GlobeCooker

Meta-Driven Software Architecture Reconstruction

Loïc Poyet

Ecole Supérieure d'Ingénieurs d'Annecy
Université de Savoie
France

Rapport Stage Master

Responsable

Prof. Dr. Stéphane Ducasse
Language and Software Evolution Group
Université de Savoie
France

Septembre 2006

Abstract

To answer new recurring requirements, software applications are doomed to continually evolve and grow. *Re-engineering* is the activity that aims at maintaining and evolving systems while *reverse-engineering* is the activity that aims at understanding them. Reverse-engineering is seen as part of re-engineering. As a cartographer who draws world maps, a reverse-engineer often draws software models. However, software models can be at various abstraction levels. Basic objects that compose a system are often highlighted. However, reverse-engineering approaches have difficulty in reconstructing architectural views that only stress few components and relationships. Working on both re-engineering and software architecture research areas, the laboratory I joined is working on a project that aims at exploring re-architecting and reverse-architecting approaches *i.e.,* respectively software architecture driven re-engineering and reverse-engineering approaches. In this context, I submitted a state of the art in reverse architecting both in an international conference and a journal. To validate a reverse-architecting approach, I also extended a re-engineering environment.

Keywords :

Reverse-Engineering
Reverse-Architecting
State of the Art
Object-Oriented Development
Metamodeling

Résumé

Pour répondre à de perpétuels nouveaux besoins, les logiciels sont condamnés à évoluer et à se complexifier continuellement. La *ré-ingénierie* désigne les activités de maintenance et d'évolution des logiciels. La *rétro-ingénierie* désigne quant à elle l'activité de compréhension des logiciels. La rétro-ingénierie est ainsi au service de la ré-ingénierie. A l'image d'un cartographe construisant des cartes d'un monde, un rétro-ingénieur représente un logiciel à l'aide de modèles. Cependant, ces modèles sont à différents niveaux d'abstraction. Les approches de rétro-ingénierie se limitent souvent à représenter uniquement les objets de base qui composent un logiciel. Les approches qui reconstruisent des vues architecturales composées seulement de quelques briques et interactions logicielles sont moins nombreuses. Du fait de sa double compétence en ré-ingénierie et en architecture logicielle, l'équipe de recherche que j'ai intégrée a débuté un projet visant à explorer le domaine de la *ré-architecturisation* et de la *rétro-architecturisation*, c'est à dire respectivement de la ré-ingénierie et de la rétro-ingénierie dirigée par les architectures logicielles. Afin d'introduire cette équipe auprès des chercheurs de ce domaine, j'ai soumis un état de l'art dans une conférence et un journal internationaux à propos de la rétro-architecturisation. Afin de valider une approche de rétro-architecturisation, j'ai également étendu un environnement de ré-ingénierie.

Mots Clés :

Rétro-Ingénierie
Rétro-Architecturisation
Etat de l'Art
Développement Orienté Objet
Metamodélisation

Remerciements

Je souhaite avant tout remercier mon responsable de stage, Monsieur Stéphane Ducasse, Professeur à l'Université de Savoie et Directeur de l'équipe de recherche Language and Software Evolution, pour d'une part son expertise dans le domaine de la ré-ingénierie, d'autre part pour m'avoir donné l'opportunité de me confronter à ce domaine et enfin pour m'avoir encadré et conseillé.

J'exprime tout particulièrement ma gratitude à Monsieur Damien Pollet, Post-Doctorant à l'Université de Savoie au sein de l'équipe de recherche Language and Software Evolution, pour avoir co-rédigé l'état de l'art et pour ses connaissances techniques appréciables.

Je souhaite également adresser un clin d'oeil à l'ensemble des enseignants chercheurs de l'équipe Language and Software Evolution pour leur travail de relecture ainsi que pour les discussions constructives que l'on a pu avoir ensemble sur des sujets divers et variés.

Je remercie de plus l'ensemble des relecteurs de l'état de l'art externes à l'équipe.

Table des matières

Chapitre 1

Introduction

1.1 Maintenance et évolution

Les logiciels sont condamnés à évoluer et à se complexifier continuellement pour satisfaire de nouveaux besoins. Les lois de l'évolution perpétuelle et de la complexité croissante dictées par Lheman et Belady le stipulent [Lehm85] :

- **Loi de l'évolution perpétuelle**
  Un logiciel vivant se doit de changer continuellement. A défaut, il deviendra progressivement obsolète.
- **Loi de la complexité croissante**
  La structure d'un logiciel supporte mal l'évolution. Des ressources sont requises pour accompagner l'évolution d'un logiciel.

Ainsi, la maintenance et l'évolution sont devenues deux activités centrales et coûteuses du cycle de vie des logiciels à longue durée de vie et de grande taille [Lien80, McKe84, Somm96]. Ces deux activités représentent à elles seules 50% à 70% du coût total d'un logiciel.

1.2 Ré-ingénierie

Cette problématique a donné naissance à de nombreux travaux. Basé sur la taxonomie de Cross et Chikofsky [Chik90], le monde s'organise de la manière suivante[1] *(Cf. Figure 1.1)* :

- **Ré-ingénierie** *Re-engineering*
  La ré-ingénierie désigne les activités de maintenance et d'évolution des logiciels. Elle vise à examiner puis à altérer un logiciel de manière itérative. Elle se compose de phases de rétro-ingénierie et de phases de direct-ingénierie.
- **Rétro-ingénierie** *Reverse-engineering*
  La rétro-ingénierie désigne l'activité de compréhension des logiciels. Elle vise à extraire des représentations conceptuelles du logiciel.
- **Direct-ingénierie** *Forward-engineering*
  La direct-ingénierie désigne l'activité d'implémentation des logiciels. Elle vise à l'inverse de la rétro-ingénierie à produire du code à partir de représentations conceptuelles.

[1] Afin d'être concis, cette structuration présente quelques abus de langage.

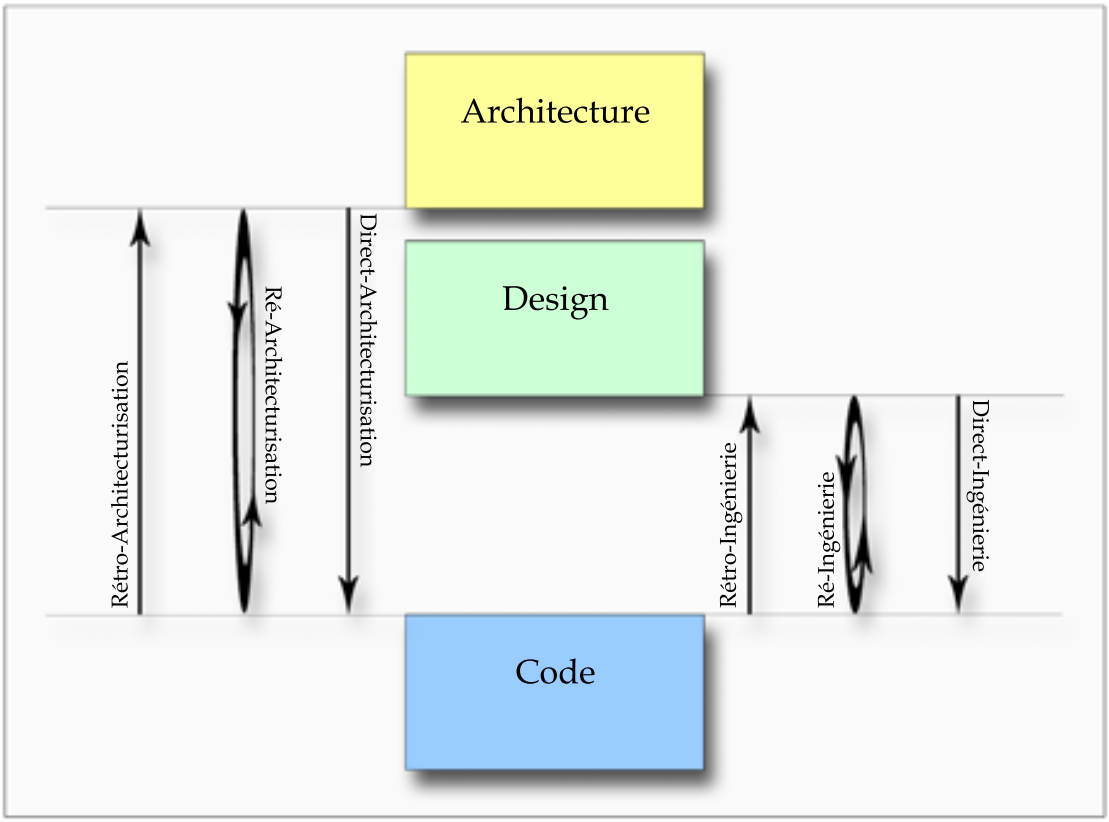

Fig. 1.1 – Les activités du monde de la ré-ingénierie

- **Ré-architecturisation** *Re-architecting*
  La ré-architecturisation désigne une activité de ré-ingenierie dirigée par les architectures logicielles. Elle considère la notion d'architecture logicielle comme un élément capital pour améliorer le processus.
- **Rétro-architecturisation** *Reverse-architecting*
  La rétro-architecturisation désigne une activité de rétro-ingenierie dirigée par les architectures logicielles.
- **Direct-architecturisation** *Forward-architecting*
  La direct-architecturisation désigne une activité de direct-ingenierie dirigée par les architectures logicielles.

1.3 Rétro-architecturisation

La rétro-architecturisation fut le thème central de mon stage. Cette activité fait ainsi l'objet de l'ensemble de cette dissertation. Un projet de recherche est à l'origine du sujet de mon stage. Dans un premier temps, ce projet est introduit. La présentation de ce projet met en évidence un ensemble d'objectifs à atteindre dont ceux de mon stage. Ces objectifs sont au nombre de deux et font chacun l'objet d'un chapitre. Le premier consiste à soumettre un état de l'art dans une conférence et un journal internationaux. Le deuxième consiste à étendre un environnement de ré-ingénierie existant. Enfin, une discussion sur divers aspects du monde de la recherche conclue cette dissertation.

Chapitre 2

Contexte

Ce présent chapitre situe le contexte professionnel de mon stage. Il introduit l'équipe de recherche LSE du laboratoire LISTIC au sein de laquelle j'ai réalisé mon stage. D'autre part, il dévoile le cheminement qui a conduit cette équipe à construire un projet de recherche inscrit dans le long terme concernant la ré-architecturisation d'applications logicielles. La présentation de l'environnement du projet conduit tout naturellement à la définition de mes objectifs.

2.1 LISTIC

Le LISTIC (Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance) est un laboratoire de l'Université de Savoie implanté sur Annecy. Le laboratoire travaille sur des thématiques axées essentiellement sur la conception, la réalisation et l'exploitation de systèmes de fusion de l'information. Le monde repose en effet sur l'information. Tout système technologique exploite de multiples sources d'informations. Cette information tend à devenir hétérogène, volumineuse et répartie. Fusionner l'information motive ainsi les recherches entreprises au sein de ce laboratoire.

Le laboratoire est composé de trois équipes de recherche distinctes *(Cf. Figure 2.1)*. Chaque équipe est statutairement animée par un enseignant chercheur habilité à diriger des recherches et se focalise sur un domaine plus spécifique. Ces trois équipes sont les suivantes :

- **IC** *Ingénierie de la Connaissance*
  Les travaux de cette équipe portent sur la modélisation des connaissances et des systèmes. Ses membres travaillent aussi bien sur leurs fondements historiques que sur leurs applications actuelles. Alors que la modélisation des connaissances repose essentiellement sur les notions d'ontologie et de terminologie, la modélisation des systèmes reposent quant à elle sur les systèmes multi-agents.
- **LS** *Logiciels et Systèmes*
  Ses chercheurs proposent des solutions pour concevoir, réaliser et déployer des systèmes de fusion en portant une attention toute particulière aux aspects logiciels, évolutifs et répartis des systèmes d'information.
- **TI** *Traitement de l'Information*
  Cette équipe oeuvre à la définition d'outils et de méthodologies pour la représentation, la fusion, l'évaluation ou bien encore la justification de l'information. Le traitement d'images,

le contrôle de la performance industrielle et la gestion de systèmes de production sont des activités choisies pour valider les approches.

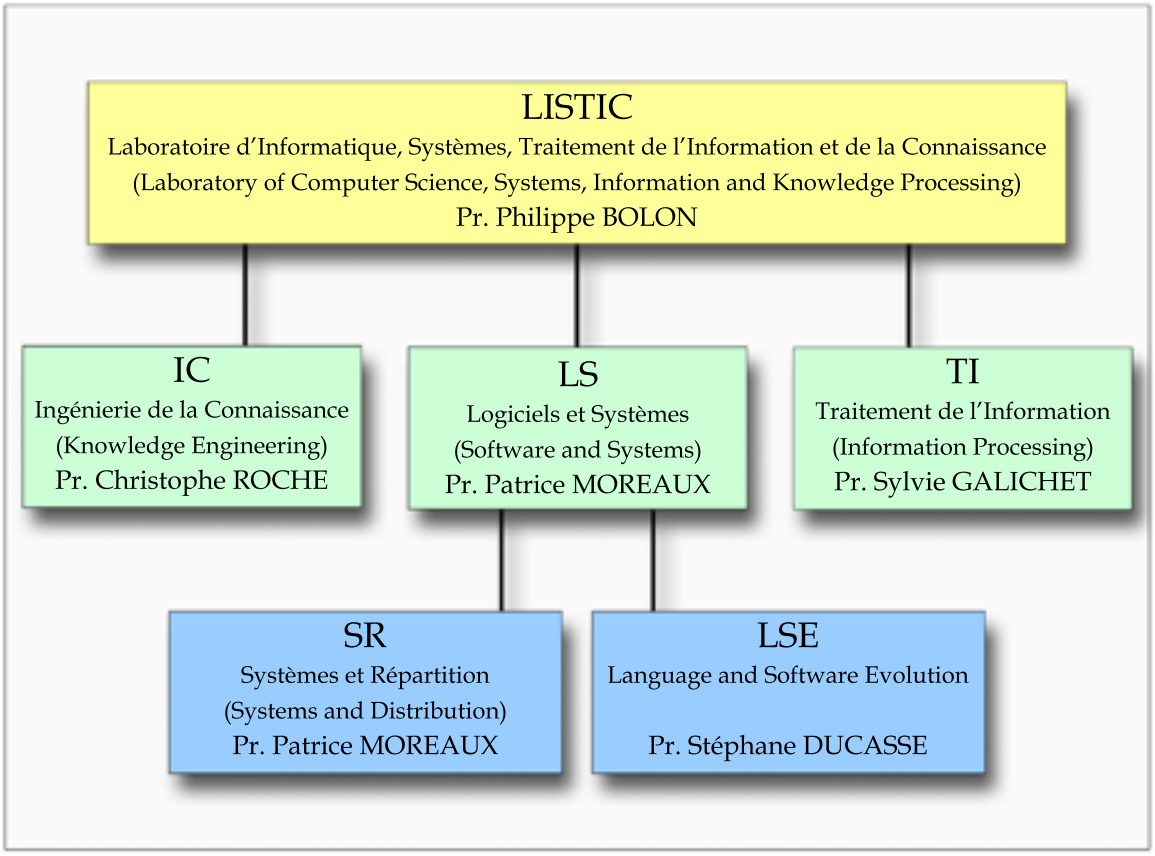

FIG. 2.1 – LSE : une équipe de recherche au sein du LISTIC

2.2 LSE

L'équipe LS se décompose en deux entités *(Cf. Figure 2.1)* :

- **SR** *Systèmes et Répartition*
  Cette entité axe sa recherche autour des systèmes distribués, à évènements discrets et multi-agents.
- **LSE** *Language and Software Evolution*
  Le génie logiciel est la compétence centrale de cette équipe. Les thèmes abordés gravitent autour des architectures logicielles, de la ré-ingénierie, des systèmes multi-agents et des services Webs.

C'est au sein de l'équipe LSE que j'ai été accueilli pour réaliser mon stage de fin d'études. D'un point de vue historique, les enseignants chercheurs de LSE sont spécialisés dans la description d'architectures logicielles et de processus logiciels. Suite au départ de leur membre le plus influent et suite au recrutement de Stéphane DUCASSE renommé dans le domaine de la ré-ingénierie, l'équipe a récemment souhaité se doter d'une nouvelle identité et fusionner ses compétences en trouvant un thème de recherche satisfaisant la majorité. C'est ainsi que le projet Cook est né.

2.3 Cook

Le nom de ce projet rend hommage au célèbre explorateur anglais du même nom. Un tel titre souligne le parallèle qui peut être fait entre partir à la découverte d'un monde et partir à la découverte d'un logiciel. Plus sérieusement, non seulement le projet Cook répond à un besoin interne de l'équipe actuellement en pleine restructuration *(Cf. Figure 2.2)* mais de plus il répond à une réelle demande de l'industrie du logiciel pour laquelle la maintenance et l'évolution sont des aspects économiquement lourds à soutenir quand on sait que plus de la moitié de l'effort de développement d'un logiciel y est consacré.

Cook a ainsi pour objectif de prendre en compte la notion d'architecture logicielle comme un élément capital dans le processus de maintenance et d'évolution des logiciels. Cinq axes de recherche sont envisagés dans Cook :
- Modéliser la notion d'architecture
- Extraire et visualiser l'architecture d'un logiciel
- Identifier les violations entre architectures
- Analyser l'évolution d'une architecture
- Refactoriser une architecture en une autre

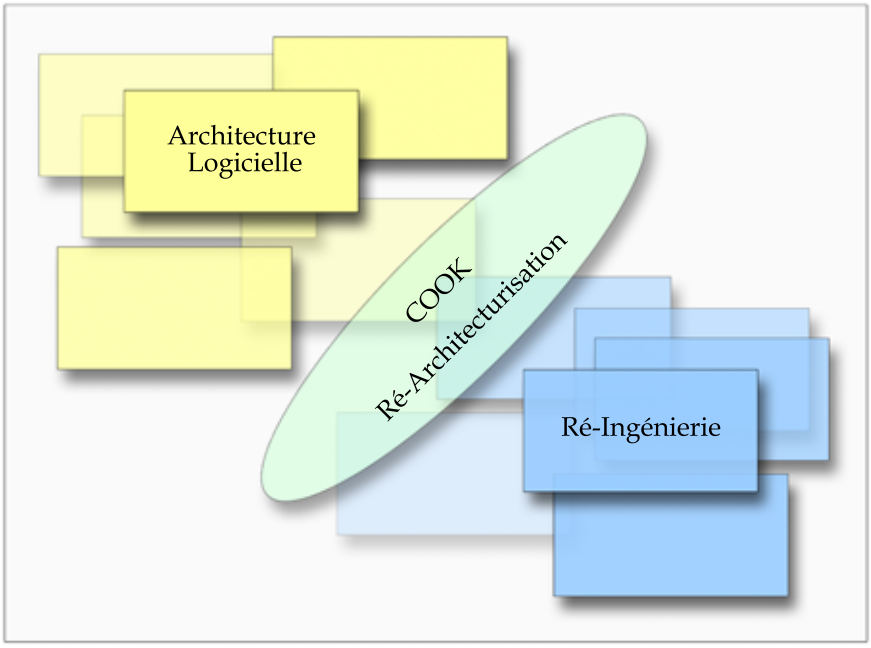

Fig. 2.2 – Cook : allier deux compétences

2.4 Objectifs

En choisissant de réaliser le projet Cook, l'équipe LSE s'oriente d'une part dans un monde qu'elle ne connaît pas et d'autre part dans un monde dans lequel on ne la connaît pas. Par chance, l'absence d'un état de l'art dans le domaine est pour l'équipe une occasion rêvée d'acquérir la connaissance nécessaire du domaine et de se présenter à la communauté.

Le premier objectif de mon stage est ainsi clairement établi. Ma mission est d'écrire une taxonomie sur le domaine de la rétro-architecturisation et de la soumettre dans une conférence et un journal internationaux. Ensuite, il est nécessaire de savoir que Stéphane Ducasse, à la tête de cette équipe, a donné naissance en collaboration avec des chercheurs de l'Université de Berne en Suisse à un environnement de ré-ingénierie propice à la validation d'approches. Mon second objectif fut de prendre en main cet environnement et d'implémenter une approche de rétro-architecturisation.

Chapitre 3

Etat de l'art

Les sections ci-dessous présentent ma première mission, à savoir l'écriture d'un état de l'art à propos de la rétro-architecturisation. Les objectifs de cette mission font l'objet de la première section. Quelques chiffres clés ainsi que la démarche suivie sont ensuite présentés à titre d'information. Succède à cette section, une synthèse de l'état de l'art. L'état de l'art complet est joint en annexe de cette dissertation en version anglaise. La section suivante présente une critique sur cet état de l'art rédigée par le comité de programmation d'une conférence dans laquelle l'état de l'art a été soumis. Enfin une discussion sur les perspectives envisagées conclura le chapitre.

3.1 Objectifs

L'écriture de l'état de l'art sur le domaine de la rétro-architecturisation présente un objectif double :

- Présenter l'équipe LSE aux chercheurs du domaine.
- Orienter l'équipe LSE dans ses futures investigations.

3.2 Chiffres clés

3.2.1 Aperçu

Les quelques chiffres clés approximatifs suivants soulignent l'investissement nécessaire pour écrire un tel papier :

- Nombre de pages : 40
- Nombre de références : 175
- Nombre d'approches classifiées : 35
- Nombre de papiers lus : 500
- Nombre de refactorisations importantes : 3
- Nombre d'auteurs : 3
- Nombre de mois : 5

Type	Quantité	Temps *(En minutes)*	Temps Total *(En minutes)*
Thèse	20	120	2400
Publication Rang 1	150	60	9000
Publication Rang 2	200	30	6000
Publication Rang 3	130	15	1950

TAB. 3.1 – L'activité de lecture

3.2.2 Activités

La section ci-dessus a insisté sur quelques chiffres clés à propos de l'état de l'art réalisé. Cependant, ces chiffres en cachent d'autres. Ils ne sont que la face visible de l'iceberg. En effet, de nombreuses autres publications ont été lues et analysées pour aboutir à un tel résultat. L'objectif de cette section est d'y accorder un peu plus d'importance.

En effet, je souhaite signaler que trois grandes activités rythmaient mon quotidien, à savoir :
- Lire : 40%
- Analyser : 40%
- Ecrire : 20%

L'activité de lecture contraint les deux autres activités. Or environ 325 heures ont été consacrées à la lecture de publications *(Cf. Figure 3.1)*. Ainsi la réalisation de cet état de l'art m'a nécessité personnellement 800 heures environ de travail. Ceci représente 5 mois de travail à un rythme de 40 heures de travail par semaine.

3.3 Démarche

La démarche a évolué au fil du temps. Aucune méthodologie m'a été réellement et explicitement transmise ce qui m'a conduit à la définir tout au long de la mission. Je souhaite cependant au sein de cette section donner les lignes directrices d'une méthodologie si une telle mission m'était donnée à refaire. Ma méthodologie serait la suivante :
- **Comprendre**
  Sélectionner les dix thèses les plus centrales au domaine. Pour chacune d'entre elles, lire ses chapitres d'ouverture, prendre conscience de ses mots-clefs et cibler à travers sa bibliographie les chercheurs centraux du domaine.
- **Recueillir**
  Constituer sans se poser de questions une base référencée volumineuse de publications choisies selon leur adéquation avec les mots clefs du domaine.
- **Extraire**
  Lire en surface chaque publication et identifier pour chacune d'elles cinq points traités dans cette publication en tâchant de rapprocher au maximum ces points avec ceux mis en évidence lors des précédentes lectures.
- **Choisir**
  Décider des points centraux du domaine à traiter et les transformer en critères de classification.
- **Filtrer**
  Eliminer les publications hors contexte.

- **Approfondir**
  Lire en profondeur chaque papier et stipuler pour chaque critère sa valeur.
- **Ecrire**
  Rédiger les parties centrales de l'état de l'art. Présenter les critères et leurs valeurs envisageables, classifier les approches dans les tables, illustrer ses propos à l'aide de publications et prendre part à des discussions.
- **Contextualiser**
  Introduire le papier, décrire la terminologie du domaine, souligner les défis à relever et conclure. Les activités de cette phase peuvent être effectuées à un autre moment.

3.4 Synthèse

Cette section présente une synthèse de l'état de l'art. Celui-ci est fournit en totalité en annexe de cette dissertation en version anglaise. Les références bibliographiques, les illustrations, les discussions scientifiques et les tables sont notamment absentes de cette synthèse. Il est nécessaire de se rapporter à l'état de l'art complet présent en annexe pour en prendre connaissance. Il est à noter que l'état de l'art en annexe est en travaux suite à des critiques de relecteurs.

3.4.1 Préambule

L'architecture joue un rôle crucial dans le cycle de développement d'un logiciel. Dans le cadre de la maintenance logicielle, altérer une application sans disposer d'une vision architecturale fiable de celle-ci est voué à l'échec. Cependant, connaître l'architecture d'un système n'est pas une tâche aisée. D'une part, l'architecture n'est pas explicitement représentée dans le code comme les classes ou les packages peuvent l'être. D'autre part, les logiciels sont condamnés à évoluer et se complexifier perpétuellement, or lorsqu'un logiciel évolue son architecture aussi. L'architecture conceptuelle présente dans l'esprit du rétro-architecte devient progressivement imprécise et obsolète vis à vis de l'architecture concrète présente dans le code. La rétro-architecturisation est une approche de rétro-ingénierie dont le but est de reconstruire des vues architecturales fiables d'un logiciel.

3.4.2 Définitions

Quelques définitions brèves permettent de mieux aborder le domaine :
- **Architecture logicielle**
  L'architecture d'un logiciel se compose d'un ensemble restreint de composants, de relations et de principes rationnels.
- **Architecture conceptuelle**
  L'architecture conceptuelle d'un logiciel est celle mentalement détenue par un rétro-architecte.
- **Architecture concrète**
  L'architecture concrète d'un logiciel est celle réellement implémentée.
- **Style architectural**
  Un style architectural est un motif organisationnel de composants, de relations et de principes rationnels auquel conforment des architectures.
- **Vue architecturale et point de vue architectural**
  Une vue architecturale est une vue d'un logiciel conforme à un certain point de vue. Un point de vue architectural stipule les concepts auxquels une vue doit se conformer pour répondre

aux attentes d'un rétro-architecte. La relation qui existe entre une vue et un point de vue est la même qui existe entre un modèle et un métamodèle (4.5.3).

3.4.3 Domaines de recherche fortement corrélés

Cet état de l'art couvre un environnement délimité. Des domaines fortement corrélés au domaine de la rétro-architecturisation pourrait lui apporter beaucoup. Ces domaines annexes dont il est question cherchent tous à extraire et à représenter des concepts présents à un niveau d'abstraction supérieur à celui du code mais inférieur au niveau architectural. Ces concepts peuvent néanmoins être au service d'une approche de rétro-architecturisation pour faire glisser un modèle du code au niveau architectural. Ces concepts dont il est question sont les modèles de conception, les aspects, les fonctionnalités ainsi que les rôles et collaborations. Ces aspects ne seront pas plus détaillés dans le cadre de cette synthèse.

3.4.4 Défis

Les défis auxquels les approches de rétro-architecturisation font face sont fortement corrélés à l'information traitée, à savoir essentiellement le code source du logiciel ainsi que l'expertise humaine à son sujet. L'expertise humaine est primordiale pour la rétro-architecturisation car elle est la source majeure d'informations conceptuelles. Quant au code source, il représente une des seules sources fiables d'information concernant le logiciel. Le défi majeur de la rétro-architecturisation réside ainsi dans l'extraction, l'abstraction et la présentation de vues architecturales à un haut niveau d'abstraction à partir d'une information hétérogène.

3.4.5 Axes de la taxonomie

Plusieurs états de l'art succincts existent sur la rétro-architecturisation ou sur des domaines proches. L'état de l'art proposé apporte à la communauté une qualité de réflexion non atteinte dans ce domaine. Les critères de classification de la taxonomie ont été sélectionnés selon le point de vue d'un rétro-architecte. Chaque critère fait l'objet d'une sous section ci-dessous. Chaque critère est illustré avec quelques exemples de valeurs envisageables.

3.4.6 Buts

La rétro-architecturisation est considérée par la communauté comme une approche proactive. En effet, une approche de rétro-architecturisation répond à un ensemble de buts propre à chaque rétro-architecte. Toutes les approches de rétro-architecturisation cherchent notamment à comprendre et documenter un logiciel. Cependant, la plupart des approches se fixent également d'autres objectifs. Deux de ceux-ci sont illustrés ci-dessous :

- **But = Conformance**
  Certaines approches visent à vérifier la conformance d'une architecture conceptuelle avec une architecture concrète.
- **But = Migration vers une famille produit**
  Dans le monde du logiciel, une famille produit est un ensemble de logiciels qui partagent des composants logiciels communs tout en les particularisant si besoin. Certaines approches visent à reconstruire les architectures respectives de plusieurs logiciels et d'extraire une architecture de référence commune à tous les produits.

3.4.7 Processus

Les approches de rétro-architecturisation suivent soit une stratégie ascendante, descendante ou hybride. La qualification du processus se base sur le niveau d'abstraction de l'information en entrée et en sortie. Le code source est une source d'information considérée comme étant à un bas niveau d'abstraction. A l'inverse, une architecture conceptuelle ou un style architectural sont des sources d'information considérées comme étant à un haut niveau d'abstraction. Ainsi :

- **Processus = Ascendant**
  L'information en entrée est de bas niveau. L'information en sortie est de haut niveau. Le but est d'élever le niveau d'abstraction progressivement. Grouper progressivement les entités présentes dans le code jusqu'à atteindre une vue architecturale est l'exemple d'une approche suivant un processus ascendant.
- **Processus = Descendant**
  L'information en entrée est de haut niveau. L'information en sortie est de bas niveau. Le but est de raffiner progressivement. Vérifier progressivement la conformance d'une vue architecturale conceptuelle vis à vis du code source est l'exemple d'une approche suivant un processus descendant.
- **Processus = Hybride**
  Les deux s'exécutent et se confrontent itérativement jusqu'à ce que les architectures issues des deux processus soient similaires en tout point. Grouper progressivement les entités présentes dans le code jusqu'à atteindre une vue architecturale concrète, vérifier progressivement la conformance d'une vue architecturale conceptuelle vis à vis du code source, confronter itérativement les architectures concrète et conceptuelle afin d'en dériver une conforme est l'exemple d'une approche suivant un processus hybride.

3.4.8 Entrées

Il est à distinguer les entrées dites architecturales et celles qualifiées de non architecturales. Une information est qualifiée d'architecturale si elle touche à des concepts architecturaux. Les points de vue et les styles architecturaux sont des informations architecturales. A l'inverse le code source ou l'information dynamique ne le sont pas.

Entrées Architecturales. La taxonomie stipule si les approches prennent en compte ou non la définition de points de vue ou la définition de styles architecturaux.

Entrées Non Architecturales. Fréquemment, le code source de l'application est requis par l'approche. Cependant d'autres sources d'informations sont considérées par certaines approches. En voici deux exemples :

- **Entrée = Information dynamique**
  En règle générale, l'idée consiste à étudier le logiciel au moment où celui-ci exécute certaines fonctionnalités, d'analyser les interactions entre les entités logicielles participantes et de regrouper ces entités autour de fonctionnalités selon leur contribution à la réalisation de celles-ci.
- **Entrée = Distribution des ressources humaines**
  Prendre en compte ce type d'information est atypique. Elle se base sur la loi de Conway qui stipule que les organisations qui conçoivent des systèmes sont contraintes de produire des architectures fortement corrélées à leur propre structure interne de communication. Ainsi,

regrouper les entités du code selon les équipes de développement peut mener à une vision architecturale fiable du logiciel.

3.4.9 Techniques

Les techniques sont nombreuses. Elles ont été dans un premier temps catégorisées selon leur niveau d'automatisation envisageable. Une technique de chaque catégorie est ci-dessous illustrée :

- **Technique (Quasi-manuelle) = Reconstruction visuelle**
  Cette technique est employée par de nombreux outils de visualisation. Elle se résume à filtrer et grouper interactivement des entités de manière ascendante.
- **Technique (Semi-automatique) = Requêtes logiques**
  Cette technique permet au rétro-architecte de spécifier des règles logiques d'abstraction réutilisables et de les exécuter de manière ascendante et automatique. Les entités du code source sont représentées sous forme de faits. Elles sont les cibles de règles logiques cherchant à les regrouper à l'aide d'une intension au sens mathématique du terme.
- **Technique (Quasi-automatique) = Clusterisation**
  Cette technique regroupe de manière ascendante les entités du code source afin de maximiser la cohésion intra composant et de minimiser la cohésion inter composants.

3.4.10 Sorties

Il n'y a pas de secret, toutes les approches offrent en sortie au rétro-architecte des vues architecturales du logiciel étudié soit graphiques soit textuelles et exprimées selon un langage de description. D'autre part, bien évidemment, les sorties sont fortement corrélées aux buts. Une approche axée sur la migration vers une famille produit produira en sortie les architectures de chaque produit et l'architecture de référence déduite. Une approche axée sur la conformance produira en sortie des informations de divergence et de convergence entre vues architecturales.

3.4.11 Conclusion

Cet état de l'art structure le monde de la rétro-architecturisation selon le point de vue d'un rétro-architecte désireux de savoir quelle approche choisir selon l'information dont il dispose, l'information qu'il souhaite obtenir ou bien encore les techniques qui lui semblent les plus appropriées. Cet état de l'art tend vers une classification de qualité, complète et précise.

3.5 Critique

Il est à rappeler que l'équipe LSE souhaite que le papier soit publié à la fois dans une conférence et dans un journal internationaux. A cet effet, le papier sous une forme simplifiée de treize pages a été soumis au comité de programmation de la conférence internationale sur la rétro-ingénierie dénommée "Working Conference on Reverse Engineering". Le papier a été cependant refusé. Le taux d'acceptation des papiers cette année fut de 28,5%. Ce taux convenable rend le refus plus difficile à digérer d'autant plus que les critiques sont globalement positives. En effet, les critiques des trois lecteurs ayant pris cette décision sont retranscrites et résumées ci-dessous et ne remettent pas en cause le fond de l'état de l'art mais plutôt sa forme. Pour mieux comprendre

cette discussion, il est à noter qu'un article de journal n'a pas de contraintes de place et oscille souvent entre trente et cinquante pages alors qu'un article de conférence est fréquemment limité à dix pages.

- **Lecteur n°1**

  Positif. Le papier présente une analyse approfondie et détaillée des méthodes et techniques de rétro-architecturisation. La taxonomie est originale. Elle présente une terminologie consistante du domaine. Elle apporte énormément en illustrant les différentes approches à travers un grand nombre d'exemples.

  Négatif. Il est cependant surprenant de ne pas voir de tables synthétiques dignes de tout état de l'art. Le papier est trop long.

  Conclusion. Le papier semble plus approprié pour un journal afin d'accueillir des tables.
- **Lecteur n°2**

  Positif. Le papier est d'une aide précieuse pour un rétro-architecte qui souhaite choisir l'approche la plus adéquate à ses attentes. Le papier analyse en détail les caractéristiques importantes d'une approche de rétro-architecturisation. Le papier fait référence aux approches proposées dans la littérature.

  Négatif. La taxonomie n'est pas formalisée. Il manque un exemple complet servant de trame au déroulement de la taxonomie.

  Conclusion. Le papier gagnerait à être plus explicite dans sa classification.
- **Lecteur n°3**

  Positif. Le spectre du papier est large ce qui est fort appréciable. Les axes de la taxonomie sont intuitifs. Je suis impatient de donner ce papier à lire à mes étudiants.

  Négatif. Il est imposible de traiter autant de références en si peu d'espace d'expression. Les approches sont décrites trop rapidement.

  Conclusion. Le papier semble un excellent papier pour un journal afin d'accueillir des illustrations, des discussions et des tables.

Les points négatifs relevés par ces lecteurs sont justifiés d'autant plus que l'équipe LSE en soumettant ce papier en était consciente. Le manque de tables, de discussions et d'illustrations résulte d'une activité de synthèse de l'article afin de le réduire à une dizaine de pages.

3.6 Perspectives

3.6.1 Publications

Conférence. L'équipe LSE souhaite camper sur ses positions. Elle souhaite vraiment signer un article de conférence. En effet son but est de se présenter à la communauté. Or un tel papier ouvre la porte sur une présentation publique contrairement à un papier de journal. Une réduction plus soigneuse va être entreprise prochainement par l'équipe. Le papier devrait être soumis à la conférence internationale sur la maintenance logicielle dénommée "International Conference on Software Maintenance" l'année prochaine.

Journal. L'état de l'art dans sa version complète est de qualité. Il résoud notamment les problèmes mis en évidence par les lecteurs de la version simplifiée. Il sera soumis prochainement dans un journal international spécialement consacré à la publication de telles taxonomies. Ce journal est intitulé "ACM Surveys". Le comité de lecture qui accepte ou non les papiers devrait faire patienter l'équipe entre trois et huit mois avant de lui faire parvenir une critique.

Citations. Quelque soit le support de publication, cet article devrait être référencé par l'ensemble des futurs papiers de l'équipe LSE qui traiteront de la rétro-architecturisation. Il devrait être également référencé par de nombreux articles du domaine au cours des prochaines années. Enfin, il sera considéré comme une contribution majeure du projet Cook quand il s'agira de présenter ce projet.

3.6.2 Un pas vers GlobeCooker

Cet état de l'art structure le monde de la rétro-architecturisation. L'équipe LSE dispose ainsi d'un véritable support pour s'orienter de manière réfléchie vers certains horizons. Elle souhaite notamment rester terre à terre lors de ses prochaines investigations. Elle ne souhaite pas traiter des problèmes liés à la définition de points de vues ou s'attaquer à la rétro-architecturisaiton basée sur les styles architecturaux. Ces deux concepts sont trop conceptuels. L'équipe désire au contraire explorer des pistes de rétro-architecturisation ascendante basée sur des techniques de modularisation. L'approche GlobeCooker, présentée dans le chapitre suivant, découle de ces remarques.

Chapitre 4

GlobeCooker

Les sections ci-dessous présentent ma seconde mission : l'implémentation de l'approche de rétro-architecturisation GlobeCooker. Les objectifs de cette mission font l'objet de la première section. Ensuite, l'approche GlobeCooker est positionnée au sein de la précédente taxonomie. Dans un troisième temps, la démarche suivie pour mener à bien cette mission est décrite. Succède à cette section une présentation de l'environnement de ré-ingénierie Moose sur lequel se base l'approche. L'approche GlobeCooker est ensuite détaillée. Enfin, une discussion sur les perspectives de cette mission conclue le chapitre.

4.1 Objectifs

Les objectifs de ma seconde mission sont au nombre de deux :
- Me familiariser avec l'environnement de ré-ingénierie Moose sur lequel GlobeCooker se base.
- Implémenter GlobeCooker au sein de cet environnement.

4.2 Positionnement

4.2.1 Orientation

L'état de l'art structure le monde de la rétro-architecturisation. Pour l'équipe, il est plus aisé de choisir son orientation. L'état de l'art a notamment souligné la difficulté des approches à traiter les concepts architecturaux tels que les styles ou les points de vue ou bien encore à traiter de l'information rationnelle et humaine justifiant la présence ou non d'un composant ou d'une relation au niveau architectural. Selon ces approches, la notion d'architecture est propre de plus à chacun d'entre nous. Ainsi, les modèles architecturaux produits par de telles approches ne se ressemblent pas. Evaluer de telles approches est voué à l'échec.

L'équipe ne souhaite pas à court terme se plonger dans ces difficultés dont la résolution nécessite de l'expertise. L'équipe souhaite par conséquent s'orienter vers l'exploration d'approches dont les résultats sont mesurables par des critères mathématiques. Elle désire notamment explorer des approches de rétro-architecturisation ascendantes basées sur des techniques de mo-

dularisation en vue éventuellement de vérifier la conformance des vues produites avec des vues conceptuelles fournies par un rétro-architecte.

4.2.2 Classification

Cette approche se positionne au sein de l'état de l'art de la manière suivante :
- Buts : Redocumentation.
- Processus : Ascendant.
- Entrées : Code source.
- Techniques : Quasi-automatique.
- Sorties : Visualisation.

Il est à noter que l'implémentation actuelle de l'approche est propice à une extension de vérification de la conformance entre une architecture concrète et une architecture conceptuelle. L'extension se doit d'apporter un moyen automatique de comparer deux modèles exprimés dans un même formalisme. La présence du métamodèle facilite l'implémentation de cette extension.

4.3 Démarche

Ma démarche fut la suivante :
- **Me familiariser avec Moose**
  Cette phase de familiarisation fut l'objet d'un autre projet en amont du stage. Elle fut complétée au cours de ce stage par la lecture de deux publications concernant respectivement la métamodélisation et l'outil de visualisation Mondrian qui sont des aspects traités par la suite [Duca06, Meye06].
- **Imaginer une approche de rétro-architecturisation**
  Cette phase a consisté à explorer des pistes de rétro-architecturisation. Ces pistes sont au nombre de deux. La première fut axée sur la rétro-architecturisation ascendante basée sur des règles logiques pour grouper des entités et ainsi produire des composants plus abstraits [Mens06]. La deuxième fut celle conservée et présentée dans ce chapitre. Elle se base sur des techniques quasi-automatiques de modularisation [Leng79, Kosc00, Coop01, Trif01, Lund03].
- **Implémenter GlobeCooker**
  Cette phase fut réalisée en plusieurs temps :

  Etendre le métamodèle Moose. L'extension du métamodèle Moose fut nécessaire pour prendre en compte les deux concepts de base d'un modèle architectural, à savoir les composants et les relations. Le concept de métamodélisation est bien évidemment introduit par la suite.

  Choisir des scénarios. Ma démarche a consisté ensuite à décrire deux scénarios devant servir de ligne de conduite à l'implémentation de mon approche de rétro-architecturisation.

  Implémenter le coeur de l'approche. Il a été ensuite nécessaire d'implémenter l'approche au sein de Moose dans un langage orienté objet nommé Smalltalk. Cette phase a nécessité l'interaction avec l'environnement Moose, l'outil de visualisation Mondrian et le métamodéle précédemment définie. Elle a également requis la lecture de plusieurs publications sur des techniques de modularisation basées sur le concept de dominance introduit par la suite [Leng79, Kosc00, Coop01, Trif01, Lund03].

Validation des scénarios. Pour finir, la validation des scénarios a permis de valider l'approche dans son ensemble.

4.4 Moose

Moose est une plate-forme outillée de ré-ingénierie d'applications objets industrielles conséquentes *(Cf. Figure 4.1)* [Duca05]. Cette plateforme est le résultat d'un programme de recherche européen dans le domaine de la ré-ingénierie, intitulé Famoos [Duca99]. Cet espace de ré-ingénierie permet essentiellement :

- L'extraction et le stockage de modèles représentatifs d'une application.
- L'interactivité avec ces modèles.
- L'interfaçage avec des outils d'analyse tels que :

  CodeCrawler et Mondrian. L'outil de ré-ingénierie CodeCrawler combine des métriques et des techniques de visualisation pour faciliter la rétro-conception d'un système [Lanz03]. Mondrian est son successeur [Meye06].

  Van. L'outil de ré-ingénierie Van visualise l'évolution d'une application [Girb05].

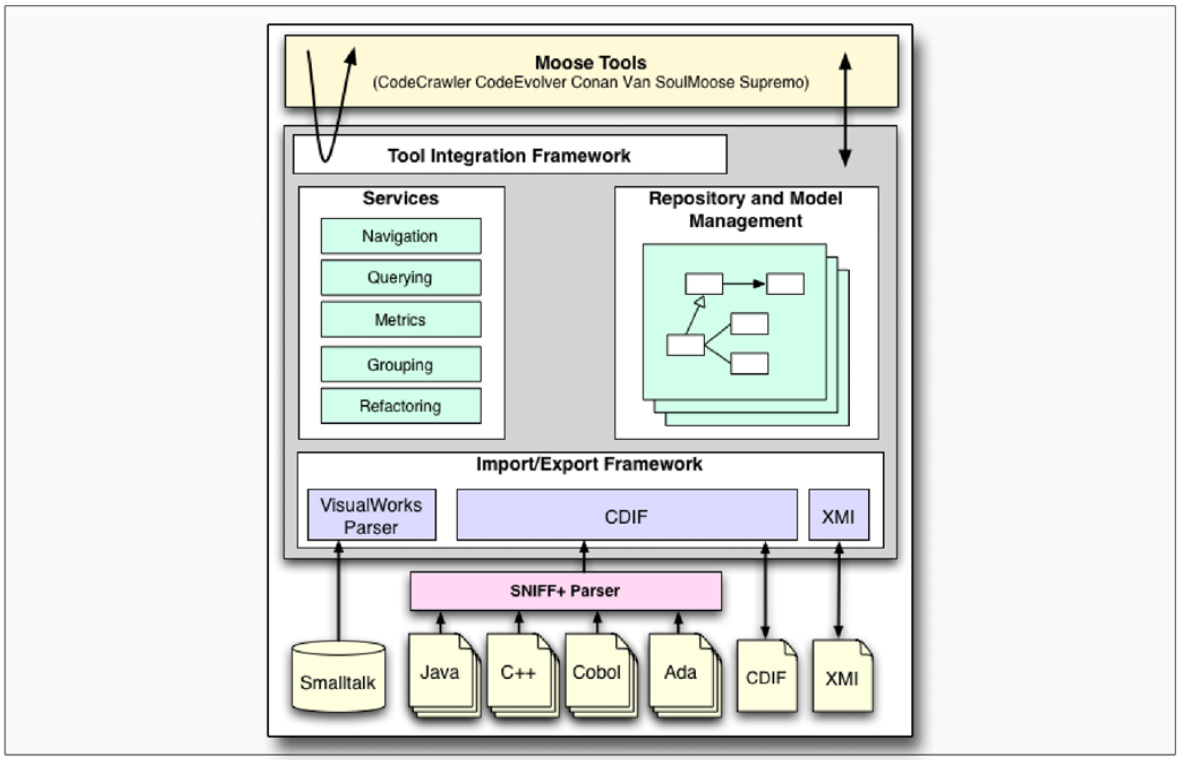

Fig. 4.1 – L'architecture de Moose

4.5 L'approche

GlobeCooker désigne l'approche de rétro-architecturisation à laquelle j'ai donnée naissance dans Moose. Son nom fait allusion à l'idée du globe trotter explorant le monde du logiciel mais aussi au nom du projet.

4.5.1 Scénarios

Les deux scénarios choisis auxquels devra répondre GlobeCooker sont les suivants :

– **Scénario 1**

Idée sous-jacente. Les modules sont les objets les plus abstraits qui composent le code d'un logiciel. Ils organisent structurellement le code. Par conséquent, ils peuvent refléter l'architecture du logiciel. Il est vrai cependant que les modules sont souvent nombreux et offrent ainsi une vue architecturale de pauvre qualité. Cependant, cette information est une des plus évidentes à extraire et à analyser. Il semble alors intéressant pour une approche de posséder cette fonctionnalité de base.

Description. Je suis rétro-architecte. Je dispose du code source de l'application. Je souhaite visualiser la structure organisationnelle de mon application. Je souhaite pouvoir naviguer au sein de cette structure et atteindre le code si besoin.

– **Scénario 2**

Idée sous-jacente. Les entités composant un logiciel se comptent par milliers. Il est nécessaire de regrouper ces entités au sein de composants de plus haut niveau. Ceci rejoint l'idée des modules. Cependant, il arrive fréquemment qu'une application soit mal structurée ou qu'elle dispose de trop de modules pour pouvoir les visualiser facilement.

Description. Je suis rétro-architecte. Je dispose du code source de l'application. Je souhaite visualiser une structure plus abstraite du logiciel que sa structure organisationnelle.

4.5.2 Description

L'approche GlobeCooker se compose de trois phases *(Cf. Figure 4.2)*. A chaque phase est associé un dictionnaire de stratégies. Ces stratégies permettent d'exécuter la phase concernée de multiples manières. D'autre part, l'approche repose sur la notion de métamodèles. Enfin, une des stratégies de modularisation possibles repose sur la notion de dominance. Ces trois aspects sont abordés dans les trois sous-sections suivantes.

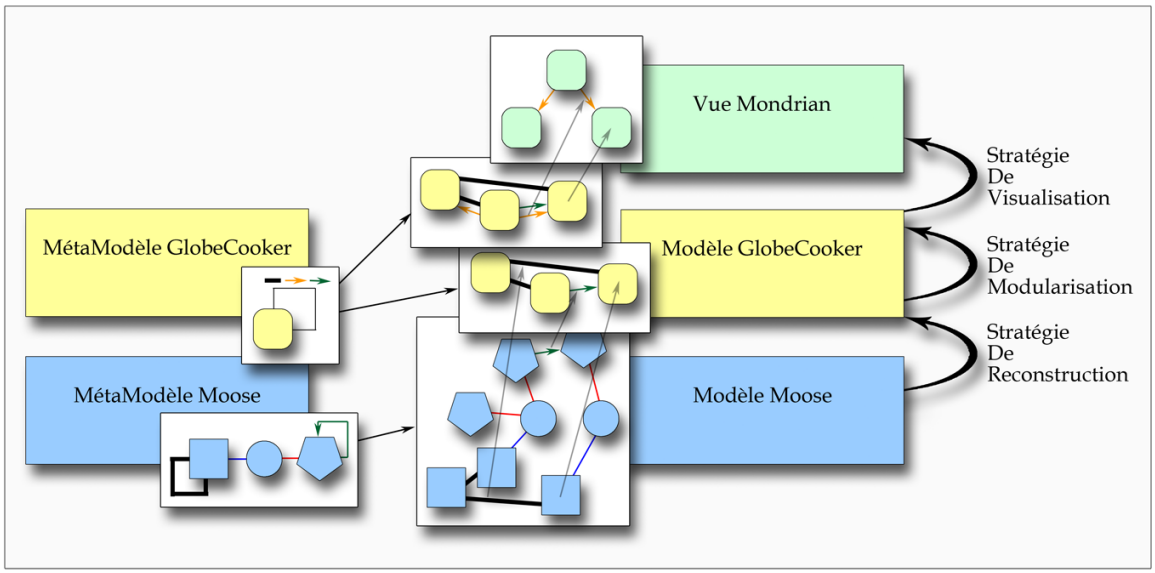

Fig. 4.2 – GlobeCooker

4.5.3 Métamodélisation

Le modèle. Le monde est complexe. Le logiciel aussi. De la même manière que l'on modélise le monde pour mieux le comprendre et le faire évoluer, on modélise un logiciel pour mieux le comprendre et le faire évoluer. Un *modèle* est une représentation de la réalité. La carte routière de la France est un exemple de modèles. Cette carte représente les villes, routes et ponts de la France et fait abstraction de ses voitures, chemins et arbres. Les modèles synthétisent l'information et simplifient ainsi la réalité. Le code informatique est verbeux. Par conséquent, les développeurs de logiciels communiquent à son sujet à l'aide de modèles. Ces modèles représentent et synthétisent le code.

Le métamodéle. Dans la plupart des domaines, des langages de modélisation existent. La légende d'une carte routière fait office de langage de modélisation pour un ensemble de cartes. Elle définit les concepts présents dans les cartes comme par exemple le concept ville. Un modèle est en réalité un ensemble d'instances de concepts. Sur la carte routière de la France, Annecy est une instance du concept ville, Lyon également et l'A46 est une instance du concept route. De la même manière, des langages de modélisation existent pour modéliser les logiciels. Ils sont appelés des *métamodèles*. La légende d'une carte est un métamodèle. Les métamodèles définissent les concepts présents dans les modèles. Dans le monde du logiciel, ces concepts sont ceux présents dans le code tels les concepts classe et méthode présents dans du code dit objet. Un modèle est ni plus ni moins qu'un ensemble d'instances de ces concepts, c'est à dire un ensemble de classes et de méthodes. En guise de rappel, Moose a pour but de favoriser la compréhension des logiciels. Il extrait les modèles représentatifs des logiciels en cours d'analyse, les stocke et les rend disponible à des outils d'analyse. Moose dispose de son propre métamodèle pour pouvoir exprimer des modèles.

Le métamétamodèle. Cependant, la multiplication des métamodèles, bien qu'elle soit nécessaire pour permettre la représentation de concepts liés à des technologies différentes, a rendu l'interfaçage entre outils basés sur la notion de modèles difficile. Suite à cette observation, la communauté a proposé un *métamétamodèle*. Il permet d'exprimer des métamodèles et facilitent leur manipulation et interfaçage. A noter qu'il n'existe pas de métamétamétamodèle. Le cycle s'arrête à ce niveau puisque le métamétamodèle est décris en lui-même.

La métadescription. Moose est un environnement de ré-ingénierie que l'on qualifie de *métadécrit*. Cette qualification stipule que Moose, c'est à dire l'implémentation de son métamodèle ainsi que l'implémentation de toutes ses fonctionnalités métiers, est lui même modélisé en accord avec un certain métamodèle. Cet aspect apporte beaucoup de flexibilité aux environnement en constante évolution et intégrant des outils de divers horizons.

Le métamodèle Moose. Le métamodèle Moose permet de représenter les concepts de base que l'on peut trouver dans le code *(Cf. Figure 4.3)*. Les classes, les méthodes, les modules, l'héritage entre classes ou bien encore l'invocation entre méthodes sont des exemples de concepts de base. Cet extrait du métamodèle de Moose stipule qu'une classe est composée de méthodes et d'attributs par exemple. Si l'on extrait le modèle du code de Moose, on va trouver que le code de Moose contient 800 classes environ et 3500 méthodes.

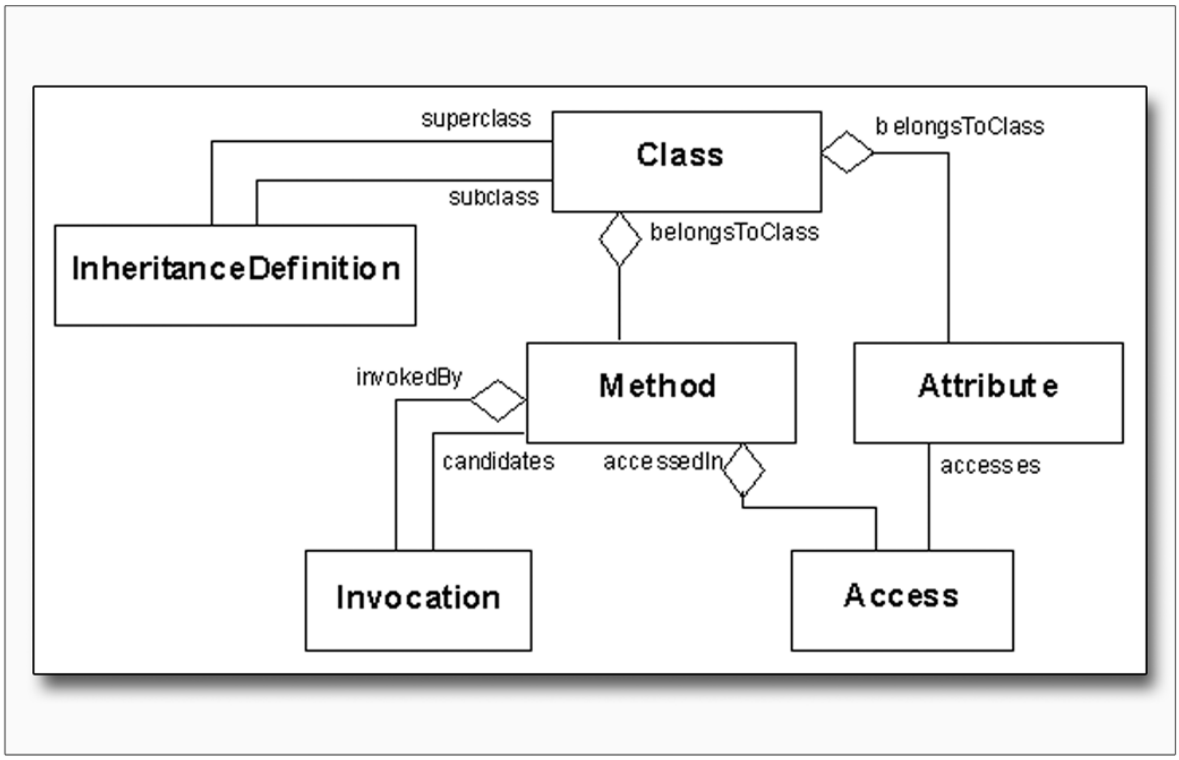

Fig. 4.3 – Le métamodèle Moose

Le métamodèle GlobeCooker. Le métamodèle Moose est par essence consacré seulement à la représentation d'objets présents dans le code. Il ne permet pas la représentation d'objets non présents dans le code tels que des objets architecturaux. Pour pallier à ce problème, j'ai implémenté un métamodèle minimal pour représenter les deux concepts de base d'une architecture logicielle, à savoir les composants et les relations *(Cf. Figure 4.4)*. Ce métamodèle met en évidence deux concepts. Le concept de composant et le concept de relation. Une relation lie deux composants entre eux. Il est à noter que les composants et relations peuvent être de différents types ce qui n'est pas stipulé sur la figure. Une relation peut être notamment une relation de contenance, de dépendance ou encore de description. D'autre part, il est également à noter que la notion fréquemment rencontrée de composants composites, c'est à dire pouvant contenir d'autres composants n'est pas explicitement stipulée dans ce métamodèle. Ceci se fait de manière différente en stipulant qu'il y a une relation de contenance entre deux composants.

Résumé. Le métamodèle Moose nécessite une extension pour représenter des concepts non présents dans le code. J'ai ainsi implémenté le métamodèle GlobeCooker. L'environnement Moose est métadécrit. Or, l'approche GlobeCooker s'y intègre. Par conséquent, le code de GlobeCooker qui comprend notamment l'implémentation du métamodèle GlobeCooker a été métadécrit de la même manière que Moose l'est.

4.5.4 Stratégies

Préambule. L'approche GlobeCooker repose sur trois phases. Chaque phase propose au rétro-architecte différentes stratégies d'exécution. Chaque phase est ci-dessous décrite selon un for-

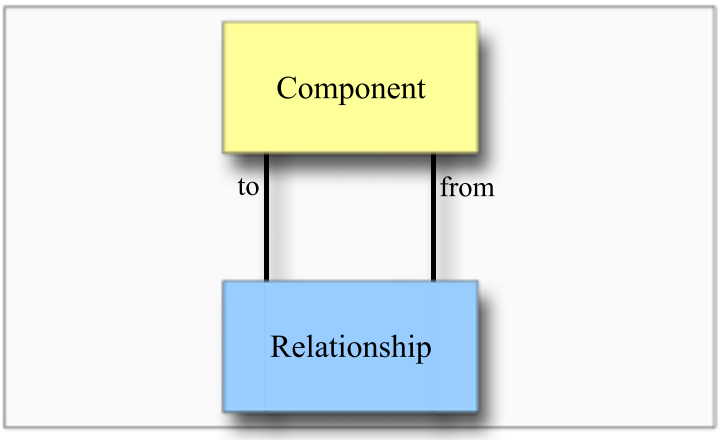

Fig. 4.4 – Le métamodèle GlobeCooker

malisme synthétique. Il comprend notamment toujours l'exemple d'une stratégie implémentée dans GlobeCooker. L'approche GlobeCooker requiert en entrée le modèle Moose du logiciel en cours d'investigation, c'est à dire ses classes, ses méthodes, ... Ainsi, la validation de l'approche repose sur un modèle Moose de test. Les concepts centraux de ce modèle sont visualisés ci-après à l'aide de l'outil de visualisation Mondrian *(Cf. Figure 4.5)*. Le modéle GlobeCooker référencé à maintes reprises ci-après fait toujours référence au même modèle et évolue au fil des phases. Ce modèle ne perd pas d'informations. Les stratégies de visualisation peuvent cependant quant elles omettre des concepts du modèles sans en altérer le modèle. Elles permettent dans le cadre de la présentation de GlobeCooker au sein de cette dissertation de mieux visualiser les changements connus par le modèle au cours du cycle d'exécution.

Phase de reconstruction

- Entrée : Un modèle Moose.
- Sortie : Un modèle GlobeCooker.
- Description : Cette phase consiste à associer des entités d'un modèle Moose à des entités d'un modèle GlobeCooker et par conséquent à regrouper, filtrer... des entités du modèle Moose.
- Exemple de stratégie : Associer chaque package d'un modèle Moose *(Cf. Figure 4.5)* à un composant d'un modèle GlobeCooker *(Cf. Figure 4.6)*. Créer les relations de contenance entre ces packages. Remonter les relations de dépendances entre méthodes du modèle Moose au niveau packages du modèle GlobeCooker.

Phase de modularisation

- Entrée : Un modèle GlobeCooker.
- Sortie : Un modèle GlobeCooker.
- Description : Cette phase consiste à transformer un modèle GlobeCooker en un modèle plus abstrait. De nombreux algorithmes savent en effet modulariser un graphe de composants et de relations en un graphe plus compact.
- Exemple de stratégie : Appliquer un algorithme d'analyse de la dominance sur un modèle GlobeCooker formé de composants et de relations *(Cf. Figure 4.7)* afin d'en obtenir un dont le niveau d'abstraction est plus élevé du style *(Cf. Figure 4.9)*. Ici on considère par exemple les relations de dépendances du modèle GlobeCooker. Ceci permettra de comparer la vue modularisée en une hiérarchie de relations de dominance entre packages *(Cf. Figure 4.9)* avec la vue hiérarchique des relations de contenance entre packages issue du code *(Cf. Figure 4.6)*.

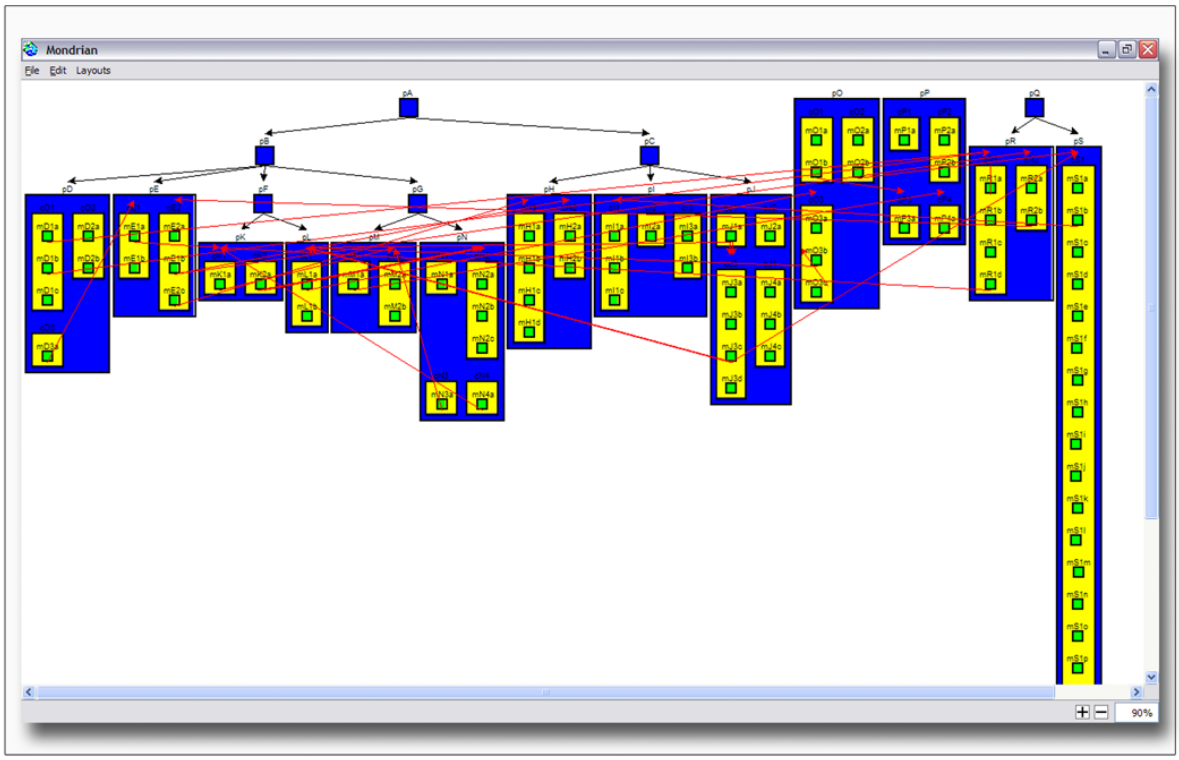

FIG. 4.5 – Un modèle Moose (Package : Bleu | Classe : Jaune | Méthode : Vert | Contenance : Noir | Invocation : Rouge)

Comme il sera vu en 4.5.5, l'algorithme implémenté ne fonctionne pas sur le modèle Moose d'étude. Dans le cadre de cette étude de cas, il faut imaginer le modèle GlobeCooker *(Cf. Figure 4.7)* dérivé du modèle Moose d'étude *(Cf. Figure 4.5)* devenir un modèle GlobeCooker hiérarchisé selon les packages par des relations de dominance comme illustré plus tard *(Cf. Figure 4.9)*. Le modèle GlobeCooker qui en résulte informe alors par exemple qu'un package domine d'autres packages. Les entités sont ainsi regroupées autour de leur dominateur respectif.

Phase de visualisation

- Entrée : Un modèle GlobeCooker.
- Sortie : Un modèle GlobeCooker.
- Description : Cette phase consiste à visualiser un modèle GlobeCooker. Les stratégies de visualisation choisissent le type de concepts qu'elles souhaitent visualiser et de quelles manières elles désirent le faire.
- Exemple de stratégie : La fenêtre de gauche de la figure *(Cf. Figure 4.8)* montre que la stratégie peut choisir de visualiser les composants et les relations de contenance entre ces composants selon une vue polymétrique [Lanz03], c'est à dire en reflétant les propriétés des composants à travers leurs dimensions visuelles.

Résumé. L'approche GlobeCooker est pragmatique. Elle est basée sur une succession de trois phases qui s'enchaînent logiquement. Les stratégies associées sont personnalisables et proches du code et de ses concepts.

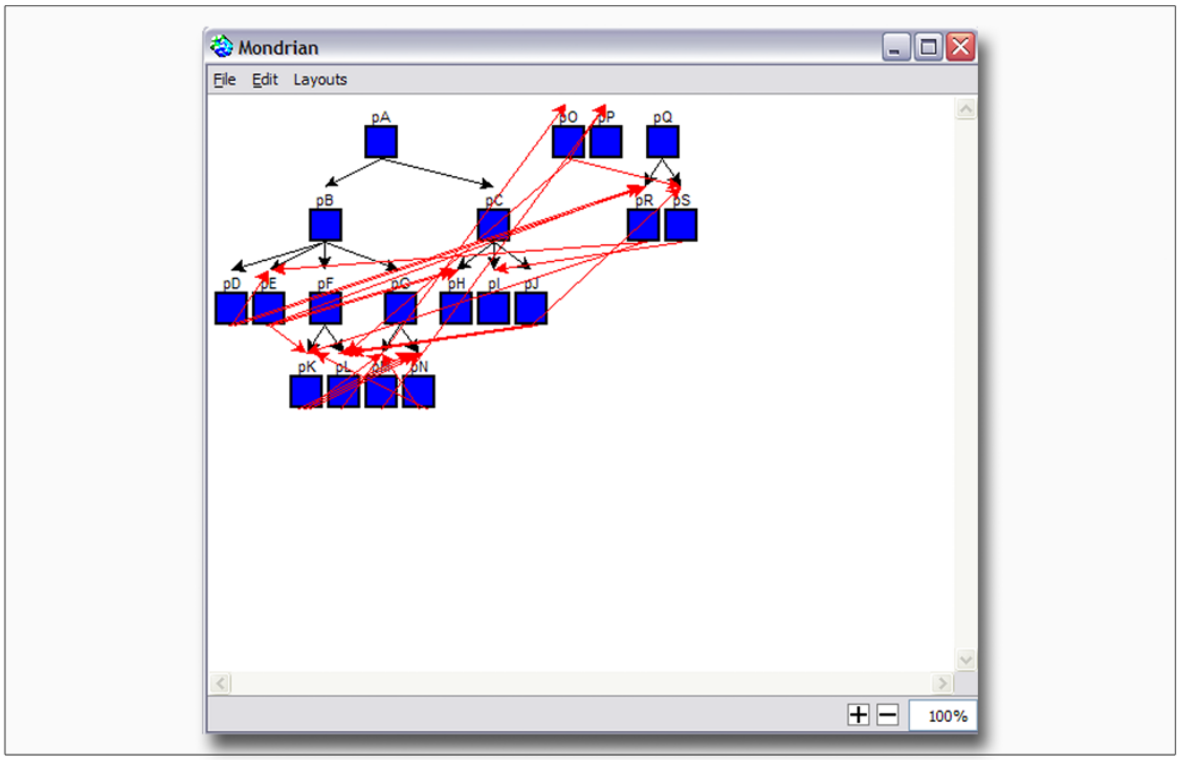

FIG. 4.6 – Le modèle structurel GlobeCooker illustrant la contenance et la dépendance (Composant : Bleu | Relation Dépendance : Rouge | Relation Contenance : Black)

4.5.5 Dominance

Préambule. Comme vu précédemment, l'équipe LSE souhaite implémenter des algorithmes de modularisation. Les raisons sont les suivantes. Ces algorithmes permettent tout d'abord d'élever le niveau d'abstraction du modèle d'un logiciel et ainsi de tendre vers son architecture. De plus, de tels algorithmes produisent des vues architecturales mesurables en terme de qualité par des critères mathématiques tels que la cohésion moyenne intra et inter composants. Ainsi il a été introduit une phase de modularisation au sein de GlobeCooker. L'implémentation d'une stratégie de modularisation basée sur l'analyse de la dominance est une stratégie de modularisation possible et explorée dans le cadre de cette deuxième mission. Ceci a nécessité la lecture de plusieurs publications sur le domaine [Leng79, Kosc00, Coop01, Trif01, Lund03].

Théorie. La dominance est une relation mathématique basée sur la théorie des graphes. Dans un graphe orienté G, un noeud A domine un noeud B si et seulement si tous les chemins de la racine R du graphe G pour se rendre au noeud B contiennent A *(Cf. Figure 4.9)*. De plus, si A domine B et si tous les noeuds qui dominent B dominent aussi A, alors A domine directement B. Enfin, si et seulement si A est l'unique prédécesseur de B dans le graphe G, alors A domine fortement B. Cette théorie permet de regrouper les noeuds d'un graphe selon qu'ils soient ou non sous l'égide d'un autre noeud.

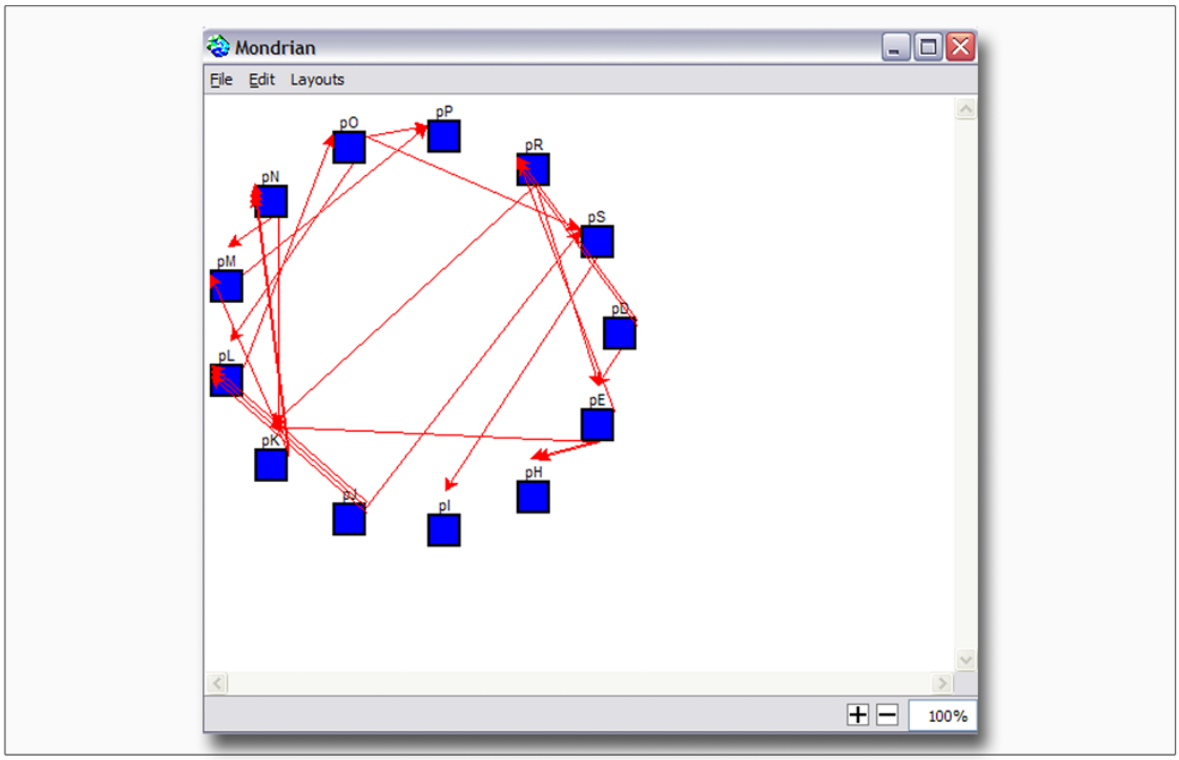

FIG. 4.7 – Le modèle comportemental GlobeCooker illustrant la dépendance (Composant : Bleu | Relation Dépendance : Rouge

Algorithme de Cooper. Un certain nombre de papiers présente un algorithme d'analyse de la dominance. J'ai opté pour celui considéré comme étant le plus simple et le plus rapide [Coop01]. J'ai ainsi implémenté cet algorithme. L'exemple présenté dans le papier m'a permis de valider l'implémentation de cet algorithme. Cependant, je me suis rendu compte que leur algorithme tel qu'il est présenté dans le papier ne fonctionne pas en réalité. Rien que sur l'exemple de leur papier, des situations conduisent l'algorithme à l'échec. Ainsi, l'implémentation actuelle de leur algorithme telle qu'il est spécifié dans le papier ne me permet pas de modulariser le modèle GlobeCooker *(Cf. Figure 4.7)* en un modèle hiérarchisé selon des relations de dominance comme illustré précédemment *(Cf. Figure 4.9)*.

Résumé. La relation de dominance permet de hiérarchiser un graphe d'entités en interaction. Ainsi des entités deviennent locales à d'autres dits dominateurs et peuvent ainsi être groupées autour de leur dominateur respectif. Malheureusement, l'algorithme tel qu'il est présenté dans le papier ne fonctionne pas.

4.5.6 Résultat

Deux scénarios ont guidé l'implémentation de GlobeCooker. Il est alors temps de vérifier que GlobeCooker satisfait ces deux scénarios. GlobeCooker satisfait le premier comme l'illustrent les deux figures précédentes : *(Cf. Figure 4.6)* et *(Cf. Figure 4.8)*. Quant au deuxième, GlobeCooker ne le satisfait pas car l'algorithme de dominance ne fonctionne pas.

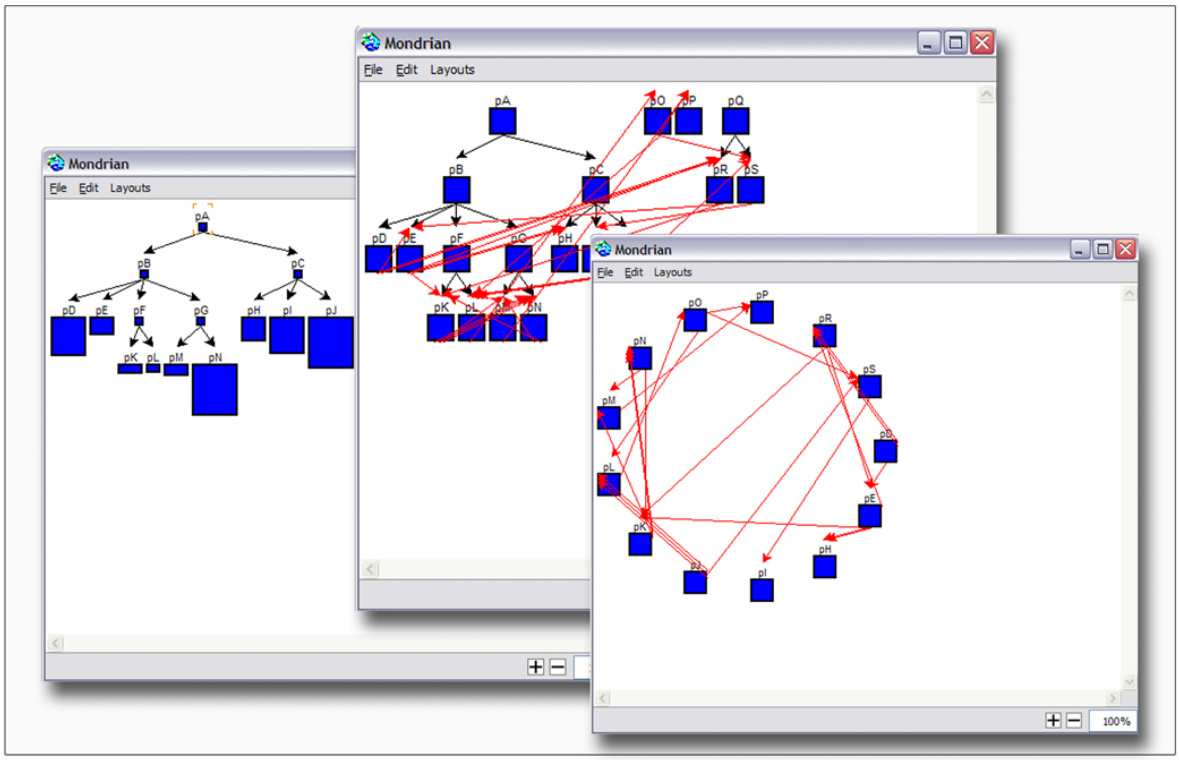

FIG. 4.8 – Le modèle GlobeCooker sous tous ses angles

4.6 Perspectives

L'implémentation de GlobeCooker est proche de l'implémentation d'un framework. L'équipe LSE pourra ainsi l'étendre pour valider ses approches si celles-ci respectent le cycle en phases présenté au cours de cette dissertation. Elle pourra notamment implémenter d'autres algorithmes de modularisation.

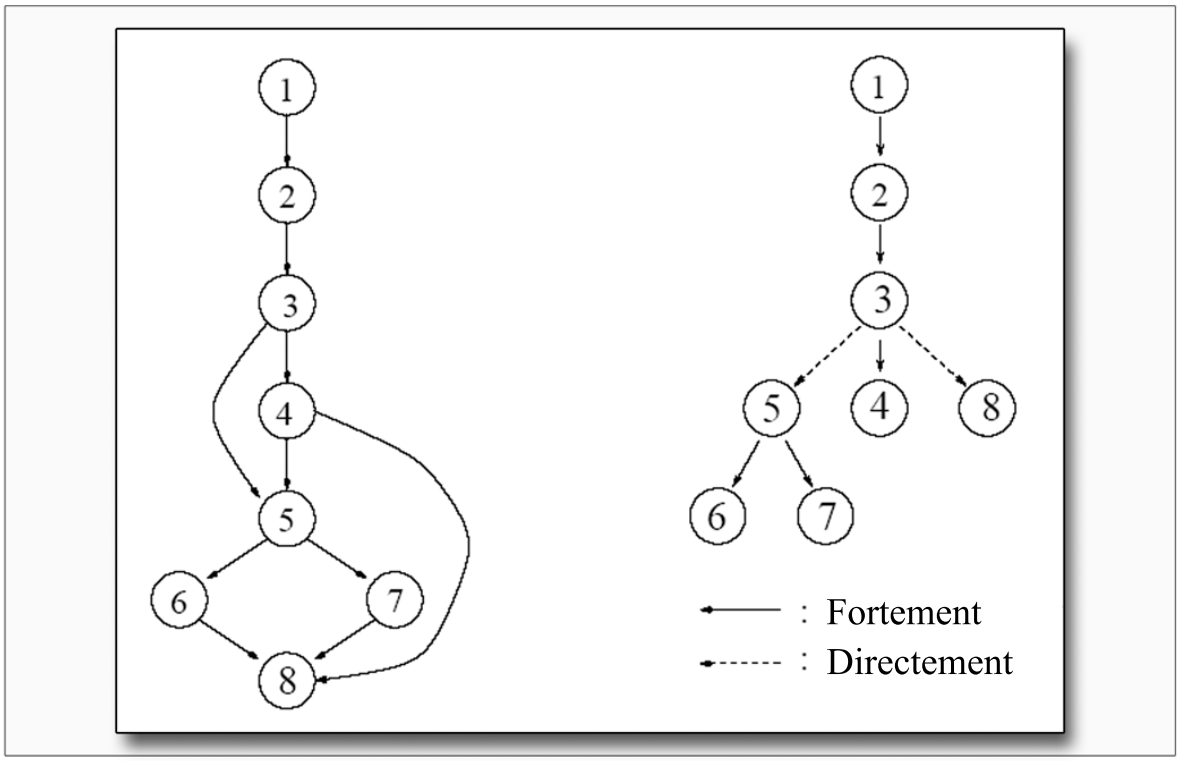

Fig. 4.9 – Analyse de la dominance

Chapitre 5

Initiation à la recherche

Pour évaluer ses qualités, pour comparer des environnements de travail différents et par conséquent pour mieux s'orienter, il est nécessaire de réaliser des expériences nouvelles et originales vis à vis des précédentes. A travers ce stage, j'ai souhaité vivre une expérience qu'il est difficile de vivre une fois ses études terminées : se mettre dans la peau d'un enseignant chercheur. Mon initiation à la recherche fait ainsi l'objet de ce chapitre. Plus précisément, je partage à travers ce chapitre mes réflexions personnelles sur des thèmes qui ont capté mon attention lors de cette expérience. Je tente également de les confronter au maximum à mon choix de ne pas réaliser de thèse à la suite de cette expérience.

5.1 Gagner sa liberté

J'apprécie tout particulièrement l'idée de pouvoir apprendre et partager, de disposer de contraintes temporelles faibles, de pouvoir profiter d'une grande flexibilité pour conduire sa recherche là où bon il nous semble et de sentir une communauté ouverte prête à vous soutenir derrière vous.

Cependant, la liberté se gagne. Le monde de la recherche est un monde que je trouve très politisé dans lequel le relationnel est prépondérant. Un chercheur doit se faire un nom au sein de la communauté. Au sein d'une société ceci est différent. Bien évidemment, un cadre doit faire sa place en interne. Cependant, en externe, il représente sa société avant de représenter sa personne. Au sein d'un laboratoire, un chercheur doit bien évidemment faire sa place en interne. Cependant, en externe, il représente également sa personne avant de représenter son laboratoire.

La liberté se gagne notamment à travers la réalisation d'une thèse. En ce qui me concerne, j'ai fait preuve de persévérance, d'autonomie et d'esprit d'analyse pour comprendre le domaine et m'y introduire intelligemment à travers l'écriture d'un état de l'art. En effet, l'écriture d'un état de l'art pour la communauté n'est pas l'écriture d'un état de l'art pour soi. La rigueur est une des qualités requises. L'investissement nécessité est de ce fait bien plus grand. La communauté sait apprécier l'investissement nécessaire pour un tel travail et sait en retour vous soutenir, vous faire un nom et par conséquent vous laisser une certaine liberté d'action. En imageant quelque peu, il est possible de dire que l'écriture d'un état de l'art ou d'une thèse est un moyen de faire

ses armes en dessinant la carte du monde et de recevoir en retour la liberté de circuler au sein de ce monde et de réaliser ses rêves.

En résumé, la communauté ne vous connaît pas et vous ne connaissez ni la communauté ni le domaine. La réalisation d'une thèse vous permet de résoudre le problème intelligemment. Dans ce monde politisé, votre liberté croit avec l'influence de votre nom. Personnellement, je manque de charisme et d'esprit stratégique pour faire ma place au sein de la communauté. Je reste persuadé que mon esprit scientifique, mon énergie ainsi que mon autonomie ne suffisent pas.

5.2 Savoir entreprendre

Indéniablement, j'ai manqué d'initiatives pour orienter le sujet de mon stage à son début. Cependant, il est difficile d'orienter le sujet de son stage quand l'environnement du stage, c'est à dire le monde de la recherche, vous est lui aussi inconnu.

Après cette expérience de six mois, je suis en mesure de proposer des axes de recherche que je souhaite explorer et de constituer un sujet de thèse. L'implémentation de mon approche GlobeCooker en fin de stage pose les bases d'un framework de rétro-architecturisation ouvert à de multiples techniques de rétro-architecturisation. Le regroupement d'entités autour de fonctionnalités est l'une des approches qui me semblent intéressantes à explorer.

Pour résumer, je ne parviens pas à être entreprenant dans un environnement dans lequel je ne maîtrise pas les finalités, les variables, la connaissance... Faire face à de l'incertitude me freine énormément. Or réaliser une thèse c'est baigner dans l'incertitude pendant trois ans. Même si l'écriture d'un tel état de l'art me permet de modéliser le domaine, je reste persuadé que la trop grande liberté d'action, que l'absence d'objectifs stables et que l'imprédictibilité des recherches entreprises sont des aspects dangereux de la thèse.

5.3 Savoir collaborer

Pour finir, j'ai fait face à un dilemme au cours de mon stage. Mon souhait le plus fort était de collaborer activement avec la communauté. Cependant, l'investissement personnel et individuel requis pour l'écriture de l'état de l'art m'a poussé à refuser les propositions de mon responsable de me joindre à la communauté notamment à rejoindre son ancienne équipe à l'Université de Berne spécialisée dans la rétro-conception. La crainte de ne pas être à la hauteur et la dimension internationale de tels projets expliquent également en partie ce refus.

Je souhaite nuancer tout de même mes propos en soulignant que je ne considère pas ceci comme de la résistance au changement. En effet, le changement il a été au rendez-vous. L'expérience vécue à travers mon stage fut pour moi totalement nouvelle. Pour la mener à terme, j'ai su adopter de nouvelles méthodologies mais également su m'adapter à un environnement au sein duquel les aspects relationnels, collaboratifs et fonctionnels ne me convenaient pas. Cependant, les challenges à relever pour rejoindre la communauté étaient pour moi trop lourd à porter. J'ai

préféré ne pas tout bousculer d'un coup. Une expérience de développement aurait été en ce qui me concerne plus propice à une collaboration active.

Ainsi, avant d'entreprendre une collaboration il me semble nécessaire d'une part d'avoir compris le domaine et d'autre part de faire face à de vrais problèmes. Maintenant que l'état de l'art est terminé et que je commence à pouvoir formaliser mes problèmes, je suis en mesure de surmonter mes craintes et à exposer mes problèmes à des chercheurs du domaine.

5.4 En résumé

Cette expérience me permet de déduire que je suis en mesure de gagner ma liberté quand il est stratégique de le faire. Je suis en mesure progressivement d'entreprendre et de collaborer activement avec la communauté une fois les différents aspects de l'environnement maîtrisés. Cependant, comme le révèle la fin de la précédente phrase je suis incapable de faire face à l'incertitude. Cet aspect m'a poussé à refuser l'idée de réaliser une thèse.

Chapitre 6

Conclusion

Contributions. La rétro-architecturisation est une activité présente dans le cycle de développement d'un logiciel. Sa finalité est de comprendre au niveau architectural un logiciel déjà existant en vue de le faire évoluer. Les contributions de mon stage sont au nombre de deux. Un état de l'art sur la rétro-architecturisation et une approche de rétro-architecturisation résultent en effet de mon stage.

Initiation à la recherche. L'écriture de l'état de l'art a nécessité un investissement personnel important alors que l'implémentation de l'approche a nécessité une rapidité d'exécution. Cette expérience, nouvelle vis à vis de mes expériences précédentes, fut formatrice. Gagner sa liberté et savoir entreprendre sont des qualités travaillées au cours de ce stage qui me seront sans aucun doute utiles pour le métier d'ingénieur vers lequel je m'oriente. Mener des projets dans des environnements inconnus et instables est une des qualités requises pour poursuivre dans le monde de la recherche. Cette qualité que je juge importante pour réaliser une thèse me fait défaut. Je me suis ainsi décidé à ne pas réaliser de thèse. Cependant cette qualité est aussi un atout pour le métier d'ingénieur. Je me dois de progresser sur cet aspect d'autant plus que ma marge de progression est à priori grande.

Perspectives. De par sa qualité, l'état de l'art devrait en premier lieu servir de support aux travaux de l'équipe LSE. Il devrait de plus être prochainement publié à la fois dans une conférence et une revue internationales. Enfin, il devrait être référencé par les articles du domaine qui sortiront dans les prochaines années. Quant à l'approche de rétro-architecturisation, elle introduit un framework qui peut accueillir bon nombre de nouvelles approches de rétro-architecturisation.

Bibliographie

[Chik90] Elliot J. Chikofsky and James H. Cross II. "Reverse Engineering and Design Recovery : A Taxonomy". *IEEE Software*, pp. 13–17, Jan. 1990.

[Coop01] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. "A Simple, Fast Dominance Algorithm". In : *Software–Practice and Experience*, 2001.

[Duca05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. "Moose : a Collaborative and Extensible Reengineering Environment". In : *Tools for Software Maintenance and Reengineering*, pp. 55–71, Franco Angeli, Milano, 2005.

[Duca06] Stéphane Ducasse and Tudor Gîrba. "Using Smalltalk as a Reflective Executable Meta-Language". In : *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, 2006. To appear.

[Duca99] Stéphane Ducasse and Serge Demeyer, Eds. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, Oct. 1999.

[Girb05] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, Nov. 2005.

[Kosc00] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[Lanz03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[Lehm85] Manny Lehman and Les Belady. *Program Evolution : Processes of Software Change*. London Academic Press, London, 1985.

[Leng79] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". pp. 121–141, ACM Press, New York, NY, USA, 1979.

[Lien80] Bennett Lientz and Burton Swanson. *Software Maintenance Management*. Addison Wesley, Boston, MA, 1980.

[Lund03] Lundberg and Löwe. "Architecture Recovery by Semi-Automatic Component Identification". *Electr. Notes Theor. Comput. Sci.*, Vol. 82, No. 5, 2003.

[McKe84] J. R. McKee. "Maintenance as a Function of Design". In : *Proceedings of AFIPS National Computer Conference*, pp. 187–193, 1984.

[Mens06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. "Co-evolving Code and Design with Intensional Views – A Case Study". *Journal of Computer Languages, Systems and Structures*, Vol. 32, No. 2, pp. 140–156, 2006.

[Meye06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. "Mondrian : An Agile Visualization Framework". In : *ACM Symposium on Software Visualization (SoftVis 2006)*, 2006. To appear.

[Somm96] Ian Sommerville. *Software Engineering.* Addison Wesley, fifth Ed., 1996.

[Trif01] Trifu. *Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems.* PhD thesis, Univ. Karlsruhe, 2001.

Les références spécifiques à la rétro-architecturisation se trouvent dans l'état de l'art présent en annexe.

Annexes

Software Architecture Reconstruction: A Process-Oriented Taxonomy

STÉPHANE DUCASSE
DAMIEN POLLET
LOÏC POYET
LISTIC Logiciels et Systèmes – Language and Software Evolution Group
Université de Savoie, France

To maintain and understand large applications, it is crucial to know their architecture. The first problem is that architectures are not explicitly represented in the code as classes and packages. The second problem is that successful applications evolve over time so their architecture inevitably drift. Reconstructing and checking whether the architecture is still valid is therefore an important aid. While there is a plethora of approaches and techniques supporting architecture reconstruction, there is no comprehensive state of the art and it is often difficult to compare the approaches. This article presents a state of the art in software architecture reconstruction approaches.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*Software Architecture Reconstruction*

General Terms: Design, Measurement

Additional Key Words and Phrases: Architecture Reconstruction, Packages, Visualization

1. INTRODUCTION

Software architecture acts as a shared mental model of a system expressed at a high-level of abstraction [Holt 2001]. By leaving details aside, this model plays a key role as a bridge between requirements and implementation [Garlan 2000]. It allows you to reason architecturally about a software application during the various steps of the software life cycle. According to Garlan, software architecture plays an important role in at least six aspects of software development: understanding, reuse, construction, evolution, analysis and management [Garlan 2000].

Software architecture is thus crucial for software development. The first problem is that architectures are not explicitly represented in the code as classes and packages. The second problem is that successful software applications are doomed to continually evolve and grow [Lehman and Belady 1985]; and as a software application evolves and grows, so does its architecture. The conceptual architecture often becomes inaccurate with respect to the implemented architecture; this results in architectural erosion [Medvidovic et al. 2003; Perry and Wolf 1992], drift [Perry and Wolf 1992], mismatch [Garlan et al. 1995], or chasm [Riva 2004].

Software architecture reconstruction (SAR) is the reverse engineering process

Authors' address: Listic – Esia, B.P. 806, 74016 Annecy cedex, France

that aims at reconstructing viable architectural views of a software application. Krikhaar specified five SAR maturity levels: initial, described, redefined, managed, optimized [Feijs et al. 1998; Krikhaar 1999]. While SAR is rarely an end in itself, it supports software development by providing high-level views of the investigated software application [Harris et al. 1995]. For example, these views help identify product line commonalities and variabilities [Stoermer and O'Brien 2001], or check their conformance to the source code [Murphy et al. 1995; Murphy 1996].

Several approaches and techniques have been proposed in the literature to support SAR. Mendonça *et al.* presented a first raw classification of SAR environments based on a few typical scenarios [ca and Kramer 1996]. O'Brien *et al.* surveyed SAR practice needs and approaches [O'Brien et al. 2002]. Still, there is no comprehensive state of the art and it is often difficult to compare the approaches. This article presents a state of the art of software architecture reconstruction approaches (SAR). It takes the perspective of a reverse engineer who would like to reconstruct the architecture of an existing application and would like to know which tools or approach to consider. We structure the field around the goals, the process, the inputs, the techniques and the outputs of SAR approaches.

About selecting the approaches. In this paper, we select works in two steps. First, in addition to works that are extracting architectural information, we also consider approaches that do not specifically extract architecture but related artefacts such as design patterns, features or roles since they often crosscut or are low-level constituents of the architecture. We also consider approaches that visualize programs since they are often the basis for abstracting and extracting architectural views, but we limit ourselves to the program visualization approaches that support the overall extraction process.

In the second step, we support the comparison of the approaches with a table for each axis that structures this survey. We only list in the tables works that are the most concerned about architectural extraction. For the sake of space, we consider only two categories of works: the important ones *i.e.* those which influenced other works or were precursors, and the original works taking a specific perspective or approach to the general problem. Such a second category is interesting because it opens space in the survey.

Again for the sake of space, we do not take into account works like ArchJava [Aldrich et al. 2002] that extend traditional languages to mix architectural and programming elements or other architectural description languages, since in such cases the architecture is not extracted from existing applications. Furthermore for the same reason we also exclude from this survey approaches proposing general methodology or guidelines that do not stress a specific point to support software architecture reconstruction [Dueñas et al. 1998; Svetinovic and Godfrey 2001; Knodel et al. 2005]. As software architecture is a blurry concept by definition it is hard to make a clear cut.

Section 2 first stresses some key vocabulary definitions and the challenges of software architecture reconstruction. Section 3 describes the criteria that we adopted in our taxonomy. Sections 4 to 8 then cover each of these criteria. Before concluding, Section 9 surveys the extraction of artefacts related to software architecture such as design patterns and features.

2. SAR CHALLENGES

Before going into depth on the challenges of SAR, we feel the need to clarify the vocabulary.

2.1 Vocabulary

Software architecture. IEEE defines *software architecture* as *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"* [IEEE 2000]. This definition is closely related to the definition of Shaw, Perry and Garlan [Shaw and Garlan 1996].

Architectural style. A software architecture often conforms to an *architectural style*, that is a class of architectures or a pattern of structural organization. An architectural style is *"a vocabulary of components and connector types, and a set of constraints on how they can be combined"* [Shaw and Garlan 1996].

Architectural views and viewpoints. We can *view* a software architecture from several *viewpoints* since the different *system stakeholders* have different expectations or *concerns* about the system [Kruchten 1995; IEEE 2000].

View. A view is *"a representation of a whole system from the perspective of a related set of concerns"* [IEEE 2000].

Viewpoint. A viewpoint is *"a specification of the conventions for constructing and using a view. A pattern or a template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis"* [IEEE 2000].

Conceptual architecture. This term refers to the architecture that exists in human minds or in the software documentation [Tran and Holt 1999; Riva 2004]. In the literature, conceptual architecture is also qualified as *idealized* [Harris et al. 1995], *intended* [Woods et al. 1999; Riva 2004], *as-designed* [Kazman and Carriere 1999; Tran and Holt 1999] or *logical* [Medvidovic and Jakobac 2006].

Concrete architecture. This term refers to the architecture that is derived from source code [Tran and Holt 1999; Riva 2004]. It is also known as the *as-implemented* [Kazman and Carriere 1999; Riva 2004], *as-built* [Tran and Holt 1999; Harris et al. 1995], *realized* [Woods et al. 1999] or *physical* [Medvidovic and Jakobac 2006] architecture.

Software architecture reconstruction (SAR). Software architecture reconstruction is a reverse engineering approach that aims at reconstructing viable architectural views of a software application. The literature uses several other terms to refer to SAR: *reverse architecting*, or architecture *extraction*, *mining*, *recovery* or *discovery*. The last two terms are more specific than the others [Medvidovic et al. 2003]: *recovery* refers to a bottom-up process while *discovery* refers to a top-down process (see Section 5).

2.2 Challenges

SAR is a multidisciplinary activity which covers several research areas centered around software engineering and maintenance of applications. In this context, two sources of information are considered: human expertise and source code artifacts.

On the one hand, human expertise is primordial to treat architectural concepts. In such a context, knowledge of business goals, requirements, product family reference architectures, or design constraints is useful to assist SAR. However, when we take human knowledge into consideration, several problems appear:

(1) Because of the high rate of turnover in experts and the lack of complete up-to-date documentation, the conceptual architecture in human minds is often obsolete, inaccurate, incomplete, or at an inadequate abstraction level. SAR should take into account the quality of the information.
(2) When reconstructing an architecture, system stakeholders have various concerns such as performance, reliability, portability or reusability; SAR should support multiple architectural viewpoints.
(3) Reverse engineers sometimes get lost in the increasing complexity of software. SAR needs to be interactive, iterative and parameterizable.

On the other hand, source code is one of the few trustworthy reliable sources of information about the software application which contains its actual architecture. However, reconstructing the architecture from the source code raises several problems:

(1) The large amount of data held by the source code raises scalability issues.
(2) Since the considered systems are typically large, complex and long-living, SAR should handle development methods, languages and technologies that are often heterogeneous and sometimes interleaved.
(3) Architecture is not explicitly represented at the source code level. In addition, language concepts such as polymorphism, late-binding, delegation, or inheritance make it harder to analyze the code [Wilde and Huitt 1992; Dunsmore et al. 2000]. The extraction of a relevant architecture is then a difficult task.
(4) The nature of software raises the questions of whether dynamic information should be extracted as the system is running, and then how do behavioral aspects appear in the architecture.

The major challenges of software architecture extraction are thus in abstracting, identifying and presenting higher-level views from lower-level and often heterogeneous information.

3. SAR TAXONOMY AXES

Mendonça and Kramer [ca and Kramer 1996] proposed a rough classification of SAR environments and distinguished five families based on the purpose of the approaches: filtering and clustering, compliance checking, analyser generators, program understanding and architecture recognition. O'Brien *et al.* surveyed SAR practice needs and approaches [O'Brien et al. 2002]. They proposed a pragmatic approach to classify SAR approaches. They introduced recurring practice scenarios

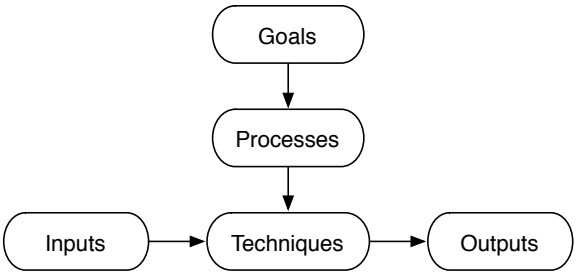

Fig. 1. A process-oriented taxonomy flow

that SAR approaches have to fulfill to improve SAR. Gallagher *et al.* [Gallagher et al. 2005] proposed a framework to assess architectural visualization tools and compare a couple of tools. Guéhéneuc *et al.* [Guéhéneuc et al. 2006] proposed a comparative framework for design recovery tools and compared three approaches.

We propose a deeper classification based on the life time of SAR approaches presented in Figures 1 and 2: intended goals, followed processes, required inputs, used techniques and expected outputs. Our taxonomy treats a larger number of approaches than the previous attempts at classifying the field. In particular, we analyze a broad range of work including program visualization, design patterns, and features extraction, since these works are related to the notion of architecture. We also mention some works that could be considered borderline, but we do not provide in depth comparisons for space reasons.

Goals. SAR is considered by the community as a proactive approach to answer stakeholder business goals [Dueñas et al. 1998; Stoermer et al. 2003]. The reconstructed architecture is the basis for redocumentation, reuse investigation and product line migration, or implementation and architecture co-evolution. Some approaches do not extract the architecture itself but related and orthogonal artifacts that provide valuable additional information to engineers such as design patterns, roles or features.

Processes. We distinguish three kinds of SAR processes based on their flow to identify an architecture: bottom-up, top-down or hybrid.

Inputs. Most SAR approaches are based on source code information and human expertise. However, other kinds of information can be exploited: dynamic information or historical information. In addition, not all approaches support the specification and use of architectural styles and viewpoints which are the paramount of architecture.

Techniques. The research community has explored various techniques to reconstruct architecture that we tentatively classify according to their level of automation.

Outputs. While all SAR approaches intend to provide architectural views, some of them however produce other valuable outputs such as conformance data.

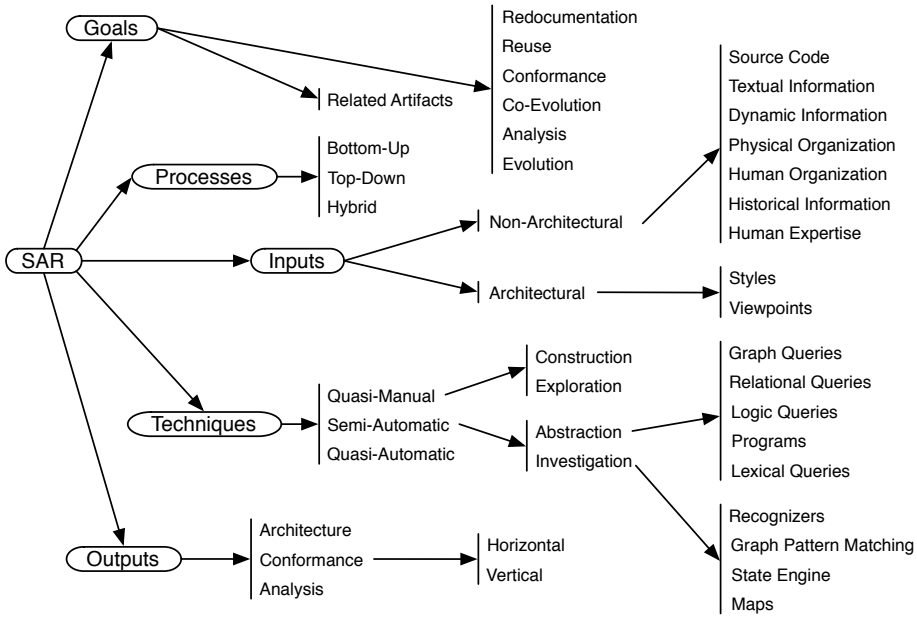

Fig. 2. A process-oriented taxonomy

4. GOALS

To put in perspective the goals of SAR approaches, we briefly present the general goals of software architecture. According to Garlan, software architecture contributes to six main goals of software development [Garlan 2000]:

Understanding. Architectural views describe a software system at a level of abstraction high enough to understand its overall design, to reason about it and make decisions taking into account its design constraints, quality attributes, rationale, possible bottlenecks, etc.

Reuse. Architectural views strongly highlight candidates for reuse such as components, frameworks and patterns, making easier software part reuse.

Construction. Architectural views are at a high-level of abstraction, allowing developers to focus their attention on the implementation of major components and relationships and iteratively to refine it.

Evolution. Architectural views make explicit the current constraints, and better expose how the software application is expected to evolve.

Analysis. By building on the high abstraction level of architectural views, new useful analyses can be performed, such as style conformance, dependence analysis, or quality attribute analysis.

Management. The clearer is the view of the software system, the more successful will be the development task.

Dali [Kazman and Carriere 1999; Kazman et al. 2001]	red				ana	
ARMIN [Kazman et al. 2003]	red				ana	
MAP [Stoermer and O'Brien 2001]	red	reus				
QADSAR [Stoermer et al. 2003; Stoermer et al. 2006]	red				ana	
ARM [Guo et al. 1999]	red					
Gupro [Ebert et al. 2002]	red					
RMTool [Murphy et al. 1995; Murphy 1996]	red		cnf			
SAVE [Miodonski et al. 2004; Naab 2005]	red		cnf	coev		
PuLSE/SAVE [Knodel et al. 2006; Pinzger et al. 2004]	red	reus	cnf	coev	ana	
W4 [Hassan and Holt 2004]	red		cnf			
PBS/SBS [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999]	red					
— [Tran and Holt 1999]	red		cnf	coev		
— [Bowman and Holt 1998]	red		cnf			
ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]	red		cnf			
ART [Fiutem et al. 1999]	red					
X-Ray [Mendonça and Kramer 2001]	red					
ARES [Eixelsberger et al. 1998]	red	reus				
SARTool [Feijs et al. 1998; Krikhaar 1999]	red		cnf			evo
Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]	red					
ArchView [Pinzger 2005]	red					
Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]	red					evo
Bunch [Mancoridis and Mitchell 1998; Mitchell and Mancoridis 2006]	red					evo
ArchVis [Hatch 2004]	red					
— [Lundberg and Löwe 2003]	red					
Bauhaus [Koschke 2000; Eisenbarth et al. 2003; Christl et al. 2005]	red		cnf			
Alborz [Sartipi 2003]	red					
URCA [Bojic and Velasevic 2000]	red					
Symphony/Nimeta [van Deursen et al. 2004]/[Riva 2004]	red		cnf		ana	
Cacophony [Favre 2004]	red					
Softwarenaut/Hapax [Lungu et al. 2006]/[Lungu et al. 2005; Kuhn et al. 2006]	red					
Softwarenaut [Lungu et al. 2006]	red					
Intensive [Wuyts 2001; Mens et al. 2006]	red		cnf	coev		
DiscoTect [Yan et al. 2004]	red		cnf			
— [Huang et al. 2006]	red			coev	ana	
— [Pashov and Riebisch 2004]	red					

red re-documentation · reus reuse · cnf conformance
coev co-evolution · ana analysis · evo evolution

Table I. SAR Goals Overview

4.1 SAR Goals

Several authors categorized architecture roles in software development [Garlan 2000]; in particular, Kazman *et al.* have a pragmatic categorization of business goals [Kazman and Bass 2005]. In the context of maintenance, a software architecture reconstruction process should answer stakeholder business objectives. It is a proactive process realized for future forward engineering tasks. Knodel *et al.* identified ten distinct purposes or needs which we grouped in six main categories described below [Knodel et al. 2006]. SAR approaches match various often inter-

leaved goals:

Redocumentation and understanding. The primary goal of SAR is to re-establish software abstractions. Recovered architectural views document software applications and help reverse engineers understand them. For instance, the software bookshelf introduced by Finningan *et al.* illustrates this goal [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999]. Svetinovic *et al.* state that not only the recovered architecture is important, but also its rationale, *i.e.* why it is as it is [Svetinovic and Godfrey 2001]. They focus on the architecture rationale forces to recover the decisions made, their alternatives, and why each one was or was not chosen.

Reuse investigation and product line migration. Systematic reuse has not yet been achieved. Software product lines allow one to share commonalities among products while getting custom products. Architectural views are useful to identify commonalities and variabilities among products in a line [Stoermer and O'Brien 2001; Pinzger et al. 2004; Eixelsberger et al. 1998].

Conformance. To evolve a software application, it seems hazardous to use the conceptual architecture because it is often inaccurate with respect to the concrete one. In this case, SAR is a means to check conformance between the conceptual and the concrete architectures. Murphy *et al.* introduced the reflexion model and RMTool to bridge the gap between high-level architectural models and the system's source code [Murphy et al. 1995; Murphy 1996]. Using SAR, reverse engineers can check conformance of the reconstructed architecture against rules or styles like in the SARTool [Feijs et al. 1998; Krikhaar 1999], Nimeta [Riva 2004], DiscoTect [Yan et al. 2004], Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]and DAMRAM [Medvidovic et al. 2003].

Co-evolution. Architecture and implementation are two levels of abstraction that evolve at different speeds. Ideally these abstractions should be synchronized to avoid architectural drift. Tran *et al.* propose a method to repair evolution anomalies between the conceptual and the concrete architectures, possibly altering either the conceptual architecture or the source code [Tran and Holt 1999]. To dynamically maintain this synchronization, Wuyts uses logic meta-programming [Wuyts 2001], and Mens *et al.* use intensional source-code views and relations through Intensive [Wuyts 2001; Mens et al. 2006]; Favre uses metaware [Favre 2003]; Huang *et al.* use a reflection mechanism too that is based on dynamic information. They keep the code synchronized with the reconstructed architectural views and vice versa.

Analysis. An analysis framework may steer a SAR framework so that it provides required architectural views to compute architectural quality analyses. Such analysis frameworks assist stakeholders in their decision-making processes. In ArchView [Pinzger 2005], SAR and evolution analysis activities are interleaved. QADSAR is a tool that is driven by several analyses [Stoermer et al. 2003; Stoermer et al. 2006]. For example, a system is able to accept and execute user commands 250 ms after power-on. To check it, this scenario is given to the SAR framework. This framework reconstruct architectural views that highlight threads, waiting points and performance properties to allow the analysis framework to perform performance analyses. Moreover, flexible SAR environments such as Dali [Kazman and Carriere 1999; Kazman et al. 2001], ARMIN [Kazman et al. 2003]or Gupro [Ebert et al.

2002]support architectural analysis methods like SAAM [Kazman et al. 1994] or ATAM [Kazman et al. 1998] by exporting the extracted architectures to dedicated tools.

Evolution and maintenance. SAR is often a first step towards software evolution and maintenance. Here we use the term evolution to mean the study of the architecture to support the application evolution and not to study the evolution of the application. Understanding the inputs on which an approach is based is key to make this distinction, some approaches consider the system history to understand its evolution but not in the exact goal of supporting directly the evolution of an application. Focus subscribes to that perspective; its strength is that the SAR scope is reduced to the system part which should evolve [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]. Krikhaar *et al.* also introduced a two-phase approach for evolving architecture based on SAR and on change impact analyses [Feijs et al. 1998; Krikhaar 1999]. Huang *et al.* also consider SAR in an evolution and maintenance perspective [Huang et al. 2006].

4.2 Related and Orthogonal Artifacts

Some SAR approaches do not directly extract the architecture of an application but correlated artifacts that crosscut and complement the architecture. Such artifacts are *design patterns*, *concerns*, *features*, *aspects*, or *roles* and *collaborations*. While these artifacts are not the architecture itself (*i.e.* view points or architecture), they provide valuable information about it [Beck and Johnson 1994].

Patterns play a key role in software engineering at different levels of abstraction: architectural patterns, design patterns or idioms [Beck and Johnson 1994; Buschmann et al. 1996]. Some reverse engineering approaches are thus based on design pattern identification [Kramer and Prechelt 1996; Antoniol et al. 1998; Wuyts 1998; Heuzeroth et al. 2003; Beyer and Lewerentz 2003; Arévalo et al. 2004; Guéhéneuc et al. 2004].

Concerns are the stakeholders' criterion for modularizing a software application into manageable and comprehensible parts [Robillard and Murphy 2002; Coelho and Murphy 2006]. Features and aspects are more specific concerns that are also extracted from existing applications [Wilde and Scully 1995; Eisenbarth et al. 2003; Pashov and Riebisch 2004; Riva 2004; Greevy and Ducasse 2005; Smith and Nair 2005]. In the context of this paper we do not take aspect mining into account since a couple of surveys have already been published on the subject [Ceccato et al. 2005; Kellens and Mens 2005; Nora et al. 2006].

Roles and collaborations are crucial to object-oriented programming: to achieve tasks, objects collaborate with each other and during these collaborations they play specific roles [Reenskaug 1996]. However roles and collaborations are not explicit but buried into programs. Both Wu [Wu et al. 2004] and Richner [Richner and Ducasse 2002] support the extraction of roles and collaborations using dynamic information following the work of Lange [Lange and Nakamura 1995].

These approaches consider that high-level knowledge is necessary to extract valuable information at the architectural level, therefore we considered them in the survey as explicit goals.

Table I classifies works according to their goals.

Dali [Kazman and Carriere 1999; Kazman et al. 2001]	bottom-up		
ARMIN [Kazman et al. 2003]	bottom-up		
MAP [Stoermer and O'Brien 2001]			hybrid
QADSAR [Stoermer et al. 2003; Stoermer et al. 2006]			
ARM [Guo et al. 1999]			hybrid
Gupro [Ebert et al. 2002]	bottom-up		
RMTool [Murphy et al. 1995; Murphy 1996]		top-down	
SAVE [Miodonski et al. 2004; Naab 2005]		top-down	
PuLSE/SAVE [Knodel et al. 2006; Pinzger et al. 2004]		top-down	
W4 [Hassan and Holt 2004]		top-down	
PBS/SBS [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999]			hybrid
— [Tran and Holt 1999]			hybrid
— [Bowman and Holt 1998]			hybrid
ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]			hybrid
ART [Fiutem et al. 1999]			hybrid
X-Ray [Mendonça and Kramer 2001]			hybrid
ARES [Eixelsberger et al. 1998]	bottom-up		
SARTool [Feijs et al. 1998; Krikhaar 1999]	bottom-up		
Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]	bottom-up		
ArchView [Pinzger 2005]	bottom-up		
Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]			hybrid
Bunch [Mancoridis and Mitchell 1998; Mitchell and Mancoridis 2006]	bottom-up		
ArchVis [Hatch 2004]	bottom-up		
— [Lundberg and Löwe 2003]	bottom-up		
Bauhaus [Koschke 2000; Eisenbarth et al. 2003; Christl et al. 2005]			hybrid
Alborz [Sartipi 2003]			hybrid
URCA [Bojic and Velasevic 2000]	bottom-up		
Symphony/Nimeta [van Deursen et al. 2004]/[Riva 2004]			hybrid
Cacophony [Favre 2004]			hybrid
Softwarenaut/Hapax [Lungu et al. 2006]/[Lungu et al. 2005; Kuhn et al. 2006]	bottom-up		
Softwarenaut [Lungu et al. 2006]	bottom-up		
Intensive [Wuyts 2001; Mens et al. 2006]	bottom-up		
DiscoTect [Yan et al. 2004]			hybrid
— [Huang et al. 2006]			
— [Pashov and Riebisch 2004]			hybrid

Table II. SAR Processes Overview

5. SAR PROCESSES

SAR follows either a bottom-up, a top-down or an hybrid opportunistic process.

5.1 Bottom-Up Processes

Bottom-up processes start with low-level knowledge to recover architecture. From source code models, they progressively raise the abstraction level until a high-level understanding of the application is reached (see Figure 3) [Brooks 1983; Storey et al. 1999].

Also called architecture *recovery* processes, bottom-up processes are closely re-

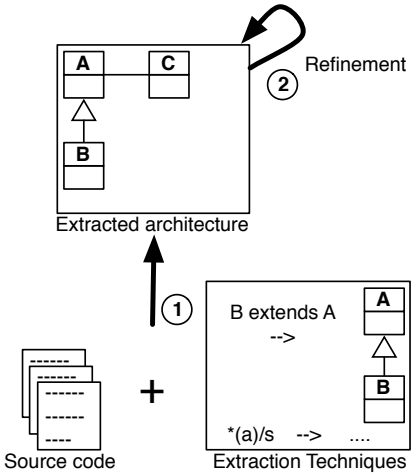

Fig. 3. A bottom-up process: from the source code views are extracted (1) and refined (2).

lated to the well-known *extract-abstract-present* cycle described by Tilley *et al.* [Tilley et al. 1996]. Source code analyses populate a repository, which is queried to yield abstract system representations, which are then presented in a suitable interactive form to reverse engineers.

Examples. The Dali tool by Kazman *et al.* [Kazman and Carriere 1999; Kazman et al. 2001] supports a typical example of a bottom-up process: (1) Heterogeneous low-level knowledge is extracted from the software implementation, fusioned and stored in a relational database. (2) Using the Rigi visualization too [Tilley 1994; Müller et al. 1995], a reverse engineer visualizes and manually abstracts this information. (3) A reverse engineer specifies patterns using SQL queries and Perl expressions. The former selects source model entities and the latter treats this set to abstract them. Based on Dali, Guo *et al.* proposed ARM [Guo et al. 1999]which focuses on design patterns conformance.

In Intensive, Mens *et al.* apply logic intension to group related source-code entities in views [Wuyts 2001; Mens et al. 2006]. Using logic queries helps them cope with code changes. Reverse engineers incrementally define views and relations by means of intensions specified as Smalltalk or logic queries. Intensive classifies the views and displays consistencies and inconsistencies with the code and between architectural views. Intensive visualizes its results with CodeCrawler [Lanza and Ducasse 2003].

Lungu *et al.* built both a method and a tool called Softwarenaut [Lungu et al. 2006]to interactively explore packages. They enhance the exploration process in the package architectural structure by guiding the reverse engineer towards the relevant packages. They characterize packages based on their relation with the other ones and on their internal structure. A set of packages are highlighted and associated to exploration operations that indicate to the reverse engineer the actions to perform to get a better understanding of the software architecture.

Other bottom-up approaches include ArchView [Pinzger 2005], Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]and ARES [Eixelsberger et al. 1998], ARMIN [Kazman et al. 2003], Gupro [Ebert et al. 2002]. We classify the works around PBS/SBS in this category, but since they consider conceptual architectures to steer the process it could be classified as a hybrid process [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999].

5.2 Top-Down Processes

Top-down processes start with high-level knowledge such as requirements or architectural styles and aim to discover architecture by formulating conceptual hypotheses and by matching them to the source code [Carmichael et al. 1995; Murphy et al. 1995; Storey et al. 1999] (see Figure 4). The term architecture *discovery* often describes this process.

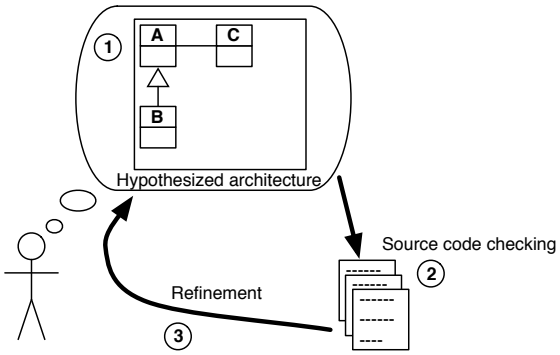

Fig. 4. A top-down process: (1) an hypothesized architecture is defined, (2) the architecture is checked against the source code, (3) the architecture is refined.

Examples. The Reflexion Model of Murphy *et al.* is a typical example of a top-down process [Murphy et al. 1995; Murphy 1996]. First, the reverse engineer defines his high-level hypothesized conceptual view of the application. Second, he specifies how this view maps to the source code concrete view. Finally, RMTool confronts both conceptual and concrete views to compute a reflexion model that highlights *convergences*, *divergences* and *absences* (see Figure 5). The reverse engineer iteratively computes and interprets reflexion models until satisfied. In a reflexion model, a convergence locates an element that is present in both views, a divergence an element that is only in the concrete view, and an absence an element that is only in the conceptual view. The reflexion model offers a better support to express the conceptual architecture and the results of the process than the approach developed in SoFi [Carmichael et al. 1995]. The reflexion model influenced other works [Tran and Holt 1999; Richner and Ducasse 2002; Hassan and Holt 2004; Christl et al. 2005; Knodel et al. 2006].

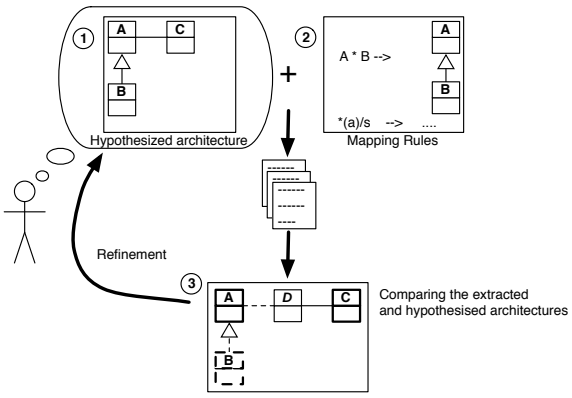

Fig. 5. The Reflexion Model, a top-down process: (1) an hypothesized architecture is defined, (2) rules map source entities to architectural elements, (3) RMTool compares the extracted and hypothesized architectures and the process iterates.

5.3 Hybrid Processes

Hybrid processes combine bottom-up with top-down processes [Storey et al. 1999]. On one hand, low-level knowledge is *abstracted* using various techniques. On the other hand, high-level knowledge is *refined* and confronted against the previously extracted views (see Figure 6). This kind of process is frequently used to stop architectural erosion by reconciling the conceptual and concrete architectures. Hybrid approaches often use hypothesis recognizers [Pashov and Riebisch 2004]. Recognizers provide bottom-up reverse engineering strategies to support top-down exploration of architectural hypothesis.

Examples. Sartipi implements a pattern-based SAR approach in Alborz [Sartipi 2003]. The architecture reconstruction has two phases. During the first bottom-up phase, Alborz parses the source code, presents it as a graph, then divides that graph in cohesive regions using data mining techniques. The resulting model is at a higher abstraction level than the code. During the second top-down phase, the reverse engineer iteratively specifies his hypothesized views of the architecture in terms of patterns. These patterns are approximately mapped with graph regions from the previous phase using graph matching and clustering techniques. Finally, the reverse engineer decides to proceed or not to a new iteration based on the partially reconstructed architecture and evaluation information that Alborz provides.

Christl *et al.* present an evolution of the Reflexion Model [Christl et al. 2005]. They enhance it with automated clustering to facilitate the mapping phase. As in the Reflexion Model, the reverse engineer defines his hypothesized view of the architecture in a top-down process. However, instead of manually mapping hypothetic entities with concrete ones, the new method introduces clustering analysis to partially automate this step. The clustering algorithm groups concrete entities that are not mapped yet with concrete entities already mapped to hypothesized entities.

To assess the creation of product lines, Stoermer *et al.* introduce the MAP

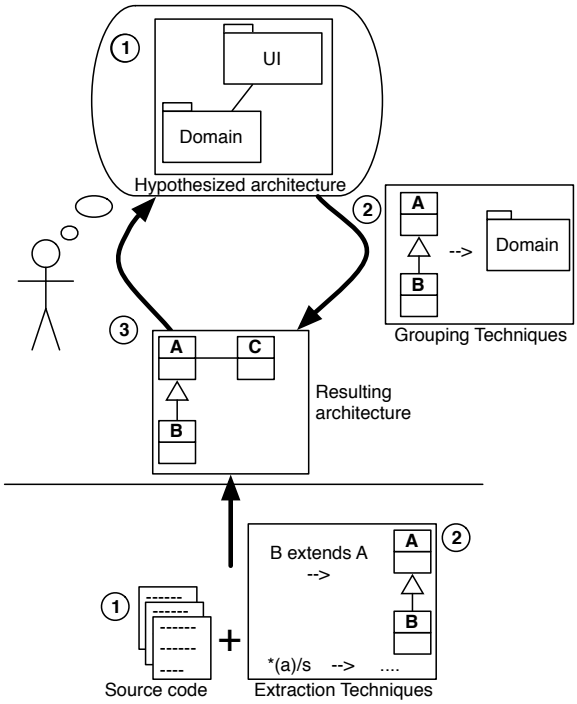

Fig. 6. An hybrid process.

method [Stoermer and O'Brien 2001]. [Stoermer and O'Brien 2001]combines (1) a bottom-up process to recover the concrete architectures of existing products; (2) a top-down process to map architectural styles onto recovered architectural views; (3) an approach to analyze commonalities and variabilities among recovered architectures. They stress the ability of architectural styles to act as the structural glue of the components, and to highlight architecture strengths and weaknesses.

Other hybrid processes include Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]and Nimeta [Riva 2004], ManSART [Harris et al. 1995; A.S.Yeh et al. 1997], ART [Fiutem et al. 1999], X-Ray [Mendonça and Kramer 2001], ARM [Guo et al. 1999]and DiscoTect [Yan et al. 2004]. In ManSART, a top-down recognition engine maps a style-compliant conceptual view with a system overview defined in a bottom-up fashion using a visualization tool [Harris et al. 1995; A.S.Yeh et al. 1997].

5.4 Discussion

As with any classification, the borders are fuzzy. For example, if the refinement step of a bottom-up approach is complex, we could categorize this approach as hybrid. We believe that this is not a real problem since the distinction still introduces important structure and flow to categorize the works. From Table II we can see that the three processes are represented in equal proportions.

6. SAR INPUTS

Most often, SAR works from source code representations, but it also considers other kinds of information, such as dynamic information extracted from a system execution, or historical data held by version control system repositories. A few approaches take other architectural elements as inputs, such as styles or viewpoints. There is no clear trend because SAR approaches are fed with heterogeneous information of diverse abstraction levels. We present first the non-architectural inputs, then the architectural inputs.

6.1 Non-Architectural Inputs

Source Code Constructs. The source code is an omnipresent trustworthy source of information that most approaches consider. Some of the approaches directly query the source code using regular expressions like in RMTool [Murphy et al. 1995; Murphy 1996]or [Pinzger et al. 2002; Pinzger and Gall 2002]. However, most of them do not work from the source code text but represent it using metamodels. These metamodels cope with the paradigm of the analyzed software. For instance, the language independent metamodel FAMIX is used to reverse engineer object-oriented applications [Demeyer et al. 2001]; its concepts include classes, methods, calls or accesses. FAMIX is used in ArchView [Pinzger 2005], Softwarenaut [Lungu et al. 2006]and Nimeta [Riva 2004]. Other metamodels such as the Dagstuhl Middle Metamodel [Lethbridge et al. 2004] or GXL [Holt et al. 2006] have been proposed with the same intent of abstracting the source code.

Symbolic Textual Information. Some approaches consider the symbolic information available in the comments [Pinzger et al. 2002; Pinzger and Gall 2002]or in the method names [Kuhn et al. 2005].

Dynamic Information. Static information is often insufficient for SAR since it only provides a limited insight into the run-time nature of the analyzed software; to understand behavioral system properties, dynamic information is more relevant [Lange and Nakamura 1995]. Some SAR approaches use dynamic information alone [Walker et al. 1998; Yan et al. 2004] while others mix static and dynamic knowledge [Riva and Rodriguez 2002; Vasconcelos and Werner 2004; Li et al. 2005; Huang et al. 2006; Pinzger 2005]. A lot of approaches using dynamic information extract design views rather than architecture [Systä 1999; Richner and Ducasse 1999; Systä 2000; Hamou-Lhadj and Lethbridge 2004; Hamou-Lhadj et al. 2005]. Huang *et al.* consider runtime events such as method calls, CPU utilization or network bandwidth consumption because it may inform reverse engineers about system security properties or system performance aspects. DiscoTect uses dynamic information too [Yan et al. 2004]. Qing *et al.* use run-time process information to derive architectural views [Li et al. 2005]. Some works focus on dynamic software information visualization [Jerding and Rugaber 1997; Systä 2000; Ducasse et al. 2004]. To get a more precise analysis of these works, we refer the reader to the survey of Hamou-Lhadj *et al.* [Hamou-Lhadj and Lethbridge 2004]. Dynamic information is also used to identify features [Eisenbarth et al. 2003; Salah and Mancoridis 2004; Greevy and Ducasse 2005], design patterns [Wendehals 2003; Heuzeroth et al. 2003], or collaborations and roles [Richner and Ducasse 2002; Wu et al. 2004].

Physical Organization. The physical organization of applications in terms of files and folders often reveals architectural information. ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]and Softwarenaut [Lungu et al. 2006]work from the structural organization of physical elements such as files, folders, or packages. Some approaches map packages or classes to components and use the hierarchical nature of the physical organization as architectural input [Langelier et al. 2005; Wu et al. 2004; Pinzger et al. 2005].

Human Organization. According to Conway [Conway 1968]: *"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations"*. It is then important to consider the influence of the human organization on the extracted architectures or views. Inspired by Conway's thesis, Bowman *et al.* use the developer organization to form an ownership architecture that helps reconstruct the software architecture [Bowman and Holt 1998].

Historical Information. Historical information is rarely used in SAR. Wuyts worked on the co-evolution between code and design [Wuyts 2001]. ArchView is a recent approach that exploits source control system data and bug reports to analyze the evolution of recovered architectural views [Pinzger 2005]. Mens *et al.* analyse the evolution of extracted software views with Intensive [Wuyts 2001; Mens et al. 2006]. To assist a reverse engineer in understanding dependency gaps in a reflexion model [Murphy et al. 1995; Murphy 1996], Hassan *et al.* annotate entity dependencies with sticky notes. These sticky notes record dependency evolution and rationale with information extracted from version control systems [Hassan and Holt 2004].

Human Expertise. Although one cannot entirely trust human knowledge, it is very helpful when it is available. At high abstraction levels, SAR is iterative and requires human knowledge to guide it and to validate results. To specify a conceptual architecture [Murphy et al. 1995; Hassan and Holt 2004; Medvidovic and Jakobac 2006], reverse engineers have to study system requirements, read available documentation, interview stakeholders, recover design rationale, investigate hypotheses and analyze the business domain. Human expertise is also required when specifying viewpoints, selecting architectural styles (Section 6.2), or investigating orthogonal artifacts (Section 4.2). While SAR processes involve strategy and knowledge of the domain and the application itself, only a few approaches take human expertise explicitly into account. Ivkovic *et al.* [Ivkovic and Godfrey 2003] propose to systematically update a knowledge base that would become an helpful collection of domain-specific architectural artifacts.

6.2 Architectural Inputs

Architectural styles and viewpoints are the paramount of software architecture, therefore we analyzed whether SAR appraoches consider them as input to steer the extraction process.

Styles. Architectural styles such as pipes and filters, layered system, data flow are popular because like design patterns, they represent recurrent architectural situations [Klein 1999; Buschmann et al. 1996]. They are valuable, expressive, and accepted abstractions for SAR and more generally for software understanding. Ex-

amples of architectural styles are pipe and filter, blackboard and layers.

Recognizing them is however a challenge because they span several architectural elements and can be implemented in various ways [Pinzger et al. 2002; Pinzger and Gall 2002]. The question that turns up is whether SAR helps reverse engineers specify and extract architectural styles.

Examples. In Focus, Ding *et al.* use architectural styles to infer a conceptual architecture that will be mapped to a concrete architecture extracted from the source code [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006].

Closely related to this work, Medvidovic *et al.* introduce an approach to stop architectural erosion. In a top-down process, requirements serve as high-level knowledge to discover the conceptual architecture [Medvidovic et al. 2003]. In a bottom-up process, system implementation serves as low level knowledge to recover the concrete architecture. Both the conceptual and the concrete architectures are incrementally built. The reverse engineer reconciles the two architectures, based on architectural styles. Their approach considers architectural styles as key design idioms since they capture a large number of design decisions, their rationale, effective compositions of architectural elements, and system qualities that will likely result from using the style.

DiscoTect reconstructs style-compliant architectures [Yan et al. 2004]. Using a state machine, DiscoTect incrementally recognizes interleaved patterns in filtered execution traces of the application. The state machine represents an architectural style; by refining it, the reverse engineer defines which hypothesized architectural style the tool should look for [Svetinovic and Godfrey 2001].

ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]and MAP [Stoermer and O'Brien 2001]are other approaches SAR taking into account styles.

Viewpoints. The system architecture acts as a mental model shared among stakeholders [Holt 2001]. Since the stakeholders' interests are diverse, viewpoints are important aspects that SAR may consider [IEEE 2000; Smolander et al. 2001]. Viewpoint catalogues were built to address this issue: the $4 + 1$ viewpoints of Kruchten [Kruchten 1995]; the four viewpoints of Soni *et al.* [Hofmeister et al. 2000; Soni et al. 1995], the build-time viewpoint introduced by Tu *et al.* [Tu and Godfrey 2001] or the implicit viewpoints inherent to the UML standard. Most SAR approaches reconstruct architectural views according only to a single or a few preselected viewpoints. Smolander *et al.* highlight that viewpoints cannot be standardized but should be selected or defined according to the environment and the situation [Smolander et al. 2001]. O'Brien *et al.* present the View-Set Scenario pattern that helps determine which architectural views sufficiently describe the system and cover stakeholder needs [O'Brien et al. 2002].

Examples. The Symphony approach of van Deursen *et al.* aims at reconstructing software architecture using appropriate viewpoints [van Deursen et al. 2004]. Viewpoints are selected from a catalogue or defined if they don't exist. Moreover, they evolve throughout the process. These viewpoints constrain SAR to provide architectural views matching the stakeholders' expectations, ideally allowing immediate use of these views. The authors show how to define viewpoints step by step. In addition, they applied their approach on four case studies whose the stakeholders'

goals are differents each others. They provide architectural views to reverse engineers following the viewpoints these reverse engineers typically use during design phases. Riva proposed a view-based SAR approach called Nimeta based on Symphony [Riva 2004] .

Favre outlines a generic SAR metamodel-driven approach called Cacophony [Favre 2004]. Like Symphony, Cacophony recognizes the need to identify the viewpoints that are relevant to the stakeholders' concerns and that SAR must considered. Contrary to Symphony, Cacophony states that metamodels are keys for representing viewpoints. Viewpoints are for views what metamodels are for models. Viewpoints specify the language that views have to conform to according to the stakeholders' expectations, *i.e.* the concepts they can use. Consequently, Favre states that viewpoints can be defined as metamodels are and that viewpoints can guide the construction of views as metamodels guide the construction of models.

The QADSAR approach both reconstructs the architecture of a system and drives quality attribute analyses on it [Stoermer et al. 2003; Stoermer et al. 2006]. To identify the relevant architectural viewpoints, reverse engineers formulate their interests in reconstructing the architecture by means of concrete quality attribute scenarios

ARES [Eixelsberger et al. 1998]and SARTool [Feijs et al. 1998; Krikhaar 1999]also take viewpoints into account.

6.3 Mixing Inputs

Most approaches work from a limited source of information, even if multiple inputs are necessary to generate rich and different architectural views. Kazman *et al.* [Kazman and Carriere 1998] advocate the fusion of multiple source of inputs to produce richer architectural views: for example, they produce interprocess communication and file access views. Lange *et al.* [Lange and Nakamura 1995] mix dynamic and static view to support design pattern extraction.

ArchVis [Hatch 2004]works from source code, dynamic information such as network log or messages sends and file structures .

Knodel *et al.* [Knodel et al. 2005] discuss the combination of different information sources such as documents, source code and historical data. However it is not clear whether the approach was used in practice. Multiple inputs must be organized and Ivkovitch proposes a systematic way to organize application domain knowledge into a unified structure [Ivkovic and Godfrey 2003].

7. SAR TECHNIQUES

There is a difference between the formalisms used to represent, query and exchange the data representing applications [Riva and Rodriguez 2002]. A couple of exchange formats exist from simple textual tuples in RSF [Wong 1998] or in TA [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999], to XML in GXL [Holt et al. 2006] and in [Ebert et al. 2002; Riva 2004], or CDIF in FAMIX [Ducasse and Wuyts 2002]. The format may limit the merging or manipulation of models [Ducasse and Tichelaar 2003]. An important property of an exchange format is that it can be easily generated and used with simple tools [Ducasse and Demeyer 1999].

Obviously, SAR techniques are often correlated whth the data they operate on. For example, Mens *et al.* express logic queries on facts [Wuyts 2001; Mens et al.

Dali [Kazman and Carriere 1999; Kazman et al. 2001]	src							exp		
ARMIN [Kazman et al. 2003]	src							exp		
MAP [Stoermer and O'Brien 2001]	src							exp	style	
QADSAR [Stoermer et al. 2003; Stoermer et al. 2006]	src							exp		viewp
ARM [Guo et al. 1999]	src							exp		
Gupro [Ebert et al. 2002]	src							exp		
RMTool [Murphy et al. 1995; Murphy 1996]	src							exp		
SAVE [Miodonski et al. 2004; Naab 2005]	src							exp		
PuLSE/SAVE [Knodel et al. 2006; Pinzger et al. 2004]	src							exp		
W4 [Hassan and Holt 2004]	src						hist	exp		
PBS/SBS [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999]	src				phys			exp		
— [Tran and Holt 1999]	src							exp		
— [Bowman and Holt 1998]	src					org	hist	exp		
ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]	src				phys			exp	style	
ART [Fiutem et al. 1999]	src							exp	style	
X-Ray [Mendonça and Kramer 2001]	src							exp		
ARES [Eixelsberger et al. 1998]	src							exp		
SARTool [Feijs et al. 1998; Krikhaar 1999]	src							exp		viewp
Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]	src	text						exp		
ArchView [Pinzger 2005]	src			dyn			hist	exp		
Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]	src							exp	style	
Bunch [Mancoridis and Mitchell 1998; Mitchell and Mancoridis 2006]	src							exp		
ArchVis [Hatch 2004]	src	text		dyn	phys				style	viewp
— [Lundberg and Löwe 2003]	src							exp		
Bauhaus [Koschke 2000; Eisenbarth et al. 2003; Christl et al. 2005]	src			dyn				exp		
Alborz [Sartipi 2003]	src			dyn				exp		
URCA [Bojic and Velasevic 2000]	src			dyn				exp		
Symphony/Nimeta [van Deursen et al. 2004]/[Riva 2004]				dyn				exp		viewp
Cacophony [Favre 2004]								exp		viewp
Softwarenaut/Hapax [Lungu et al. 2006]/[Lungu et al. 2005; Kuhn et al. 2006]	src	text			phys			exp		
Softwarenaut [Lungu et al. 2006]					phys			exp		
Intensive [Wuyts 2001; Mens et al. 2006]	src							exp		
DiscoTect [Yan et al. 2004]	src			dyn				exp	style	
— [Huang et al. 2006]	src			dyn					style	
— [Pashov and Riebisch 2004]	src			dyn				exp	style	

src source code · text textual information · dyn dynamic information
phys physical organization · org human organization · hist historical information
exp human expertise · style styles · viewp viewpoints

Table III. SAR Input Overview

2006]while Ebert *et al.* perform queries on graphs [Ebert et al. 2002]. Instead we classify techniques in to three automation levels:

Quasi-manual. the reverse engineer manually identifies architectural elements using a tool to assist him to understand his findings;

Semi-automatic.. the reverse engineer manually instructs the tool how to automatically discover refinements or recover abstractions;

Suasi-automatic.. the tool has the control and the reverse engineer steers the iterative recovery process.

Of course, the boundaries between the classifications are not clear-cut. Moreover, to understand the results of their analysis, reverse engineers often use visualization tools, however we consider that a comparison of the visualization tools to be be beyond the scope of this article.

7.1 Quasi-Manual Techniques

SAR is a reverse engineering activity which faces scalability issues in manipulating knowledge. In response to this problem, researchers have proposed slightly assisted SAR approaches; we consider two categories: construction-based and exploration-based approaches.

Construction-based Techniques. These techniques reconstruct the software architecture by manually abstracting low-level knowledge, thanks to interactive and expressive visualization tools — Rigi [Tilley 1994; Müller et al. 1995], CodeCrawler [Lanza and Ducasse 2003], Shrimp/Creole [Storey and Müller 1995; Storey et al. 1997], Verso [Langelier et al. 2005], 3D [Marcus et al. 2003] or GraphViz [Gansner and North 2000].

Exploration-based Techniques. These techniques give reverse engineers an architectural view of the system by guiding them through the highest-level artifacts of the implementation, like in Softwarenaut [Lungu et al. 2006]. The architectural view is then closely related to the developer's view.

Instead of providing guidance, the SAB browser [Erben and Löhr 2005] allows reverse engineers to assign architectural layers to classes and then to navigate the resulting architectural views.

ArchView[1] [Feijs and Jong 1998] visualizes simple architectural elements and their relationships in 3D.

7.2 Semi-Automatic Techniques

Semi-automatic techniques automate repetitive aspects of SAR. The reverse engineer steers the iterative refinement or abstraction, leading to the identification of architectural elements.

Abstraction-based Techniques. These techniques aim to map low-level concepts with high-level concepts. Reverse engineers specify reusable abstraction rules and to execute them automatically. Explored approaches are:

[1]Different of ArchView Pinzger's approach [Pinzger 2005], though homonymous.

Graph queries. Gupro queries graphs using a specialized declarative expression language called GReQL [Ebert et al. 2002]. Rigi is based on graph transformations written in Tcl [Tilley 1994; Müller et al. 1995].

Relational queries. Often, relational algebra engines abstract data out of entity-relation databases. Dali uses SQL queries to define grouping rules [Kazman and Carriere 1999; Kazman et al. 2001] and ARMIN too [Kazman et al. 2003]. Relational algebra defines a repeatable set of transformations such as abstraction or decomposition to create a particular architectural view. In PBS/SBS, Holt *et al.* propose the Grok relational expression calculator to reason about software facts [Holt 1998]. Krikhaar presents a SAR approach based on an extension of relational algebra [Feijs et al. 1998; Krikhaar 1999]. The ArchView abstraction algorithm is based on relational algebra too and is moreover based on metrics [Pinzger 2005].

Logic queries. Logic queries are powerful because of the underlying unification mechanism which allows the writing of dense multi valued queries. Kramer and Prechelt [Kramer and Prechelt 1996], Wuyts [Wuyts 1998], Gueneheuc [Guéhéneuc et al. 2004] use Prolog queries to identify design patterns. Mens and Wuyts use Prolog as a meta programming language to extract intensional source-code views and relations in Intensive [Wuyts 2001; Mens et al. 2006]. Richner also chose a logic query based approach to reconstruct architectural views from static and dynamic facts [Richner and Ducasse 1999].

Programs. Some approaches build analyses as plain object-oriented programs. For example, the analyses made in the Moose environment are performed as object-oriented programs that manipulate models representing the various inputs [Ducasse et al. 2005].

Lexical and structural queries. Some approaches are directly based on the lexical and structural information in the source code. Pinzger *et al.* state that some hot-spots clearly localize patterns in the source code and consider them as the starting point of SAR [Pinzger et al. 2002; Pinzger and Gall 2002]. To drive a pattern-supported architecture recovery, they introduce a pattern specification language and the Revealer tool. RMTool identifies architectural elements and relations using lexical queries [Murphy et al. 1995; Murphy 1996]. The Searchable Bookshelf is a typical example of supporting navigation via queries [Sim et al. 1999].

ArchVis [Hatch 2004]supports multiple inputs (files, programs, ACME informations), works from static and dynamic information (program execution but also log files and network traffic), and provides different views to specific stakeholders (component, developer, manager views).

Investigation-based Techniques. These techniques map high-level concepts with low-level concepts. The high-level concepts considered cover a wide area from architectural descriptions and styles to design patterns and features. Explored approaches are:

Recognizers. ManSART [Harris et al. 1995; A.S.Yeh et al. 1997], ART [Fiutem et al. 1999], X-Ray [Mendonça and Kramer 2001]and ARM [Guo et al. 1999]are based on recognizers for architectural styles or patterns written in a query language. The tools then report the source code elements matching the recognized structures. More precisely, pattern definitions in ARM are progressively refined

and finally transformed in SQL queries exploitable in Dali [Kazman and Carriere 1999; Kazman et al. 2001]. The design patterns extraction approaches fit in this category (see Section 9).

Graph pattern matching. In ARM [Guo et al. 1999], pattern definitions can also be transformed into graphs pattern to match with a graph-based source code representation; this is similar to what does [Sartipi 2003].

State engine. In DiscoTect state machines are defined to check architectural styles conformance [Yan et al. 2004]. A state engine tracks the system execution at run-time and outputs architectural events when the execution satisfies the state machine description.

Maps. SAR approaches based on the Reflexion Model [Murphy et al. 1995; Murphy 1996]use rules to map hypothesized high-level entities with source code entities. Since these Perl-like rules take plain source code as input, we could have classified the reflexion model in the *lexical and structural queries* group mentioned previously, but the intention here is really mapping. SoFi [Carmichael et al. 1995] use naming conventions of files and folders to automatically group entities.

7.3 Quasi-Automatic Techniques

Purely automated software architecture extraction techniques do not exist. Reverse engineers must still steer the most automated approaches. Approaches in this area often combine concept, dominance and cluster analysis techniques.

Concepts. Formal concept analysis is a branch of lattice theory used to identify design patterns [Arévalo et al. 2004], features [Eisenbarth et al. 2003; Greevy and Ducasse 2005] , or modules [Siff and Reps 1997]. Tilley *et al.* [Tilley et al. 2003] present a survey of work using formal concept analysis.

Clustering Algorithms. Clustering algorithms identify groups of objects whose members are similar in some way. They have been used to produce software views of applications. To identify subsystems, Anquetil and Lethbridge cluster files using naming conventions [Anquetil and Lethbridge 1999b]. Some approaches automatically partition software products into cohesive clusters that are loosely interconnected [Wiggerts 1997; Mancoridis and Mitchell 1998; Anquetil and Lethbridge 1999a; Trifu 2001; Mitchell and Mancoridis 2006]. Clustering algorithms are also used to extract features from object interactions [Salah and Mancoridis 2004]. Koschke emphasizes the need to refine existing clustering techniques, first by combining them, and second by integrating the reverse engineer as a conformance supervisor of the reconstruction process [Koschke 2000; Eisenbarth et al. 2003; Christl et al. 2005].

Dominance. Dominance analysis identifies the nodes of a directed graph that have only one ancestor. In software maintenance it identifies the related parts in an application. In the context of software architecture extraction, adhering to Koschke's thesis, Trifu unifies cluster and dominance analysis techniques to recover architectural components in object-oriented legacy systems [Trifu 2001]. Similarly, Lundberg *et al.* outline a unified approach centered around dominance analysis [Lundberg and Löwe 2003]. On one hand, they demonstrate how dominance analysis identifies passive components. On the other hand, they state that dominance analysis is not sufficient to recover the complete architecture: it requires other techniques

Tools	Quasi-manual	Semi-automatic Abstr.	Semi-automatic Invest.	Quasi-auto.
Dali [Kazman and Carriere 1999; Kazman et al. 2001]	cns	rel		
ARMIN [Kazman et al. 2003]		gra		
MAP [Stoermer and O'Brien 2001]				
QADSAR [Stoermer et al. 2003; Stoermer et al. 2006]				
ARM [Guo et al. 1999]	cns	gra, rel		
Gupro [Ebert et al. 2002]		gra		
RMTool [Murphy et al. 1995; Murphy 1996]			map	
SAVE [Miodonski et al. 2004; Naab 2005]			map	
PuLSE/SAVE [Knodel et al. 2006; Pinzger et al. 2004]			map	
W4 [Hassan and Holt 2004]			map	
PBS/SBS [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999]		rel	map	
— [Tran and Holt 1999]		rel	map	
— [Bowman and Holt 1998]				
ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]	cns		rec	
ART [Fiutem et al. 1999]			rec	
X-Ray [Mendonca and Kramer 2001]			rec	auto
ARES [Eixelsberger et al. 1998]				
SARTool [Feijs et al. 1998; Krikhaar 1999]		rel		
Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]		lex		
ArchView [Pinzger 2005]		rel		
Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]	cns			
Bunch [Mancoridis and Mitchell 1998; Mitchell and Mancoridis 2006]				auto
ArchVis [Hatch 2004]	cns	rel, prg		auto
— [Lundberg and Löwe 2003]				auto
Bauhaus [Koschke 2000; Eisenbarth et al. 2003; Christl et al. 2005]			rec, map	auto
Alborz [Sartipi 2003]			gpm	auto
URCA [Bojic and Velasevic 2000]				auto
Symphony/Nimeta [van Deursen et al. 2004]/[Riva 2004]				
Cacophony [Favre 2004]				
Softwarenaut/Hapax [Lungu et al. 2006]/[Lungu et al. 2005; Kuhn et al. 2006]		gra		
Softwarenaut [Lungu et al. 2006]	exp			
Intensive [Wuyts 2001; Mens et al. 2006]		log		
DiscoTect [Yan et al. 2004]			sta	
— [Huang et al. 2006]				auto
— [Pashov and Riebisch 2004]	cns, exp			auto

cns construction · exp exploration · gra graph queries · rel relational queries
log logic queries · prg programs · lex lexical queries · rec recognizers
gpm graph pattern matching · sta state engine · map maps · auto quasi-automatic

Table IV. SAR Techniques Overview

such as concept analysis to take component interactions into account.

8. SAR OUTPUTS

While most approaches focus on identifying and presenting software architectures, some provide valuable additional information such as conformance information.

Indeed, goals and outputs are clearly related. In this section we highlight some points to further classify the approaches.

8.1 Visual Software Views

A lot of approaches offer architectural views or use visualizations as output. As we mentioned earlier, several tools such as Rigi [Tilley 1994; Müller et al. 1995], Shrimp/Creole [Storey and Müller 1995; Storey et al. 1997], GraphViz [Gansner and North 2000] or CodeCrawler [Lanza and Ducasse 2003] are used to visualize graph representations of software views [Finnigan et al. 1997; Kazman and Carriere 1999; Koschke 2000; Pinzger and Gall 2002; Sartipi 2003; Riva 2004]. Some authors propose open toolkits to build architectural extractors [Telea et al. 2002; Lowe and Panas 2005; Meyer et al. 2006].

Classifying the outputs of the various visualization approaches is difficult and outside of the scope of this article, but we can still distinguish some groups: some visualization approaches present source code entities and group them as boxes using the visualization tools mentioned above [Finnigan et al. 1997; Kazman and Carriere 1999; Koschke 2000; Pinzger and Gall 2002; Sartipi 2003; Riva 2004]. Some offer enhanced views that provide a bit more architectural information [Pinzger 2005; Mens et al. 2006; Lungu et al. 2006]. In this context some approaches improve their visualizations with 2D/3D [Feijs and Jong 1998; Telea et al. 2002; Marcus et al. 2003; Lowe and Panas 2005; Langelier et al. 2005]. Finally some approaches define dedicated tool support to represent architectural elements and layers; for example, the Software Architecture Browser is a graphical editor dedicated to navigation in layers [Erben and Löhr 2005]. Pacione proposed both the architecture-oriented visualization tool Vanessa, and a taxonomy in which he surveyed related tools [Pacione 2005].

As shown in Section 6, some SAR approaches focus on the behavior of software. Hamou-Lhadj *et al.* surveyed some of the tools supporting traces visualization [Hamou-Lhadj and Lethbridge 2004]. To offer multiple views of an application, it is interesting to combine static and dynamic analysis [Lange and Nakamura 1995; Richner and Ducasse 1999; Systä et al. 2001; Hatch 2004]. For example, Shimba [Systä et al. 2001] combines static and dynamic information to produce high-level views of Java systems; it displays static information with Rigi [Tilley 1994; Müller et al. 1995], and dynamic information as state diagrams. Both views are thus displayed separately, but the reverse engineers can constrain the abstraction of each view from the other.

8.2 Architecture

Since SAR approaches focus on providing better understanding of the applications, they tend to present reconstructed architectural views to stakeholders. As the code evolves, some approaches focus on the co-evolution of the reconstructed architecture: Intensive [Wuyts 2001; Mens et al. 2006]synchronizes the architecture with its implementation and highlight the differences due to evolution.

Iterative approaches based on the reflection model [Murphy et al. 1995; Richner and Ducasse 2002; Knodel et al. 2006; Christl et al. 2005] make explicit the absences, convergences and divergences between the conceptual architecture and the architecture that results from mapping source code elements to architectural

elements.

Architecture Description Languages (ADLs) have been proposed both to formally define architectures and to support architecture-centric development activities [Medvidovic and Taylor 2000]. In the context of SAR, X-Ray [Mendonça and Kramer 2001]uses Darwin [Magee et al. 1995] to express reconstructed architectural views. Darwin was also extended by Eixelsberger *et al.* [Eixelsberger et al. 1998]. Acme [Garlan and Métayer 1997] has ADL-like features and is used in DiscoTect [Yan et al. 2004]. Huang *et al.* specify architectures with the ABC ADL [Huang et al. 2006]. They reconstruct architectural views and express them according to the ADL language in use to be coherent with an architecture-centric software development. In addition, ADL feautres allow reverse engineers to give information in an ADL compliant format to improve the SAR process such as the layout of an architectural view.

8.3 Conformance

Some approaches focus on determining the conformance of an application to a given architecture. We distinguish two kinds of architecture conformance: horizontal conformance between similar abstractions and vertical conformance between different abstraction levels.

Horizontal conformance is checked between two reconstructed views, or between a conceptual and a concrete architecture, or between a product line reference architecture and the architecture of a given product. For example, SAR approaches for product line migration identify commonalities and variabilities among products, like in MAP [Stoermer and O'Brien 2001]. Sometimes, SAR requires to define a conceptual architecture and to compare it with the reconstructed one [Guo et al. 1999; Tran and Holt 1999]. Sometimes, an architecture must conform to architectural rules or styles; this was discussed in Nimeta [Riva 2004], the SARTool tool [Feijs et al. 1998; Krikhaar 1999], Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]and DiscoTect [Yan et al. 2004].

Vertical conformance assesses whether the recontructed architecture conforms to the implementation. Both Reflexion Model-based [Murphy et al. 1995; Murphy 1996]and co-evolution-oriented [Wuyts 2001; Mens et al. 2006]approaches revolve around vertical conformance.

8.4 Analysis

Some approaches perform extra analysis on the extracted architecture to qualify it or to refine it further. Reverse engineers use modularity quality metrics either to iteratively assess current results and steer the process, or to get cues about reuse and possible system improvement [Koschke 2000; Sartipi 2003].

A few SAR approaches propose other analyses: ArchView [Pinzger 2005]provides structural and evolutionary properties of a software application. Eixelsberger *et al.* in ARES [Eixelsberger et al. 1998], and Stoermer in QADSAR [Stoermer et al. 2003; Stoermer et al. 2006]reconstruct software architectures to highlight properties like safety, concurrency, portability or other high-level statistics [Huang et al. 2006].

Finally, some approaches highlight architectural patterns or orthogonal artifacts: ARM [Guo et al. 1999], Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]or Alborz [Sartipi 2003].

Dali [Kazman and Carriere 1999; Kazman et al. 2001]	vis	desc			ana
ARMIN [Kazman et al. 2003]	vis				ana
MAP [Stoermer and O'Brien 2001]	vis				
QADSAR [Stoermer et al. 2003; Stoermer et al. 2006]	vis				ana
ARM [Guo et al. 1999]	vis				
Gupro [Ebert et al. 2002]	vis				
RMTool [Murphy et al. 1995; Murphy 1996]	vis			vert	
SAVE [Miodonski et al. 2004; Naab 2005]	vis			vert	
PuLSE/SAVE [Knodel et al. 2006; Pinzger et al. 2004]	vis			vert	ana
W4 [Hassan and Holt 2004]	vis			vert	ana
PBS/SBS [Finnigan et al. 1997; Holt 1998; Bowman et al. 1999; Sim et al. 1999]	vis				
— [Tran and Holt 1999]	vis			vert	
— [Bowman and Holt 1998]	vis		horz		
ManSART [Harris et al. 1995; A.S.Yeh et al. 1997]	vis				
ART [Fiutem et al. 1999]	vis				
X-Ray [Mendonça and Kramer 2001]	vis	desc			
ARES [Eixelsberger et al. 1998]	vis	desc			ana
SARTool [Feijs et al. 1998; Krikhaar 1999]	vis		horz	vert	ana
Revealer [Pinzger et al. 2002; Pinzger and Gall 2002]	vis				
ArchView [Pinzger 2005]	vis				
Focus [Ding and Medvidovic 2001; Medvidovic and Jakobac 2006]	vis				
Bunch [Mancoridis and Mitchell 1998; Mitchell and Mancoridis 2006]	vis				
ArchVis [Hatch 2004]	vis	desc			
— [Lundberg and Löwe 2003]	vis				
Bauhaus [Koschke 2000; Eisenbarth et al. 2003; Christl et al. 2005]	vis			vert	
Alborz [Sartipi 2003]	vis				ana
URCA [Bojic and Velasevic 2000]	vis				
Symphony/Nimeta [van Deursen et al. 2004]/[Riva 2004]	vis		horz	vert	ana
Cacophony [Favre 2004]					
Softwarenaut/Hapax [Lungu et al. 2006]/[Lungu et al. 2005; Kuhn et al. 2006]	vis				
Softwarenaut [Lungu et al. 2006]	vis				
Intensive [Wuyts 2001; Mens et al. 2006]	vis				
DiscoTect [Yan et al. 2004]	vis	desc	horz	vert	
— [Huang et al. 2006]		desc	horz		ana
— [Pashov and Riebisch 2004]	vis				

vis architecture visualization · desc architecture description
horz horizontal conformance · vert vertical conformance · ana analysis

Table V. SAR Outputs Overview

9. ORTHOGONAL OR RELATED ABSTRACTIONS

A large body of work focuses on extracting design or on reverse engineering applications. It is difficult to clearly separate these approaches from SAR since architecture has many forms and design information is crucial to characterize architecture. These approaches focus on identifying artifacts that either support the architecture, such as design patterns [Beck and Johnson 1994], or crosscut the architecture, such as features and roles. These related artifacts convey important information about the architecture; this is why we included them in this survey in a section of their

own.

9.1 Design Patterns

Design patterns are important abstractions in programming and designing applications because they create a common vocabulary [Gamma et al. 1995]. A design pattern highlights a recurring problem that arises in a specific design context, and discusses the possible solutions.

Beck and Johnson use a set of patterns to derive an architecture [Beck and Johnson 1994]. Deducing an architecture from patterns records the design decisions that were made, and hints at their underlying motivations. Buschmann *et al.* mention that patterns are useful mental building-blocks which compose and document the architecture [Buschmann et al. 1996]. Patterns span several levels of abstraction from architecture through design to language, and they are interwoven with each other. Architectural patterns or styles express high level fundamental organizations of systems; design patterns describe medium level structures of communicating components; language patterns or idioms present low level aspects of programming languages. For all these reasons researchers have been drawing considerable attention onto design pattern identification [Lange and Nakamura 1995].

Shull *et al.* proposed a method to manually identify workable domain-specific design patterns and create customized catalogs of the identified patterns [Shull et al. 1996]. Brown automatically identifies design patterns using the reflective capabilities of Smalltalk [Brown 1996]. Keller *et al.* promote pattern analysis as well as human expertise to extract design pattern [Keller et al. 1999]. Bergenti *et al.* provide critiques about the design patterns identified [Bergenti and Poggi 2000]. Philippow *et al.* promote a design pattern based approach to reconstruct the reference architecture of a product line [Philippow et al. 2004].

Several approaches use Prolog to represent and query source code [Kramer and Prechelt 1996; Wuyts 1998]. Design patterns are then represented as logic queries. Lange *et al.* represent both static and dynamic information as logic facts to generate interactive design views and help understanding frameworks [Lange and Nakamura 1995]. Design patterns like Chain of Responsibility are based on specific interactions among the pattern participants, so researchers investigated dynamic analysis to extract design patterns [Heuzeroth et al. 2003; Wendehals 2003].

One of the main problem in pattern identification is the search space. To reduce the search space, Wendehals [Wendehals 2003] proposes to combine static and dynamic analyzes, the first one reducing the search space of the second one. Therefore, static and dynamic analyses narrow the set of candidates: the static analysis searches for sets of candidates that respect the static structure of the design pattern, while the dynamic analysis monitors candidates and checks whether the observed interactions satisfy the behavioral rules of the design patterns [Heuzeroth et al. 2003]. Antoniol *et al.* propose a multi stage reduction strategy [Antoniol et al. 1998]: software metrics and structural properties computed on design patterns become constraints that design pattern candidates must satisfy. Guéhéneuc *et al.* reduce the search space using metrics to define design pattern fingerprints of the design pattern participants [Guéhéneuc et al. 2004].

A design pattern has several design variants and can be implemented in different ways. Niere *et al.* overcome these two problems by using fuzzy logic [Niere et al.

2001]. Wendehals rates each instance candidate with a fuzzy value to support inexact mismatch [Wendehals 2003].

9.2 Concerns

Concerns are the criteria a stakeholder is interested in to modularize a software application into manageable and comprehensible parts [Robillard and Murphy 2002; Coelho and Murphy 2006]. Features and aspects are kinds of concerns. In spite of an explicit boundary, features often relate to functional requirements while aspects often relate to non functional requirements. A key to software understanding is to locate source code entities according to the concerns they fulfill. However, concerns are not explicitly linked to source code entities. In fact concerns are often scattered and tangled throughout the system artifacts: they often crosscut its physical decomposition. Recovering crosscutting concerns is thus an active research area, but nowadays, researchers essentially focus their attention on mining concerns and rarely link their works with SAR. However, aggregating source code entities around the concerns they find could be a useful means of abstraction for SAR.

Features. According to Eisenbarth *et al.* a feature is *"an observable behavior of a system that can be triggered by a user"* and a computational unit is *"an executable part of a system"* [Eisenbarth et al. 2003]. A feature in the minds of reverse engineers is implemented through several computational units in the source code. To understand the how a set of features is implemented, one must identify the computational units that contribute to these features and optionally the way they interact together. The feature location activity often follows an as-needed approach that is oriented towards focused maintenance tasks such as feature enhancement or bug fixing. Features are high level knowledge while computational units are low level knowledge. More generally, features acts as a bridge between the requirements and the architecture of the system [Pashov and Riebisch 2004]. Therefore, a feature view improves software understanding by mapping functional requirements in the minds of reverse engineers with architectural elements and indirectly with source code entities. A feature view could help SAR by hiding implementation details around features.

The Software Reconnaissance method is a promising approach in the feature location field [Wilde and Scully 1995]. To identify the computational units related to a given feature, this method compares computational units invoked by different scenarios which trigger or not this feature. In a similar way, Wong *et al.* proposed an approach that analyses execution slices of different scenarios [Wong et al. 1999]. Chen *et al.* outlined a human-guided approach [Chen and Rajlich 2000]. Assisted by a tool, a reverse engineer explores a statically derived dependency graph and iteratively decides whether each considered computational unit is relevant to the feature or not. A case study applies both Software Reconnaissance and the approach of Chen *et al.* to locate features [Wilde et al. 2003]; Marcus *et al.* complement the conclusions of this case study [Marcus et al. 2005].

Following feature location is feature extraction. Eisenbarth *et al.* [Eisenbarth et al. 2003] combine static and dynamic analyses to derive the map linking features with computational units. To collect the computational units contributing to a feature, they trigger this feature by executing scenarios. Using concept analysis,

they obtain a map of relationships between features and computational units; this map is subject to human interpretation. Finally, they refine the map by deriving more relevant computational unit sets using static analysis such as dominance analysis. Salah *et al.* derive a feature map from source code and more precisely from object interactions. This feature map is linked to a navigable hierarchy of low-level interaction views; their method progressively raises the abstraction level from object interactions to feature interactions [Salah and Mancoridis 2004]. Greevy *et al.* characterize features and computational units according to two complementary perspectives, both a feature perspective and a computational unit perspective [Greevy and Ducasse 2005]. If you to collect the computation units that interact together to execute a feature, you have to select the feature perspective and vice versa if you want to collect the features a computational unit participate to, you have to select a computational unit persepctive. The approach allows for instance a reverse engineer to know how some computational units participate at the realization of a given feature.

Pashov *et al.* promote the use of feature modeling to improve SAR [Pashov and Riebisch 2004]. Their feature-oriented approach iteratively reconstructs the architecture by establishing and verifying functional and architectural hypotheses. These hypotheses link features, architectural elements and source code entitites in cross-referencing tables which are verified iteratively. Cleary *et al.* specified a viewpoint suitable to express feature-based architecture and thus to improve feature reuse investigation [Cleary et al. 2005].

Finally, several researchers also focus on the evolution of features [Fischer et al. 2003; Greevy et al. 2005]. In the future, SAR approaches could use these works to obtain better architectural insights.

Aspects. Aspect mining is currently receiving a lot of attention. In object-oriented applications, the implementation of a concern is typically scattered over many locations, duplicated in several places, and tangled with the implementation of other concerns; this makes systems harder to understand, maintain and evolve. Aspect mining is a reverse engineering process which aims to find and isolate these crosscutting concerns. Aspect mining is mainly explored to better understand a piece of software or to refactor it in an aspect-oriented one. Since there are already several surveys of the subject [Ceccato et al. 2005; Kellens and Mens 2005; Nora et al. 2006], we do not cover it here. It is however worth to mention that there is no approaches linking mining aspects with architecture extraction.

9.3 Collaborations and Roles

To understand an object-oriented application, one must understand the collaborations and the roles that objects play [Reenskaug 1996; Wu et al. 2004]. Collaborations are goal-oriented interactions between participants, while roles describe the participants' responsibilities in a collaboration. Researchers focused on recovering collaborations and roles of objects and indirectly of classes [Richner and Ducasse 2002]. De Hondt proposes collaboration contracts as a basis to control the evolution of collaborations [Hondt 1998].

Some approaches deduce class collaborations by visualizing object interactions [Hamou-Lhadj and Lethbridge 2004]. Richner *et al.* propose an approach to recover

collaborations and roles that does however not rely on visualization techniques [Richner and Ducasse 2002]; they work from both dynamic information and human expertise. Pattern matching tools extract collaboration patterns from execution traces that record method invocation information. To only focus on relevant class collaborations and roles, reverse engineers then steer the process through querying and visualization facilities. Wu *et al.* applied a closely related approach to procedural legacy systems [Wu et al. 2004].

10. DISCUSSION

Here are some general points that appeared to us at the light of this survey. A lot of approaches visualize software entities but few work from diverse information or even take advantage of having different kinds of information. Several times this paper stresses the need to provide a large variety of views at different levels of abstraction. We advocate that viewpoints should be defined consistently. SAR must integrate in an environment that provides reverse engineers with views at different levels of abstraction and means to navigate horizontally and vertically. To fulfill this requirement, we state that a mechanism is required to express consistently viewpoints whatever the level of abstraction of the views they respectively describe. In this perspective, the metamodel-based SAR outlined by Favre is promising [Favre 2004].

Lots of works focused on extracting design information such as design patterns but stopped building on this knowledge up to the architectural level. Similarly few works bring together features and architectural information.

Because it is complex to extract architectural components from source code, those are often simply mapped to packages or files. Even if this practice is understandable, we think it limits and overloads the *component* term.

We see that few works really take into account architectural styles. This may be the result of having different communities working on architectural description languages and maintenance.

SAR is complex and time consuming. The iterative aspects of SAR imposed themselves as a key point to ensure a successful reconstruction. Now to reach a high-level of maturity in leading such an activity, we advocate that SAR has to support co-evolution and conformance mechanisms. Indeed both horizontal and vertical conformance help bringing all the recovered views face to face. This confrontation allows reverse engineers to refine views iteratively, to identify commonalities and variabilities among views (especially if they represent product lines architecture), to lead impact analysis or still to update views when the system evolves.

Since successful systems are doomed to continually evolve and grow, SAR approaches should support co-evolution mechanisms to keep all recovered views synchronized with the source code. The logic-based approach of Intensive proved to be efficient in checking horizontal and vertical conformance and in allowing co-evolution [Wuyts 2001; Mens et al. 2006].

11. CONCLUSION

It is hard to classify research approaches in a complex field where the subject matter is as fuzzy as software architecture. Still this survey has provided an organization of

the significant fundamental contributions made within software architecture reconstruction. To structure the paper, we followed the general process of SAR: what are the stakeholders' goals; how does the general reconstruction proceed; what are the available sources of information; based on this, which techniques can we apply, and finally what kind of knowledge does the process provide. We believe that software architecture is still an important topic since it is crucial for the understanding of large industrial applications and their evolutions.

Acknowledgments. We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872). We would like to thanks Tudor Girba and Orla Greevy for the early feedback on the paper.

REFERENCES

Aldrich, J., Chambers, C., and Notkin, D. 2002. Architectural reasoning in archjava. In *Proceedings ECOOP 2002*. LNCS, vol. 2374. Springer Verlag, Malaga, Spain, 334–367.

Anquetil, N. and Lethbridge, T. 1999a. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*. 235–255.

Anquetil, N. and Lethbridge, T. C. 1999b. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice 11*, 201–21.

Antoniol, G., Fiutem, R., and Cristoforetti, L. 1998. Design pattern recovery in object-oriented software. In *6th International Workshop on Program Comprehension (Ischia, Italy)*. 153–160.

Arévalo, G., Buchli, F., and Nierstrasz, O. 2004. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*. IEEE Computer Society Press, 122–131.

A.S.Yeh, Harris, D., and Chase, M. 1997. Manipulating recovered software architecture views. In *Proceedings of International Conference Software Engineering (ICSE'97)*.

Beck, K. and Johnson, R. 1994. Patterns generate architectures. In *Proceedings ECOOP '94*, M. Tokoro and R. Pareschi, Eds. LNCS, vol. 821. Springer-Verlag, Bologna, Italy, 139–149.

Bergenti, F. and Poggi, A. 2000. Improving UML designs using automatic design pattern detection. In *12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 336–343.

Beyer, D. and Lewerentz, C. 2003. CrocoPat: A tool for efficient pattern recognition in large object-oriented programs. Tech. Rep. I-04/2003, Institute of Computer Science, Brandenburgische Technische Universität Cottbus. Jan.

Bojic, D. and Velasevic, D. 2000. A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, Los Alamitos, CA, USA, 23–33.

Bowman, I. and Holt, R. 1998. Software architecture recovery using conway's law. In *Proceedings of the Centre for Advanced Studies Conference, CASCON'98*. 123–133.

Bowman, I. T., Holt, R. C., and Brewster, N. V. 1999. Linux as a case study: its extracted software architecture. In *International Conference on Software Engineering (ICSE'99)*. IEEE CS, 555–563.

Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 543–554.

Brown, K. 1996. Master's Thesis, North Carolina State University.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stad, M. 1996. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley Press.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA.

ca, N. C. M. and Kramer, J. 1996. Requirements for an effective architecture recovery framework. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*. ACM Press, 101–105.

Carmichael, I., Tzerpos, V., and Holt, R. C. 1995. Design maintenance: Unexpected architectural interactions. In *International Conference on Software Maintenance (ICSM)*. IEEE CS, 134–140.

Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwe, T. 2005. A qualitative comparison of three aspect mining techniques. In *13th International Workshop on Program Comprehension (IWPC)*. IEEE CS, 13–22.

Chen, K. and Rajlich, V. 2000. Case study of feature location using dependence graph. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 241–249.

Christl, A., Koschke, R., and Storey, M.-A. 2005. Equipping the reflexion method with automated clustering. In *Working Conference on Reverse Engineering (WCRE)*. 89–98.

Cleary, B., Le Gear, A., Exton, C., and Buckley, J. 2005. Specifying a software reconnaissance architectural viewpoint. In *FIX ME*.

Coelho, W. and Murphy, G. C. 2006. Presenting crosscutting structure with active models. In *AOSD'06: Proceedings of the 5th international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 158–168.

Conway, M. E. 1968. How do committees invent ? *Datamation 14,* 4 (Apr.), 28–31.

Demeyer, S., Tichelaar, S., and Ducasse, S. 2001. FAMIX 2.1 — The FAMOOS Information Exchange Model. Tech. rep., University of Bern.

Ding, L. and Medvidovic, N. 2001. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In *Working Conference on Software Architecture (WICSA)*. 191–.

Ducasse, S. and Demeyer, S., Eds. 1999. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern.

Ducasse, S., Gîrba, T., Lanza, M., and Demeyer, S. 2005. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*. RCOST / Software Technology Series. Franco Angeli, Milano, 55–71.

Ducasse, S., Lanza, M., and Bertuli, R. 2004. High-level polymetric views of condensed run-time information. In *Proceedings of CSMR 2004 (Conference on Software Maintenance and Reengineering)*. 309–318.

Ducasse, S. and Tichelaar, S. 2003. Dimensions of reengineering environment infrastructures. *International Journal on Software Maintenance: Research and Practice 15,* 5 (Oct.), 345–373.

Ducasse, S. and Wuyts, R. 2002. Supporting objects as an anthropomorphic view at computation or why Smalltalk for teaching objects? In *Proceedings of the Ecoop'02 International Educator Symposium*.

Dueñas, J., Lopes de Oliveira, W., and de la Puente, J. 1998. Architecture recovery for software evolution. In *Conference on Software Maintenance and Reengineering (CSMR)*. 113–120.

Dunsmore, A., Roper, M., and Wood, M. 2000. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*. ACM Press, 467–476.

Ebert, J., Kullbach, B., Riediger, V., and Winter, A. 2002. GUPRO – generic understanding of programs, an overview. Fachberichte Informatik 7–2002, Universität Koblenz-Landau.

Eisenbarth, T., Koschke, R., and Simon, D. 2003. Locating Features in Source Code. *IEEE Computer 29,* 3 (Mar.), 210–224.

Eixelsberger, W., Ogris, M., Gall, H., and Bellay, B. 1998. Software architecture recovery of a program family. In *International Conference on Software Engineering (ICSE)*. 508–511.

Erben and Löhr. 2005. Sab - the software architecture browser. In *International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE CS.

Favre, J.-M. 2003. Meta-model and model co-evolution within the 3d software space. In *Proceedings of ELISA 2003*.

Favre, J.-M. 2004. CacOphoNy: Metamodel-driven software architecture reconstruction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*. IEEE Computer Society Press, Los Alamitos CA, 204–213.

Feijs, L. and Jong, R. D. 1998. 3d visualization of software architectures. *Communication of the ACM 41,* 12, 73–78.

Feijs, L., Krikhaar, R., and van Ommering, R. 1998. A relational approach to support software architecture analysis. *Software – Practice and Experience 28,* 4 (Apr.), 371–400.

Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis, K., Mueller, H., Mylopoulos, J., Perelgut, S., Stanley, M., and Wong., K. 1997. The software bookshelf. *IBM Systems Journal 36,* 4 (Nov.), 564–593.

Fischer, M., Pinzger, M., and Gall, H. 2003. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*. IEEE Computer Society Press, Los Alamitos CA, 90–99.

Fiutem, R., Antoniol, G., Tonella, P., and Merlo, E. 1999. Art: an architectural reverse engineering environment. *Journal of Software Maintenance: Research and Practice 11,* 5, 339–364.

Gallagher, K., Hatch, A., and Munro, M. 2005. A framework for software architecture visualization assessment. In *VISSOFT*. IEEE CS, 76–81.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass.

Gansner and North. 2000. An open graph visualization system and its applications to software engineering. *Software Practice Experience. 30,* 11, 1203–1233.

Garlan, D. 2000. Software architecture: a roadmap. In *ICSE - Future of SE Track*. 91–101.

Garlan, D., Allen, R., and Ockerbloom, J. 1995. Architectural mismatch: Why reuse is so hard. *IEEE Software 12,* 6 (Nov.), 17–26.

Garlan, D. and Métayer, D. L., Eds. 1997. *Coordination Languages and Models*. LNCS, vol. 1282. Springer-Verlag, Berlin, Germany.

Greevy, O. and Ducasse, S. 2005. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 314–323.

Greevy, O., Ducasse, S., and Gîrba, T. 2005. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*. IEEE Computer Society Press, 347–356. TR-LISTIC-0520.

Guéhéneuc, Y.-G., Mens, K., and Wuyts, R. 2006. A comparative framework for design recovery tools. In *Conference on Software Maintenance and Reengineering (CSMR 2006)*. IEEE Computer Society Press, Los Alamitos CA.

Guéhéneuc, Y.-G., Sahraoui, H., and Zaidi, F. 2004. Fingerprinting design patterns. In *Working Conference on Reverse Engineering (WCRE'04)*. IEEE Computer Society Press, Los Alamitos CA, 172–181.

Guo, Y., Atlee, and Kazman. 1999. A software architecture reconstruction method. In *Working Conference on Software Architecture (WICSA)*. 15–34.

Hamou-Lhadj, A., Braun, E., Amyot, D., and Lethbridge, T. 2005. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press.

Hamou-Lhadj, A. and Lethbridge, T. 2004. A survey of trace exploration tools and techniques. In *Proceedings ofi 14th Annual IBM Centers for Advanced Studies Conferences (CASON)*. IBM Press, 42–55.

Harris, D. R., Reubenstein, H. B., and Yeh, A. S. 1995. Reverse engineering to the architectural level. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*. Association for Computing Machinery, Inc., Seattle, Washington USA.

Hassan and Holt. 2004. Using development history sticky notes to understand software architecture. In *International Workshop on Programming Conference*. 183–193.

Hatch, A. 2004. Software architecture visualisation. Ph.D. thesis, Research Institute in Software Engineering, University of Durham.

Heuzeroth, D., Holl, T., Högström, G., and Löwe, W. 2003. Automatic design pattern detection. In *International Workshop on Program Comprehension*. 94–104.

Hofmeister, C., Nord, R. L., and Soni, D. 2000. *Applied Software Architecture*. Addison Wesley.

Holt, Schürr, Sim, and Winter. 2006. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming 60,* 2 (Apr.), 149–170.

Holt, R. 1998. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of WCRE '98*. IEEE Computer Society, 210–219. ISBN: 0-8186-89-67-6.

Holt, R. 2001. Sofware architecture as a shared mental model. In *ASERC Workshop on Software Architecture*. University of Alberta.

Hondt, K. D. 1998. Ph.D. thesis. Ph.D. thesis, Vrije Universiteit Brussel,Departement of Computer Science, Brussels — Belgium.

Huang, G., Mei, H., and Yang, F.-Q. 2006. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering 13,* 2, 257–281.

IEEE. 2000. Ieee recommended practice for architectural description for software-intensive systems. Tech. rep., The Architecture Working Group of the Software Engineering Committee. Oct.

Ivkovic and Godfrey. 2003. Enhancing domain-specific software architecture recovery. In *International Workshop on Program Comprehension (IWPC)*. 266–276.

Jerding, D. and Rugaber, S. 1997. Using visualization for architectural localization and extraction. In *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, I. Baxter, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society Press, 56–65.

Kazman, R. and Bass, L. 2005. Categorizing business goals for software architectures. Cmu/sei-2005-tr-021, Carnegie Mellon University, Software Engineering Institute. Dec.

Kazman, R., Bass, L. J., Webb, M., and Abowd, G. D. 1994. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering (ICSE)*. 81–90.

Kazman, R. and Carriere, S. J. 1998. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*. Victoria, B.C.

Kazman, R. and Carriere, S. J. 1999. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*.

Kazman, R., Klein, M. H., Barbacci, M., Longstaff, T. A., Lipson, H. F., and Carrière, S. J. 1998. The architecture tradeoff analysis method. In *ICECCS*. 68–78.

Kazman, R., O'Brien, L., and Verhoef, C. 2001. Architecture reconstruction guidelines. CMU/SEI-2001-TR-026, Carnegie Mellon University, Software Engineering Institute. Aug.

Kazman, R., O'Brien, L., and Verhoef, C. 2003. Architecture reconstruction guidelines, third edition. CMU/SEI-2002-TR-034, Carnegie Mellon University, Software Engineering Institute. Nov.

Kellens, A. and Mens, K. 2005. A survey of aspect mining tools and techniques. Tech. rep., UCL, Belgium. June.

Keller, R. K., Schauer, R., Robitaille, S., and Pagé, P. 1999. Pattern-Based Reverse Engineering of Design Components. In *Proceedings of ICSE '99 (21st International Conference on Software Engineering)*. IEEE Computer Society Press / ACM Press, 226–235.

Klein, G. 1999. *Sources of Power — How People Make Decisions*. Addison Wesley.

Knodel, J., John, I., Ganesan, D., Pinzger, M., Usero, F., Arciniegas, J. L., and Riva, C. 2005. Asset recovery and their incorporation into product lines. In *Proceedings of Working Conference on Reverse Engineering (WCRE 05)*. IEEE Computer Society, Washington, DC, USA, 120–129.

Knodel, J., Muthig, D., Naab, M., and Lindvall, M. 2006. Static evaluation of software architectures. In *CSMR'06*. IEEE Computer Society, Los Alamitos, CA, USA, 279–294.

Koschke, R. 2000. Atomic architectural component recovery for program understanding and evolution. Ph.D. thesis, Universität Stuttgart.

Kramer, C. and Prechelt, L. 1996. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of WCRE '96 (3rd Working Conference on Reverse Engineering)*. IEEE Computer Society Press, 208–216.

Krikhaar, R. 1999. Software architecture reconstruction. Ph.D. thesis, University of Amsterdam.

Kruchten, P. B. 1995. The 4+1 view model of architecture. *IEEE Software 12,* 6 (Nov.), 42–50.

Kuhn, A., Ducasse, S., and Gîrba, T. 2005. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference on Reverse Engineering (WCRE 2005)*. IEEE Computer Society Press, Los Alamitos CA, 113–122. TR-LISTIC-0519.

Kuhn, A., Ducasse, S., and Gîrba, T. 2006. Semantic clustering: Identifying topics in source code. *Information and Software Technology*. To appear.

Lange, D. B. and Nakamura, Y. 1995. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*. ACM Press, 342–357.

Langelier, G., Sahraoui, H. A., and Poulin, P. 2005. Visualization-based analysis of quality for large-scale software systems. In *ASE*. 214–223.

Lanza, M. and Ducasse, S. 2003. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering 29,* 9 (Sept.), 782–795.

Lehman, M. and Belady, L. 1985. *Program Evolution: Processes of Software Change*. London Academic Press, London.

Lethbridge, T., Tichelaar, S., and Plödereder, E. 2004. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*. Vol. 94. 7–18.

Li, Q., Chu, H., Hu, S., Chen, P., and Yun, Z. 2005. Architecture recovery and abstraction from the perspective of processes. In *Working Conference on Reverse Engineering (WCRE)*. 57–66.

Lowe, W. and Panas, T. 2005. Rapid construction of software comprehension tools. In *International Journal of Software Engineering and Knowledge Engineering*.

Lundberg, J. and Löwe, W. 2003. Architecture recovery by semi-automatic component identification. *Electronic Notes in Theoretical Computer Science 82,* 5.

Lungu, M., Kuhn, A., Gîrba, T., and Lanza, M. 2005. Interactive exploration of semantic clusters. In *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*. 95–100.

Lungu, M., Lanza, M., and Gîrba, T. 2006. Package patterns for visual architecture recovery. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*. IEEE Computer Society Press, Los Alamitos CA, 183–192.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. 1995. Specifying distributed software architectures. In *Proceedings ESEC '95*. LNCS, vol. 989. Springer-Verlag, 137–153.

Mancoridis, S. and Mitchell, B. S. 1998. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press.

Marcus, A., Feng, L., and Maletic, J. I. 2003. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*. IEEE, 27–ff.

Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A. 2005. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*.

Medvidovic and Taylor. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering 26,* 1, 70–93.

Medvidovic, N., Egyed, A., and Gruenbacher, P. 2003. Stemming architectural erosion by architectural discovery and recovery. In *Proceedings of the 2nd Second International Workshop from Software Requirements to Architectures (STRAW).*

Medvidovic, N. and Jakobac, V. 2006. Using software evolution to focus architectural recovery. *Automated Software Engineering 13,* 2, 225–256.

Mendonça, N. C. and Kramer, J. 2001. An approach for recovering distributed system architectures. *Automated Software Engineering 8,* 3-4, 311–354.

Mens, K., Kellens, A., Pluquet, F., and Wuyts, R. 2006. Co-evolving code and design with intensional views – a case study. *Journal of Computer Languages, Systems and Structures 32,* 2, 140–156.

Meyer, M., Gîrba, T., and Lungu, M. 2006. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006).* To appear.

Miodonski, P., Forster, T., Knodel, J., Lindvall, M., and Muthig, D. 2004. Evaluation of software architectures with eclipse. Tech. rep., Fraunhofer IESE. nov.

Mitchell, B. S. and Mancoridis, S. 2006. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering 32,* 3, 193–208.

Müller, H. A., Wong, K., and Tilley, S. R. 1995. Understanding software systems using reverse engineering technology. In *Object-Oriented Technology for Database and Software Systems*, V. Alagar and R. Missaoui, Eds. World Scientific, 240–252.

Murphy, G., Notkin, D., and Sullivan, K. 1995. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.* ACM Press, 18–28.

Murphy, G. C. 1996. Lightweight structural summarization as an aid to software evolution. Ph.D. thesis, University of Washington.

Naab, M. 2005. Evaluation of graphical elements and their adequacy for the visualization of software architectures. M.S. thesis, Fraunhofer IESE.

Niere, J., Wadsack, J. P., and Wendehals, L. 2001. Design pattern recovery based on source code analysis with fuzzy logic. tr-ri-01-222, Software Engineering Group, Department of Mathematics and Computer Science, University of Paderborn, Paderborn, Germany.

Nora, B., Said, G., and Fadila, A. 2006. A comparative classification of aspect mining approaches. *Journal of Computer Science 4,* 2, 322–325.

O'Brien, L., Stoermer, C., and Verhoef, C. 2002. Software architecture reconstruction: Practice needs and current approaches. Tech. Rep. CMU/SEI-2002-TR-024, Carnegie Mellon University. Aug.

Pacione, M. 2005. A novel software visualisation model to support object-oriented program comprehension. Ph.D. thesis, Univ. Strathclyde.

Pashov, I. and Riebisch, M. 2004. Using feature modelling for program comprehension and software architecture recovery. In *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04).*

Perry, D. E. and Wolf, A. L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes 17,* 4 (Oct.), 40–52.

Philippow, I., Streitferdt, D., and Riebisch, M. 2004. *Design Pattern Recovery in Architectures for Supporting Product Line Development and Application.* BoD GmbH, 42–57.

Pinzger, M. 2005. Archview – analyzing evolutionary aspects of complex software systems. Ph.D. thesis, Vienna University of Technology.

Pinzger, M., Fischer, M., Gall, H., and Jazayeri, M. 2002. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002).* 170–178.

Pinzger, M. and Gall, H. 2002. Pattern-supported architecture recovery. In *10th International Workshop on Program Comprehension (IWPC'02).* 53–61.

Pinzger, M., Gall, H., and Fischer, M. 2005. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science 127,* 3, 183–196.

Pinzger, M., Gall, H., Girard, J.-F., Knodel, J., Riva, C., Pasman, W., Broerse, C., and Wijnstra, J. G. 2004. Architecture recovery for product families. In *Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5).* LNCS 3014. Springer-Verlag, 332–351.

Reenskaug, T. 1996. *Working with Objects: The OOram Software Engineering Method.* Manning Publications.

Richner, T. and Ducasse, S. 1999. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings ICSM '99 (International Conference on Software Maintenance)*, H. Yang and L. White, Eds. IEEE Computer Society Press, 13–22.

Richner, T. and Ducasse, S. 2002. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM '2002 (International Conference on Software Maintenance).*

Riva, C. 2004. View-based software architecture reconstruction. Ph.D. thesis, Technical University of Vienna.

Riva, C. and Rodriguez, J. V. 2002. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR).* IEEE Computer Society, Washington, DC, USA, 47.

Robillard, M. P. and Murphy, G. C. 2002. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering.* ACM Press, New York, NY, USA, 406–416.

Salah, M. and Mancoridis, S. 2004. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004).*

Sartipi, K. 2003. Software architecture recovery based on pattern matching. Ph.D. thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada.

Shaw, M. and Garlan, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall.

Shull, F., Melo, W. L., and Basili, V. R. 1996. An inductive method for discovering design patterns from object-oriented software systems. Tech. Rep. CS-TR-3597, University of Maryland Computer Science Department.

Siff, M. and Reps, T. 1997. Identifying Modules via Concept Analysis. In *Proceedings of ICSM '97 (International Conference on Software Maintenance).* IEEE Computer Society Press, 170–179.

Sim, S. E., Clarke, C. L., Holt, R. C., and Cox, A. M. 1999. Browsing and searching software architectures. *Proceedings of the International Conference on Software Maintenance (ICSM) 00*, 381.

Smith, J. E. and Nair, R. 2005. *Virtual Machines.* Morgan Kaufmann.

Smolander, Hoikka, Isokallio, Kataikko, Mäkelä, and Kälviäinen. 2001. Required and optional viewpoints – what is included in software architecture? Tech. rep., Univ. Lappeenranta.

Soni, D., Nord, R. L., and Hofmeister, C. 1995. Software architecture in industrial applications. In *Proceedings ICSE '95.* ACM Press, Seattle, 196–207.

Stoermer, C. and O'Brien, L. 2001. Map - Mining architectures for product line evaluations. In *Working Conference on Software Architecture (WICSA).* 35–41.

Stoermer, C., O'Brien, L., and Verhoef, C. 2003. Moving towards quality attribute driven software architecture reconstruction. In *Working Conference on Reverse Engineering (WCRE).* IEEE Computer Society, Los Alamitos, CA, USA, 46–56.

Stoermer, C., Rowe, A., O'Brien, L., and Verhoef, C. 2006. Model-centric software architecture reconstruction. *Software — Practice and Experience 36,* 4, 333–363.

Storey, M.-A. D., Fracchia, F. D., and Müller, H. A. 1999. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems 44*, 171–185.

Storey, M.-A. D. and Müller, H. A. 1995. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*. IEEE Computer Society Press, 275–284.

Storey, M.-A. D., Wong, K., and Müller, H. A. 1997. How do program understanding tools affect how programmers understand programs? In *Proceedings Fourth Working Conference on Reverse Engineering*, I. Baxter, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society, 12–21.

Svetinovic, D. and Godfrey, M. 2001. A lightweight architecture recovery process. In *Proceedings Eight Working Conference on Reverse Engineering (WCRE'01)*.

Systä, Koskimies, and Müller. 2001. Shimba — an environment for reverse engineering Java software systems. *Software — Practice and Experience 1,* 1 (Jan.).

Systä, T. 1999. On the relationships between static and dynamic models in reverse engineering java software. In *Working Conference on Reverse Engineering (WCRE99)*. 304–313.

Systä, T. 2000. Static and dynamic reverse engineering techniques for Java software systems. Ph.D. thesis, University of Tampere.

Telea, Maccari, and Riva. 2002. An open visualization toolkit for reverse architecting. In *Proceedings of International Workshop on Program Comprehension (IWPC)*. IEEE CS, 3–13.

Tilley, S. R. 1994. Domain-retargetable reverse engineering II: Personalised user interfaces. In *Proceedings of The International Conference on Software Maintenance*. IEEE Computer Society.

Tilley, S. R., Smith, D. B., and Paul, S. 1996. Towards a framework for program understanding. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*. IEEE Computer Society, 19.

Tilley, T., Cole, R., Becker, P., and Eklund, P. 2003. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In *Proceedings of ICFCA '03 (1st International Conference on Formal Concept Analysis)*, G. Stumme, Ed. Springer-Verlag.

Tran, J. and Holt, R. 1999. Forward and reverse repair of software architecture. In *Proceedings of CASCON*.

Trifu, A. 2001. Using cluster analysis in the architecture recovery of object-oriented systems. Ph.D. thesis, Univ. Karlsruhe.

Tu, Q. and Godfrey, M. W. 2001. The build-time software architecture view. In *International Conference on Software Maintenance (ICSM 2001)*. 398–407.

van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., and Riva, C. 2004. Symphony: View-driven software architecture reconstruction. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 122–134.

Vasconcelos, A. and Werner, C. 2004. Software architecture recovery based on dynamic analysis. In *Proceedings of the 18th Brazilian Symposium on Software Engineering*.

Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J. 1998. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*. ACM, 271–283.

Wendehals, L. 2003. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA)*.

Wiggerts, T. 1997. Using Clustering Algorithms in Legacy Systems Remodularization. In *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, I. Baxter, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society Press, 33–43.

Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L. 2003. A comparison of methods for locating features in legacy software. *Journal of Systems and Software 65,* 2, 105–114.

Wilde, N. and Huitt, R. 1992. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering SE-18,* 12 (Dec.), 1038–1044.

Wilde, N. and Scully, M. C. 1995. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice 7,* 1, 49–62.

Wong, K. 1998. The rigi user's manual — version 5.4.4. Tech. rep., University of Victoria.

Wong, W. E., Horgan, J. R., Gokhale, S. S., and Trivedi, K. S. 1999. Locating program features using execution slices. *asset 00*, 194.

Woods, S. G., Carrière, S. J., and Kazman, R. 1999. The perils and joys of reconstructing architectures. *SEI Interactive, The Architect 2*.

Wu, L., Sahraoui, H., and Valtchev, P. 2004. Program comprehension with dynamic recovery of code collaboration patterns and roles. In *CASCON'04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 56–67.

Wu, X., Murray, A., Storey, M.-A., and Lintern, R. 2004. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*. IEEE Computer Society Press, Los Alamitos CA, 90–99.

Wuyts, R. 1998. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*. IEEE Computer Society Press, 112–124.

Wuyts, R. 2001. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Ph.D. thesis, Vrije Universiteit Brussel.

Yan, H., Garlan, D., Schmerl, B., Aldrich, J., and Kazman, R. 2004. Discotect: A system for discovering architectures from running systems. In *International Conference on Software Engineering (ICSE)*. 470–479.