Approche de multimodélisation pour une meilleure gestion de l'évolution des logiciels

Adeel Ahmad, Henri Basson, Laurent Deruelle, Mourad Bouneffa

Chapitre 13

Approche de multimodélisation pour une meilleure gestion de l'évolution des logiciels

13.1. Introduction

Le développement à grande échelle d'applications distribuées et hétérogènes a donné lieu à des applications de plus en plus complexes et de plus en plus évolutives. Leur évolution est dictée par des mutations technologiques et des attentes de plus en plus exigeantes des métiers des utilisateurs. Les expressions des besoins des applications sont de plus en plus spécifiques incluant un large éventail d'objectifs qualitatifs en termes de sûreté de fonctionnement, de performance ou d'interopérabilité. Ceci a conduit les chargés de l'évolution à mener une gestion tenant à éviter que tout impact des opérations de modification ne puisse provoquer un fonctionnement du logiciel non conforme aux attentes comportementales ou qualitatives de l'application. Une telle gestion nécessite des outils d'aide au contrôle de l'évolution moyennant, entre autres, une analyse prévisionnelle des impacts précédant la mise en œuvre des modifications et puis leur validation une fois qu'elles ont été réalisées. Notre approche part, dans un premier temps, de la modélisation architecturale décrivant l'organisation des constituants de haut niveau composant le logiciel. Cette description est devenue primordiale pour une meilleure conception et construction de tout système logiciel.

En pratique, les effets que peut engendrer une modification du logiciel, ne sont souvent pas limités au seul composant architectural, cible directe de la modification.

Chapitre rédigé par Adeel AHMAD, Henri BASSON, Laurent DERUELLE et Mourad BOUNEFFA.

Ils peuvent se propager à d'autres composants liés, par différentes relations d'interdépendance, au composant modifié[HAS 10, LAV 10]. Ces relations pouvant véhiculer des impacts des modifications entre composants. Or, une description architecturale est une spécification de haut niveau des composants et leurs connecteurs alors que la portée des impacts des modifications concerne l'intégralité des constituants concernés par la modification.

Ainsi, un raisonnement sur l'évolution qui se limite au seul niveau architectural s'avère insuffisant et requiert pour sa complétude une modélisation structurelle fine par son niveau de détail prenant en compte la granularité des variables individuelles des composants logiciels. En outre, l'élaboration d'un bilan qualitatif des impacts des modifications est importante au vu des priorités qualitatives exigées pour un nombre grandissant d'applications. Cette élaboration nécessite l'intégration d'une modélisation qualitative permettant d'estimer, individuellement et globalement les variations qualitatives des composants lors des modifications. L'objectif global étant de fournir à la gestion de l'évolution une analyse exhaustive permettant de mieux estimer les impacts d'une modification avant sa mise en œuvre et de l'aider à dresser un bilan de ses effets une fois que la modification a été réalisée.

Les connaissances acquises en évolution ne peuvent pas se dispenser d'un historique de l'essentiel des évolutions justifiées, décidées, réalisées puis vérifiées. Il est important de pouvoir donc formaliser les connaissances issues de l'expérience afin que l'on puisse facilement les réutiliser chaque fois que les conditions préalables à une évolution sont validées.

Dans tous les cas, quel que soit l'effort investi en raisonnement préalable au changement, une analyse insuffisamment outillée conduirait à un bilan entaché d'erreurs d'appréciation des réels effets du changement. Ces erreurs peuvent concerner la portée des effets, le coût de l'évolution ou son délai de réalisation. L'objectif de notre approche est de minimiser ces erreurs en procédant à une spécification systématique du contexte de chaque cas d'évolution pour y appliquer une analyse soutenue des règles adaptées à ce contexte. Dans la suite, sont présentés les éléments principaux de cette spécification à laquelle sont associées deux types d'analyse : une analyse *a priori* et une analyse *a posteriori* d'impacts des modifications. La première vise à simuler l'opération de modifications pour étudier ses effets et leur propagation à travers l'ensemble des constituants concernés du logiciel et dresser un bilan prévisionnel de ses effets. La seconde analyse consiste en une réalisation de l'opération de modification suivie d'une réapplication de tests permettant de constater ses effets réels pour dresser un bilan effectif de l'évolution.

Le chapitre est organisé de la manière suivante : la section 13.2 introduit la problématique de modélisation et d'évolution des logiciels, nous y rappelons les concepts essentiels des modélisations et de l'analyse d'impact des modifications. La section 13.3 présente le modèle ASCM dédié à la représentation des architectures logicielles

spécifiées à l'aide de langages de description d'architecture. En raison de la taille du présent chapitre la modélisation qualitative est simplement référencée sans y être présentée. La section 13.4 décrit le modèle structurel des composants logiciels. Une mise en correspondance entre le modèle architectural et structurel est détaillée dans la section 13.5. La section 13.6 présente le processus de propagation d'impact suivi par un système à base des connaissances, dans lequel nous définissons des règles génériques pour la réalisation des opérations de modification et pour identifier la propagation de leurs impacts. La section 13.7 décrit l'implémentation d'une plate-forme, basée sur l'environnement Eclipse, pour l'analyse d'impact des modifications. Nous terminons par une conclusion et des perspectives de nos travaux dans la section 13.8.

13.2. Modélisation pour l'évolution des logicielles

L'expression de l'évolution des architectures logicielles nécessite de disposer d'un référentiel, doté de représentation formelle des éléments descriptifs du logiciel. Cette représentation prend en compte les constituants communs aux différentes phases telles que représentées par les principaux modèles du développement des logiciels. Le référentiel est formalisé par des modèles à base de graphes dénommés *Architectural Software Components Model* (*ASCM*), *Software Component Structural Model* (*SCSM*) et *Software Component Qualitatif Model* (*SCQM*). Ceci permet d'une part, la formalisation des composants et des faits associés aux différentes phases de développement, et d'autre part, l'expression des règles adaptées à chaque contexte d'évolution.

Dans ce cadre, le modèle ASCM de haut niveau d'abstraction représente l'architecture d'un logiciel. ASCM [HAS 10] est indépendant de tout langage particulier de description d'architecture ADL (*Architectural Description Language*). Nous retenons, dans ce modèle, les concepts-clés partagés par la plupart des ADL. ASCM est faiblement couplé avec le modèle SCSM [AHM 08], pour représenter les relations existantes entre les composants d'une architecture et ceux des codes sources correspondants. Ce couplage est réalisé par la définition d'une relation dite de projection liant les éléments des deux modèles respectifs ASCM et SCSM. Par exemple, dans le langage AADL, l'interface d'un composant est projeté en une « fonction » dans le langage Ada. Cela se traduit par la création d'une relation de projection entre le composant ASCM « Interface » et le composant SCSM « fonction ». Pour contrôler les variations qualitatives des artéfacts logiciels suite au changement, une mise en correspondance a été également définie entre le modèle SCQM [AHM 10] et le modèle SCSM.

13.3. Modèle des composants architecturaux du logiciel

Le modèle ASCM est défini comme un modèle de description des architectures logicielles indépendamment de tout ADL particulier (figure 13.1), les éléments communs aux ADL sont tels que : un composant est représenté par une interface qui permet

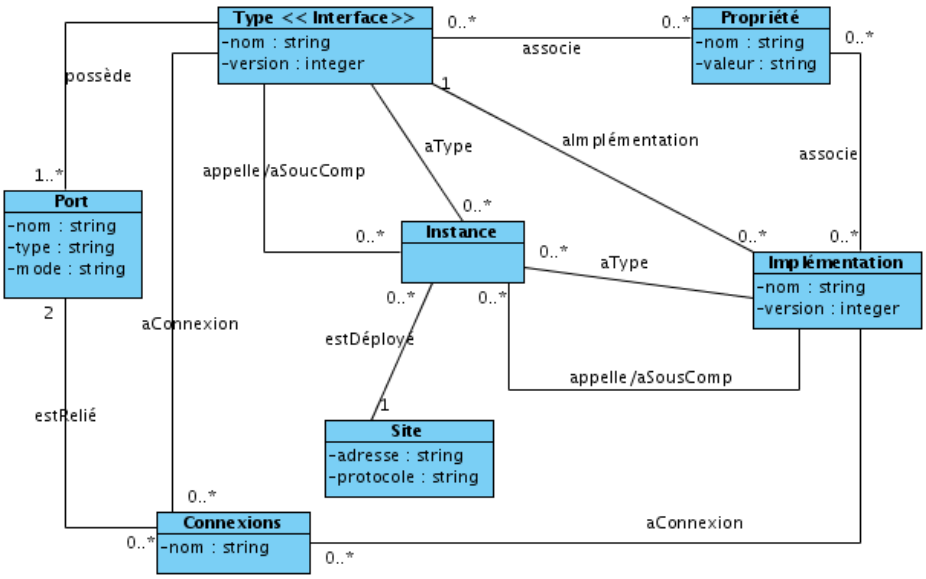

Figure 13.1. *ASCM : classes des éléments partagés par les ADL.*

de spécifier ses ports. Certains langages (Unicon, AADL) définissent les composants en deux parties : l'interface et l'implémentation. Cette séparation dans la définition de composants n'existe pas dans d'autres langages. Par conséquent, dans notre modèle, il est possible d'associer ou non une implémentation à une interface d'un composant. Cependant, la description de l'interface est obligatoire. Nous raffinons l'interface et l'implémentation en proposant de les représenter à l'aide d'une version. Ceci permettra de faire cohabiter plusieurs versions d'un même composant au sein de la représentation de l'architecture. L'interface d'un composant possède au moins un port qui lui permet d'interagir avec les autres composants à travers les connecteurs. Un port peut participer à plusieurs connexions et à un composant peuvent être associées plusieurs propriétés non fonctionnelles décrites dans son interface ou son implémentation. L'interface et l'implémentation d'un composant peuvent être instanciées plus d'une fois dans ASCM. Elles peuvent également avoir des sous-composants qui représentent des instances d'une autre interface ou implémentation d'un composant. L'instance d'un composant est déployée sur un site pour fournir une infrastructure logicielle et matérielle pour son exploitation. Une caractéristique majeure de notre modèle est son extensibilité par raffinement progressif en éléments des niveaux granulaires stratifiés et leurs constituants respectifs sous-jacents. Par exemple, le langage AADL identifie pour les composants les catégories suivantes : système, processus, *thread*, sous-programme ou données. Nous pouvons ainsi étendre la représentation des composants en utilisant le mécanisme d'héritage. Pour d'autres ADL, il est aussi possible d'étendre notre modèle en spécifiant de nouvelles catégories de composants.

L'évolution d'une architecture peut être vue comme un ou plusieurs changements effectués sur un ou plusieurs constituants de l'architecture. Ces modifications impliqueront des effets qui peuvent se propager sur plusieurs éléments architecturaux du fait de l'interdépendance existant entre le composant modifié et le reste de l'architecture. Il devient ainsi primordial de procéder à un examen exhaustif de l'impact de la modification d'un élément architectural en déterminant tous les éléments pouvant être affectés par sa modification.

Nous pouvons identifier deux catégories d'impact suite à des modifications architecturales. La première concerne les effets de la modification sur la structure de l'architecture, tandis que la seconde porte sur les effets comportementaux pendant l'exécution du logiciel. Notre travail focalise dans un premier temps sur l'étude des effets de modification d'un élément architectural qui peut avoir des impacts sur les autres éléments pris en compte par ASCM. Nous identifierons d'abord les éléments directement affectés par cette modification pour procéder ensuite à une application récursive de cette identification des autres éléments concernés. Les impacts comportementaux au niveau du code source correspondant à l'architecture seront également considérés.

La modélisation s'appuie également sur la formalisation de l'ensemble des opérations de modification pouvant affecter les éléments architecturaux. A chaque type d'opération, un ensemble d'invariants est défini pour inclure les conditions préalables à la réalisation de ces opérations et les postconditions attendues suite à la réalisation. Une étape préliminaire précède ainsi l'application de toute opération et permet ainsi le respect des invariants de l'opération. A titre d'exemple, considérons une opération de suppression d'une implémentation d'un composant appelé « cimp ». Avant d'exécuter la suppression, il faudra vérifier la possibilité de supprimer ses connecteurs, ses sous-composants et les sous-composants instanciés à partir de « cimp », autrement dit le cas où « cimp » devient isolé du reste de l'architecture pour pouvoir ensuite le supprimer. Ce cas de suppression ne génère alors pas d'impact sur le reste de l'architecture. Si ces vérifications ne sont pas effectuées avant la suppression du sous-composant, les invariants de l'opération de suppression sont alors violés. La simulation de l'exécution de l'opération de modification se fait en marquant les nœuds et les arcs affectés par sa réalisation. Cette simulation permet de ressortir les vérifications qui devaient être effectuées préalablement à l'exécution de l'opération. Dans un second temps, nous définissons le processus générique de propagation d'impacts déclenché lors de l'application des opérations de modification.

13.4. Modèle structurel des composants logiciels

Dans son mémoire d'habilitation à diriger des recherches, H. Basson [BAS 98] a présenté le modèle structurel générique de développement des logiciels (MSGDL). Ce modèle offre une description couvrant l'ensemble des composants des différentes

phases du développement d'applications. Il englobe, dans une même description structurelle et qualitative les constituants communs aux différents modèles existants de développement du logiciel. Les caractéristiques qualitatives des composants ainsi que leur description sont définies par l'intermédiaire de deux modèles englobés par le MSGDL : le modèle structurel de composants logiciels (MSCL) permettant la représentation structurelle des composants et le modèle qualitatif de composants logiciels (MQCL) pour spécifier les caractéristiques qualitatives associées à ces composants.

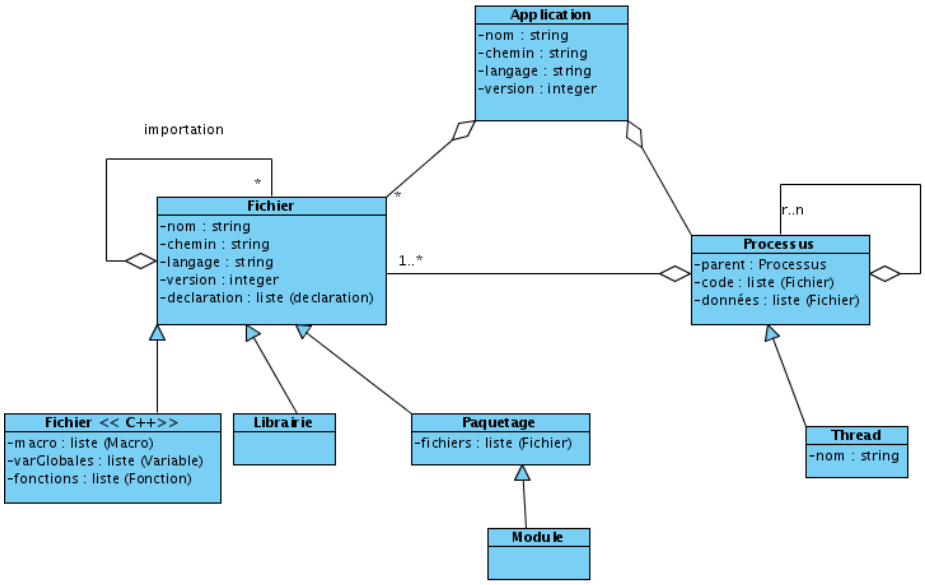

Figure 13.2. *Représentation des composants logiciels de grosse granularité*

Le modèle MSCL est d'un niveau d'abstraction trop élevé pour la représentation du code source. Celui-ci a été raffiné par le modèle SCSM pour représenter de façon exhaustive la structure des composants et leurs relations extraites à partir des codes sources du logiciel. Par rapport à d'autres modèles existants [CHE 90, RAJ 97], ce dernier a l'avantage d'être multilangages (par exemple Ada, C, C++, perl, python, php et Java). Il représente, de manière uniforme, les composants logiciels et leurs interactions indépendamment des langages de programmation utilisés.

SCSM inclut deux types d'éléments : les composants logiciels et leurs relations. Les composants logiciels sont catégorisés selon trois niveaux de granularité. Le premier est celui des composants logiciels appartenant au niveau dit de grosse granularité (figure 13.2). Les composants de ce niveau peuvent être des applications, des

fichiers, des bibliothèques, des paquetages, des modules ou des processus liés par différents types des relations (communication, importation, composition, etc).

La deuxième catégorie de composants est celle de granularité moyenne présentée dans la figure 13.3. Les composants de cette catégorie sont, par exemple, les attributs et les méthodes d'une classe. Nous pouvons trouver deux types de composants à ce niveau :

– les composants de type déclaration, qui représentent les déclarations des structures, de classes ou type de variables.

– les composants de type traitements, désignant les opérations définies dans des fonctions ou des méthodes associées aux classes. Ces dernières sont définies par leur signature et leur corps. La signature d'une méthode indique le type de la valeur de retour, le nom de la méthode et la liste des arguments. Le corps est une suite d'instructions représentée dans la troisième catégorie de granularité fine.

La troisième catégorie concerne les composants logiciels appartenant à un niveau de granularité fine (figure 13.4). Ils caractérisent les instructions et les structures de contrôle du programme. A ce niveau, les composants sont souvent regroupés en blocs. En effet, une instruction caractérise une succession de symboles qui représentent des appels de méthodes ou de fonctions ou des affectations. Le composant « structure de contrôle » représente l'ensemble des structures de contrôle existantes dans les langages procéduraux et orientés objet. Les composants, appartenant à ce niveau de granularité, peuvent être liés à des composants du même niveau de granularité ou d'un niveau de granularité moyenne. La liaison s'effectue par des relations telles que la relation d'appel ou la relation d'utilisation.

Le modèle SCSM offre une représentation fine du logiciel basée sur l'ensemble de ses codes sources, indépendamment de son architecture. Il ne traite cependant pas le logiciel à un niveau d'abstraction plus élevé, à savoir, son architecture. Afin de pouvoir offrir une représentation de l'architecture, nous pouvons procéder selon deux approches :

– la première approche consiste à spécifier les critères de recouvrement d'architecture à partir de la représentation du code source exprimé à l'aide du modèle SCSM. Le recouvrement d'architecture entre dans le cadre des activités de rétro-ingénierie. De nombreux travaux ont été effectués dans ce domaine [CHI 90, FAV 06, MAW 07] et présentent de nombreux défis :

- la multiplicité des sources de données en entrée (code source, bytecode, binaires, librairies, etc.) implique une grande complexité dans la tâche de recouvrement d'architecture. Cette complexité s'accroît dans le cas des systèmes distribués déployés sur des sites géographiquement distants,

- la multiplicité de vues à produire en sortie est également un défi pour les outils de la rétro-ingénierie. En effet, il peut y avoir plusieurs points de vue d'un logiciel en fonction du métier et des compétences des différents intervenants [GRO 00],

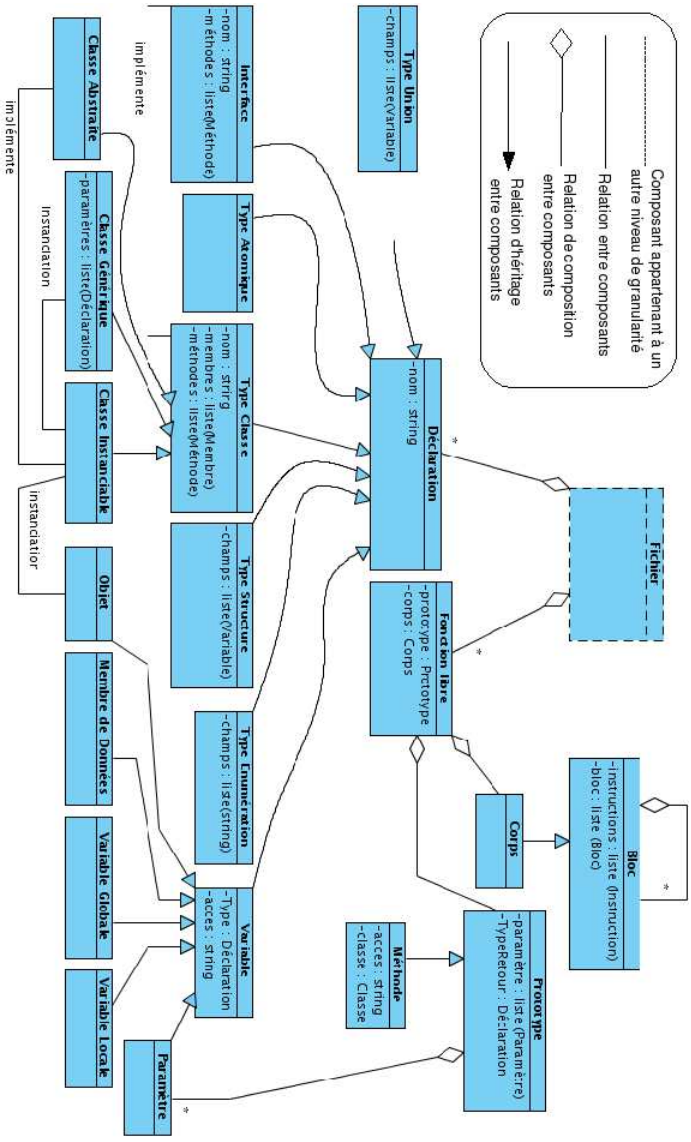

Figure 13.3. *Représentation des composants logiciels de granularité moyenne*

- la plupart des outils de rétroingénierie n'offrent que très peu de souplesse. Ils ne couvrent généralement que certaines fonctionnalités, d'où la difficulté de trouver une manière de couvrir tous les besoins du projet ;

– la seconde approche consiste à : (1) fournir un modèle représentant des architectures logicielles indépendamment des codes sources du logiciel qui sont modélisés à l'aide du modèle SCSM puis (2) de lier le modèle architectural au modèle SCSM par un couplage faible. Cette approche, simple à mettre en œuvre, nous permettra de faire évoluer les deux modèles indépendamment l'un de l'autre. Nous pouvons ainsi étudier l'impact des modifications au niveau code source, au niveau architectural ou sur les deux niveaux.

13.5. *Mapping* entre ASCM et SCSM

L'ensemble des éléments constituant les codes sources d'un logiciel sont représentés par le modèle SCSM. Dans les travaux de L. Deruelle [DER 01], une approche pour la propagation d'impacts a été proposée sur la base de ce modèle. Cependant, cette approche ne tient pas compte de l'aspect architectural d'une application. Le couplage du modèle SCSM et du modèle ASCM consiste à formaliser les liens pouvant exister entre les éléments de ces deux modèles. Les relations entre modèles seront utilisées pour propager l'impact d'une modification réalisée sur la description architecturale vers son code source correspondant et inversement. Les figures 13.5 et 13.6 montrent les relations établies entre les éléments des deux modèles.

Un composant ASCM est défini par son interface et son implémentation. Au niveau de la grosse granularité, ceux-ci se projettent dans le code source d'un logiciel sous la forme de fichiers, contenant l'ensemble des instructions réalisant les éléments de l'architecture. Ces instructions sont structurées généralement sous la forme de fonctions ou méthodes (qui est un cas particulier d'une fonction définie dans une classe). Nous considérons que les classes, les interfaces, les structures, les méthodes et les fonctions reflètent l'interface et l'implémentation des composants ASCM. Tous les éléments de notre modèle ASCM peuvent être projetés vers SCSM. Pour des raisons de clarté, nous ne montrons que les projections des éléments « interface du composant » et « implémentation du composant » sur les niveaux de grosse et moyenne granularité. Notre *mapping* s'applique également sur les autres éléments de ASCM tels que le port qui est projeté en un paramètre dans le modèle SCSM et la propriété qui est projetée en une variable ou une suite d'instructions dans le niveau de granularité fine.

Les langages de description architecturale proposent généralement une syntaxe permettant de spécifier le nom de l'implémentation en termes de codes sources du composant. Dans le cas où le langage ne dispose pas d'une syntaxe particulière pour spécifier l'implémentation, une analyse de code source sera nécessaire pour établir le lien de projection entre les composants ASCM et leur implémentation en code source. L'analyse permet ensuite de fournir des annotations dans le code source où chaque annotation peut désigner le type de composant ASCM comportant des attributs. Ceci permet de lever toute ambiguïté lorsque plusieurs composants de même type existent dans la même architecture.

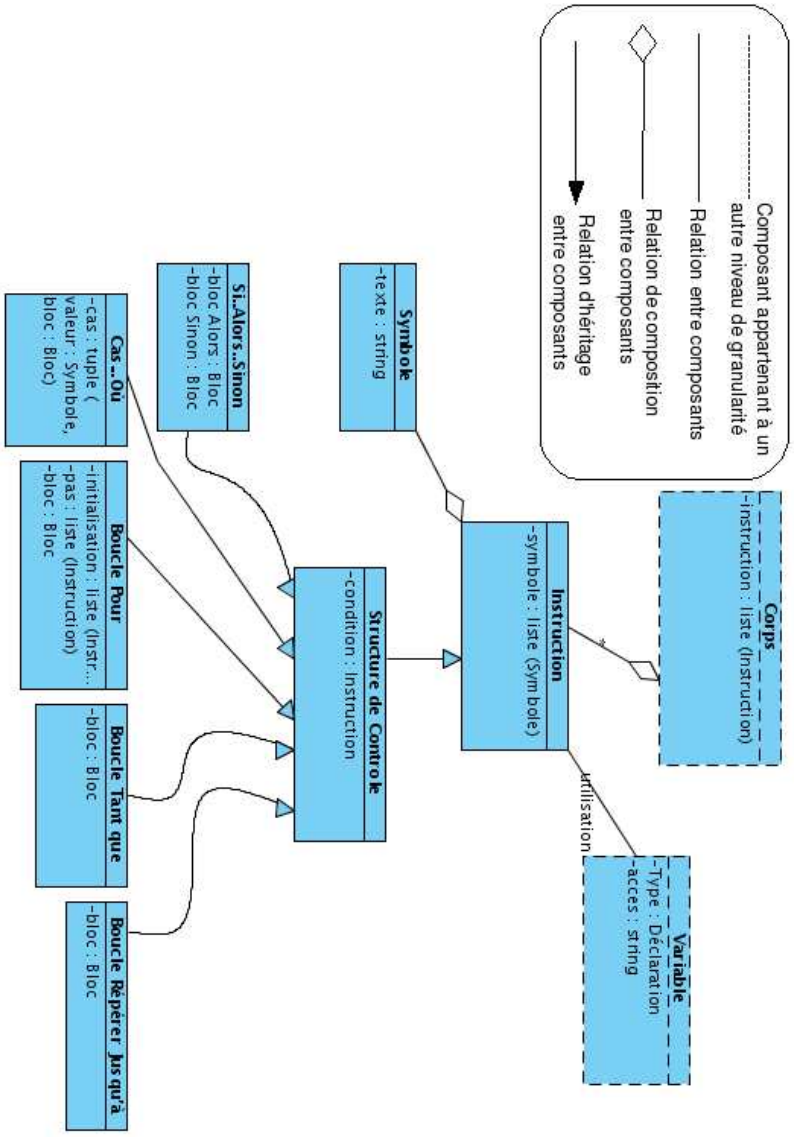

Figure 13.4. *Représentation des composants logiciels de granularité fine*

A titre d'exemple, une annotation $@Port(name = portA, param = paramM)$ peut être définie, au niveau d'une méthode, afin d'indiquer que le paramètre $paramM$ de la méthode est associé au port ASCM ayant le nom $portA$. SCSM et ASCM permettent ainsi d'identifier la propagation l'impact d'une modification, effectuée soit au

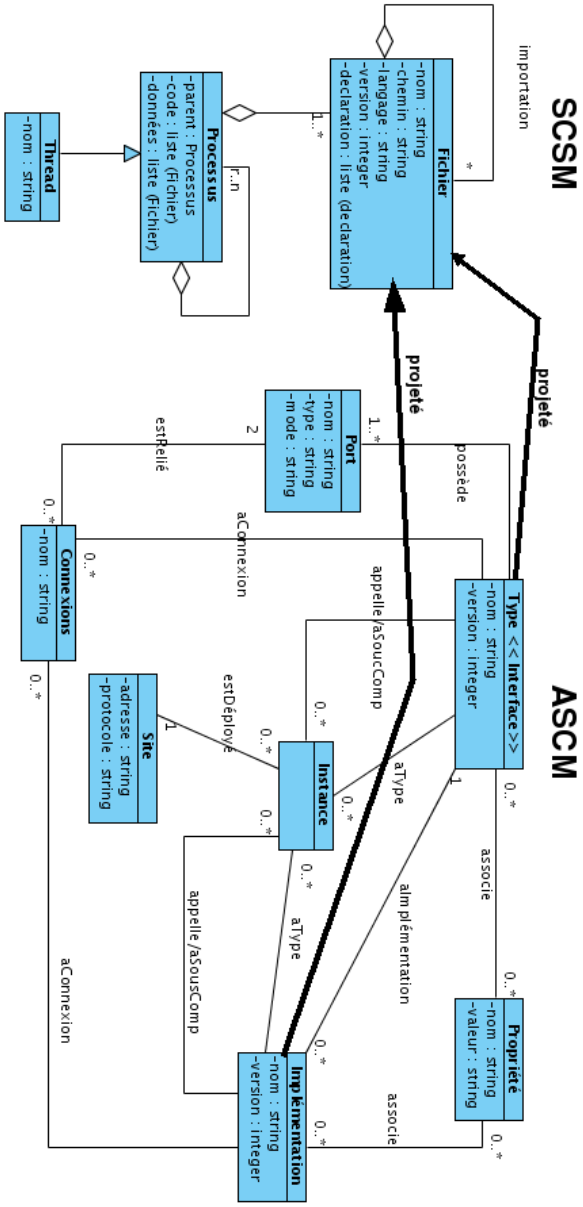

Figure 13.5. Mapping *entre ASCM et SCSM : niveau grosse granularité*

niveau de l'architecture (ASCM), soit au niveau du code source (SCSM), respectivement sur le code source et l'architecture. Les modèles ASCM et SCSM faiblement couplés par la relation de projection, peuvent être utilisés indépendamment.

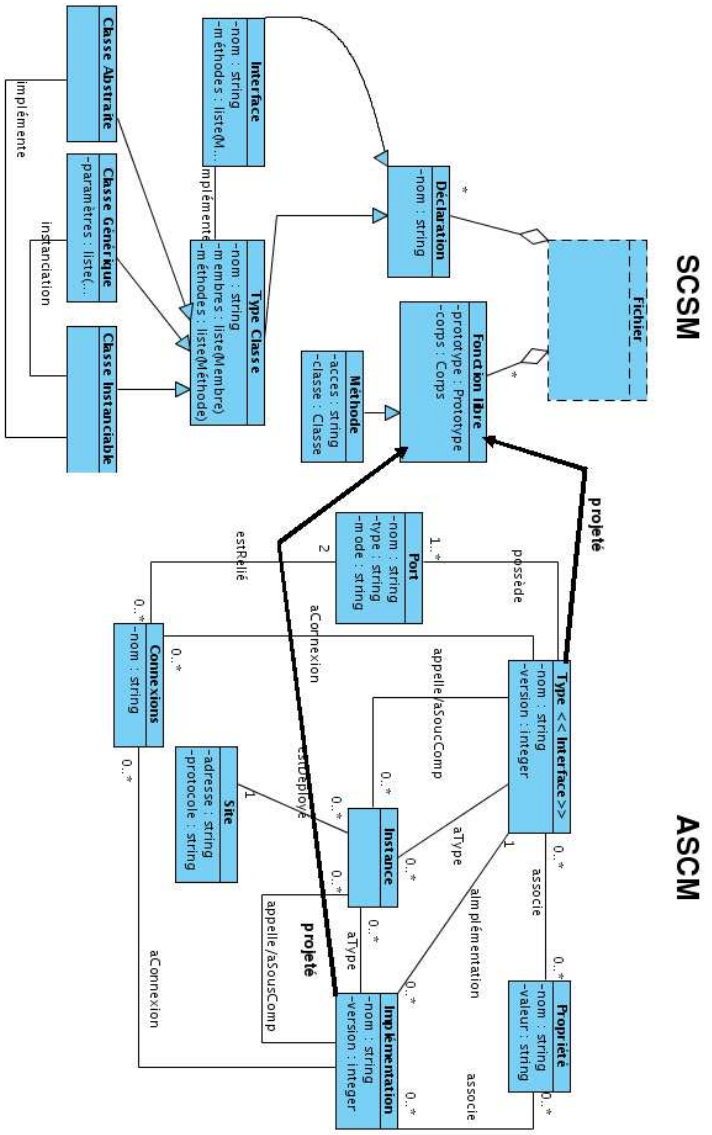

Figure 13.6. Mapping *entre ASCM et SCSM : niveau moyenne granularité*

La figure 13.7 illustre un exemple de *mapping* entre ASCM et SCSM. La partie (A) de la figure correspond à la description architecturale exprimée en AADL. La partie (B) représente le graphe associé à (A). La partie (C) correspond au code source du sous-programme spécifié dans l'architecture et enfin la partie (D) correspond au

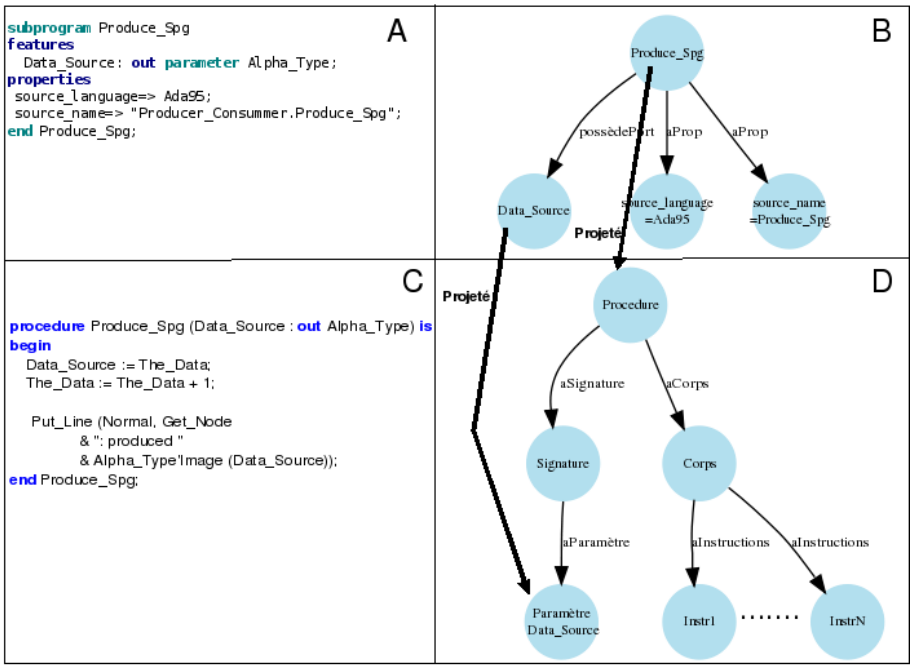

Figure 13.7. *Exemple de* mapping *entre ASCM et SCSM*

graphe associé à (C). Le sous-programme $Produce\_Spg$ possède deux propriétés : la première, $source\_language$, indique que ce sous-programme est exprimé dans le langage Ada dans le code source ; la deuxième, $source\_name$, indique que le fichier contenant le code source est $Producer\_Consumer$ et que la fonction ou la procédure associée au sous-programme est $Produce\_Spg$. Les liens entre les deux représentations sont ainsi établis moyennant la relation de projection.

13.6. Processus de propagation d'impact des modifications

Un changement survenu sur un constituant architectural ou son implémentation en termes de code source peut avoir un impact sur la globalité du logiciel. Afin d'étudier les effets de chaque modification il est nécessaire de disposer d'un processus de contrôle des effets locaux de l'impact et à l'identification de sa propagation aux parties affectées par la modification.

13.6.1. *Approche basées sur les règles*

Notre approche consiste à analyser l'impact au niveau d'un composant logiciel en définissant des règles génériques pouvant s'appliquer indépendamment des langages de développement. Ces règles propagent l'impact de la modification du niveau architectural vers le code source correspondant.

L'intérêt d'utiliser une approche à base de règles réside dans la séparation de la partie fonctionnelle d'une application de celle gérant son évolution. Les règles d'évolution et le code applicatif sont spécifiés séparément. Ces règles sont écrites par des experts du domaine et peuvent être modifiées pendant le processus de propagation d'impact. Elles sont manipulées par un système expert.

Le système expert développé est constitué de trois composants : la base de faits, la base de règles et le moteur d'inférence. La base de faits constitue la mémoire de travail du système à base de connaissances. Elle contient l'ensemble des faits permettant de déclencher les règles d'analyse d'impact. Il s'agit de la partie dynamique de la base de connaissances. Les faits représentent les nœuds et les arcs du graphe associés aux éléments architecturaux représentés par le modèle ASCM. Ceux-ci sont ajoutés à la base de faits lors de la construction du graphe d'architecture et ils peuvent être modifiés au fur et à mesure par le moteur d'inférence en déclenchant des règles sur ces faits. Les règles permettent l'ajout, la modification ou le retrait d'un ou plusieurs faits.

De manière simplifiée, le moteur d'inférence applique à la base de faits toutes les règles non marquées. Une règle est marquée chaque fois qu'elle a été déclenchée. Initialement, toutes les règles sont non marquées. L'application des règles permet d'obtenir une nouvelle base de faits mise à jour avec les nouvelles contraintes introduites. Les règles non marquées sont ensuite déclenchées par le moteur d'inférence, pour modifier effectivement la base de faits et devenir ainsi marquées.

13.6.2. *Régles de propagation d'impact*

Ces règles décrivent l'exécution des opérations d'évolution sur les éléments des modèle ASCM, SCSM et SCQM. Elles permettent de modifier la base de faits en exécutant les opérations de modification sur ces éléments. Le déclenchement du processus de propagation se traduit par l'insertion d'un événement. Des règles de modification sont associées à chaque opération de modification. La prémisse de la règle représente une liste de faits devant être vérifiés afin d'exécuter la règle associée à l'opération de modification. Un exemple de faits à vérifier dans le cas de la suppression d'un composant « C » est l'absence de tout sous-composant ayant le type « C ». Cette prémisse est donnée sous la forme d'une précondition et d'un invariant à respecter avant l'opération. Le résultat de la règle est une liste d'actions à opérer. Celle-ci est donnée sous la

forme d'une postcondition et d'un invariant. Le non-respect de la prémisse de la règle implique que l'opération décrite par la règle ne peut pas être exécutée. Plusieurs cas de figures se présentent :

– si la précondition et l'invariant sont vérifiés alors l'opération sera exécutée sans engendrer une propagation d'impact sur le reste de l'architecture ;

– si la précondition n'est pas vérifiée, l'opération n'est pas réalisée et il n'y a pas d'impacts sur le graphe. Cela correspond à des opérations interdites comme vouloir ajouter un composant déjà existant ;

– si la précondition est vérifiée mais au moins un des invariants ne l'est pas alors l'opération est exécutée et il y a un processus de propagation d'impact. En effet, l'invariant de la règle peut à son tour déclencher le marquage des éléments concernés. Ce cas est le plus répandu et le plus intéressant pour le processus de propagation d'impact. Supprimer un composant qui a des sous-composants ou supprimer un port utilisé par un connecteur sont des exemples illustrant ce cas de figure.

Le marquage des éléments affectés par la modification dépend de la nature de la relation qui les relie. Il consiste à indiquer que l'élément marqué est affecté par l'opération de modification réalisée. Une modification apportée sur un élément de l'architecture est répercutée sur son graphe selon un processus de propagation. La propagation d'impact entre un nœud du graphe et son voisinage dépend de la nature des relations entrantes et sortantes de ce nœud ainsi que de l'opération appliquée sur le nœud. Nous définissons l'assertion $+conductrice(op, e, A, B)$ indiquant qu'une relation $e : A \longrightarrow B$ est conductrice d'impact de A vers B après la réalisation de l'opération op. Citons, par exemple, la relation $hasSubComp$ entre un composant A et un sous-composant B qui propage l'impact de modification dans les deux sens de la relation après l'exécution d'une opération de suppression : si A est supprimé, B sera affecté par cette suppression et inversement. La relation « hasSubComp » est dite conductrice de l'impact dans les deux sens de la relation pour l'opération de suppression. Tandis que pour la même relation, l'exécution d'une opération de modification telle que l'ajout d'un port à A n'affecte pas le sous-composant B.

L'assertion $marquer(el)$, définie dans le tableau 13.1, consiste à marquer un nœud ou un arc comme étant affecté par une opération de modification.

L'assertion $marquer(v, op)$ précise que suite à la modification de v par une opération op, les arcs entrants et sortants de v seront marqués et propagent l'impact vers leurs composants source et destination selon la nature de relation que l'arc représente et le sens de la conductivité d'impact pour l'opération exécutée.

Le processus de marquage des nœuds et de propagation s'arrête lorsque toutes les règles ont été déclenchées et qu'il n'existe plus aucun fait candidat. Nous définissons trois règles génériques pour la réalisation des opérations de modification et pour la propagation d'impact.

Assertion	Signification
$marquer(el)$	$\forall el \in E \vee el \in N,\ etat(el) =$*affecté*
$marquer(v, op)$	$\forall e \in E, v_1 \in N : ((+arc(e, v, v_1) \vee +arc(e, v_1 v)) \wedge +conductrice(op, e, v, v_1)) \longrightarrow marquer(e) \wedge marquer(v_1)$

Tableau 13.1. *Liste des assertions de marquage du graphe*

$Regle\_Declencher\_Operation(Op, el)\{$
$TRUE(Op.$pré-conditions$) \wedge TRUE(Op.invariant)$
$\longrightarrow Op.$post-conditions
$\}$

$Regle\_Impact\_Operation(Op, el)\{$
$TRUE(Op.$pré-conditions$) \wedge FALSE(Op.invariant)$
$\longrightarrow marquer(el)$
$\}$

$Regle\_Propager\_Impact(Op, el)\{$
$+ etat(el,'$*affecté*$')$
$\longrightarrow marquer(el, op)$
$\}$

La première règle consiste à déclencher une opération de modification op sur un élément architectural el dans le cas où les préconditions et les invariants de l'opération sont vérifiés. Par conséquent, les propositions données dans la postcondition de l'opération sont exécutées. Dans ce cas, il n'y a pas de processus de propagation d'impact.

Dans le cas où la précondition est vérifiée tandis qu'au moins un invariant est violé, la deuxième règle est déclenchée. Celle-ci considère que l'élément el passe dans un état affecté.

La troisième règle déclenche la propagation d'impact dans le cas où un élément el est affecté par une opération de modification ayant violé des invariants. Cette règle s'applique à l'ensemble des nœuds, dans l'état affecté, de la base de connaissances. Pour chaque nœud, l'assertion de marquage consistera à visiter chaque relation source et destination, conductrice d'impact, afin de la marquer comme étant affectée par l'opération. La base de connaissances est alors à nouveau modifiée, déclenchant ainsi la règle jusqu'à ce que toutes les relations conductrices d'impact soient traitées.

Les modèles ASCM et SCSM sont formalisés par des graphes dont les types de relations diffèrent. Les règles de propagation d'impact sont différentes selon qu'elles sont destinées à l'architecture ou au code source. La relation de projection établie entre

les deux modèles permet de propager l'impact entre le niveau architectural et le code source du logiciel et inversement. A titre d'exemple, une opération de modification réalisée sur un nœud Nd_i du graphe de ASCM est propagée sur les autres nœuds de l'architecture ayant un lien avec Nd_i mais également vers les éléments du code source ayant une relation de projection avec Nd_i. Dans ce cas, le moteur d'inférence propage l'impact sur l'élément cible de la relation de projection. Le nœud correspondant du modèle SCSM est alors marqué selon la règle de propagation d'impact. Ce marquage permet de déclencher des règles génériques et spécifiques sur les éléments du code source du logiciel afin de propager l'impact.

13.7. Plate-forme pour l'évolution architecturale

Nous avons mis en œuvre un système qui implémente les modèles SCSM et ASCM ainsi que le processus de propagation d'impact des modifications. Nous proposons une implémentation basée sur l'environnement de développement intégré Eclipse en fournissant des *plugins* permettant l'extension de ses fonctionnalités. Dans l'environnement Eclipse, les modèles ASCM et SCSM fournissent une représentation sous forme de graphes de chaque fichier d'un projet Eclipse, concernant respectivement son architecture et ses codes sources. Ces graphes permettent une représentation des relations existant entre les composants architecturaux qui sont projetés vers un code source qui implémente la description architecturale. Ces relations permettent la propagation des impacts de modifications effectuées sur les éléments du graphe. Le processus de propagation est déclenché lors de l'application d'une modification sur un élément du graphe puis par l'exécution des règles formelles définies dans un système expert. Une interface graphique permet la visualisation des graphes, l'exécution des opérations de modification et la visualisation de leurs impacts.

La figure 13.8 montre l'architecture globale de notre plate-forme basée sur Eclipse et étendue pour l'analyse d'impacts. Eclipse permet de développer des logiciels, gérés sous la forme de projets. Un projet Eclipse regroupe un ensemble de ressources qui peuvent être des fichiers de codes sources, des librairies ou des fichiers de description d'architecture. L'ensemble des ressources d'un projet doit être analysé pour être représenté selon nos modèles SCSM et ASCM. Chaque modèle est instancié sous la forme d'un graphe sur lequel il est possible d'exécuter des opérations de modifications. L'impact de ces opérations est calculé et propagé lors du déclenchement de règles introduites dans un système expert. Les extensions de la plate-forme Eclipse sont réparties selon quatre composants principaux [HAS 09] :

– l'analyseur multilangages permet d'analyser les codes sources et les descriptions architecturales d'un logiciel ;

– le modeleur logiciel permet de représenter les informations issues de la phase d'analyse sous forme de graphes correspondant aux instanciations de nos modèles ASCM et SCSM ;

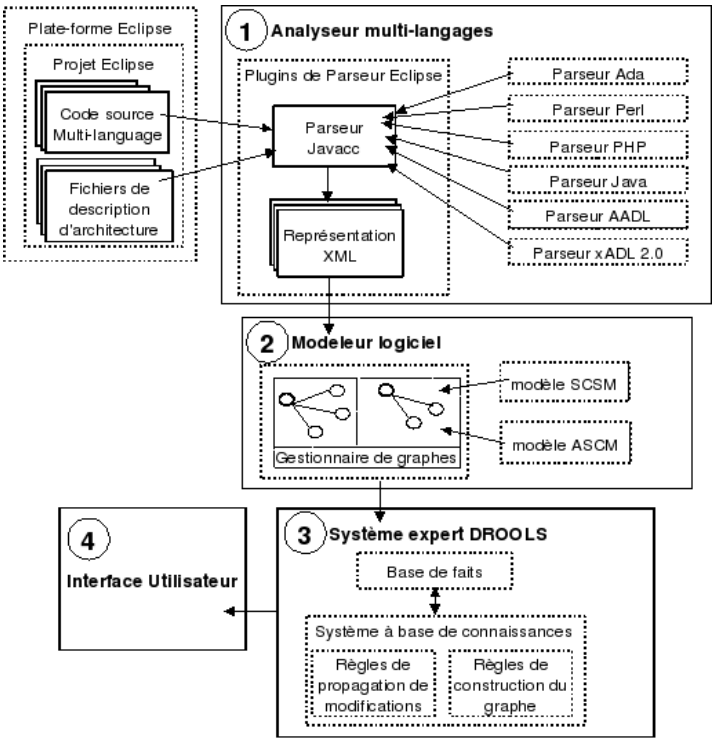

Figure 13.8. *Architecture globale du système basé sur l'environnement de développement Eclipse*

– le moteur du système expert DROOLS[1] permet, d'une part, d'enrichir le graphe obtenu en ajoutant des relations qui sont difficiles à extraire par une analyse lexicale et syntaxique, et d'autre part, d'implémenter le processus de propagation d'impact des modifications ;

– l'interface utilisateur permettant au chargé d'évolution d'effectuer la gestion de l'évolution et la maintenance d'un système logiciel de façon interactive.

13.8. Conclusion

Dans l'objectif d'une analyse plus fine de la propagation d'impacts des modifications, le chapitre propose une modélisation stratifiée des composants logiciels partant du plus haut niveau architectural pour inclure les différents niveaux structurels sous-jacents. La mise en correspondance par un couplage faible des éléments architecturaux

1. JBoss - « *Drools* », www.jboss.org/drools/.

et des codes sources correspondants a été présentée et illustrée. Ce couplage a permis d'assurer un moyen de propager l'impact d'une modification du niveau architectural vers le code source et inversement. La validation de la modélisation par une implémentation de plate-forme a été menée en utilisant une approche évolutive à base de connaissances. Une élaboration progressive et détaillée des opérations de modification et proposée en spécifiant ces opérations par des invariants et des pré et postconditions. Des règles génériques sont ensuite proposées afin d'identifier les chemins de propagation d'impacts des modifications.

13.9. Bibliographie

[AHM 08] AHMAD A., BASSON H., DERUELLE L., BOUNEFFA M., « Towards a better control of Change Impact Propagation », *INMIC'08 : 12th IEEE International Multitopic Conference*, IEEE Computer Society, p. 398-404, décembre 2008.

[AHM 10] AHMAD A., BASSON H., DERUELLE L., BOUNEFFA M., « Towards an integrated quality-oriented modeling approach for software evolution control », *IEEE ICSTE 2010 : 2nd International Conference on Software Technology and Engineering*, San Juan, Puerto Rico, Etats-Unis, IEEE Computer Society, p. V2-320 - V2-324, octobre 2010.

[BAS 98] BASSON H., *Contôle de l'évolution des logiciels : modélisation pour l'analyse d'impact des modifications*, Document HDR, Université de Nancy I, Nancy, décembre 1998.

[CHE 90] CHEN Y.-F., NISHIMOTO M. Y., RAMAMOORTHY C. V., « The C Information Abstraction System », *IEEE Transactions on Software Engineering*, vol. 16, n°3, p. 325-334, IEEE Press, 1990.

[CHI 90] CHIKOFSKY E. J., CROSS II J. H., « Reverse Engineering and Design Recovery : A Taxonomy », *IEEE Software*, vol. 7, n°1, p. 13-17, IEEE Computer Society Press, 1990.

[DER 01] DERUELLE L., *Analyse d'impact de l'évolution des applications distribuées multi-langages et à bases de données hétérogènes*, thèse, Université du Littoral Côte d'Opale, Calais, décembre 2001.

[FAV 06] FAVRE J. M., ESTUBLIER J., BLAY M., *L'ingénierie Dirigée par les Modèles : au-delà du MDA*, Hermès - Paris, 2006.

[GRO 00] GROUP I. A. W., IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems, Rapport, IEEE, 2000.

[HAS 09] HASSAN M. O., DERUELLE L., BASSON H., « Towards a Change Propagation Process in Software Architecture », *18th International Conference on Software Engineering and Data Engineering (SEDE-2009)*, Las Vegas, Nevada, Etats-Unis, p. 85-90, juin 2009.

[HAS 10] HASSAN M., DERUELLE L., BASSON H., AHMAD A., « A Change Propagation Process For Distributed Software Architecture », *ENASE 2010 : 5th International Conference on Evaluation of Novel Approaches to Software Engineering*, Athènes, Grèce, 22 - 24 juillet 2010.

[LAV 10] LAVAL J., DENIER S., DUCASSE S., FALLERI J.-R., « Supporting Simultaneous Versions for Software Evolution Assessment », *Journal of Science of Computer Programming*, elsevier, décembre 2010.

[MAW 07] MAWEED Y., *Modélisation architecturale pour la gestion de l'évolution des applications distribuées*, thèse, Université du Littoral Côte d'Opale, Calais, juin 2007.

[RAJ 97] RAJLICH V., « A Model for Change Propagation Based on Graph Rewriting », *ICSM '97 : Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society, p. 84-91, octobre 1997.