UNIVERSITE SAAD DAHLAB DE BLIDA
Faculté des Sciences
Département d'Informatique

N° D'ordre :.....

Faculté des sciences

Département d'informatique

Mémoire Présenté par :

HAIBAOUI Fardous BEN MESSAOUD Zineb

En vue d'obtenir le diplôme de master
Domaine : Mathématique et informatique
Filière : Informatique
Spécialité : Informatique
Option : Ingénierie de logiciel

Thème : **Editeur Graphique pour l'aide à la spécification des architectures logicielle dynamiques**

Soutenu le : juin 2016, devant le jury composé de :

M. CHIKHI Imane	Président
M. MANCER Yasmine	Examinateur
M.	Examinateur
Mme. GUESSOUM D	Promotrice

Promotion
2015/ 2016

Remerciement

Nous tenons tout d'abord à remercier ***Dieu***, tout puissant de nous avoir accordé la connaissance, donner le courage, la patience et la santé pour mener à réaliser notre projet de fin d'étude.

Nous tenons à remercier vivement notre promotrice **Mme. *Dalila GUESSOUM***, d'avoir accepté de nous encadrer, ainsi que pour ses remarques constructives, ses recommandations, son grand soutien, et sa disponibilité.

Nous remercions également l'ensemble des enseignants qui ont assuré notre formation durant les cinq ans d'études.

Que l'ensemble du jury, trouvent ici l'expression de nos remerciements les plus s'incères d'avoir accepté d'examiner ce mémoire.

Enfin, nous remercions tous ceux qui, de près ou de loin ont contribué à la préparation de ce mémoire.

Dédicace

Le parcours d'une vie est jalonné d'opportunités qui dépendent de nous, mais également à des personnes qu'il nous a été donné de rencontrer: des personnes qui nous guident, qui nous conseillent et qui nous font confiance, j'ai eu de la chance de rencontrer quelques-uns d'entre eux et à qui je tiens à dédier ce mémoire :

À mes chers et adorables Parents: pour votre amour, votre affection et votre sacrifice consentis, pour votre soutien indéfectible, bienveillance et Vos encouragements, Que Dieu bienveillant vous garde à mon côté

À mes sœurs et mon frère

À tout la famille HAIBAOUI

À mon binôme Zineb qui m'a accompagné dans tout le long de ce travail, pour son courage et sont sérieux, ainsi que sa gentillesse.

À tous ceux que j'aime.

H. Fardous

Dédicace

C'est avec une immense fierté que je dédie ce mémoire:

*À celle qui a fait de moi ce que je suis aujourd'hui, grâce ses sacrifices, tendresses et son amours : **ma très chère mère Zahra** que dieu me la garde.*

À la mémoire de mon Père.

À mes sœurs et frères, cousins et cousines.

À tout la famille Ben Messaoud.

À Mes chères amies.

À tous ceux qui sont chères, proches de mon cœur, et à tous ceux qui m'aiment.

B.Zineb

Résumé

Dans ce travail, nous avons conçu une nouvelle approche de spécification d'architecture logicielle dynamique nommée ***DynamiqueIASA***, en se basant principalement sur les architectures orientée événements, Le méta-modèle proposé pour notre approche sera basé sur des concepts primitifs qui peuvent facilement être analysées avec des outils utilisant des notations graphiques proches de la perception intuitive des architectes logiciels. Nous avons développé pour notre approche un éditeur graphique correspondant à notre méta-modèle sur GMF (Graphical Modeling Framework).

Mots clés: Architecture logicielle dynamique; Architecture orientée événements; IDE; GMF; X3ADL.

Abstract

The aim of this work , is to develop a new approach for dynamic software architecture specification called ***DynamiqueIASA***, based mainly on event-driven architectures, the proposed meta-model of our approach will be based on primitive concepts that can be easily analyzed with tools using graphical notations close to the intuitive perception of software architects. We have developed a graphical editor corresponding to our meta-model based on GMF (Graphical Modeling Framework).

Keywords: Dynamic Software Architecture; Events Driven Architecture; IDE; GMF; X3ADL.

ملخص

في هذا العمل قمنا بطرح مقاربة جديدة لوصف هندسة البرمجيات الديناميكية تسمى ***DynamiqueIASA*** , تقوم اساسا على هندسيات الاحداث الموجهة .النموذج المقترح يستند اساسا على المفاهيم الأولية التي يمكن تحليلها بسهولة بالأدوات التي تستخدم رسومات قريبة لتصور مهندسي البرمجيات.

لقد قمنا بتطوير مقاربتنا الى محرر رسومات بيانية موافق للنموذج المقترح باستخدام الأداة GMF (نطاق تصميم الرسومات).

Glossaire

ADL	*Architecture Description Languages*
IASA	*Integrated Approach for Software Architecture*
XML	*Extensible Markup Language*
EDA	*Event Driven Architectures*
IDE	*Integrated Development Environment*
GMF	*Graphical Modeling Framework*
EMF	*Eclipse Modeling Framework*
GEF	*Graphical Editing Framework*

Table des matières

Liste des tableaux

Liste des figures

Introduction générale

Les architectures logicielles modélisent un système en termes de composants représentants les fonctionnalités de ce système et des connecteurs décrivant les interactions entre ces composants. Actuellement plusieurs langages (dits ADL : Architecture Description Languages) sont proposés pour aider à la spécification, l'analyse et à la vérification des architectures logicielles des systèmes à base de composants. Ils ont tous pour objectif de fournir des notations expressives afin de représenter la structure conceptuelle d'un système.

Le caractère dynamique dans les architectures logicielles introduit des difficultés supplémentaires concernant la description. En effet, Dans le cas d'une architecture statique, la description consiste à "simplement" énumérer les différents composants et connexions qui la constituent. Cette approche est inadaptée pour la description des architectures dynamiques dont la configuration, par définition, change pendant l'exécution et dont les composants et les connexions peuvent être introduits et supprimés tout au long du cycle de vie de l'application.

Plusieurs chercheurs se sont concentrés sur le développement d'approches et de formalismes fournissant des systèmes d'architecture dynamique. Parmi ces approches on cite l'approche IASA (integrated Approch for Software Archietcture) et son ADL X3ADL qui prend en charge le processus de cycle de vie complet des systèmes de logiciels à base de composants et gère l'évolution architecture globale ainsi que son comportement.

Problématique

Les architectures à base de composants se sont avérés être très adapté à l'auto-adaptation en particulier avec leurs capacités de reconfiguration dynamique, la majorités des solutions existantes pour le support de ce dynamisme comptent souvent sur le faible niveau, impératif, des langages non formelles ou ne proposent pas des mécanismes pour le support de l'adaptation comportementale. Souvent les ADLs dynamique proposent une notation spécifique, très formelle, ce qui nécessite un effort de plus pour l'architecte non seulement pour comprendre le langage de spécification mais même

pour pouvoir spécifier l'architecture de son système à concevoir. De plus, l'interopérabilité de ces langages formels avec une autre description technique des architectures est limitée.

Objectif

Notre objectif dans ce projet est de décrire et de concevoir, en premier lieu un méta modèle décrivant les concepts relatifs à la spécification de l'évolution (dynamisme) des architectures logicielles selon l'approche IASA (integrated Approch for Software Archietcture).Le méta-modèle proposé sera basé sur des concepts primitifs qui peuvent facilement être analysées avec des outils ou être utilisés pour représenter une représentation actuelle de l'architecture du système en utilisant des notations graphiques proches de la perception intuitive des architectes logiciels.

Notre approche nommée ***DynamiqueIASA*** se base principalement sur l'architecture orientée événement qui semble d'être bien adaptée à notre problématique. Le protocole de gestion dynamique de l'architecture dans l'approche IASA va être défini par l'ensemble des événements impliquant une adaptation dynamique qui peut être spécifié d'une manière graphique et la transformation de cet ensemble en un ADL par l'extension de X3ADL (ADL de l'approche IASA).

Sur la base de ce méta modèle proposé on va en second lieu concevoir un Editeur graphique d'architecture logicielle permettant d'offrir une aide à la représentation graphique de cette dernière.

Organisation du mémoire : Notre mémoire est organisé en quatre chapitres :

Le premier chapitre : ***Etat de l'art sur l'architecture logicielle*** : nous présenterons dans ce chapitre les éléments et les concepts de base des architectures logicielles notamment les architectures logicielles dynamiques.

Le deuxième chapitre ***Etat de l'art sur les ADLs*** : ce chapitre est consacré à une étude détaillée sur les langages de description de l'architecture dynamique.

Le troisième chapitre ***Spécification du méta-model de l'architecture Dynamique***-IASA : nous présenterons les éléments essentiels rentrant dans la spécification du méta-model de notre approche.

Le quatrième chapitre ***L'implémentation de l'éditeur graphique*** : nous présenterons les éléments de réalisation et de test de l'IDE(Integrated Development Environment) développé pour notre approche.

Nous terminons le mémoire par une conclusion générale et des perspectives.

Chapitre 1 :

Etat de l'art sur l'architecture logicielle

1. Introduction

Actuellement un grand intérêt est porté au domaine des architectures logicielles(1).

Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement de tels systèmes. En effet, l'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de composants logiciels. Nous présenterons dans ce qui suit, nous présentons une introduction à l'architecture logicielle, les principaux concepts et terminologies, ainsi qu'un aperçu des principaux avantages de son utilisation dans le développement logiciel. Enfin, nous nous penchons sur l'aspect des architectures logicielles dynamiques.

2. Historique

Le génie logiciel vise à définir des méthodes et proposer des outils de production et de maintenance du logiciel. Il est défini comme l'ensemble des activités de conception et de mise en œuvre de procédures tendant à rationaliser le développement du logiciel et son suivi. Cette discipline est née suite à la crise qu'a connue le logiciel dans les années 70 (2). Son objectif principal est de définir des approches de conception et des paradigmes de programmation permettant la maîtrise de la complexité croissante des systèmes logiciels et la proposition de métriques d'estimation de l'effort requis pour aboutir à un logiciel fiable. La décomposition ; à travers la définition de la structure des systèmes logiciels, et la réutilisation; à travers la définition de l'unité de réutilisation, sont au centre des concepts clé proposés par le génie logiciel pour justement maîtriser la complexité et réduire l'effort et le coût de conception d'un logiciel.

Les concepts et paradigmes clé ayant contribué à l'évolution de la discipline du génie logicielle et la naissance de la branche architectures logicielles sont les suivants:

Le concept de structuration : Djikstra (2) a été un des pionniers à proposer une structuration impérative du système à concevoir avant de se lancer dans l'écriture de son code. Il a mis en évidence le besoin de la notion d'abstraction dans la conception de systèmes de taille réelle.

La modularité : introduite par Parnas (3)au début des années 70, elle a constitué un concept clé pour la structuration d'une application puisqu'une importance majeure est consacrée aux décisions de conception ; qui doivent se faire avant d'initier les travaux

de mise en œuvre. La modularité a apporté une nette amélioration quant à la souplesse et le contrôle conceptuel du logiciel tout en réduisant le temps de développement des systèmes.

Le paradigme objet : le concept d'encapsulation en particulier a constitué une contribution considérable dans la discipline du génie logiciel en améliorant l'analyse, la conception et le développement des systèmes logiciels. Car une application n'est plus maintenant vue comme une suite d'instructions mais plutôt comme une collection d'objets encapsulant données et comportement et coopérant afin de réaliser l'objectif global de l'application. Néanmoins, la structure de l'application reste toujours peu visible du fait que les couplages entre objets ne sont pas explicites, mais plutôt enfouis dans l'implémentation de ceux-ci sous forme d'instructions d'invocation des méthodes des objets. Ce qui constitue un handicap pour la réutilisation de l'objet dans d'autres contextes, d'une part. Puisque cela nécessite une remise en cause de l'implémentation pour corriger ses liens avec l'environnement. D'autre part, cette façon de faire représente une contrainte pour la conception des applications actuelles qui sont de plus en plus concurrentes, distribuées, et pouvant être exploitables sur des plateformes hétérogènes grâce à l'ubiquité numérique(les réseaux domotiques, réseaux ad-hoc, réseaux de capteurs, etc.)

A la mise en évidence des lacunes du paradigme objet dans le domaine de la réutilisation vers la fin des années 90 (voir figure1), la notion de **composant** logiciel a été proposée comme unité alternative de réutilisation. Le découplage entre les dimensions calcul (implémentation) et interaction (coordination) constitue l'apport principal du paradigme composant par rapport au paradigme objet. Il est mis en œuvre par la spécification, dans l'interface du composant, non seulement des services fournis mais aussi des services requis de l'environnement. La description de ces derniers dans l'interface met en évidence une description claire des contraintes de réutilisation d'un composant dans un contexte quelconque. Alors que le découplage entre l'implémentation et l'interface met en avant la description de l'assemblage (interactions) des composants. Les concepts d'interface de composant et d'assemblage constituent les éléments de base de définition de l'architecture logicielle d'une application. Le principe de base de l'architecture logicielle est la séparation entre l'aspect fonctionnel des composants et les interactions qu'ils entreprennent avec les autres composants et ce pour une meilleure maîtrise et une bonne gestion de la complexité des systèmes logiciels.

Malgré que cette définition mette à l'écart d'autres aspects structurels et conceptuels des systèmes, elle met à pied d'égalité le calcul et l'interaction (4) .

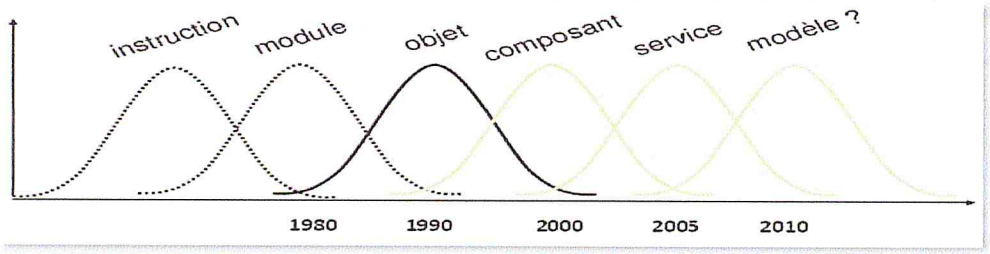


Figure 1 : Historique de la granularité des applications (4).

3. Définition

Dans le monde des sciences exactes, les définitions sont dotées d'une précision irréprochable. Par contre, dans le monde du génie logiciel d'une manière générale, les définitions et les concepts ne sont pas aussi précis. S. THOMASON (5) a affirmé : "vous *pouvez trouver autant de définitions d'un concept qu'il y a de lecteurs d'un* article" (6).

Mais d'une manière générale on peut considérer l'architecture logicielle se définit comme une spécification abstraite d'un système en termes de composants logiciels ou modules qui le constituent, des interactions entre ces composants (connecteurs) et d'un ensemble de règles qui gouvernent cette interaction ((7), (8), (9), (10) , (11)). Les composants encapsulent typiquement l'information ou la fonctionnalité. Tandis que les connecteurs assurent la communication entre les composants. Cette architecture possède, généralement, un ensemble de propriétés d'ordre topologique ou fonctionnelle qu'elle doit respecter tout au long de son évolution (6).

4. Les concepts de base de l'architecture logicielle

Le composant et le connecteur représentent les concepts fondamentaux sur lesquels, repose la spécification d'architecture logicielle. Les composants et les connecteurs sont décrits par les aspects suivants: l'interface, le type, la sémantique ou comportement, les contraintes d'exploitation et les propriétés non fonctionnelles (12).

4.1. Le concept de composant

La notion de composant logiciel est introduite comme une suite à la notion d'objet. Il offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires en favorisant la réutilisation de ces briques (13).

La notion de composant a pris de nombreuses formes dans les différentes approches de l'architecture logicielle (14). La définition suivante, globalement acceptée, illustre bien ces propriétés : "Un composant est une unité de calcul ou de stockage. Il peut être primitif ou composé. On parle dans ce dernier cas de composite. Sa taille peut aller de la fonction mathématique à une application complète. Deux parties définissent un composant. Une première partie, dite externe, comprend la description des interfaces fournies et requises par le composant. Elle définit les interactions du composant avec son environnement.
La seconde partie correspond à son contenu et permet la description du fonctionnement interne du composant" (15). Les caractéristiques globales d'un composant définies par Medvidovic et Taylor (16)sont les suivantes :

4.1.1. L'interface

L'interface d'un composant est la description de l'ensemble des services offerts et requis par le composant sous la forme de signature de méthodes, de type d'objets envoyés et retournés, d'exceptions et de contexte d'exécution. L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur (17) .

4.1.2. Le type

Le type d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet. Ainsi, un type de composant permet la réutilisation d'instances de même fonctionnalité soit dans une même architecture, soit dans des architectures différentes. En fournissant un moyen de décrire, de manière explicite, les propriétés communes à un ensemble d'instances d'un même composant, la notion de type de composant introduit un classificateur qui favorise la compréhension d'une architecture et de sa conception (17).

4.1.3. La sémantique

La sémantique du composant est exprimée en partie par son interface. Cependant, l'interface, telle qu'elle est décrite ci-dessus, ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements. La sémantique d'un composant s'apparente à la notion de type dans le modèle orienté objet (17).

4.1.4. Les contraints

Les contraintes définissent les limites d'utilisation d'un composant et ses dépendances intra composants. Une contrainte est une propriété devant être obligatoirement vérifiée sur un système ou une de ces parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Elles permettent ainsi de décrire de manière explicite les dépendances des parties internes d'un composant comme la spécification de la synchronisation entre composants d'une même application (dépendance intra composant).

4.1.5. Les contraints non fonctionnelles

La propriété non fonctionnelle d'un composant représente toute contrainte particulière liée à l'environnement d'utilisation du composant. Les propriétés non fonctionnelles sont nécessaires pour permettre la simulation du comportement des composants, leur analyse, leur traçabilité depuis leur conception jusqu'à leur implémentation et l'aide dans la gestion de projet (par exemple, en définissant des conditions de performance rigoureuses, le développement d'un composant peut devoir être assigné au meilleur ingénieur) (18).

4.2. Le concept de connecteur

Un connecteur est un bloc de construction utilisé pour exprimer les interactions entre composants ainsi que les règles qui gouvernent cette interaction. Les connecteurs sont des entités architecturales qui relient des ensembles de composants et agissent en tant que médiateurs entre eux. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure et l'émission d'évènements.

Les connecteurs peuvent également représenter des interactions complexes, comme un protocole client/serveur ou un lien SQL entre une base de données et une application.
Contrairement aux composants, les connecteurs peuvent ne pas correspondre à des unités de compilation (18)

Tout comme le composant, le connecteur est un couple spécification-code : la spécification décrit les rôles des participants à une interaction ; le code correspond à l'implantation du connecteur. Cependant, la différence avec le composant est que le connecteur ne correspond pas à une unique, mais éventuellement à plusieurs unités de programmation.
Six caractéristiques importantes sont à prendre en compte pour spécifier de manière exhaustive un connecteur. Ces caractéristiques sont les suivantes :

4.2.1. L'interface d'un connecteur

L'interface d'un connecteur, appelée aussi rôles dans certains langages de description d'architecture (19).
Définit les points de connexions entre connecteurs et composants. Elles servent à déclarer les participants à l'interaction décrite par le connecteur.
Comme celles des composants. Néanmoins, à la différence des composants, les interfaces ne décrivent pas de services fonctionnels mais des mécanismes de connexion.Elles décrivent également le rôle de chacun des composants impliqués (20).

4.2.2. Le type d'un connecteur

Le type d'un connecteur correspond à sa définition abstraite qui reprend les mécanismes de communication entre composants. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles (20).

4.2.3. La sémantique des connecteurs

Comme pour les composants, la sémantique des connecteurs est définie par un modèle de haut niveau spécifiant le comportement du connecteur. A l'opposé de la sémantique du composant qui doit exprimer les fonctionnalités déduites des buts ou des besoins de l'application, la sémantique du connecteur doit spécifier le protocole d'interaction. De plus, celui-ci doit pouvoir être modélisé et raffiné lors du passage d'un niveau de description abstraite à un niveau d'implantation (20).

4.2.4. Les contraintes

Les contraintes permettent de définir les limites d'utilisation d'un connecteur, c'est-à dire les limites d'utilisation du protocole de communication associé. Une contrainte est une propriété devant être vérifiée sur un système ou sur l'une de ses parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Par exemple, le nombre maximum de composants interconnectés à travers le connecteur peut être fixé et correspond alors à une contrainte (20).

4.2.5. Les propriétés non fonctionnelles

Les propriétés non fonctionnelles d'un connecteur concernent tout ce qui ne découle pas directement de la sémantique du connecteur. Elles spécifient des besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur. Par exemple, elles peuvent concerner la performance ou la sécurité. La spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement à l'exécution, l'analyse, la définition de contraintes et la sélection des connecteurs.

4.3. La configuration d'architecture

Une configuration définit la structure et le comportement d'une application formée de composants et de connecteurs. Une composition de composants, appelée dans certains contextes composite, est une configuration. La configuration structurelle de l'application correspond à un graphe connexe des composants et des connecteurs formant l'application.

Elle détermine les composants et les connecteurs appropriés à l'application et vérifie la correspondance entre les interfaces des composants et des connecteurs. La configuration comportementale, quant à elle, modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles comme les propriétés spatio-temporelles ou la qualité de service. Elle définit également le schéma d'instanciation des composants au moment de l'initialisation de l'application, ainsi que le placement des composants sur les sites au moment du démarrage du système et leur évolution pendant la vie de l'application. (21)

Le rôle clé d'une configuration est de faciliter la communication entre les différents intervenants dans le développement d'un système. En effet, en faisant abstraction des détails des composants et des connecteurs, les configurations offrent une vision du

système à un haut niveau d'abstraction qui peut être potentiellement comprise par des personnes avec différents niveaux d'expertise et de connaissance techniques (22).

4.4. Style architectural

Un style architectural ((23); (24)) caractérise une famille de systèmes qui partagent des propriétés structurelles et des éléments de sémantique.

Plus précisément, un style architectural permet de décrire un vocabulaire commun en définissant un ensemble de types de composants et de connecteurs, des contraintes de configuration déterminant les compositions autorisées (propriétés et contraintes partagées par toutes les configurations appartenant à ce style) d'éléments architecturaux qui sont des instances des types préalablement définis et un ensemble de propriétés fournissant des informations de sémantique et de comportement. Un style architectural peut être défini comme un type que l'architecture doit avoir pendant l'exécution (25). Les éléments du système ne peuvent interagir qu'à travers les liens spécifiés par le style. Les styles architecturaux permettent donc de décrire des patrons d'architecture pour la structuration des systèmes par analogie avec les patrons de conception (26)pour la programmation orientée objet.

L'utilisation des styles architecturaux a les avantages suivants :

- Au niveau modélisation, l'utilisation des styles architecturaux favorise la réutilisation de la conception. En effet, des applications modélisées selon un style et des propriétés architecturales bien définies peuvent être encore appliquées à de nouvelles applications.
- Au niveau conception, elle facilite la compréhension de l'organisation de l'architecture de l'application. Par exemple, concevoir une application selon le style Client/Serveur peut donner une idée claire sur le type de composants utilisés et les interactions entre les différents composants.
- Finalement, l'utilisation des styles architecturaux peut contribuer considérablement dans la réutilisation significative du code dans la phase implémentation.

Par ailleurs, Les systèmes construits selon un même style architectural sont plus compatibles que ceux qui mélangent plusieurs styles. Car cela facilite considérablement l'interopérabilité et la réutilisation de parties de la même famille de systèmes (4).

5. Avantages des architectures logicielles

La description de l'architecture logicielle s'impose de plus en plus comme une étape indispensable du développement des systèmes logiciels en permettant au concepteur de raisonner sur les propriétés fonctionnelles et non fonctionnelles du système à un haut niveau d'abstraction. Il est bien admis aujourd'hui qu'une bonne architecture peut amener à un produit qui répond aux besoins des utilisateurs et qui peut être modifié facilement et qu'une mauvaise architecture peut avoir des conséquences désastreuses sur le système (18).

D'autres avantages ont été reconnus pour les architectures logicielles (27) :

- L'architecture donne une représentation d'un système à haut niveau d'abstraction, offrant ainsi une compréhension meilleure. Cette vue du système met en valeur la plupart des décisions de conception ainsi que leurs conséquences.
- Identification plus facile des composants réutilisables d'un système.
- L'architecture offre une vue précise des dépendances entre les composants. Cette vue nécessaire pour connaitre les impacts de la modification d'un composant sur les autre composants du système et donc les conséquences des différentes évolutions.
- La vue abstraite fournie par l'architecture permet de connaitre différentes caractéristiques telles que la consistance du système, son respect du style architectural ou encoure d'autre attributs de qualité.
- En se basant sur les dépendances entre les composants, l'architecture garantit une gestion plus précise des couts et des risques de modification. Elle permet également une évaluation des qualités du système dans son ensemble, mais aussi des qualités de chaque composant.

L'approche architecture logicielle tente de donner des réponses efficaces à un bon nombre de problèmes usuels et de grande importance dans le monde du logiciel, tels que la réutilisation de composants et de connecteurs, l'interopérabilité, la mise à jour de logiciel, l'isolation des technologies complexes et la prise en charges des propriétés non fonctionnelles durant tout le processus de conception d'un logiciel (12).

6. L'architecture logicielle dynamique

"Nothing is as constant as the occurrence of changes" d'après Chris Salzman et al (28). Les systèmes logiciels aussi n'échappent à cette règle car ils sont toujours amenés à évoluer pour diverses raisons : évolution de l'architecture, ajout de fonctionnalités, intégration de nouvelles technologies de développement ou de communication, modification de l'environnement d'exécution, personnalisation des services, etc.

Les architectures dynamiques correspondent à des applications dont les composants sont créés, interconnectés, et supprimés pendant l'exécution. Le caractère dynamique des architectures implique des difficultés supplémentaires pour la description. Depuis quelques années, la dynamique des architectures est devenue une activité qui commence à prendre de l'ampleur puisqu'elle donne la possibilité aux administrateurs d'agir sur l'architecture de l'application (17).

Les modifications des architectures logicielles de manière générale peuvent se produire soit au moment de :

- la conception,
- au moment de la pré-exécution
- ou au moment de l'exécution.

Les architectures dynamiques changent de structures pendant l'exécution du système et donnent une description à propos de ce changement.

La dynamique de l'architecture consiste donc à adapter une architecture pour prendre en compte de nouveaux besoins. Ceci permet de faire passer une architecture d'une configuration C à une autre configuration C'.une opération peut être initiée soit par l'administrateur soit par l'application elle-même.

6.1. Les avantages par rapport aux architectures statiques

- **la correction :** Si l'application en cours d'exécution ne se comporte pas correctement comme prévu l'architecture logicielle donne la permission de identifier le composant de l'application qui pose problème et le remplacer par une nouvelle version supposée correcte, cette nouvelle version fournit la même fonctionnalité que l'ancienne, elle se contente simplement de corriger ses défauts.

- ***L'adaptabilité*** **:** Même si l'application s'exécute correctement, par fois l'environnement d'exécution change (le système d'exploitation, les composants matériels ou d'autres applications ou données dont elle dépend), L'architecture doit donc s'adapter et évoluer soit en **ajoutant** des nouveaux composants/connexions ou en **remplaçant** des composants/connexions par d'autres.
- **Evolution :** Au moment du développement de l'application, certaines fonctionnalités ne sont pas prises en compte. Avec l'évolution des besoins de l'utilisateur, l'application doit être étendue avec de nouvelles fonctionnalités.
  Cette extension peut être réalisée en ajoutant un ou plusieurs composants/connexions pour assurer les nouvelles fonctionnalités ou gardant la même architecture de l'application et étendre simplement les composants existants.
- **La perfection :** Améliorer les performances du système par exemple si un composant qui reçoit beaucoup de requêtes et qui n'arrive pas à les satisfaire afin éviter la dégradation des performances de l'application on diminue la charge de ce composant en installant un autre qui lui partage sa tâche.

6.2. Types d'architectures dynamiques

Pour soutenir le développement des architectures dynamiques, plusieurs chercheurs se sont concentrés sur le développement d'approches et de formalismes fournissant des systèmes d'architecture dynamique.

Suite à ces recherches, différents types de dynamiques sont apparus. Parmi ces types nous trouvons (6) :

- ***Dynamique d'implémentation*** : ce type de dynamique permet de modifier la mise en œuvre d'un composant, c'est-à-dire la manière avec laquelle un composant a été développé (le code du composant) sans changer ni les interfaces ni les connexions. Ce type de changement permet de faire évoluer un composant d'une version à une autre (voir figure 2).

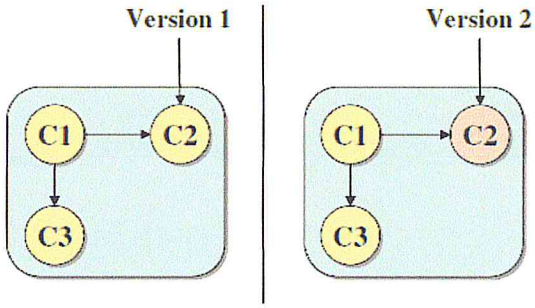


Figure 2 : ***Exemple de changement d'implémentation*** **(6)**

- **Dynamique d'interfaces** : comme nous l'avons détaillé précédemment, un composant fournit ses services à travers des interfaces. La dynamique d'interfaces consiste modifier les services fournis par un composant au biais de ses interfaces. Ceci, se traduit soit par la modification de l'ensemble des services fournis soit par la modification de la signature d'un service (voir figure 3).

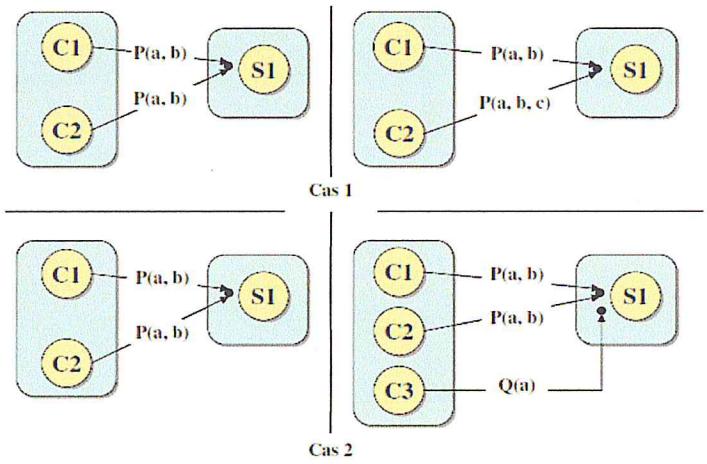


Figure 3 : ***Exemple de changement d'interface*** **(6)**

- **Dynamique de géométrie** : elle est appelée aussi dynamique de localisation ou encore de migration. La dynamique de géométrie modifie la distribution géographique d'une application. En effet, ce type de dynamique 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 4, le composant C4 est déplacé du site 2 vers le site 3.

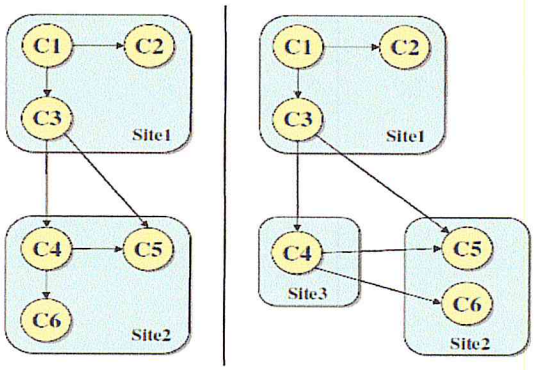


Figure 4 : Exemple de changement géométrique (6)

- **Dynamique de structure** : elle modifie la topologie de l'application. C'est-à-dire dans ce type de dynamique on s'intéresse uniquement au changement de la structure de l'application en termes de composants et de connexions. Cette dynamique se réalise à l'aide de quatre opérations de base (29): ajout d'un composant, suppression d'un composant, ajout d'une connexion et suppression d'une connexion. Comme le montre la figure 5, La connexion T1 est redirigée vers le composant S1.

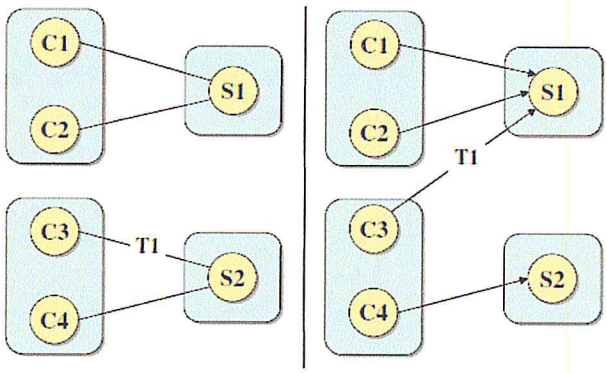


Figure 5 : Exemple de changement de structure (6)

7. Conclusion

Nous avons présenté dans ce chapitre le contexte global dans lequel s'intègre ce travail de recherche: le domaine des architectures logicielles. Pour cerner la notion d'architecture logicielle nous avons présenté les principaux concepts et terminologies de l'architecture logicielle notamment les architectures logicielles dynamiques. L'étude de cette terminologie nous a permis de mettre en lumière les différentes définitions et notions que nous jugeons nécessaires pour aborder notre problématique.

Chapitre 2 :

Etat de l'art sur les ADLs

1. Introduction

Pour décrire les architectures logicielles, plusieurs langages de description (ADL: Architecture Description Languages) ont été proposés. Les ADLs offrent un niveau d'abstraction élevé pour la spécification et le développement des systèmes logiciels, qui se veut indépendant des langages de programmation et de leurs plates-formes d'exécution(18).

Une étude des ADLs dynamique fera l'objet de ce chapitre.

2. Les langages de description d'architecture

Les langages de description d'architecture (ADLs) sont des langages formels qui peuvent être utilisés pour représenter l'architecture d'un système logiciel. Comme l'architecture est devenue un thème important dans le développement de systèmes logiciels, les méthodes pour spécifier de façon non ambiguë une architecture, deviennent indispensables (31).

Les modifications dans une architecture peuvent être planifiées ou non planifiées. Elles peuvent également se produire avant ou pendant la phase d'exécution. Les ADLs doivent supporter de tels changements grâce à des mécanismes opérationnels (l'instanciation, l'héritage et le sous-typage, la composition, la généricité, le raffinement et la traçabilité). En plus, les architectures sont prévues pour fournir aux développeurs des abstractions dont ils ont besoin pour faire face à la complexité et à la taille des logiciels. C'est pourquoi les ADLs doivent fournir des outils de spécification et de développement pour pouvoir prendre en compte des systèmes à grand échelle susceptibles d'évoluer. Aussi, pour améliorer l'évolutivité et le passage à l'échelle et pour augmenter la réutilisabilité et la compréhensibilité des architectures. Les mécanismes spéciaux doivent être pris en compte par les ADLs.

La totalité des ADLs intègre la notion de composants qui sont définis selon l'approche boîte noire, c.à.d. leur structure interne et leur implémentation sont cachées et seule leur interface d'interaction est visible aux autres composants.

2.1. Darwin

Le langage Darwin a été développé par "Distributed Software Engineering Group" de l'Imperial College. Darwin propose un modèle de composants pour la construction d'applications distribuées. Ce langage se centre sur la description de la configuration et sur l'expression du comportement d'une application plutôt que sur la description structurelle de l'architecture d'un système. La particularité de Darwin est de permettre la spécification d'une partie de la dynamique d'une application en terme de schéma de création de composants logiciels avant ou pendant son exécution (6).

Le langage de description Darwin, se base sur une description hiérarchique d'instances de composants interconnectés. Chaque niveau de la hiérarchie représente un composant dit de type composite, et les feuilles de la hiérarchie représentent les instances des composants dits de type primitif. Chaque type de composant est décrit via son interface qui est constituée d'une collection de services qu'il procure (i.e. déclarés par le composant), et de services qu'il attend (i.e. qui se produisent dans l'environnement) (32) .

Les connecteurs quant à eux ne sont pas considérés comme des entités de première classe. Chaque interaction est représentée par un lien entre un service requis et un service fourni de composants différents. Les configurations (composants composites) sont décrites par les déclarations d'instanciation des composants et par les liaisons entre les services requis et les services fournis de ces instances de composants. Un composant composite peut être décrit par une configuration interne (18).

2.1.1 Aspect dynamique

Darwin permet de spécifier des composants dynamiques. Il permet de créer des composants en cours d'exécution du système. Le système n'est plus considéré comme un ensemble de composants figés lors de la phase de conception, mais il est possible de spécifier les instants et les emplacements de création dynamique de composants (6). Darwin fournit deux mécanismes pour décrire la création dynamique : l'instanciation paresseuse et l'instanciation dynamique (32).

- *l'instanciation paresseuse* permet de retarder l'instanciation de certains composants. Ainsi, un composant offrant un service et déclaré dynamiquement en utilisant cette approche, ne sera instancié que lorsqu'un utilisateur de ce service tente d'y accéder.

- *l'instanciation dynamique* permet la définition de structures qui peuvent évoluer par réplication de composants. Chaque événement introduit une nouvelle instance d'un composant spécifié. Les connexions des composants créés selon cette approche sont définies de manière générique pour chaque composant répliqué.

Les deux approches définies par le langage Darwin pour la description des architectures dynamiques correspondent au traitement de plusieurs cas classiques de reconfiguration. Elles présentent, néanmoins, plusieurs limitations.

Ainsi, la première approche nécessite d'énumérer et de déclarer tous les composants potentiels de l'architecture. Elle est inadéquate si, par exemple, l'architecture possède un nombre non borné de composants. Cette approche n'adresse réellement l'évolution dynamique de l'architecture que du point de vue de l'activation des composants.

La deuxième approche adresse l'évolution dynamique par la création de nouvelles instances de composants, mais le fait que les instances créées par réplication soient anonymes et soient, donc, toutes connectées de la même manière constitue une limitation très contraignante. De plus, comme le langage Darwin interdit de connecter plusieurs services offerts à un seul service requis, le raccordement des services offerts par les composants créés par réplication ne peut être décrit par Darwin. Un autre aspect non considéré par Darwin est la spécification des suppressions de composants ou de raccordements.

2.2. Wright

Wright est un langage de description d'architecture créé par ALLEN et GARLAN il fournit des bases formelles pour spécifier les interactions entre les composants (via des connecteurs). Wright est un langage de description orienté plus vers la vérification des protocoles entre les composants, que vers la correction fonctionnelle de l'architecture globale (6).

Il permet de décrire formellement une architecture à l'aide de l'algèbre de processus CSP (32). Comme les autres ADLs, Wright reprend les trois concepts de l'architecture logicielle à savoir le composant, le connecteur et la configuration.

- Le composant dans l'ADL Wright est une unité abstraite et localisée et indépendante. Un composant est décrit par une interface et une partie calcul. Une interface est composée de ports, et chaque port représente une interaction

dans laquelle le composant peut participer. À chaque port est associée une description formelle par le langage CSP (Communicating Sequential Processes) spécifiant son comportement par rapport à l'environnement. La partie calcule consiste à décrire le comportement du composant en indiquant comment celui-ci utilise les ports. Ainsi, les ports, qui sont décrits indépendamment dans l'interface, sont utilisés pour décrire le comportement du composant dans le calcul.

- Le connecteur en général est un bloc de construction utilisé pour exprimer les interactions entre composants ainsi que les règles qui gouvernent cette interaction. Le connecteur dans Wright représente une interaction entre une collection de composants. Le connecteur contient deux parties importantes qui sont un ensemble des rôles et la glue. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. La glu d'un connecteur décrit comment les participants travaillent ensemble pour créer une interaction.
- La configuration est une collection d'instances de composants liés par des instances de connecteurs. La description d'une configuration est composée de trois parties qui sont la déclaration des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composant et de connecteurs, les descriptions des liens entre les instances de composant par les connecteurs.

2.2.1. Aspect dynamique

La première version de l'ADL Wright a été proposée par Allen et Garlan en 1997, cette version traite uniquement les architectures statiques Wright propose une version qui traite la dynamique des architectures, cette version est appelée Dynamic Wright, il a introduit des événements de contrôle permettant de spécifier les conditions sous lesquelles les transformations de l'architecture sont autorisées. Ces événements de contrôle induisent l'exécution d'une séquence d'actions élémentaires de reconfiguration comprenant l'introduction (l'action new) et la suppression de composants et de connecteurs (l'action del) et l'introduction et la suppression des liens (les actions attach et detach) Les programmes de reconfiguration (appelés Configuror) spécifient les politiques qui caractérisent l'évolution dynamique de l'architecture globale.

Le formalisme de description pour les architectures dynamiques défini par l'extension Dynamic Wright est plus expressif que celui spécifié par Darwin. Cependant, sa puissance d'expression reste limitée par le fait qu'il implique

l'énumération de toutes les configurations possibles de l'architecture. Dans la mesure où la taille des architectures dynamiques (en nombre de composants et de connecteurs) est généralement non bornée et que, par conséquent, le nombre de configurations possible est infini, cette limite réduit le champ d'application de ce formalisme aux applications de petite taille avec une faible composante dynamique.

2.3. Rapide

Rapide est un langage de description d'architecture dont le but principal est de vérifier, par simulation, la validité d'une architecture logicielle. Il fut proposé à l'origine au projet ARPA (Advanced Research Projects Agency) en 1990 par David Luckham à l'université de Stanford aux Etats Unis. Il est basé sur des évènements concurrents. Il est conçu essentiellement pour prototyper des architectures dans des systèmes distribués.

Le langage Rapide ne fournit pas d'outils de génération de code, il permet cependant la description du comportement dynamique et il offre des méthodes de validation (simulation) des architectures qu'il décrit. Il permet aussi de vérifier certaines propriétés comme l'inter-blocage lors de l'exécution de l'application (6). Les concepts de base de Rapide sont le composant, l'événement et l'architecture (20).

- *L'événement* est une information transmise. Il permet de construire des expressions appelées event patterns qui caractérisent les événements circulant entre composants. La construction de ces expressions se fait avec l'utilisation d'opérateurs qui définissent les dépendances entre événements. Ainsi, l'événement correspond à une information permettant de spécifier le comportement d'une application.
- Le composant est défini par une interface. Cette dernière est constituée d'un ensemble de services fournis et d'un ensemble de services requis. Les services sont de trois types :
  - Les services Provides: fournis par le composant appelés de manière synchrone par d'autres composants ;
  - Les services Requires: demandés par le composant appelés de manière synchrone ;
  - Les Actions: qui correspondent à des appels asynchrones entre composants. Deux types d'actions existent : les actions in et out qui sont des événements acceptés et envoyés par un composant.

L'interface contient également une section de description du comportement (clause

behavior) du composant. Cette dernière correspond au fonctionnement observable du

composant comme, par exemple, l'ordonnancement des événements ou des appels aux

services. Ainsi, l'environnement Rapide peut simuler le fonctionnement de l'application.

De plus, Rapide permet également de spécifier des contraintes (clause constraint) qui sont des patrons d'événements qui doivent ou non se produire pour un composant lors de son exécution. Par exemple, une contrainte peut fixer un ordre obligatoire pour une séquence d'événements d'un composant. En général, ces contraintes permettent de spécifier des restrictions sur le comportement des composants.

- *L'architecture* contient la déclaration des instances de composants et les règles de connexions entre ces instances. Toutes les instances sont déclarées sous forme de variables.La règle d'interconnexion est composée de deux parties. La première est la partie gauche qui contient une expression d'événements qui doit être vérifiée, la seconde est la partie droite qui contient également une expression d'événements qui doivent être déclenchés après la vérification de l'expression de la partie de gauche. Les contraintes (clause constraint) peuvent être utilisées pour décrire l'architecture. Elles permettent de restreindre le comportement de l'architecture en définissant des patrons d'événements à appliquer pour certaines connexions entre composants

2.3.1. Aspect dynamique

Rapide est capable de modéliser l'architecture des systèmes dynamiques dans lesquels le nombre de composants peut varier quand le système est exécuté. Pour décrire la dynamique,

Rapide introduit deux types de variables particulières : les placeholder et les iterator auxquelles sont associées des règles de création. Le nom d'une variable placeholder commence toujours par " ?". Cette variable désigne un objet qui est susceptible d'être présent. Le nom d'une variable iterator commence toujours par " !". Cette variable

désigne une conjonction d'instances d'un certain type. Les règles de création définissent quand les composants doivent être créés ou détruits lors de l'exécution.

Rapide supporte uniquement des manipulations dynamiques où tous les changements d'exécutions doivent être connus a priori. Il permet donc de créer des composants dynamiques et possède des mécanismes permettant leur gestion. Cependant, Rapide ne supporte aucun opérateur de création, suppression, migration de composants ou modification d'interconnexions. En plus, dans le modèle, le concept de connecteur n'apparaît que de manière implicite. Ceci limite la réutilisation (6).

2.4. C2S ADEL

C2SADEL (33) (C2 Software Architecture Description and Evolution Language) est l'un des ADL dont l'objectif est de décrire des architectures de systèmes distribués, évolutifs et dynamiques. Est un exemple de langage de description d'architecture dynamique basée sur le style C2 (34) qui organise les composants d'une architecture logicielle en couches reliées par des connecteurs. Les briques de base de C2SADEL sont aussi les composants, les connecteurs, et la topologie (configuration) (18). Les composants et les connecteurs sont vus comme des types qui peuvent être instanciés plusieurs fois au sein d'une topologie.

- Un composant dans C2SADL peut avoir un état et ses propres taches de contrôle. Il possède aussi une structure externe et interne particulières. La structure externe est constituée des deux interfaces top et bottom. Son interface top définit l'ensemble des notifications auxquelles le composant répond et l'ensemble des requêtes qu'il envoie aux composants du niveau supérieur. L'interface bottom d'un composant définit l'ensemble des notifications qu'il émet vers le bas de l'architecture et l'ensemble des requêtes auxquelles il répond. En fait, les requêtes ne peuvent être envoyées que vers le haut de l'architecture à travers l'interface top, alors que les notifications ne seront envoyées que vers le bas à travers l'interface bottom. La structure interne définit le comportement du composant suite à une invocation externe sous forme d'un message. L'interface d'un composant est constituée de l'ensemble des messages pouvant être reçus et l'ensemble de messages pouvant être émis. Chaque composant C2 est défini par un nom, les interfaces top et down, un comportement et une éventuelle implémentation. La signature de chaque élément de l'interface comporte les

rubriques suivantes : un modificateur indiquant le sens de communication (Provided / Request), un nom, un ensemble de paramètres et un éventuel résultat. Par ailleurs, chaque paramètre a un nom et un type.

- Les connecteurs sont définis aussi explicitement, leur principale tâche est de coordonner la communication entre les composants. Dans le style C2, ils jouent le rôle de médiateur d'interactions entre composants car ils contrôlent la transmission et la distribution des messages, définis par un nom et un ensemble de paramètres typés. Ils permettent de lier plusieurs composants, d'une part par la diffusion des messages (demandes de requêtes ou événements) envoyés et destinés aux composants et d'autre part le filtrage de certains messages.

  Un nombre quelconque de composants peut être attaché à un même connecteur. Ainsi, l'interface d'un connecteur est définie en fonction des interfaces des composants qui communiquent à travers elle. Le seul aspect modélisé des connecteurs est le mécanisme de filtrage des informations qui le traverse dénoté par le mot clé message_filter.
- C2 permet de définir la configuration d'une architecture en spécifiant :

➢ la structure de l'application, c.à.d. les composants utilisés et les connecteurs assurant les interactions entre ces composants,

➢ la dynamique de l'application, c.à.d. les changements de l'architecture au cours de l'exécution comme par exemple l'ajout ou la suppression de composants.

La topologie, nommée dans la spécification par Architecture, désigne la connexion entres les instances de composants

2.4.1. Aspect dynamique

C2SADEL spécifie un ensemble d'opérations pour l'insertion, la suppression, le remplacement et la re-connexion des éléments de l'architecture pendant l'exécution du système tel que : add, remove, weld et unweld. Ces évolutions sont réalisées d'une façon interactive via l'outil Archstudio1 couplé à l'outil DRADEL (Developement of Robust Architectures using a Description and Evolution Language) (35) DRADEL est un environnement pour supporter la modélisation, l'analyse et l'évolution des architectures décrites en C2SADEL.

C2SADL est certainement, parmi les ADLs présentés ici, celui qui offre la spécification de l'évolution dynamique la plus complète puisqu'il admet toutes les actions de reconfiguration de base. Cependant les contraintes introduites par le style C2

se révèlent très contraignantes. La communication entre composants ne peut être structurée qu'en couches superposées où les interactions (concernant chaque type) sont unidirectionnelles. Ceci exclut, par exemple, la description de deux composants s'envoyant mutuellement des requêtes ou des architectures comprenant des cycles. De plus, le fait que les composants soient contraints à exactement un port top et un port bottom leur interdit d'être connectés à plusieurs connecteurs ce qui, combiné au fait que les composants ne possèdent qu'un port pour chaque direction, établit une restriction supplémentaire pour la communication entre composants : par exemple, si un composant C est connecté en haut d'un composant C' et si un autre composant C" est connecté en haut du même composant C', alors C et C" sont forcément au même niveau et ne peuvent pas, par exemple, s'envoyer des requêtes ou des notifications.

2.5. ACME

The Architectural Based Language and Environment (ACME) (36) .Le projet ACME[1], commencé en 1995, a pour principal but de fournir un langage commun permettant l'échange de descriptions architecturales entre plusieurs outils de conception d'architecture. Il s'agit d'un langage de description d'architecture logicielle fournissant une base conceptuelle abstraite et suffisamment générale pour permettre la description de nouveaux outils et notations (37).

Dans ACME, la structure architecturale est décrite à l'aide de sept types de concepts : le composant, le connecteur, le système, le port, le rôle, la représentation et la carte de représentations. Le composant est une entité de calcul ou de stockage de données. Il est décrit par des interfaces composées de ports (fournis ou requis). Un composant peut être primitif ou composé. Le connecteur décrit une connexion entre les composants. Il est également décrit par des interfaces définies par un ensemble de rôles. Un système dans ACME décrit la configuration de l'architecture en termes d'instances de composants et de connecteurs reliés par les attachements et les liaisons. Un attachement relie le port d'un composant à un rôle d'un connecteur. Une liaison relie les ports d'une configuration ou d'un composant composite aux ports de ses composants internes.

[1] Cf. http://www-2.cs.cmu.edu/~acme/

ACME propose le mécanisme de composition hiérarchique pour l'évolution des composants et connecteurs types. Ce mécanisme est supporté par ACME, grâce au concept de représentation. En effet, un composant ou un connecteur peut être décrit d'un niveau général à un niveau plus détaillé. Chaque nouvelle description d'un composant ou d'un connecteur est appelée représentation. La correspondance entre un élément et ses représentations est spécifiée grâce à la carte de représentation (rep-maps). Ainsi la spécification d'un composant ou d'un connecteur peut évoluer d'un niveau de détail élevé vers d'autres niveaux plus raffinés (18).

2.5.1. Aspect dynamique

Dynamic ACME propose des extensions à ACME pour gérer l'évolution dynamique des architectures logicielles. En effet Dynamic ACME étend la syntaxe d'ACME par des constructions syntaxiques pour spécifier des évolutions d'une architecture logicielle préalablement prévues et connues (anticipées). Pour se faire, il distingue deux catégories d'éléments architecturaux : les éléments fermés qui indique que la spécification de ces éléments est complète et les éléments ouverts qui indique que la spécification de ces éléments peut évoluer. Un élément décrit en ACME est implicitement fermé et pour expliciter qu'un élément peut évoluer, le modificateur open est ainsi utilisé. Dynamic ACME associe aussi une multiplicité à un composant ou un connecteur type qui indique le minimum et le maximum de ses instances (18).

2.6. Fractal

Fractal est un modèle de composants développé par France Télécom R&D et l'INRIA. Contrairement à d'autres modèles comme les EJB ou CCM dont les composants sont plutôt de grain moyen et destinés aux applications de gestion tournées vers l'Internet, la granularité des composants Fractal est quelconque. Leurs caractéristiques font qu'ils conviennent aussi bien à des composants de bas niveau (par exemple un pool d'objets) que de haut niveau (par exemple une IHM complète). Le but de Fractal est de développer et de gérer des systèmes complexes comme les systèmes distribués. Fractal est composé de deux modèles : un modèle abstrait et un modèle d'*implantation*. Fractal fournit un langage de description d'architecture (Fractal ADL) dont la caractéristique principale est d'être extensible. La motivation pour une telle extensibilité est double. D'une part, le modèle de composants étant lui-même extensible, il est possible d'associer un nombre arbitraire de contrôleurs aux composants. Supposons, par exemple, qu'un contrôleur de journalisation (LoggerController) soit ajouté à un composant, il est nécessaire que l'ADL puisse être facilement étendu pour prendre en compte ce nouveau contrôleur. C'est-à dire pour que le déployeur d'application puisse spécifier, via l'ADL, le nom du système de journalisation ainsi que son niveau (exemple : debug, warning, error). La seconde motivation réside dans le fait qu'il existe de multiples usages qui peuvent être faits d'une définition ADL : déploiement, vérification, analyse, etc. Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage. On retrouve au niveau du modèle concret de composant Fractal les concepts de composant, connecteur et configuration (31).

- Les composants possèdent une membrane qui délimite clairement leur contenu de leur environnement extérieur afin de structurer l'application. Cette membrane dispose d'interfaces externes (resp. internes) permettant la communication des composants avec l'extérieur (resp. au sein de leur contenu). Leur contenu consiste en un ensemble fini de sous-composants. Le modèle est donc hiérarchique permet ainsi une construction récursive qui s'arrête aux composants primitifs qui sont directement programmés dans un langage de programmation donné.

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : les interfaces métier et les interfaces de contrôle (la membrane correspond à l'ensemble de ces interfaces de contrôle). L'ensemble des interfaces métier et de contrôle d'un composant définit son type. Les interfaces métier sont les points d'accès externes au composant alors que les interfaces de contrôle prennent en charge des propriétés non fonctionnelles du composant comme la gestion de son cycle de vie ou de ses liaisons. Une interface Fractal est composée d'un nom, d'une signature et d'un type. Dans la projection du modèle en Java, la signature d'une interface est une interface Java. Une interface Fractal métier est du type client ou serveur. Une interface serveur identifie les services offerts par un composant alors qu'une interface client spécifie les fonctionnalités qu'un composant requiert pour son fonctionnement.

- Fractal ne possède pas de notion de connecteur explicite avec une sémantique de processus. Cependant, il utilise la notion de liaison pour spécifier les interactions entre composants. Une liaison Fractal est définie comme un lien orienté entre une interface client et une interface serveur. Ce lien permet aux composants d'interagir. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée. La liaison est assimilable à un connecteur implicite, elle ne possède pas de comportement propre.

  La membrane possède à la fois des interfaces externes, qui sont accessibles de l'extérieur du composant, et des interfaces internes, accessibles seulement par son contenu. Dans le modèle Fractal, une interface interne ne peut exister que symétriquement à une interface externe. Ce mécanisme d'interface interne sert principalement à permettre les traversées de membranes des composites en conservant la sémantique de la liaison. Ainsi la liaison permet de définir à la fois les interactions entre deux composants mais permet aussi de définir les liens de délégation entre les interfaces d'un composite et les composants de son contenu.
- Fractal est un modèle hiérarchique, un composant peut posséder au sein de son contenu d'autres composants. Il est identifié dès lors comme un composite. Le contenu d'un composite définit une configuration pour la mise en œuvre des services définis au niveau de ses interfaces métiers. Enfin, et c'est une spécificité

du modèle Fractal, un composant peut appartenir au contenu de deux composites qui ne sont pas imbriqués l'un dans l'autre. Il est alors dit partagé.

2.7. Π-ADL

Le langage Π-ADL fournit les constructions de base pour la description de la structure et du comportement d'architectures logicielles statiques, dynamiques et mobiles. C'est un langage de spécification formelle supportant la vérification automatique de certaines propriétés. Comme la plupart des ADLs, Π-ADL offre les éléments de base pour la description d'une architecture logicielle : les composants, les connecteurs et leur schéma de composition (configurations) (4).

- Le rôle architectural d'un composant dans Π-ADL est de spécifier les éléments de calcul d'un système logiciel. Un composant est défini par un ensemble de ports externes, représentant l'interface du composant, et un comportement interne. Les ports sont décrits par des connexions entre le composant et son environnement. Des canaux de communication entre éléments architecturaux ; appelés connexions, permettent de relier les ports des composants aux rôles des connecteurs. Ces connexions représentent les points d'interaction de base. Un composant peut envoyer ou recevoir des données via ces connexions une fois établies.

  Deux types de connexions sont offerts par le langage Π -ADL, les connexions à sens unique véhiculent des données d'entrée, par contre les connexions bidirectionnelles peuvent véhiculer des données d'entrée et de sortie.
- Les connecteurs sont des composants particuliers dans le sens où ils ont des ports externes et un comportement interne. Cependant, leur rôle architectural est de relier les composants d'une architecture logicielle, ils spécifient les interactions entre composants.
- L'assemblage des instances de composants et connecteurs est réalisée à travers leurs ports respectifs. Attacher un port d'un composant à un rôle d'un connecteur nécessite l'existence d'au moins une connexion de chaque côté. Cet attachement est effectué par unification des deux connexions ou par passage de valeur (qui peut être une donnée, une connexion ou un élément architectural entier).

2.7.1. Aspect dynamique

Π-ADL propose un opérateur compose pour construire un élément architectural par assemblage d'un ensemble de composants. Cet opérateur permet par ailleurs de créer les instances de composants cités dans l'assemblage, mais qui n'existent pas encore dans la configuration actuelle. Il crée aussi les liens de ces nouvelles instances.

Π-ADL permet de vérifier que l'instance n'a pas encore été créée à l'aide de la fonction booléenne isConnectionUnified(C) qui teste la connexion de cette instance au reste de l'architecture logicielle.

Π-ADL propose aussi l'opérateur replicate qui permet de définir un composite comme une réplication infinie de lui-même.

2.8. π-Space

π-Space est un ADL pour la description des architectures dynamiques à base de composants. Il est formellement fondé sur le π-calcul pour décrire les systèmes dynamiques. Ce langage est étendu par $\sigma\pi$-Space pour la description des styles architecturaux.

Les architectures sont représentées par des configurations de composites, de composants et de connecteurs. Un composite est un élément composé permettant de définir une configuration.

Les composants et les connecteurs sont composés d'un ensemble de ports, d'un comportement et d'un ensemble d'opérations (pour les composants) (6).

2.8.1. Aspect Dynamique

π-Space permet de formaliser la dynamique d'une architecture. Lors d'une description, nous pouvons définir quels éléments sont dynamiques. Leurs noms sont alors suffixés du caractère π. Cela signifie que plusieurs occurrences du composant dynamique peuvent être créées dynamiquement. La clause whenever permet de spécifier des réactions à des événements.

De même, π-Space permet de définir des règles de dynamique permettant de décrire les changements topologiques dus à la création dynamique d'un nouvel élément. La dynamique peut être déclenchée par un composant. Pour cela, des ports spécifiques permettant d'évoquer des reconfigurations attachementEvolutionPort, ComponentEvolutionPort et EvolutionPort sont définis. Ces ports sont sollicités via le mot-clé evolvable (6).

2.9. LEDA

Comme Darwin il regard à l'architecture comme une collection des composants, les composants sont spécifiées en terme d'une interface, composition, et l'attachement. L'interface est définie en utilisant des instances des rôles qui définissent le comportement des composants avec des autres composants (40).

2.9.1. Aspect dynamique

C'est d'un ADL dynamique, il utilise le π-calcul pour spécifié les architectures logicielles dynamiques.

La spécification de comportement des composants dans LEDA est double: comportement externe et le comportement interne. Le comportement externe est spécifié par types de rôles, qui sont spécifiés séparément et instanciées dans les types de composants; comportement interne est spécifiée l'aide (en option) Spec construire en types de composants. Le rôle types et la partie de spécification des types de composants sont spécifiés formellement en tant que processus en π-calcul.

Sémantique de LEDA Comme les composants originaux de Darwin et les liaisons, la sémantique des composants et des pièces jointes dans LEDA sont formellement définie en utilisant π-calcul aussi. L'attachement est une connexion entre n'importe quelles instances des composants. L'attachement en LEDA peut être statique ou reconfigurable, l'attachement reconfigurable est utilisé pour supporter la notion de dynamisme.

2.10. Pilar

C'est un ADL qui permet la représentation des architectures logicielles hiérarchiques qui reconfigure dynamiquement. Le système dans Pilar Peut avoir deux types de composants, composant primitive et composant composite (40).

2.10.1. Aspect dynamique

Pilar a deux types primitifs : opérateur (composant) et opération (reconfiguration). Comportement en pilar est fourni par un certain nombre de règles, dispersés à travers le composant définitions dans la section contrainte définie en CCS qui contient la reconfiguration. Ils sont ensuite liés à un structure défini, où ils assurent des propriétés et de réagir à des situations.

3. Tableau de spécification dynamique des ADLs étudiés

L'ADL	Spécification dynamique	Les limites
Darwin	**-Instanciation dynamique Paresseuse:** Permet de retarder l'instanciation d'un composant. Le composant offre un service, ce composant ne sera instancié que lorsqu'un utilisateur de ce service tente d'y accéder. Réparation d'un composant possible **-instanciation dynamique direct :** duplication d'instanciation est possible	- L'utilisation de divers modes de communication (synchrone, asynchrone) n'est pas configurable, - Tous les évolutions qui doivent opérer dynamiquement doivent être connues au préalable, -Il n'est pas possible de supprimer des composants dynamiquement. - Impossibilité de décrire les propriétés non fonctionnelles.
Wright	-Introduit des **événements de contrôles** pour spécifier les conditions sous lesquelles les transformations sont autorisées. -permet la reconfiguration des composants et des connecteurs : • Introduction : **new** • Suppression : **del** • Introduction de lien : **attach** • Suppression de lien : **detach**	-L'évolution dynamique considérée est anticipée. Dans **Dynamic Wright**, aucune autre évolution ne peut être réalisée durant l'exécution du système, en dehors de celles préalablement prévues et spécifiées dans l'architecture. - Peu de moyens pour séparer les spécifications fonctionnelles et non fonctionnelles.
Rapide	-Tous les changements d'exécution doivent être connus a priori. -Introduit deux types de variable : **placeholder** et **iterator**. • Placeholder : (**?**) cette variable désigne un objet qui est susceptible d'être présent. • Iterator : (**!**) cette variable désigne une conjonction d'instances d'un certain type.	-Pas de représentation explicite de connecteur, - Ne fournit pas d'outils de génération de code, - Pas de moyens pour séparer les spécifications fonctionnelles et non fonctionnelles, - Rapide ne supporte aucun opérateur de création, suppression, migration de composants ou modification d'interconnexions.

LEDA	-utilise ***π*-calculus** : pour exprimer le dynamisme. -les attachements peuvent être reconfigurables pour supporter la notion de dynamisme.	-ne supporte pas la suppression des composants et des connecteurs.
C2SADL	-permet la création, la suppression, le remplacement, la re-connexion des éléments architecturaux. -définir deux interfaces bottom et top : **Top:** • définit l'ensemble des notifications auxquelles le composant répond et l'ensemble des requêtes qu'il envoie aux composants du niveau supérieur. • Envoyer des requêtes vers le haut **Bottom:** • Définir l'ensemble des notifications qu'il émet vers le bas, • Les requêtes qu'il répond, • les notifications ne seront envoyées vers le bas qu'à travers l'interface bottom.	-L'évolution dynamique de la configuration, se fait via une interface interactive. Ni l'origine de la création ou de la suppression de ces composants, ni les moyens de gestion de ces composants, une fois crées ne sont spécifiés. -Les primitives pour l'ajout, la suppression des composants et connecteurs ou de leur connexion ne précisent pas les impacts de ces opérations sur le reste de l'architecture.

ACME	-La **multiplicité** d'un composant ou d'un connecteur. -**Open**: qui indique que la spécification d'un élément peut évoluer.	- la spécification de l'évolution dynamique aboutissent à des spécifications très complexes, difficiles à comprendre. - Langage de modélisation qui n'offre pas de moyen de projeter la spécification d'une architecture logicielle vers un système.
***Π*-ADL**	-opérateur **compose** : • pour construire un élément architectural par assemblage d'un ensemble de composants, • créer des instances des composants et les liens entre ces nouvelles instances. -Opérateur **Replicate** : • pour définir un composite comme une réplication infinie de lui-même	- Il n'est pas possible de supprimer les éléments architecturaux
***Π*-Space**	-***suffix π*** : pour définir qu'un élément est dynamique. -définir des règles de dynamique permettant de décrire les changements de topologie dus la création des nouveaux éléments : • Les composants déclenche la dynamique • Des portes spécifiques déclenchent la reconfiguration de la topologie (AttachementEvolutionport, componentEvolutionport, Evolutionport).	-n'est pas spécifiques à un domaine, -propose une notation spécifique très difficile

FRACTEL	-Fractel API : pour l'introspection et la reconfiguration dynamiques des composants et d'assemblage de composants : • Création des types des composants • Création des composants • Assemblage des composants • Démarrage de l'application	Fractal n'intègre pas explicitement la notion de port. Elle est intégrée dans la notion d'interface cela provoque bien souvent des ambiguïtés, - Le support des connecteurs via les binding components semble assez primitif et peu étudié, - Fractal ne permet pas une séparation claire entre les propriétés fonctionnelles et non fonctionnelles prises en charge par la membrane.
Pilar	-la sémantique de Pilar est définie par **π-calculus ,** -le comportement des composants et connecteurs est spécifiée par CCS.	-Notation très implicite -ne supporte pas la notion de style architectural

Tableau 1:Tableau de spécification dynamique des ADLs

4. Conclusion

Dans ce chapitre, nous avons passé en revue les travaux les plus représentatifs des langages de spécification d'architecture logicielle dynamiques. Plusieurs efforts et tentatives étaient consacrés pour proposer des solutions pour supporter la notion de dynamisme, la majorité des solutions existantes comptent souvent sur le faible niveau des langages non formelles ou au contraires sur des notations spécifiques, très formelles, ce qui nécessite un effort de plus pour l'architecte non seulement pour comprendre le langage de spécification mais même pour pouvoir spécifier l'architecture de son système à concevoir.

Chapitre 3 :

Spécification du méta-model *de l'architecture* Dynamique IASA

1. Introduction

L'anticipation du changement consiste dans le domaine de l'architecture logicielle à construire un modèle conscient du besoin permanent de changement du système, capable de spécifier et de « cadrer » ces changements. Cette fonctionnalité est devenue un véritable besoin pour le développement de logiciels de nos jours. En effet, les systèmes logiciels modernes et leur architecture doivent s'adapter dynamiquement aux événements provenant de l'environnement (par ex., les changements de fonctionnalité) et la plate-forme d'exécution (par exemple, la disponibilité des ressources).

Nous considérons deux principaux types de changement associés à une description d'architecture logicielle. Le premier concerne la dynamique structurelle de l'architecture aussi appelée reconfiguration structurelle. Le deuxième concerne l'adaptation la dynamique comportementale de l'architecture (possibilité pour un seul composant à se comporter différemment selon l'environnement dans lequel elle opère ou dans le temps).

La description de l'architecture logicielle doit donc prendre en compte ces deux types d'évolution au niveau du modèle pour permettre à l'architecte d'intégrer dans sa spécification ces possibilités d'évolution. Notre contribution qui sera présentée à travers ce chapitre s'incorpore dans ce contexte.

2. Rappel de la problématique

Les architectures à base de composants se sont avérés être très adapté à l'auto-adaptation en particulier avec leurs capacités de reconfiguration dynamique. Cependant, selon l'étude des langages de spécification d'architecture logicielle que nous avons menée dans le chapitre 2, la majorités des solutions existantes pour le support de ce dynamisme comptent souvent sur le faible niveau, impératif, des langages non formelles ou ne proposent pas des mécanismes pour le support de l'adaptation comportementale contrairement à la reconfiguration structurelle qui est bien prise en charge par ces ADLs. Souvent ces ADLs dynamique proposent une notation spécifique, très formelle, ce qui nécessite un effort de plus pour l'architecte non seulement pour comprendre le langage de spécification mais même pour pouvoir spécifier l'architecture de son

système à concevoir. De plus, l'interopérabilité de ces langages formels avec une autre description technique des architectures est limitée.

Ces observations nous ont motivés pour de fournir, en premier lieu un soutien de haut niveau pour décrire les comportements et les politiques d'adaptation (d'évolution) dans les architectures à base de composants. En se basant sur l'architecture orientée événements. Le choix de l'architecture orientée événements est justifié par le fait que cette dernière semble être une bonne base pour réduire la complexité de l'évolution dynamique des architectures. En effet, les avantages clés sur les qualités architecturales de l'architecture orientée événements sont sa grande flexibilité, son évolutivité, d'une part et d'autre part, l'architecte peut compter sur des outils qui apprivoisent la nature faiblement couplée de ces architectures et les rendent maniable et analysable.

Nous proposons dans ce travail d'appliquer notre solution dans le cadre de l'approche IASA (Integrated Approach for Software Architecture) (12). IASA (voir section 3) propose un ADL nommé X3ADL (20)extensible et souple ce qui nous permet d'enrichir et d'étendre facilement cet ADL pour supporter les différents types d'évolution de l'exécution et de l'adaptation dynamique.

La deuxième étape est de concevoir un outil d'aide à la spécification (Editeur graphique) pour supporter notre métamodèle proposé. Ce métamodèle sera basé sur des concepts primitifs qui peuvent facilement être analysées avec des outils ou être utilisés pour représenter une représentation actuelle de l'architecture du système en utilisant des notations graphiques proches de la perception intuitive des architectes logiciels.

3. L'approche IASA

Nous proposons dans ce travail d'appliquer notre solution de l'architecture logicielle dans le cadre de l'approche IASA (Integrated Approach for Software Architecture) (12).Cette approche est basée sur un ensemble de concepts que nous essayons de définir ci-dessous.

3.1. Les composants

Le concept de composant est utilisé pour représenter n'importe quel élément rentrant dans la définition fonctionnelle d'une application. Cela veut dire que toute fonctionnalité faisant partie de la logique d'une application est explicitement prise en charge par un composant. Dans IASA, nous distinguons deux types de composants : les composants primitifs et les composants composites. La structure interne d'un composant primitif est inaccessible. Celle d'un composant composite possède une organisation bien précise. Elle est composée de trois parties. Une première partie, appelée partie opérative (représenté par le type OperativePart), comprend les composants réalisant les fonctionnalités pures de l'architecture. La deuxième partie, appelée partie contrôle (représenté par le type ControlPart), contient des composants qui réalisent les opérations de contrôle global sur les autres composants des deux autres parties. La troisième partie, appelée partie aspect (représenté par le type OptionPart), est optionnelle, elle est dédiée à contenir les aspects techniques d'une application.

L'instanciation d'un composant est réalisée dans le contexte du concept d'enveloppe. Une enveloppe permet d'isoler l'instance pure d'un composant de son environnement d'exploitation en fournissant à ce dernier les éléments nécessaires à l'exploitation de l'instance. L'enveloppe est en réalité l'endroit où seront solutionnés les divers problèmes liés au déploiement de l'instance du composant et à la spécification de topologies très variées, notamment celle mettant en œuvre directement les points d'accès de port.

La description en X3ADL d'un composant composite est hiérarchisé en plusieurs niveaux, le premier niveau est illustré dans la figure 6 .La balise <Component> possède les attributs name qui donne le nom du composant et deployedAs qui renseigne sur le cas de déploiement du composant.

```
- <Component name="" deployedAs="">
    <!-- DEFINITION DU COMPOSANT  -->
  - <Ports>
      <!-- DEFINITION DES PORTS  -->
    </Ports>
  - <Connectors>
      <!-- DEFINITION DES CONNECTEURS  -->
    </Connectors>
  - <OperativePart>
      <!--  DEFINITION DE LA PARTIE OPERATIVE  -->
    </OperativePart>
  - <ControlPart>
      <!--  DEFINITION DE LA PARTIE CONTROLE  -->
    </ControlPart>
  - < OptionPart>
      <!--  DEFINITION DE LA PARTIE ASPECT     -->
    </ OptionPart>

  - <Properties>
      <!-- SPECIFICATION DES PROPRIETES  -->
    </Properties>
</Component>
```

Figure 6: Les balises du langage x3ADL (20)

3.2. Les ports

Les ports modélisent la vue externe d'un composant. IASA supporte deux types de port : Les ports orientée flux de données et les ports orientés actions. Ces derniers sont des ports de services. Un port est un ensemble de point d'accès. Un point d'accès est accessible individuellement. Comme les ports les points d'accès sont soit de points d'accès orientés flux de données (DOAP pou Data Oriented Access point), soit des points d'accès représentant un service (ACTOAP pour Action Oriented Access Point). Les points d'accès ASPOAP sont des points d'accès utilisés lors de la spécification orienté aspect d'une architecture.

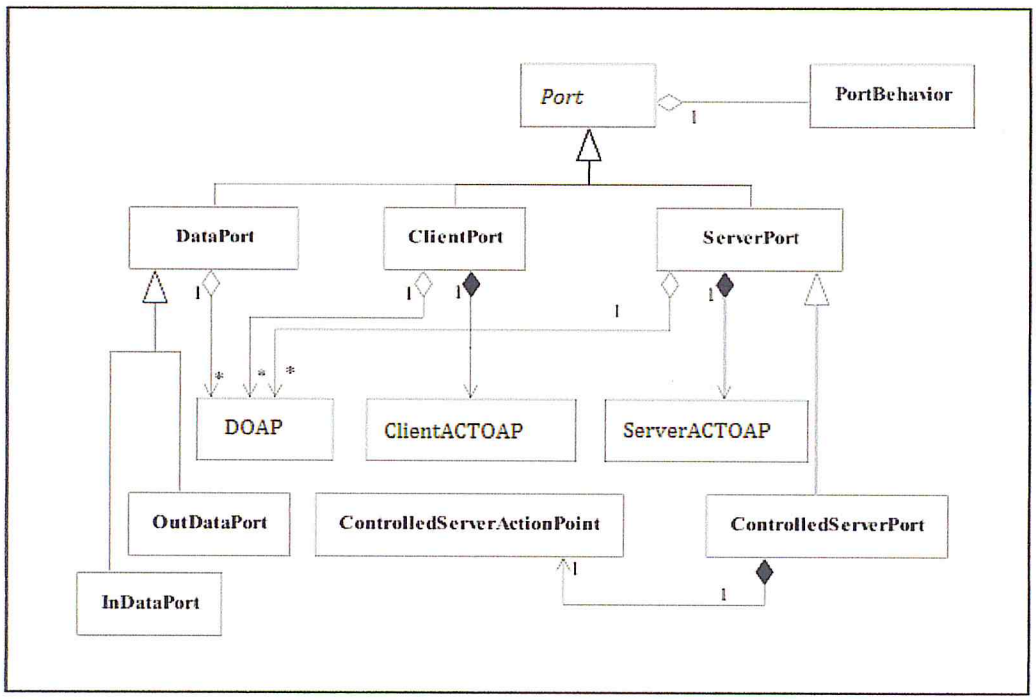


Figure 7: Extrait du Méta-modèle du port de IASA (20)

Dans IASA un service est un ensemble d'actions. Une action pourrait correspondre à une méthode d'un objet, une fonction ou une procédure. Au sein d'un même port une action peut être associée à un ou plusieurs points d'accès orientés données et plusieurs actions peuvent être associées aux mêmes points d'accès de données.

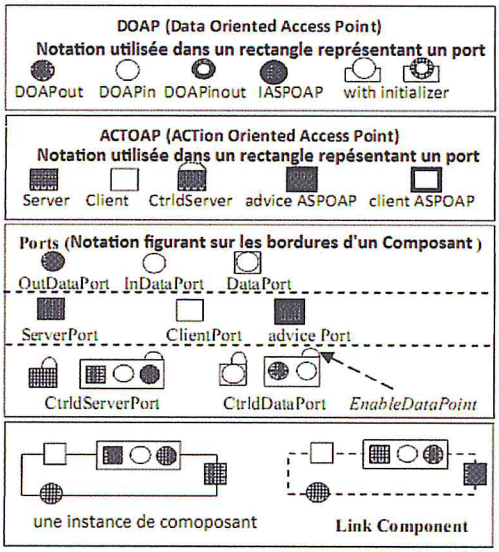


Figure 8: Les principales notations graphiques de *l'approche IASA* (12)

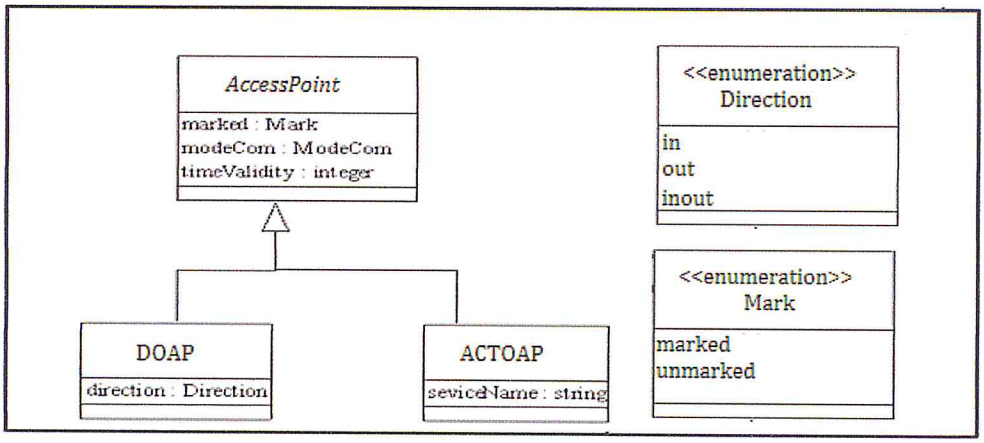


Figure 9: Extrait du Méta-modèle des points d'accès (20)

3.3. Les connecteurs

Lors de la définition des connexions entre les ports de composants, l'architecte spécifie parfois une interaction et parfois indique que la connexion doit être réalisée par une technologie bien connue tels qu'un protocole de communication ou un appel distant de procédure dans le contexte d'une infrastructure de communication (i.e. Bus CORBA, RMI). Afin de pouvoir spécifier librement diverses topologies, IASA utilise un seul modèle de connecteur de base qui est le connecteur direct représenté par un appel de procédure.

4. Architecture orientée événement

4.1. Motivations

L'architecture orientée évènements est un modèle de conception d'un système, plus particulièrement d'un système informatique. L'architecture proposée par l'EDA est modulaire et découplée ; les différents composants interagissent entre eux simplement par la diffusion d'évènements. C'est une architecture "capable de détecter les évènements et d'y réagir intelligemment", où un évènement est un changement d'état du système.

Le concept n'est pas nouveau, puisqu'il est à la base de n'importe quel système de régulation. Un changement d'état survient, il est détecté par le système, le système y réagit pour le compenser. L'exemple le plus simple est le thermostat, qui régule l'ouverture d'une valve en fonction de la température de la pièce. Si on ouvre une fenêtre

en plein hiver, le thermostat détectera une forte baisse de la température et il allumera le chauffage à pleine puissance.

S'intéresser aux évènements est donc quelque chose de tout à fait naturel : l'humain lui-même peut être vu sous cet angle-là. Ainsi, Taylor, Yochem, Philips et Martinez proposent une métaphore de l'EDA comme "système nerveux" de l'entreprise (38). Ils prennent comme exemple une situation très simple : un chat qui marche sur le pied de quelqu'un. Il est fortement probable qu'à l'instant d'après, cette personne aura effectivement réalisé qu'un chat lui avait marché dessus. Dans l'intervalle, pourtant, beaucoup de choses se seront passées. D'abord, l'orteil malmené va envoyer un signal au cerveau pour le prévenir de la sensation (pression, texture, etc). Le cerveau va corréler les différentes informations reçues et sans doute en déduire que, probablement, le responsable est un animal de petite taille. Ceci va provoquer l'envoi d'un évènement aux yeux leur demandant de se baisser pour vérifier l'information. Au final, l'ensemble des informations reçues et la manière dont elles sont liées entre elles dans le temps et l'espace permet au cerveau de savoir ce qui s'est réellement passé et d'y réagir intelligemment : la personne va sans doute repousser le chat fermement.

Si le coupable avait été tout autre, un lion par exemple, une réaction plus prudente aurait été certainement préférée.

On peut donc voir un EDA exactement comme le système nerveux dont nous venons de discuter. Pour être efficace, il doit détecter correctement les évènements et les transmettre de manière fiable aux composants concernés. De plus, pour pouvoir y réagir intelligemment, il aura besoin d'être capable de faire efficacement les liens entre les différents évènements reçus : le composant chargé de cette tâche sera appelé processeur d'évènements.

L'EDA est une architecture logicielle reposant sur le principe d'un découpage de l'application en plusieurs parties, interagissant entre elles uniquement par la diffusion d'évènements (asynchrones et sans destinataires) en temps réel. Le fait que cette architecture permette de détecter ces évènements et d'y réagir intelligemment constitue l'élément clé la caractérisant (38).

EDA semble donc, être une solution prometteuse pour notre problématique, elle se compose d'un certain nombre de composants de calcul ou de données qui communiquent les uns avec les autres en émettant et recevant des événements. Dans un système basé sur des événements, le composant est absolument pas au courant des autres et est déclenché indirectement par des événements particuliers émis par les autres

composants, ce qui conduit à un degré élevé de flexibilité. Il y a un riche corpus de travail dans différents domaines de recherche qui étudient et exploitent les avantages importants de l'EDA. Ceci nous a amène à proposer une nouvelle approche exploitant le style de l'architecture EDA pour permettre la flexibilité des architectures pour supporter différents types d'évolution de l'exécution et de l'adaptation dynamique en se basant sur des concepts primitifs qui peuvent facilement être analysées avec des outils ou être utilisés pour représenter une représentation actuelle de l'architecture du système en utilisant des notations graphiques proches de la perception intuitive des architectes logiciels (39).

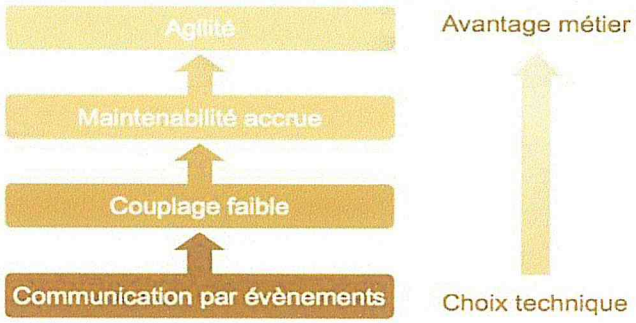


Figure 10 : ***Avantages d'une architecture orientée événements (EDA)*** (39)

4.2. Concepts du base

4.2.1. Événement

On peut voir un évènement comme quoi que ce soit de notable survenant à l'intérieur ou à l'extérieur du système. Plus formellement, un évènement sera défini comme un changement d'état, c'est à dire une variation suffisamment significative des paramètres définissant l'état système. N'importe quelle donnée apportant une information inédite peut donc être considérée comme un évènement. Cette définition permet d'inclure n'importe quelle action, n'importe quelle variation d'une valeur. Cela peut être :

- Une action effectuée par un utilisateur du système
- Une donnée envoyée par un capteur, à fréquence fixe
- Un e-mail envoyé sur une mailing-list

Ces quelques exemples tentent de montrer que la définition d'évènement est volontairement très large et recouvre tout type d'information pertinente pour le système. Un évènement contiendra l'état complet du processus qui lui est associé. En plus de permettre aux composants du système de ne pas maintenir d'état, ce qui est notre but, ceci permet de rendre le système plus souple, offrant la possibilité d'ajouter plus tard des éléments réagissant à cet évènement sans devoir modifier le reste de l'application (39).

4.2.2. Chanel d'événement (Event Channel)

L'efficacité d'une EDA repose entièrement sur la bonne diffusion des évènements. Le composant utilisé pour gérer cette tâche est appelé **Event Channel**.
Cet élément doit pouvoir efficacement transmettre les évènements depuis et vers les différents **actors**.

4.2.3. Acteurs(Actors)

Reliés à l'**Event Channel**, les acteurs sont les éléments qui vont réellement effectuer le travail dans l'architecture. On considère deux types d'acteur : ceux qui produisent, ceux qui consomment évènements sont présentés ici de manière distincte. En pratique, un acteur sera souvent une combinaison de ces éléments, tour à tour consommateur et producteur d'évènements.

4.2.3.1. Générateur d'événement

Cet élément va s'occuper de faire la transition entre la source de l'évènement (qui peut être complètement externe au système, venant d'une application tierce, fournie manuellement par une personne, etc) et le système EDA. Il va convertir l'évènement dans une représentation standardisée compatible avec le reste du système.

4.2.3.2. Consommateur d'évènements

Ce composant sera le consommateur d'évènements, il recevra tous les évènements pour lesquels il a exprimé un intérêt. Cet évènement sera alors traité par un processeur d'évènements.

Processeur d'évènements

Un module logiciel qui effectue des traitements sur les événements. Le traitement peut inclure des opérations comme le filtrage, la corrélation (temporelle, causale ou spatiale), l'agrégation, etc.

5. Dynamique IASA

Dans cette section, nous présentons les concepts et les techniques développés pour l'approche IASA afin de supporter la spécification du comportement dynamique, selon un modèle à un haut niveau d'abstraction basé sur les architectures orientées évènements.

Le modèle de composant de l'approche IASA et son ADL X3ADL permettent une haute flexibilité dans la spécification des propriétés dynamiques d'une architecture. Nous proposons d'enrichir X3ADL en ajoutant une partie (une balise) «DynamiqueBehavior» pour modéliser le comportement dynamique, Les composants de cette balise seront détaillés dans la section (5.2).

```
- <Component name="" deployedAs="">
    <!-- DEFINITION DU COMPOSANT -->
  - <Ports>
      <!-- DEFINITION DES PORTS -->
    </Ports>
  - <Connectors>
      <!-- DEFINITION DES CONNECTEURS -->
    </Connectors>
  - <OperativePart>
      <!-- DEFINITION DE LA PARTIE OPERATIVE -->
    </OperativePart>
  - <ControlPart>
      <!-- DEFINITION DE LA PARTIE CONTROLE -->
    </ControlPart>
  - < OptionPart>
      <!-- DEFINITION DE LA PARTIE ASPECT -->
    </ OptionPart>
  - <Properties>
      <!-- SPECIFICATION DES PROPRIETES -->
    </Properties>
     <DynamiqueBehavior>
        <!-- specification de comportement -->
     </DynamiqueBehavior>
    </compenent>
```

Figure 11 : Dynamique X3ADL

5.1. Métamodele proposé

Dans notre approche, un composant est représenté par un acteur d'événement (ou **Actor**) qui représente une unité de calcul ou de traitement des données. Un événement peut être considéré essentiellement comme «tout événement d'intérêt qui peut être observée à partir d'un ordinateur" (ou un système logiciel) (20).

Les acteurs sont les éléments qui vont réellement effectuer le travail dans l'architecture. On considère deux rôles d'acteurs se qui produisent(Générateur) et ce qui consomment

(Consommateur), en pratique un acteur sera souvent une combinaison de ces éléments tour à tour consommateur et générateur d'événement. Acteurs fournissent deux types d'interfaces **input** et **output**. Chaque acteur est composé d'une ou plusieurs interfaces bien définies et représenté dans la classe **Interface**. Un consommateur reçoit des événements à partir d'une interface **input** et un générateur fournie des événements à travers une interface **output**. Les interfaces sont reliées par un **EventChannel** qui transmet les événements entre les acteurs. Le comportement de chaque acteur est défini dans la classe **Behavior.**

Nous avons défini trois types d'acteurs :

1. Le **Trigger** (déclencheur) est caractérisé par un ensemble d'interfaces de type **output.**
2. Une **Condition** est composée d'une ou plusieurs prédicats.
3. **Barrier** (barrière) peut être nécessaire quand nous avons besoin d'attendre plus d'un événement d'arriver avant d'exécuter d'autres tâches.

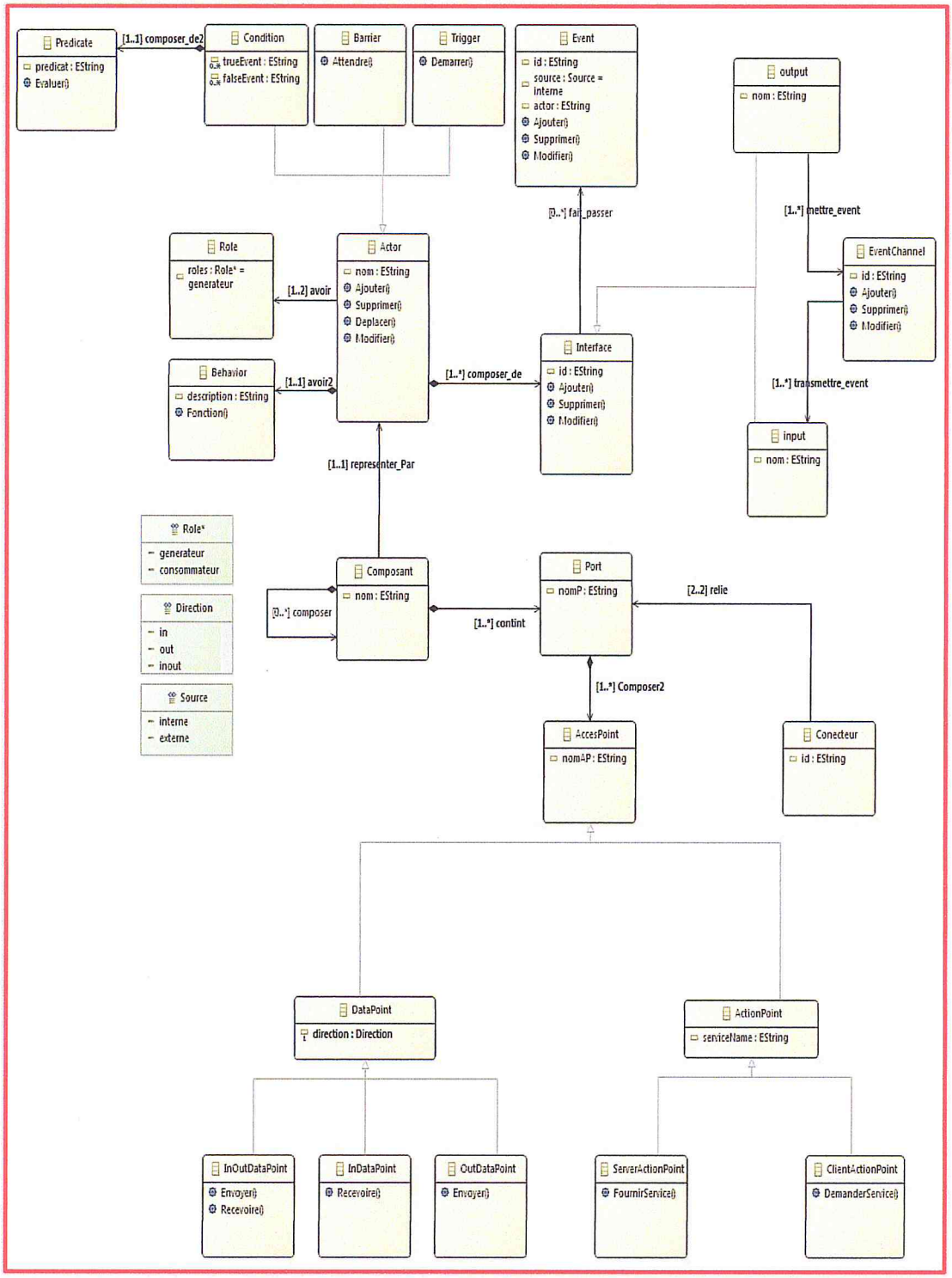


Figure 12 : Méta-modèle de DynamiqueIASA

5.2. Représentations graphiques des concepts

Les notations graphiques des concepts de l'approche proposés peuvent être utilisées pour représenter visuellement un instantané actuel de l'architecture.

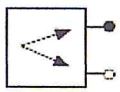
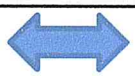
Concept	Notation	DTD
Actor		<!ELEMENT Actor (input+,output+,Behavior)> <!ATTLISTE Actor nom CDATA mode> <!ELEMENT input (Event+)> <!ELEMENT output(Event+)> <!ELEMENT Behavior (# PCDATA)>
Barrier		<!ELEMENT Barrier (input+,output+)> <!ATTLIST Barrier nom CDATA mode> <!ELEMENT input (Event+)> <!ELEMENT output(Event+)>
Condition		<!ELEMENT Condition (input+,predicate,trueEvent+,falseEvent+)> <!ATTLIST Condition nom CDATA mode> <!ELEMENT input(Event+)> <!ELEMENT trueEvent(Event+)> <!ELEMENT falseEvent(Event+)> <!ELEMENT predicate(# PCDATA)>
Trigger		<!ATTLIST Trigger nom CDATA mode> <!ELEMENT Trigger (output+)> <!ELEMENT output(Event+)>
Event		<!ELEMENT Event EMPTY> <!ATTLIST Event id CDATA mode createur CDATA mode >
EventChannel		<!ELEMENT EventChannel EMPTY> < !ATTLIST EventChannel id CDATA mode>
output		<!ELEMENT output EMPTY> < !ATTLIST output id CDATA mode>
input		<!ELEMENT input EMPTY> < !ATTLIST input id CDATA mode>

Tableau 2 : Représentation graphique des concepts

5.3. Exemples

Premier exemple : Montre digitale

Une montre affiche l'heure. Quand un utilisateur appuie 2 fois sur le bouton B1, la montre passe vers le mode « **Modification heures** », chaque pression sur le bouton B2 incrémente l'heure d'une unité, ensuite si l'utilisateur appuie sur le bouton B1 la montre passe vers le mode « **Modification minutes** » et chaque pression sur le bouton B2 incrémente les minutes d'une unité. Après une minute la montre revient automatiquement au mode « **affichage de *l'heure*** ». Si l'utilisateur appuie une quatrième fois sur le bouton B1, la montre affiche l'heure courante.

La figure suivante représente un diagramme état-transition qui montre d'une façon plus claire cet exemple :

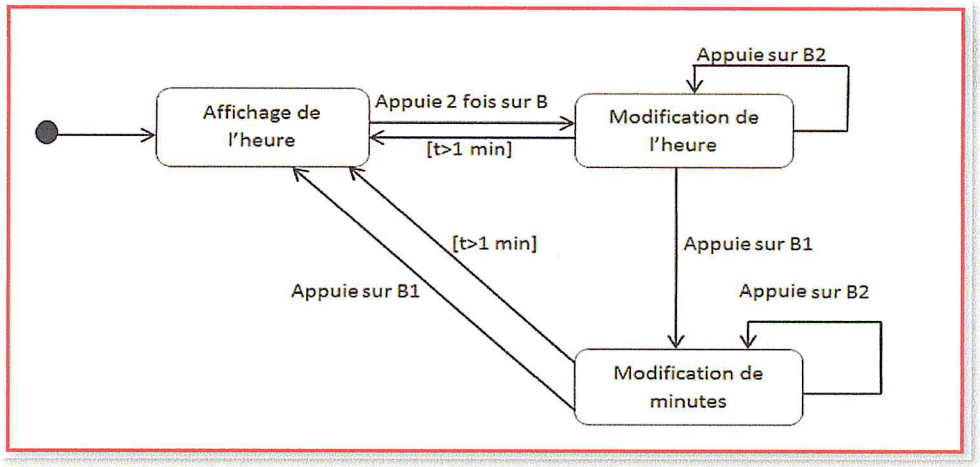


Figure 13:diagramme Etat-transition de la montre digitale

On va appliquer cet exemple sur notre approche :

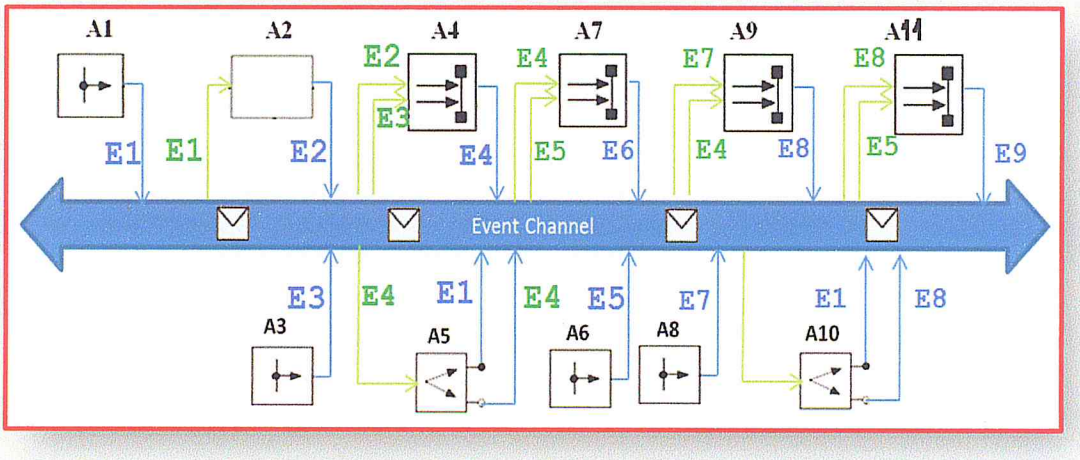


Figure 14:l'application de l'exemple sur notre approche

Chaque boite représente un acteur (**Trigger,Barrier,Actor**),les flèches bleus désigne les **outputs** des acteurs et les flèches verts sont des **inputs,** les évènements fournie par ces acteurs sont émis dans **Event Channel** à travers les interfaces **outputs** et les acteurs reçoit les évènements qu' ils les intéresse par **Event Channel** à travers les interfaces **inputs.**

Dans cet exemple on a besoin des acteurs suivants:

- Quatre acteurs trigger :
  - **A1** pour déclencher le système.
  - **A3** pour produire l'événement **E3**.
  - **A6** pour produire l'événement **E5**.
  - **A8** pour produire l'événement **E7**.
- Deux conditions **A5** et **A10** : pour vérifier si on a dépassé une minutes ou non.
- Quatre barrières :
  - **A4** pour attendre l'apparition des événements **E2** et **E3** avant d'exécuter et émettre l'événement **E4** en sortie.
  - **A7** pour attendre l'apparition des événements **E4** et **E5** avant d'exécuter et émettre l'événement **E6** en sortie
  - **A9** pour attendre l'apparition des événements **E7** et **E4** avant d'exécuter et émettre l'événement **E8** en sortie.
  - **A11** pour attendre l'apparition des événements **E8** et **E5** et émettre l'événement **E9**.
- Un acteur (actor) **A2**.

Liste des événements :

Symbole	événement
E1	démarrageMontre
E2	modeAffichageHeure
E3	appuieB1-2fois
E4	modeModificationHeure
E5	appuieB2
E6	heureChangée
E7	appuieB1
E8	modeModificationMin
E9	minuteChangée

Tableau 3: Liste des événements de l'exemple1

La figure suivante représente le code XML généré par notre approche :

```
<Trigger nom="A1">
    <outPut>
        </Event id="demarrageMontre"createur="A1">
    </outPut>
</Trigger>
<Actor nomA ="A2">
    <input>
        </Event id="demarrageMontre"createur="A1" >
    </input>
    <behavior>afficherHeure"</behavior>
    <output>
        </Event id="modeAffichageHeure" createux="A2">
    </output>
</Actor>
<Trigger nom="A3">
    <outPut>
        </Event id="appuie2FoisB1"createur="A3" >
    </outPut>
</Trigger>

<Barrier nom="A4">
    <input>
        </Event id="modeAffichageHeure"createur="A2">
        </Event id= "appuie2FoisB1" createur="A3" >
    </input>
    <behavior>demarrerModificationHeure</behavior>
    <output>
        </Event id="modeModificationHeure" createur="A4" >
    </output>
</Barrier>

<Condition nom="A5">
    <input>
        </Event id="modeModificationHeure" createur="A4">
    </input>
    <Predicat> t<=1 min </predicat>
    <outPut>
        <!--si le predicat est vrai alors l'outPut= « ModeModificationHeure »
        sinon l'outPut=modeAffichageHeure » -->
    </outPut>
</Condition>

<Trigger nom="A6">
    <outPut>
        </Event id= "appuieB2" createur="A6" >
    </outPut>
</Trigger>

<Barrier nom="A7">
    <input>
        </Event id="modeModificationHeur" createur="A4">
        </Event id="appuieB2" createur="A6">
    </input>
    <behavior>modifierHeure</behavior>
    <output>
        </Event id="heureChangée"createur="A7"" >
    </output>
</Barrier>

<Trigger nom="A8">
    <outPut>
        </Event id= "appuieB1"createur="A8" >
    </outPut>
</Trigger>

<Barrier nom= "A9">
    <input>
        </Event id="modeModificationHeure" createur="A4">
        </Event id="appuieB1" createur="A8">
    </input>
    <behavior>demarrerModificationMin</behavior>
    <output>
        </Event id="modeModificationMin"createur="A9">
    </output>
</Barrier>

<Condition nom="A10">
    <input>
        </Event id="modeModificationMin" createur="A9">
    </input>
    <Predicat> t<=1 min </predicat>
    <outPut>
        <!--si predicat est vrai alors l'OutPut= « modeModificationMin »
        sinon l'outPut= Mode AffichageHeue » -->
    </outPut>
</Condition>

<Barrier nom="A11">
    <input>
        </Event id="ModeModificationMin" createur="A9">
        </Event id="appuieB2" createur="A6">
    </input>
    <behavior>modifierMin</behavior>
    <output>
        </Event id="minutesChangée"createur="A12" >
    </output>
</Barrier>
```

Figure 15 : Code XML du l'exemple (Montre digitale)

Deuxième exemple : réservation voyage

On va présenter un deuxième exemple pour illustrer notre approche est celui de la : **procédure de réservation des vols, hôtels et locations de voiture**.

Description de l'exemple

Après avoir reçu une demande de voyage d'un client, exprimé par l'acteur **Receive-Itinerary-Request** , l'événement **initialized(E1)** est émis. Cet événement encapsule les données d'itinéraire. L'acteur **Book-Flight** reçoit cet événement, interprète les données et tente de trouver un vol approprié. Quand un vol est trouvé, l'événement **flightFound(E2)** est émis, ce qui provoque l'acteur **Book-Hotel** à s'exécuter. **Book-Hotel** tente de trouver un adapté hôtel et émet l'événement **HotelFound(E3).** Cela provoque l'acteur **Book-Car** pour réserver une voiture de location, en émettant l'événement **CarFound(E4)**. Cet événement déclenche l'acteur **Confirm-with-constumer** qui permet au client d'approuver les réservations proposées. Quand le client est satisfait, l'événement **confirmed(E5)** est émis.

Cet exemple est représenté dans notre approche par la figure ci-dessous

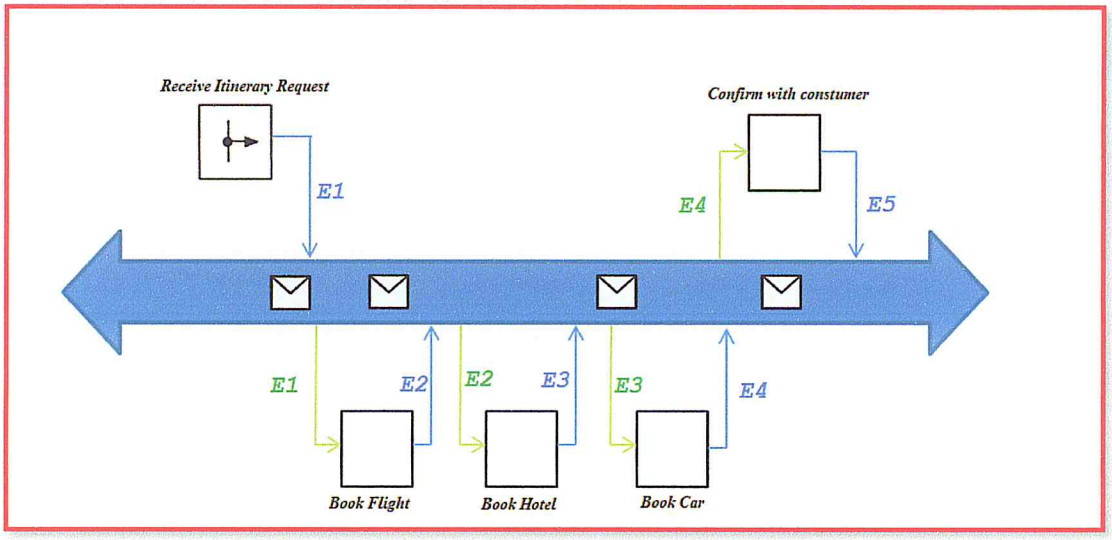


Figure 16:Application de notre approche sur l'exemple 2

Nous voulons améliorer l'architecture de la demande de réservation de voyage. En particulier, dans la première conception du système, l'exécution séquentielle des acteurs ne peuvent pas être appropriée, étant donné que toutes les données pertinentes pour les acteurs **Book-Flight, Book-Hotel**, et **Book-Car** sont déjà contenues dans l'événement **initialized**. Une conception améliorée serait de paralléliser les acteurs concernés, comme représenté sur la figure 17. Cette modification permet aux acteurs, **Book-Flight**, **Book-Hotel** et **Book-Car** à exécuter en parallèle après l'événement **initialized (E1)**. Nous avons aussi insérer une barrière supplémentaire, en attendant l'apparition des trois événements **flightFound (E2)**, **hotelFound(E3)**, et **CarFound(E4)** puis déclencher le dernier acteur comme montre la Figure 17. A cet effet, nous avons inséré un nouvel acteur **Aggregate Bookings** (une barrière). L'acteur barrière attend l'apparition de tous les événements avant d'exécuter et émettre ses événements de sortie (E5). Après la fin de l'exécution de l'acteur **Aggregate-Bookings**. L'acteur **Confirm-with-constumer** est déclenché. A la fin l'événement **confirmed(E6)** est émis.

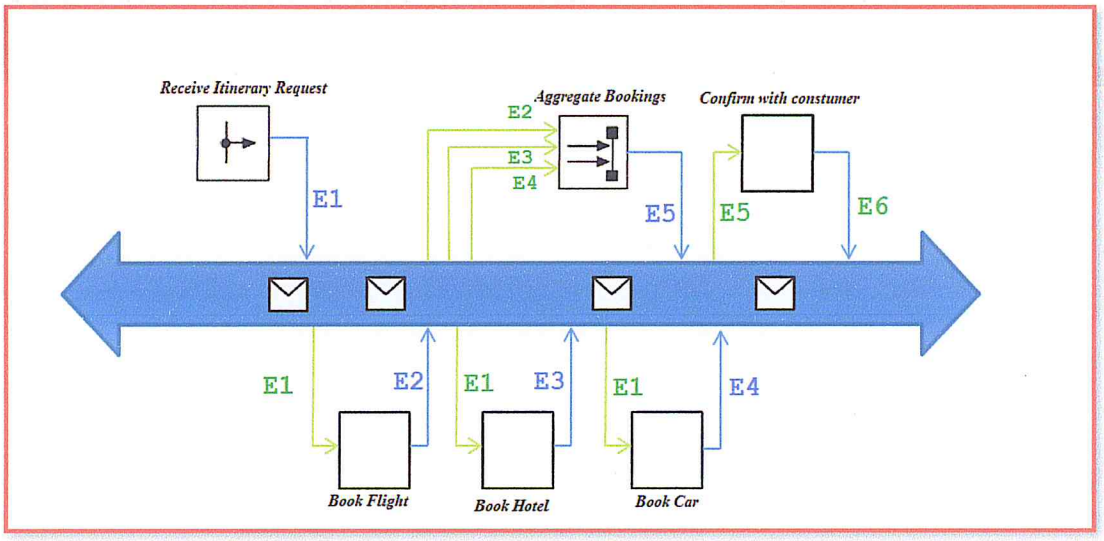


Figure 17:deuxieme application de notre approche sur l'exemple 2

6. Conclusion

Dans ce chapitre nous avons présenté les concepts fondamentaux de notre approche en spécifiant les différents concepts du méta-modèle de notre approche nommée **Dynamique IASA**. Nous avons clôturé le chapitre par deux exemples détaillés.

Chapitre 4 :

L'implémentation de l'éditeur graphique

1. Introduction

Dans le chapitre précédent nous avons présenté notre méta-model **DynamiqueIASA**, et donné la description détaillée concernant ce dernier, nous allons dans ce chapitre traiter les étapes d'implémentation de l'éditeur graphique, on a commencé par décrire l'environnement et l'outil de travail utilisé pour sa réalisation.

2. Environnement et outils de travail

Le choix des outils de travail pour notre travail de fin d'étude n'été pas facile vue la variété des technologies existantes, notre choix était fait par rapport aux nos objectifs.

2.1. L'environnement de développement Eclipse

Eclipse[2] est un environnement de développement intégré libre, puissant, extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importent quel langage de programmation.

Il est développé par IBM, il est gratuit et disponible pour la plupart des systèmes d'exploitation, il est principalement écrit en java, il offre une plateforme ouverte pour le développement d'application, il est facile à comprendre mais aussi facile à étendre.

Il existe plusieurs versions d'Eclipse on note : Eclipse 2.0, Eclipse 3.0, Eclipse 3.2 Europa, Eclipse 3.7 Indigo, Eclipse 4.2 Juno, Eclipse Mars. dans notre cas on utilise **Eclipse Mars.** Eclipse est fournie sous forme de plusieurs package comme une commodité pour les utilisateurs ; ils représentent des configurations communes de projets Eclipse qui sont utilisés ensemble pour le développement web. Dans notre cas nous allons utiliser Eclipse ModelingTools [3] qui propose des outils et des runtimes pour la création des applications à la base de modèles.

[2] http://www.eclipse.org
[3] http://www.eclipse.org/downloads/eclipse-modeling-tools/keplersrs2

2.2. Le Framework GMF

GMF[4] (Graphical Modeling Framework) est un framework de l'environnement de travail Eclipse. Il propose un composant génératif et une infrastructure permettant l'exécution d'éditeurs graphiques basés sur **EMF**[5](Eclipse Modeling Framework) qui est un framework de manipulation de modèle de données structurées et **GEF** (Graphical Editing Framework) qui est un framework de création d'éditeurs graphiques.

GMF permet aux développeurs de crée un éditeur graphique riche orienté modèle, à partir d'un modèle de domaine existant. Son efficacité réside dans le fait qu'il fournit rapidement un aspect visuel à presque tous des modèles de domaine, il génère une surface de création de diagramme permettant de travailler visuellement avec ce modèle.

[4] http//www.eclipse.org/modeling/gmf/
[5] http:// www.eclipse.org/modeling/emf/

3. Implémentation

Nous avons choisi GMF pour implémenter notre éditeur graphique. GMF requiert trois modèles de base en entrée décrivant les différents aspects d'un éditeur graphique, et peut générer le code de l'éditeur graphique, et générer le code de l'éditeur en tant que plugin Eclipse ou application indépendante.

GMF fournit une vue qui facilite la création et la modification des divers modèles **(GMF Dashboard)**.elle permet également de faire les liens entre eux.

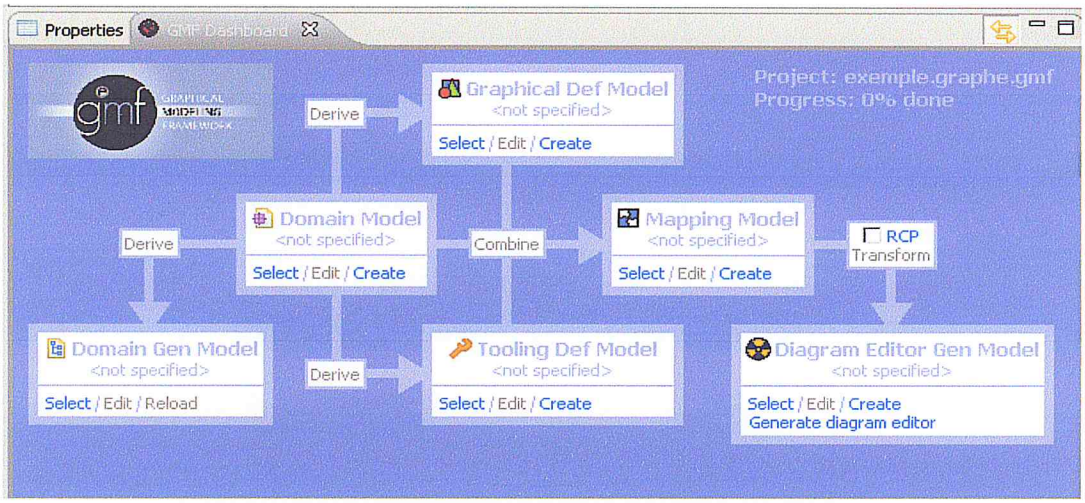


Figure 18: Le tableau de bord GMF

1. **Modèle de domaine:** le méta-modèle nous voulons utiliser pour créer l'éditeur graphique.
2. **Domain Gen Model (.genmodel):** ce fichier est utilisé pour générer le code du modèle de domaine avec EMF (il est le fichier EMF genModel).
3. **Graphical Def Model (.gmfgraph):** ce fichier est utilisé pour définir les éléments graphiques pour notre modèle de domaine.
4. **Tooling Def Model (.gmftool):** ce fichier est utilisé pour définir la palette d'outils que nous pouvons utiliser dans l'éditeur graphique.
5. **Modèle de cartographie (.gmfmap):** ce fichier relie le modèle de domaine, le modèle graphique (.gmfgraph) et le modèle d'outillage (.gmftool).
6. **Diagram Editor Gen Model (.gmfgen):** ce fichier final nous utilisé pour générer l'éditeur graphique GMF en plus du code EMF généré par le fichier .genmodel.

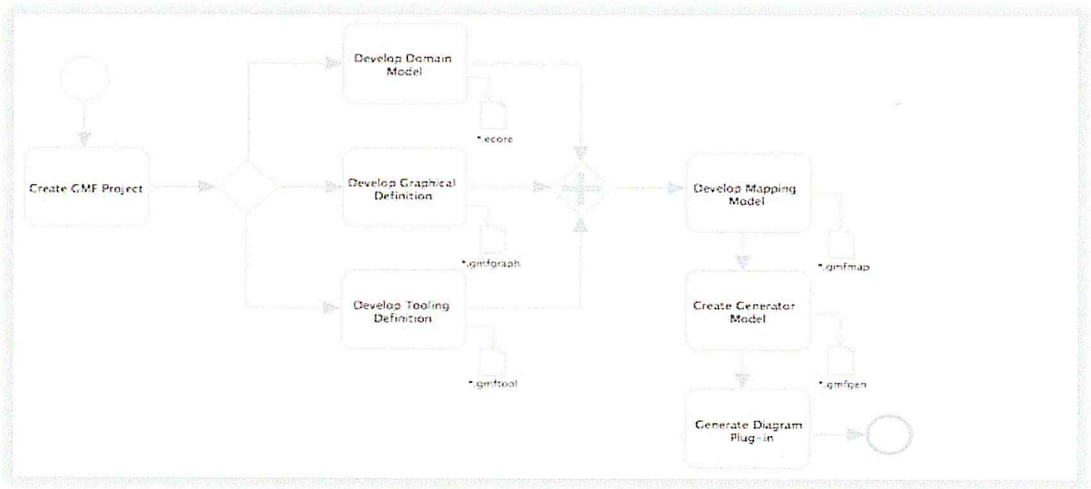


Figure 19: Les modèles requis par GMF

3.1. Le model de domaine (*.ecore)

C'est le méta-modèle décrivant le domaine de Dynamique IASA. Il a été importé à partir de notre méta-modèle que nous avons défini dans le chapitre précédent. Il présente les concepts, les attributs, et les relations entre les concepts.

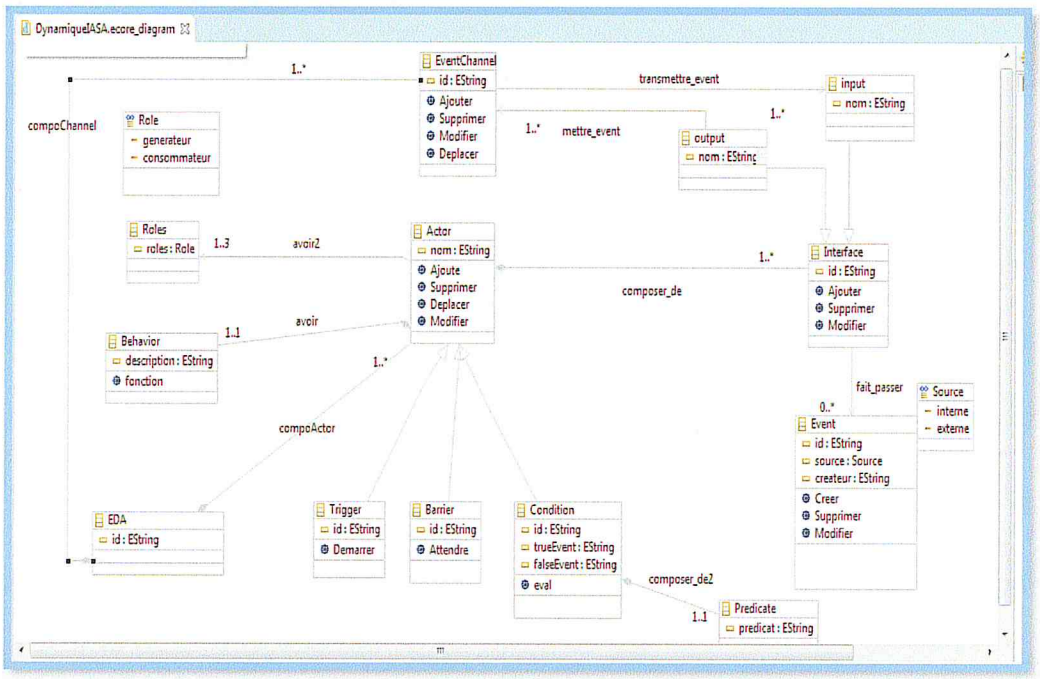


Figure 20: Le méta-modèle utilisé

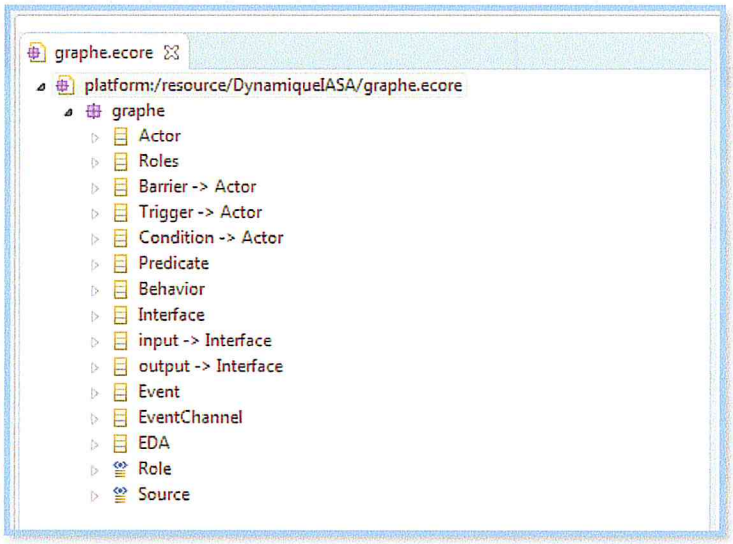


Figure 21: Le modèle de domaine de l'éditeur graphique Dynamique IASA

3.2. Le model de génération (*.genmodel)

Comporte les informations de génération de code.

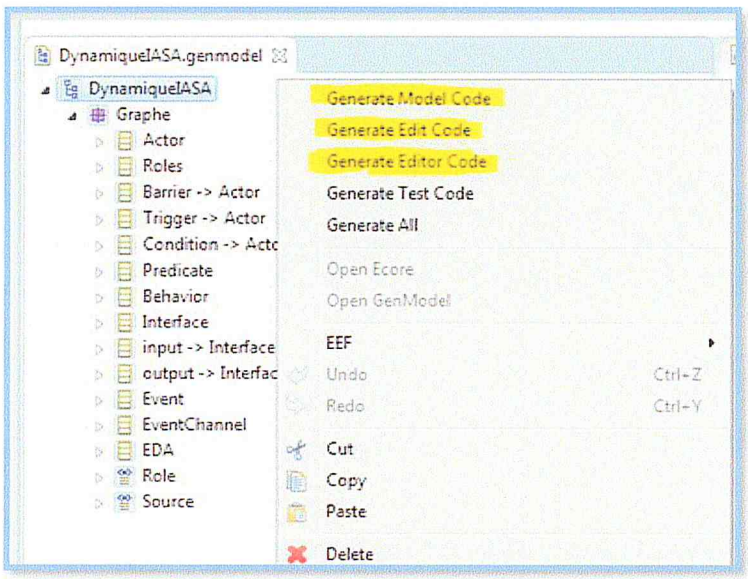


Figure 22: Le modèle de génération de l'éditeur Dynamique IASA

3.3. Le modèle graphique (*.gmfgraphe)

Contient les informations relatives aux éléments graphiques

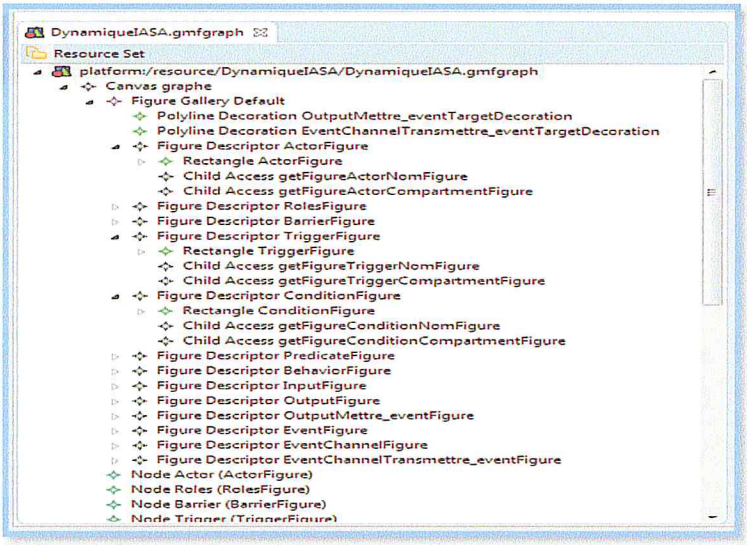


Figure 23: Le modèle graphique du l'éditeur Dynamique IASA

3.4. Le modèle d'outils (*.gmftool)

Permet d'élaborer la palette d'outils de l'éditeur, de même que les icones et les menus.

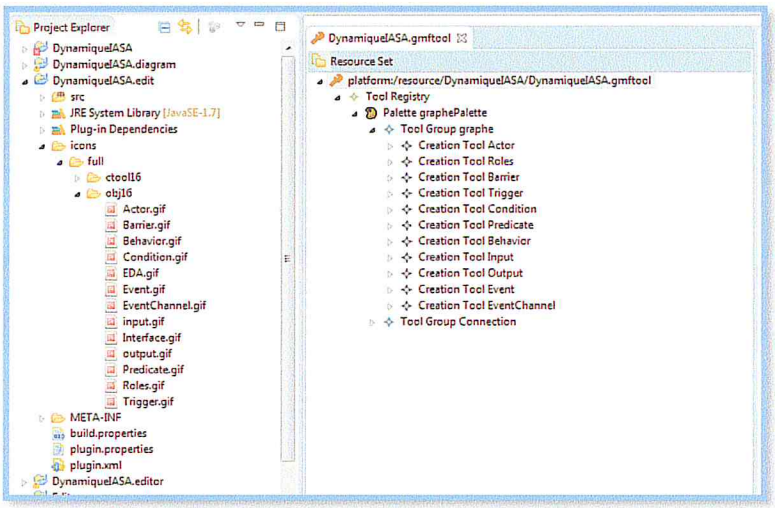


Figure 24: le modèle d'outils de l'éditeur Dynamique IASA

3.5. Le modèle d'association (*.gmfmap)

Permet de faire les relations entre le modèle de entre le modèle de domaine, le modèle graphique, et le modèle d'outils c'est-à-dire faire le lien entre un concept du modèle de domaine, sa représentation graphique et sa représentation sur la palette.

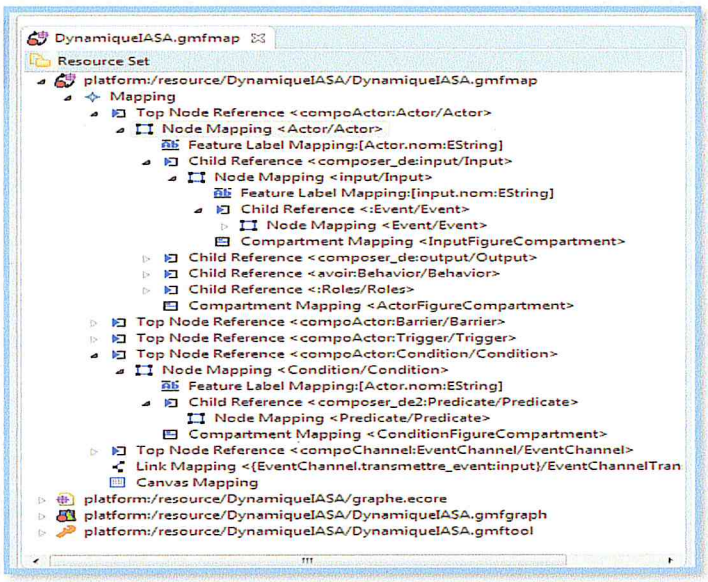


Figure 25: Le modèle d'association du l'éditeur Dynamique IASA

3.6. Le model de génération de l'éditeur graphique (*.gmfgen)

Définie les détails de l'implémentation pour la phase de génération code.

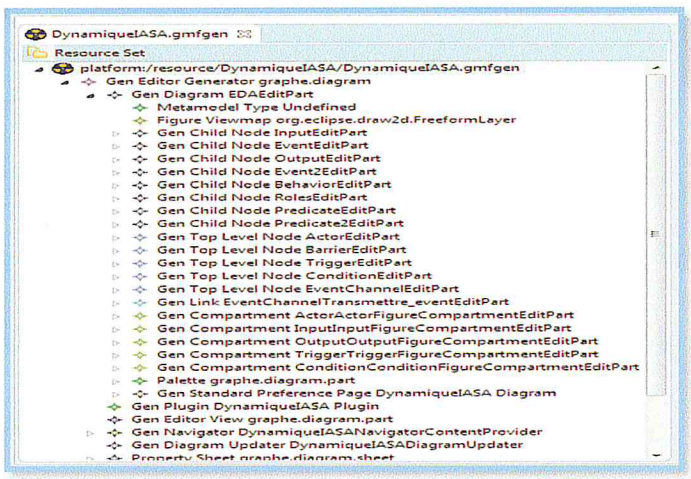


Figure 26:Le modèle de génération du l'éditeur Dynamique IASA

4. Présentation de l'application

4.1. Objectifs

L'idée principale derrière le développement de cet éditeur graphique de model Dynamique IASA est de donner une aide à la représentation aux architectes pour représenter une architecture logicielle dynamique selon notre approche.

L'éditeur fournit un outil de conception assistée qui a comme entrée les différents concepts de notre architecture et donne comme sortie un fichier XML[6] (Extensible Markup Language) représentant notre modèle.

4.2. Fonctionnalités

L'éditeur reprend l'interface général d'Eclipse, son fonctionnement général est assez intuitif pour permettre une prise en main rapide. Il est composé de cinq vues principales

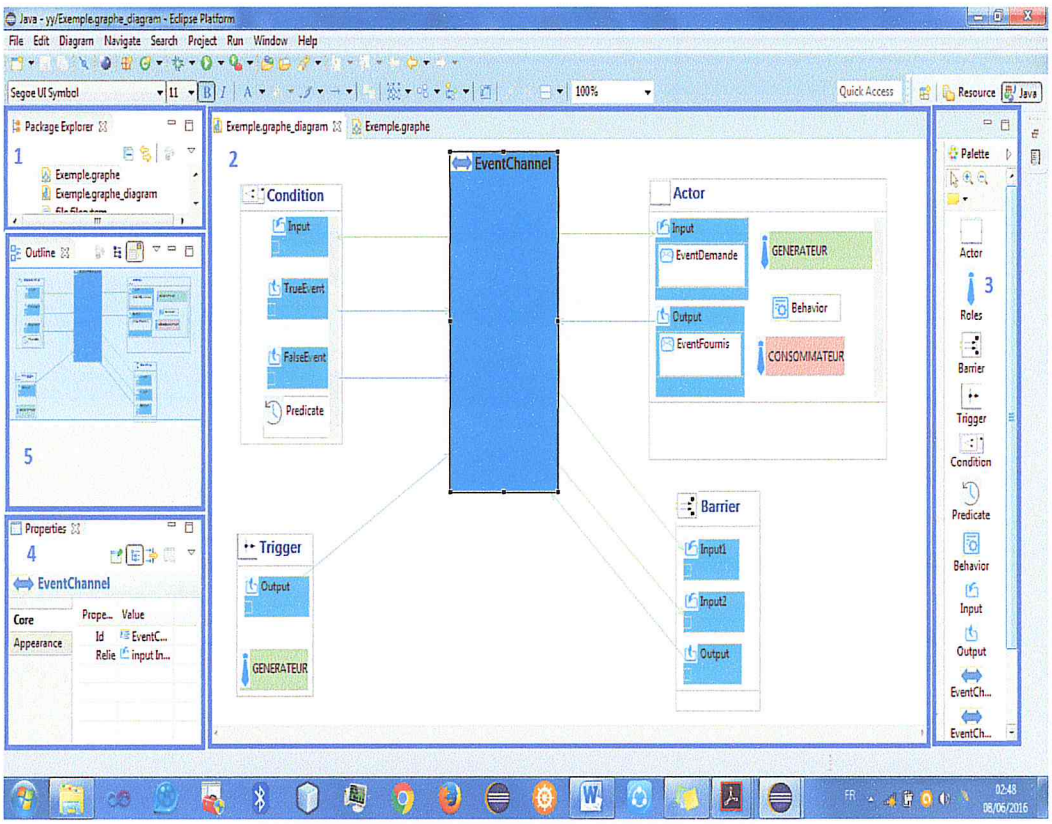


Figure 27: Vue d'ensemble de l'éditeur

[6] http://www.w3.org/xml/

1. La vue Explorateur de projets : la vue physique du modèle, utilisée pour gérer des projets au niveau du système de fichiers et donne accès aux divers modèles disponibles.
2. La vue éditeurs de modèle : un espace vide permettant la représentation et la modification graphiques du modèle.
3. La palette : contient les éléments graphiques de création des concepts.
4. La vue de propriétés : permet le renseignement des propriétés des éléments du modèle également celles non représentées sur le graphique.
5. La vue outline : donne un aperçu général du modèle (lecture seule)

La création d'un nouveau diagramme (fichier*.DynamiqueIASA_diagram) assure la création d'un fichier XML associé (*.DynamiqueIASA), qui est conforme à notre méta-modèle.

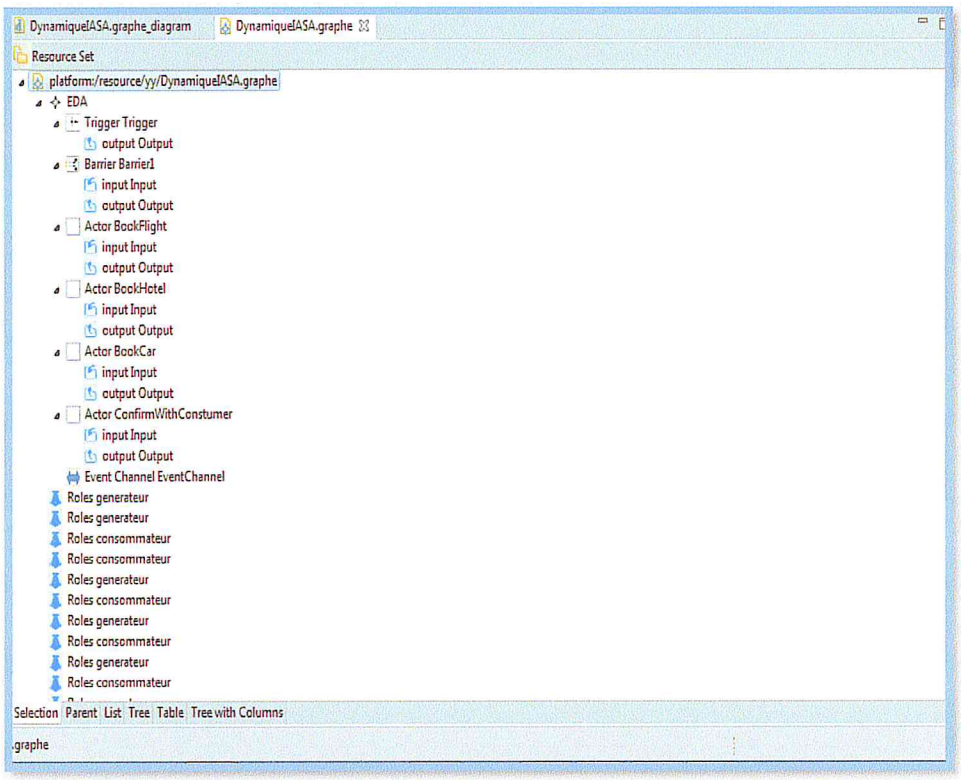


Figure 28: Le fichier XML vu par notre éditeur

4.3. Implémentation d'un exemple

Notre exemple est **(procédure de réservation des vols, hôtels et locations de voiture)** traité dans le troisième chapitre.

Comme on a déjà mentionné dans le troisième chapitre, on a besoins des concepts suivants :

- Des acteurs: **Receive-Itinerary-Request , Book-Flight, Book-Car, Book-Hotel, Aggregate-Booking, Confirm-with-constumer.**
- Des événements : **initialized, flightFound, hotelFound,CarFound, confirmed, Bookings.**
- Des interfaces : **inputs** et **outputs.**
- EventChannel.

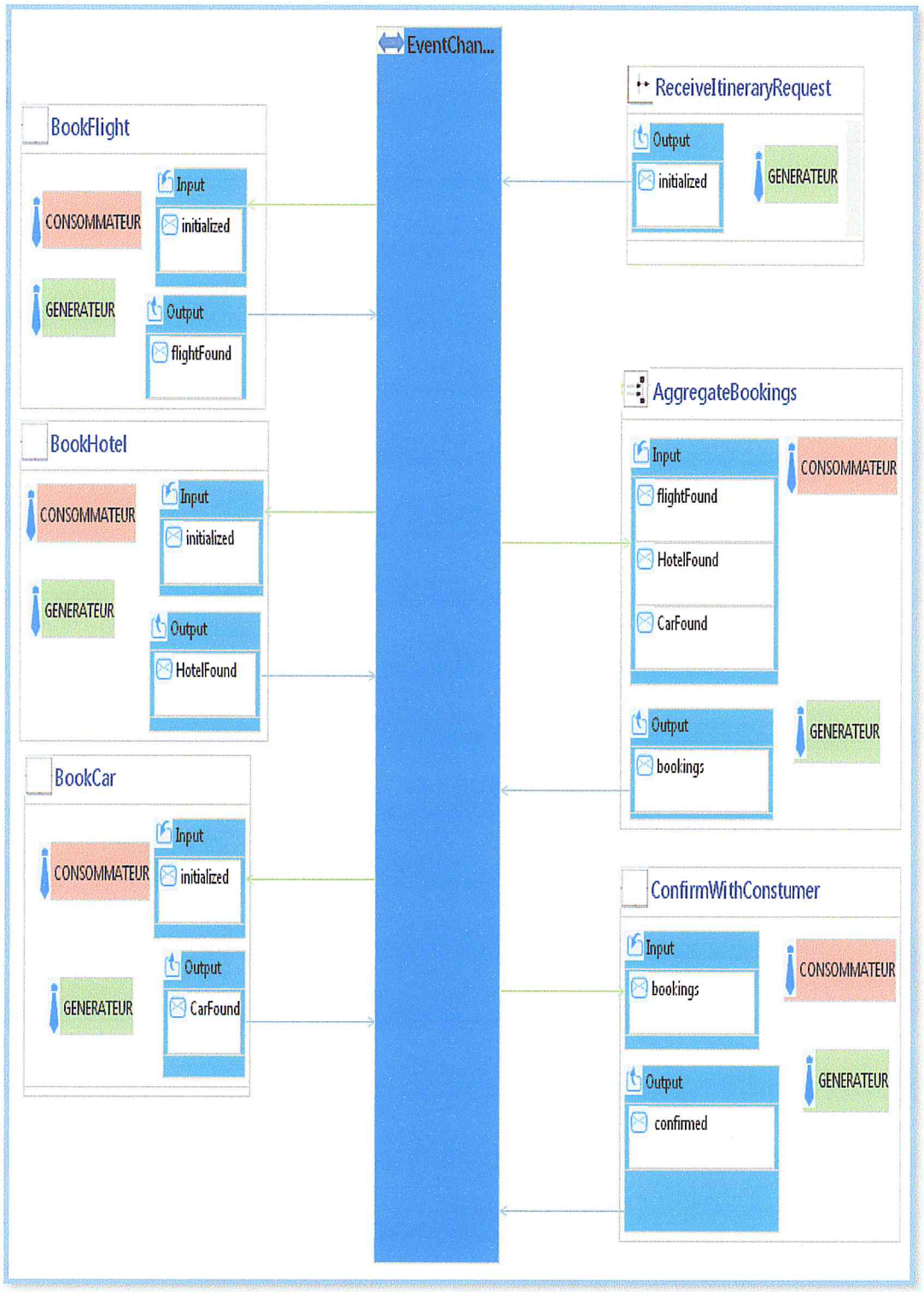


Figure 29: Diagramme de réservation voyage

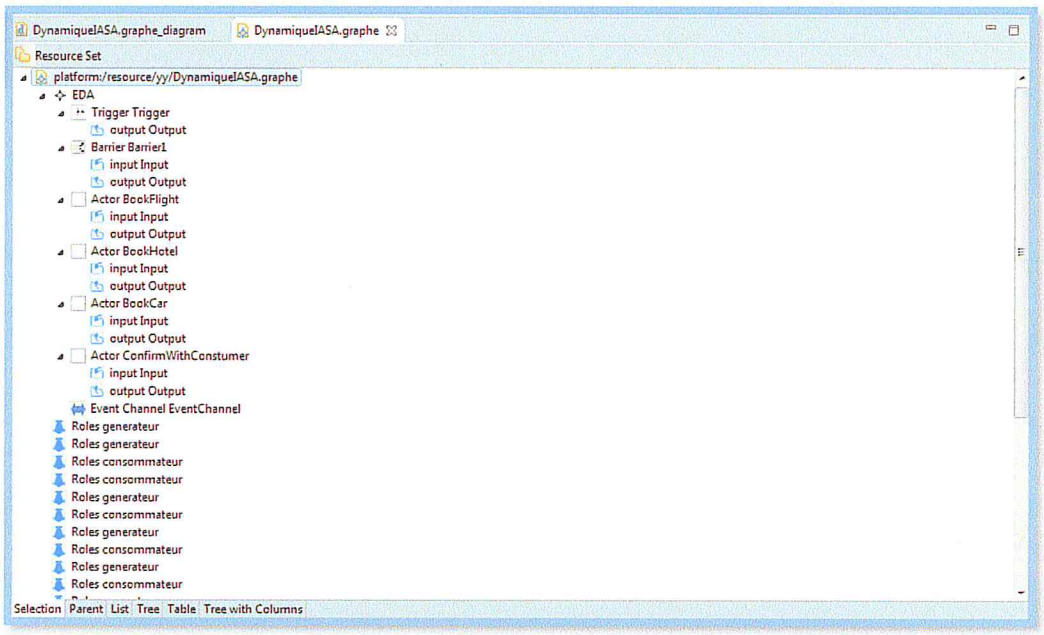


Figure 30 : le fichier XML obtenue

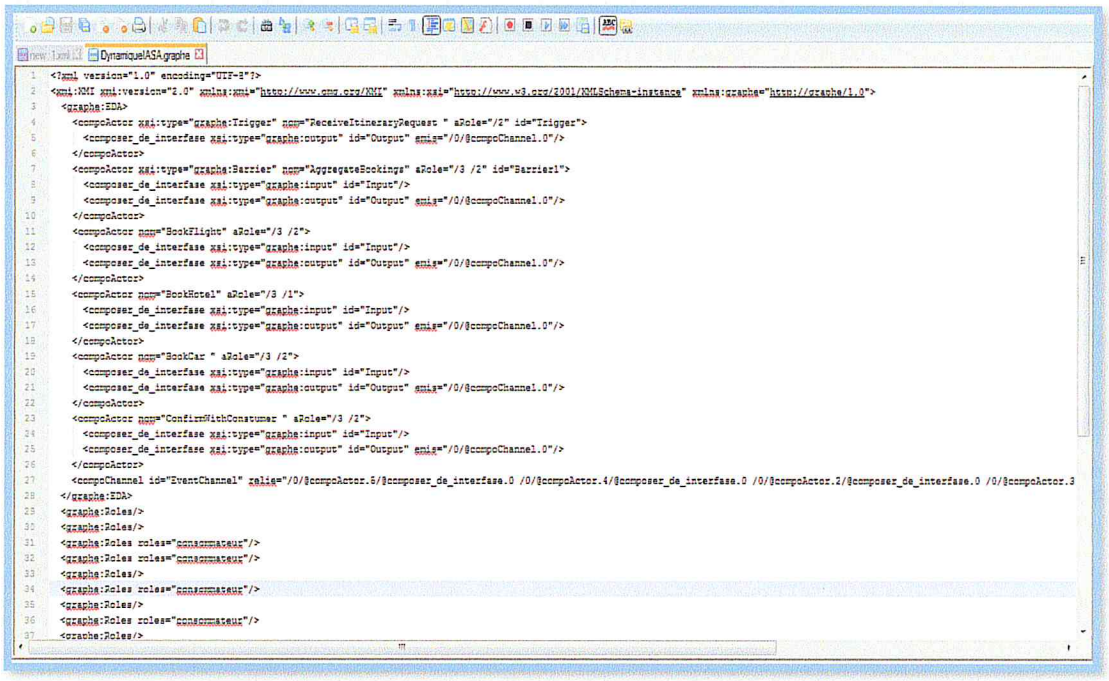


Figure31: Le fichier XML obtenue (vu par l'éditeur de texte)

5. Conclusion

Le présent chapitre constitue le dernier chapitre de ce mémoire, il a été dédié à la représentation des fonctionnalités de notre éditeur graphique **DynamiqueIASA**, l'environnement et outils de travail ainsi que les étapes de l'implémentation ont été aussi abordées.

Conclusion générale

La spécification d'architecture logicielle est une étape importante dans le cycle de vie de développement d'un logiciel et un facteur critique dans sa réussite. En effet l'architecture logicielle permet de mieux contrôler la complexité du logiciel en se détachant des détails techniques propres à l'environnement de développement, en d'autre terme la définition de l'architecture d'un système correspond à l'établissement du plan de construction de ce système. Ce plan peut être mené à changer durant la durée de vie d'un logiciel, on parle alors des architectures logicielles dynamiques. Ce caractère dynamique introduit des difficultés supplémentaires concernant la description. En effet, Dans le cas d'une architecture statique, la description consiste à "simplement" énumérer les différents composants et connexions qui la constituent. Cette approche est inadaptée pour la description des architectures dynamiques dont la configuration, par définition, change pendant l'exécution et dont le comportement des composants peut être changé tout au long du cycle de vie de l'application.

Dans ce travail, nous avons conçu une nouvelle approche de spécification d'architecture logicielle dynamique nommée **DynamiqueIASA,** en se basant principalement sur les architectures orientée événements. Nous avons étendu l'approche IASA pour supporter le comportement dynamique des composants en se basant sur l'architecture orientée événement pour arriver à une adaptation dynamique qui peut être spécifié d'une manière graphique et la transformation de cet ensemble en un ADL par l'extension de X3ADL (ADL de l'approche IASA).

Une première nécessité été de faire une étude sur les architecture logicielles statique et dynamique et les différents concepts proposés par les deux approches. Cette étude nous a permet de mettre en lumière les différentes définitions et notions nécessaires pour aborder notre problématique.

Par la suite, nous avons mené une étude détaillée sur certains ADLs dynamiques. L'étude bibliographique nous a permis de tirer l'aspect dynamique de chaque ADLs étudié, par cette étude nous avons remarqué que la majorité des solutions existantes comptent souvent sur le faible niveau, impératif, des langages non formelles ou ne proposent pas des mécanismes pour le support de l'adaptation comportementale contrairement à la reconfiguration structurelle qui est bien prise en charge par ces

ADLs, ce qui nous a amené à la fin de proposer une nouvelle approche pour remédier à ces problèmes.

Notre approche est basé sur l'architecture orientée événement (EDA), cette architecture semble être une bonne base pour réduire la complexité de l'évolution dynamique des architectures. Cette architecture est très flexible, évolutive, d'une part et d'autre part, l'architecte est supportée par des outils qui apprivoisent la nature faiblement couplée de ces architectures et les rendent maniable et analysable.

Nous avons utilisé dans ce travail l'approche IASA (Integrated Approach for Software Architecture). IASA propose un ADL nommé X3ADL extensible et souple ce qui nous permet d'enrichir et d'étendre facilement cet ADL pour supporter les différents types d'évolution de l'exécution et de l'adaptation dynamique. Dans notre travail nous avons développé l'approche IASA afin de supporter la spécification du comportement dynamique, selon un modèle à un haut niveau d'abstraction basé sur les architectures orientées évènements.

Pour arriver à développer notre éditeur graphique, il été nécessaire de concevoir un méta-modèle générique qui combine le méta-modèle de IASA avec le méta-modèle de l'architecture orientée événement, ce méta-modèle comporte tous les concepts définie dans notre approche. Cet éditeur graphique a été développé sous l'environnement Eclipse GMF, qui est en fait, un plug-in Eclipse, il faut préciser qu'une approche plug-in- peut faciliter l'adoption des utilisateurs, parce que les utilisateurs sont déjà probablement familier avec l'interface du framework dont le plug-in est basé en particulier un framework largement utilisé, comme Eclipse.

Ce travail peut être amélioré en future, en intégrant l'aspect graphique statique (composants, connecteurs, ports…) de l'approche IASA qui est déjà supporté par un autre editeur graphique (IASAStudio) pour le combiner avec l'aspect dynamique que nous avons développé , donc nous pouvons imaginer notre IDE comparant deux cadres, un pour l'aide à la spécification de l'aspect statique (tout ce qui concernes les composants , connecteurs, ports..) et l'autre pour la spécification de l'aspect dynamiques(Events, condition…..).

Bibliographie

1. **P, Clements.** *Documenting Software Architectures: Views and Beyond. Addison-Wesly Publishing, Reading, Massachusetts.* 2003.

2. **Djikstra.** *The structure of the multiprogramming system (ouvrage),* Communications of the ACM, 1972.

3. **Parnas.** *On the criteria to be used in decomposing systems into modules(ouvrage),* Communications of the ACM, 1972.

4. **Chafia, Bouanaka.** *Conception d'un ADL pour les Applications Distribuées et Mobiles,basé Logique de Réécriture(THÈSE De Doctorat en Sciences Informatique).* Université Mentouri de Constantine , 2010.

5. **Budgen, Pearl Brereton and David.** *Component-based systems : A classification of issues (ouvrage),* 2000.

6. **Mohamed, HADJ KACEM.** *Modélisation des applications distribuées à architecture dynamique : Conception et Validation (thèse de doctorat).* Paul Sabatier,Sfax : l'Université Toulouse III et Faculté des Sciences Economiques et de Gestion - Sfax, 13 Novembre 2008.

7. **Boasson, Maarten.** *The artistry of software architecture (ouvrage),* EEE Software, 12(6) :13–16, 1995.

8. **Thierry Coupaye, Romain Lenglet, Mikael Beauvois, Eric Bruneton, and Pascal Dechamboux.** *Composants et composition dans l'architecture des systèmes répartis (ouvrage).* Octobre 2001.

9. **David, Garlan.** *Software architecture(ouvrage).* New York,USA : ACM Press, 2000.

10. **Vladimir , Issarny, Luc , Bellissard et Riveill, Michel.***Component-based programming of distributed applications (Article de journal),* 2000.

11. **David C. Luckham, James Vera, and Sigurd Meldal.** *Three concepts of system Architecture(rapport).* Stanford, CA, USA , 1995.

12. **Djamel, Bennouar.** *« Une approche intégrée pour l'architecture logicielle »,(thèse en vue de l'obtention de diplôme docteur d'état en informatique).* El Harrache : ESI, 2009.

13. **Adel Smeda, Mourad Oussalah, and Tahar Khammaci.** *A multi-paradigm apapproach to describe complex software system (ouvrage),* WSEAS Transactions on Computers, October 2003.

14. **Szyperski, Clemens.** *Component Software - Beyond Object-Oriented Programming (ouvrage),* Addison-Wesley, Novembre 2002.

15. **Olivier, Goaer.** *De l'adaptation des composants logiciels vers leur évolution (ouvrage).* Mastersthesis : Laboratoire d'Informatique de Nantes Atlantique, 7 septembre 2005.

16. **Reiko Heckel, Alexey Cherchago,Marc Lohmann.** *A formal approach to service specification and matching based on graph transformation(ouvrage),* Electronic Notes in Theoretical Computer Science, 2004.

17. **Francisco José, MOOMENA.** *Modélisation Des Architectures Logicielles Dynamique(thèse de doctorat),* 2007.

18. **Nassima, SADOU-HARIRECHE.** *Evolution Structurelle dans les Architectures Logicielles à base de Composants(thèse de doctorat).* Université de Nantes : Thèse de doctorat, 2008.

19. **Chardigny, Sylvain.** *Extraction d'une architecture logicielle à base de composants (ouvrage),* 2010.

20. **GUESSOUM, Dalila.** *SPECIFICATION HAUTEMENT FLEXIBLE D'ARCHITECTURE LOGICIELLE(MEMOIRE DE MAGISTER).* BLIDA : UNIVERSITE SAAD DAHLAB DE BLIDA, 2012.

21. **ACCORD, Projet.** *Etat de l'art sur les Langages de Description(ouvrage),* 2002.

22. **SEMEDA, A.** *constribution à l'élaboration d'une métamodélisation de discription d'architecture logicielle(ouvrage),* 2006.

23. **Abwod G.D, Allen R, and Garlan D.** *Using style to understand descriptions of software architecture (ouvrage).* New York, NY, USA, ACM : ACM SIGSOFT Symposium on Foundations of Software, 1993.

24. **G. D. Abowd, R. Allen, and D. Garlan.** *Formalizing style to understand descriptions of software architecture (ouvrage),* ACM Transactions on Software Engineering Methodologies, 1995.

25. **D, Le Métayer.** *Describing Software Architecture Styles Using Graph Grammars (ouvrage),* IEEE Transactions on Software Engineering, 1998.

26. **Gamma E, Helm R,Johnson R,and Vlissides J.** *Design Patterns (ouvrage).* Addison-Wesley : Professional Computing Series, October 1994.

27. **Yassmin, MANCER.** Introduction à l'Architecture logicielle (coursde l'architecture logicielle). Blida : Universite Saad Dahleb, 2015.

28. **Schoenmakers et Salzman , chris .** *dunamics and mobility in software architecture* ,1997.

29. **Wermelinger, Michel.** *Specification of software architecture reconfiguration(ouvrage),* Juin 1999.

30. **ALTI, Adel.** *Coexistence de la modélisation à base d'objets et de la modélisation à base de composants architecturaux pour la description de l'architecture logicielle (thèse du doctorat).* Sétif-Algérie : Université Ferhat Abbas de Sétif - UFAS, 22 Juin 2011.

31. **Mohammed Karim, Guennoun.** *Architectures dynamiques dans le contexte des applications à base de composants et orientées service(thèse du doctorat).* universitéTOULOUSE III : universitéTOULOUSE III Networking and Internet Architecture, 2006.

32. **Hoare, C. A. R.** *Communicating Sequential Processes (ouvrage).* s.l. : Prentice-Hall, 1985.

33. **MEDVIDOVIC, N, ROSENBLUM , D.S et TAYLOR, R.N.** *A language and environment Architecture-based Software development and evolution (Rapport Dans Proceeding of the 21st international conference on Software engineering),*1999.

34. **MEDVIDOVIC, N, TAYLOR , R et WHITEHEAD, E.***Formal modeling of software arrchitectures at multiple levels of abstraction, (Dans Proceedings of the 1996 California Software Symposium),*California , 1996.

35. **Seinturier, L, et al., et al.** *A component model model engineered with components and aspects(ouvrage),*2006.

36. **Björnander, Stefan.** *Architecture Description Languages (Rapport).* Mälardalen University : The Department of Computer Science and Electronics, 2007.

37. **Benoit Combemale, Sylvain Rougemaille.** *Les Langages de Description d'Architectures (rapport de MASTER 2 RECHERCHE SLCP - MODULE RTM),* 2005.

38. **Hugh Taylor, Angela Yochem, Les Philips, and Frank Martinez.** *Event-Driven Architecture :How SOA Enables the Real-Time Enterprise (ouvrage),* Addison-Wesley, 2009.

39. **Jehan, Bruggeman.** *Prototypage d'une application basée sur les Architectures Orientées Évènements :Application à la gestion d'essais cliniques(thème d'ingérierie informatique),* UNIVERSITAIRE EUROPÉEN BRUXELLES, 2010.

40.Jeremy S. Bradbury. Organizing Definitions and Formalisms for Dynamic Software Architectures: School of Computing, Queen's University, Canada,2004.