuellement enfreindre cette propriété sont celles qui relient une infirmière à un service. Les règles concernées sont donc *insert_CN* et *connect_CN*. Or celles-ci possèdent la condition suivante : $Nb_CN(_x) < 5$, et qui les empêche d'être exécutées si le service en question contient déjà cinq infirmières. Ainsi la propriété ne sera jamais violée.

Parmi les contraintes du *PMS*, il existe une qui stipule qu'un patient doit être toujours affecté à un service. Seules les règles qui manipulent les connexions des composants de type *BED_MONITOR* avec les composants de type *EV_SER* peuvent violer cette propriété. Dans notre cas, il s'agit des règles *insert_PB* et *supp_PB*. Or, la règle *insert_PB* permet d'insérer une instance de composant de type *BED_MONITOR* tout en la connectant à un service. Quant à la règle *supp_PB* elle permet de supprimer complètement une instance *BED_MONITOR*. Ainsi, cette contrainte ne risque pas d'être violée.

Nous pouvons vérifier également la contrainte qui stipule que le service auquel est affecté le patient doit contenir au moins une infirmière. Le risque réside dans les règles pouvant supprimer ou déconnecter une infirmière ou encore connecter un composant *BED_MONITOR* à un service. Les règles concernées sont donc: *insert_PB*, *supp_CN* et *disconnect_CN*. Or la règle *insert_PB* n'affecte un contrôleur de lit à un service que si ce dernier contient au moins une infirmière, ce qui est conforme à la propriété. La règle *supp_CN* ne supprime une infirmière que si elle n'est pas connectée à un service contenant des patients et ne contenant pas d'autres infirmières. Enfin, la règle *disconnect_CN* ne procède à la déconnexion d'une infirmière de son service que si ce dernier ne contient pas de patients ou alors il contient au moins une autre infirmière. De ce fait, la propriété est effectivement préservée.

5 Conclusion

Nous avons présenté dans cet article une approche de spécification formelle des architectures dynamiques des systèmes orientés composants. Cette approche repose sur une intégration des sémantiques basées sur les graphes dans le framework de la notation Z . Ceci facilite la tâche du développeur en lui offrant une technique de spécification facile à appréhender tout en lui permettant de raisonner rigoureusement sur les styles architecturaux. De plus, notre approche permet de décrire formellement la dynamique d'une architecture logicielle via des règles de réécriture de graphe. Ces règles prennent en considération les propriétés spécifiant les contraintes pour appliquer des opérations de reconfiguration, ce qui assure la consistance du système durant son évolution. Nous utilisons actuellement $Z/EVES$ [16], un outil qui supporte la notation Z , pour spécifier les architectures logicielles et raisonner sur leur dynamique. En perspective, nous envisageons de prendre en considération la description de l'aspect comportemental des composants logiciels. Pour cela nous sommes en cours d'étudier l'intégration d'un langage de calcul de processus avec la notation Z .

References

- [1] J. Goguen. The dry and the wet. Technical Report Monograph PRG-100, Programming Research Group, Oxford University Computer Laboratory England.
- [2] R.F. Gamble, P.R. Stiger, and R.T. Plant. Rule-based systems formalised within a software architectural style. *Elsevier Knowledge-Based Systems*, 12:13–26, 1999.
- [3] R.T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, pages 43–52, 1997.
- [4] J. Dulay, N. Kramer, and J. Magee. Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2):73–82, Mars 1993.
- [5] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE transactions on Software Engineering*, 21(9):717–734, Septembre 1995.
- [6] *Guide to the Rapide 1.0 Language Reference Manuals*. Rapide Design Team Program Analysis and Verification Group Computer Systems Lab Stanford University, 1997.
- [7] R. Allen, R. Douence, and D. Garlan. Specifying and analysing dynamic software architectures. *Journal of Fundamental Approaches to Software Engineering*, 1382:21–37, 1998.
- [8] C. Canal, E. Pimentel, and J.M. Troya. Specification and refinement of dynamic software architectures. *Journal of Software Architecture*, pages 107–125, Avril 1999.
- [9] N. Aguirre and T. Maibaum. A temporal logic approach to component-based system specification and reasoning. In *5th ICSE Workshop on component-based software engineering*, Orlando, Florida, USA, Avril 2002.
- [10] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.
- [11] D. Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, Juillet 1998.
- [12] L. Baresi, R. Heckel, S. Thone, and D. Varro. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE 2003*. Helsinki. Finland, ACM Press (2003), 2003.
- [13] M. Bettaz, M. Maouche, and R. Heckel. Graph transformation: A semantic framework for mobile z. In *ETAPS-WADT04*. Barcelona Spain, Mars 2004.
- [14] J. R. Abrial. *Programming as a mathematical exercise*. In C.A.R. Hoare, editor, *Mathematical Logic and Programming Languages*. Prentice-Hall International, 1985.
- [15] S. M. Kaplan, J. P. Loyall, and S. K. Goering. Specifying concurrent languages and systems with δ -grammars. *Research Directions in Concurrent Object-Oriented Programming*, 1993.
- [16] $Z/eves$. <http://www.ora.on.ca/Z-eves>.
- [17] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of z in isabelle/hol. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag, 1996.
- [18] Appligraph. applications of graph transformations. Technical Report 22565, Hans-jörg Kreowski and Detlef plump editors, Mai 2001.
- [19] M.A. Wermelinger. *Specification of software architecture reconfiguration*. PhD thesis, Université Nova de Lisbon, Septembre 1999.

- [20] H. Göttler. Attributed graph grammars for graphics. graph grammars and their application to computer science. *Computer Sciences*, 153:130–142, 1982.
- [21] H. Göttler. Diagram editors = graphs + attributes + graph grammars. *International Journal of Man-Machine Studies (IJMMS)*, 37:481–502, 1992.
- [22] I. Warren and I. Sommerville. Dynamic configuration abstraction. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 173–190. Springer-Verlag, 1995.