REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE M'HAMED BOUGARA-BOUMERDES

Faculté des **S**ciences

Mémoire de Magister

Présenté par

BOUDANE Fatima

En vue de l'obtention du **diplôme de magister** en **:**

Filière : Systèmes Informatiques et Ingénierie des logiciels
Option : Spécification Logiciels et Traitement de l'Information

---

Spécification formelle des systèmes adaptatifs

Devant le jury :

Mr. AHMED NACER	Mohamed	Prof	USTHB	Président
Mr. ABDELLI	Abdelkrim	MC/A	USTHB	Examinateur
Mr. AIT BOUZIAD	Ahmed	MC/B	UMBB	Examinateur
Mr. MEZGHICHE	Mohamed	Prof	UMBB	Encadreur

Année Universitaire : 2012/2013

Remerciements

Tout d'abord, je tiens à remercier le bon Dieu pour m'avoir illuminée et menée jusqu'ici.
Nombreux sont ceux qui m'ont aidé, encouragé, soulagé, tout au long de ces années, et je ne saurais leur exprimer mes remerciements autant que je le souhaiterais ; Merci pour leur présence, leurs discussions, leurs conseils et suggestions éclairées.

En premier lieu, je tiens à remercier chaleureusement Monsieur Mohammed MEZGHICHE, Professeur à l'UMBB d'avoir accepté d'être mon directeur de mémoire. Je le remercie pour sa disponibilité, sa compétence, son précieux soutien, ses conseils judicieux et la confiance dont il m'a fait part lors de la réalisation de ce travail.

Je souhaite adresser mes sincères remerciements aux membres de mon jury de soutenance :
Monsieur Mohammed AHMED NACER, de m'avoir accordé l'honneur de présider mon jury de soutenance, Monsieur Abdelkrim ABDELLI et Monsieur Ahmed AIT BOUZIAD, d'avoir accepté la tâche dévaluer, en qualité d'examinateurs, mon travail.

Mes profonds et mes plus grands remerciements vont aux membres de ma famille : aux deux personnes qui me sont les plus chères au monde ; mes parents, à mes sœurs et frères. Merci pour être toujours à côté de moi. Merci du fond du cœur.

Table des Matières

CHAPITRE 3. Méthodes formelles et sûreté de fonctionnement

PARTIE II. Contributions

CHAPITRE 4. Modélisation des reconfigurations dynamiques

CHAPITRE 5. Vérification et validation du modèle

Liste des figures

Résumé

Les systèmes logiciels adaptatifs modifient leur comportement seuls à travers des opérations de reconfigurations dynamiques telles que l'insertion, la suppression et le remplacement de composants et/ou de connexions entre ces composants en utilisant des mécanismes de reconfiguration.

Afin d'assurer la correction du comportement de ces systèmes pendant et après l'adaptation, nous proposons dans ce mémoire une méthode de modélisation et de vérification formelle permettant au concepteur de modéliser ces systèmes, de spécifier ses propriétés et de les vérifier.

La méthode proposée s'articule autour de deux approches. Dans la première, nous proposons un modèle pour représenter un système adaptatif par composant, en utilisant le formalisme de transformation de graphe et le formalisme fonctionnel, tout avec prise en compte des contraintes imposées au système (invariant, pré\post-conditions). Quant à la deuxième approche, elle permet de transformer le modèle semi-formel, construit par la première approche, en une spécification formelle exprimée avec le langage Event-B. Nous automatisons cette transformation par la proposition de certaines règles.
Pour valider notre spécification et garantir sa cohérence, nous utilisons le système de preuve de l'atelier B.

Mots Clés: *systèmes adaptatifs, reconfigurations dynamiques, transformation de graphe, formalisme fonctionnel, vérification formelle, cohérence.*

Abstract

Self-adapting software systems adapt their behavior alone by the means of dynamic reconfigurations operations such as insertion, suppression and replacement of components and/or connections between these components by using reconfiguration mechanisms.

In order to ensure the correction of the behavior of these systems during and after the adaptation, we propose in this document a method of modeling and formal verifying allowing the designer to model these systems, to specify its properties and to verify them.

The proposed method is articulated around two approaches. In the first, we propose a model to represent an adaptive system of component by using the graph transformation formalism and the functional formalism, all with taking into account the constraints imposed on the system (invariant, pre\post-conditions). As for the second approach, it makes it possible to transform the semi-formal model built by the first approach into a formal specification expressed with the formal language Event-B. We automate this transformation by the proposal of certain rules.
To validate our specification and to guarantee its coherence, we use the proof system of the Atelier-B.

Keywords: *self-adapting systems, dynamic reconfigurations, graph transformation, functional formalism, formal verification, coherence.*

ملخص

ان البرامج المتكيفة مع العوامل المحيطة بها تقوم بتغيير سلوكها لوحدها من خلال عمليات إعادة تشكيل ديناميكية مثل إدراج، حذف واستبدال مكونات و / أو وصلات بين هذه المكونات، باستخدام آليات إعادة التشكيل.

لضمان السلوك الصحيح لهذه البرامج أثناء وبعد التغيير، نقترح في هذه المذكرة طريقة لنمذجتها والتحقق رسميا من صحتها، مما يتيح للمصمم بتمثيل خصائصها وبرهنة موافقتها لها.

وتستند هذه الطريقة المقترحة على نهجين. في الأول، نقترح نموذجا لتمثيل النظام المتكيف بالعناصر، وذلك باستخدام الرسم البياني و الدوال، مع الأخذ في الاعتبار القيود المفروضة على النظام (ثابتة، قبل \ بعد التغيير). اما بالنسبة للنهج الثاني، فإنه يسمح بتحويل النموذج شبه الرسمي الذي بناه النهج الأول، الى نموذج دقيق Event-B
لتسهيل هذه العملية، نقترح مجموعة من القواعد.
للتحقق من صحة النموذج Event-B نستخدم نظام Atelier B

***كلمات البحث**: البرامج المتكيفة، إعادة تشكيل ديناميكية، الرسم البياني، الدوال، التحقق الرسمي، برهنة دقيقة*

Introduction générale

Les systèmes informatiques actuels s'exécutent dans des contextes qui varient beaucoup au cours du temps. Ces systèmes doivent s'adapter donc en réponse à ces changements. Cette adaptation (modification) peut concerner la correction d'erreurs de conception « bugs » dans le système, l'ajout de nouvelles fonctionnalités ou aussi l'optimisation de fonctionnalités existantes [LEG09].

En général, pour modifier une application, cette dernière doit être arrêtée, modifiée, recompilée puis démarrée à nouveau. Cependant, ce processus classique de maintenance ne peut pas être appliqué à certains systèmes (tels que les systèmes de contrôle du trafic aérien, de transaction financière…etc.) dont la continuité de services est nécessaire [KET04]. Ces systèmes doivent être adaptés d'une manière dynamique pour assurer le minimum de services avec une meilleure performance.
L'adaptation dynamique est une technique avec laquelle un système peut être adapté sans arrêter son exécution [STY07].

Un système adaptatif peut répondre automatiquement aux besoins de l'adaptation. La construction de ce type de systèmes, repose sur des architectures dynamiquement reconfigurables et nécessite des règles pour la modification dynamique de la configuration et son déploiement par l'ajout et/ou la suppression de nouveaux composants et services (interfaces), ainsi que la modification des connexions entre ces derniers. L'application de ces règles se fait conformément aux besoins soutenus par le système [BHA07]. Le système nécessite dans ce cas un mécanisme de reconfiguration dynamique qui va accroitre sa complexité. Par conséquent, il faudrait garantir sa cohérence pendant et après l'adaptation.

La garantie de correction des systèmes a été apportée par les méthodes formelles. Ces méthodes proposent un cadre formel pour décrire explicitement le comportement attendu d'un système : sa spécification formelle. Ensuite, une démarche de preuve est proposée pour établir par un raisonnement mathématique la cohérence de la spécification formelle [LOU09].

Pour la conception d'un système adaptatif cohérent, plusieurs questions doivent avoir une réponse, à savoir :

- Quel est le comportement attendu après l'adaptation? (post-conditions)
- Quelles contraintes existent pour que les adaptations se produisent? (pré-conditions)
- Quelles sont les propriétés qui doivent être préservées durant et après l'adaptation? (invariant)

La spécification d'un système adaptatif nécessite de prendre en compte toutes ces questions. Nous pouvons donc répondre avec précision à ces questions par une spécification formelle.

En effet, la conception et la maintenance de systèmes adaptatifs est une tâche complexe. Pour maîtriser cette complexité, il faut disposer d'un cadre de spécification formelle (méthodes et outils) abordable par les non spécialistes en techniques de description formelle. Ce cadre doit permettre de modéliser ces systèmes, de spécifier les règles d'adaptation, d'exprimer des

propriétés et de les vérifier. Dans ce mémoire, nous avons étudié d'abord les diverses propositions définies dans le cadre de ces problèmes. Partant des points forts et aussi des inconvénients de ces propositions, nous proposons une approche de modélisation des systèmes adaptatifs, facile à utiliser et compréhensible. Par la suite, nous montrons comment transformer cette modélisation en une spécification formelle B, étendue et vérifiable automatiquement grâce à l'atelier B, pour la production de systèmes adaptatifs corrects. Nous utilisons un exemple d'application pour illustrer comment cette méthode est employée pour modéliser les systèmes adaptatifs et assurer la correction de leurs comportement.

Ce document est organisé comme suit :

Le chapitre 1 est une introduction sur les systèmes adaptatifs et le concept d'adaptation dynamique. Nous introduisons tout d'abord, la définition de la notion d'adaptabilité des applications. Ensuite, nous décrivons d'une manière détaillée les différentes propriétés des systèmes adaptatifs.

Le chapitre 2 présente les différents concepts liés aux architectures logicielles qui sont utilisés pour la mise en œuvre de l'adaptabilité des systèmes logiciels. Ensuite, il analyse le problème de la fiabilité des mises à jour dynamiques liée à ces architectures.

Le chapitre 3 comporte deux parties. Nous décrivons dans la première partie, les différentes méthodes formelles utilisées pour la spécification et la vérification des systèmes informatiques. Dans la deuxième partie, nous présentons les principaux travaux proposés appliquant les méthodes formelles pour la spécification et la validation des reconfigurations dynamiques des logiciels.

Le chapitre 4 présente la première approche de notre solution qui permet la modélisation d'un système adaptatif par composants.

Le chapitre 5 concerne la proposition d'une approche de vérification et de validation composée de deux parties. Nous présentons tout d'abord, la première partie qui permet d'assurer une transformation de notre modèle proposé dans le chapitre 4 vers le langage formel Event-B. Ensuite nous passons à la deuxième partie qui permet de vérifier la consistance du système initialement et la conformité de l'évolution de son architecture par rapport à ses contraintes structurales et fonctionnelles.

Dans **la conclusion générale** nous synthétisons d'abord le travail réalisé. Par la suite nous présentons quelques perspectives que nous comptons explorer pour enrichir notre méthode et pouvoir l'exploiter.

Partie I : Etat de l'art

CHAPITRE 1

Les systèmes logiciels adaptatifs

1.1. Introduction

Les systèmes informatiques s'exécutent dans des contextes de plus en plus variables [DAV05] [XAV10]. C'est pourquoi, ils doivent êtres évolutifs. Cette évolution (modification) peut concerner la correction d'erreurs de conception « bugs » dans le système, l'ajout de nouvelles fonctionnalités, ou aussi l'optimisation de fonctionnalités existantes [LEG09].

Nous distinguons deux types d'adaptation selon l'état du cycle de vie du système considéré. Les adaptations statiques qui interviennent quand le système est arrêté, et les adaptations dynamiques réalisées dans un système en exécution [LEG09].

Ce chapitre présente un aperçu sur les systèmes adaptatifs et l'adaptation dynamique. Il introduit tout d'abord, la définition de la notion d'adaptabilité des applications. Par la suite, il enrichit cette définition par l'étude des points suivants :

- Nécessité de l'adaptation dynamique dans les systèmes informatiques.
- Comment et à quel moment l'adaptation dynamique est réalisée ?

1.2. Définitions et terminologies

Cette section introduit la définition des principales notions qui seront utilisées dans la suite de ce document.

1.2.1. Système informatique

Correspond à l'ensemble des éléments que l'on veut l'adapter pour que l'application s'exécute avec les meilleures performances. Les autres éléments qui interagissent avec le système constituent le contexte d'exécution.

1.2.2. Contexte d'exécution

Une comparaison des définitions de certains chercheurs comme celle de Schilit donnée dans [SCH94] et [SAW94] a conduit à une définition d'ordre général donnée dans [DEY99] :
« Le contexte constitue n'importe qu'elle information pouvant être utilisée pour caractériser la situation d'une entité. Une entité peut être une personne, un lieu, ou un objet pertinent pour l'interaction entre l'utilisateur et l'application, y compris l'utilisateur et l'application ».

Le contexte d'exécution correspond donc à l'ensemble des éléments qui influencent le système lors de son exécution. Ce contexte prend en compte à la fois l'environnement physique (matériel et logiciel) et aussi l'attente des utilisateurs.

1.2.3. Le critère de performance

C'est ce qui permet d'évaluer l'adéquation du système vis-à-vis de son contexte d'exécution. Il est lié à la fonction du système et au besoin de l'utilisateur. Il peut être composé de plusieurs objectifs [XAV10].

1.2.4. Notions d'adaptation

Pour que la notion d'adaptation soit précise, nous étudions tout d'abord l'idée générale grâce à des exemples dans quelques domaines. Puis nous présentons sa définition et utilité.

1.2.4.1. Idée générale grâce à des exemples

Historiquement, c'est dans le contexte des environnements mobiles que les premières techniques permettant aux applications de s'adapter sont apparues [AND00]. Dans ce contexte, la principale variation de l'environnement d'exécution qui peut se produire concerne la connexion réseau. Les caractéristiques de cette connexion en termes de débit et de latence sont différentes, selon le cas où la connexion est filaire ou sans fil. Le réseau peut également être déconnecté. Malgré ça, les applications doivent continuer leur fonctionnement.

Dans [DAV05], un exemple d'application d'un lecteur électronique, qui offre les fonctionnalisées suivantes, est décrit :

- Téléchargement de message depuis un serveur distant,
- Consultation de ces messages et
- L'envoi de courriers.

Pour l'envoi de courriers, Cette application a deux modes de fonctionnement : soit elle transmet immédiatement les courriers électroniques à leurs destinataires respectifs, soit elle stocke les courriers dans une file en mémoire. Lorsque le réseau est déconnecté, l'application ne peut pas envoyer les courriers à leurs destinataires, ce qui veut dire qu'elle ne peut fonctionner qu'avec le mode de fonctionnement stockant les courriers en mémoire. Mais, lorsque le réseau est connecté, l'application peut utiliser les deux modes de fonctionnement. Cependant, il est préférable d'utiliser le mode transférant immédiatement les courriers, qui permet de remettre le plus tôt possible les courriers à leurs destinataires. L'application s'adapte donc en adoptant le mode de fonctionnement qui correspond au mode du réseau (connecté ou déconnecté) qui lui est imposé tout en transmettant les courriers au plus tôt.

L'adaptation a également été étudiée dans le contexte des applications de diffusion multimédia, qui doivent s'adapter aux capacités du réseau et d'affichage des terminaux [FIT98].

Aussi, dans celui des grilles de calcul [BOU06] qui sont des plates-formes dont la quantité de ressources disponibles varie au cours du temps (des ressources apparaissent et d'autres disparaissent). Afin de persister aux disparitions de ressources et de profiter des apparitions, les applications doivent prendre en compte ces changements de situation [SAT05].

De même, les travaux de N.Loriant [LOR07], présentent des cas d'adaptation dans le contexte des systèmes d'exploitation. Dans ce contexte, des problèmes de synchronisation comme certains inters blocages peuvent ne parvenir qu'avec certains ordonnancements des processus. Donc, si le système d'exploitation ordonnance les processus intelligemment, certaines applications ayant des défauts de synchronisation peuvent fonctionner correctement malgré les problèmes de synchronisation de leurs algorithmes.

La figure 1.1 montre le cas ; à la détection d'un inter blocage, le système d'exploitation peut essayer de réexécuter l'application en ordonnançant les processus différemment. Le système d'exploitation s'adapte donc, en choisissant parmi ceux dont il dispose, l'ordonnanceur de processus qui convient.

Processus 1 :	**Processus 2 :**
...	...
Acquérir verrou 1	Acquérir verrou 2
Acquérir verrou 2	Acquérir verrou 1
...	...
Relâcher verrou 2	Relâcher verrou 1
Relâcher verrou 1	Relâcher verrou 2
...	...

Fig.1.1. Exemple d'algorithme à deux processus qui peut soit produire un inter blocage, soit fonctionner correctement selon l'ordonnancement des processus [LOR07].

Comme résumé, on retrouve dans les deux exemples précédents, les mêmes éléments. Tout d'abord, pour pouvoir s'adapter, les applications doivent disposer de plusieurs modes de fonctionnement. Après, l'adaptation se fait par rapport aux éléments qui sont imposés à ces applications (état de connexion réseau, algorithmes des applications …): l'adaptation doit garantir que les applications fonctionnent correctement au vue de ces éléments.

En plus, des objectifs de fonctionnalité sont à prendre en compte pendant l'adaptation (envoyer les courriers au plus tôt ; éviter les inters blocages…). On peut dire donc, qu'une application est adaptée lorsque son mode de fonctionnement correspond à la fois aux éléments imposés (respect des contraintes) et aux objectifs de fonctionnalité (optimisation).

1.2.4.2. Définition du mot adaptation

Le mot adaptation signifie une modification du comportement d'une partie du système en fonction du changement de son contexte pour atteindre un nouveau point de fonctionnement plus satisfaisant [SOU02].

L'adaptation a donc pour objectif de rendre le système résultant plus performant (au sens des critères de performance choisis).

Dans le domaine de la sensibilité au contexte [CHA07], afin d'améliorer les performances d'un système ou de continuer son exécution dans des environnements distincts, la notion

d'adaptabilité étend celle de l'adaptation tout en lui permettant de changer son comportement lui-même.

1.2.4.3. Utilité (Pourquoi ?)

L'adaptation est nécessaire lorsque les besoins exprimés (attendus) ne peuvent être satisfaits tous, et qu'il faut prendre des décisions pour maximiser la satisfaction globale sur des critères variables.

Dans [SAT01], M. Satyanarayanan a dit sur l'utilité de l'adaptation, *«L'adaptation est nécessaire quand il y a une disparité significative entre l'offre et la demande d'une ressource».*

Plusieurs raisons peuvent conduire à l'adaptation d'une application. Ces raisons peuvent être classées en quatre catégories [CHA07] [TOU10] :

- **Correction (adaptation correctionnelle)**

Dans certains cas, où l'application ne se comporte pas comme prévu, une adaptation est nécessaire pour corriger les défauts. La solution est d'identifier le module de l'application qui cause le problème et de le remplacer par une nouvelle version supposée correcte.

- **Changement d'environnement (Adaptation adaptative ou réactive)**

Consiste à adapter l'architecture de l'application, dans le cas où l'environnement d'exécution, quelques composants matériels ou d'autres applications ou ressources dont un élément dépend changent.

- **Extension (Adaptation évolutive)**

Avec l'évolution des besoins de l'utilisateur, l'application doit être étendue avec de nouvelles fonctionnalités qui ne sont pas prises en compte au moment du développement de l'application. Cette extension peut être réalisée en ajoutant un ou plusieurs modules pour assurer les nouvelles fonctionnalités ou même en modifiant les modules existants pour étendre leurs fonctionnalités.

- **Perfectionnement (Adaptation perfective)**

Ce type d'adaptation permet d'améliorer les performances de l'application. À titre d'exemple, si un module reçoit beaucoup de requêtes et n'arrive pas à les satisfaire. Pour éviter la dégradation des performances du système, une solution consiste à installer un autre module qui partage la charge avec le premier.

1.2.4.4. Adaptation dynamique [BAL10]

Une adaptation est dite dynamique si elle est effectuée au cours de l'exécution du système. Cette adaptation a pour but de maintenir la continuité des services assurés par le système. Elle est nécessaire lorsque le contexte d'exécution peut changer à tout moment.

Dans ce qui suit, nous intéressons essentiellement à l'adaptation dynamique.

1.3. Systèmes adaptatifs

1.3.1. Définition d'un système adaptatif

Un système est dit adaptable si une entité externe (logicielle ou humaine) peut le faire évoluer. Il est dit **adaptatif** s'il est capable tout seul de s'adapter automatiquement. Un système adaptatif est ainsi, à la fois, le sujet et l'acteur de l'adaptation [DOW01]: il se modifie de façon autonome en fonction de son contexte. Cette modification se fait donc dynamiquement pendant son exécution. Par conséquent, un système adaptatif peut reposer sur des reconfigurations dynamiques de son architecture pour réaliser son adaptation [ORE99].

1.3.2. Caractéristiques d'un système adaptatif [DAV05]

Un système adaptatif est caractérisé par:

- Un ensemble d'opérations pour modifier, adapter le système.
- Un contexte qui regroupe tous les éléments externes au système et qui influencent son fonctionnement.
- Une fonction d'adéquation qui permet à tout moment de savoir dans quelle mesure le système réalise ses objectifs relativement à son contexte.
- Une stratégie d'adaptation chargée de la mise en œuvre du processus d'adaptation dans le système.

1.3.3. Difficulté de construire des systèmes adaptatifs [DAV05]

Les systèmes adaptatifs doivent être au moins en partie ouverts (supportent des modifications non anticipées au moment du développement) et extensibles, afin de pouvoir évoluer en même temps que leur environnement. Cependant, il n'est pas facile de trouver le bon compromis entre forme et ouverture [DAV05], afin d'assurer à la fois l'évolutivité du système, et le maintien de sa cohérence interne. Le système requiert dans ce cas un mécanisme de reconfiguration dynamique (pour définir et redéfinir la configuration), ce qui va accroitre sa complexité et par conséquent, de devoir garantir la cohérence du système pendant et après l'adaptation.

À tout moment, pour un système et un contexte d'exécution donnés, il est nécessaire d'avoir une réponse aux questions suivantes [XAV10].

Quels sont les critères à optimiser ? Ce qui permet de définir le but de l'adaptation. Ce dernier, dépend évidemment du type d'application et des besoins de l'utilisateur. Par exemple, dans le cas du calcul haute performance, le critère qui est généralement considéré est le temps d'exécution.

À quels changements souhaite-t-on s'adapter ? Les changements à prendre en considération sont ceux qui affectent les performances de l'application. Par exemple, il sera intéressant d'adapter une application parallèle à l'ajout de machines s'il est possible d'extraire plus de parallélisme et de l'exécuter sur ces processeurs supplémentaires.

Quels sont les changements de configuration possibles ? Les différentes configurations peuvent s'exprimer de plusieurs manières: changement de la valeur d'un paramètre, changement de la méthode de calcul, changement du protocole de communication, …etc. Certaines configurations auront plus ou moins d'influence sur les performances de l'application.

Comment choisir la nouvelle configuration en fonction des nouvelles contraintes ? Le choix de la nouvelle configuration doit être fait pour améliorer les performances de l'application. La décision est prise en tenant compte de plusieurs éléments : les critères à optimiser, les nouvelles contraintes d'exécution et les changements de configuration possibles.

1.3.4. Etapes et mécanismes de l'adaptation dynamique (Comment ?)

L'adaptation dynamique est le processus par lequel une application logicielle est modifiée. Ce processus d'adaptation ne doit pas empêcher le fonctionnement de l'application. Il est donc nécessaire de trouver à quel moment l'exécuter et aussi comment appliquer correctement les modifications.

Trois fonctions essentielles composent ce processus [XAV10] [PAY06] [CHE01], à savoir:

A. L'**observation (**prise de décision de l'adaptation) : Consiste à récupérer des informations sur le contexte d'exécution, ce qui implique la détection des changements et le choix de les considérer comme significatifs ou non.

B. La **reconfiguration (**politiques d'adaptation) : correspond à l'étape modification du système, pour le passer de son ancienne configuration à sa nouvelle configuration. L'adaptation architecturale, comme le montre la figure 1.2, fait passer l'architecture de l'application d'une configuration (n) à une configuration (n+1) selon un contexte bien déterminé et en suivant des politiques d'adaptation.

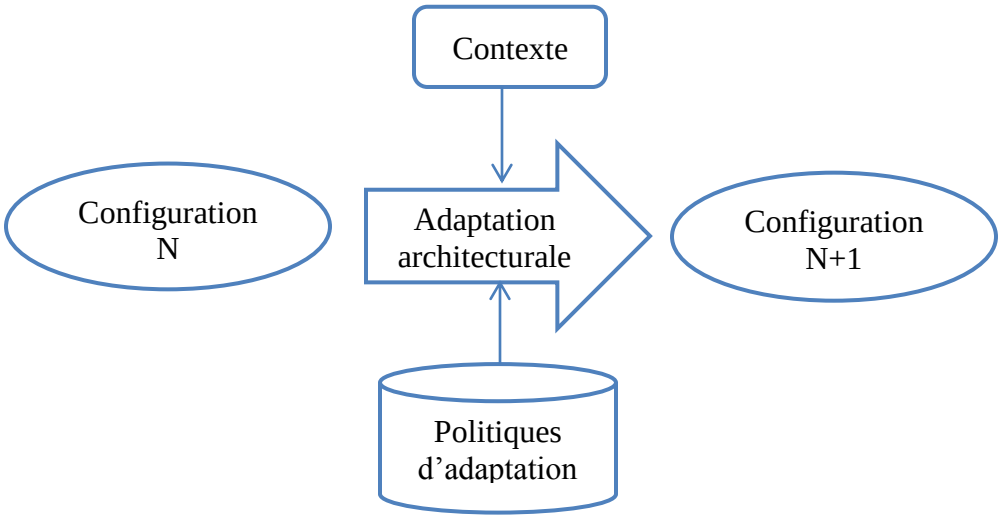


Fig.1.2. Description de l'adaptation architecturale.

Des modèles de reconfiguration plus généraux reposent généralement sur un modèle par composants [LEG09], permettent de modifier simplement la configuration de l'application et d'apporter certaines garanties sur la cohérence.

La politique d'adaptation a un rôle double : décider si le composant doit s'adapter, et décider comment il doit s'adapter.

Généralement, une politique d'adaptation est un ensemble de règles et de stratégies conçues pour accomplir un ensemble particulier de buts. Elle peut être définie comme une fonction qui transforme une série d'évènements qui représentent le contexte a un ensemble d'actions en respectant un ensemble de conditions et en partant d'un état initial [LOB99] (figure 1.3).

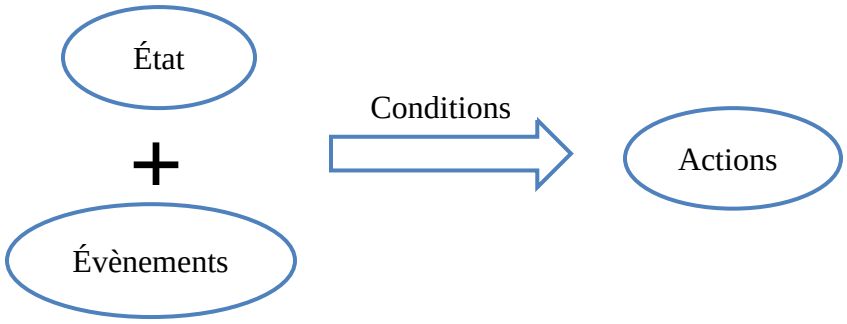


Fig.1.3. Description de la politique d'adaptation

Les politiques d'adaptation sont décrites sous la forme E-C-A (Évènement si Condition alors Action).

La politique peut avoir plusieurs niveaux de précision. Par exemple, dans DYNACO [PAY06], la politique va définir vers quel état le composant doit évoluer, mais pas la façon de le faire. Elle peut également définir la façon dont le composant doit s'adapter (suite d'opérations à exécuter pour l'adaptation) (voir exemple de la figure 1.4).

Règle 1 : qualité 1.0
A la réception de **l'évènement** « variation de la bande passante »,
Si la bande passante est supérieure ou égale à 56 Kbps,
Alors modifier le paramètre charger images à vrai ;

Règle 2 : qualité 0.5
A la réception de **l'évènement** « variation de la bande passante »,
Alors modifier le paramètre charger images à faux ;

Fig.1.4. Exemple de politique d'adaptation ACEEL extrait de [CHE05], pour un navigateur web qui s'adapte en ne téléchargeant les images que si la bande passante est suffisamment élevée.

À la réception d'un évènement qui déclenche l'adaptation, ACEEL recherche parmi les règles, celles dont la partie évènement correspond à l'évènement reçu. Puis, pour chacune de ces règles,

ACEEL évalue la condition, éventuellement en demandant au système d'observation les informations nécessaires. Enfin, parmi les règles dont la condition est satisfaite, ACEEL sélectionne celle qui a la meilleure qualité indiquée dans la politique et exécute l'action correspondante.

C. La décision (coordination) : Consiste à choisir la meilleure reconfiguration, en fonction du critère de performance considéré et en fonction du contexte d'exécution observé.

Une fois que l'application sait vers quel état elle doit évoluer, il faut définir de quelle façon elle doit le faire (plan d'exécution) et exécuter l'adaptation (exécution) [PAY06].

- **Plan d'exécution :** Le plan d'exécution définit la liste des opérations à exécuter par l'application pour qu'elle atteigne son nouvel état, ainsi que l'ordre d'exécution de ces opérations. Ce plan peut-être défini, comme pour la politique, par un fichier composé de règles suivant le modèle « Evènement - Condition - Action » donnant, pour un état initial et un état final donnés, la liste des actions à exécuter.

- **Exécution :** L'exécution de l'adaptation est la phase la plus sensible. À ce niveau, l'application connaît l'ordre d'exécution des opérations d'adaptation. Toutes les parties de l'application ne sont pas forcément touchées par l'adaptation. Néanmoins, il faut s'assurer de la cohérence de l'application dans sa globalité avant de pouvoir lancer l'adaptation.
Exécution de l'adaptation de façon à ce que l'application soit toujours dans un état cohérent.

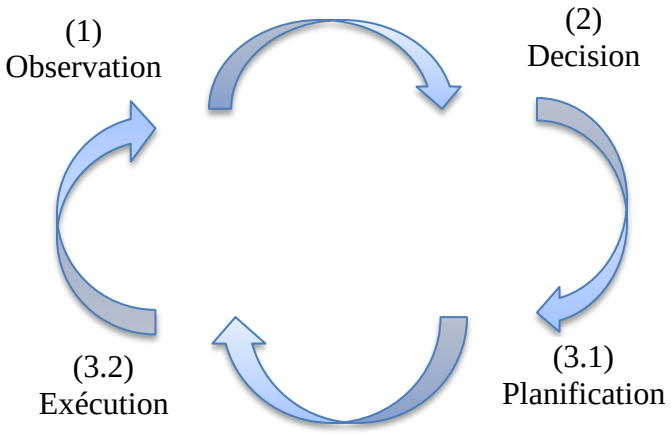


Fig.1.5. Processus d'adaptation dynamique.

1.4. Conclusion

Le sujet de l'adaptation est largement étudié en informatique avec les logiciels adaptatifs. Dans ce chapitre, nous avons présenté et précisé les concepts d'adaptation et de reconfiguration dynamique et le lien qu'existe entre ces deux notions.

L'adaptation est un processus qui vise à modifier le système considéré, en réponse à un changement dans son contexte d'exécution afin de le rendre plus performant dans son nouveau contexte.

Deux types d'adaptation peuvent être distinguées, à savoir: l'adaptation statique et dynamique. Cette dernière vise à permettre à un programme de se modifier au cours du temps avec prise en compte de l'évolution de son environnement. Elle est différente de l'adaptation statique dans le sens où le programme ne connaît pas à priori les ressources auxquelles il a accès, et que ces ressources peuvent évoluer non seulement d'une exécution à l'autre, mais également au cours d'une exécution [PAY06].

L'adaptation passe généralement par 3 étapes. En premier lieu, l'observation des ressources permet de prendre la décision d'adapter ou non. Une fois cette décision prise, il faut déterminer vers quel état va évoluer le système, et surtout finalement, la façon dont il va le faire (nature et ordre des opérations à effectuer pour le changement d'état**).**

Dans toutes les études existantes sur l'adaptation, nous avons constaté deux éléments clefs de l'adaptabilité, à savoir : la modification et la modularité. Ainsi, pour aboutir à un système plus adaptable, il faut autoriser et faciliter les modifications par l'adoption d'une modélisation architecturale à base de composants [ZHA09].

CHAPITRE 2

Programmation par composants et dynamicité des architectures logicielles

2.1. Introduction

Les systèmes adaptatifs intègrent des mécanismes de reconfigurations dynamiques pour s'adapter à des changements de contexte.

L'approche architecturale repose sur les architectures à base de composants pour faciliter la modification des systèmes informatiques à l'exécution [ORE98]. Cette approche permet d'exploiter la modularité et l'encapsulation forte des composants afin d'isoler précisément les effets des reconfigurations dans le système. Le choix d'une décomposition spécifique de l'architecture d'un système en composants et de la granularité de ces composants est par conséquent important, car il détermine la granularité des reconfigurations dynamiques possibles [LEG09].

La programmation par composants est la base des mécanismes de reconfigurations dynamiques. En effet, les reconfigurations reposent par hypothèse, sur les architectures à composants des systèmes considérés et sur leur dynamicité.

Ce chapitre présente les différents concepts liés aux architectures logicielles, qui sont utilisés pour la mise en œuvre de l'adaptabilité des systèmes logiciels.

2.2. Programmation par composants et architectures logicielles

2.2.1. Lien avec l'adaptation

La programmation par composants offre des mécanismes permettant de réduire le couplage entre les différents éléments constituant une application (séparation entre l'interface d'un composant et son implémentation) et supporte une notion d'architecture définie, permettant de structurer les applications. C'est pourquoi cette dernière est devenue l'approche privilégiée pour la réalisation d'applications adaptatives: d'une part, la réduction du couplage garantit que les modifications propres à une partie de l'application, n'auront pas ou peu d'effets sur le reste de cette application, et d'autre part l'architecture décrite explicitement par le programmeur offre les points de liaisons nécessaires à la réalisation d'opérations de reconfiguration. [ALD02]

Pour permettre la spécification et l'exécution des reconfigurations structurelles nécessaires à l'adaptation d'une application, nous avons besoin d'un modèle de composants réflexif, qui permet à la fois l'introspection (architecture explicite) et l'intercession (manipulation) de la structure de l'application en cours d'exécution [CAZ98].

L'adaptation d'une application se traduit par une reconfiguration dynamique de l'architecture de cette application. Le comportement de l'application étant déterminé par sa structure (le code source, bien que statique, détermine le comportement du logiciel). Ces modifications structurelles permettent d'adapter le comportement de l'application [ORE98] [ORE99].

2.2.2. Définition d'une architecture logicielle

La définition de l'architecture est une étape importante dans la conception d'un logiciel. Elle doit permettre la réalisation d'un produit qui répond aux besoins des utilisateurs, et qui est également évolutif. [POR03]

Dans la littérature, nous trouvons plusieurs définitions pour l'architecture logicielle comme celle donnée dans [KRU06]. D'après ces définitions, une architecture logicielle, comme le montre la figure 2.1, se définit comme la spécification abstraite d'un système, en précisant les parties (composants ou modules) qui le constituent, les connecteurs qui modélisent les interactions entre ces parties, et l'ensemble des règles qui gèrent ces interactions [CHE05].

Les composants encapsulent spécifiquement l'information ou la fonctionnalité. Tandis que les connecteurs assurent la communication entre les composants. Toute architecture possède généralement un ensemble de propriétés d'ordre topologique ou fonctionnel qu'elle doit respecter tout au long de son évolution [HAD08].

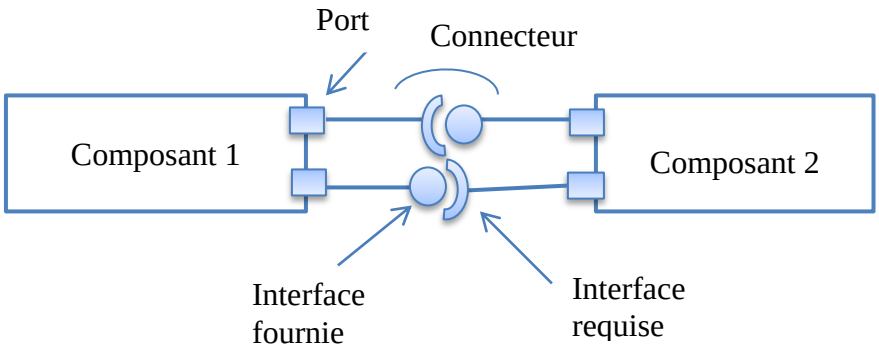


Fig.2.1. Les éléments d'une architecture logicielle.

2.2.3. Les différents éléments d'une architecture logicielle

Les concepts communs ou proches concernant la plupart des modèles de composants sont: les composants, les connecteurs et les configurations. Ces éléments d'architectures sont définis d'une façon générale par une définition qui correspond à la définition reprise dans la plupart des modèles de composants [GAR00].

A. Les composants

Un composant est une unité de calcul ou de stockage. Il peut être simple (primitif) ou composé (composite) [LEG09] (voir figure 2.2).

Deux parties définissent un composant. Une partie externe, comprend la description des interfaces fournies et requises par le composant, qui sont des ports d'entrée/sortie pour interagir avec l'environnement. La seconde partie correspond à son contenu et permet la description du fonctionnement interne du composant.

Le composant offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires [HAD08].

D'une manière générale, un composant rend un service en utilisant une ou plusieurs interfaces externes. Il a vocation à être intégré et à servir dans plusieurs applications. Pour ce faire, il doit être suffisamment autonome et comporter toutes les informations lui permettant, éventuellement, d'être assemblé avec d'autres composants [POR03], en respectant une architecture logicielle précise [CHE05].

Dans ce mémoire, nous parlerons de type de composant et d'instance de composant.

- Un type de composant est la définition abstraite d'une entité logicielle (représente le rôle du composant dans le système).
- Une instance de composant est équivalente à une instance d'objet (une entité existante et s'exécutant dans un système). Elle est caractérisée par une référence unique.

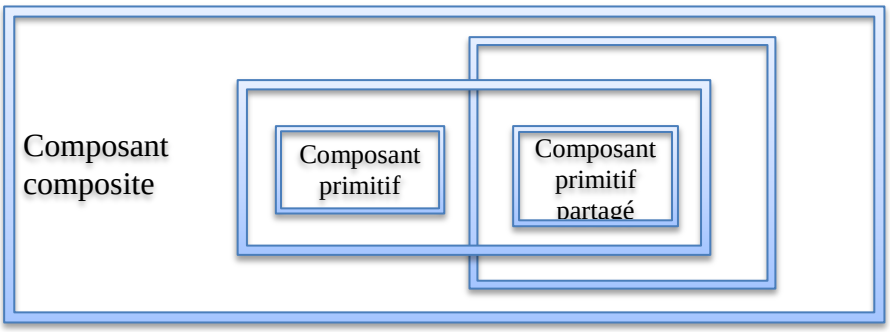


Fig.2.2. Structure interne d'un composant.

B. Les connecteurs

Le connecteur est un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants par la définition des règles qui conduisent ces interactions [TOU10].

La notion de connecteur recouvre l'ensemble des moyens nécessaires pour assurer l'interaction entre les composants par la connexion des interfaces requises et offertes. Un connecteur peut être défini de deux manières :

- Au plan conceptuel par la spécification des propriétés des interfaces requises et des interfaces offertes.
- Au plan communication par la spécification d'un mode de communication.

Le connecteur peut se voir implanté par un ensemble de moyens logiciels et matériels réels pour assurer concrètement l'adaptation des propriétés requises à celles offertes. [POR03]

C. Les ports

Un composant interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Un port représente un point d'accès à certains services du composant. Le comportement interne du composant ne doit être ni visible, ni accessible que par ses ports. [CAR03]

D. Les interfaces

L'interface est le point de communication qui permet d'interagir avec l'environnement. On la retrouve donc associée aux composants et aux connecteurs. Elle spécifie des ports dans le cas des composants et des rôles dans le cas des connexions.

Pour un composant, il existe deux types d'interfaces:

– Les interfaces fournies décrivent les services proposés par le composant.

– Les interfaces requises décrivent les services que les autres composants doivent fournir pour assurer le bon fonctionnement du composant dans un environnement particulier. [HAD08]

Ces interfaces sont éventuellement exprimées par l'intermédiaire des ports.

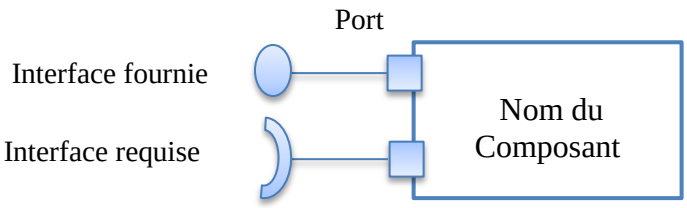


Fig.2.3. Modélisation d'un composant par un graphe.

E. La configuration d'une application [LEG09] [POR03]

Comme le montre la figure 2.4, une configuration définit la structure et le comportement d'une application par assemblage de composants et de connecteurs à partir de leurs interfaces. La configuration structurelle de l'application correspond à un graphe connexe des composants et des connecteurs qui constituent l'application. La configuration comportementale, quant à elle, modélise le comportement par la description de l'interaction et l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles.

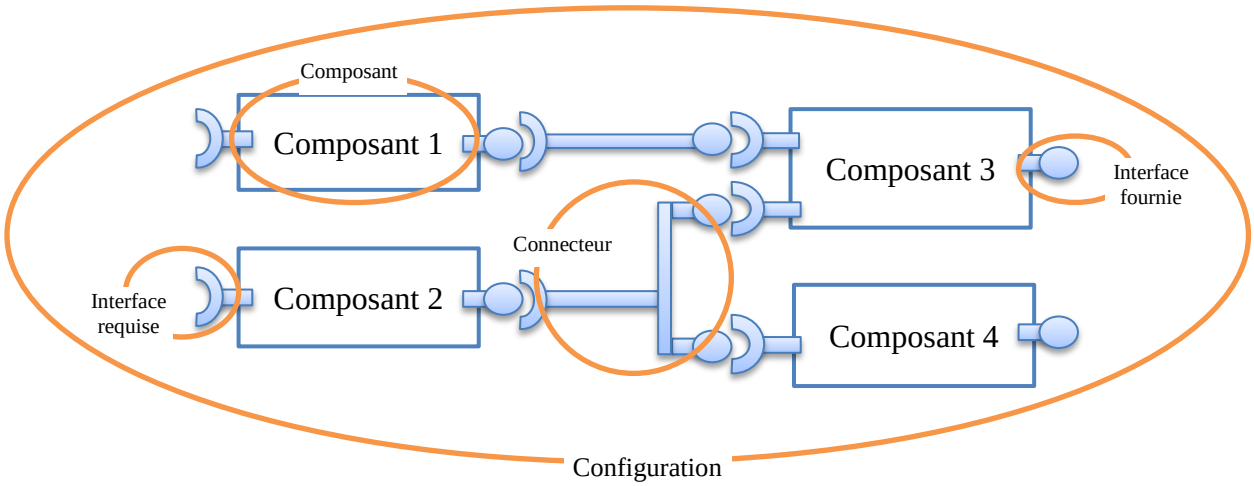


Fig.2.4. Exemple de configuration d'une application.

2.2.4. Types d'évolutions dynamiques d'une architecture [CHA07] [XAV10] [HAD08]

Différents types d'évolution dynamique d'une architecture existent. Parmi ces types nous trouvons :

A. Changement de l'implémentation d'un composant

Ce type d'évolution consiste à changer l'implémentation interne d'un composant (le code du composant) sans changer la structure de l'application. Ce type de changement permet d´évoluer un composant d'une version à une autre (voir figure 2.5) pour une raison perfective, adaptative ou correctionnelle.

L'opération de remplacement d'un composant est généralement considérée comme la composition d'une opération de retrait suivie d'une opération d'ajout de composant. [LEG09]

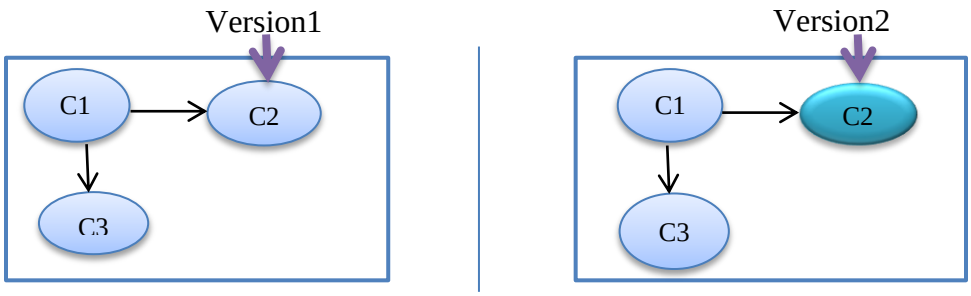
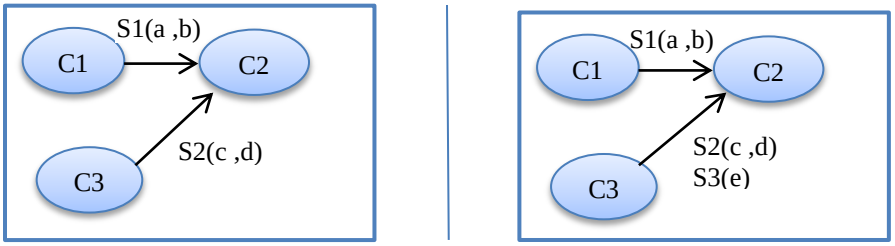


Fig.2.5. Exemple de changement d'implémentation.

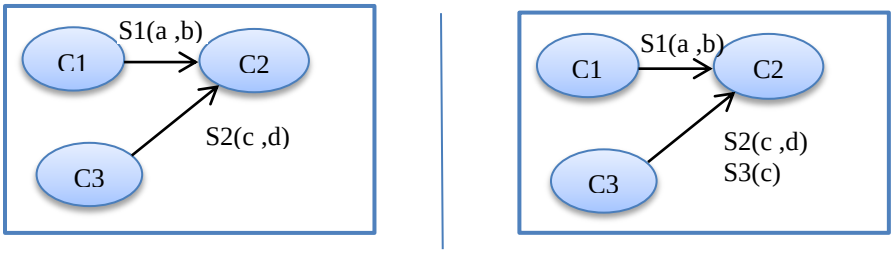
B. Changement d'interfaces

Ce type de changement des interfaces d'un composant correspond à la modification de la liste des services qu'il fournit (voir la figure 2.6). Ceci se traduit par :

- L'ajout de nouvelles interfaces pour étendre les fonctionnalités du composant. Les fonctionnalités exposées à travers les autres interfaces restent inchangées.
- La suppression des interfaces initialement supportées par le composant (et par conséquent ses implémentations) si ses fonctions ne seront pas utilisées pour des raisons de performances.
- La modification des opérations déclarées dans les interfaces [KET04]



Cas 1 : Ajout de l'interface e au composant C3 permet de fournir le service S3 au composant C2.



Cas 2 : Modification des opérations déclarées dans l'interface c du composant C3 permet de fournir le service S3 au composant C2.

Fig.2.6. Exemple de changement d'interfaces.

C. Changement de géométrie

Elle est appelée aussi dynamique de localisation ou encore de migration. Elle modifie la distribution géographique des composants d'une application. En effet, ce type de changement altère uniquement l'emplacement des composants.

Ainsi, un composant peut changer de localisation en migrant d'un site vers un autre. Comme le montre la figure 2.7, le composant C4 est déplacé du site 2 vers le site 3.

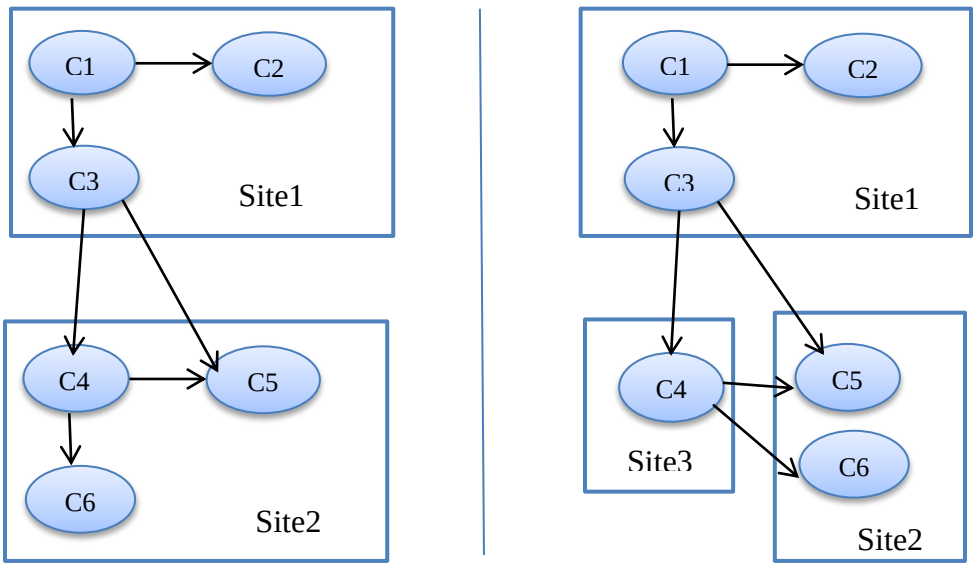


Fig.2.7. Exemple de changement géométrique.

D. Changement de la structure (architecture conceptuelle d'une application)

Dans ce type de dynamique on s'intéresse uniquement au changement de la structure de l'application en termes de composants et de connexions. Ce changement se réalise à l'aide de quatre opérations de base:

- **L'ajout de nouveaux composants**

En général, quand un composant est ajouté à une application en cours d'exécution, il doit prendre en compte l'état de l'application, il doit donc récupérer l'état à partir duquel il doit démarrer et se personnaliser en fonction de cet état.

- **Le retrait de composants existants**

Le composant supprimé ne doit pas bloquer l'exécution des autres composants. Il doit aussi être dans un état stable avant d'être supprimé. Par exemple, si un composant est en cours d'écriture de données dans un fichier, il ne doit pas être supprimé avant qu'il termine sa tâche. Un autre cas concerne les données et les messages échangés entre le composant en question et les autres composants, ces données ou messages ne doivent pas être perdus pour maintenir le bon fonctionnement de l'application.

- **L'ajout\la suppression des interconnexions entre composants**

Quand un composant est ajouté\supprimé, il doit être connecté\déconnecté aux composants existants. En général, pour connecter deux composants, les types de leurs ports connectés doivent être compatibles.
L'adaptation des interconnexions doit préserver les messages et les données en transit.

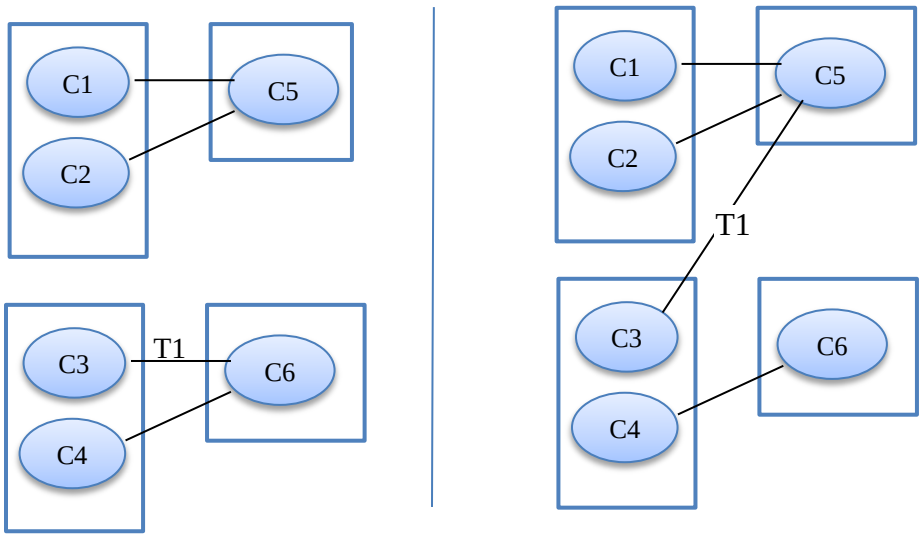


Fig.2.8 Exemple de changement de structure.

Comme le montre cette figure la connexion T1 est redirigée vers le composant C5.

2.3. Adaptations dynamiques dans les architectures à base de composants

2.3.1. Problèmes posés par les reconfigurations dynamiques [ORZ98]

Les reconfigurations dynamiques posent plusieurs problèmes dans les architectures à composants. Ainsi, il faut prendre en compte les considérations suivantes :

- Les reconfigurations dynamiques doivent préserver la cohérence de la structure du système et la validité de son comportement, de même que certaines propriétés non-fonctionnelles (sécurité, qualité de service, etc.).

L'ajout et la vérification de contraintes (section 2.3.2) permettent par exemple de restreindre les transformations possibles des architectures et d'assurer leur cohérence.

- L'exécution des reconfigurations dynamiques doit être synchronisée avec l'exécution fonctionnelle du système. Par exemple, il ne doit pas être possible pour un composant de continuer à invoquer des méthodes de l'interface fonctionnelle d'un autre composant alors que ce dernier est en train d'être déconnecté. Il existe des algorithmes de synchronisation [KRA90] qui visent à fixer l'état des composants avant une reconfiguration.

- Un problème complémentaire à la synchronisation est la gestion de l'état des composants reconfigurés, et plus précisément le transfert d'état entre composants après la reconfiguration. Lorsqu'un composant s'adapte, il modifie son fonctionnement. Cette modification peut se traduire par un changement du comportement associé au composant. [BOU06]

Ainsi, il peut être nécessaire de transférer l'état d'un ancien composant vers le nouveau composant qui le remplace dans une architecture.

- L'occurrence possible de plusieurs reconfigurations simultanément dans un système en exécution, nécessite une synchronisation afin d'éviter les conflits entre reconfigurations qui peuvent mettre le système dans un état incohérent.

➢ De ces problèmes, plusieurs propriétés du processus de reconfiguration sont à assurer [XAV10].

Parmi ces propriétés on peut citer:

- La **cohérence :** signifie qu'une reconfiguration doit laisser l'application dans un état correct. La cohérence est une propriété nécessaire d'une reconfiguration car le système peut devenir inutilisable si elle n'est pas respectée. La problématique de la cohérence est traitée plus en détail dans la section 2.3.3.
- La **généralité :** indique que le processus de reconfiguration doit pouvoir supporter tous les types de reconfigurations sur tous les types d'éléments.

- La **scalabilité :** indique que la reconfiguration doit pouvoir s'appliquer à tout le système ou seulement à une petite partie. Dans le cas où seulement une partie du système est concernée par la reconfiguration, le reste du système doit pouvoir continuer son fonctionnement pendant la reconfiguration.
- L'**efficacité :** signifie que le temps de reconfiguration doit être aussi petit que possible, de manière à réduire l'interruption de service.

2.3.2. Contraintes [LEG09]

Une contrainte est une propriété ou une assertion sur un système ou une de ses parties, sa violation met le système dans un état inadmissible ou dégradé par rapport à sa spécification et à ses fonctionnalités attendues.

Les trois types d'assertions possibles sont :

- Les invariants : caractérisent une condition qui doit être toujours vraie. Ce sont des prédicats portant sur tout ou une partie du système. Ces prédicats sont exprimés en fonction des variables d'état du système.
- Les pré-conditions: spécifient des conditions qui doivent être garanties avant l'exécution d'un traitement quelconque.
- Les post-conditions: spécifient des conditions qui doivent être garanties après l'exécution d'un traitement quelconque.

2.3.3. Cohérence [BAL10]

On dit qu'un système est cohérent avant et après son adaptation si :

- Le système respecte ses contraintes structurelles (cardinalité des liaisons, correspondance des interfaces requise et fournie...).
- Les entités du système sont dans des états mutuels consistants (deux entités sont dans des états mutuels consistants si leurs interactions font progresser ces états).
- Les invariants du système (conditions booléennes sur les propriétés du système) sont vrais.

La fiabilité des reconfigurations repose sur le maintien de la cohérence des architectures des systèmes reconfigurés. Pour préserver cette cohérence, des critères sont à évaluer par la réponse à quelques questions, à savoir: [LEG09]

- Type de contraintes : Quelles sont les contraintes prises en compte pour définir la cohérence des architectures (invariants, pré/post-conditions, contraintes structurelles ou comportementales, etc.) ?
- Langage de contraintes : Existe-il un langage dédié pour la spécification des contraintes et avec quel formalisme ?
- Système de typage des éléments de l'architecture : Quelle sont les particularités du système de typage des éléments de l'architecture?

- Mécanisme de vérification de la cohérence des architectures et réaction en cas de violation de la cohérence : Comment et quand sont vérifiées les contraintes dans le système et quelle est la réaction adoptée en cas de violation de la cohérence des architectures?

2.3.4. Gestion de l'adaptation

L'adaptation dynamique est une opération risquée qui peut rendre le système incohérent. Elle doit donc avoir lieu à un moment maitrisé dans l'exécution du système afin de maintenir ses fonctionnalités et sa réactivité.
Différentes approches peuvent être considérées pour éviter l'inconsistance des systèmes adaptatifs.

Des approches où les mécanismes de reconfiguration font partie du code source de l'application, comme Polylith [HOF93] [PUR94]. Dans ces approches, les points d'application de l'adaptation sont précisés par le concepteur. Lorsque ces points sont atteints au cours de l'exécution, le système sait qu'il est dans un état lui permettant d'être adapté. Cependant, avec ces approches la réactivité est insuffisante étant donné que le système doit attendre d'atteindre ces points pour être adapté. Aussi, elles comportent des risques d'erreurs car elles dépendent fortement de la capacité de compréhension du système par le concepteur. De plus, les concepteurs doivent connaitre les mécanismes d'adaptation afin de contrôler et gérer la consistance du système.

Afin de systématiser les moments de l'adaptation et libérer les concepteurs de tâches complexes et répétitives, des approches comme Conic [KRA89], où les mécanismes de reconfiguration sont indépendants du code applicatif ont été proposées. Conic évite l'inconsistance par la désactivation d'une partie du système, ce qui cause une importante perturbation des services. De plus, l'algorithme de reconfiguration utilisé ne peut pas être appliqué dans tous les cas de systèmes.

De palma et al [PAL99] [PAL01] ont proposé une approche assez efficace combinant Polylith et Conic, malgré qu'un peu de travail de programmation additionnel est exigé. Nous décrivons brièvement cette approche dans ce qui suit.

L'approche générale de reconfiguration dynamique de De palma et al présentée dans [PAL99] et spécifiée dans [COR01] considère que l'adaptation est toujours possible. Cette approche de reconfiguration est basée sur un modèle à base d'agents où les éléments du système (agents) réagissent selon le modèle « Evènement-Condition-Action ». Elle assure la consistance du système pendant et après la reconfiguration dynamique par la préservation des noms, états et canaux de communication des agents.

Cette approche fournit un ensemble de primitives pour piloter le processus de reconfiguration:
ADD (ajout d'un nouvel agent à l'application),
DELETE (suppression d'un agent de l'application),
MOVE (migration d'un agent d'un site vers un autre), pour le cas des systèmes distribués.
BIND et REBIND (création et modification d'un canal de communication entre deux agents).

L'implémentation des primitives REBIND, MOVE, et DELETE doit éviter l'inconsistance en imposant certaines pré-conditions.
Intuitivement, quand un agent est entrain d'être reconfiguré, son exécution doit être suspendue. Ceci peut être obtenu par l'assurance que l'agent ne reçoit aucune requête d'invocation de services durant sa reconfiguration. La pré-condition pour une exécution correcte des primitives de reconfiguration REBIND, MOVE, et DELETE peut se résumer comme suit:
Tous les canaux de communication impliqués doivent être vides avant l'occurrence de la reconfiguration.

Pour l'implémentation de ces primitives la notion d'états abstrait a été utilisée. À un moment donné, un agent peut être dans l'un des états abstraits ci-dessous:

Etat	Sens
Active	L'agent est en exécution, il peut donc fournir et recevoir des services
Passive	L'agent peut recevoir et répondre à des requêtes mais ne peut pas demander un service d'un autre agent; toutes ses demandes doivent attendre sa réactivation.
Frozen	L'agent ne peut pas recevoir aucune requête de d'autres agents; tous les agents ayant une référence vers lui sont dans l'état « passive » et les canaux correspondants sont vides.

Avant l'exécution des commandes de reconfiguration, le configurateur (agent qui intègre l'algorithme de reconfiguration) impose des états abstraits à certains agents pour permettre l'exécution de la reconfiguration et préserver la consistance.

2.4. Conclusion

Dans ce chapitre, nous avons mis notre travail dans son contexte. Notre étude se base essentiellement sur les architectures logicielles à base de composants. Ces dernières peuvent changer de structure et de comportement suite à la variation des exigences des utilisateurs ou à la variation du contexte de l'application.

Notre travail consiste à tirer profit des avantages des techniques formelles existantes, en les appliquant à la conception architecturale. L'objectif est de pouvoir offrir aux développeurs la possibilité de raisonner, analyser et vérifier l'architecture d'un système logiciel adaptatif.

Avant d'entamer la présentation de notre proposition, Nous présentons dans le chapitre suivant un aperçu sur les diverses méthodes formelles dont nous avons besoins pour établir une méthode permettant d'assurer la fiabilité de l'adaptation dynamique aux variations de l'environnement.

CHAPITRE 3

Méthodes formelles et sûreté de fonctionnement

3.1. Introduction

Les spécifications formelles permettent une description non ambiguë, précise, complète et indépendante de toute implémentation, ce qui conduit à une compréhension approfondie du système à développer, et beaucoup d'erreurs sont ainsi évitées.
La possibilité d'appliquer des vérifications et des validations sur ces spécifications est l'un des points qui rend leur utilisation de plus en plus fréquente [KOR00].

Ce chapitre est constitué de deux parties. Dans la première, nous proposons une vue globale du monde des méthodes formelles. Nous essayons de montrer la diversité des techniques de modélisation aujourd'hui disponibles et la difficulté que représente le choix de telle ou telle technique pour la modélisation d'un système particulier. En effet, ce choix dépend en grande partie des objectifs visés par la modélisation [CAS02]. Dans la deuxième partie, nous présentons un état de l'art des méthodes de spécification et de validation des reconfigurations dynamiques des logiciels.

3.2. Méthodes Formelles

3.2.1. Définition

Les méthodes formelles représentent l'ensemble des techniques disponibles permettant la modélisation de tout ou partie d'un système (particulièrement les applications logicielles). Ces techniques reposent sur des fondements mathématiques précis qui permettent de raisonner sur des propriétés et de construire des preuves afin de montrer que les propriétés exprimées sur un programme sont bien respectées. [CAS02]
Pour cela, les méthodes formelles décrivent [DUF03]:

- Les hypothèses sur l'environnement dans lequel le système va évoluer.
- Les contraintes que le système doit établir.
- Une implémentation respectant ces conditions.

Ainsi, elles peuvent être utilisées lors des différentes étapes du processus de développement d'un système.

Une méthode formelle comporte [LIE06] :

- Un langage formel dans lequel le système et ses propriétés pourront être représentés, c'est-à-dire formalisés ou modélisés formellement.
- Un ensemble d'outils pour raisonner sur des éléments de ce langage.

3.2.2. Les techniques formelles

Dans cette section nous donnons un simple aperçu sur les principales techniques et les différentes approches des méthodes formelles. Nous distinguons les deux faces des méthodes

formelles: la modélisation d'un système d'une part et la preuve de ses propriétés d'autre part [LIE06].

3.2.2.1. Les outils de modélisations formelles

Selon la nature des mathématiques retenues pour exprimer et formaliser un système, la modélisation peut prendre différentes formes. Ainsi, différents types de modèles sont à distinguer, dont les modèles algébriques, logiques, fonctionnels, les modèles basés sur les machines d'états abstraits et les modèles basés sur les automates. [CAS02] [LIE06] [DUF03]

Dans ce qui suit, nous présentons des outils représentatifs de chacun de ces modèles.

A. Le modèle algébrique [DUF03]

Le modèle algébrique est représenté par des outils basés sur des théories mathématiques telles que la théorie des ensembles ou la théorie des catégories (par exemple le système Specware). La méthode formelle Larch, par exemple, possède un langage de spécification algébrique [LOU09].

B. Le modèle logique [DUF03]

Ce modèle est centré sur les notions de prédicats, de règles et de déduction sur ces règles.
La programmation logique (prolog est un outil de programmation logique), considère les formules logiques comme des programmes et la construction de leurs preuves comme l'exécution de ces programmes.

Une approche assez similaire au modèle logique est donnée par les systèmes de réécriture tels qu'Elan et Spike. Dans cette approche les axiomes et règles de la logique sont remplacées par un système de règles de réécriture du premier ordre.

C. Le modèle fonctionnel [LIE06]

La modélisation d'un système dans ce cas, se fait par une fonction (au sens mathématique du terme) d'un état global vers un nouvel état global. Pour un programme, cela correspond à sa sémantique dénotationnelle : le programme est représenté par une fonction prenant en argument l'état global de la mémoire avant l'exécution (ensemble des valeurs des variables manipulées par le programme) et donnant comme résultat l'état global après l'exécution.

Le système modélisé est alors représenté par une série de définitions de fonctions. Chaque fonction $f : E \rightarrow F$ est définie par son domaine (E) et son co-domaine (F).
Le fondement théorique sous-jacent de ce modèle fonctionnel est le lambda calcul. Les fonctions sont considérées comme des objets de calcul et peuvent être arbitrairement manipulées: elles peuvent elles-mêmes être les arguments ou les résultats de d'autres fonctions. En effet, elles peuvent être composées et la composition est une fonction. Dans ce formalisme, elles sont éventuellement récursives. [CAS02]
Le lumbda-calcul typé [DUF03] représente un cadre général de développement pour quelques ateliers dédiés au développement de preuves formelles comme Coq ou Isabelle. [CAS02]

D. Le modèle Etat-Transition

Le système peut être représenté sous la forme d'un système de transitions comprenant : d'une part, un ensemble d'états où chaque état représente l'état global du système à un instant donné, et d'autre part, un ensemble de transitions entre ces états où chaque transition représente une façon possible pour faire passer le système d'un état à un autre.
Lorsqu'il s'agit de modéliser un programme, la représentation sous la forme d'un système de transitions correspond à la sémantique opérationnelle du programme, c'est-à-dire que le programme est représenté par la suite des différents états successifs de son exécution.

Selon les outils utilisés pour fournir le modèle, nous distinguons [LIE06]:

- **Les modèles basés sur les machines d'états abstraits (ASM: Abstract State Machine)**

Le système est défini par un ensemble d'états et par un ensemble de règles de transitions qui peuvent être appliquées aux états. Ces règles de transitions peuvent s'exécuter sous certaines conditions (appelées gardes) pour faire évoluer le système en exécutant la machine. [LIE06]

Dans ce formalisme, il est possible de spécifier un système non-déterministe par la donnée de plusieurs règles de transition sur un même état [DUF03].
De nombreux outils sont issus des ASM, comme par exemple le langage de programmation ASMGopher qui étend le langage de programmation fonctionnel Gopher. [LOU09]

- **Les modèles basés sur les automates**

Un modèle mathématique proche des ASM est fourni par les automates. Dans ce cas, les transitions représentent des réactions d'un système à des stimuli extérieurs. Ce type de modèles est utilisé pour représenter des systèmes réactifs ou des protocoles de communications. [LIE06]

Contrairement aux ASM, les états doivent être en nombre fini et connus à l' avance. Il s'agit dans ce cas de la principale limitation des automates qui pénalise la vérification, puisque les propriétés sont prouvées par exploration de tous les états. [DUF03]
Pour ces raisons, Les automates sont beaucoup plus adaptés à la vérification de protocoles (pour lesquels le nombre d'états est limité), qu'à la spécification et à la vérification de logiciels [CAS02].
Parmi les outils développés pour spécifier un système formel de protocoles de communications, nous citons LOTOS qu'a été utilisé dans [COR01] pour la spécification d'un protocole de reconfiguration.

E. Le modèle à base d'états

Pour finir avec les outils permettant la spécification et la vérification de systèmes, nous mentionnons ceux proposant l'annotation de programmes. [DUF03]

Il s'agit du modèle de la logique de Hoare. Dans ce cadre, le système est annoté par un énoncé logique décrivant son comportement: la donnée de pré-conditions et de post-conditions pour les opérations du système et par la donnée d'invariants. [LOU09]
Les annotations sont traduites vers un système de preuve existant pour être vérifiées.
L'avantage de cette approche est que l'implémentation elle-même du programme, et non pas une spécification, sert à la vérification. D'autre part, le langage d'annotation est suffisamment simple pour être accessible et utilisé par des personnes non-expertes en méthodes formelles, même si la vérification pourra requérir une expérience plus large.
Des langages tels que VDM, B et Z reposent sur ce modèle.

3.2.2.2. Quelques techniques pour prouver des propriétés [BAR05] [CAS02] [LIE06] [STO07]

Nous proposons de décrire dans cette section les deux principales techniques (le modèle checking et la preuve de théorèmes) permettant de prouver des propriétés sur des modèles et de vérifier la cohérence. Ces techniques sont utilisées pour raisonner sur des spécifications et ainsi, détecter et éviter des fautes dans la spécification.

Le choix d'une technique par rapport à une autre est lié à la nature de la modélisation.

A. Le modèle checking

Il est utilisé sur des modèles ayant un nombre fini d'états [STO07]. Cette technique consiste à vérifier les propriétés d'un modèle d'un système par rapport aux propriétés qui sont attendues sur ce modèle.
Cette vérification se fait automatiquement sur tous les états possibles du système pour renvoyer à la fin, soit une confirmation que chaque propriété est maintenue par le modèle, soit qu'elle ne l'est pas en donnant un contre exemple qui montre comment la propriété n'est pas maintenue.

Il y a trois étapes dans la mise en œuvre d'un model checking : [CAS02]

- Modéliser le système par un automate d'états finis.
- Définir les propriétés à vérifier sur le modèle dans une logique temporelle.
- Vérifier les propriétés définies sur le modèle.

La principale contrainte de ce modèle est l'exploration de tous les états à analyser. Cette exploration peut prendre un très long temps, voire ne jamais s'exécuter sur la machine par manque de ressources. Cela a donné naissance à d'autres techniques telles que l'abstraction de modèles réduisant le nombre d'états à analyser, mais dépendantes des propriétés à prouver. [LIE06]

B. La preuve de théorèmes

C'est une technique où le système et les propriétés recherchées sont exprimés comme des formules dans une logique mathématique. Cette logique est décrite par un système formel défini par une théorie composée d'axiomes (des formules supposées vraies sans démonstrations) et des

règles d'inférences permettant de déduire une formule (la conclusion) à partir d'un ensemble de formules vraies (les prémisses).

La preuve de théorèmes est le processus de recherche de la preuve d'une propriété à partir des axiomes du système. Les étapes pendant la preuve font appel aux axiomes et aux règles, ainsi qu'aux définitions et lemmes qui ont été possiblement dérivés.

Le principe est de générer des obligations de preuve (des formules logiques) à partir du modèle et de la propriété, de telle sorte que, si elles sont prouvées, alors le modèle vérifie nécessairement la propriété. La difficulté se trouve alors dans la recherche d'une preuve permettant d'établir le respect de la propriété voulue.

Au contraire du modèle checking, la preuve de théorèmes peut s'utilisé avec des espaces d'états infinis grâce à des techniques telle que l'induction structurelle.

Récapitulatif

De la classification établie dans la section précédente, nous avons deux types de modèles qui conduisent à la construction puis à la preuve de théorèmes: les modèles basés sur les machines d'états abstraits, les modèles à états et les modèles fonctionnels. [CAS02]

Les premiers modèles reposent sur la théorie des ensembles et la logique du premier ordre qui est moins expressive que la logique d'ordres supérieurs. Parmi les outils proposés comme Z et B, des prouveurs automatiques comme Z/EVES et Atelier B respectivement, permettent de prouver une bonne partie des propriétés attachées aux modèles.

Les seconds modèles (les modèles fonctionnels) reposent sur la logique d'ordre supérieur et le lambda calcul. Plusieurs outils sont proposés dans cette vision comme l'assistant de preuve Coq.

3.2.3. Choix spécifique d'une méthode formelle

Le choix d'une technique de vérification par rapport à une autre est lié à la nature de modélisation. Par exemple, le model-checking est limité à un nombre fini d'exécutions. Pour qu'il puisse couvrir un ensemble plus important d'exécutions, on peut s'intéresser à vérifier une abstraction du système, en utilisant des techniques d'interprétation abstraite [LIE06]. On se trouve donc avec un problème lié en même temps au choix de l'abstraction et à la vérification de la propriété sur l'abstraction.

De la même manière, pour les prouveurs de théorèmes, une trés grande complexité du modèle formel peut interdire l'utilisation de vérificateurs automatiques qui nécessitent beaucoup de travail et des utilisateurs très spécialistes. Pour diminuer cette complexité, il est possible de décomposer le problème, en réalisant une description modulaire du système. Cependant, cette approche ne garantit pas toujours de diminuer la complexité de la vérification. En effet, il faut pour cela effectuer des choix pertinents qui permettent d'établir la propriété par parties et ensuite pour l'intégration des parties.

Ainsi, on trouve dans tous les cas un problème d'indécidabilité dans lequel seule l'expérience du vérifieur permet d'effectuer de bons choix.

En conséquence, l'expertise du spécifieur est un facteur important durant toutes les phases du développement d'un système (modélisation, validation). C'est pourquoi, il faut lui fournir des outils permettant de l'assister dans sa tâche.

3.3. Modélisation et vérification des reconfigurations dynamiques

Nous illustrons dans cette section l'application des méthodes formelles pour la spécification de la reconfiguration dynamique des systèmes.
Nous présentons, dans ce qui suit, une étude de quelques approches avec leurs avantages et leurs limites.

3.3.1. Approche du le métayer

Pour la spécification de l'évolution des systèmes, Le Métayer a proposé dans [MET96] de définir formellement l'architecture par un graphe (modèle mathématique), où, les nœuds représentent les composants et les arcs dénotent les liens de communication entre ces composants. Ainsi, le style architectural a été défini par une grammaire contexte libre et les opérations de reconfigurations via des règles conditionnelles de réécriture de graphe.

Formellement, un graphe est un ensemble de relation R(e1,…..,en) (où ei représente le nom d'une entité). Dans ce modèle, deux types de relation existent :

- Une relation binaire L(e1,e2) dénote un lien de nom L entre e1 et e2, et
- Une relation unaire U(e) caractérise le rôle de l'entité e dans l'architecture.

Une grammaire de graphe est un quadruplet <NT, N, PR, AX>, où NT est l'ensemble des symboles non terminaux, N est l'ensemble des terminaux, PR l'ensemble des règles de production et AX est l'axiome (AX ∈ NT). Le style défini par la grammaire est une classe de terminaux généré par la grammaire.

La modélisation a été illustrée par un exemple d'architecture client/serveur qui a été décrit par le graphe suivant :

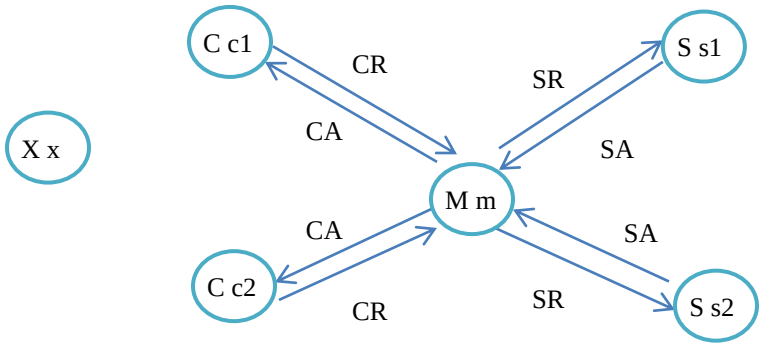


Fig.3.1. Représentation d'un modèle client/serveur par un graphe.

Comme le montre la figure ci-dessus, une relation binaire se représente par un arc et une relation unaire par un cercle. C, S, M, X représentent respectivement, les entités client, serveur, manager et entité externe.

Formellement, le système est représenté par l'ensemble :{ C(ci), S(si), M(m), X(x), CR(ci, m), CA(m, ci), SR(m, si), SA(si, m)} avec i =1,2.
Chaque client (C) envoi une requête (CR) au manager (M) qui à son tour, la transmet avec (SR) vers (S). La réponse du serveur (SA) revient au client avec (CA).
Le style architectural client\serveur est défini par la grammaire :
<{CS,CS1},{M,X,C,S,CR,CA,SR,SA}, P,CS>, où, P sont les règles de production.

P ≡

CS	→	CS1(m)
CS1(m)	→	CR(c, m), CA(m, c), C(c), CS1(m)
CS1(m)	→	SR(m, s), SA(s, m), S(s), CS1(m)
CS1(m)	→	M(m), X(x)

Dans cette approche, l'évolution de l'architecture a été décrite par des règles conditionnelles de réécriture de graphe. La condition d'une règle décrit une supposition sur l'état des composants impliqués. Pour l'exemple donné, l'auteur suppose que les entités externes aient une variable booléenne newc qui se met à vrai pour créer un client, et ce dernier a une variable booléenne leave qui doit se mettre à vrai quand il veut quitter le système.
Ainsi, la création et la suppression des clients peuvent être décrites par les règles suivantes:

1-	X(x), M(m), x.newc = true	→	X(x'), M(m), CR(c, m), CA(m, c), C(c)
2-	CR(c, m), CA(m, c), C(c), c.leave = true	→	Ø

La première règle crée un client et ses liens vers un manager existant; notant qu'elle remplace l'entité externe originale par une nouvelle (si x n'était pas enlevé, cette règle peut être immédiatement réappliquée, conduisant possiblement à un comportement infini). Quant à la deuxième règle, elle supprime un client et ses liens.

Cette approche propose de spécifier la topologie du système visuellement et indépendamment d'une spécification abstraite pour assurer que l'évolution de l'architecture est conforme à son style. La séparation entre le comportement individuel des composants et ses coordinations facilite le contrôle des propriétés globales du système. Ce qui exprime bien le changement structurel réalisé. Cependant, certaines conditions logiques ne peuvent pas être exprimées par ce type de règles (comme l'absence de liens entre deux composants) et nous ne pouvons pas raisonner sur certaines propriétés (comme le nombre d'instances d'un composant).

3.3.2. Approche de Michel Wermelinger

Dans [WER98] et [WER99] l'auteur a utilisé CHAM (Chemical Abstract Machine) pour la spécification de l'évolution des architectures. CHAM est une technique de description formelle décrivant l'état d'un système par un ensemble d'éléments appelés molécules et les opérations réalisées par le système comme une séquence de réactions entre ces molécules. Les réactions

possibles sont données par des règles de la forme : M_1, M_2,….. , M_i → M_1', M_2',……, M_j. (Où, $M_{1..i}$', $M_{1..j}$' sont des molécules)
Une règle peut être appliquée si l'état du système contient les molécules données à gauche, en remplaçant ces molécules par ceux de la droite.

L'auteur a adapté l'approche proposée dans [MET96] qui s'adapte bien au CHAM. Ainsi, il a spécifié le style architectural et les opérations de reconfiguration par deux CHAMs différentes: CHAM de création pour imposer les contraintes globales du système (par exemple le nombre maximal de composants) et CHAM de l'évolution dont les règles de réactions décrivent comment se fait la transformation de la configuration d'une architecture.

Cette méthode a été illustrée par un exemple de système client/serveur avec un manager central. Dans ce système, le client envoie une requête vers le manager qui à son tour l'envoie vers un serveur disponible. La réponse du serveur est renvoyée au client par l'intermédiaire du manager (voir la figure (3.2).

Molecule	:=	Component | Link | Command
Component	:=	Id :Type
Type	:=	C | M | S
Link	:=	Id—Id
Command	:=	cc(Component) | rc(Id)

Fig.3.2.a. La définition de la structure des molécules par une grammaire.

D'après cette figure, il est clair que l'auteur a utilisé la notation usuelle de typage de composant c:T ou lieu de la notation T(c) utilisé dans [MET96]. Aussi, entre deux composants, il existe un seul lien.

cc(m :M)	→	c :C, c — m, cc(m :M)
cc(m :M)	→	s :S, m — s, cc(m :M)
s :S, cc(m :M)	→	s :S, m :M

Fig.3.2.b. CHAM spécifiant le style architectural client/serveur (CHAM de création).

cc(c :C), m :M	→	c :C, c—m , m:M
cc(s :S), m :M	→	s :S, m—s, m:M
rc(c), c :C, c—m	→	
s' :S, rc(s), s :S, m—s	→	s':S
m :M, rc(m), cc(m' :M)	→	m' :M
m—s, m' :M	→	m'—s, m':M
c—m, m' :M	→	c—m', m':M

Fig.3.2.c. Spécification de l'évolution (CHAM de l'évolution).

Fig.3.2. Spécification d'un système client\serveur par CHAM.

Comme le montre la figure 3.2.b, la configuration initiale contient juste la commande cc() de création avec le nom du manager, la première règle ajoute un client avec son lien, la deuxième ajoute un serveur et son lien, et la dernière crée le manager si un serveur a été précédemment créé.

Les quatre premières règles de la figure 3.2.c spécifient les opérations de création et de suppression de clients et serveurs. La cinquième règle spécifie la substitution du manager qui est indiquée par une paire de commandes de création/suppression. Et les deux dernières règles reconnectent les clients et serveurs existants au nouveau manager.

Afin d'imposer un ordre d'exécution entre les différentes actions de reconfiguration et par conséquent, assurer la consistance du système lors de la reconfiguration dynamique, l'auteur a adopté les quatre commandes primitives utilisées pour créer, supprimer des composants ou des liens : create(), delete(), link(), unlink(). Ainsi, la syntaxe des molécules a été étendu par : Command := create(Component) | delete(Id) | link(Id, Id) | unlink(Id, Id), et le CHAM de reconfiguration, comme le montre la figure 3.3, devient :

cc(c :C), m :M	→	c :C, c—m , m :M , create(c:C), link(c, m)
cc(s :S), m :M	→	s :S, m—s, m :M , create(s:S), link(m, s)
rc(c), c :C, c—m	→	delete(c), unlink(c, m)
s' :S, rc(s), s :S, m—s	→	s':S, delete(s), unlink(m, s)
m:M, rc(m), cc(m' :M)	→	m':M, delete(m), create(m' :M)
m—s, m':M	→	m'—s, m':M, unlink(m, s), link(m', s)
c—m, m':M	→	c—m', m':M, unlink(c, m), link(c, m')

Fig.3.3. CHAM de reconfiguration dynamique du système client/serveur.

Avec un tel CHAM, des règles comme « composant à enlever ne doit avoir aucun raccordement » et « un composant doit être créé avant qu'il puisse être relié » peuvent être respectées. Dans ce cas, à partir d'une solution initiale : « m:M, m—s, s:S, cc(c:C), rc(m), cc(m':M) », la séquence d'exécution suivante peut être exécutée (d'autres sont possibles) :
unlink(m, s), create(c:C), delete(m), create(m':M), link(c, m'), link(m', s).

Malgré que la spécification avec cette technique est facile en utilisant des concepts simples (molécules, règles de réécriture), chaque CHAM présente sa propre syntaxe et exigences (la définition des molécules est laissée au concepteur). Ceci exige plus d'explications et ne permet pas la réutilisation des caractéristiques (c'est mieux d'avoir une représentation fixe pour les concepts les plus importants comme les composants et les connecteurs). Également, les propriétés relatives aux composants ne peuvent pas être spécifiées. Et finalement, la preuve de correction des propriétés du style se fait manuellement par induction, en assurant que chaque règle du CHAM de reconfiguration préserve ces propriétés invariantes (aucun outil formel n'a été utilisé pour tester la correction de la spécification écrite dans CHAM).

3.3.3. Approche de I.Loulou et al

Le but principal des travaux présentés dans [LOU05] est de vérifier la reconfiguration de l'architecture en exprimant sa conformité par rapport à un style architectural.
Les auteurs ont essayé d'offrir au concepteur une solution lui permettant de raisonner sur des systèmes complexes en les caractérisant à un haut nivaux d'abstraction, et d'autres part d'exploiter des modèles récurrents d'organisation de systèmes connus sous le nom de styles architecturaux. Ceci facilite le processus de conception en recourant à des solutions connues pour certaines classes de problèmes.

L'approche proposée dans ce contexte profite des avantages de l'aptitude expressive des langages fonctionnels. Elle exploite également les avantages qu'offrent les grammaires de graphes notamment en termes de manipulation d'architectures. Elle est basée sur une notation mixte qu'améliore la lisibilité, réduit la complexité de la description et qui prend en charge également la dynamique du système.

Le style architectural d'un système logiciel a été décrit avec la notation Z et les opérations de reconfiguration à travers des règles de réécriture de graphes. Ces règles sont exprimées par une intégration du langage fonctionnel Z avec la notation Δ (notation inspirée des grammaires de graphes). Cette intégration (la notation mixte), comme la montre la figure 3.4.a ci-après, intègre une partie graphique et une partie fonctionnelle. La partie graphique est exprimée par la notation Δ (grammaire de graphe). Quant à la partie fonctionnelle, elle exprime les pré-conditions imposées aux règles et les actions à entreprendre avec la notation Z.

Ces règles prennent en considération les contraintes structurelles et fonctionnelles définies pour le système dans leurs conditions d'application, ce qui assure sa consistance durant son évolution. Toute la sémantique est décrite en notation Z. Ainsi, chaque règle a été traduite par un schéma Z (voir la figure 3.4.b ci-après). Obtenant donc une approche qu'assure la cohérence entre l'aspect statique et l'aspect dynamique.

Cependant, cette approche a des limites. En effet, elle ne prend pas en considération l'aspect coordination entre les différentes opérations de reconfiguration (les politiques d'adaptation). Aussi, la vérification des règles se fait manuellement (aucun outil n'a été utilisé pour tester la cohérence de l'évolution d'un système). En plus, le choix d'un style architectural adéquat pour représenter l'architecture de communication du système pose un problème de correspondance entre les différents éléments. Également, la description de l'aspect comportemental des composants logiciels n'a pas été prise en considération. Enfin, comme toute technique formelle, elle exige une certaine connaissance mathématique.

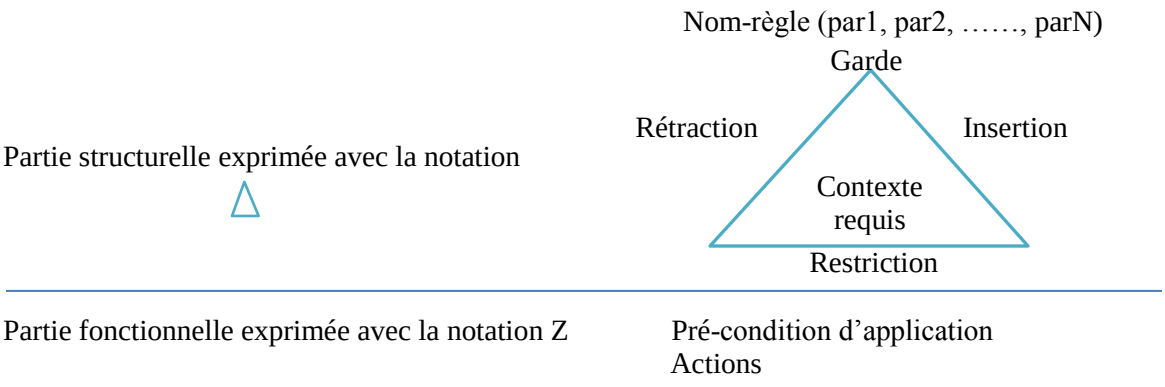


Fig.3.4.a. La notation mixte.

Nom-règle

par1? ; par2? ; ; parN?

△ System

Pré-conditions

Contraintes fonctionnelles: conditions sur les paramètres; conditions sur les attributs des noeuds; etc

Contraintes structurelles: identification de gl (= contexte requis ∪ rétraction / restriction) dans le graphe du système

Actions

Operations sur les ensembles des nœuds/arcs (insertion/rétraction)

Fig.3.4.b. Sémantique d'une règle de reconfiguration décrite via un schéma Z.

Fig.3.4. Représentation d'une règle de reconfiguration selon l'approche de I.Loulou et al.

3.3.4. Approche de Ji Zhang et Betty H.C. Cheng

Dans [ZHA06], les auteurs ont introduit une approche pour spécifier formellement l'adaptation en utilisant la logique temporelle.

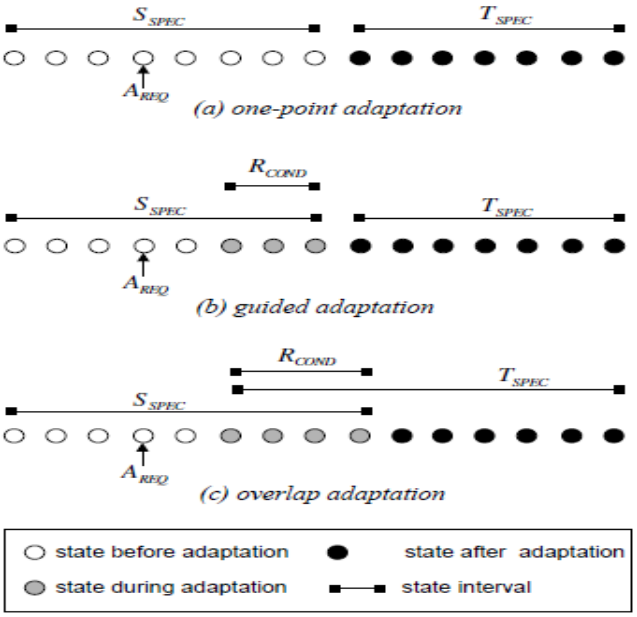
Avec cette approche un programme adaptatif est vu comme une composition d'un nombre fini de programmes considérés dans un état correct.

Pour spécifier le comportement de l'adaptation, LTL (linear temporal logic) a été étendu en A-LTL (Adapt operator-extended LTL) par l'operateur ($\overset{\Omega}{\rightarrow}$).

Trois sémantiques de l'adaptation ont été définies pour spécifier :

- Les variantes de l'adaptation d'un programme source donné vers un programme cible donné.
- Les invariants de l'adaptation.
- Et finalement, le comportement complet d'un système adaptatif (la composition de ses programmes) par la combinaison des variantes de l'adaptation avec les invariants. Le programme source et le programme cible, ainsi que ses contraintes durant l'adaptation ont été spécifié par des formules en LTL et l'adaptation a été exprimée en A-LTL.

Informellement, un programme satisfait $\Phi \xrightarrow{\Omega} \Psi$ (Φ, Ψ et Ω sont trois formules exprimées en logique temporelle) signifie que le programme initialement vérifie Φ. Dans un certain état A il arrête de satisfaire Φ et il commence à satisfaire Ψ dans le prochain état B. La notation Ω peut être utilisée pour exprimer les propriétés du système avant et après l'adaptation.



$$(S_{SPEC} \wedge \Diamond A_{REQ}) \xrightarrow{\Omega} T_{SPEC} \quad (1)$$

$$(S_{SPEC} \wedge (\Diamond A_{REQ} \xrightarrow{\Omega 1} R_{COND})) \xrightarrow{\Omega 2} T_{SPEC} \quad (2)$$

$$((S_{SPEC} \wedge (\Diamond A_{REQ} \xrightarrow{\Omega 1} R_{COND})) \xrightarrow{\Omega 2} true) \wedge (\Diamond A_{REQ} \xrightarrow{\Omega 1} (T_{SPEC} \wedge (R_{COND} \xrightarrow{\Omega 2} true))). \quad (3)$$

Fig.3.5. La sémantique d'adaptation avec A-LTL.

L'utilisation de la logique temporelle permet d'exécuter de nombreuses analyses automatisées d'un programme adaptatif, comme la spécification et l'utilisation du modèle checking pour la vérification de la correction du modèle du programme.

Bien que l'utilisation de A-LTL est simple et suffisamment efficace pour spécifier le comportement des adaptations, les sémantiques définies décrivent le programme adaptatif dans sa globalité, ce qui nécessite de prendre en considération toutes les propriétés même si elles sont propres à une partie du programme non considéré par l'adaptation. En plus, l'algorithme du modèle checking développé pour la vérification de la correction du modèle a une complexité exponentielle à la longueur des formules et la taille du programme.

3.3.5. Approche de Marianne Simonot et al

Les travaux présentés dans [SIM08], [SIM09] et [APO09] se situent dans le contexte de la programmation par composants. La spécification a été faite à la base du modèle de composant fractal, qui permet la conception et la programmation de systèmes reconfigurables. Par la suite, les auteurs ont choisi l'atelier de développement formel FOCAL pour la preuve des propriétés liées aux aspects métiers de l'application et aux aspects de reconfigurations.

- La première étape de ces travaux a permis de produire une spécification formelle du modèle fractal nommée Fracl, ceci par :

a- La définition de ses entités essentielles (composants, ports, interfaces, etc.), des fonctions et contraintes portant sur ces entités et des liens qui les assemblent.

b- La définition de chaque méthode d'introspection ou de reconfiguration par la donnée d'une pré-condition et d'une post-condition exprimant respectivement, le domaine de définition et la modification de l'état (produite par l'exécution de la méthode) d'une application.

La figure 3.6 montre un exemple de méthodes d'introspection : un port peut découvrir son propriétaire (méthode getFcItfOwner()) et un composant peut connaître l'ensemble de ses ports (méthode getFcInterfaces()).

```
P (Port) getFcInterfaces  (Comp c );
pre   : true
post : return ports(c)
```

```
Comp  getFcItfOwner (Port p );
Pre   : true
post : return comp(p)
```

Fig.3.6. Spécification de deux méthodes d'introspection en Fracl.

Les méthodes de reconfiguration dynamique concernent essentiellement les liaisons entre ports et l'état des composants. À titre d'exemple, la figure 3.7 donne la spécification de bindFc (c, n, ps) qui lie le port client de nom n du composant c au port serveur ps. Cette méthode est spécifiée à l'aide des fonctions :

sameNSpace(c, m), qui est vraie si deux composants c et m sont soit dans une même boite, soit dans aucune boite, et

portOfName(n, c), qui donne le port de nom n d'un composant c.

```
Void bindFc (Comp c, Pname n, PortS ps)
Pre : ∃ pc.portOfName(n, c) = Pc                  //  n est le nom d'un port de c
        ∧ Pc ∈ PortC ∧ Pc ∉ dom(bindings)         //  qui est un port client non lié
        ∧ Sig(Pc) ⊆ sig(Pc) ∧ status( c)=stopped
        ∧ SameNSpace(c,comp(Ps))
Post: bindings:= bindings ∪{(portOfName(n,c),Ps)}
```

Fig.3.7. Spécification de la méthode de reconfiguration bind.

➢ La seconde étape consiste à réaliser un encodage de Fracl en FOCAL en s'appuyant sur la notion des espèces Focal. Ceci, en réalisant :

- Un modèle modulaire par un encodage superficiel de la notion de composant qui permet de rendre compte de leur aspect fonctionnel.
- Et un modèle d'architecture de composants par un encodage en profondeur de manière à exprimer des propriétés d'introspection et de reconfiguration dynamique. Ce modèle est basé sur la spécification formelle des notions de port, de composant, des invariants, ainsi que les méthodes de contrôle pour l'introspection et la reconfiguration dynamique.

➢ La dernière étape dans la mise en place de cette modélisation a été d'intégrer les deux modèles précédemment exposés en un seul dans le but d'obtenir un environnement de

spécification et de preuve ayant la capacité d'exprimer des propriétés mélangeant les aspects métiers et architecturaux des applications.

Le cadre formel proposé possède beaucoup d'avantages, notamment :

- Il autorise l'expression et l'analyse des propriétés portant autant sur des aspects métiers d'une application que sur sa reconfiguration dynamique et son introspection.
- Il est plus détaillé puisqu'il définit des concepts comme la compatibilité entre les interfaces, l'état d'un composant et le typage.
- Finalement, l'utilisation de Focal permet de faire des preuves.

Contre les avantages présentés, quelques limites apparaissent, notamment :

- La notion de composant correspond au composant primitif de Fractal qui ne prend pas en charge ni la notion d'attribut de composant, ni les connexions multiples pour une interface.
- En plus, le modèle proposé exprime les propriétés minimales à valider par tout contrôleur qui implante la même capacité de contrôle du modèle fractal, négligeant ainsi ceux qui implantent une capacité de contrôle particulière.
- Finalement, le cadre formel proposé exige une connaissance du modèle de composant fractal et certaines connaissances mathématiques et nécessite un certain niveau d'expertise pour la maitrise de la complexité de l'atelier focal.

3.3.6. Synthèse

L'étude menée dans la section 3.3, nous a permis de constater que les travaux réalisés appliquant les méthodes formelles pour la vérification des reconfigurations dynamiques sont articulés autour de deux formalismes: les grammaires de graphes et les langages fonctionnels.

Dans le contexte du premier formalisme, la notion de graphe a été utilisée pour la représentation de la structure du système, étant donné qu'elle représente un moyen mathématique simple. Avec ce formalisme, l'architecture du système a été représenté dans [MET96] par un graphe et le style architectural par une grammaire de graphe contexte-libre. Quant aux opérations de reconfiguration, elles sont décrites simplement par des règles de réécritures compréhensibles qui expriment bien le changement structurel réalisé. Cependant, certaines conditions logiques ne peuvent pas être exprimées par ce type de règles (comme l'absence de liens entre deux composants) et nous ne pouvons pas raisonner sur certaines propriétés (comme le nombre d'instances d'un composant). De même, [WER99] a proposé d'utiliser CHAM qu'est une grammaire de graphe < N, T, P, O> où, N et T sont des molécules, P les règle de réaction et O la configuration initiale. Dans cette approche, le style architectural a été représenté par CHAM de création et les reconfigurations par les règles de réaction du CHAM de l'évolution.
Néanmoins, chaque CHAM a sa propre syntaxe et besoins, ce qui ne permet pas la réutilisation des caractéristiques et exige des explications supplémentaires. En plus, les propriétés relatives aux composants ne peuvent pas être spécifiées et la preuve de correction des propriétés du système dans ce contexte se fait manuellement (aucun outil formel n'a été utilisé pour tester la correction de la spécification).

Concernant le second formalisme, il consiste a utilisé la logique pour exprimer des conditions de reconfigurations, des propriétés variantes et invariantes représentant l'état du système (pré/post conditions et invariants). Dans ce contexte, un cadre nommé Fracl fondé sur une description axiomatique du modèle de composant Fractal en logique du premier ordre a été présenté dans [SIM08]. Fracl a été encodé par la suite dans l'atelier FOCAL à travers la notion d'espèce. Bien que l'utilisation de FOCAL permet de faire des preuves, cette approche permet d'exprimer les propriétés minimales à valider par tout système qui implante la même capacité de contrôle du modèle fractal, oubliant ainsi ceux qui implantent une capacité de contrôle particulière. D'autre part, la logique temporelle linéaire (LTL) a été étendue dans [ZHA06] en A-LTL pour permettre l'expression des propriétés relatives aux reconfigurations dynamiques. Cependant, les changements structuraux non pas été pris en considération et le nombre de configuration du système doit être connu à l'avance pour permettre la vérification par le modèle checking, ce qu'ignore les changements non prévus au début. De plus, cette logique est un formalisme difficile à comprendre, ce qui requiert des experts qualifiés.

Pour combler quelques insuffisances des deux formalismes, les auteurs ont essayé dans [LOU05] de combiner les avantages de ces deux derniers. L'approche proposée cherche à éviter les problèmes de lisibilité des langages fonctionnels en les combinant avec les transformations de graphe. Dans cette approche le style architectural d'un système logiciel est décrit avec la notation *Z* et les opérations de reconfiguration à travers des règles de réécriture de graphe. Cette intégration (notation mixte) offre au développeur une technique de spécification facile à comprendre et qui permet l'expression des propriétés complexes.
Malgré ça, cette approche n'a pas pris en considération l'aspect fonctionnel des composants (ajout/suppression des interfaces), l'aspect coordination entre les différentes opérations de reconfiguration (les politiques d'adaptation) est toujours absent et la preuve de cohérence de l'évolution a été faite manuellement par rapport à chaque règle de reconfiguration.

Nous avons noté de cette analyse que peu de travaux présentent un processus clair, un environnement de spécifications complet qui prend en compte l'aspect structural et comportemental des systèmes adaptatifs et permettant la preuve automatique de la cohérence de la spécification.

3.4. Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art des techniques formelles existantes ainsi que quelques approches permettant la description de la dynamique des systèmes logiciels. Les avantages ainsi que les limites de ces approches ont été mentionnés. Afin de combler le manque de ces approches, nous proposerons, dans le chapitre suivant notre approche, qui consiste à apporter une solution simple pour la modélisation de la dynamique des systèmes logiciels, tout avec prise en compte des deux aspects, architectural et fonctionnel.

Partie II : Contributions

CHAPITRE 4

Modélisation des reconfigurations dynamiques

4.1. Introduction

Dans ce chapitre, nous nous concentrons sur la modélisation des architectures logicielles des systèmes dynamiques.

Le plus souvent, les travaux présentés dans [LOU05], [MET96] et [WER99] qui cherchent à modéliser les propriétés liées à la reconfiguration dynamique d'une architecture à composants, élaborent des modèles purement architecturaux et ne permettent pas de traiter les aspects métiers offerts ou requis par les composants. Notre approche cherche à compléter ces travaux, et nous essayons d'élaborer un modèle où l'on peut à la fois exprimer des propriétés sur les aspects architecturaux et sur les aspects fonctionnels, tout en permettant la représentation des politiques d'adaptation.

Dans ce qui suit, nous présenterons notre approche de modélisation d'un système adaptatif par composants.
Nous proposons dans notre modèle de prendre en charge les deux grandes parties relatives aux composants:

- L'aspect fonctionnel (comportemental), qui modélise la notion de code exécutable à l'intérieur d'un composant (contraintes sur les interfaces, attributs des composants…etc).
- L'aspect architectural, qui modélise les liaisons entre composants (contraintes structurales).

Suite à notre étude menée dans la section 3.3 du chapitre 3, nous proposons de simplifier et compléter l'approche proposée dans [LOU05]. Nous essayons donc de combiner les deux formalismes, de transformation de graphes et le formalisme fonctionnel. Ceci, en exploitant leurs avantages et en essayant de combler leurs limites, pour obtenir un environnement de spécification complet (voir figure 4.1) qui prend en compte l'aspect structural et comportemental du système.

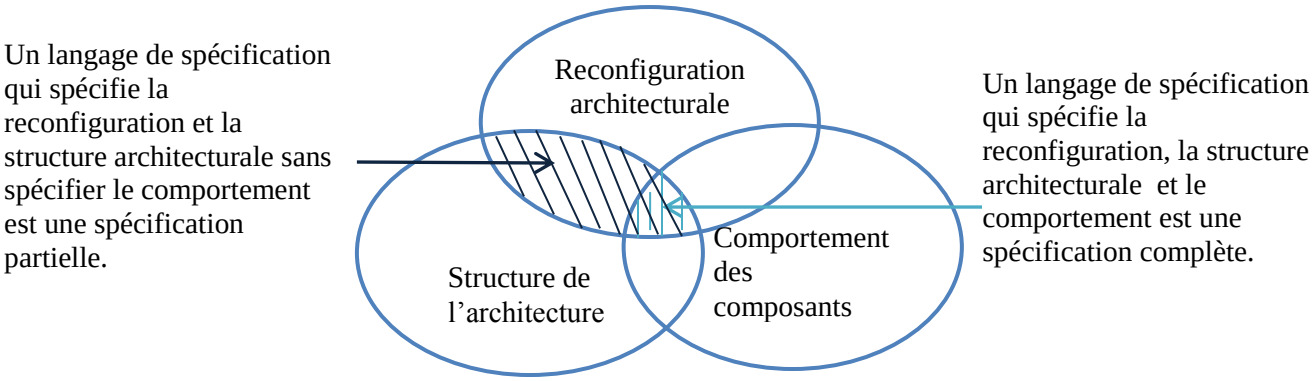


Fig.4.1. Spécification complète et partielle des architectures logicielles dynamiques [BRA04].

4.2. L'approche proposée

4.2.1. Définition du modèle

Notre approche permet de décrire la reconfiguration dynamique des architectures logicielles, par la donnée de trois sous modèles, possédant chacun un symbole permettant de le distinguer. Ces trois sous modèles décrivent :

- La structure architecturale du système, en donnant l'ensemble des types de composants pouvant intervenir dans le système, les types de connexions existantes entre eux, ainsi que l'ensemble des propriétés architecturales et fonctionnelles qui doivent être satisfaites par toutes les configurations.

- L'évolution et la dynamique de l'architecture, en termes d'opérations de reconfiguration que peut avoir une architecture tout au long de son cycle de vie, ainsi que les actions à entreprendre pour chaque opération.

- La coordination, par la définition de l'enchaînement et l'ordre d'exécution des différentes opérations de reconfiguration résultantes d'un évènement donné, ainsi que l'ordre d'exécution des actions de chaque opération. Ceci permet de représenter la transformation de la configuration d'un état vers un autre.

Dans notre modélisation, nous nous intéressons à la dynamique structurelle (ajout et\ou suppression de composants et\ou de connexions) et comportementale (ajout et\ou suppression des interfaces et le remplacement de composants « modification de code »).

Notre modèle tire profit de la puissance des graphes pour décrire la dynamique structurelle et offre une notation graphique qui simplifie et visualise clairement le typage de composants et de connexions. Nous allons le détailler par la suite à travers l'étude de cas du système de contrôle de patients, étudié dans le cadre des travaux de [MET96] et [LOU05].

Représentation des éléments de l'architecture par un graphe.

La notion de graphe utilisée est inspirée du travail présenté dans [MET96], où les nœuds représentent les composants et les arcs dénotent les connecteurs.
Un graphe est défini formellement comme un ensemble de relations $R(e_1,\ldots..,e_n)$, où R est une relation n-aire et e_i est le nom d'une entité.

Dans notre contexte, pour la description d'une configuration de l'architecture, nous considérons deux types de relations : binaire et unaire.

- Une relation binaire $C(e_1,e_2)$ représente un connecteur de nom C (type C) entre les composants e_1 et e_2, où, e_1 est le composant client ayant l'interface requise et e_2 est le composant serveur ayant l'interface fournie.

- Si dans le système e_1 fournit un service à e_2 et e_2 fournit un service à e_1, on doit représenter ces deux connecteurs par deux relations $C'(e_2,e_1)$ et $C(e_1,e_2)$ respectivement.

- De même, une relation unaire U(e) représente le rôle (type) du composant e dans le système.

Dans le graphe, nous représentons une relation unaire par un cercle et une relation binaire par un arc. Nous nous intéressons donc, aux graphes orientés et typés.

Un graphe d'architecture représente une configuration donnée du système décrit comme suit :
$G = < U, B >$ où :
U : L'ensemble de relations unaires (l'ensemble des nœuds). $U \subseteq C \times TC$.
B : L'ensemble de relations binaires (l'ensemble des arcs). $B \subseteq C \times TL \times C$.
C : L'ensemble d'instances de composants.
TC : L'ensemble des types de composants. $TC = \{C_1, C_2, \ldots\ldots, C_m\}$
TL : L'ensemble des types de connecteurs. $TL = \{T_1, T_2, \ldots\ldots.., T_n\}$

- Dans un graphe, pour décrire le type d'un composant, nous adoptons la notation usuelle c:T au lieu de T(c).

- Pour éviter la surcharge du graphe représentant une configuration de l'architecture, nous proposons de représenter toutes les relations du même sens, entre deux composants donnés, par un même connecteur portant un type. C'est-à-dire, dans un graphe, on ne peut pas voir plus de deux arcs entre deux composants (un arc dans chaque sens). Afin de définir les relations d'un même connecteur, nous définissons une fonction Relation comme suit :

Relation : $B \rightarrow \mathbb{P}(Rel)$, où :
$Rel \subseteq Ir \times If$, où, $I = Ir \cup If$, avec:
I : L'ensemble des interfaces des composants.
If : L'ensemble des interfaces fournies.
Ir : L'ensemble des interfaces requises.

- Pour permettre l'expression des propriétés sur les ensembles de composants et de connecteurs portant un type spécifique, nous mettons les instances du même type dans le même ensemble ayant le nom du type.

- L'ensemble des composants sera défini comme suit :

$C = C_1 \cup C_2 \cup \ldots \cup C_i \cup \ldots \cup C_m$; où C_i, $i=\overline{1,m}$ dénote l'ensemble d'instances de composants de type C_i. Ainsi, si par exemple le nombre d'instances de type C_1 ne peut pas dépasser une limite N, ceci s'exprime par : $|C_1| \leq N$, qui indique que le cardinal de l'ensemble C_1 ne peut pas dépasser N.

- De même pour l'ensemble des connecteurs, $B = T_1 \cup T_2 \cup \ldots \cup T_i \cup \ldots \cup T_n$; où T_i, $i=\overline{1,n}$ est l'ensemble des connecteurs de type T_i. Ainsi, nous pouvons exprimer des propriétés sur les différents types de connecteurs.

4.2.1.1. L'architecture du système

Tout d'abord, nous proposons une nouvelle notation (sous modèle) décrite dans la figure 4.2 ci-dessous, pour la représentation de la structure architecturale ainsi que ses contraintes.

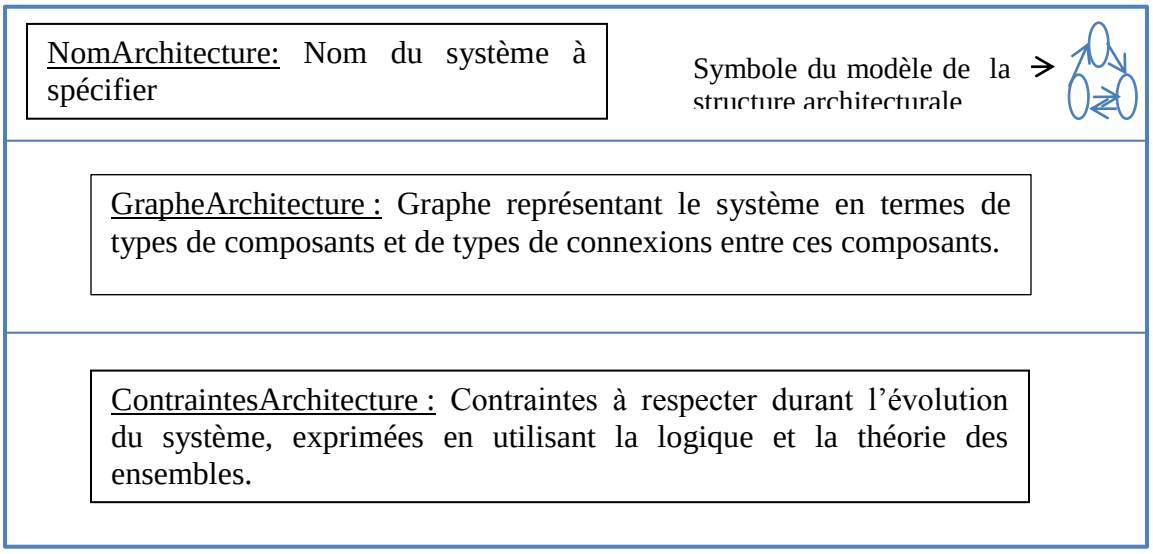


Fig.4.2. Le modèle de la structure architecturale.

Ce modèle est composé de trois parties :

- **NomArchitecture:** Cette partie consiste à donner le nom du système à spécifier.
- **GrapheArchitecture :** Afin de visualiser clairement les différents types de composants et de connexions pouvant exister dans le système, nous spécifions la structure du système à travers un graphe. Nous décrivons donc chaque type de composants par un nœud, et chaque type de connexions par un arc.
- **ContraintesArchitecture:** Pour préciser les propriétés qui doivent être préservées pendant et après l'évolution du système (invariant).

4.2.1.2. Les opérations de reconfiguration

Pour la préservation du fonctionnement du système pendant et après l'évolution de l'architecture, nous proposons un deuxième modèle permettant de décrire les différentes opérations de reconfiguration assurant l'évolution de l'architecture d'une application en termes d'ajout et/ou de suppression de composants, d'interfaces et de connexions. Ce modèle permet la transformation du système d'une configuration vers une autre, tout en exprimant les changements tant structurels que fonctionnels (comportementaux).

Ce modèle est inspiré des modèles de transformation de graphes [MET96] [WER99] et plus précisément de la notation mixte proposée par I.LOULOU *et al* dans [LOU05].

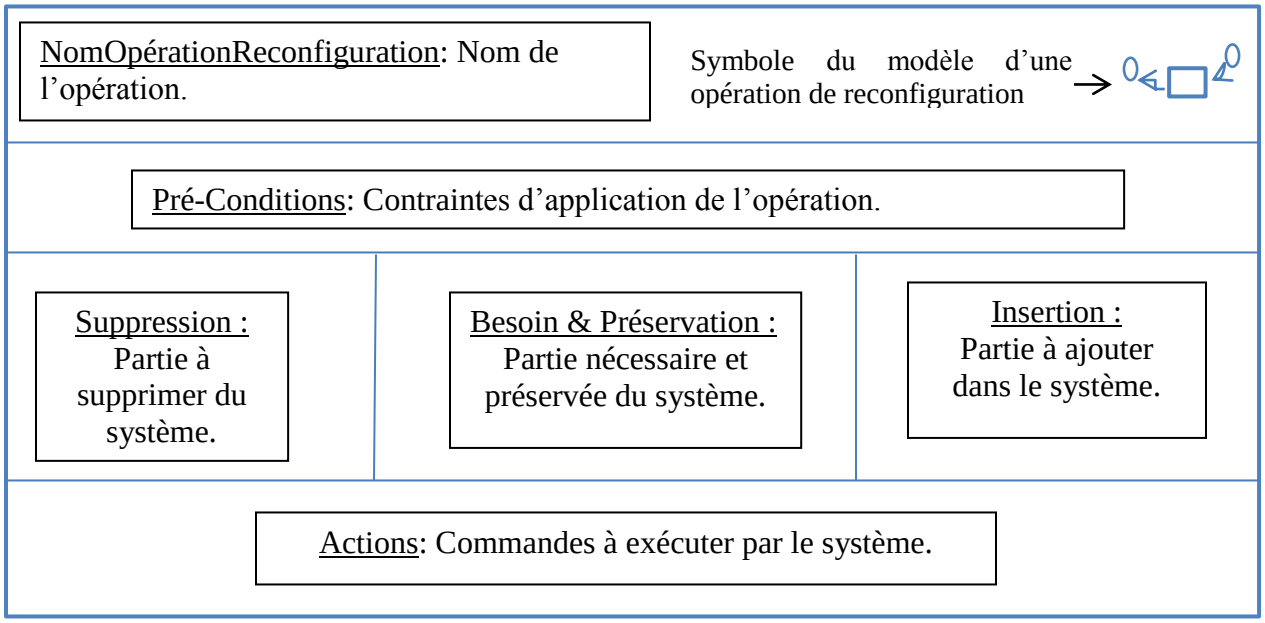


Fig.4.3. Le modèle d'une opération de reconfiguration.

Comme le montre la figure 4.3 ci-dessus, ce modèle est composé de six parties :

- « NomOpérationReconfiguration»: Pour exprimer le nom de l'opération de reconfiguration à exécuter.
- « Pré-Conditions »: Cette partie est utilisée pour énoncer les contraintes nécessaires à l'application de l'opération de reconfiguration (conditions sur le nombre d'instances, sur les attributs des composants, …etc.). Ces contraintes permettent d'assurer une évolution conforme aux propriétés invariantes présentées dans la partie « ContraintesArchitecture » du modèle de la structure architecturale.
- « Suppression »: Pour préciser la partie du système à supprimer pendant l'exécution de l'opération de reconfiguration.
- « Insertion »: Pour préciser la partie à ajouter dans le système durant l'exécution de l'opération de reconfiguration.
- « Besoin & Préservation »: Pour préciser la partie dont l'opération dépend et qui est non modifiable durant l'exécution de l'opération de reconfiguration.
- « Actions »: Cette partie est utilisée pour citer les commandes à entreprendre pour exécuter l'opération de reconfiguration (opérations sur les composants/interfaces/connecteurs, à savoir : insertion, suppression, remplacement).

Les commandes (actions) de base sont :

- InsertComp(T(c)) : Désigne l'insertion d'un composant c de type T.
- InsertConnect(T'(c,c')): Désigne l'insertion d'un connecteur de type T' entre les composants c et c'.

- InsertInterf(T(c),T'(f)) : Désigne l'insertion d'une interface f de type T' au composant c de type T (le type T' exprime le rôle de l'interface ; fournie ou requise).
- SuppComp(T(c)): Désigne la suppression d'un composant c de type T.
- SuppConnect(T'(c,c')) : Désigne la suppression d'un connecteur de type T' entre les composants c et c'.
- SuppInterf(T(c), T'(f)) : Désigne la suppression d'une interface f de type T' du composant c de type T.
- MigratComp(T(c), S1, S2): Désigne la migration d'un composant c de type T du site S1 vers le site S2 ; où, un site est modélisé dans le graphe d'architecture par un sous ensemble de composants (le cas d'un système distribué).

Remarques

- L'opération de remplacement d'un composant T(c) par un autre T(c') se fait par l'imbrication des deux commandes SuppComp(T(c)) et InsertComp(T(c')) successivement. De même pour l'opération de remplacement d'une interface T'(f) d'un composant T(c) par une autre T'(f'), elle peut se faire par l'imbrication des deux commandes SuppInterf(T(c), T'(f)) et InsertInterf(T(c), T'(f')) successivement.

- Les trois parties : « Suppression », « Insertion », « Besoins & Préservation » sont représentées par la notion de graphe définie précédemment pour la représentation d'une configuration de l'architecture.

- Les différentes actions sont décrites indépendamment de ce modèle sous forme de paire pré/post-conditions.

- Les propriétés architecturales et fonctionnelles caractérisant le système après l'exécution d'une opération de reconfiguration (changements des attributs d'un composant...etc.) peuvent être décrites sous forme d'actions.

- Les différentes contraintes sont décrites en utilisant la logique de prédicats du premier ordre et la théorie des ensembles.

Avantages

La représentation d'une opération de reconfiguration par notre modèle rend facile la tâche de spécification et la compréhension de l'opération par des expressions logiques simples. En effet, l'introduction de la partie « pré-conditions » évite les problèmes rencontrés par une notation seulement graphique en ce qui concerne les conditions non structurelles (conditions sur les attributs des éléments du graphe par exemple, conditions contenant des disjonctions et toutes autres contraintes fonctionnelles). De plus, la partie « Action » permet de modéliser de façon modulaire les différentes opérations de reconfiguration (réutilisation des actions). Elle permet aussi, la représentation des changements de l'ensemble des services (les interfaces) d'un composant donné. Ainsi, les actions d'ordre fonctionnel (comme le remplacement d'un

composant par un autre qui étend ses fonctionnalités par l'ajout de nouvelles interfaces) peuvent être exprimées avec cette partie.

4.2.1.3. La coordination

A moyen terme, un système adaptatif est guidé par un modèle de contexte et des politiques d'adaptation, qui permettent de définir vers quel état il va évoluer. Ces politiques sont décrites sous la forme E-C-A (Evènement si Condition alors Action), à travers un ensemble d'opérations de reconfiguration (voir section 1.3.4 du chapitre 1). Une fois que le système sait vers quel état il doit évoluer, il faut définir de quelle façon il doit le faire (plan d'exécution de la reconfiguration), et exécuter l'adaptation (exécution) [PAY06]. Nous spécifions donc dans cette section les deux étapes composant la coordination.

.

A. L'enchainement entre les différentes opérations de reconfiguration (Plan d'exécution)

Dans cette section, Nous traitons essentiellement le problème des reconfigurations complexes, déclenchées suite à un évènement donné du système modélisé.
Ces reconfigurations sont composées d'opérations plus simples modélisées à travers le modèle des opérations de reconfiguration.

La description de l'enchainement entre ces différentes opérations de reconfiguration s'avère donc nécessaire, car la description de l'architecture et des opérations de reconfiguration est insuffisante pour spécifier l'évolution de l'architecture logicielle d'un système dynamique. Pour cela, nous proposons de décrire la coordination entre les différentes opérations d'une reconfiguration, par un graphe de dépendances fonctionnelles, ceci permet d'imposer un ordre d'exécution entre les différentes opérations (plan d'exécution).

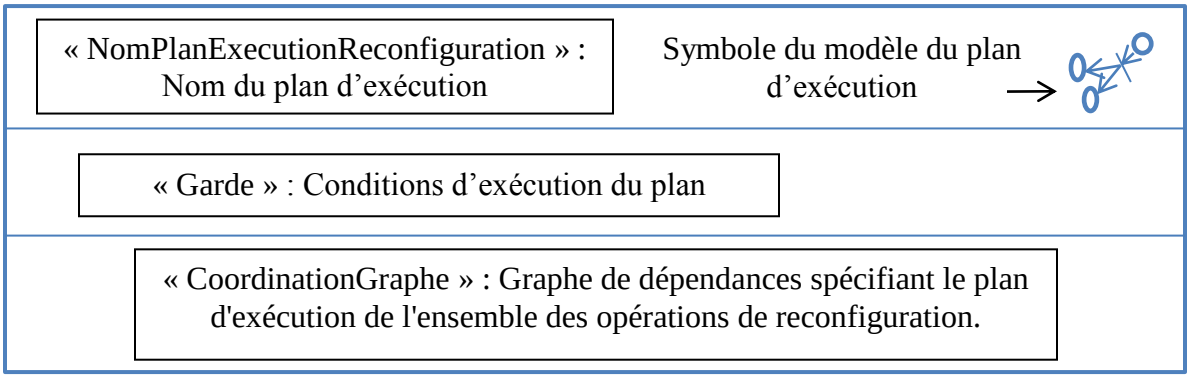


Fig.4.4. Le modèle du plan d'exécution d'une reconfiguration.

Le modèle du plan d'exécution d'une reconfiguration est composé de trois parties :

- « NomPlanExecutionReconfiguration »: Pour préciser le nom de la reconfiguration donnée par le plan.
- « Garde » : Cette partie est utilisée pour citer les conditions d'exécution du plan.

- « CoordinationGraphe »: Cette partie spécifie l'ordre d'exécution des opérations impliquées par le plan, à travers un graphe de dépendances constitué de deux niveaux (voir figure 4.5 ci-dessous), à savoir :

➢ **Niveau global:** Ce niveau est donné par un graphe, où, les nœuds représentent les opérations et les arcs représentent les dépendances entre ces opérations.
Exemple : Dans la figure ci-dessous l'opération B dépend de l'opération A, c'est-à-dire que B s'exécute après l'exécution de A.

➢ **Niveau local:** Ce niveau est constitué d'un ensemble de graphes locaux. À chaque nœud du niveau global correspond un graphe local, et à chaque arc correspond un arc au niveau local. Un graphe de dépendances local présente l'ordre d'exécution des actions de l'opération correspondante au niveau global.

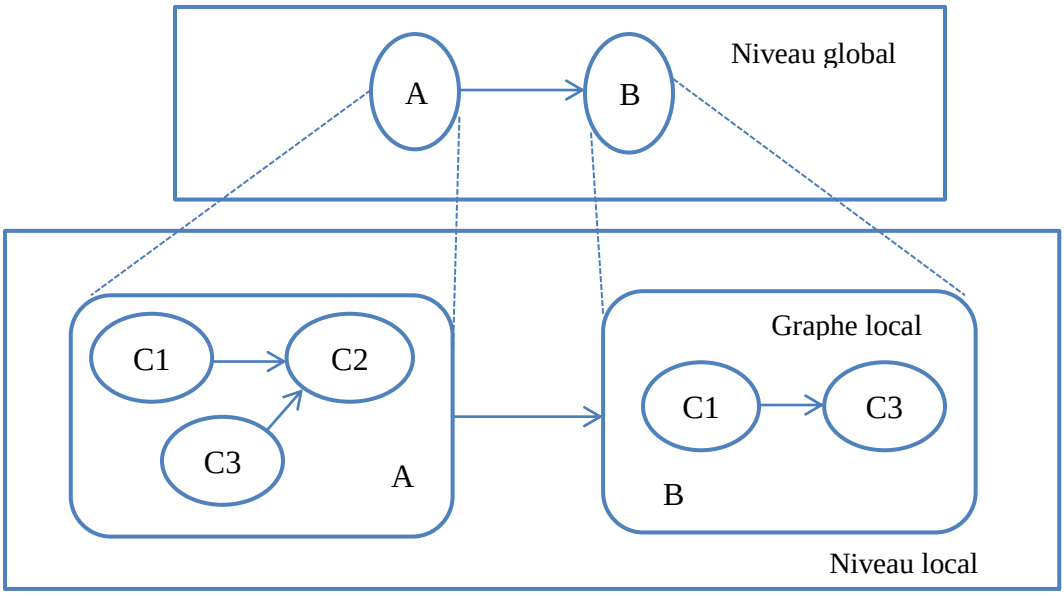


Fig.4.5. Graphe de coordination des opérations d'une reconfiguration.

B. Gestion de l'état des composants pour l'exécution des opérations (exécution)

À l'exécution, l'application connaît l'ordre d'exécution des opérations d'adaptation. Tous les composants de l'application ne sont pas forcément touchés par l'adaptation. Néanmoins, il faut s'assurer de la cohérence de l'application dans sa globalité avant de pouvoir lancer l'adaptation (exécution de l'adaptation de façon que l'application soit toujours dans un état cohérent) [PAY06].

De ce fait, l'exécution des reconfigurations dynamiques doit être synchronisée avec l'exécution fonctionnelle du système. Par exemple, il ne doit pas être possible pour un composant, de continuer à invoquer des méthodes de l'interface fonctionnelle d'un autre composant, alors qu'il est en train d'être déconnecté. Il faut donc, figer l'état des composants dépendants avant une reconfiguration (voir la section 2.3.4 du chapitre 2), avec une assurance d'un maximum de services.

Pour assurer la synchronisation et par conséquent la cohérence des reconfigurations, il est nécessaire d'imposer un état à certains composants dépendants du composant en reconfiguration avant l'exécution de l'adaptation. Pour cela, nous proposons d'adopter le protocole proposé dans [PAL99] et spécifié dans [COR01] qui permet d'imposer un état abstrait à certains composants par un composant gestionnaire de la reconfiguration.
Pour cela, nous ajoutons pour certaines commandes (actions), dans la partie pré-condition, des propriétés exprimant respectivement l'état de l'ensemble des composants impliqués par l'opération avant l'exécution de la commande (état abstrait).

Pour déterminer l'état d'un composant, nous définissons donc la fonction, Cetat : C→Etat qui associe à un composant de l'ensemble C un état de l'ensemble Etat = {Active, Passive, Frozen}. Ces états sont imposés à deux commandes seulement comme suit:

- SuppConnect(T'(c, c')) : Cette commande peut s'exécuter si la connexion T'(c, c') n'est utilisé par aucun composant, c'est-à-dire que la pré-condition est la suivante.
  Cetat(c) = passive.

- MigratComp(T(c), S1, S2): Tous les connecteurs entrants au composant T(c) doivent être vides (aucune demande de service) avant d'exécuter cette commande. C'est-à-dire qu'il faut leur attribuer l'état passive et au composant migré l'état frozen.

Remarques

- Pour la commande SuppComp(T(c)), T(c) doit être isolé, ce qui signifie que toutes les liaisons de ce composant doivent être supprimées avant de pouvoir lancer cette commande.

- Pour la commande SuppInterf(T(c), T'(f)), il ne doit pas y avoir un connecteur dont f fait partie d'une relation de son ensemble de relations. Dans ce cas, si T' désigne une interface fournie et si un composant client a besoin de cette interface, c'est au gestionnaire de reconfiguration de gérer la situation, peut être par la redirection du connecteur vers un autre composant serveur ayant cette interface. Pour notre modélisation, nous resterons indépendants de l'algorithme de synchronisation implémenté par le système, c'est-à-dire nous spécifions les actions à exécuter mais pas comment les exécuter.

4.2.2. Fonctions et actions en rapport avec la reconfiguration dynamique

Nous présentons maintenant quelques fonctions et actions qui rendent compte de quelques aspects de contrôle des reconfigurations dynamiques.
Si nous résumons notre modélisation, nous trouvons les entités suivantes :

C: l'ensemble des instances de composants
TC : l'ensemble des types de composants
U : C×TC // l'ensemble des composants typés
TL : l'ensemble des types de connecteurs
B : C×TL×C // l'ensemble des connecteurs typés
If : l'ensemble des interfaces fournies
Ir : l'ensemble des interfaces requises
I : Ir ∪ If // l'ensemble des interfaces
Rel : Ir×If // l'ensemble des liaisons entre les interfaces requises et fournis
Rôle = {fourni, requis} // les catégories des interfaces
Etat = {active, passive, frozen} // les états possibles des composants

Pour pouvoir exprimer quelques contraintes sur le modèle, nous avons besoin de définir quelques fonctions d'ordre général sur l'ensemble des composants et des interfaces.

- L'ensemble des interfaces est muni des fonctions suivantes :

RôleI ∈ I → Rôle {1}
CompI ∈ I → C {2}
Connect ∈ Ir → $\mathbb{P}$(If) {3} // Une interface requise peut être liée à plusieurs interfaces fournies et plusieurs interfaces requises peuvent être liées à une même interface fournie. De même, nous autorisons la liaison entre une interface requise et une interface fournie d'un même composant.
Relation ∈ B → $\mathbb{P}$(Rel) {4}

- L'ensemble des composants est muni des fonctions 5 et 6 suivantes :

Cetat ∈ C → Etat {5}
Icomp ∈ C → $\mathbb{P}$(I) {6}

- Les composants sont soumis aux contraintes suivantes :

$\forall c \in C, \forall I (I \in Icomp(c) \Leftrightarrow compI(I) = c)$ {7} // Les interfaces d'un composant ont ce composant comme propriétaire.

$\forall c \in C, (Cetat(c) = active \vee Cetat(c) = frozen) \Rightarrow Icomp(c) \cap Ir \subseteq dom(Connect)$ {8} // Si un composant est dans l'état active ou frozen, les liaisons de ses interfaces requises doivent être effectives. C'est-à-dire, les émissions et réceptions d'appels de méthodes métiers doivent s'exécuter normalement.

//Nous définissons les deux fonctions qui associent à un composant l'ensemble de ses interfaces fournies et requises respectivement, comme suit :

$Ifournis(c) \equiv Icomp(c) \cap If$ {9}

$Irequis(c) \equiv Icomp(c) \cap Ir$ {10}

// Nous définissons aussi, une fonction qui associe à une interface fournie l'ensemble des interfaces requises dont elle est liée comme suit :

$requisI \in If \rightarrow \mathbb{P}(Ir)$ {11}

- Les contraintes de 1 à 11 appartiennent à l'invariant du modèle.

Définition des actions

Nous spécifions maintenant, un ensemble d'actions qui modifient l'état du système, en donnant une précondition et une postcondition. Ces actions concernent l'ensemble des composants, des interfaces et de liaisons entre ces interfaces (connecteurs).

Modification de l'état des composants

Ces actions sont exécutées par le composant gestionnaire de la reconfiguration, avant (états abstraits) et après (états réels) l'exécution de certaines opérations de reconfiguration, pour assurer la cohérence du système après l'adaptation.

ActiveComp(T(c)) =
Pre : Irequis(c) ⊆ dom(Connect) // toutes les liaisons sont faites
Post: Cetat(c) := Active.

PassiveComp(T(c)) =
Pre: true
Post: Cetat(c) := passive

FrozenComp(T(c)) =
Pre: Irequis(c) ⊆ dom(Connect) ∧
∀I(I∈(Ifournis(c)∩codom(connect)) ⇒ (∀I'(I'∈requisI(I) ⇒ Cetat(compI(I'))=passive)
// Tous les composants qui font appel au composant c doivent être dans l'état passive.
Post: Cetat(c) := frozen

- **Modification de l'ensemble des composants**

InsertComp(T(c)) =
Pre : c ∉ C
Post : C := C ∪ {T(c)}
SuppComp(T(c)) =
Pre : c ∈ C ∧ (Irequis(c)∩dom(connect))∪(Ifournis(c)∩codom(connect))=Ø
// La suppression d'un composant nécessite que celui-ci soit non lié.
Post : C := C/{T(c)}

- **Modification de l'ensemble des interfaces**

InsertInterf(T(c), T'(f)) =
Pre: f ∉ I
Post: I := I ∪{T'(f)}∧ Icomp(c):=Icomp(c) ∪ {T'(f)}
SuppInterf(T(c), T'(f)) =
Pre: f ∈ I ∧ f ∉ dom(connect) ∧ f ∉ codom(connect) // l'interface f ne doit pas être liée.
Post: I:=I/{T'(f)} ∧ Icomp(c):=Icomp(c)/{T'(f)}

- **Modification de l'ensemble des connecteurs (liaisons)**

InsertConnect(T(c, c')) =
Pre : c, c' ∈ C ∧ T(c, c') ∉ B
Post : B:= B ∪ {T(c,c')}
SuppConnect(T(c,c')) =
Pre : T(c, c') ∈ B ∧ Cetat(c)= passive. // Connecteur non utilisé.
Post : B:=B/{T(c, c')}

4.3. Etude de cas

Pour illustrer notre approche, nous présentons dans cette section un exemple simple qui a été également utilisé pour détailler les travaux de [MET96] et [LOU05]. Il s'agit du système de contrôle de patients ; PMS (Patient Monitoring System).

Principe de fonctionnement du système PMS

Dans ce système, les infirmières peuvent contrôler leurs patients à distance au sein d'une clinique. Un contrôleur permettant de prendre des mesures, est attaché au lit de chaque patient. Avec ce contrôleur, chaque infirmière peut vérifier l'état de ses patients en demandant à distance les informations concernant la tension, la température, ...etc. De même, lorsque l'état d'un patient devient anormal, c'est à dire quand l'une des données sort de son intervalle, le contrôleur de lit envoie un signal d'alarme à l'infirmière responsable.

Description informelle de l'architecture du système PMS

À chaque service de la clinique (pédiatrie, cardiologie, maternité, etc.) est associé un service d'évènement (EventService) pour gérer les communications entre les infirmières (Nurses) et les contrôleurs de lit (Patients) rattachés à ce service.
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 devient anormal, son contrôleur de lit envoie un signal d'alarme au service d'événement auquel il est lié. Ce service transmet à son tour ce signal à l'infirmière responsable.

4.3.1. Spécification du système

En plus des propriétés architecturales, une application peut avoir des propriétés spécifiques qui doivent être toujours satisfaites durant l'évolution de son architecture.
Nous allons spécifier le système PMS avec prise en compte de quelques propriétés architecturales tel 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.

Par la suite, nous présentons une modélisation de ce système selon le modèle que nous avons proposé. Les contraintes que nous présentons dans les différents exemples sont écrites en utilisant la logique de prédicats du premier ordre et la théorie des ensembles.

4.3.1.1. Le modèle de l'architecture du système

La figure 4.6 ci-après montre la structure architecturale du système PMS selon la notation proposée dans la section 4.2.1.1.

- Le nom du système est PMS.
- Pour la partie « GrapheArchitecture », le système contient trois types de composants (ES:« EventService », P: « Patient » et N: « Nurse »). Et six types de connexions (ES-N, N-ES, ES-P, P-ES, P-N, N-P). Les types P-N et N-P sont utilisés respectivement pour connecter une infirmière à un patient et un ensemble de patients à une infirmière.

- L'ensemble des composants $C = ES \cup N \cup P$.
- L'ensemble des connecteurs, $L = ES\text{-}N \cup N\text{-}ES \cup ES\text{-}P \cup P\text{-}ES \cup P\text{-}N \cup N\text{-}P$.

Le type Patient et le type Nurse communiquent à travers le type EventService, en utilisant l'ensemble des interfaces $I = If \cup Ir$, où :

If = { RecpEtEv, RecpEtP, EnvoiAlarm, Soin}.
Ir = {Décision, demEtP, AlarmEv, AlarmN}.

- Dans la partie « ContraintesArchitecture », nous spécifions les propriétés données précédemment.

1. Le système peut contenir au maximum trois instances de type EventService.
2. Chaque instance de type EventService peut comporter entre zéro et cinq instances de type Nurse et peut contenir entre zéro et quinze instances de type Patient. Pour exprimer cette propriété nous définissons les 2 fonctions :
   nbES-N : ES→ $\mathbb{N}$ // pour déterminer le nombre d'infirmières attachées à un service.
   nbES-P : ES→ $\mathbb{N}$ // pour déterminer le nombre de patients affectés à un service.
3. Une instance de type Patient doit être liée à une seule de type EventService, et une seule instance de type Nurse. Pour exprimer cette propriété nous définissons les 2 fonctions :
   nbP-ES : P→ $\mathbb{N}$ // pour déterminer le nombre de services dont un patient appartient.
   nbP-N : P→ $\mathbb{N}$ // pour déterminer le nombre d'infirmières qui s'occupent d'un patient.
4. Une instance de type Nurse peut s'occuper au plus de trois instances de type Patient et peut être liée au plus à une instance de type EventService. Pour exprimer cette propriété nous définissons les 2 fonctions :
   nbN-P : N→ $\mathbb{N}$ // pour déterminer le nombre de patients dont une infirmière s'occupe.
   nbN-ES : N→ $\mathbb{N}$ // pour déterminer le nombre de services dont une infirmière est liée.

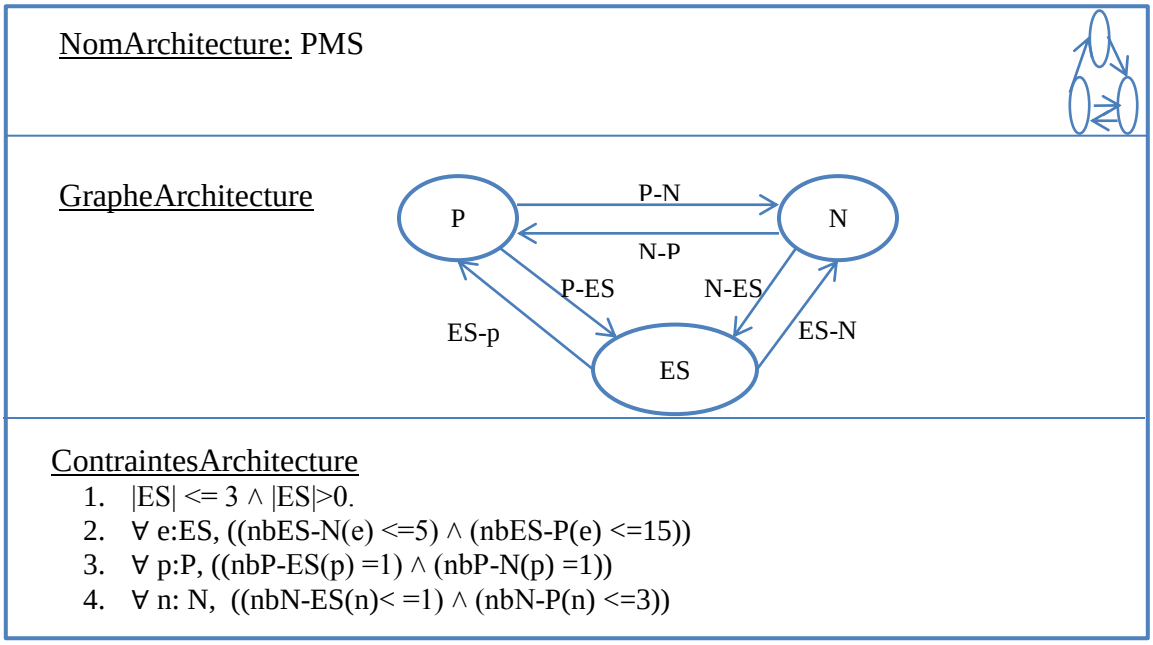


Fig.4.6. Le modèle de la structure architecturale du système PMS.

4.3.1.2. Le modèle des opérations de reconfiguration

Nous spécifions dans cette section, les différentes opérations permettant de faire évoluer le système, tout en prenant compte des propriétés invariantes décrites dans le modèle de la structure architecturale.

Pour la description de la transformation de la configuration du système, nous commençons à partir d'une configuration possible de l'architecture PMS qui est donnée dans la figure 4.7 ci-dessous. Cette configuration contient une instance de composant de type EventService, une instance de composant de type Patient et une instance de composant de type Nurse. Ces instances sont interconnectées par des instances connecteurs.

L'ensemble des relations est donné par : Connect = {(Décision, RecepEtEv), (demEtP, RecepEtP), (AlarmEv, EnvoiAlarm), (AlarmN, Soin)}

Ces relations sont associées aux connecteurs comme suit :

Relation={(N-ES(n1,e1),(Décision,RecepEtEv)),(ES-P(e1,p1),(demEtP,RecepEtP)),(P-ES(p1,e1), (AlarmEv,EnvoiAlarm)) ,(ES-N(e1,n1), (AlarmN, Soin)) }.

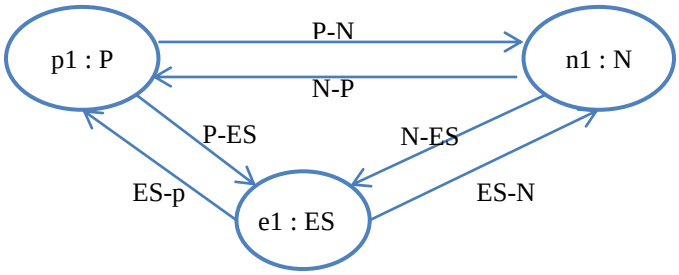


Fig.4.7. Exemple de configuration du système PMS.

➢ **Insertion d'un service d'évènement**

Cette opération permet d'insérer une instance de composant de type ES, à condition que le système ne contient pas déjà trois services d'évènement comme l'impose la première contrainte du système PMS. La représentation de cette opération de reconfiguration avec notre notation est donnée par la figure 4.8 ci-dessous.

- Le nom de l'opération de reconfiguration donné par la partie « NomOpérationReconfiguration » est Insert-EventService(e2).
- Dans la partie « Insertion » nous présentons l'instance e2 de type ES à ajouter dans le système.
- La partie « Suppression » est vide car nous n'avons rien à supprimer avec cette opération.
- Dans la partie « Besoin & Préservation », nous n'avons besoin d'aucune instance de composant pour l'insertion de e2.
- Pour exécuter correctement l'insertion d'un service d'évènement, il faut vérifier la contrainte |ES| < 3 donnée dans la partie « Pré-Conditions », qui signifie que le nombre d'instances de composant de type ES doit être strictement inférieur à 3.
- Dans la partie « Actions » la commande à exécuter est « InsertComp (ES(e2)) » définie comme suit :

InsertComp(ES(e2)) =
Pre : e2∉ ES ∧ |ES| < 3
Post : ES := ES ∪ {ES(e2)}.

- Après l'exécution de l'opération, l'instance e2 de type ES sera dans l'ensemble des composants du système.

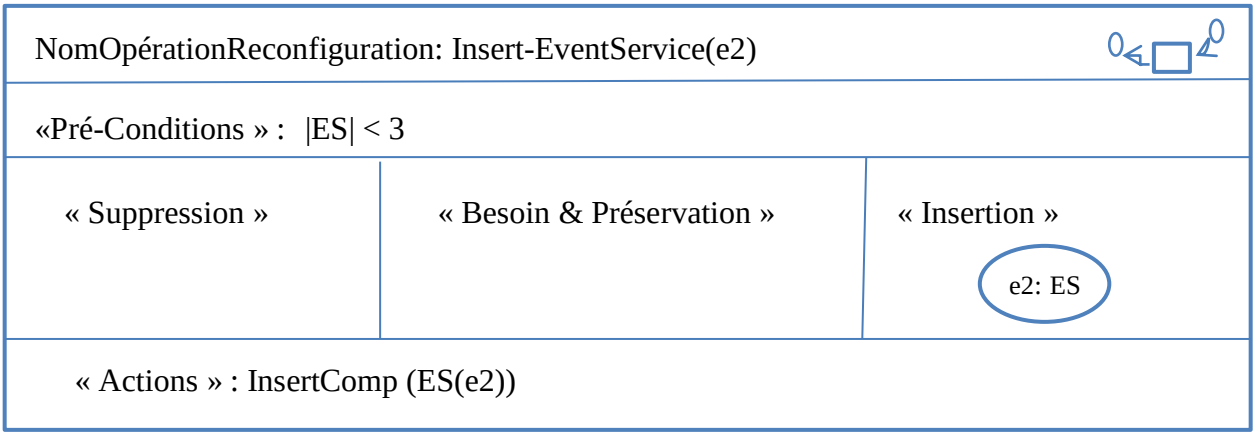


Fig.4.8. Opération d'insertion d'un service d'évènement.

La figure 4.9 ci-après montre la configuration obtenue suite à cette opération.

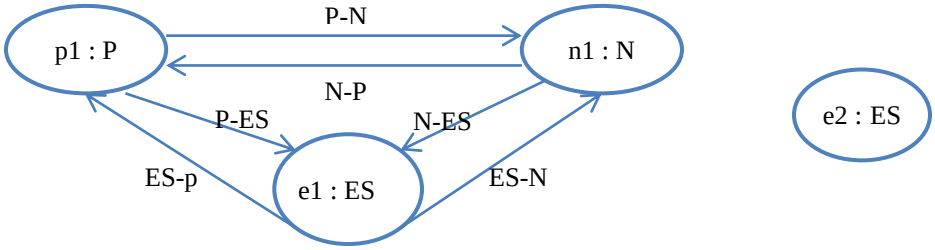


Fig.4.9. Configuration obtenue suite à l'insertion d'un service d'évènement.

➢ **Insertion d'un patient**

Cette opération permet l'insertion d'une instance de composant p2 de type Patient et de l'attacher à une instance de composant e1 de type EventService sous deux conditions :

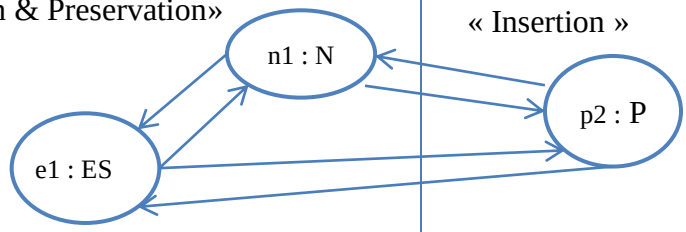
- Il faudrait d'abord qu'il y ait au moins une infirmière n1 appartenant à ce service pour pouvoir s'occuper de ce patient: ∃ n1 : N, ES-N(e1, n1) ∈ ES-N.
- En plus, il faudrait vérifier que ce service ne contient pas déjà quinze patients, et que le nombre de patients pour l'infirmière n1 en question est strictement inférieur à 3: nbES-P(e1) < 15 ∧ nbN-P(n1)< 3, et ceci conformément à la deuxième contrainte du système PMS.

Dans la partie « Insertion », il faut mettre une instance de composant de type Patient. L'insertion d'un patient nécessite l'existence d'une instance de type Nurse connectée à une instance de type EventService. Ces instances doivent être représentées dans la partie « Besoin & Préservation ».

« NomOpérationReconfiguration »: Insert-Patient (p2, e1, n1)

« Pré-Conditions » : ∃ e1:ES, ∃ n1:N, (ES-N(e1,n1) ∈ ES-N ∧ nbES-P(e1)< 15 ∧ nbN-P(n1)< 3)

« Suppression »	«Besoin & Preservation»	« Insertion »



« Actions » : InsertComp (P(p2)), InsertConnect(P-N(p2,n1)) , InsertConnect(N-P(n1,p2)), InsertConnect (ES-P(e1,p2)), InsertConnect (P-ES(p2,e1))

Fig.4.10. Opération d'insertion d'un patient.

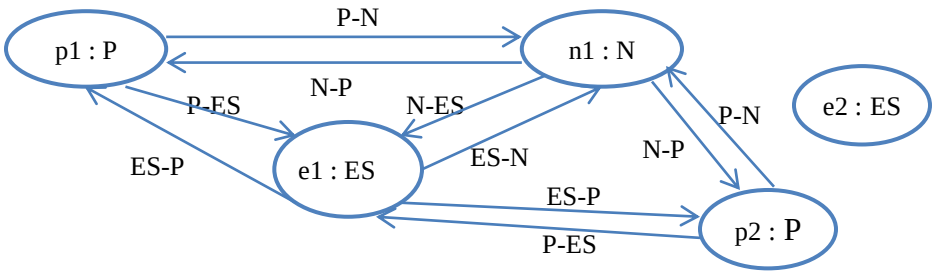


Fig.4.11. Configuration obtenue suite à l'opération d'insertion d'un patient.

➢ **Insertion d'une infirmière**

Cette opération de reconfiguration permet d'insérer une instance de composant n2 de type Nurse et de la lier à une instance de composant e2 de type EventService (devant exister dans le système). L'application de cette opération exige que l'instance de composant EventService en question existe et ne contienne pas déjà cinq instances de composants de type Nurse :
nbES-N(e2)< 5 , conformément à la deuxième contrainte architecturale du système.

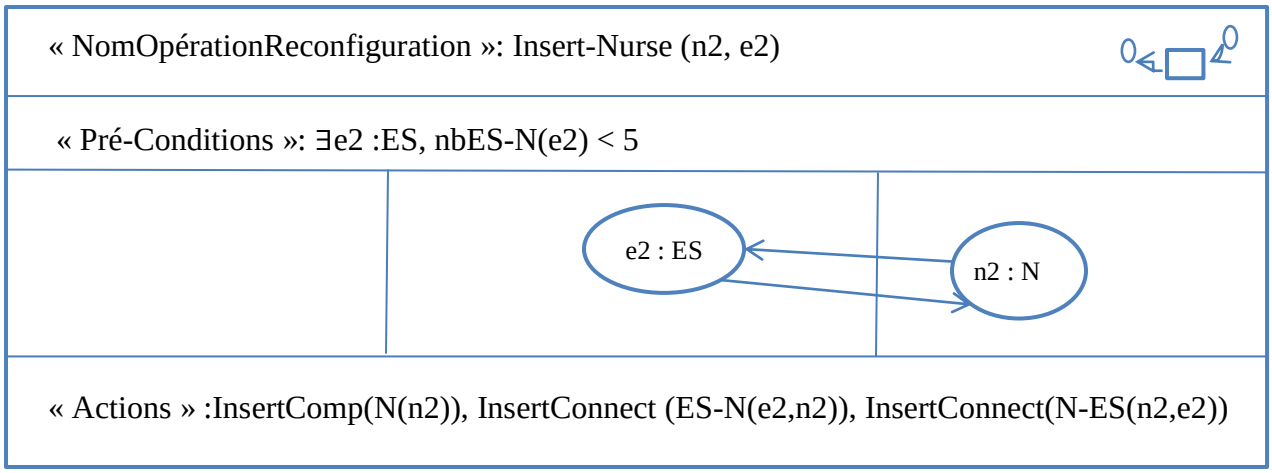


Fig.4.12. Opération d'insertion d'une infirmière.

La figure 4.13 montre une configuration possible obtenue suite à cette opération.

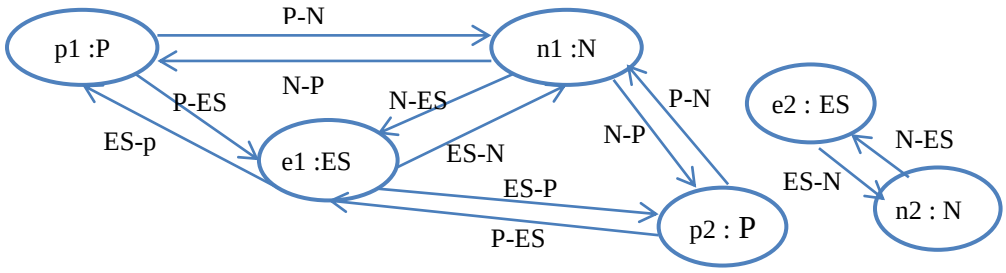


Fig.4.13. Configuration obtenue suite à l'insertion d'une infirmière.

➢ **Suppression d'un patient**

Lorsqu'un patient quitte le système à la décision de l'infirmière responsable, son contrôleur de lit est déconnecté du service en question et supprimé du système.

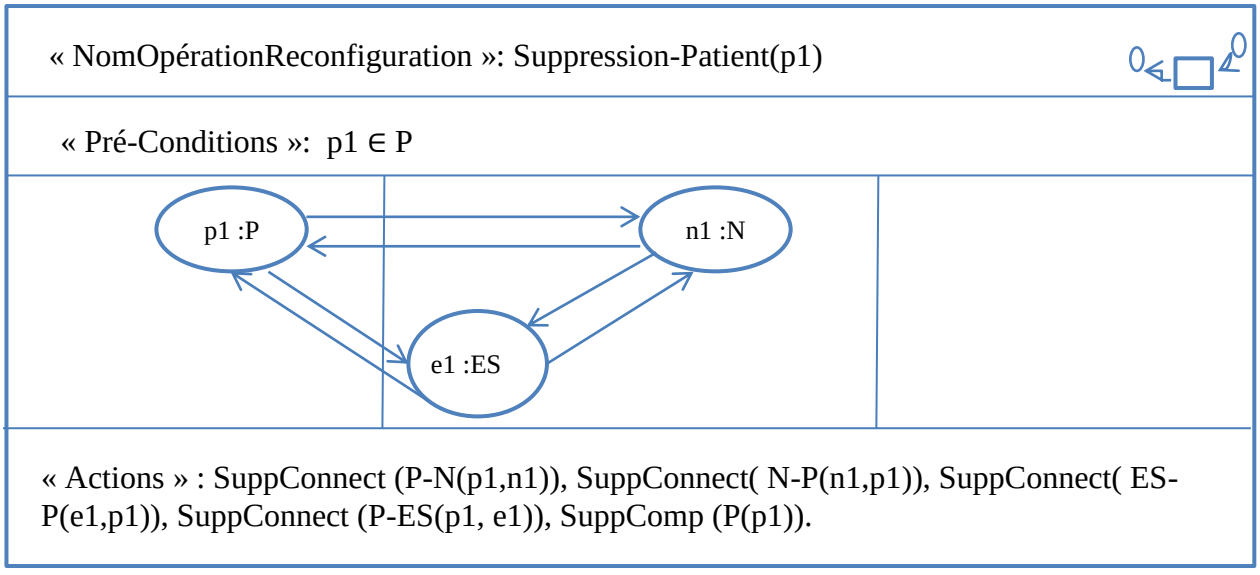


Fig.4.14. Opération de suppression d'un patient.

La figure 4.15 montre la configuration obtenue suite à cette opération.

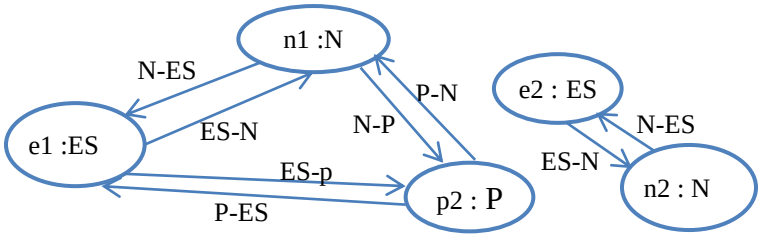


Fig.4.15. Configuration obtenue suite à la suppression d'un patient.

➢ **Déconnexion d'une infirmière**

Une infirmière peut quitter son service (sans être supprimée du système) seulement si ce service ne contient pas de patients, ou s'il contient d'autres infirmières qui pourraient s'occuper des patients. Ainsi, cette opération ne permet de déconnecter un composant de type Nurse que si l'une des deux conditions citées est satisfaite.

La représentation de cette opération selon notre modèle est donnée par la figure 4.16.

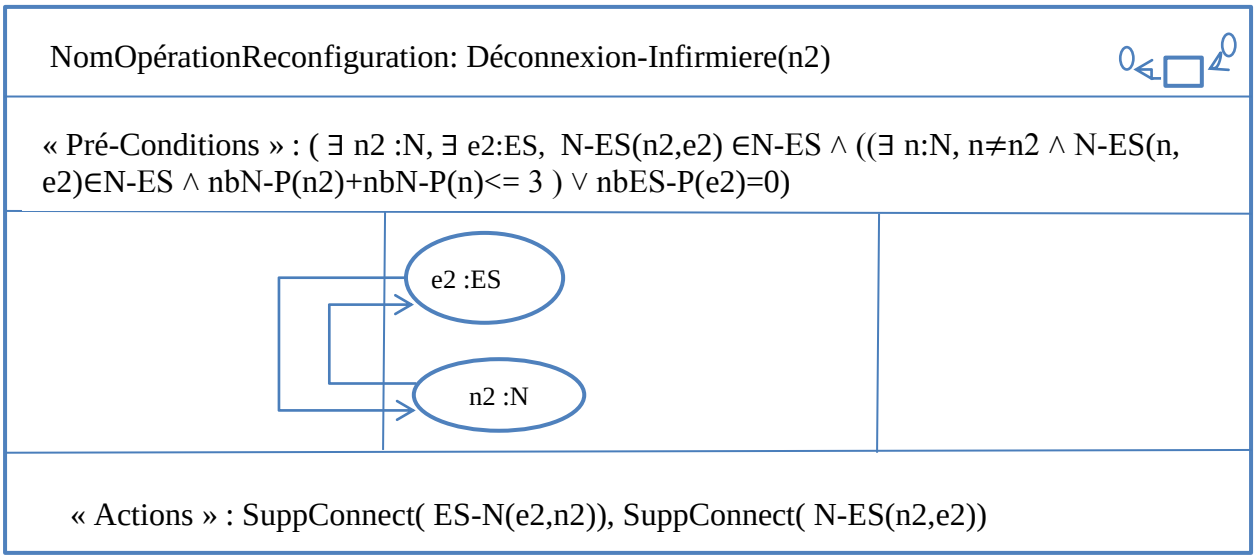


Fig.4.16. Opération de déconnexion d'une infirmière.

Remarque : Dans ce cas précis, nous pouvons voir une limite d'une notation purement graphique ; ce qui est exprimé simplement par l'opérateur de disjonction dans la pré-condition ne peut pas être exprimé graphiquement dans la partie « Besoins & Préservation».

➢ **Connexion d'une infirmière**

Pour cette opération, il faudrait d'abord vérifier que l'infirmière est libre, c'est-à-dire qu'elle n'est pas attachée à aucun service. En plus, elle ne pourrait être connectée à un service, que si ce dernier ne contient pas déjà cinq infirmières conformément à la deuxième contrainte du système. Dans ce cas, il s'agit de la même remarque faite pour l'opération de déconnexion.

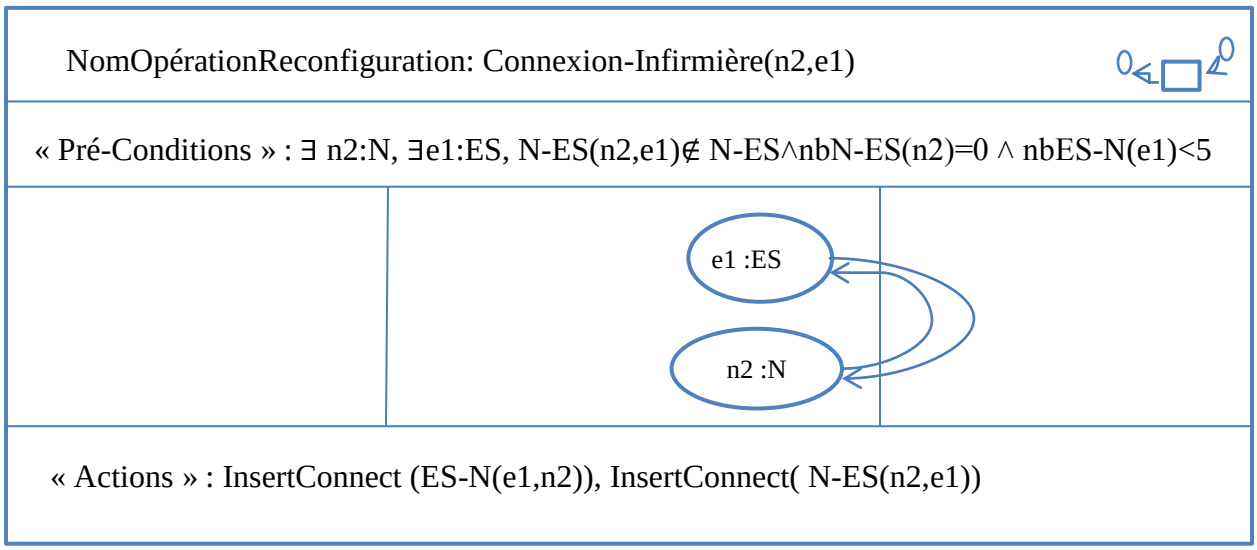


Fig.4.17. Opération de connexion d'une infirmière.

Définition des actions

InsertComp(P(p2)) =
Pre: p2∉P
Post: P: = P ∪ {P(p2)}.
InsertConnect(P-N(p2,n1)) =
Pre: P-N(p2,n1) ∉ P-N
Post: nbP-N(p2):=1 ∧ P-N:=P-N∪{P-N(p2,n1)}.
InsertConnect(N-P(n1,p2)) =
Pre: N-P(n1,p2)∉ N-P ∧ nbN-P(n1)< 3
Post: nbN-P(n1):=nbN-P(n1)+1 ∧ N-P:=N-P ∪ {N-P(n1,p2)}.
InsertConnect (ES-P(e1,p2)) =
Pre: ES-P(e1,p2) ∉ ES-P ∧ nbES-P(e1)< 15
Post: nbES-P(e1):= nbES-P(e1)+1 ∧ ES-P:=ES-P ∪{ES-P(e1,p2)}.
InsertConnect (P-ES(p2,e1)) =
Pre: P-ES(p2,e1) ∉ P-ES
Post: nbP-ES(p2):=1 ∧ P-ES:=P-ES ∪ {P-ES(p2,e1)}.
InsertComp(N(n2)) =
Pre: n2∉N
Post: N= N ∪{N(n2)}.
InsertConnect(ES-N(e2,n2)) =
Pre: ES-N(e2,n2) ∉ ES-N ∧ nbES-N(e2) < 5
Post: nbES-N(e2):=nbES-N(e2)+1 ∧ ES-N:=ES-N∪{ES-N(e2,n2)}.
InsertConnect(N-ES(n2,e2)) =
Pre: N-ES(n2,e2) ∉ N-ES
Post: nbN-ES(n2):= 1 ∧ N-ES:=N-ES∪{N-ES(n2,e2)}.

SuppConnect (P-N(p1,n1)) =
Pre: P-N(p1,n1) ∈ P-N ∧ Cetat(p1)= passive
Post: nbP-N(p1):=0 ∧ P-N:=P-N\{P-N(p1,n1)}.
SuppConnect(N-P(n1,p1)) =
Pre: N-P(p1,n1) ∈ N-P ∧ Cetat(n1)= passive
Post: nbN-P(n1):= nbN-P(n1)-1 ∧ N-P:=N-P\{N-P(n1,p1)}.
SuppConnect(ES-P(e1,p1)) =
Pre: ES-P (e1,p1) ∈ ES-P ∧ Cetat(e1)= passive
Post: nbES-P(e1):= nbES-P(e1)-1 ∧ ES-P:=ES-P\{ES-P(e1,p1)}.
SuppConnect (P-ES(p1, e1)) =
Pre: P-ES(p1,e1) ∈ P-ES ∧ Cetat(p1)= passive
Post: nbP-ES(p1):=0 ∧ P-ES:=P-ES\{P-ES(p1,e1)}.
SuppComp (P(p1))=
Pre: p1∈ P
Post: P:=P\{P(p1)}.
SuppConnect (ES-N(e1,n1)) =
Pre: ES-N(e1,n1)∈ES-N ∧ Cetat(e1)= passive
Post: nbES-N(e1):=nbES-N(e1)-1 ∧ ES-N:=ES-N\{ES-N(e1,n1)}.
SuppConnect(N-ES(n1,e1)) =
Pre: N-ES(n1,e1) ∈ N-ES ∧ Cetat(n1)= passive
Post: nbN-ES(n1):= 0 ∧ nbN-P(n1) :=0 ∧ N-ES:=N-ES\{N-ES(n1,e1)}.

4.3.1.3. Le modèle du plan d'exécution

Nous décrivons, dans cette section, un exemple simple d'un plan d'exécution.

- **Transfert d'une infirmière**

Il s'agit dans ce cas, du transfert de l'infirmière d'un service vers un autre. Cette opération de reconfiguration consiste à transférer une instance de composant de type Nurse vers une autre instance de composant de type EventService seulement si, elle n'a pas une instance de type Patient en occupation, ou si le service dont elle dépend contient d'autres infirmières qui pourraient s'occuper des patients et que la nouvelle instance de EventService contient moins de cinq instances de type Nurse. Cette opération est composée donc des deux opérations citées précédemment, une opération de déconnexion suivie par une opération de connexion.

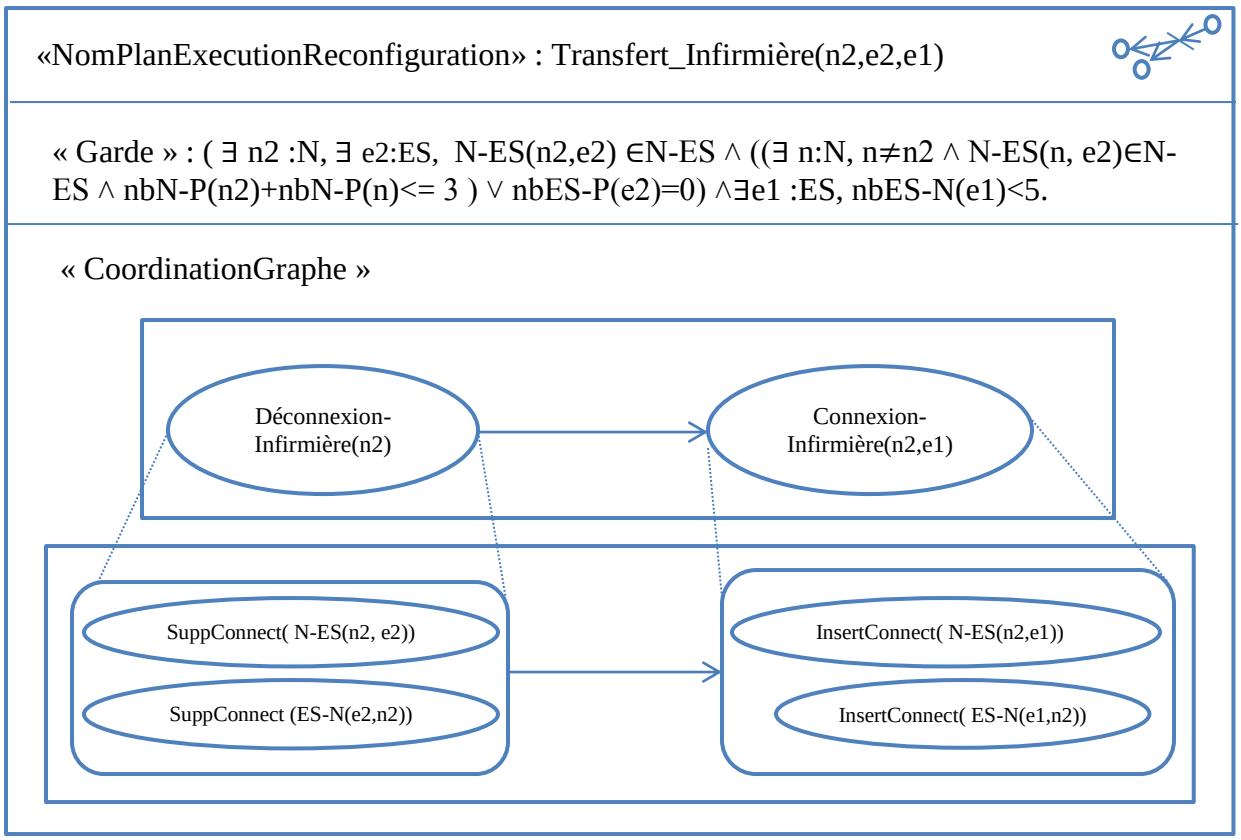


Fig.4.18. Modèle du plan d'exécution de l'opération de transfert d'une infirmière.

Le plan d'exécution de l'opération de transfert d'une infirmière peut être exécuté si les contraintes citées dans la partie « Garde » sont vérifiées.

Selon le graphe de coordination, la première opération de reconfiguration à exécuter est « Déconnexion-Infirmière ». L'opération de reconfiguration « Connexion-Infirmière » ne peut être exécutée que si l'opération de reconfiguration « Déconnexion-Infirmière» a été déjà exécutée.

4.4. Conclusion

Dans ce chapitre, nous avons proposé une approche de modélisation des systèmes adaptatifs basée sur la théorie des graphes, la théorie des ensembles et la logique de prédicats. Notre modélisation traite essentiellement l'évolution des exigences du système modélisé. Pour adapter le système à ces exigences, Nous avons associé des opérations de reconfiguration à ces situations, tout en prenant en compte les propriétés architecturales et fonctionnelles.

Le modèle que nous avons proposé, est constitué de trois sous modèles. Le premier décrit la structure de l'architecture, en termes de type de composants et de connexions. Le deuxième étend le premier modèle et décrit la dynamique structurelle de l'architecture en termes d'opérations de

reconfiguration. Quant au troisième modèle, il permet de décrire l'enchainement et l'ordre d'exécution de ces opérations de reconfiguration, suite à un évènement donné.

Notre approche est détaillée dans ce mémoire, à travers l'étude de cas du système logiciel de contrôle de patients PMS (Patient Monitoring System).

Afin de vérifier si l'exécution d'une opération de reconfiguration préserve les contraintes déjà définies dans le modèle de la structure architecturale, et si le système est consistant ou non après la reconfiguration, nous proposons dans le chapitre suivant une approche de vérification qui fait recours aux techniques formelles pour analyser et vérifier le modèle. Nous adoptons une approche basée sur la transformation des modèles semi-formels (basés sur les graphes) vers des spécifications formelles validées.

CHAPITRE 5

Vérification et validation du modèle

5.1. Introduction

Durant l'étape de modélisation, le concepteur peut tomber facilement dans l'erreur (par exemple, la définition d'une pré-condition d'une opération qui ne préserve pas l'invariant du système). Afin de faciliter l'identification des incohérences (contradiction entre les prédicats) et détecter toutes les erreurs faites lors de la modélisation, nous proposons une approche permettant dans un premier temps, d'assurer une transformation de notre modèle proposé dans le chapitre précédant (modèle de la structure architecturale, de chaque opération de reconfiguration et de chaque règle de reconfiguration) vers le langage formel Event-B [EVB05]. Par la suite, elle permet de vérifier la consistance du système initialement, et la conformité de l'évolution de son architecture par rapport a ses contraintes invariantes grâce à l'atelier B [MRB05].

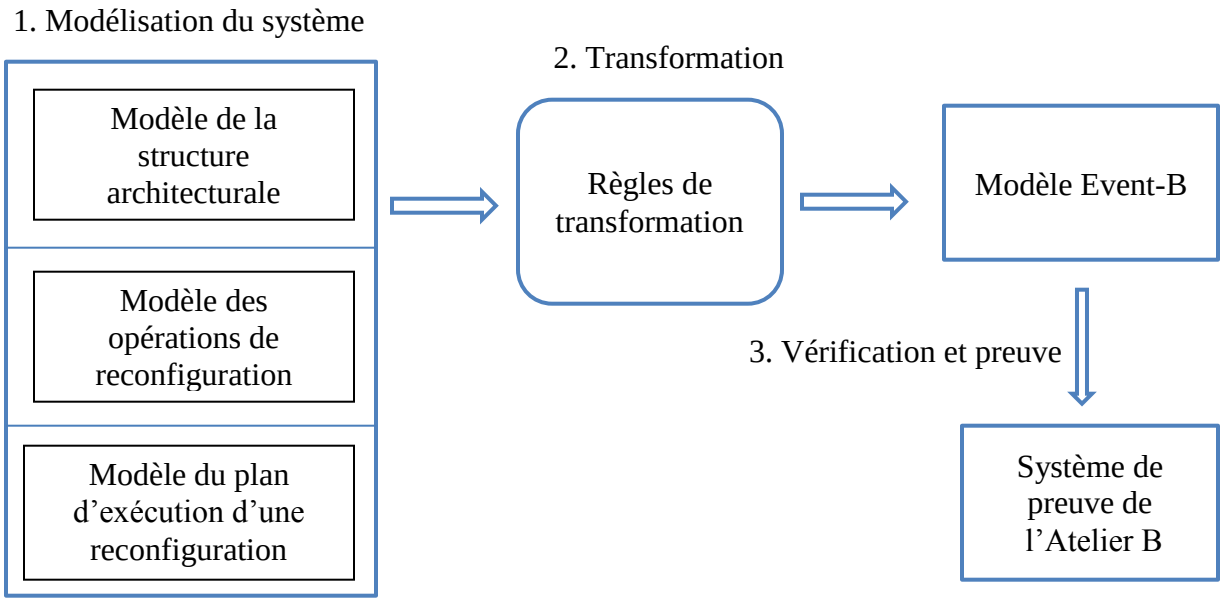


Fig.5.1. Approche de vérification et de validation.

5.2. Justification du choix de la méthode B et plus précisément le langage Event-B

Dans le chapitre précédant, nous avons introduit un modèle (modèle de la structure architecturale, modèle des opérations de reconfigurations et le modèle des plans d'exécution des reconfigurations) pour la spécification des types de données, des actions et des assertions (invariant, pré\post conditions). Ce modèle est basé sur une modélisation architecturale à base de composants, où, les composants sont abstraits de façon à pouvoir y'exprimer les propriétés les plus importantes. De plus, notre modèle supporte la transformation de l'état de l'architecture d'une configuration à une autre. Ainsi, nous avons choisi d'utiliser la méthode B pour mettre en œuvre la démarche de preuve de correction, et pour l'assistance à la preuve de cohérence, car elle

modélise un système par des transformations d'états et adopte les même concepts de formalisation de notre modèle (assertions « invariant, pré\post conditions »). La vérification de la cohérence des modèles construits se fait par rapport à l'invariant.

La méthode B classique ne définit qu'un ensemble d'opérations, mais pas les événements qui permettent de les déclencher et leur ordre. Puisque nous nous s'intéressons à la spécification des systèmes adaptatifs (réactifs), basés sur le modèle E-C-A, nous préférons la méthode B évènementielle qui étend la méthode B classique, tout en permettant la spécification abstraite des évènements provocants la reconfiguration.

L'atelier B [MRB05] supportant le langage de spécification B, permet la vérification automatique de la cohérence par la génération automatique de toutes les obligations de preuve, selon la sémantique de préservation des propriétés invariantes. Elle est dotée à la fois d'un prouveur automatique et interactif.

Ce cadre de preuve est adéquat à notre modélisation de la reconfiguration dynamique. Nous transposons donc la question de vérification de la cohérence de notre modèle en une question de vérification de la cohérence de modèle Event-B.

5.3. Extraction de machines B à partir de la modélisation semi-formelle

Pour pouvoir faire des raisonnements rigoureux sur l'évolution de l'architecture d'un système dynamique et de son fonctionnement, nous proposons des règles permettant de transformer le modèle de la structure architecturale (contraintes architecturales et fonctionnelles), le modèle de chaque opération de reconfiguration et le modèle du plan d'exécution d'une reconfiguration vers une spécification formelle Event-B (machines abstraites B), tout en transmettant les notations graphiques et les contraintes associées. Cette spécification sera utilisée afin de vérifier la conformité du système par rapport à son style architecturale après l'exécution d'une opération de reconfiguration. Ceci, en faisant des raisonnements et des preuves formelles.

- Nous allons illustrer ces règles de transformation, par la suite, à travers l'étude de cas PMS.

5.3.1. Transformation du modèle de la structure architecturale

Le modèle de la structure architecturale est transformé vers une machine abstraite B, en utilisant un ensemble de règles de R1 à R5. Cette transformation préserve les types de composants, de connecteurs et les contraintes invariantes.

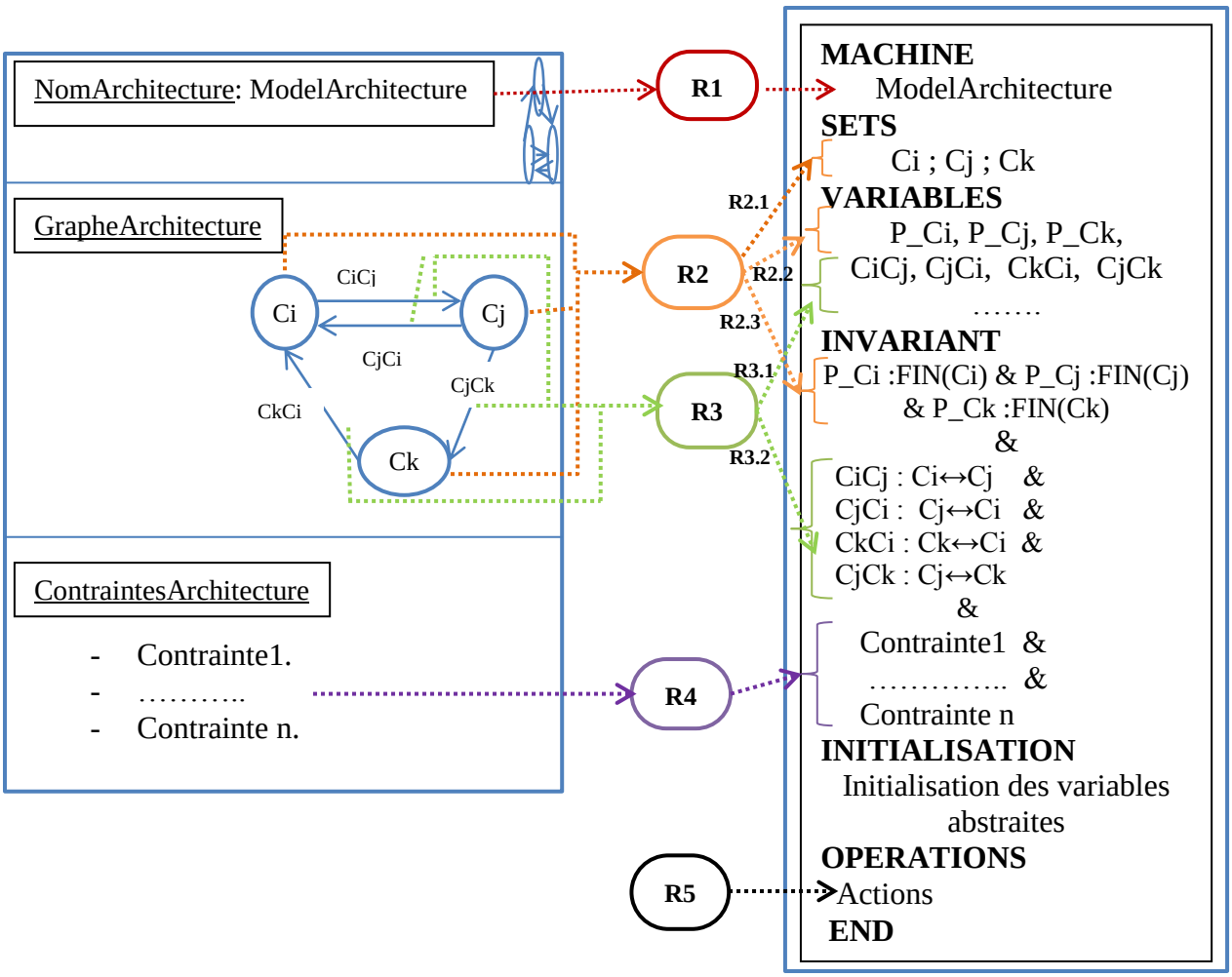


Fig.5.2. Transformation du modèle de la structure architecturale vers une machine B.

➢ **Règle 1 : Définition du nom de la machine abstraite**

Le nom de la machine abstraite est défini à partir du nom du système, donné par la partie « NomArchitecture » de notre modélisation.

➢ **Règle 2 : Définition des types de composants**

R2.1. Chaque type de composant, existant dans la partie « GrapheArchitecture », est transformé vers un ensemble de composants contenant tous les composants de ce type. Le nom de chaque ensemble correspond au type de ses composants. Ces ensembles sont déclarés dans la clause SETS de la machine B.

R2.2. Chaque ensemble de composants du même type, appartenant au système, est traduit par un ensemble fini (FIN) de composants. Le nom de chaque ensemble de composants commence par le préfixe « P_ » suivi par le type de ses composants. Ces ensembles sont déclarés dans la clause VARIABLES de la machine B.

R2.3. Le type de chaque ensemble de composants généré par la règle2.2 correspond à un type basic défini avec la règle2.1. Ces types sont déclarés dans la clause INVARIANT de la machine B.

- **Règle 3 : Définition des Types de connexions**

R3.1. Chaque type de connexion, entre deux types de composants Ci, Cj (ensembles définis dans la clause SETS), présenté dans la partie «GrapheArchitecture» est convertie vers une relation CiCj, définie comme étant une variable abstraite dans la clause VARIABLES de la machine B. Le nom d'une relation est composé du type des composants clients suivi par le type des serveurs.

R3.2. Le type de chaque relation générée par la règle 3.1 (CiCj : Ci ↔ Cj) est déclaré dans la clause INVARIANT de la machine B. La source de chaque relation Ci, porte le nom du type des composants clients, et la cible de la relation Cj, porte le nom du type des composants serveurs.

- **Règle 4 : Définition des contraintes (propriétés invariantes du système)**

Nous proposons de transformer les contraintes du système, exprimées dans la partie « ContraintesArchitecture », vers des prédicats B déclarés dans la clause INVARIANT de la machine B. Ainsi, une description précise de la syntaxe de génération des prédicats B est nécessaire, afin d'accomplir la transformation des contraintes de notre modèle (voir la section 2 de l'annexe B).

- Voici un exemple de transformation d'une contrainte du système PMS, vers un prédicat B

$$|ES| \leq 3 \wedge |ES| > 0$$

Cette expression exprime que le système peut contenir de zéro à trois instances de composants de type EventService. Sa transformation vers le langage B génère le prédicat suivant: card(P_ES) <= 3 ∧ card(P_ES) > 0, où, card (P_ES) désigne le cardinal de l'ensemble P_ES.

- **Règle 5 : Définition des différentes actions élémentaires à exécuter par le système**

Nous proposons également de définir les différents types d'actions utilisées pour l'exécution des opérations de reconfigurations, sous forme d'opérations ayant une pré/post-condition. Ceci, en utilisant la substitution pré-condition (**PRE ….. THEN**) dans la clause OPERATIONS de la machine B, comme suit :

NomAction(par_1, ... , par_n) = **PRE** par_1 : type & ... & par_n : type & « pré-conditions »
THEN « post-conditions »
END

Avec :

- Par_1, …, Par_n représentent les paramètres d'entrée de l'action du nom NomAction.
- Le type des paramètres d'entrée doit être mentionné après le mot clé PRE, avant de définir les prés-conditions.
- Les « pré-conditions » expriment des conditions qui doivent être évaluées en vrais pour que la production des « post-conditions » (contraintes prévues après l'exécution de l'action) auras lieu. Ces conditions définissent des contraintes sur les valeurs des attributs (par exemple l'état d'un composant, le nombre d'instances d'un composant, …etc).

Voici un exemple d'actions (l'action d'insertion d'un composant de type Ci)

InsertCompCi(ci) = PRE ci:Ci & ci ∉ P_Ci THEN P_Ci := P_Ci \/ {ci} END

Remarques

- Puisque les « pré-conditions » diffèrent selon les types des paramètres impliqués, la même action doit être définie autant de fois que le nombre de combinaisons des types des paramètres.

- Pour exprimer certaines propriétés sur les instances de composants et de connecteurs, nous utilisons des fonctions déclarées comme étant des variables abstraites dans la clause VARIABLES. Ces dernières doivent être typées et contraintes dans la clause INVARIANT.

- Toutes les variables abstraites doivent être initialisées dans la clause INITIALISATION.

5.3.2. Transformation du modèle d'une opération de reconfiguration

Nous proposons dans cette section, comme le montre la figure 5.3 ci-dessous, un ensemble de règles de R6 jusqu'à R11, permettant de convertir une opération de reconfiguration simple (constituée seulement des actions élémentaires définies dans la machine ModelArchitecture) de notre modèle vers une opération B, dans une machine portant le nom Opérations.

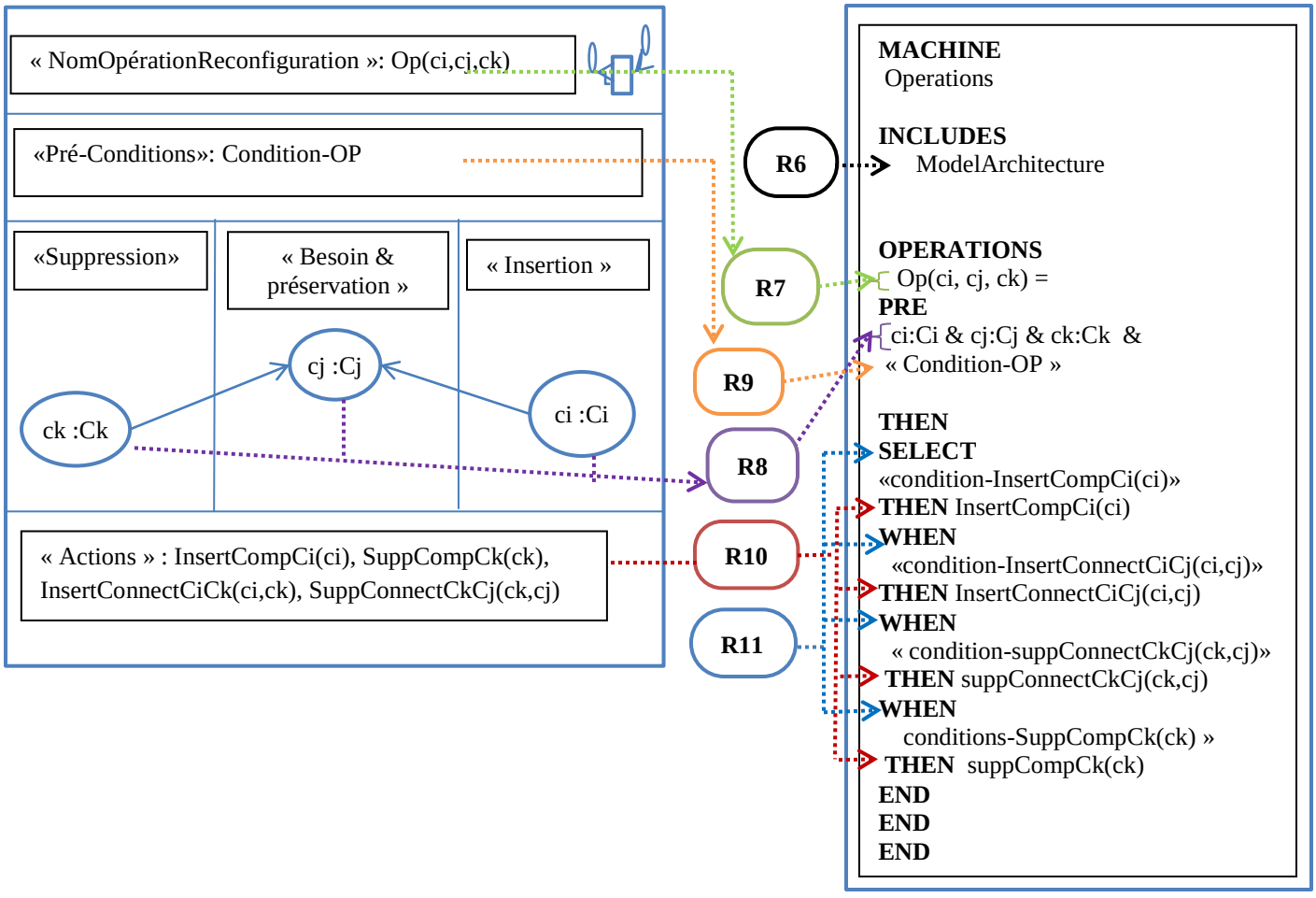


Fig.5.3. Transformation du modèle d'une opération de reconfiguration vers une opération B.

- **Règle 6 : Déclaration de l'instance de machine incluse ModelArchitecture**

Cette règle permet de regrouper dans la clause INCLUDES de la machine abstraite Operations, les constituants (ensembles, et variables) de la machine abstraite ModelArchitecture ainsi que ses propriétés (clause INVARIANT) et opérations (actions utilisées par les opérations).

- **Règle 7 : Définition du nom de l'opération avec ses paramètres d'entrée**

Le nom de l'opération et ses paramètre d'entrée sont déterminés à partir de la partie « NomOpérationReconfiguration » du modèle de l'opération. Il est défini dans la clause OPERATIONS de la machine Opérations.

- **Règle 8 : Typage des paramètres d'entrée de l'opération**

Les paramètres d'entrée de l'opération sont typés à partir des instances de composants présentées dans les parties « Suppression », « Besoins & Préservation » et « Insertion ».
Chaque instance de composant est transformée, dans la partie PRE de l'opération B, vers le nom de l'instance de composant suivi par son type. Le type de chaque instance est le type abstrait associé au composant qui lui correspond.

- **Règle 9 : Définition des contraintes d'exécution propres à l'opération**

En utilisant la substitution pré-condition, les conditions d'exécution de l'opération présentées dans la partie « Pré-Condition » du modèle de l'opération de reconfiguration sont transformées vers des prédicats B déclarés dans la partie PRE après le typage des paramètres d'entrée. Cette règle se réalise conformément à la section 2 de l'annexe B.

- **Règle 10 : Appel des actions à exécuter par l'opération**

L'appel des actions à exécuter par l'opération se fait en utilisant la substitution sélection comme suit :
SELECT prédicat THEN **action** WHEN $prédicat_1$ THEN $\mathbf{action_1}$….WHEN $prédicat_n$ THEN $\mathbf{action_n}$.

- Chaque action est appelée après le mot clé THEN de la substitution SELECT.

- **Règle 11 : Transformation des dépendances du niveau local de la partie « CoordinationGraphe » du modèle du plan d'exécution vers des prédicats B**

Chaque dépendance entre deux actions d'une même opération (graphe local) est transformée vers un prédicat exprimant que l'action dont l'autre dépend a été déjà exécutée (voir figure 5.4).
Ce prédicat est défini après le mot clé SELECT ou WHEN de la substitution sélection. Si aucune dépendance n'existe pour une action, son prédicat redéfini le type de ses paramètres.

5.3.3. Transformation du modèle du plan d'exécution d'une reconfiguration

Nous proposons dans cette section des règles de R12 à R16, permettant une transformation du modèle du plan d'exécution vers une machine B, spécifiant l'ordre d'exécution entre les différentes opérations d'une reconfiguration. Il s'agit dans ce cas, de transformer la partie « CoordinationGraphe » du modèle du plan d'exécution vers le corps d'un évènement B.

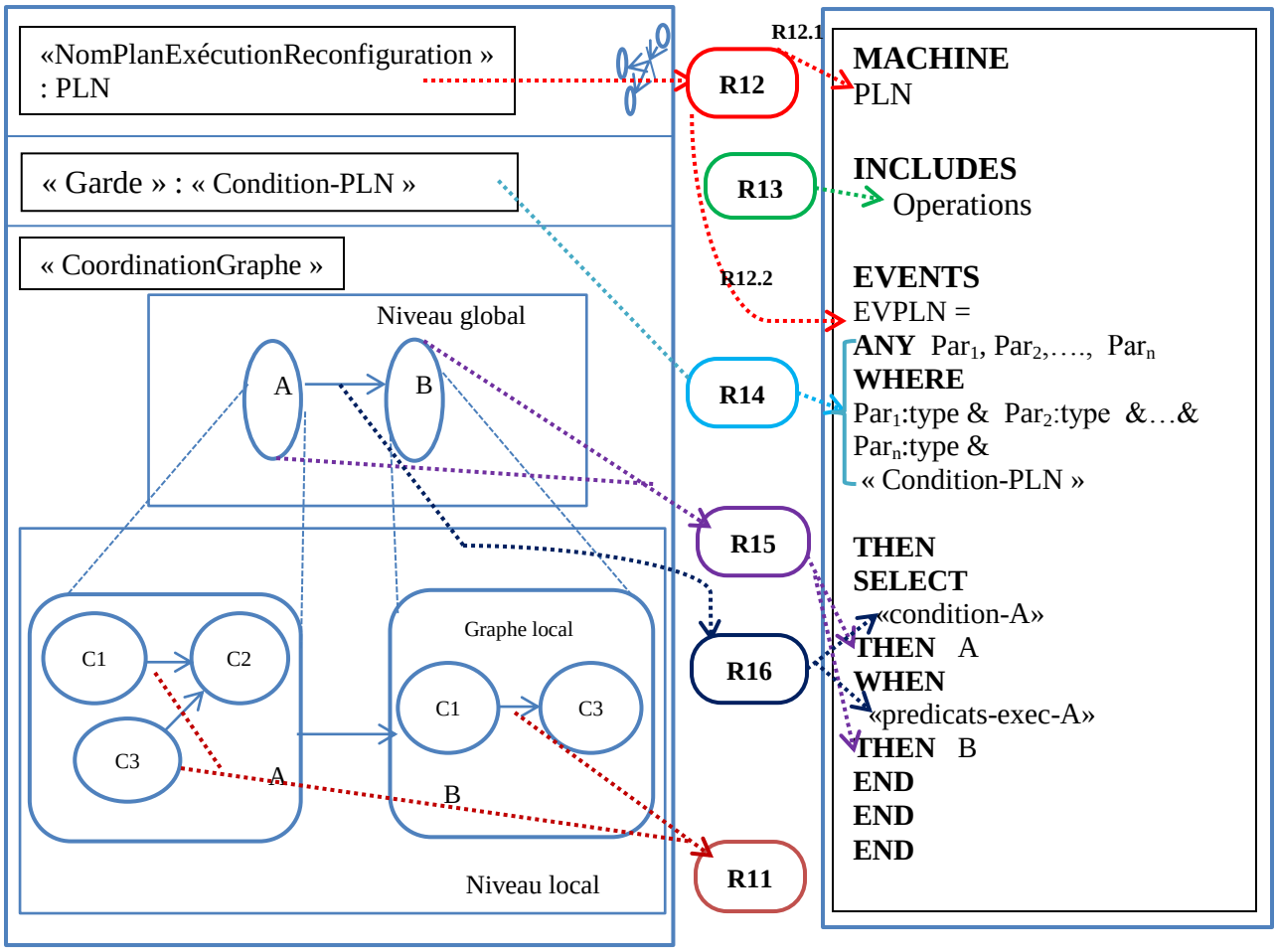


Fig.5.4.Transformation du modèle du plan d'exécution d'une reconfiguration vers une machine B

➢ **Régle12**

- **R 12.1. Définition du nom de la machine abstraite**

Le nom de la machine abstraite spécifiant le plan d'exécution est défini à partir du nom du plan donné par la partie «NomPlanExécutionReconfiguration».

- **R 12.2. Définition du nom de l'évènement**

Le nom de l'évènement est déterminé à partir de la partie «NomPlanExécutionReconfiguration » du modèle du plan d'exécution, précédé du préfixe EV. Il est défini dans la clause EVENTS de la machine PLN.

- **Règle 13 : Déclaration de l'instance de machine incluse Operations**

Cette règle permet de regrouper dans la clause INCLUDES de la machine abstraite PLN, les opérations spécifiées dans la machine Operations, ainsi que les constituants (ensembles, et variables) de la machine abstraite ModelArchitecture, ses propriétés (clause INVARIANT) et opérations (actions utilisées par les opérations).

N.B : Si un évènement dépend d'un autre, il suffit d'ajouter le nom de la machine spécifiant l'évènement dont il dépend dans la clause INCLUDES de sa machine. Ainsi, le lien INCLUDES permet de construire d'une manière modulaire des machines abstraites spécifiant les différents évènements.

- **Règle 14 : Définition des contraintes d'exécution de l'évènement & typage des variables associées à l'évènement**

En utilisant la substitution ANY, Les paramètres associés à l'évènement défini par la règle 12.2 sont déterminés à partir des paramètres des opérations impliquées par le plan de cet évènement. Ces paramètres doivent être typés dans la partie WHERE de la substitution ANY. Les conditions d'exécution du plan « Conditions_PLN » de la partie « Garde » (garde de l'évènement) sont transformées vers des prédicats B déclarés dans la partie ANY après le typage des paramètres d'entrée.

- **Règle 15 : Appel des opérations à exécuter suite à l'évènement**

L'appel des opérations à exécuter par l'évènement se fait en utilisant la substitution sélection comme suit :

SELECT prédicat THEN **opération** WHEN $prédicat_1$ THEN **$opération_1$**….WHEN $prédicat_n$ THEN **$opération_n$**.

- Chaque opération est appelée après le mot clé THEN de la substitution sélection.

- **Règle 16 : Transformation des dépendances du niveau global de la partie « CoordinationGraphe » du modèle du plan d'exécution vers des prédicats B**

Chaque dépendance entre deux opérations du plan d'exécution est transformée vers un prédicat, exprimant que l'opération dont l'autre dépend a été déjà exécutée.

Ce prédicat est défini après le mot clé WHEN de la substitution sélection. Si une opération ne dépend d'aucune autre (aucun arc entrant à l'opération dans le graphe global) son prédicat redéfini le type de ses paramètres.

Remarque

Noter que chaque plan d'exécution d'un évènement est considéré comme une opération complexe (constituée d'opérations plus simples définies dans la clause OPERATIONS de la machine Operations). Cette opération complexe se transforme vers une machine B incluant la machine Operations dans la clause INCLUDES. La transformation se fait de la même façon qu'une opération simple toute en remplaçant les actions impliquées par une opération simple par les opérations impliquées par l'opération complexe.

5.4. Application des règles de transformation sur l'exemple PMS

Afin d'illustrer nos règles de transformation, Nous rejoignons notre exemple PMS et nous traduisons le modèle généré dans le chapitre 4 vers le langage B.
L'application des règles de transformation de R1 à R5 et l'utilisation de la grammaire de génération des prédicats B sur le modèle de la structure architecturale donnent la machine B suivante.

MACHINE PMS {R1}
SETS
patient; eventservice; nurse {R2.1}
VARIABLES
P_patient, P_eventservice, P_nurse, {R2 .2}
P_ES, P_N, N_ES, N_P, ES_N, ES_P {R3.1}
,nbES_N, nbES_P, nbP_ES, nbP_N, nbN_ES, nbN_P

INVARIANT
P_patient:FIN(patient) & P_eventservice:FIN1(eventservice) & P_nurse:FIN(nurse) {R2.3}
& nbES_N: eventservice-->NATURAL & nbES_P: eventservice-->NATURAL &
nbP_ES: patient-->NATURAL & nbP_N: patient-->NATURAL &
nbN_ES: nurse-->NATURAL & nbN_P: nurse-->NATURAL &
P_ES: patient <-> eventservice & P_N: patient<->nurse &
N_ES: nurse<->eventservice & N_P: nurse<->patient & {R3.2}
ES_N:eventservice<->nurse & ES_P:eventservice<->patient &
card(P_eventservice)<=3 & card(P_eventservice)>0 &
!ev.(ev:P_eventservice => nbES_N(ev)<=5 & nbES_P(ev)<=15) & {R4}
!pt.(pt:P_patient => nbP_N(pt)=1 & nbP_ES(pt)=1) &
!nrs.(nrs:P_nurse => nbN_P(nrs)<=3 & nbN_ES(nrs)<=1)

INITIALISATION
……..
OPERATIONS
InsertcompES(es)=
PRE es:eventservice & card(P_eventservice)< 3 THEN
NbES_P(es):=0 || nbES_N(es):=0 || P_eventservice:=P_eventservice \ /{es}
END; {R5}
InsertcompP(ps)=
PRE ps:patient THEN
NbP_ES(ps):=1 || nbP_N(ps):=1 || P_patient:=P_patient\ /{ps}
END;
……
END

Fig.5.5. Transformation du modèle de la structure architecturale du système PMS vers une machine B.

- Nous avons donné dans la figure 5.5 juste une illustration des règles de transformation (quelques fonctions et actions seulement). La transformation complète du modèle PMS est donnée dans l'annexe A de ce document.
- Après la définition des propriétés du système PMS à travers l'invariant, nous traduisons le modèle des opérations de reconfiguration dans le langage B. L'application successive des règles de transformation de R6 jusqu'à R11 sur le modèle de l'opération Insert-Patient et l'utilisation de la grammaire de génération des prédicats B, donne l'opération présentée dans la machine B de la figure 5.6 suivante.

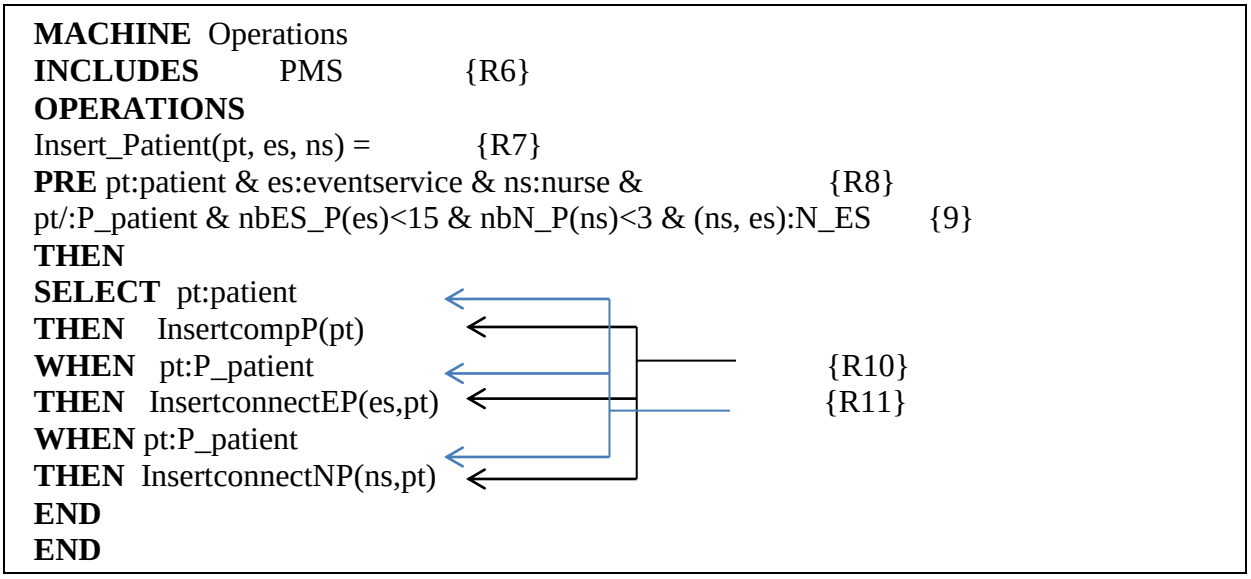


Fig.5.6. Transformation du modèle de l'opération d'insertion d'un patient vers une opération B.

- Après la transformation de toutes les opérations de reconfiguration vers la machine Operations, nous passons à la transformation des règles de reconfiguration.

La figure 5.7 ci-dessous montre la transformation du modèle du plan d'exécution de l'évènement de transfert d'une infirmière par l'application des règles de R12 jusqu'à R16.

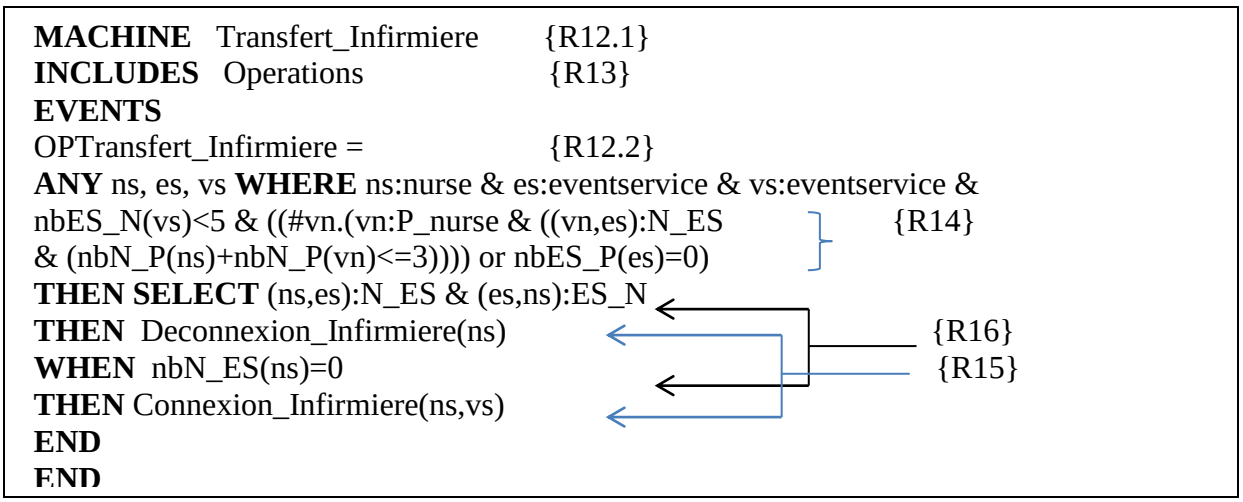


Fig.5.7. Transformation du modèle du plan de transfert d'une infirmière vers un évènement B.

5.5. Vérification et preuve

Le but principal de ce chapitre est de pouvoir faire la vérification et le raisonnement formel sur la spécification formelle d'un système adaptatif. En effet, la phase de génération d'une spécification formelle par une transformation vers B est une phase intermédiaire pour entamer l'étape de vérification et preuve. La preuve de cohérence de notre modèle revient donc en une preuve en B d'un modèle abstrait.

Comme nous avons dit précédemment, la cohérence d'un système abstrait est donnée par la préservation de ses propriétés invariantes et est garantie par des obligations de preuve [CAN06]. Ces obligation de preuve sont générées à partir des clauses INITIALISATION, OPERATIONS et EVENTS d'une machine B, afin de prouver que sous l'invariant en question, et sous la garde de chaque évènement et la pré-condition de chaque opération, cet invariant est préservé après la modification de l'état du système impliqué par un évènement ou une opération.

Nous avons proposé dans la première partie de ce chapitre, une transformation de la modélisation d'un système adaptatif vers un modèle B constitué de certains ensembles de machines, à savoir : la machine ModelArchitecture, la machine Operations et d'autres machines représentant chacune un plan d'exécution d'une reconfiguration.
Les dépendances entre ces machines sont données dans la figure 5.8 ci-dessous.

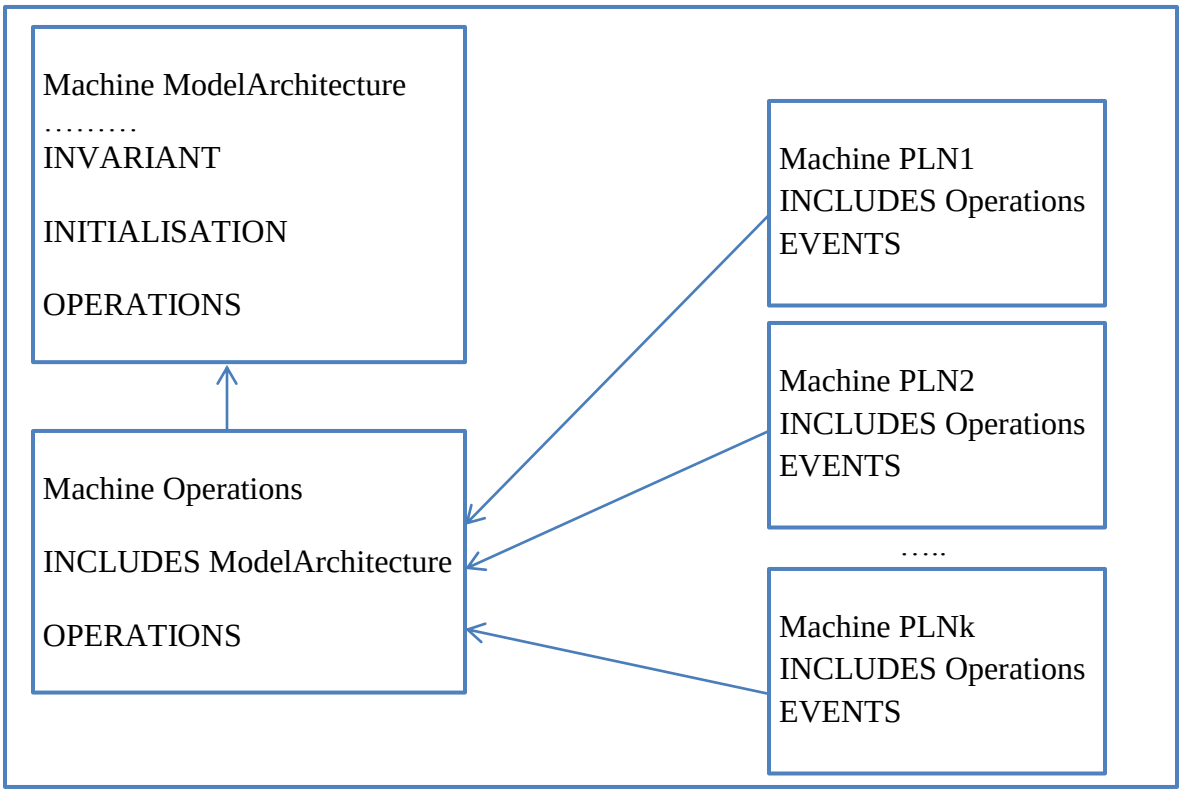


Fig.5.8. Graphe de dépendances entre les différentes machines du modèle construit.

D'après cette figure, les différentes machines représentant les plans d'exécution (évènements) dépendent de la machine Operations qui représente les opérations élémentaires à exécuter par ces plans. La machine Operations elle-même dépend de la machine ModelArchitecture, ce qui implique par transitivité que toutes les machines représentants les évènements, dépendent aussi de la machine ModelArchitecture. Par conséquent, les obligations de preuve de toutes les machines sont générées par rapport à l'invariant de la machine ModelArchitecture.

Cette phase de vérification permet de confirmer sans ambiguïté, la cohérence du système initialement (vérification de la configuration initiale) et aussi, après l'exécution d'une reconfiguration. Ceci, en démontrant la préservation des propriétés invariantes déclarées dans la clause INVARIANT de la machine ModelArchitecture.

Pour vérifier la consistance et la conformité de l'évolution du système par rapport à sa structure architecturale, nous procédons comme suit:

5.5.1. Préservation de l'invariant par l'initialisation du modèle abstrait

L'initialisation des variables d'état donnée par la clause INITIALISATION décrit l'état initial du système. Cette opération d'initialisation est considérée comme inconsistance si elle contient un prédicat non satisfaisant et/ou une contradiction entre les prédicats.

La vérification de la cohérence du système par rapport à son initialisation revient à prouver que l'obligation de preuve suivante est vraie.

$$\mathbf{Init(x) \Rightarrow I(x)} \qquad \mathbf{(INV1)}$$

Cette expression signifie que l'initialisation du système donnée par la clause INITIALISATION (Init), et qui décrit l'état du système par la variable d'état x, doit établir l'invariant I du système.
Dans notre cas, la clause INITIALISATION de la machine ModelArchitecture décrit la configuration initiale du système. Il s'agit donc de prouver que toutes les contraintes décrites dans la clause INVARIANT sont vérifiées par cette configuration.
Si nous montrons que INV1 est vrai, alors nous pouvons dire que l'initialisation du système est cohérente.

Dans notre spécification, nous avons défini le système par un ensemble de composants, de relations et de contraintes. Nous pouvons donc l'initialiser par l'instanciation de ces ensembles, de telle sorte qu'ils vérifient certaines contraintes.
Dans notre exemple PMS, comme le montre la figure 5.9, nous avons défini une configuration initiale du système qui contient une instance de composant de type EventService, une instance de type Nurse et une instance de type Patient connectées entre elles, et qui vérifient certaines propriétés.

```
INITIALISATION
P_ES, P_N, N_ES, N_P, ES_N, ES_P, nbES_N, nbES_P, nbP_ES, nbP_N, nbN_ES,
nbN_P, P_eventservice, P_patient, P_nurse :(
P_eventservice:FIN1(eventservice) & P_patient:FIN(patient) & P_nurse:FIN(nurse) &
P_ES:patient<->eventservice & dom(P_ES)<:P_patient & ran(P_ES)<:P_eventservice &
P_N:patient<->nurse & dom(P_N)<:P_patient & ran(P_N)<:P_nurse &
N_ES:nurse<->eventservice & dom(N_ES)<:P_nurse & ran(N_ES)<:P_eventservice &
N_P:nurse<->patient & dom(N_P)<:P_nurse & ran(N_P)<:P_patient &
ES_N:eventservice<->nurse & dom(ES_N)<:P_eventservice & ran(ES_N)<:P_nurse &
ES_P:eventservice<->patient & dom(ES_P)<:P_eventservice & ran(ES_P)<:P_patient &
nbES_N:eventservice-->NATURAL & dom(nbES_N)<:P_eventservice &
nbES_P:eventservice-->NATURAL & dom(nbES_P)<:P_eventservice &
nbP_ES:patient-->NATURAL & dom(nbP_ES)<:P_patient &
nbP_N:patient-->NATURAL & dom(nbP_ES)<:P_patient &
nbN_ES:nurse-->NATURAL & dom(nbN_ES)<:P_nurse &
nbN_P:nurse-->NATURAL & dom(nbN_P)<:P_nurse &
card(P_eventservice)=1 & (!ev.(ev:P_eventservice=>(nbES_P(ev)=1 & nbES_N(ev)=1)))
& card(P_patient)=1 & (!pt.(pt:P_patient=>(nbP_ES(pt)=1 & nbP_N(pt)=1))) &
card(P_nurse)=1 & (!ns.(ns:P_nurse=>( nbN_ES(ns)=1 & nbN_P(ns)=1)))
)
```

Fig.5.9. Initialisation du système PMS dans la machine B.

5.5.2. Préservation de l'invariant par chaque évènement

Afin de prouver que le système évolue conformément aux contraintes données par le modèle de sa structure architecturale, il faudrait vérifier que chaque plan d'exécution d'un évènement, préserve les propriétés invariantes définies dans la clause INVARIANT de la machine ModelArchitecture. L'obligation de preuve qui permet de vérifier la consistance de l'évolution du système par un évènement e, est donnée comme suit :

$$\mathbf{I(x) \wedge BA(e)(x, x0)) \Rightarrow I(x0)} \qquad \mathbf{(INV2)}$$

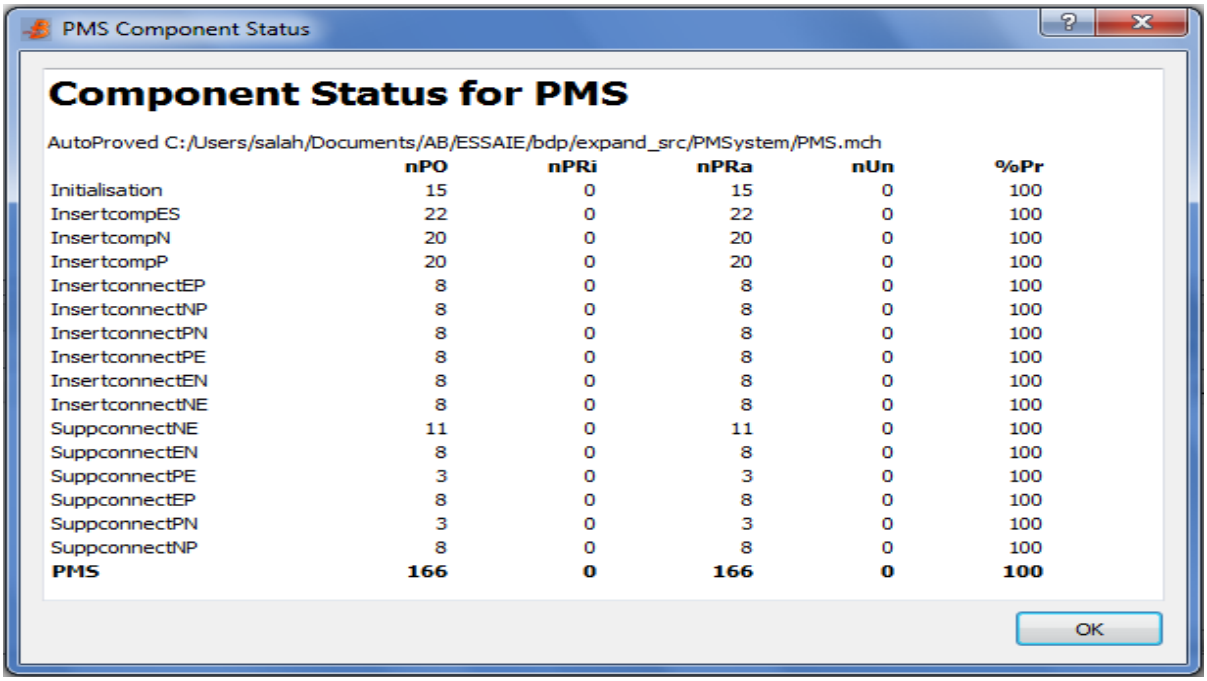
Cette expression signifie que si l'état de la variable x établit l'invariant I et après l'application de l'évènement e l'état de la variable x est transformé en x0 alors x0 aussi doit établir I.
Nous avons spécifié précédemment chaque plan d'exécution (évènement) dans la clause EVENTS d'une machine abstraite. Ces évènements dépendent des opérations déclarées dans la machine Operations qui elles-mêmes dépendent des actions déclarées dans la clause OPERATION de la machine ModelArchitecture. Il faudrait donc tout d'abord vérifier la préservation de l'invariant de la machine ModelArchitecture par toutes ces opérations avant de la vérifier pour chaque évènement.

Pour évaluer l'effet de toutes les reconfigurations sur le système, nous devons prouver toutes les obligations de preuve générées par tous les évènements et toutes les opérations. Nous utilisons

pour ça, le système de preuve de l'atelier B, qui dispose d'un générateur des obligations de preuve et d'un prouveur automatique et interactif.

Pour la vérification de la cohérence de la spécification donnée au système PMS, nous avons prouvé automatiquement à travers l'atelier B, les obligations de preuve générées par les différentes opérations et évènements.
Nous avons utilisé la commande « Generate Pos » pour la génération des obligations de preuve et « Automatic Proof » pour la preuve automatique des obligations générées.

Voici le statut de la machine principale.



	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	15	0	15	0	100
InsertcompES	22	0	22	0	100
InsertcompN	20	0	20	0	100
InsertcompP	20	0	20	0	100
InsertconnectEP	8	0	8	0	100
InsertconnectNP	8	0	8	0	100
InsertconnectPN	8	0	8	0	100
InsertconnectPE	8	0	8	0	100
InsertconnectEN	8	0	8	0	100
InsertconnectNE	8	0	8	0	100
SuppconnectNE	11	0	11	0	100
SuppconnectEN	8	0	8	0	100
SuppconnectPE	3	0	3	0	100
SuppconnectEP	8	0	8	0	100
SuppconnectPN	3	0	3	0	100
SuppconnectNP	8	0	8	0	100
PMS	**166**	**0**	**166**	**0**	**100**

Fig.5.10. Preuve de l'initialisation et de quelques actions du système PMS.

5.6. Conclusion

Dans ce chapitre, nous avons proposé une approche de vérification et de validation basée sur la transformation d'une modélisation manuelle pouvant avoir des incohérences vers une spécification formelle vérifiable automatiquement. Pour cela, nous avons tout d'abord présenté un pont (ensemble de règles) permettant de transformer un modèle défini selon notre notation proposée dans le chapitre 4, vers un modèle B comportant des expressions exprimées dans le langage B. Ce pont est constitué d'un ensemble de règles de passage qui définissent l'équivalent de chaque partie de notre modèle dans le langage B.
Par la suite, nous avons opté pour une vérification automatique de la consistance d'un système. Ce qui permet de prouver que son évolution est toujours conforme à ses propriétés architecturales et fonctionnelles. Ces preuves ont été réalisées en utilisant le système de preuve de l'Atelier B.

Conclusion et perspectives

Ce mémoire a comme objectif, l'étude du problème de la spécification formelle des systèmes adaptatifs et la preuve de sa validation.

Pour mener à bien cette étude, nous avons décomposé ce mémoire en deux parties.

La première partie de notre travail a été consacré à l'état de l'art, dans lequel nous avons débuté par une brève présentation d'un aperçu sur les systèmes adaptatifs et l'adaptation dynamique. Nous avons vu que les deux éléments clés de ces systèmes sont la modification et la modularité. De ce fait, nous avons présenté par la suite, la programmation par composants et son lien avec l'adaptation. Après avoir étudier tous les concepts liés à la reconfiguration dynamique, et puisque notre travail est orienté méthodes formelles, nous avons étudié quelques méthodes formelles existantes permettant la modélisation des systèmes. Et comme dernier point de cette partie, nous avons présenté quelques approches ayant traité la reconfiguration dynamique, et nous avons terminé par un bilan qui détermine l'orientation de notre approche.

Quant à la deuxième partie, elle a été consacrée à notre proposition d'une méthode permettant une spécification formelle valide d'un système adaptatif. Nous avons, dans ce but là, proposé une méthode basée sur la transformation d'un modèle semi-formel, vers une spécification formelle utilisant le langage Event-B et vérifiable automatiquement à l'aide de l'Atelier B.

Pour mener à bien notre proposition, tout d'abord, nous avons modélisé un système adaptatif par composant en utilisant le formalisme de transformation de graphes et le formalisme fonctionnel, tout en prenant compte des contraintes imposées au système (les différentes contraintes sont écrites en utilisant la logique de prédicats du premier ordre et la théorie des ensembles). Ainsi, le modèle obtenu est constitué de trois sous modèles. Le premier représente la structure du système avec ses propriétés invariantes. Le second représente les différentes opérations permettant la transformation du système d'une configuration à une autre. Et le dernier sous modèle a été consacré aux règles de reconfiguration (notre approche est guidée par un modèle de politiques d'adaptation décrites sous la forme E-C-A « Evènement si Condition alors Action » et utilisant l'ensemble des opérations de reconfiguration représentées par le modèle précédant). Ces règles d'adaptation sont décrites dans un modèle séparé du modèle représentant le système, ce qui permet sa modification dynamique (même pendant l'exécution du système).

Ces différents sous modèles sont transformés par la suite vers une spécification formelle exprimée avec le langage formelle Event-B, en utilisant certaines règles de transformation proposées, afin de garantir leurs validités, leurs consistances et leurs cohérences.
Nous avons illustré notre méthode à travers l'étude de cas du système de contrôle de patients PMS (Patient Monitoring System).

Dans notre modélisation, nous avons adopté une modélisation modulaire qui permet la réutilisation. Aussi, nous avons pris en considération l'ordonnancement entre les différentes

actions d'une reconfiguration sous contraintes de précédence. Ce qui n'a pas été pris par les différentes approches existantes dans la littérature.

Comme perspectives:

Nous prévoyons d'améliorer la partie vérification en y'ajoutant d'autres propriétés à vérifier autre que la cohérence de l'évolution du système. À titre d'exemple, l'absence de blocage (deadlock-freeness) doit être établie pour tout système abstrait (la disjonction des gardes des évènements doit être vraie).

Pour que notre approche soit exploitée, elle doit offrir aux utilisateurs les moyens pour la manipuler. Nous prévoyons une implémentation d'un outil permettant le passage automatique entre le modèle proposé et l'approche de validation et de vérification par une intégration de ce dernier avec l'Atelier B.

Le travail que nous avons réalisé dans ce mémoire, peut être étendu dans d'autres directions. Par exemple, nous planifions de modéliser un système en utilisant la programmation par aspects, pour détailler plus l'aspect fonctionnel des composants.

Les Annexes

Spécification B du système de contrôle de patients (PMS) à l'aide des règles de transformation proposées dans le chapitre 5.

MACHINE PMS

SETS
Interface; Etat={active,passive,frozen}; Role={fourni,requis}; patient; eventservice; nurse

VARIABLES
If, Ir, Rel, RoleI, Connect, Petat, Netat, ESetat, IPT, INS, IES, PTI, NSI, ESI,
P_patient, P_eventservice, P_nurse, P_ES, P_N, N_ES, N_P, ES_N, ES_P,
nbES_N, nbES_P, nbP_ES, nbP_N, nbN_ES, nbN_P

INVARIANT
If:FIN(Interface) & Ir:FIN(Interface) & Rel:Ir<->If & RoleI:Interface-->Role & Connect:Ir-->If
& P_patient:FIN(patient) & P_eventservice:FIN1(eventservice) & P_nurse:FIN(nurse) &
Petat:P_patient-->Etat & Netat:P_nurse-->Etat & ESetat:P_eventservice-->Etat &
IPT:P_patient-->POW(Interface) & INS:P_nurse-->POW(Interface) &
IES:P_eventservice-->POW(Interface) & PTI:Interface-->P_patient & NSI:Interface-->P_nurse
& ESI:Interface-->P_eventservice &
nbES_N:eventservice-->NATURAL & dom(nbES_N)<:P_eventservice &
nbES_P:eventservice-->NATURAL & dom(nbES_P)<:P_eventservice &
nbP_ES:patient-->NATURAL & dom(nbP_ES)<:P_patient &
nbP_N:patient-->NATURAL & dom(nbP_N)<:P_patient &
nbN_ES:nurse-->NATURAL & dom(nbN_ES)<:P_nurse &
nbN_P:nurse-->NATURAL & dom(nbN_P)<:P_nurse &
P_ES:patient<->eventservice & dom(P_ES)<:P_patient & ran(P_ES)<:P_eventservice &
P_N:patient<->nurse & dom(P_N)<:P_patient & ran(P_N)<:P_nurse &
N_ES:nurse<->eventservice & dom(N_ES)<:P_nurse & ran(N_ES)<:P_eventservice &
N_P:nurse<->patient & dom(N_P)<:P_nurse & ran(N_P)<:P_patient &
ES_N:eventservice<->nurse & dom(ES_N)<:P_eventservice & ran(ES_N)<:P_nurse &
ES_P:eventservice<->patient & dom(ES_P)<:P_eventservice & ran(ES_P)<:P_patient &
card(P_eventservice)<=3 & card(P_eventservice)>0 &
!ev.(ev:P_eventservice => nbES_N(ev)<=5 & nbES_P(ev)<=15) &
!pt.(pt:P_patient => nbP_N(pt)=1 & nbP_ES(pt)=1) &
!nrs.(nrs:P_nurse => nbN_P(nrs)<=3 & nbN_ES(nrs)<=1) &
/* Si un composant est dans l'état active ou frozen, les liaisons de ses interfaces requises doivent être effectives. C'est-à-dire, les émissions et réceptions d'appels de méthodes métiers doivent s'exécuter normalement */
!cp.((cp:P_patient & (Petat(cp)=active or Petat(cp)=frozen))=>(IPT(cp) /\ Ir <:dom(Connect))) &
!ns.((ns:P_nurse & (Netat(ns)=active or Netat(ns)=frozen))=>(INS(ns) /\ Ir <: dom(Connect))) &
!ev.((ev:P_eventservice & (ESetat(ev)=active or ESetat(ev)=frozen))=>(IES(ev) /\ Ir <:
dom(Connect)))

INITIALISATION
If, Ir, Rel, RoleI, Connect, Petat, Netat, ESetat, IPT, INS, IES, PTI, NSI, ESI,
P_ES, P_N, N_ES, N_P, ES_N, ES_P, nbES_N, nbES_P, nbP_ES, nbP_N, nbN_ES, nbN_P,
P_eventservice, P_patient, P_nurse:(

If:FIN(Interface) & Ir:FIN(Interface) & Rel:Ir<->If & RoleI:Interface-->Role & Connect:Ir-->If & Petat:P_patient-->Etat & Netat:P_nurse-->Etat & ESetat:P_eventservice-->Etat & IPT:P_patient-->POW(Interface) & INS:P_nurse-->POW(Interface) & IES:P_eventservice-->POW(Interface) & PTI:Interface-->P_patient & NSI:Interface-->P_nurse & ESI:Interface-->P_eventservice &
P_eventservice:FIN1(eventservice) & P_patient:FIN(patient) & P_nurse:FIN(nurse) &
P_ES:patient<->eventservice & dom(P_ES)<:P_patient & ran(P_ES)<:P_eventservice &
P_N:patient<->nurse & dom(P_N)<:P_patient & ran(P_N)<:P_nurse &
N_ES:nurse<->eventservice & dom(N_ES)<:P_nurse & ran(N_ES)<:P_eventservice &
N_P:nurse<->patient & dom(N_P)<:P_nurse & ran(N_P)<:P_patient &
ES_N:eventservice<->nurse & dom(ES_N)<:P_eventservice & ran(ES_N)<:P_nurse &
ES_P:eventservice<->patient & dom(ES_P)<:P_eventservice & ran(ES_P)<:P_patient &
nbES_N:eventservice-->NATURAL & dom(nbES_N)<:P_eventservice &
nbES_P:eventservice-->NATURAL & dom(nbES_P)<:P_eventservice &
nbP_ES:patient-->NATURAL & dom(nbP_ES)<:P_patient &
nbP_N:patient-->NATURAL & dom(nbP_ES)<:P_patient &
nbN_ES:nurse-->NATURAL & dom(nbN_ES)<:P_nurse &
nbN_P:nurse-->NATURAL & dom(nbN_P)<:P_nurse &
card(P_eventservice)=1 & (!ev.(ev:P_eventservice =>(nbES_P(ev)=1 & nbES_N(ev)=1))) &
card(P_patient)=1 & (!pt.(pt:P_patient =>(nbP_ES(pt)=1 & nbP_N(pt)=1))) &
card(P_nurse)=1 & (!ns.(ns:P_nurse =>(nbN_ES(ns)=1 & nbN_P(ns)=1))) &
PTI={} & NSI={} & ESI={})

OPERATIONS
InsertcompES(es) =
PRE es:eventservice & card(P_eventservice)<3 THEN
nbES_P(es):=0 || nbES_N(es):=0 || P_eventservice:=P_eventservice\/{es} || IES(es):={}
END;
InsertcompN(ns) =
PRE ns:nurse THEN
nbN_P(ns):=0 || nbN_ES(ns):=0 || P_nurse:=P_nurse\/{ns} || INS(ns):={}
END;
InsertcompP(ps) =
PRE ps:patient THEN
nbP_ES(ps):=1 || nbP_N(ps):=1 || P_patient:=P_patient\/{ps} || IPT(ps):={}
END;
InsertconnectEP(es,pt) =
PRE es:eventservice & pt:patient & (es,pt)/:ES_P & nbES_P(es)<15 THEN
ES_P:=ES_P\/{(es|->pt)} || nbES_P(es):=nbES_P(es)+1
END;
InsertconnectNP(ns,pt) =
PRE ns:nurse & pt:patient & (ns,pt)/:N_P & nbN_P(ns)<3 THEN
N_P:=N_P\/{(ns|->pt)} || nbN_P(ns):=nbN_P(ns)+1
END;
InsertconnectPN(pt,ns) =
PRE ns:nurse & pt:patient & (pt,ns)/:P_N THEN
P_N:=P_N\/{(pt|->ns)} || nbP_N(pt):=1
END;

InsertconnectPE(pt,es) =
PRE es:eventservice & pt:patient & (pt,es)/:P_ES THEN
P_ES:=P_ES∨{(pt|->es)} || nbP_ES(pt):=1
END;
InsertconnectEN(es,ns) =
PRE es:eventservice & ns:nurse & (es,ns)/:ES_N & nbES_N(es)<5 THEN
ES_N:=ES_N∨{(es|->ns)} || nbES_N(es):=nbES_N(es)+1
END;
InsertconnectNE(ns,es) =
PRE es:eventservice & ns:nurse & (ns,es)/:N_ES THEN
N_ES:=N_ES∨{(ns|->es)} || nbN_ES(ns):=1
END ;
SuppconnectNE(ns,es) =
PRE ns:nurse & es:eventservice & Netat(ns)=frozen & ESetat(es)=passive THEN
ANY es WHERE es:P_eventservice & (ns,es):N_ES THEN
N_ES:=N_ES-{(ns|->es)} || nbN_ES(ns):=0 || nbN_P(ns):=0 END
END;
SuppconnectEN(es,ns) =
PRE ns:nurse & es:eventservice & ESetat(es)=frozen & Netat(ns)=passive THEN
ANY es WHERE es:P_eventservice &(es,ns):ES_N & nbES_N(es)>=1 THEN
ES_N:=ES_N-{(es|->ns)} || nbES_N(es):=nbES_N(es)-1 END
END;
SuppconnectPE(ps,es) =
PRE ps:patient & es:eventservice & Petat(ps)=frozen & ESetat(es)=passive THEN
ANY es WHERE es:P_eventservice &(ps,es):P_ES THEN
P_ES:=P_ES-{(ps|->es)} END
END;
SuppconnectEP(es,ps) =
PRE ps:patient & es:eventservice & ESetat(es)=frozen & Petat(ps)=passive THEN
ANY es WHERE es:P_eventservice & (es,ps):ES_P & nbES_P(es)>=1 THEN
ES_P:=ES_P-{(es|->ps)} || nbES_P(es):=nbES_P(es)-1 END
END;
SuppconnectPN(ps,ns) =
PRE ns:nurse & ps:patient & Petat(ps)=frozen & Netat(ns)=passive THEN
ANY ns WHERE ns:P_nurse &(ps,ns):P_N THEN
P_N:=P_N-{(ps|->ns)} END
END;
SuppconnectNP(ns,ps) =
PRE ns:nurse & ps:patient & Netat(ns)=frozen & Petat(ps)=passive THEN
ANY ns WHERE ns:P_nurse &(ns,ps):N_P & nbN_P(ns)>=1 THEN
nbN_P(ns):=nbN_P(ns)-1 || N_P:=N_P-{(ns|->ps)} END
END
END

****

MACHINE Operations

INCLUDES PMS
OPERATIONS
Insert_EventService(xc) =
PRE xc:eventservice & xc/:P_eventservice & card(P_eventservice)<3 THEN
InsertcompES(xc)
END;
Insert_Patient(pt,es,ns) =
PRE pt:patient & es:eventservice & ns:nurse & pt/:P_patient & nbES_P(es)<15 & nbN_P(ns)<3 & (ns,es):N_ES THEN SELECT pt:patient THEN InsertcompP(pt)
WHEN pt:P_patient THEN InsertconnectEP(es,pt) WHEN pt:P_patient THEN
InsertconnectNP(ns,pt) END
END;
Insert_Nurse(ns,es) =
PRE ns:nurse & es:eventservice & nbES_N(es)<5 & ns/:P_nurse THEN SELECT ns:nurse THEN InsertcompN(ns) WHEN ns:P_nurse & (ns,es)/:N_ES THEN InsertconnectNE(ns,es) WHEN ns:P_nurse & (es,ns)/:ES_N THEN InsertconnectEN(es,ns) END
END;
Deconnexion_Infirmiere(ns) =
PRE ns:nurse THEN ANY es WHERE es:eventservice & ((#vn.(vn:P_nurse &((vn,es):N_ES & (nbN_P(ns)+nbN_P(vn)<=3)))) or nbES_P(es)=0) THEN
SELECT(ns,es):N_ES & Netat(ns)=frozen & ESetat(es)=passive THEN
SuppconnectNE(ns,es) WHEN (es,ns):ES_N & ESetat(es)=frozen & Netat(ns)=passive THEN SuppconnectEN(es,ns) END END
END;
Connexion_Infirmiere(ns,es) =
PRE ns:nurse & nbN_ES(ns)=0 & es:eventservice & nbES_N(es)<5 THEN SELECT(ns,es)/:N_ES THEN InsertconnectNE(ns,es) WHEN (es,ns)/:ES_N THEN
InsertconnectEN(es,ns) END
END
END

****

MACHINE Transfert_Infirmiere
INCLUDES Operations
EVENTS
OPTransfert_Infirmiere =
ANY ns, es, vs WHERE ns:nurse & es:eventservice & vs:eventservice & nbES_N(vs)<5 & ((#vn.(vn:P_nurse & ((vn,es):N_ES & (nbN_P(ns)+nbN_P(vn)<=3)))) or nbES_P(es)=0) THEN SELECT (ns,es):N_ES & (es,ns):ES_N THEN
Deconnexion_Infirmiere(ns) WHEN nbN_ES(ns)=0 THEN Connexion_Infirmiere(ns,vs)
END
END
END

1. Un aperçu sur la méthode B

La méthode B est une méthode à base d'état développée par Abrial pour spécifier, concevoir et coder des systèmes logiciels. Elle est basée sur la théorie des ensembles, la logique de prédicats du premier ordre et un langage de substitutions généralisées. Les ensembles sont utilisés pour la modélisation des données, les substitutions généralisées sont utilisées pour décrire les modifications d'états et des mécanismes de raffinement sont utilisés pour décrire le système aux différents niveaux d'abstraction (machine, raffinement, implémentation). Le but principal d'un développement B est d'obtenir un modèle prouvé correcte [CAN06].

B événementiel est une extension de la méthode B classique. La méthode Event-B est utilisée pour spécifier correctement et modéliser itérativement des systèmes complexes, distribués ou réactifs, par un mécanisme de raffinement. Le raffinement est une technique incrémentale visant à transformer un modèle abstrait du système en un modèle plus concret ; c'est-à-dire un modèle contenant plus de détails dans sa spécification, ou étant plus près d'une implémentation [CHA10].
La correction d'un modèle Event-B est donnée par les obligations de preuve qui sont générées pour montrer sa cohérence [YAN10]. La preuve des obligations de preuve se fait par un outil de preuve, tel que l'Atelier B, en utilisant des procédures automatiques ou interactives.
Au niveau le plus abstrait, il est nécessaire de décrire les propriétés statiques du modèle au moyen d'un prédicat invariant. Ces propriétés invariantes doivent être préservées par tous les évènements et les opérations du modèle pour assurer sa consistance [CAN06]. La réutilisation des modèles développés et les mécanismes de structuration disponible en B aident à diminuer la complexité.

1.1. Spécification abstraite

La modélisation abstraite d'un système dynamique est la première étape d'une représentation qui spécifie le comportement minimal du système, ses propriétés attendues, les lois fondamentaux qui le caractérisent ainsi que son environnement. L'état du système modélisé peut évoluer par l'application des événements. Ceux-ci sont décrits en termes d'actions gardées, et ils sont susceptibles d'être déclenchés quand leur gardes deviennent vraies. Si l'un des événements est appliqué, l'état du système change en fonction des actions associées à l'évènement.
Cette spécification n'indique pas comment fonctionne le système, mais indique la manière de s'assurer que ce qui sera réalisé en fin de conception sera jugé correct.
Dans cette spécification l'expression de propriétés est basée sur la logique de prédicats du premier ordre et la théorie des ensembles [CHA10].

Un modèle B évènementiel décrit un système de transitions par un ensemble d'états, un ensemble d'actions, un état initial et une relation de transition. Les actions effectuées par le système permettent la transition entre états. Le système de transitions est décrit via des événements dont le développeur donne la condition de déclenchement (garde de l'événement) et les actions instantanées associées (corps de l'événement) [AAM10].

Parmi les clauses composant la structure de ce modèle, nous citons les suivantes.

- **Model :** introduisant un nom unique au modèle.
- **Sets :** contenant les définitions des ensembles abstraits du modèle.
- **Variables :** déclarant une liste des variables (états) du système.
- **Invariant :** correspondant à la définition des propriétés des variables ; il s'agit des propriétés logiques qui expriment en logique du premier ordre les propriétés invariantes d'un modèle.
- **Initialisation**: permettant de spécifier, en utilisant des substitutions, l'initialisation des variables (état initial).
- **Events :** regroupant une collection de transitions. Chaque évènement est constitué d'une garde et d'un corps. Lorsque la garde est satisfaite, l'évènement est activé. Lorsque les gardes de plusieurs évènements sont satisfaites en même temps, le choix de l'évènement à activer est indéterministe.

1.2. Raffinement

Le raffinement est une technique visant la transformation d'un modèle abstrait en un modèle plus concret, c'est-à-dire un modèle contenant plus de détails dans sa spécification. La progression vers une implantation s'effectue par un processus de raffinement. De nouvelles variables peuvent être introduites et les anciennes raffinées en des variables plus concrètes. De nouveaux évènements peuvent aussi être introduits, ils ne doivent pas empêcher les anciens d'être activés. La clause VARIANT satisfait cette propriété; il s'agit d'une expression à valeur entière décrémentée par chaque nouvel évènement.

À chaque étape du développement du système par raffinement, on vérifie la correction et la cohérence du modèle raffiné par rapport au modèle abstrait associé par la démonstration d'obligations de preuve spécifiques : préservation de l'invariant, décroissance du variant, non contradiction entre substitutions concrètes et abstraites, etc. [CHA10]

2. Définition des prédicats B

En effet, le langage B supporte bien les concepts que nous avons utilisé dans notre modélisation pour décrire les contraintes, à savoir: la théorie des ensembles et la logique de prédicats du premier ordre. Nous présentons dans ce qui suit la syntaxe de génération des prédicats B à travers une grammaire. [MRB05]

Pour la grammaire de génération des prédicats B défini ci-dessous, nous utilisons les conventions suivantes :

– Les choix sont séparés par des barres verticales : |

– Tous les opérateurs indiqués entre guillemets " " sont des symboles terminaux, les autres mots désignent les non-terminaux.

– Les règles de production sont des paires formées d'un non-terminal à gauche et d'une suite de terminaux et de non-terminaux à droite.

Prédicat ::=

Prédicat_parenthésé | Prédicat_conjonction | Prédicat_négation |
Prédicat_disjonction | Prédicat_implication | Prédicat_équivalence — « **Propositions** »

| Prédicat_universel | Prédicat_existentiel — « **Prédicats quantifiés** »

| Prédicat_égalité | Prédicat_inégalité — « **Prédicats d'égalité** »

| Prédicat_appartenance | Prédicat_non_appartenance — « **Prédicats d'appartenance** »

| Prédicat_inclusion | Prédicat_inclusion_stricte
| Prédicat_non_inclusion | Prédicat_non_inclusion_stricte — « **Prédicats d'inclusion** »

| Prédicat_inférieur_ou_égal
| Prédicat_strictement_inférieur — « **Prédicats de comparaison d'entiers** »
| Prédicat_supérieur_ou_égal
| Prédicat_strictement_supérieur

➢ **Propositions**

Opérateur	**Syntaxe**
( ) Parenthèses	Prédicat_parenthésé ::= "(" Prédicat ")"
∧ Conjonction	Prédicat_conjonction ::= Prédicat "∧" Prédicat
¬ Négation	Prédicat_négation ::= "¬" "(" Prédicat ")"
∨ Disjonction	Prédicat_disjonction ::= Prédicat "∨" Prédicat
⇒ Implication	Prédicat_implication ::= Prédicat "⇒" Prédicat
⇔ Équivalence	Prédicat_équivalence ::= Prédicat "⇔" Prédicat

➢ **Prédicats quantifiés**

Opérateur	**Syntaxe**
∀ Quantificateur universel	Prédicat_universel ::= "∀" Liste_ident "." "(" Prédicat "⇒" Prédicat ")"
∃ Quantificateur existentiel	Prédicat_existentiel ::= "∃" Liste_ident "." "(" Prédicat ")"

N.B : Les variables X introduites par un prédicat universel de la forme ∀ X. (P ⇒ Q) où un prédicat existentiel de la forme ∃ X. (P) doivent être typées par un prédicat de typage dans une liste de conjonctions situées au plus haut niveau d'analyse syntaxique du prédicat P. Ces variables ne peuvent pas être utilisées dans P avant d'avoir été typées.

- **Prédicats d'égalité**

Opérateur	Syntaxe
= Égalité ≠ Inégalité	Prédicat_égalité ::= Expression "=" Expression Prédicat_inégalité ::= Expression "≠" Expression

N.B : Dans les prédicats *x = y* et *x≠ y*, les expressions *x* et *y* doivent avoir le même type.

- **Prédicats d'appartenance**

Opérateur	Syntaxe
∈ Appartenance ∉ Non appartenance	Prédicat_appartenance ::= Expression "∈" Expression Prédicat_non_appartenance ::= Expression "∉" Expression

N.B : Dans les prédicats x ∈ E et x ∉ E, si le type de l'expression x est T alors le type de E doit être P(T).

- **Prédicats d'inclusion**

Opérateur	Syntaxe
⊆ Inclusion ⊂ Inclusion stricte ⊄ Non inclusion ⊄ Non inclusion stricte	Prédicat_inclusion ::= Expression " ⊆ " Expression Prédicat_inclusion_stricte ::= Expression " ⊂ " Expression Prédicat_non_inclusion ::= Expression " ⊄ " Expression Prédicat_non_inclusion_stricte ::= Expression " ⊄ " Expression

N.B : Dans les prédicats X ⊆ Y, X ⊂ Y, X ⊄ Y, X ⊄ Y, les expressions X et Y sont du même type, et leur type est de la forme P(T).

- **Prédicats de comparaison d'entiers**

Opérateur	Syntaxe
≤ Inférieur ou égal < Strictement inférieur ≥ Supérieur ou égal > Strictement supérieur	Prédicat_inférieur_ou_égal ::= Expression "≤" Expression Prédicat_strictement_inférieur ::= Expression "<" Expression Prédicat_supérieur_ou_égal ::= Expression "≥" Expression Prédicat_strictement_supérieur ::= Expression ">" Expression

N.B : Dans les prédicats x ≤ y, x < y, x ≥ y, x > y, les expressions x et y doivent représenter des entiers.

Remarque : La liste des mots réservés et des opérateurs du Langage B ainsi que les règles de génération des Expressions sont données dans l'Annexe A et B respectivement du manuel de référence B [MRB05].

Bibliographie

[AAM10]: Yamine Ait-Ameur, Idir Ait-Sadoune et Mickael Baron. Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes multi-modaux. Journal d'Interaction Personne-Système, Vol. 1, No. 1, Art. 3, Septembre 2010.

[ALD02]: Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJav: Connecting software architecture to implementation. In International Conference on Software Engineering, ICSE 2002, Orlando, Florida, USA, May 2002.

[AND00]: Françoise Andre, Anne-Marie Kerrnarrec, Frederic Le Mouel. Improvement of the QoS via an Adaptive and Dynamic Distribution of Applications in a Mobile Environment. In Proc of the 19TH IEEE Symp, On Reliable Distributed Systems, Nurnberg 2000.

[APO09]: M. Aponte **,** V. Benayoun **,** M. Simonot**.** Modélisations de la reconfiguration dynamique en Focal**.** AFADL'09 Approches formelles dans l'assistance au développement des Logiciels, Toulouse, January 2009, pp.105-119.

[BAL10]: Cyril BALLAGNY. MOCAS : Un modèle de composants basé états pour l'auto-adaptation. Thèse de doctorat, Université de Pau et des Pays de l'Adour , Mars 2010.

[BAR05]: Tomàs BARROS. Formal specification and verification of distributed component systems. Thèse de doctorat, Université de Nice-Sophia Antipolis, novembre 2005.

[BHA07] : Riadh Ben Halima, Mohamed Jmaiel, Khalil Drira. Une approche orientée règle pour la spécification formelle des architectures dynamiquement configurables. 7ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE 2007). 4-8 Juin, 2007, Marrakech, Maroc. Pages 405- 412.

[BOU06]: Jérémy Bouisson. Adaptation dynamique de programmes et composants parallèles. Thèse de doctorat, Institut National des Sciences Appliquées de Rennes, Septembre 2006.

[BRA04] : Jeremy S. Bradbury. Organizing Definitions and Formalisms for Dynamic Software Architectures. Technical Report, School of Computing, Queen's University; March 2004.

[CAN06]: Dominique Cansell, Dominique Méry. Tutorial on the event-based B method: Concepts and Case Studies. 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems, September 26-29 2006, Paris, France.

[CAR03]: Cyril Carrez. Contrats Comportementaux pour Composants. Thèse de Doctorat, école Nationale Supérieure des Télécommunications, Paris, Décembre 2003.

[CAS02] : Ludovic Casset. Construction Correcte de Logiciels pour Carte à Puce, Développement formel d'un vérifieur embarqué de byte code Java Card à l'aide de la méthode B. Thèse de doctorat, Université d'Aix-Marseille II, Université de la Méditerranée, octobre 2002.

[CAZ98] : W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection: Bridging the gap between a running system and its architectural specification. In Proc Reengineering Forum '98, 1998.

[CHA10]: Jean-Charles Chaudemar. Analyse de sécurité de systèmes autonomes : Formalisation et évaluation en B. Toulouse, Journées des Thèses 25-27 janvier 2010.

[CHA07]: Tarak CHAARI. Adaptation d'applications pervasives dans des environnements multi-contextes. Thèse de doctorat, Institut national des sciences appliquées de Lyon, 2007.

[CHE05]: Djalel Chefrour. Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique. Thèse de doctorat, Universite de Rennes 1, novembre 2005.

[CHE01]: Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In Proceedings of the 21st International Conference on Distributed Computing Systems, 2001.

[COR01]: Manuel Aguilar Cornejo, Hubert Garavel, Radu Mateescu, Noël de Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems, Juillet 2001.

[DAV05]: Pierre-Charles DAVID. Développement de composants Fractal adaptatifs : Un langage dédié à l'aspect d'adaptation. Thèse de doctorat, Université de Nantes, Juillet 2005.

[DEY99]: Anind K. Dey and Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. In proceeding HUC'99 Proceeding of the 1st international symposium on Handheld and Ubiquitous Computing, Springer-Verlag London, 1999.

[DOW01]: Jim Dowling and Vinny Cahill. The K-Component architecture meta-model for self adaptive software. In A. Yonezawa and S. Matsuoka, editors, Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

[DUF03]: Guillaume Dufay. Vérification formelle de la plate-forme Java Card. Thèse de doctorat, Université de Nice-Sophia Antipolis, Décembre 2003.

[EVB05]: C. Métayer J. R. Abrial, L. Voisin. Event-B Language. May 2005.

[FIT98]: Tom Fitzpatrick, Gordon S. Blair, Geoff Coulson, Nigel Davies and Philippe Robin. A Software Architecture for Adaptive Distributed Multimedia Systems. IEE Proceedings-Software 145(5): 163-171, Lancaster University (1998).

[GAR00]: David Garlan, Robert T. Monroe, and David Wile. Acme : architectural description of component-based systems. In Foundations of component-based systems, Cambridge University Press, New York, NY, USA, 2000.

[HAD08]: Mohamed HADJ KACEM. Modélisation des applications distribuées à architecture dynamique: Conception et Validation. Thèse de doctorat, Université Toulouse III - Paul Sabatier et Université de Sfax, novembre 2008.

[HOF93]: C. Hofmeister, J. Purtilo. "Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for replacement". Proc. 13th International Conference on Distributed Computing Systems, pp. 101-110, 1993.

[KET04]: Abdelmadjid KETFI. Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels. Thèse de doctorat, Université Joseph Fourier, Décembre 2004.

[KOR00]: Pascal POIZAT. KORRIGAN : Un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes. Thèse de doctorat, Université de Nante, décembre 2000.

[KRA90]: J. Kramer, J. Magee. The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering, Vol. 16, No. 11, pp. 1293-1306, November 1990.

[KRA89]: Kramer J., Magee J., Sloman M. Constructing Distributed Systems in CONIC. IEEE Trans. Software Engineering, vol.SE-15(N.6), June 1989, pp.663-675.

[KRI09] **:** Fatma KRICHEN. Techniques et mécanismes de support d'exécution pour la reconfiguration dynamique des architectures logicielles à composant pour les systèmes embarqués. METHODICA-II'09 ; 5ème colloque sur les méthodes pour les logiciels distribués adaptatifs, Université de Sfax (Tunisie), Décembre 2009.

[KRU06]: Philippe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. In: Software, IEEE**,** Issue Date: March-April 2006 ,Volume: 23 Issue: 2 On page(s): 22 – 30.

[LEG09] : Marc LEGER. Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composants. Thèse de doctorat, Université de Nante, Mai 2009.

[LIE06] : June ANDRONICK LIEGE. Modélisation et vérification formelles de systèmes embarqués dans les cartes à microprocesseurs, plates-forme java card et système d'exploitation. Thèse de doctorat, Université de Paris XI, Orsay, décembre 2006.

[LOB99]: Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence, 1999.

[LOR07] : Nicolas Loriant. Évolution dynamique des systèmes d'exploitation ; une approche par la programmation par aspects. Thèse de doctorat, U niversité de Nante, Décembre 2007.

[LOU09]: LOUNAS Razika. Preuve en Coq de propriétés de programmes numériques partant du code en C. Mémoire de magister, Université de Boumerdès, 2009.

[LOU05]: Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, Khalil Drira. Approche formelle intégrée pour la spécification des architectures dynamiques orientées composants. Information Sciences for Decision Making, Special Issue of The 8Th MCSEAI'04, (19). Pages 29-42, 2005.

[MET96]: D. L. Métayer. Software architecture styles as graph grammars. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1996.

[MRB05] : Manuel de référence du langage B. Version 1.8.7 du 10 février 2009.

[ORE99] : Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L.Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 1999.

[ORE98]: Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 1998.

[ORZ98]: Peyman Oreizy. Issues in Modeling and Analyzing Dynamic Software Architectures. Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis, Marsala, Sicily, Italy, June 30 - July 3, 1998.

[PAL01]: N. De Palma, P. Laumay, and L. Bellissard. Ensuring dynamic reconfiguration consistency. In 6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop, pages 18-24, 2001.

[PAL99] :Noel de Palma, Luc Bellissard, Michel Riveill. Dynamic Reconfiguration of Agent-Based Applications. Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, 1999.

[PAY06]: François-Xavier Payet. Adaptation dynamique de composants parallèles. Rapport bibliographique, février 2006.

[POR03]: Vincent PORTIGLIATTI. Contribution a l'allocation dynamique de ressources pour les composants expressifs dans les systèmes repartis. Thèse de doctorat, Université de Franche-Comté, décembre 2003.

[PUR94]: Purtilo J. M.. the POLYLITH software bus. ACM TOPLAS, vol.16(N.1), Jan. 1994, pp.151-174.

[SAT05]: Sathish S, Vathiyar and Jack J.Dongarra. Self adaptivity in grid computing. University of Tennessee, Knoxville, Concurrency & Computation: Practice & Experience, Volume 17, issue 2-4, pages 235-257. Februry-april 2005.

[SAT01]: M. Satyanarayanan. Pervasive Computing: Vision and Challenges. IEEE Personal Communications, 2001.

[SAW94]: B.N.Schilit, N.I.Adams and R.Want. Context-Aware Computing Applications. In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA94), Santa Cruz CA, Décember 1994.

[SCH94]: B.N. Schilit, M. Theimer. Disseminating Active Map Information to Mobile Hosts, Network, IEEE, September/October 1994.

[SIM09]: Marianne Simonot, Maria-Virginia Aponte. A Declarative Formal Approach to Dynamic Reconfiguration. In proceeding IWOCE'09 Proceedings of the 1stinternational workshop on Open component ecosystems, 2009.

[SIM08]: Marianne Simonot, Maria-Virginia Aponte. Une approche formelle de la reconfiguration dynamique. Journal: Logiciel, Base De Données, Réseaux / Software, Databases, Networks - LOBJET **,** vol. 14, no. 4, pp. 73-102, 2008.

[SOU02]: Jean Louis Sourrouille, José Lino Contreras. Objets autonomes adaptables. Journées Systèmes à composants adaptables et extensibles, Grenoble, 2002.

[STO07] : Nicolas STOULS. Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés. Thèse de doctorat, Institut polytechnique de Grenoble, Décembre 2007.

[STY07]: Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, Iulian Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. Journal ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 29 Issue 4, August 2007.

[TOU10]: Imen Tounsi. Une Approche pour la Modélisation et la Vérification des Politiques d'Adaptation pour le Style P/S. mémoire de master, Université de Sfax, Mai 2010.

[WER98]: Michel Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. In: IEE Proceedings - Software, Vol. 145, Nr. 5 (1998) , p. 130-136.

[WER99]: Michel Wermelinger. Specification, Testing and Analysis of (Dynamic) Software Architecture with the Chemical AbstractMachine. In Newsletter ACM SIGSOFT Software Engineering Notes ,Volume 24, Issue 4,July 1999.

[XAV10]: Xavier Besseron. Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle. Thèse de doctorat, Université de Grenoble, Avril 2010.

[YAN10]: Faqing Yang, Jean-Pierre Jacquot. Prouvé ? Et après ?. 10es Journées Francophones Internationales sur les Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL 2010, 133-147.

[ZHA09]: Xun ZHANG. Contribution aux architectures adaptatives : étude de l'efficacité énergétique dans le cas des applications a parallélisme de données. Thèse de doctorat, Université Nancy-1, 2009.

[ZHA06]: Ji Zhang and Betty H.C. Cheng. Using Temporal Logic to Specify Adaptive Program Semantics. Journal of systems and software, JSS-vol .79, n° .10, pp.1361-1369, 2006.