Une Ingénierie Centrée Architecture de Lignes de Produits Logiciels à Base de Composants

Sylvain Vauttier

Ecole Doctorale I2S, Spécialité Informatique

Université de Montpellier

Habilitation à Diriger des Recherches

Une Ingénierie Centrée Architecture de Lignes de Produits Logiciels à Base de Composants

Sylvain VAUTTIER

Maitre-Assistant à l'IMT Mines Alès
France
sylvain.vauttier@mines-ales.fr
http://sylvainvauttier.wp.mines-telecom.fr/

26 juin 2018

Mme Mireille Blay-Fornarino	Université de Nice, Professeur	Rapporteur
Mme Isabelle Borne	Université de Bretagne Sud, Professeur	Rapporteur
M Philippe Merle	Inria Lille-Nord Europe, Professeur	Rapporteur
Mme Marie-Pierre Gervais	LIP6, Professeur	Examinateur
Mme Marianne Huchard	Université de Montpellier, Professeur	Examinateur
M Jacky Montmain	IMT Mines Alès, Professeur	Examinateur
M Mourad Oussalah	Université de Nantes, Professeur	Examinateur
M Camille Salinesi	Université Paris-Sorbonne, Professeur	Examinateur

Table des matières

Liste des figures

Remerciements

Je remercie chaleureusement l'ensemble des personnes qui ont contribué, de multiples façons, à ce que cette HDR vienne synthétiser un parcours de recherche déjà long.

Je remercie en premier lieu l'IMT Mines Alès et son laboratoire LGI2P, pour le cadre de travail qu'ils m'ont fournis depuis près de 20 ans. Je voudrais en particulier remercier Pascal Poncelet, ancien sous-directeur du LGI2P, pour la confiance qu'il m'a accordée en me donnant ma première bourse de thèse. Je voudrais également remercier Yannick Vimont, ancien directeur de la recherche et du LGI2P, pour son engagement à défendre une recherche académique de qualité. Merci enfin au directeur actuel du LGI2P, Jacky Montmain, pour son amitié et la préservation bienveillante de notre collectif. Tiens bon, chef !

Je remercie les professeurs Mireille Blay-Fornarino, Isabelle Borne et Philippe Merle d'avoir accepté d'évaluer mon manuscrit.

Je remercie les professeurs Marie-Pierre Gervais, Marianne Huchard, Jacky Montmain, Mourad Oussalah et Camille Salinesi d'avoir accepté de participer à mon jury d'habilitation.

Je remercie l'ensemble de mes co-auteurs, doctorants et collègues chercheurs, avec qui j'ai eu le plaisir de travailler : Christophe, Gabi, Guy, Chouki, Luc, Djamel, David, Frédéric, Nicolas, Yulin, Fady, Matthieu, Guillaume, Rafat, Abderrahman, Alexandre, Martine, Mourad, Marianne et Christelle.

Je remercie Mourad d'avoir donné le virus de la recherche au jeune élève ingénieur que j'étais et pour m'avoir ensuite appris le métier.

Je remercie tout particulièrement Marianne pour ces années de collaboration, de soutien et d'amitié. Il est précieux dans la carrière d'un jeune enseignant-chercheur de croiser le chemin d'un sénior aussi bienveillant.

Je remercie encore plus particulièrement Christelle, ma complice de presque toujours, dans bien des domaines ...

Je remercie bien sûr mes collègues du LGI2P et au delà mes collègues du site EERIE de Nîmes devenu le bâtiment M de Croupillac pour la dose d'humour et de dérision qui nous ont permis de travailler dans une bonne ambiance durant toutes ces années.

Je remercie enfin mon fils Lucas, ma famille, mes amis et mes amours pour tout le reste...

Chapitre 1

Introduction

« *Un poète mort n'écrit plus. D'où l'importance de rester vivant* »

Michel Houellebecq. Rester vivant : méthode (1991)

1.1 Crise du logiciel

La problématique de la crise du logiciel apparait dès 1968, lors de la première conférence sur le génie logiciel organisée par l'OTAN. Dijksta en donnera une définition plus personnelle en 1972 [49] : "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful ! To put it quite bluntly : as long as there were no machines, programming was no problem at all ; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem".

Il est étonnant de constater que la crise du logiciel est une problématique aussi ancienne et qu'elle continue à être un sujet de recherche actif. Mais, comme l'observe Dijkstra, la crise du logiciel est entre autres la conséquence naturelle de l'augmentation de la puissance de calcul de nos machines, qui permet d'envisager d'exécuter des logiciels de plus en plus complexes. Jusqu'à présent, la conjecture de Moore [99], qui prévoit le doublement de la puissance de calcul des machines tous les deux ans à coût constant, par une meilleure intégration des transistors, une augmentation de la fréquence de fonctionnement ou maintenant la parallélisation des architectures, n'a pas été démentie, même si elle atteint actuellement des limites imposées par la physique (cf. figure 1.1).

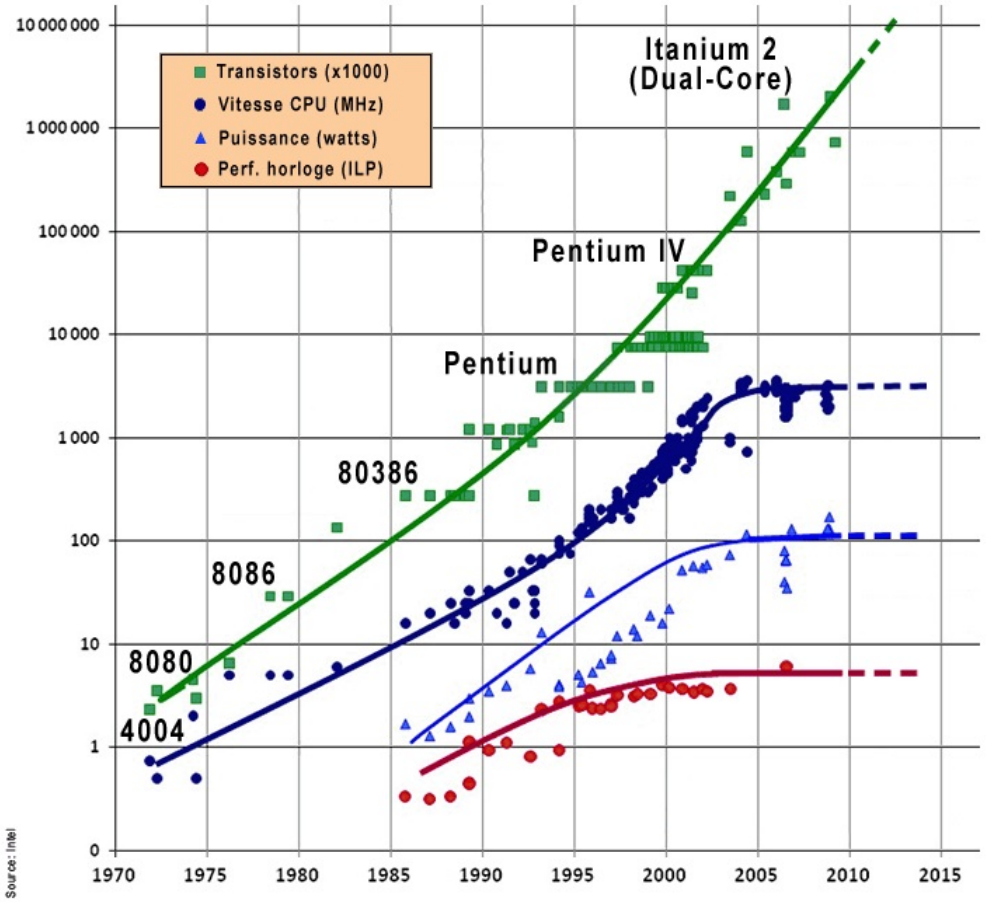


FIGURE 1.1 – La conjecture de Moore [1]

Parallèlement, le développement d'internet a également fortement contribué au développement de logiciels toujours plus riches en contenus et complexes en fonctionne-

ment avec la possibilité d'accéder à des ressources distantes ou de distribuer l'exécution. Plus récemment, cette tendance s'est encore accélérée avec le cloud computing [35], qui facile non seulement l'accès à des services distants (Saas, software as a service) mais également à des ressources logicielles (PaaS, platform as a service) et matérielles (Iaas, infrastructure as a service) pour développer et déployer des applications.

Enfin, l'émergence de l'internet des objets (IoT) [21] va entraîner une généralisation de la présence des logiciels, jusque dans les plus simples objets. L'exploitation des fonctionnalités de ces objets pour créer des environnements intelligents (ville, quartier, campus, habitation) facilitant notre quotidien ou optimisant notre consommation de ressources nécessite le développement de logiciels assurant l'orchestration ou la chorégraphie de fonctionnalités élémentaires pour construire les fonctionnalités complexes répondant aux besoins spécifiques de chacun. A cette échelle de complexité et de distribution, le développement des logiciels se conçoit comme le développement de systèmes de systèmes [84].

Pour résumer ce préambule en une phrase : la crise du logiciel est une crise ancienne mais plus que jamais d'actualité.

1.2 Réutilisation et Ingénierie de lignes de produits logiciels

Dans ce contexte de complexité croissante des logiciels, les travaux de recherche que j'ai contribué à développer visent à résoudre l'équation fondamentale du génie logiciel : produire le plus efficacement possible des logiciels de qualité répondant aux besoins de leurs utilisateurs. La réutilisation, dans son principe général, est la solution la plus courante de cette équation. L'étude de son application disciplinée et systématique, en vue de son industrialisation, a conduit au développement des approches d'ingénierie de lignes de produits logiciels [15]. Ces approches introduisent la distinction entre deux types de processus de développement (cf. figure 1.2) : les processus de développement pour la réutilisation, qui soutiennent l'ingénierie de domaine, et les processus de développement par réutilisation, qui soutiennent l'ingénierie des applications.

L'ingénierie de domaine a pour objectif de produire des artefacts réutilisables couvrant l'ensemble des étapes de développement d'un logiciel, depuis l'analyse du besoin, par l'élaboration de modèles de features (modèles définissant les caractéristiques et fonctionnalités abstraites de systèmes) [73], jusqu'à l'implémentation en passant par les étapes de conception, par le développement de composants logiciels de différentes natures (modèles, codes sources, codes objets, ...) [120]. Il est possible de réaliser l'ingénierie de domaine a priori afin de proposer des bibliothèques standardisées de composants métier génériques pour des domaines applicatifs bien identifiés. Des initiatives ont déjà été lancées, par exemple par l'OMG (Corba Domain Interface) [113] ou par IBM (projet San Francisco, précurseur de Websphere) [25]. Mais elles ont jusqu'à présent échoué, malgré leur intérêt, souligné par les analystes, qui les considèrent comme ayant été trop en avance par rapport à la pratique.

Ainsi, l'ingénierie de domaine se pratique essentiellement a posteriori, dans une dé-

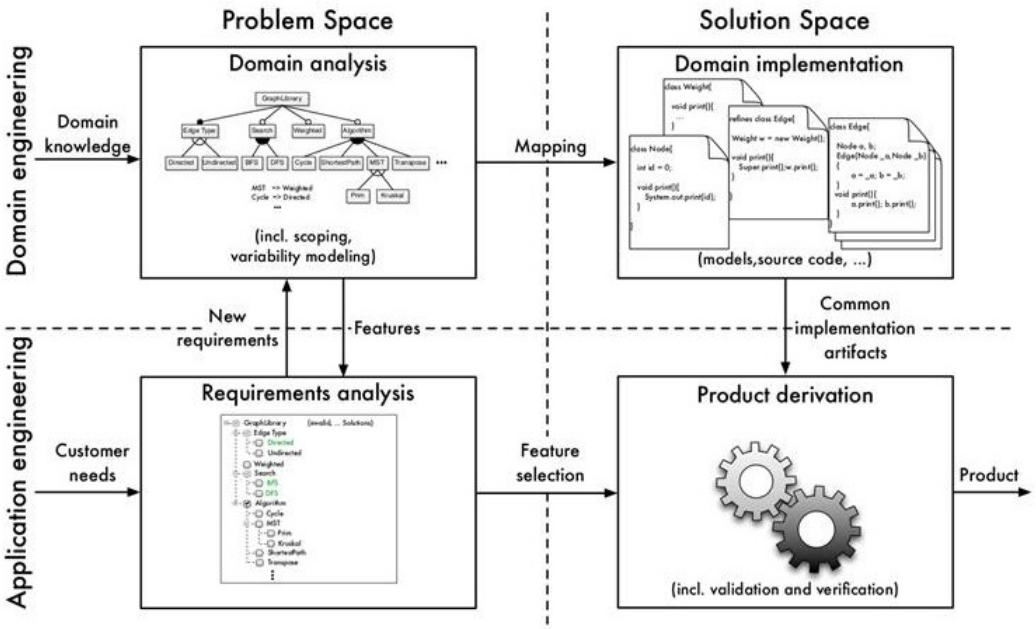


FIGURE 1.2 – Ingénierie de lignes de produits logiciels [15]

marche opportuniste et pragmatique qui cadre bien avec les approches de développement agiles [38] : au lieu d'essayer d'anticiper des besoins potentiels, les équipes de développement constatent l'opportunité de réutiliser, pour satisfaire des besoins similaires, des éléments de projets existants. Conformément aux principes de l'agilité, cette démarche doit être facilitée par une attention constante portée à la qualité des logiciels et en particulier à la qualité de leurs architectures qui, par leur découpage, leur structuration et le découplage de leurs modules, doivent faciliter la réutilisation et l'adaptation de tout ou partie de leurs éléments. A défaut, le clonage indiscipliné de projets conduit à la création de familles de produits implicites (clone and own [57]), engendrant des développements redondants, par absence de visibilité sur les éléments existants, leur révisions et variantes, ou des problèmes de fonctionnement, par absence de gestion de configuration spécifiant la compatibilité des éléments (éventuellement de leurs différentes versions). Il faut alors procéder à la rétro-ingénierie de la ligne de produits correspondante pour restaurer tous les bénéfices de la réutilisation [6].

Conjointement, l'ingénierie des applications consiste à développer des applications par la réutilisation systématique des artéfacts réutilisables produits par l'ingénierie de domaine [39]. Initié classiquement par une étape d'analyse du besoin, l'élaboration des exigences fonctionnelles est réalisée par la sélection d'un ensemble de features, parmi ceux définis dans le modèle de features réutilisé, en respectant les contraintes d'exclusion ou d'implication qu'il impose. Cet ensemble de features permet alors la sélection d'artéfacts de développement réutilisables (modèles, code, ...), stockés dans des bibliothèques et indexés afin de pouvoir être efficacement associés à l'implémentation de features (mappings). Une fois la sélection d'artéfacts établie, l'étape d'implémentation consiste alors à assembler les artéfacts pour produire une application répondant au besoin identifié.

Cette description idéale fait l'impasse sur toutes les difficultés de sa mise en œuvre concrète. Il n'est pas certain, en effet, que l'identification et la sélection d'un ensemble de features traduisant les besoins des utilisateurs en exigences se déroule sans problème.

Il est probable qu'une partie des besoins ne corresponde que partiellement aux features existants et que certains besoins ne soient actuellement pas satisfaits.

Dans le second cas, il faudra procéder au développement de nouveaux artéfacts et documenter de nouveaux features pour étendre la définition du domaine couvert (ce que l'on peut considérer comme de l'ingénierie de domaine agile).

Le premier cas relève d'une problématique plus générale d'appariement entre des éléments définissant des caractéristiques ou des éléments de logiciels à différents niveaux d'abstraction pour correspondre aux différentes phases du cycle de développement. Dans le schéma général du processus d'ingénierie des applications, la problématique de l'appariement intervient principalement lors de la transition entre l'ingénierie des besoins et l'ingénierie des exigences (appariement entre les besoins utilisateurs et les features de la ligne de produits logiciels) puis entre l'ingénierie des exigences et la conception/implémentation du logiciel (appariement entre les features et les artéfacts définissant des réalisations possibles des features).

On peut distinguer un mapping entre éléments, qui est un appariement statique, d'un matching, qui est un appariement dynamique résultant de l'exécution d'une requête définie par un ensemble de critères de sélection. La précision des appariements conditionne l'efficacité de la réutilisation, donc l'efficacité de l'approche par ligne de produits. Il faut par exemple concevoir des mécanismes d'indexation permettant de réaliser une large sélection d'éléments qui pourra être raffinée par la suggestion de critères supplémentaires. En cas de matching approché, il faut pouvoir estimer la proximité des résultats avec les critères de recherche pour les ordonner en fonction de leur pertinence et par exemple aider l'architecte à estimer le coût d'adaptation des éléments suggérés à ses besoins spécifiques. Cette information lui permettra de décider s'il convient de réutiliser partiellement des artéfacts existants ou s'il convient d'en créer de nouveaux.

On peut ainsi résumer la problématique de la mise en œuvre de la réutilisation dans les approches de développement, telle que celle proposée par l'ingénierie des ligne de produits logiciels, par l'ensemble des questions de recherche suivantes :

Q1 - Quels artefacts réutilisables doit-on produire pour faciliter les développements ?

Q2 - Comment identifier et produire des artefacts réutilisables par rétro-ingénierie de logiciels existants ?

Q3 - Comment indexer les artefacts réutilisables pour permettre leur recherche et leur sélection ?

Q4 - Comment estimer l'effort d'adaptation d'artefacts répondant partiellement ou approximativement à un besoin ?

Q5 - Comment assembler un ensemble d'artéfacts réutilisables pour produire un logiciel cohérent ?

Q6 - Comment gérer l'évolution d'un logiciel ?

Pour répondre à ces questions, l'ensemble des travaux de recherche que j'ai co-encadrés vise à concrétiser la mise en œuvre du concept de réutilisation par la proposition d'une démarche outillée de développement d'applications à l'aide de composants logiciels.

1.3 Développement par réutilisation de composants logiciels

Malgré l'importante activité de recherche ayant été menée ces dernières années dans le domaine de l'ingénierie dirigée par les modèles [31], approche dans laquelle nos travaux s'inscrivent pleinement, la réutilisation est actuellement essentiellement effective au niveau des implémentations et, comme nous l'avons souligné dans la section précédente, concerne essentiellement la partie technique des logiciels. On peut considérer ce dernier point comme naturel, le code technique correspondant à des besoins très génériques et ayant ainsi le plus fort potentiel de réutilisabilité.

Des concepts de nature et de granularité différentes, accompagnant l'évolution des pratiques et de paradigmes de développement, ont été proposés pour outiller la réutilisabilité : fragments de code (aspect, trait, mixin, snipet) [30, 52, 76], fonctions ou classes (bibliothèques, développement kits), squelette applicatif (frameworks), applications embarquées ou utilisées en mode client/serveur au travers d'une API.

Le concept de composant a été introduit pour améliorer la pratique de la réutilisation dans les processus de développement logiciels. Certaines définitions, trop génériques, le rendent parfois difficile à distinguer de celui de classes en programmation orientée objets, les deux concepts possèdant les propriétés de modularité, d'abstraction, de découplage et de composabilité qui permettent leur réutilisation [41].

L'une des définitions les plus pertinentes du concept de composants est donnée, à mon avis, dans [120] : "a software component is a unit of composition with contractually specified interfaces and explicit dependencies only. A sofware component can be deployed independently and is subject to composition by third parties".

Cette définition met en lumière la caractéristique fondamentale du concept de composant : le principe d'inversion de contrôle [72]. Un composant est un module décrit de manière abstraite par un ensemble d'interfaces qui définissent les interactions possibles entre le composant et son environnement. Le composant se comporte alors comme une boîte noire, cachant les détails et la complexité de son implémentation, utilisée exclusivement au travers de ses interfaces.

Un premier type d'interfaces, les interfaces fournies (interfaces serveur), décrit les connexions qui pourront être établies avec le composant pour utiliser les services qu'il implémente. Ces interfaces sont fonctionnellement comparables aux API (fonctionnalités publiques) des classes d'objets définies, par exemple, grâce au concept d'interface (type abstrait d'objets) dans le langage Java.

Un second type d'interfaces, les interfaces requises (interfaces clientes), décrit les connexions qui doivent être établies entre le composant et d'autres composants de son environnement afin d'utiliser leurs services. En d'autres termes, les interfaces requises définissent de manière explicite les dépendances du composant. Cette partie de la description externe des composants est distinctive : elle intègre l'inversion de contrôle comme une caractéristique du paradigme, exprimée à l'aide d'un concept dédié et non simplement comme l'application optionnelle d'un patron de conception. L'expression des dépendances au travers des interfaces requises renforce l'abstraction, le découplage et la composabilité

des composants, donc leur réutilisabilité.

Différents modèles de composants ont été proposés pour répondre aux besoins de domaines d'applications ou d'approches de développement spécifiques [40]. Ils diffèrent par les types d'interfaces qu'ils proposent : certaines interfaces gèrent des échanges d'événements ; d'autres la production et la consommation de flux de données ; d'autres encore la transmission d'un signal de contrôle gérant l'activation en chaîne des composants. Nous nous intéressons dans nos travaux à des interfaces fonctionnelles, gérant des interactions de type client/serveur entre composants. Dans ces modèles de composants, chaque interface est définie par un ensemble de signatures de fonctions. Les signatures attachées à une interface fournie (serveur) définissent les fonctions pouvant être invoquées sur le composant au travers de l'interface. Réciproquement, les signatures de fonctions attachées à une interface requise (cliente) définissent les invocations de fonctions qui peuvent être émises par le composant au travers de l'interface. Généralement, la représentation graphique de ces interfaces adoptent les conventions définies en UML [101], soit respectivement des "prises" en forme de cercle pour les interfaces fournies et en arc de cercle pour les interfaces requises (cf. figure 1.3).

FIGURE 1.3 – Représentation graphique des interfaces requises et fournies [101]

Certains modèles ont par ailleurs introduit le concept de port [14][63][101][118]. Alors qu'une interface décrit un point de communication unidirectionnel, du point de vue de l'émission ou de la réception des messages d'invocation de fonctions, centré sur une fonctionnalité unique (par respect du principe de séparation des préoccupations [85]), un port rassemble l'ensemble des interfaces fournies et requises nécessaires à l'exécution de processus transverses, impliquant des protocoles de collaboration complexes et éventuellement des invocations réciproques de fonctions entre composants. En UML, les ports sont graphiquement représentés par des carrés positionnés en bordure des composants (cf. figure 1.5).

La construction d'un logiciel à l'aide d'un ensemble de composants consiste à connecter des couples d'interfaces requises et fournies afin de définir les interactions client/serveur qui permettent son fonctionnement global (cf. figure 1.4). La connexion entre deux ports est réalisée par la connexion des interfaces qui les définissent. Dans certains modèles de composants, les connexions peuvent être directes. On parle alors de liaison ("binding") entre interfaces : l'interface requise possède la référence de l'interface fournie qui lui est connectée afin de pouvoir invoquer ses fonctionnalités. Au contraire, les connections peuvent être assurées par des éléments intermédiaires, appelés connecteurs, repré-

sentant comme des concepts de première classe les connexions entre composants. Les connecteurs servent à gérer des politiques spécifiques d'interaction entre les composants (distribution, sécurité, adaptation de protocole, ...) [125]. Dans nos travaux, à l'instar du modèle de composants Fractal [33], les connecteurs éventuels ne sont pas modélisés par des concepts spécifiques mais par des composants qui jouent fonctionnellement le rôle de connecteur entre d'autres composants.

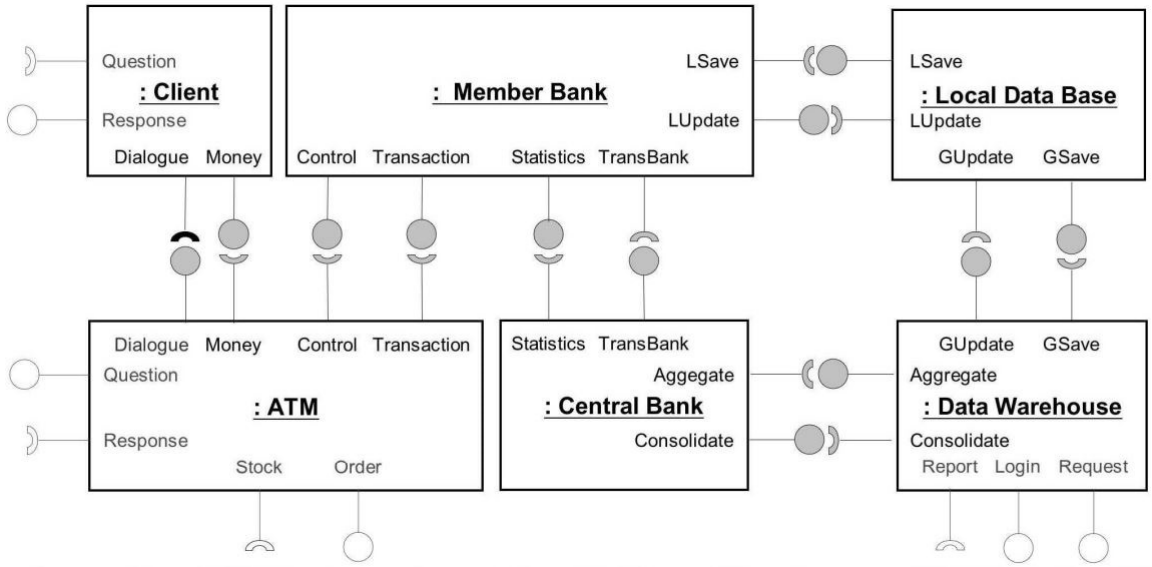


FIGURE 1.4 – Assemblage de composants logiciels

Le développement de logiciels à base de composants a ainsi conduit à la création de langages de description d'architectures (ADLs) [88]. Ces langages ont pour rôle la définition de l'ensemble des composants et des connections formant l'architecture d'un logiciel. Ces architectures peuvent être définies de manière hiérarchique à l'aide de composants composites. Il s'agit de composants définis récursivement par un ensemble de composants, connectés entre eux pour définir l'architecture interne du composant composite (cf. figure 1.5). Le composant composite encapsule son architecture interne et réalise son abstraction à l'aide d'un ensemble d'interfaces qui expose, par délégation, certaines interfaces de ses composants internes. Les composants composites sont autant un moyen de structurer et de réduire la complexité des architectures qu'un moyen de réutiliser des architectures en tant que composants. Les composants composites assurent ainsi la parfaite composabilité des composants : un assemblage de composants forme un nouveau composant à part entière.

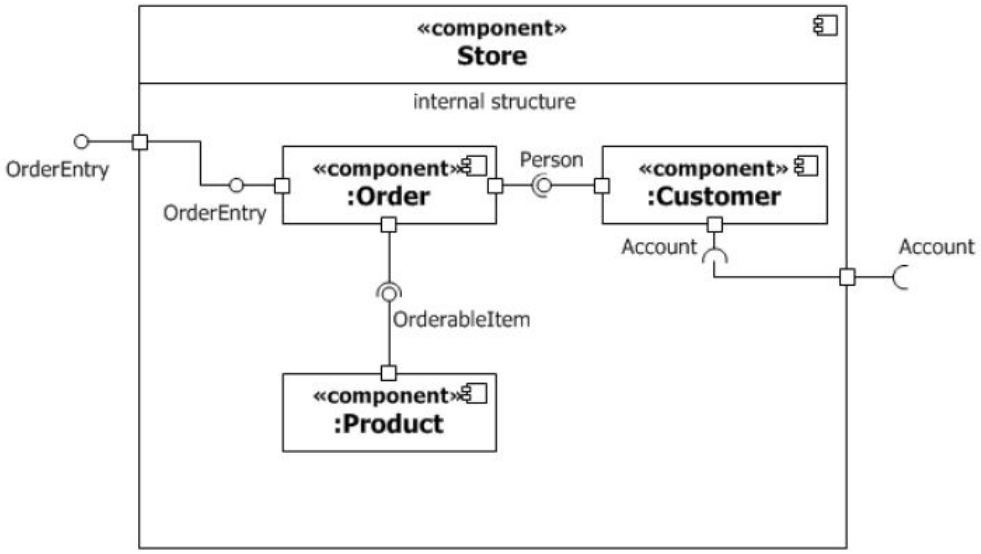


FIGURE 1.5 – Composant composite [101]

1.4 Organisation du mémoire

Ce mémoire est organisé en quatre chapitres.

Le chapitre 1 est constitué de la présente introduction.

Le chapitre 2 est classiquement consacré à une synthèse de nos principales contributions au domaine du développement d'applications à l'aide de composants logiciels.

La section 2.1 aborde les extensions des langages de représentation d'architectures à base de composants que nous avons proposées, afin d'améliorer le support qu'ils apportent aux processus de développement. Elle présente un modèle de port composites facilitant la construction d'architectures complètes et valides puis Dedal, un ADL multi-niveaux, facilitant la séparation des préoccupations dans les prises de décisions architecturales et la réutilisation de modèles d'architectures tout au long du processus de développement.

La section 2.2 décrit nos propositions pour assister le développement d'architectures à base de composants. Elle aborde les mécanismes de construction automatique d'architectures que nous avons conçus à l'aide de notre modèle de ports composites et appliqués entre autres au remplacement automatique de composants. La section présente ensuite les solutions que nous proposons pour traiter la problématique de la co-évolution des différents modèles décrivant une même architecture à différents niveaux d'abstraction (évaluation d'impact des changements, propagation des changements, ...). Elle aborde enfin nos propositions pour la construction automatique d'annuaires ou de bibliothèques de composants, organes essentiels pour référencer, rechercher et sélectionner les éléments disponibles dans d'un environnement de développement basé sur la réutilisation.

La section 2.3 décrit nos travaux sur l'adaptation autonomique d'architectures pour gérer des applications sensibles au contexte. Elle présente dans un premier temps SAASHA, un système multi-agents capables de modifier de manière réflexive leurs architectures internes à base de composants pour gérer le déploiement et l'exécution de scénarios définis par des ensembles de règles ECA. Une deuxième partie présente SAAS, un frame-

work SCA (Service Component Architecture [110]) permettant la définition, le déploiement et l'exécution d'orchestration de services. Une troisième partie présente Arold, un langage permettant de définir des stratégies de déploiement de missions, définies sous forme d'architectures SCA, sur un système multi-agents, en fonction des ressources matérielles et logicielles disponibles.

La section 2.4 présente enfin nos travaux dans le domaine de la rétro-ingénierie de lignes de produits logiciels. Cette section décrit un processus de localisation d'implémentation de features combinant à la fois des techniques d'analyse structurelle, d'analyse formelle de concepts (FCA) [26][60] et d'analyse lexicale (Latent Semantic Indexing) [78]. Ces techniques sont également appliquées à la documentation des applications, par exemple aux cas d'utilisations, pour extraire automatiquement une documentation des features, telle que des propositions de noms. Ces différentes informations sont utilisées pour produire automatiquement un modèle de features décrivant les configurations identifiées dans la famille de logiciels.

Le chapitre 3 présente un ensemble de perspectives qu'il serait intéressant de développer dans la suite de nos travaux.

Un premier ensemble de perspectives poursuit nos travaux sur l'automatisation du développement des applications (voir la section 3.1). Nous avons abordé cette problématique par une approche de search-based software engineering [69] (calcul automatique d'une solution par modélisation et exploration d'un espace de recherche à l'aide d'heuristiques) appliquée à la construction automatique d'architectures et à la génération de plans d'évolution. Une première perspective serait d'étendre la couverture du cycle de développement de nos travaux en étudiant l'automatisation du passage entre les exigences fonctionnelles et la spécification de l'architecture conceptuelle d'un logiciel.

De manière complémentaire, nous allons prolonger nos travaux sur la réutilisation des architectures pour en généraliser le principe. Nos travaux ont pour l'instant abordé la réutilisation des architectures au travers d'un mécanisme mécanisme de raffinement entre trois niveaux d'abstraction (conception, implémentation et instanciation) ainsi qu'un mécanisme d'évolution. Nous voulons étudier d'autres mécanismes permettant d'assurer la réutilisation d'architectures, en introduisant par exemple une relation de spécialisation entre architectures, qui permettrait de réutiliser des modèles d'architectures génériques, abstraits, pour concevoir des architectures spécifiques.

Nous allons également poursuivre nos travaux dans le domaine de la rétro-ingénierie d'architectures logicielles. Avec la généralisation de la programmation orientée-objets, des systèmes distribués et la part croissante de code réutilisé, la modularité et la complexité des logiciels n'a jamais été aussi forte, au point que certains logiciels s'envisagent maintenant comme des systèmes de systèmes [83]. Il nous semble primordial dans ce contexte de maîtriser l'ingénierie de leurs architectures au travers de représentations explicites et adaptées. Malheureusement, la documentation des projets, sous forme de modèles conceptuels, qui devraient être le produit naturel d'une phase de conception, préférentiellement dans une démarche d'ingénierie dirigée par les modèles, est trop souvent négligée. De manière pragmatique, nous voulons identifier et documenter a posteriori les architectures logicielles à bases de composants qui structurent implicitement les logiciels.

Enfin, il serait intéressant d'étudier la généralisation des concepts de composants et d'architectures que nous avons introduits dans notre ADL Dedal de manière à étendre la portée de son l'utilisation. En effet, le principe de modélisation de systèmes comme un ensemble de modules gérant leurs interactions avec leur environnement au travers

d'interface est assez universel. Nous voudrions ainsi étudier les ADLs comme des outils génériques d'ingénierie, servant de base au développement d'outils spécifiques par raffinement ou extension des concepts d'architecture et de composant pour produire des DSLs (Domain Specific Language) [59].

La section 3.2 décrit un ensemble de perspectives visant à d'aider les équipes de développement à adopter les bonnes pratiques du génie logiciel.

Malgré les bénéfices de moyen et long termes apportés par une approche disciplinée de développement, on ne peut que constater et regretter que de nombreux projets n'aient d'autres objectifs que la satisfaction au moindre coût des exigences fonctionnelles contractées. Ce contexte explique le succès des approches agiles [38], basées sur des règles simples, pragmatiques, compatibles avec la réalité de terrain. Au lieu d'imposer un cadre méthodologique et conceptuel complexe, elles proposent des principes de bon sens, facilement adaptables aux différentes pratiques de développement, permettant d'assurer a minima la maîtrise d'un projet en termes planification et de qualité. Dans le même esprit, je souhaite étudier des mécanismes de génie logiciel qui accompagnent les développements de manière opportuniste et éventuellement transparente, au lieu de contraindre les développeurs et de générer la perception négative d'une charge supplémentaire.

Une première perspective consiste à étudier la (rétro)-ingénierie de lignes de produits logiciels par exploitation des historiques de versionnement disponibles dans les dépôts de code.

Une autre perspective est d'exploiter les grandes masses d'information de suivi de projet rendues disponibles grâce aux dépôts de code open source, tel que GitHub ou BitBucket. De manière complémentaire à la capacité de faire la rétro-ingénierie de descriptions architecturales depuis le code des applications ainsi que la rétro-ingénierie de lignes de produits depuis les historiques des dépôts de code, l'idée générale est d'élaborer des métriques permettant d'évaluer la qualité d'une architecture dans sa capacité, par exemple, à limiter la dette technique, ou à faciliter ses futures évolutions.

Une perspective plus générale est de favoriser l'adoption des bonnes pratiques du génie logiciel dans l'industrie en les rendant aussi intégrées, assistées ou transparentes que possible. L'Ingénierie Dirigée par les Modèles fournit des solutions à cette problématique, en augmentant la proportion de code généré mettant systématiquement en œuvre les meilleurs principes du génie logiciel. Mais les développeurs perçoivent la gestion conjointe des modèles et des codes sources comme une difficulté, une charge supplémentaire. Une idée alternative est de doter les langages de programmation existants d'une syntaxe graphique afin que programmation et conception soient envisagées comme une seule et même activité de modélisation de la structure et du comportement d'un logiciel, à différents niveaux d'abstraction, dans un processus de développement par raffinement.

Dans un même esprit d'amélioration de l'outillage, afin qu'il permette une meilleure agilité dans les processus de développement, une perspective est d'étudier comment réaliser de manière pragmatique l'ingénierie des exigences par la co-conception, avec les utilisateurs finaux, de maquettes d'interfaces (UI) et de scénarios d'utilisation (UX).

Le chapitre 4 propose une conclusion sous la forme d'une synthèse (voir la section 4.1) et d'une présentation de la sélection d'articles proposée dans les annexes pour illustrer notre travail de recherche (voir la section 4.2).

Chapitre 2

Synthèse des contributions

« *Nothing can cure the soul but the senses*
just as nothing can cure the senses but the soul. »

Oscar Wilde, The Picture of Dorian Gray (1890)

2.1 Extension des langages de description d'architectures logicielles

2.1.1 Modélisation de ports composites

Comme introduit dans la section 1.3, l'objectif d'un composant est de permettre la réutilisation d'un module en s'appuyant sur une documentation abstraite décrivent les interactions du composant avec son environnement [120]. Une première catégorie d'interactions est définie par les interfaces fournies. Elles décrivant les fonctionnalités pouvant être invoquées par d'autres composants. La deuxième catégorie d'interactions est définie par les interfaces requises. Elles décrivent les fonctionnalités que le composant doit invoquer sur des composants de son environnement, en d'autres termes ses dépendances fonctionnelles.

Le concept d'interface répond ainsi correctement au principe d'inversion de contrôle [72], qui assure un bon découplage des composants. Mais nous avons rapidement constaté ses limites en termes de réutilisabilité effective des composants.

En effet, le concept d'interface ne permet pas de documenter les différents cas d'usage d'un composant, ce qui faciliterait le travail des architectes en définissant les différents ensembles d'interface à connecter en fonction des objectifs fonctionnels souhaités. La figure 2.1 illustre cette problématique. Ayant pour objectif fonctionnel de connecter l'interface *Dialogue* du composant *Client*, l'architecture proposées est-elle complète, considérant que certaines interfaces requises ne sont pas connectées ? Comment choisir les connexions si plusieurs possibilités existent au regard de la compatibilité (typage) des interfaces ? Comment s'assurer qu'une architecture, complète et sans ambiguïté d'un point de vue structurel, soit cohérente d'un point de vue fonctionnel ?

Par défaut, dans la plupart des modèles, il faut connecter toutes les interfaces requises définies, quel que soit l'usage effectif des composants, ce qui multiplie les dépendances à satisfaire et limite ainsi les possibilités de réutilisation [5]. D'autres modèles [122][34] ont introduit la notion d'interface requise facultative, mais sans support pour permettre aux architectes de déterminer les ensembles cohérents d'interfaces à connecter en fonction des scénarios d'utilisation des composants. D'autres modèles proposent ainsi de calculer automatiquement les dépendances entre interfaces en utilisant différentes modélisations formelles du comportement des composants [42][61][106][107][119]. Mais cette solution, si elle évite les incohérences, ne propose pas de documentation explicite des différents cas d'usage implémentés par les composants.

C'est pour répondre à cette problématique que nous avons conçu un modèle de composants, pour améliorer les ADLs existants, dont l'une des originalités est de proposer le concept de port composite [45][46][48][47]. Dans notre modèle, la base de la description d'un composant reste un ensemble d'interfaces requises et fournies. Elles constituent les points de communication qui, une fois liés par des connexions, permettent les échanges de messages entre composants. Elles sont décrites par un ensemble de signatures de fonctions définissant les contenus possibles des messages (émissions ou réceptions d'invocations de fonctions). Elles sont par ailleurs décrites par un protocole dit de communication [104]. Il s'agit d'une expression régulière définissant toutes les séquences valides de messages pouvant être émis ou reçus par une interface requise ou fournie, conformément au protocole de comportement du composant (expression régulière décrivant toutes les sé-

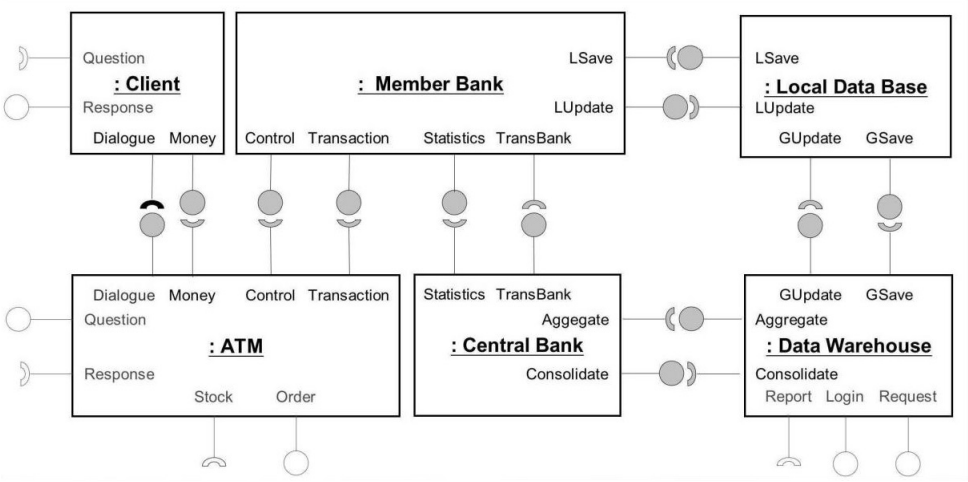


FIGURE 2.1 – Architecture construite par connexion d'interfaces

quences d'émission et de réception de messages acceptées par le composant). L'ensemble de ces éléments (direction, signatures de fonctions et protocole de communication) définit le type abstrait d'une interface.

Nous utilisons alors le concept de ports, tel qu'il existe dans de nombreux modèles [14][63][101][118], mais en lui donnant le rôle conceptuel spécifique de documenter les cas d'usage des composants. Dans notre modèle, un port regroupe l'ensemble des interfaces requises et fournies qui sont nécessaires à l'exécution d'un cas d'usage, en tant que points de communication élémentaires. Le port est par ailleurs documenté par un protocole dit de collaboration. Il s'agit également d'une expression régulière mais décrivant toutes les séquences d'émission et de réception de messages, échangées au travers des interfaces du port, qui constituent des scénarios valides d'utilisation du composant permettant d'atteindre un ensemble d'objectifs fonctionnels déterminés. Notre définition du concept de port explicite ainsi les dépendances existant entre les différentes interfaces d'un composant et les contextualise en les associant à la réalisation de contrats logiciels. Le protocole de collaboration associé à un port doit être conforme aux protocoles de communication associés à ses différentes interfaces.

Les ports fournissent alors un moyen structurel, simple, de vérifier certaines propriétés de cohérence et de complétude des architectures. En effet, la connexion d'un port, par la connexion de l'ensemble de ses interfaces, rend possible l'exécution de l'ensemble des scénarios définis par le protocole de collaboration du composant. Ainsi, sur la figure 2.2, la connexion du port *Money_Dialogue* du composant *Client* garantit de pouvoir exécuter tous les scénarios de retrait d'argent qui lui sont associés.

Mais la description de certains ports peut alors devenir complexe (forte granularité). Par ailleurs, si un port définit l'ensemble des interfaces à connecter, il ne définit pas si ces interfaces doivent être connectées aux interfaces d'un même port (donc implicitement à un même composant). Enfin, il convient de fournir une solution adaptée à la composition hiérarchique des composants permettant de définir les ports d'un composant composite à partir des ports de ses composants internes.

Pour répondre à ces questions, nous avons proposé un modèle de port composite qui reste, à notre connaissance, une proposition originale. L'idée première de ce modèle est

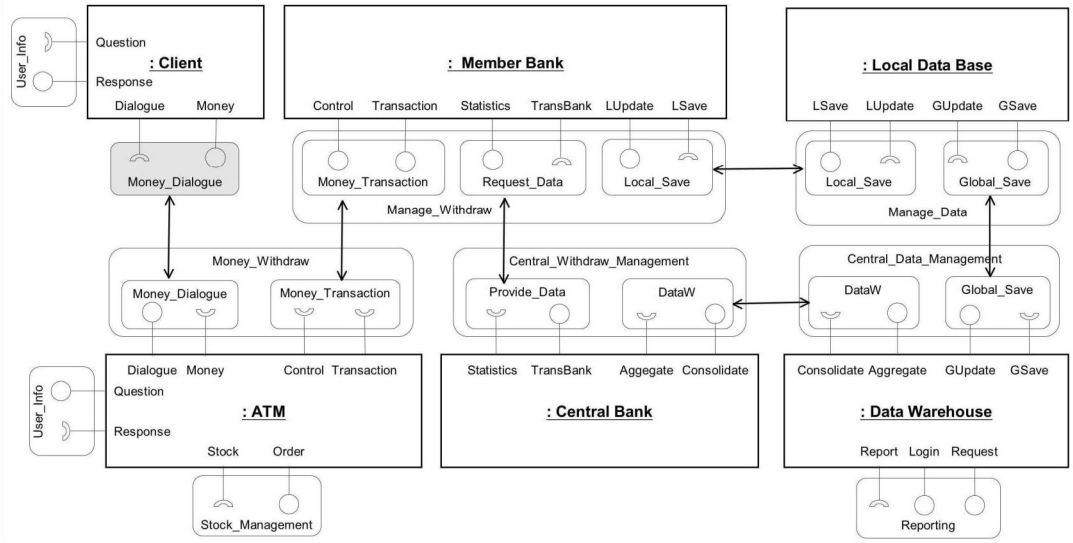


FIGURE 2.2 – Architecture construite par connexion de ports

qu'un port de forte granularité, supportant un ensemble complexe de scénarios d'exécution, peut être défini par la composition hiérarchique d'un ensemble de ports de plus faible granularité, jusqu'à la définition de ports directement composés d'interfaces, appelés ports primitifs. Sur la figure 2.2, le port *Money_Withdraw* est ainsi composé de deux ports primitifs, *Money_Dialogue* et *Money_Transaction*.

Ce modèle offre réciproquement la possibilité de réutiliser des ports existants pour définir, par composition, de nouveaux ports. Cette possibilité offre en particulier un support naturel à la définition des ports des composants composites par composition des ports de ses composants internes. Par comparaison, UML ne propose que la relation de délégation pour lier les ports externes du composant composite et les ports de ses composants internes (voir la figure 2.3). Cela introduit une distorsion entre la composabilité des composants et la non-composabilité de leurs ports.

Par ailleurs, ce modèle de port composite nous a permis de préciser et de formaliser les règles de connexion des composants. L'obligation de connexion de l'ensemble des interfaces d'un port permet d'expliciter les dépendances comportementales existant entre les interfaces d'un composant. Mais elle n'offre aucun guide à l'architecte sur le choix des composants et des interfaces à utiliser pour établir les connexions nécessaires. Or certaines collaborations nécessitent qu'une partie des interactions soient réalisées par un couple de composants pour produire un résultat cohérent (gestion d'un état conversationnel). Nous avons ainsi imposé que la connexion d'un port primitif soit une opération atomique, connectant l'ensemble des interfaces d'un port primitif à l'ensemble des interfaces d'un autre port primitif. La validité des connexions entre interfaces repose sur des règles de typage (voir la section 2.2.3). Ainsi, toutes les interfaces d'un port atomique doivent être connectées non seulement simultanément mais également à un même autre port (donc à un même autre composant). Les ports atomiques permettent ainsi de traduire toutes les dépendances comportementales existant entre les interfaces. Sur la figure 2.2, les deux interfaces du port *Money_Dialogue* du composant *Client* seront connectées à un même port et établiront donc une collaboration avec un composant unique, ce qui permettra de gérer de manière cohérente la session du composant *Client*.

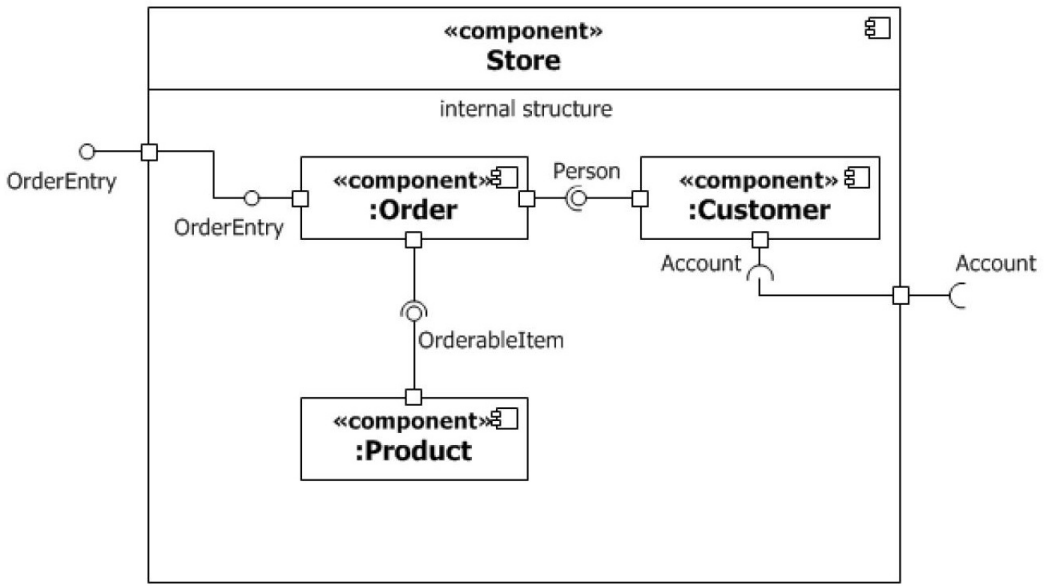


FIGURE 2.3 – Example de composant composite en UML 2.5 [101]

Mais cette contrainte forte peut rendre difficile la connexion de ports atomiques de forte granularité, qui possèdent alors un type très spécifique. Nous avons donc proposé d'utiliser notre modèle de ports composites pour relâcher cette contrainte, en permettant la connexion des différents ports composant un port composite à des composants différents. Sur la figure 2.2, le composant ATM est connecté via les deux ports composants de son port *Money_Withdraw* aux composants *Client* et *Member_bank*. Un port composite est alors correctement connecté lorsque tous les ports primitifs qui le composent récursivement sont correctement connectés.

Notre modèle de ports composites fournit ainsi aux architectes un moyen intuitif de construire des architectures offrant de bonnes garanties de correction structurelle et comportementale. En effet, une architecture est considérée comme correcte lorsque tous les ports de ses composants sont dans un état cohérent, c'est-à-dire entièrement connectés ou déconnectés, ce qui est une propriété très simple à calculer. Les figures 2.2 et 2.4 fournissent des exemples d'architectures correctes permettant de satisfaire un même objectif fonctionnel qui est de pouvoir exécuter les scénarios de retrait d'argent associés au port *Money_Dialogue* du composant *Client*.

Cette propriété de correction, basée sur des règles de typage entre interfaces et ports, ne fournit qu'une condition nécessaire à une preuve de correction comportementale de l'architecture, qui ne peut être établie qu'avec les moyens des méthodes formelles appliquées à des descriptions comportementales plus précises [89]. Mais à l'image du typage fort, introduit dans les langages de programmation pour éliminer dès l'étape de compilation certaines incohérences dans la définition d'expressions ou dans l'utilisation de fonctionnalités, réduisant ainsi le recours aux tests pour détecter les problèmes de réalisation, notre modèle de ports composites est proposé comme un premier filtre permettant d'explorer des possibilités de construction d'architectures a priori correctes, avant le recours, pour des applications critiques, à une validation formelle.

Nous avons expérimenté ce concept en l'intégrant au framework Fractal [33]. Une par-

tie de cette contribution est détaillée par l'article présenté dans la section 4.2.4 et inclus dans l'annexe E.

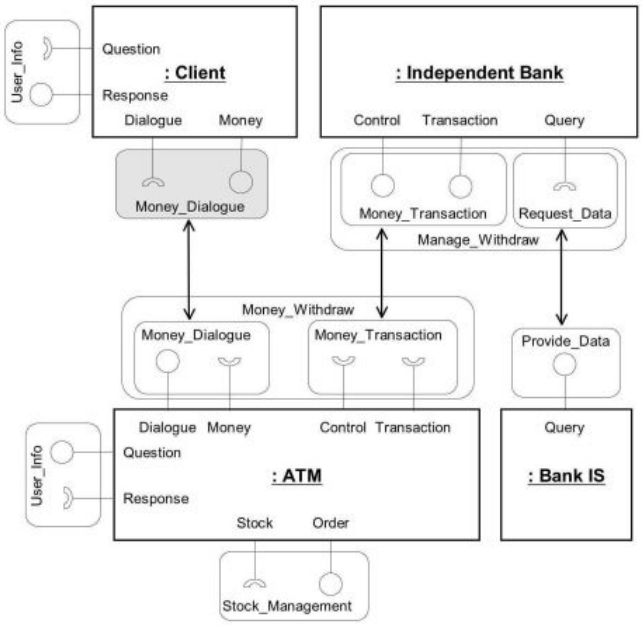


FIGURE 2.4 – Complétude et cohérence d'architectures construites à l'aide de ports composites

2.1.2 Représentation d'architectures logicielles suivant différents niveaux d'abstraction

Le développement par réutilisation de composants conduit à modéliser des architectures logicielles décrivant l'ensemble des composants et des connexions qui définissent l'organisation structurelle des applications. Ces tâches de modélisation d'architectures sont outillées par des langages dédiés, appelés langages de description d'architectures (ADLs) [88]. En étudiant les ADLs existants, dans une perspective d'ingénierie logicielle dirigée par les modèles [31], nous sommes arrivés à la conclusion qu'ils étaient essentiellement destinés à gérer le déploiement d'applications et qu'ils ne proposaient pas suffisamment de niveaux d'abstraction pour supporter un processus de développement complet, par raffinements successifs de modèles.

Pour répondre à cette problématique, nous avons conçu Dedal [129][126][127][128], un ADL proposant trois niveaux d'abstraction distincts dans la modélisation des architectures logicielles. Ces trois niveaux ont pour objectif de permettre une représentation séparée des décisions architecturales prises lors des phases principales du processus de développement : la conception, l'implémentation et le déploiement des architectures (voir la figure 2.5).

Un premier type de modèles, appelé *spécification d'architecture*, décrit des modèles conceptuels d'architectures, traduisant les exigences fonctionnelles issues d'un cahier des charges en une solution idéale. Une spécification d'architecture est composée de composants conceptuels appelés *rôles* (voir la figure 2.6). Les rôles décrivent les types de composants idéaux permettant de réaliser l'architecture conformément aux exigences et indépendamment de toute contrainte technique ou contextuelle. Une spécification

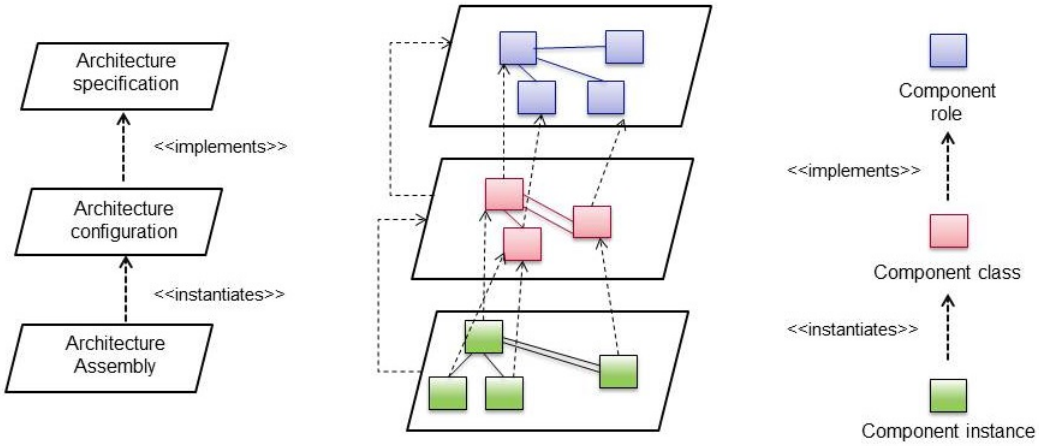


FIGURE 2.5 – Les trois niveaux d'abstraction de l'ADL Dedal

d'architecture contient ainsi uniquement les décisions architecturales correspondant à la conception d'une solution répondant au cahier des charges. Si l'on se réfère à l'approche MDA [86], une spécification d'architecture peut être vue comme un PIM (platform independant model).

```
1 component_role BikeCourse
2  required_interfaces BikeQS; CourseQS
3  provided_interfaces BikeOprs; CourseOprs
4  component_behavior
5   (!BikeCourse.BikeOprs.selectBike,
6   ?BikeCourse.BikeQS.findBike;)
7   +
8   (!BikeCourse.CourseOprs.startC,
9   ?BikeCourse.CourseQS.findCourse;)
```

FIGURE 2.6 – Définition d'un rôle en Dedal

Un deuxième type de modèles, appelé *configuration d'architecture*, décrit des modèles concrets d'architectures, définissant comment une spécification d'architecture est réalisée. Une configuration d'architecture est composée de *classes de composants* (voir la figure 2.7). Une classe de composants décrit une implémentation concrète d'un type de composants. Une configuration d'architecture est typiquement définie comme le raffinement d'une spécification d'architecture, consistant à associer à chaque rôle de composants de la spécification une classe de composants définissant son implémentation. Une configuration d'architecture contient ainsi toutes les décisions prises lors de la réalisation de l'architecture. Si l'on se réfère à l'approche MDA [86], une configuration d'architecture peut être vue comme un PSM (platform specific model). Dans un processus de développement par réutilisation, les rôles de composants servent à définir les requêtes adressées aux bibliothèques pour rechercher et sélectionner parmi les classes de composants disponibles celles qui seront utilisées (voir la section 2.2.3).

Un troisième type de modèles, appelé *assemblage d'architecture*, décrit des d'architectures destinées à être déployées et exécutées. Un assemblage est composé d'instances de classes de composants. Un assemblage peut être défini comme le raffinement d'une

```
1 component_class BikeTrip
2  implements BikeTripType
3  using fr.ema.BikeTripImpl
4  versionID 1.0
5  attributes string company; string currency
```

FIGURE 2.7 – Définition d'une classe de composants en Dedal

configuration d'architecture, les instances de l'assemblage étant associées aux classes de l'implémentation. Chaque instance de composant décrivant une personnalisation, un paramétrage particulier, un assemblage contient toutes les décisions nécessaires à une utilisation spécifique de l'architecture.

Dedal rend ainsi possible la réutilisation des définitions d'architectures : une spécification peut être réutilisée pour définir une nouvelle configuration d'une architecture ; une configuration peut être réutilisée pour définir un assemblage à déployer dans un nouveau contexte d'exécution (voir la figure 2.8).

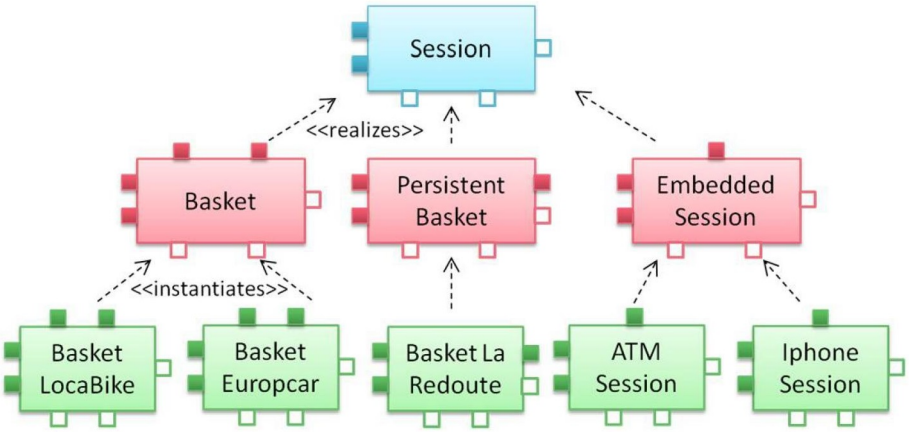


FIGURE 2.8 – Réutilisation de modèles d'architectures avec Dedal

Ces mécanismes de réutilisation et de raffinement sont basés sur un ensemble de relations liant les différents modèles décrivant une même architecture aux différents niveaux d'abstraction (voir la figure 2.5). Les modèles de configuration sont liés aux modèles de spécification par des relations d'implémentation. Les modèles d'assemblages sont liés aux modèles de configuration par des relations d'instanciation. Chaque modèle d'architecture étant défini par un ensemble de composants, la sémantique de ces relations dérive des relations existant entre leurs composants. Ainsi, les rôles et les classes de composants sont liés par des relations d'implémentation. De même, les (instances de) composants et les classes de composants sont liés par des relations d'instanciation. Il est alors possible d'établir, par exemple, qu'une configuration d'architecture est une implémentation conforme d'une spécification d'architecture lorsque tous les rôles de la spécification ont une implémentation conforme dans la configuration. La conformité de la relation d'implémentation qui doit être établie entre les rôles et les classes de composants repose sur des règles de typage (voir la section 2.2.3) : le type implémenté par la classe de composants (voir la figure 2.9) doit être une spécialisation du type défini par le rôle.

```
component_type BikeTripType
  required_interfaces BikeQS ; CourseQS; LocOprs
  provided_interfaces BikeOprs; CourseOprs
  component_behavior
    (?BikeTripType.BikeOprs.selectBike,
    ?BikeTripType.LocOprs.findStation,
    !BikeTripType.BikeQS.findBike;)
    +
    (?BikeTripType.CourseOprs.startC,
    !BikeTripType.CourseQS.findCourse;)
```

FIGURE 2.9 – Définition d'un type de composants en Dedal

Définies dans un premier temps de manière conceptuelle [129][126][127], ces relations ont été redéfinies en langage B lors de la formalisation du métamodèle de Dedal [93][94]. Cette formalisation permet de calculer des propriétés de correction des différents modèles d'une architecture (complétude, connexité, validité des connexions) mais également de cohérence globale entre les différents modèles. Dedal supporte ainsi des relations complexes, telle que la relation d'implémentation entre les rôles d'une spécification et les classes d'une configuration : un rôle peut être implémenté par un ensemble de classes ou inversement une classe peut implémenter plusieurs rôles (relation n-m). La figure 2.10 donne un exemple de rôle implémenté par un ensemble de classes de composants pouvant être encapsulées dans un composant composite.

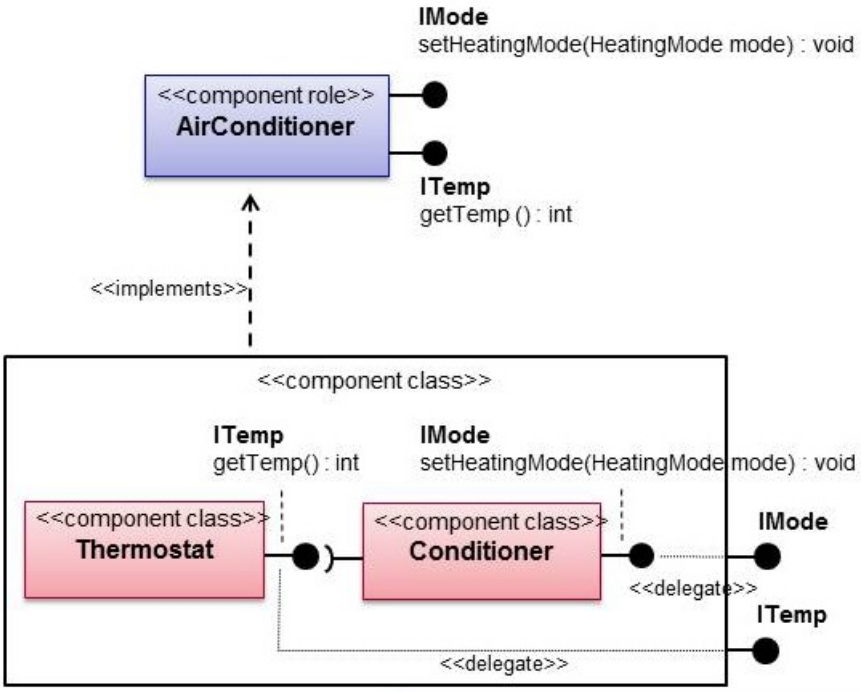


FIGURE 2.10 – Implémentation d'un rôle par un ensemble de classes de composants

Dedal est implémenté dans un atelier prototype (voir la figure 2.11) basé sur EMF/Sirius (syntaxe abstraite et syntaxe concrète graphique). La vérification des propriétés repose sur un métamodèle défini en langage B et l'intégration de l'outillage ProB à notre atelier.

Ces travaux sont en partie détaillés par l'article présenté dans la section 4.2.1 et inclus

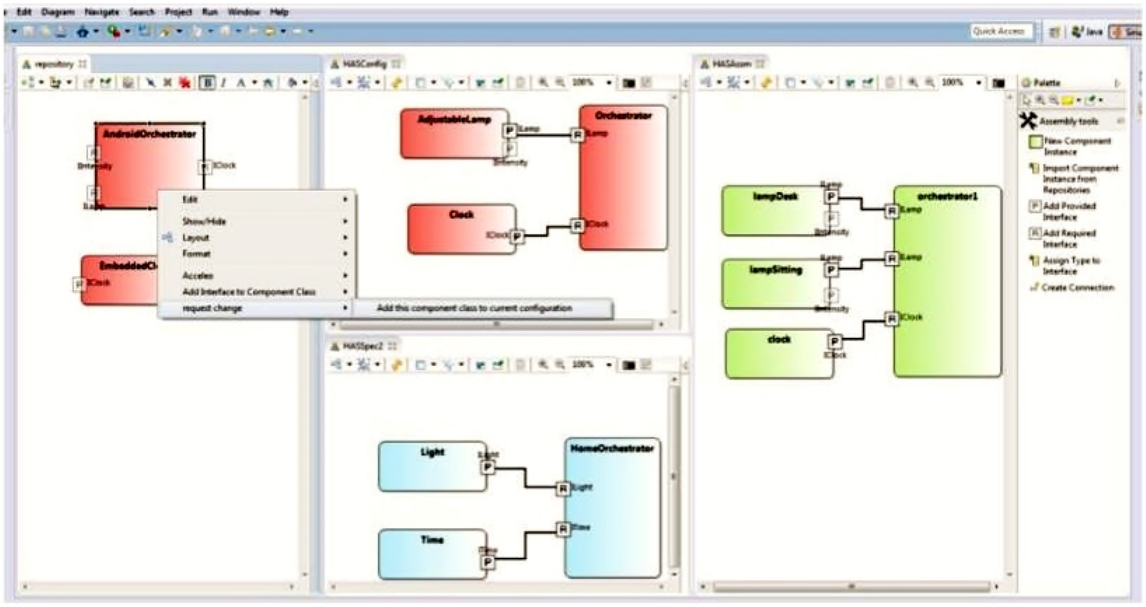


FIGURE 2.11 – DedalStudio, l'AGL prototype développé pour l'ADL Dedal

dans l'annexe B.

2.2 Assistance au développement d'architectures logicielles à base de composants

2.2.1 Construction automatique d'architectures logicielles à base de composants

Comme présenté dans la section 2.1.1, notre modèle de ports composites [48] fournit une assistance guidant la construction d'architectures à base de composants et permet d'évaluer très simplement certaines propriétés de complétude et de correction des architectures. Nous avons utilisé ces caractéristiques, dans une approche de search-based software engineering [69], pour concevoir un mécanisme automatique de construction d'architectures [45][46][47].

L'idée principale de ce mécanisme est de modéliser la construction d'une architecture à base de composants comme un problème de satisfaction de contraintes puis d'utiliser un algorithme de résolution pour obtenir automatiquement la définition d'une architecture solution.

Un premier ensemble de contraintes traduit les propriétés de bonne construction d'une architecture (complétude et correction). Comme présenté dans la section 2.1.1, elles correspondent à un état cohérent de tous les ports des composants de l'architecture (entièrement connectés ou déconnectés). Un deuxième ensemble de contraintes modélise les objectifs fonctionnels recherchés. Elles correspondent à la nécessité de connecter l'ensemble des ports associés aux scénarios fonctionnels à exécuter.

Dans l'exemple de la figure 2.2, l'objectif fonctionnel correspond à la connexion du port *Money_Dialogue* du composant *Client*. La nécessité de connecter ce port entraîne l'ajout du composant *ATM* à l'architecture et la connexion de son port *Money_Dialogue*. Cette connexion entraîne la nécessité de connecter le port *Money_Transaction* afin que le port composite *Money_Withdraw* soit dans un état cohérent. Le processus automatique de construction se déroule ainsi, de manière itérative, jusqu'à ce que toutes les contraintes soient satisfaites.

Nous avons conçu un algorithme qui, étant donnés une bibliothèque de composants et un ensemble d'objectifs fonctionnels, calcule toutes les solutions de construction possibles. Cet algorithme utilise classiquement une recherche arborescente associée à un mécanisme de backtracking [77] pour réaliser l'exploration complète de l'espace de recherche. Cet espace est modélisé par un ensemble de variables représentant les ports des composants appartenant à l'architecture. Chaque variable a pour domaine de valeur l'ensemble des ports compatibles des composants référencés dans les bibliothèques disponibles, plus la valeur *nil* correspondant à la déconnexion du port. Un ensemble de contraintes portant sur la valeur des variables traduit les propriétés de correction et de complétude à établir pour l'architecture.

Nous avons optimisé cette exploration complète de l'espace de recherche en adoptant un ensemble de stratégies classiques [77], telle qu'une réduction des domaines de valeur des variables par arc consistance, ainsi qu'un branch-and-bound basé sur le nombre de connexions utilisées dans l'architecture. Nous privilégions ainsi la recherche des architectures les plus simples en nombre d'éléments, en considérant qu'elles sont en général

également les plus efficaces et économes, voire les plus fiables, en absence de propriétés permettant de faire des calculs ou des estimations de qualités non fonctionnelles [121]. Ce branch-and-bound est accompagné d'une stratégie de prévision des connexions minimales requises par un composant (look-ahead), permettant d'éliminer a priori les composants qui engendreraient une solution trop complexe.

De même, nous avons expérimenté un ensemble d'heuristiques permettant d'accélérer la recherche : le choix prioritaire des ports ayant le moins de possibilités de connexion (détection précoce des échecs) ; la sélection prioritaire des ports dont la connexion induit le moins de connexions supplémentaires (ports primitifs appartenant aux ports composites de plus faible granularité). Un prototype a été réalisé et intégré au framework Fractal [33] puis validé sur des bases de composants générées aléatoirement, avec différents niveaux de complexité.

Nous avons également proposé ce mécanisme de construction automatique comme une assistance à l'évolution d'architectures. Dans ce contexte, l'objectif est de calculer une solution pour remplacer un composant supprimé. Au-delà d'une simple substitution par un composant appartenant au même type ou à un sous-type (voir la section), le mécanisme est capable de générer un assemblage de composants pour compléter l'architecture, donc de proposer une substitution 1-n.

Pour affiner notre assistance à l'évolution, nous avons ajouté au mécanisme de reconstruction une détection de "composants morts" (voir la figure 2.12). Il s'agit des composants qui ne sont plus reliés aux objectifs fonctionnels de l'architecture, après la suppression du composant à remplacer. Conformément à notre stratégie de recherche de la solution la plus simple, et pour permettre la recherche de toutes les solutions possibles, les composants morts sont supprimés de l'architecture pour ne conserver que les contraintes de connexion de ports servant à supporter directement les objectifs fonctionnels.

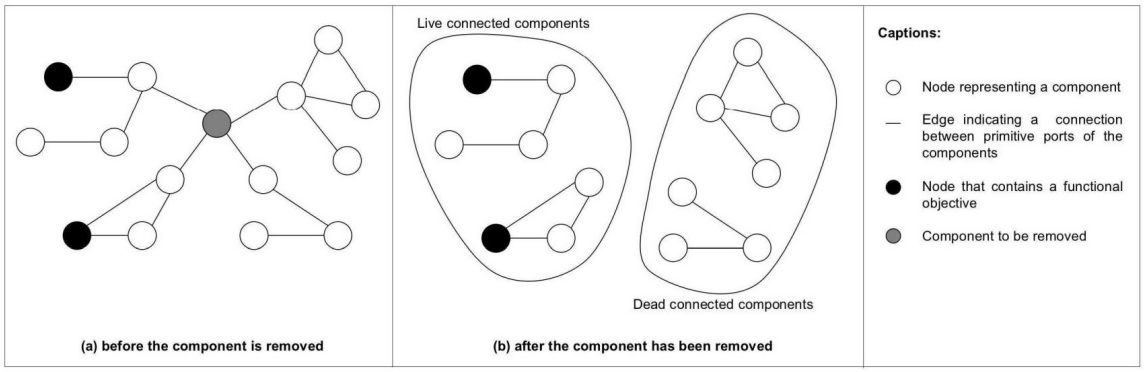


FIGURE 2.12 – Détection des composants morts

La figure 2.13 donne un exemple concret de reconstruction. Le composant *MemberBank* est supprimé de l'architecture. Les composants *LocalDataBase, CentralBank* et *DataWarehouse* ne sont plus reliés à un objectif fonctionnel de l'architecture (représenté par le composant *Client*). Ils sont supprimés conjointement de l'architecture (leur appartenance à l'architecture n'était justifiée a priori que par la satisfaction de dépendances du composant *MemberBank*). Une architecture différente, éventuellement plus simple, peut alors être reconstruite. S'il existe une substitution 1-1 pour le composant supprimé, l'an-

cienne architecture fournit une première solution et une première borne pour le branch-and-bound.

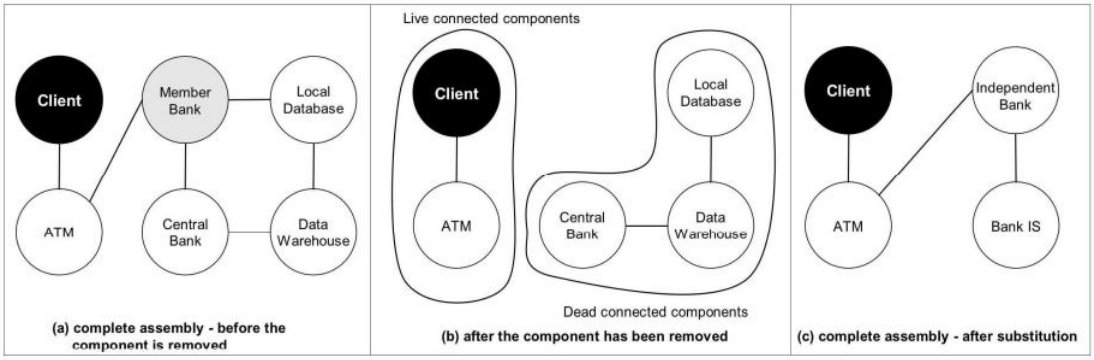


FIGURE 2.13 – Reconstruction automatique d'architectures

Ce travail est en partie détaillé par l'article présenté dans la section 4.2.4 et inclus dans l'annexe E.

2.2.2 Co-évolution de modèles d'architectures logicielles multi-niveaux

Comme présenté dans la section 2.1.2, nous avons proposé Dedal, un ADL permettant de définir des modèles d'architectures ayant trois niveaux d'abstraction distincts, afin de représenter séparément les décisions architecturales qui interviennent dans les phases principales du processus de développement d'une application (conception, implémentation, déploiement). Cette séparation des niveaux d'abstraction rend possible la réutilisation des modèles d'architectures abstraits pour définir, par raffinement, des architectures concrètes (par exemple différentes implémentations ou différents déploiements d'un même type d'architecture). Pour prolonger cette possibilité de réutiliser des modèles d'architectures dans les processus de développement, nous avons étudié des mécanismes d'évolution de modèles d'architectures [98][97][96] [95].

L'objectif est de permettre soit la maintenance corrective ou adaptative d'une architecture existante, pour prévenir son obsolescence [79][80], soit la production d'une nouvelle architecture dérivant d'une architecture existante, dans une approche opportuniste de "clone-and-own" (copie et modification) [57] ou disciplinée d'ingénierie de ligne de produits [15].

Gérer l'évolution des architectures consiste en particulier à prévenir les incohérences pouvant apparaître entre les différents modèles (niveaux d'abstration) définissant une même architecture, typiquement entre les modèles conceptuels (design-time) et les modèles exécutés (runtime) ou entre les modèles abstraits (spécifications) et les modèles plus concrets (implémentations). Ces incohérences sont classiquement appelées érosion [87] et dérive (drift) [103]. L'érosion correspond à la violation de décisions architecturales définies par les modèles abstraits dans les modèles plus concrets. La dérive correspond à l'introduction de décisions architecturales dans un modèle concret, sans violer les décisions architecturales définies dans les modèles plus abstraits, mais sans mise-à-jour des modèles plus abstraits.

Pour prévenir l'érosion et la dérive, nous avons choisi, parmi les différentes solutions possibles [43], de proposer un mécanisme de co-évolution des différents modèles définissant une même architecture, basé sur une analyse d'impact et une propagation des changements.

Notre mécanisme de co-évolution met en œuvre le processus suivant :

— Analyse locale de l'impact des changements. Cette étape consiste à vérifier que les changements n'invalident pas la correction du modèle d'architecture sur lequel ils sont initiés, à un niveau d'abstraction donné, indépendamment des modèles correspondant aux autres niveaux d'abstraction. La correction intra-niveau sur des propriétés basées sur les relations intra-niveau liant les éléments au sein d'un même modèle d'architecture (voir la section 2.1.2) telles que la connexité de l'architecture, la validité des connexions entre interfaces (compatibilité des types), la satisfaction des dépendances des composants (connexion des interfaces requises).

— Restauration de la correction (locale). Lorsque les changements réalisés impactent la correction du modèle d'architecture, des modifications complémentaires sont réalisées pour rétablir les propriétés qui ne sont plus vérifiées (suppressions/ajouts de composants ou de connexions). Nous parlons alors de propagation locale des changements (les changements initiaux sont préservés mais impliquent des changements supplémentaires).

— Analyse globale de l'impact des changements. Une fois la modification initiale réalisée et vérifiée par les deux premières étapes, nous procédons à une vérification de la cohérence globale de la définition de l'architecture, fournie par les modèles correspondant aux différents niveaux d'abstraction. La vérification de la cohérence globale repose sur des propriétés liées aux relations inter-niveaux existant entre les différents modèles d'une architecture : le modèle de configuration d'une architecture doit être une implémentation de son modèle de spécification ; un modèle de déploiement doit être une instanciation de son modèle de configuration. Ces relations sont elles-mêmes basées sur des relations inter-niveaux existant entre les composants définissant les différents modèles d'une architecture. Par exemple, chaque rôle du modèle de spécification doit être implémenté dans le modèle de configuration par une classe de composants dont le type spécialise le type du rôle. Autrement dit, les classes de composants du modèle de configuration doivent être liées aux rôles du modèle de spécification par des relations d'implémentation. De même, chaque classe de composants du modèle de configuration doit être instanciée dans le modèle de déploiement, ce qui se traduit par des relations d'instanciation entre les instances de composants du modèle de déploiement et les classes de composants du modèle de configuration. Les problèmes de cohérence globale (érosion ou dérive) se traduisent ainsi par la non vérification des propriétés des relations inter-niveaux qui doivent être établies entre les modèles définissant les différents niveaux d'abstraction d'une architecture (donc par extension la non vérification des propriétés des relations inter-niveaux qui doivent être établies entre les composants appartenant aux différents modèles).

— Restauration de la cohérence (globale). A l'instar de la restauration de la cohérence locale, lorsqu'une incohérence globale est détectée, des modifications complémentaires sont réalisées pour rétablir les propriétés qui ne sont pas ou plus vérifiées (suppressions/ajouts de composants ou de connexions). Nous considérons ces modifications comme la propagation sur l'ensemble des niveaux d'abstraction des mo-

dications initiées sur un premier niveau d'abstraction afin de restaurer la cohérence globale de la définition de l'architecture.

Ce processus est outillé grâce à la formalisation du métamodèle de Dedal en langage B [4], résultant en un nouveau métamodèle que nous appelons FormalDedal. D'un point de vue pratique (voir la figure 2.14), nous avons intégré FormalDedal dans DedalStudio (voir la figure 2.11) grâce à des transformations de modèles depuis et vers le métamodèle de Dedal, défini en EMF. D'un point de vue conceptuel, nous ne considérons pas avoir implémenté un ADL formel mais simplement défini et implémenté une sémantique formelle pour notre ADL Dedal.

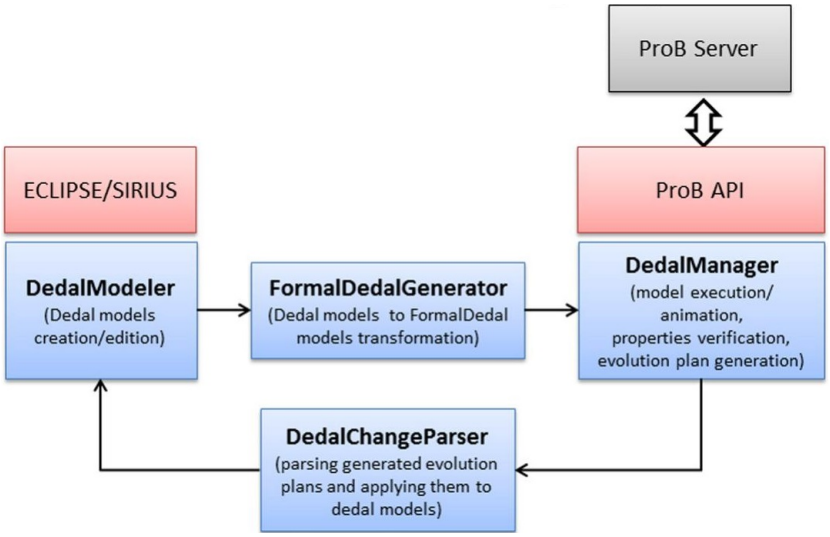


FIGURE 2.14 – Formalisation de l'ADL Dedal avec l'outillage du langage B

Disposer d'un métamodèle formel permet d'utiliser l'outillage du langage B pour faire automatiquement la vérification des propriétés de correction et de cohérence sur les définitions d'architectures produites avec Dedal.

Au-delà de ce premier niveau d'assistance, nous avons montré qu'il était possible d'utiliser un solver pour calculer automatiquement les modifications permettant de restaurer les propriétés de correction et de cohérence d'un modèle d'architecture (voir la figure 2.15).

Un modèle d'architecture Dedal peut être considéré comme une machines à états et les opérations de modification du modèle comme les règles de transition gérant l'évolution de l'état du modèle. Définissant les modifications initiales et le respect des propriétés de correction et de cohérence comme un objectif à atteindre, le solver explore automatiquement l'espace des états possibles afin de trouver un nouvel état qui restaure au besoin les propriétés de correction et de cohérence du modèle tout en conservant les modifications souhaitées. Nous avons utilisé avec succès les solvers génériques fournis par l'environnement ProB mais également implémenté notre propre solver, en utilisant les API de ProB, afin de pouvoir y intégrer des heuristiques et accélérer les calculs.

La séquence d'opérations de modification ainsi générée permet de proposer automatiquement un plan d'évolution. La figure 2.16 présente l'exemple d'un plan d'évolution généré automatiquement pour un modèle de spécification d'architecture. Ce plan d'évolution permet de restaurer les propriétés de la relation d'implémentation devant exister

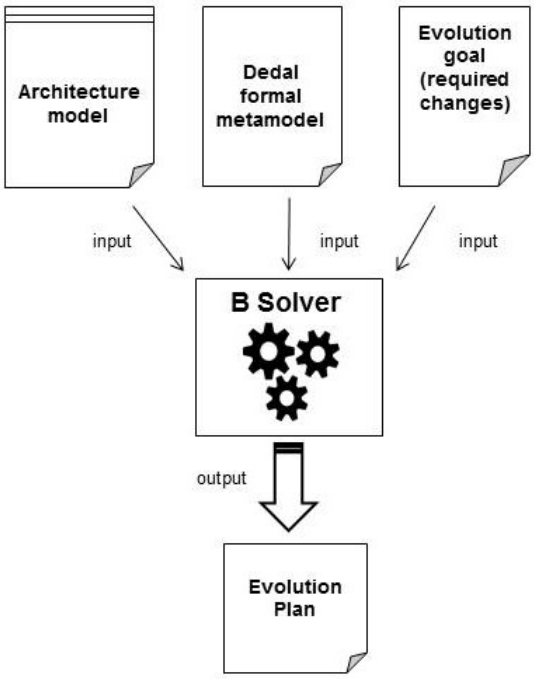
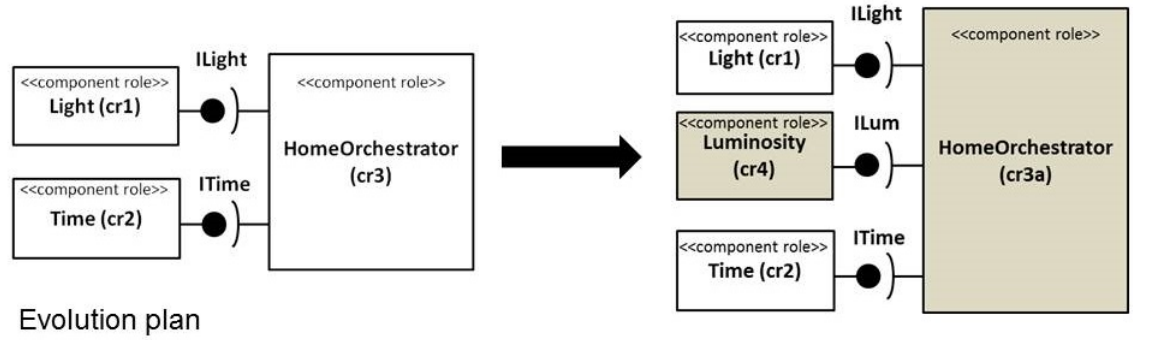


FIGURE 2.15 – Generation automatique de plans d'évolution à l'aide d'un solver B

entre le modèle de spécification et le modèle de configuration de l'architecture. Il consiste à déconnecter le rôle *HomeOrchestrator*, à le supprimer puis à ajouter et à connecter le rôle *Luminosity* ainsi qu'une variante du rôle *HomeOrchestrator*.



Evolution plan

$spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1, pintILight))$
$spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime))$
$spec\_deleteRole(HAS\_spec, cr3)$
$spec\_addRole(HAS\_spec, cr3a)$
$spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1, pintILight))$
$spec\_connect(HAS\_spec, (cr3a, rintITime), (cr2, pintITime))$
$spec\_addRole(HAS\_spec, cr4)$
$spec\_connect(HAS\_spec, (cr3a, rintIIntensity), (cr4, pintIIntensity))$

FIGURE 2.16 – Exemple de plan d'évolution généré par DedalStudio

Il s'agit d'un autre exemple, dans nos travaux, d'assistance fournie par une approche de search-based software engineering [69] (voir la section 2.2.1).

Ces mécanismes de co-évolution sont en partie détaillés par l'article présenté dans la section 4.2.1 et inclus dans l'annexe B.

2.2.3 Construction automatique d'annuaires ou de bibliothèques de composants et de services

La construction de bibliothèques (repositories) ou d'annuaires (registries) d'éléments réutilisables est indispensable pour outiller les processus de développement. Les annuaires sont utilisés à l'exécution et référencent les instances disponibles. Les bibliothèques par opposition contiennent les modèles ou les codes permettant la conception ou l'implémentation des applications. L'élément essentiel d'un annuaire ou d'une bibliothèque est sa capacité à structurer et indexer son contenu afin de supporter efficacement les requêtes de recherche d'éléments. Dans les processus de construction et d'évolution d'architectures logicielles que nous étudions [1], il s'agit de requêtes permettant de trouver des composants ou des services capables de jouer un rôle défini, donc de correspondre à un type. Nous avons ainsi étudié la construction automatique de hiérarchies de types de composants et de services, afin de proposer une solution pour structurer le contenu d'annuaires ou de bibliothèques [20][19][18][17][22][16][24][23].

Le type d'un composant ou d'un service peut être défini par différentes catégories d'informations. On distingue en général la classification sémantique et la classification comportementale des éléments.

La classification sémantique repose sur l'analyse lexicale de l'ensemble des éléments décrivant les composants et les services. Le vocabulaire utilisé permet d'associer chaque élément à un ensemble de concepts. Ces concepts permettent d'évaluer la distance sémantique entre les éléments afin de regrouper ceux développés pour des domaines et des fonctionnalités similaires. Nos travaux n'ont pas directement porté sur la classification sémantique car c'est une problématique très largement étudiée, bénéficiant des recherches menées dans le domaine du web sémantique [111]. Nous considérons la classification sémantique comme une première étape, permettant de découper thématiquement le contenu d'une bibliothèque ou d'un annuaire en un ensemble de domaines séparés, afin de réduire la complexité des autres étapes de classification. Pour cette étape, nous avons soit utilisé des approches et outils classiquement référencés dans la littérature (WorldNet [91], Latent Semantic Indexing [44]) soit considéré la classification sémantique comme donnée (en classifiant, par exemple, des fonctions de même nom).

Nos travaux ont plus spécifiquement porté sur la classification comportementale des composants et des services, telle que définie antérieurement pour les fonctions et les objets dans les travaux de Liskov [81][82]. Nous proposons cette deuxième étape de structuration du contenu d'un annuaire ou d'une bibliothèque afin de pouvoir rechercher des composants correspondant à un type de comportement spécifié, donc pouvant être intégrés à une architecture pour remplir un rôle précis. A l'instar des objets, la définition du comportement d'un composant peut comporter différents types d'informations. Classiquement, on distingue tout d'abord la définition dynamique et statique du comportement. La définition dynamique du comportement consiste à modéliser l'évolution du comportement d'un composant en fonction de son état, donc en fonction de l'historique des interactions du composant avec son environnement. Le comportement dynamique peut être ainsi défini par des machines à états [68][101] ou de manière duale par des pro-

1. Pour simplifier la lecture de la section, les orchestrations de services sont assimilées à une forme particulière d'architecture logicielle. En fonction du paradigme utilisé, c'est un abus de langage plus ou moins important qu'il faudra me pardonner.

tocoles de comportement [104]. La définition du comportement statique d'un composant repose sur la description de ses différentes fonctions. Une fonction est classiquement décrite par un contrat [90] spécifiant sa signature (son type syntaxique) et sa sémantique à l'aide d'un ensemble d'assertions (préconditions, postconditions et invariants exprimés par des expressions logiques).

De nombreux travaux ont également étudié la spécialisation des types de comportements dans les approches orientées objets et plus récemment dans les approches à bases de composants. Nous avons ainsi utilisé les travaux menés par [104] sur le modèle de composant SOFA pour définir la spécialisation des protocoles de comportements que nous utilisons dans le modèle de composants de Dedal.

Notre contribution originale porte sur la classification du type syntaxique des composants. Les travaux existants ne considèrent en effet que les interfaces fournies des composants. L'idée sous-jacente, implémentée par les annuaires fournis par certaines plateformes, tel que OSGI Registy [64], est que seules les interfaces fournies ont besoin d'être recherchées, donc classifiées et indexées, afin de satisfaire les dépendances exprimées par les interfaces requises des composants. Cette solution est effectivement suffisante pour établir des connexions et construire des architectures. Mais elle ne permet pas de réaliser, par exemple, des opérations de remplacement de composants qui nécessitent de tenir compte de toutes les interfaces connectées, requises ou fournies, donc directement du type syntaxique des composants.

De même, les modèles de composants que nous avons étudiés dans nos travaux proposent au mieux des mécanismes de typage ad hoc reposant sur des règles de bon sens, non formalisées, de spécialisation du type des interfaces (utilisant souvent celles du langage orienté objets sous-jacent). Ils négligent ainsi que les types des interfaces requises et fournies ne doivent pas suivre les mêmes règles de spécialisation pour respecter le principe de substituabilité entre types génériques et spécialisés. Ils ne proposent pas non plus de règles précises concernant la spécialisation des types de composants par ajout ou suppression d'interfaces. Il nous est donc apparu pertinent de formuler une proposition complète sur le typage syntaxique des composants [45].

Le typage syntaxique d'un composant suit le schéma général de typage syntaxique des structures complexes. Le type d'un composant résulte du type des interfaces (ou des ports) qui le définissent. De la même manière, le type d'une interface dépend de sa direction et du type des fonctions qui la définissent. Enfin, le type d'une fonction dépend du type et de la direction des paramètres qui composent sa signature. Il est ainsi possible de calculer le type syntaxique d'un composant à partir d'un ensemble de types de données élémentaires pour lesquels les règles de spécialisation sont connues (définies par le langage utilisé).

Considérons la signature d'une fonction associée à une interface fournie. Ses paramètres d'entrée définissent une pré-condition : l'appelant doit fournir des valeurs correspondant aux types des paramètres d'entrée pour permettre l'exécution de la fonction. De même, ses paramètres de sortie définissent une post-condition : l'exécution de la fonction produit en réponse un ensemble de valeurs correspondant aux types des paramètres de sortie. Le principe de spécialisation des signatures devant permettre de substituer les fonctions spécifiques aux fonctions génériques, les pré-conditions peuvent être assouplies (pour accepter au moins les mêmes données en entrée) et les post-conditions ren-

forcées (pour retourner des résultats au moins aussi riches). Il est ainsi possible (voir la figure 2.17) de remplacer le type d'un paramètre d'entrée par un type plus générique (contravariance du type des paramètres d'entrée) et inversement le type d'un paramètre de sortie par un type plus spécifique (covariance des paramètres de sortie). Nous permettons également la suppression de paramètres d'entrée et l'ajout de paramètres de sortie, en prenant comme hypothèse qu'il est possible d'ignorer un paramètre non utilisé. En fonction du contexte technique (cas des langages fortement typés), la génération d'un adaptateur peut être nécessaire pour permettre la substitution effective. Ce n'est qu'un point de variation, permettant d'étendre les possibilités de spécialisation, qu'il est possible d'adopter ou non en fonction des besoins et des contraintes. Il serait d'ailleurs intéressant d'étudier les possibilités d'adaptation automatique de signatures de fonctions pour proposer une palette d'options de paramétrisation des mécanismes de spécialisation.

La sémantique de la signature d'une fonction associée à une interface requise s'interprète différemment. Les paramètres d'entrée de la fonction représentent les données envoyées par le composant tandis que les paramètres de sortie représentent les données reçues par le composant. Pour assurer la compatibilité entre une signature de fonction requise générique et une signature de fonction requise spécialisée, il est possible de spécialiser le type des paramètres d'entrée (fournir au moins des données d'entrée aussi riches) et de généraliser le type des paramètres de sortie (pour accepter au moins les mêmes réponses). Autrement dit, la spécialisation des signatures des fonctions requises doit être covariantes pour les paramètres d'entrées et contravariante pour les paramètres de sortie (voir la figure 2.17). Suivant le même principe, il est possible d'ajouter des paramètres d'entrée et de supprimer des paramètres de sortie à la signature d'une fonction requise pour la spécialiser.

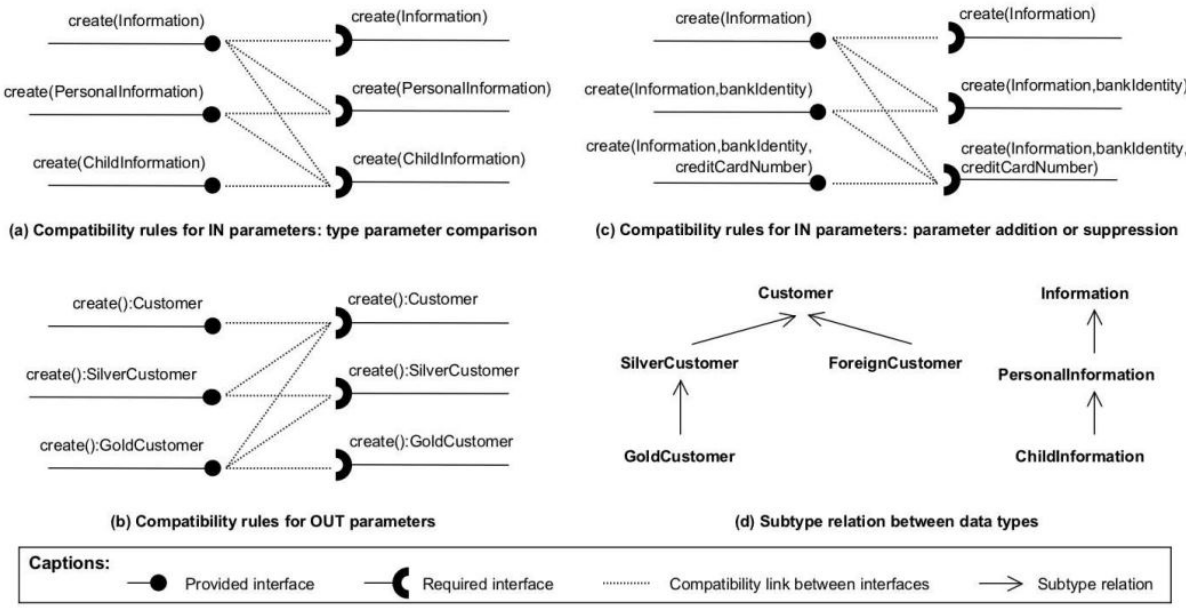


FIGURE 2.17 – Règles de spécialisation des types d'interfaces

Ces règles spécifiques de spécialisation des signatures des fonctions requises ne sont pas spécifiées, à notre connaissance, par les modèles de composants existants. Nombreux modèles utilisent en effet les mécanismes de typage du langage orienté-objet au dessus

duquel ils sont implémentés. Il est alors possible de typer les interfaces avec des types abstraits d'objets, qui sont similaire à des types d'interfaces fournies et de réaliser des opérations de vérification de compatibilité des types d'interfaces lors des opérations de connexion (le type syntaxique de l'interface fournie doit spécialiser le type syntaxique de l'interface requise). Mais il est par contre impossible de gérer une hiérarchie de spécialisation des types de fonctions requises, donc par extension une hiérarchie des types d'interfaces requises et donc une hiérarchie des types de composants.

En effet, les principes de spécialisation des interfaces reposent sur ceux des fonctions. Il est possible de spécialiser un type d'interface fournie ou requise en spécialisant le type d'une fonction. Un type d'interface fournie peut également être spécialisé par ajout de fonctions supplémentaires (fournir plus de fonctions à un composant client). De même, un type d'interface requise peut être spécialisé par suppression de fonctions (exiger moins de fonctions d'un composant serveur). Sur le même principe, un type de composant peut être spécialisé par spécialisation du type d'une interface, par ajout d'interfaces fournies ou suppression d'interfaces requises.

Nous avons utilisé ces règles de typage dans l'ensemble de nos travaux, pour formaliser les différentes relations (connexion, implémentation, spécialisation, instantiation, ...) entre les composants et entre les modèles d'architectures (voir la section 2.1.2) et plus directement encore dans la proposition d'une solution pour structurer le contenu d'annuaires ou de bibliothèques de composants. L'idée principale de ces travaux est que la hiérarchie de spécialisation soit construite automatiquement pour garantir l'optimalité de la structuration et par conséquence l'optimalité des requêtes de recherche de composants connectables ou substituables.

Nous avons défini un processus de structuration automatique [20][19][18][17], basé sur l'analyse formelle de concepts (FCA) [26][60]. FCA construit un treillis de concepts (une hiérarchie de types partiellement ordonnée par une relation de spécialisation) à partir d'un ensemble d'objets décrits par un ensemble d'attributs. Chaque concept est décrit par son intention (un ensemble d'attributs partagés par un ensemble d'objets) et par son extension (l'ensemble des objets possédant les attributs de d'intention). L'intention d'un concept générique est incluse dans l'intention des concepts qui le spécialisent (héritage des attributs). L'extension d'un concept inclut l'extension de tous les concepts qui le spécialisent. Le treillis de concepts est un treillis de Gallois : il est optimal en nombre (minimum) et en taille (maximale) des concepts proposés.

Notre processus construit dans une première phase (voir la figure 2.18) la hiérarchie des types de signatures de fonctions. Elle est utilisée pour construire, dans une deuxième phase, la hiérarchie des types des interfaces. Cette hiérarchie est à son tour utilisée pour construire la hiérarchie des types des composants. Ce processus a été implémenté et validé sur des composants extraits de codes source Java (outil Dicosoft, réalisé dans le cadre d'un TER de M1 et d'un stage de M2 recherche, à l'aide de la plateforme Eclipse et de l'outil de FCA eRCA [2]).

Réaliser une requête consiste alors à parcourir le treillis, depuis sa racine, pour classifier le type de composant défini par la requête (en comparant l'intention des concepts du treillis à l'ensemble des interfaces du type de composant recherché). Une fois classifié, l'extension du concept auquel correspond le type fournit la liste des composants de ce

type, donc des composants pouvant remplacer un composant ou se connecter à un ensemble d'interfaces donné (en utilisant dans la requête des types d'interfaces de direction opposée). Les éventuels sous-concepts permettent alors de naviguer parmi l'ensemble des composants compatibles et d'utiliser la classification fournie par le treillis pour affiner de manière itérative les critères de sélection.

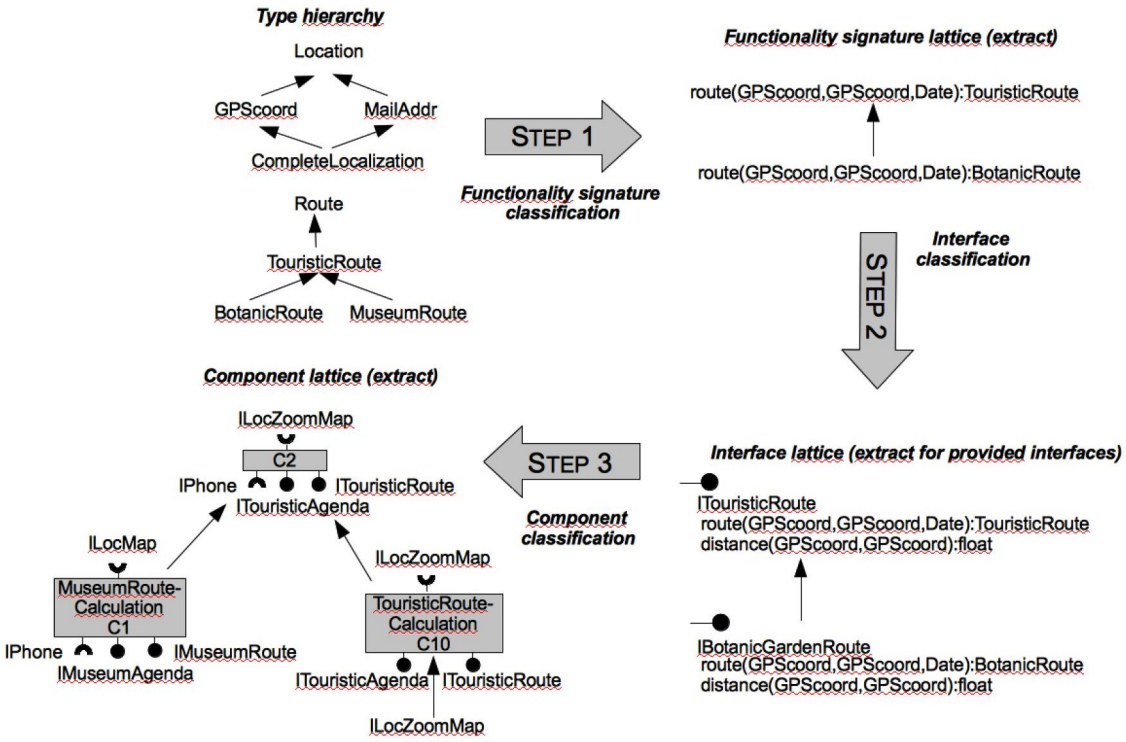


FIGURE 2.18 – Processus de construction d'une hiérarchie de types de composants avec FCA

Ce travail est en partie détaillé par l'article présenté dans la section 4.2.3 et inclus dans l'annexe D.

Une adaptation à l'indexation de services web documentés par des descripteurs WSDL a également été étudiée [22][16][24][23].

2.3 Adaptation autonomique d'applications sensibles au contexte

Nous avons également abordé la réutilisation sous l'angle des approches orientées services [3]. Dans ce paradigme, l'objectif n'est plus de construire l'architecture organique d'une application par la connexion d'un ensemble de composants. Il s'agit de réutiliser un ensemble de fonctionnalités, fournies par un ensemble de serveurs, par la définition de processus contrôlant l'invocation des fonctionnalités pour produire des fonctionnalités de plus haut niveau. En fonction de la logique d'exécution du processus, centralisée ou distribuée, ces processus sont appelés des orchestrations ou des chorégraphies de services. L'approche SCA (Service Component Architecture) [110] propose une convergence entre les paradigmes composants et services. Un service y est représenté par un composant possédant une interface fournie documentant le service. Une orchestration de service est encapsulée dans un composant exposant sous forme d'interfaces requises tous les services qui doivent être invoqués par l'orchestration. Il est alors possible de mettre en

œuvre une orchestration de services par la connexion d'un ensemble de composants.

Nous avons étudié la conception et l'implémentation d'architectures SCA pour contrôler des environnements intelligents, reposant sur l'utilisation des fonctionnalités fournies par un ensemble d'objets connectés. Nous avons adapté une partie de ces travaux, dans le contexte des webservices, pour proposer la gestion d'orchestrations BPEL génériques et de mécanismes de remplacement dynamiques de services [23][24][22].

Ces environnements intelligents ont pour caractéristique d'être dynamiques, ouverts et partagés. Ces environnements sont ainsi soumis à de nombreux changements : disponibilité des objets connectés et de leurs services (fiabilité et mobilité) ; évolution des besoins des utilisateurs (déploiement de nouvelles orchestrations ou modification des orchestrations existantes). Pour maintenir la continuité des services rendus aux utilisateurs, il est nécessaire que les applications (orchestrations) s'adaptent automatiquement aux changements qui surviennent dans leurs contextes d'exécution. Nous avons ainsi étudié les concepts de sensibilité au contexte [108] et d'adaptation autonomique [105].

Notre conception de ces mécanismes a été fortement influencée par les systèmes multi-agents [56][124], dont le paradigme repose sur la capacité des agents à percevoir leur environnement, à raisonner sur ces perceptions et à agir de manière autonome pour réaliser un ensemble d'objectifs. Notre idée est de concevoir des agents capables de percevoir les changements du contexte d'exécution et d'y réagir de manière autonome en adaptant les architectures logicielles afin de maintenir la continuité de leurs fonctionnalités.

Dans SAASHA [66][67][65], nous avons étudié une forte intégration des paradigmes agents et composants. Les agents sont définis par une architecture interne à base de composants (voir la figure 2.19). Parmi ces composants, on distingue premièrement les composants d'infrastructure (en gris clair) gérant le modèle d'agents. Le composant *Detector* gère la sensibilité au contexte et permet à l'agent de détecter automatiquement la présence d'objets intelligents et d'autres agents dans son environnement (par utilisation du protocole UPnP [71]). Les composants *Registry* gèrent les informations recueillies par l'agent sur son environnement (*AgentRegistry*, *DeviceRegistry*, *ScenarioRegistry*) mais également sur son architecture interne (*ComponentRegistry*). Les agents ont la capacité de modifier de manière réflexive leur architecture interne pour l'adapter aux changements de leur environnement (en utilisant le support du framework OSGi [64]). Ainsi, le composant *ComponentGenerator* a la capacité de générer des composants permettant de contrôler les objets intelligents détectés (composants en bleu moyen sur la figure 2.19). De même, il génère des composants coordinateurs (en bleu foncé sur la figure), chargés d'exécuter des orchestrations de services correspondant aux scénarios définis par les utilisateurs.

Ces scénarios sont définis sous la forme d'ensembles de règles ECA (Evenement, Condition, Action) diffusés aux agents contrôleurs et enregistrés dans leur composant *ScenarioRegistry*. Ces règles utilisent une typologies des services fournis par les objets connectés distinguant les services permettant de détecter des événements, d'obtenir une information ou de réaliser une action (voir la figure 2.20). Saasha propose ainsi un framework permettant de définir, déployer et exécuter des scénarios définis par les utilisateurs. L'exécution des scénarios peut être distribuée sur l'ensemble des agents présents dans l'environ-

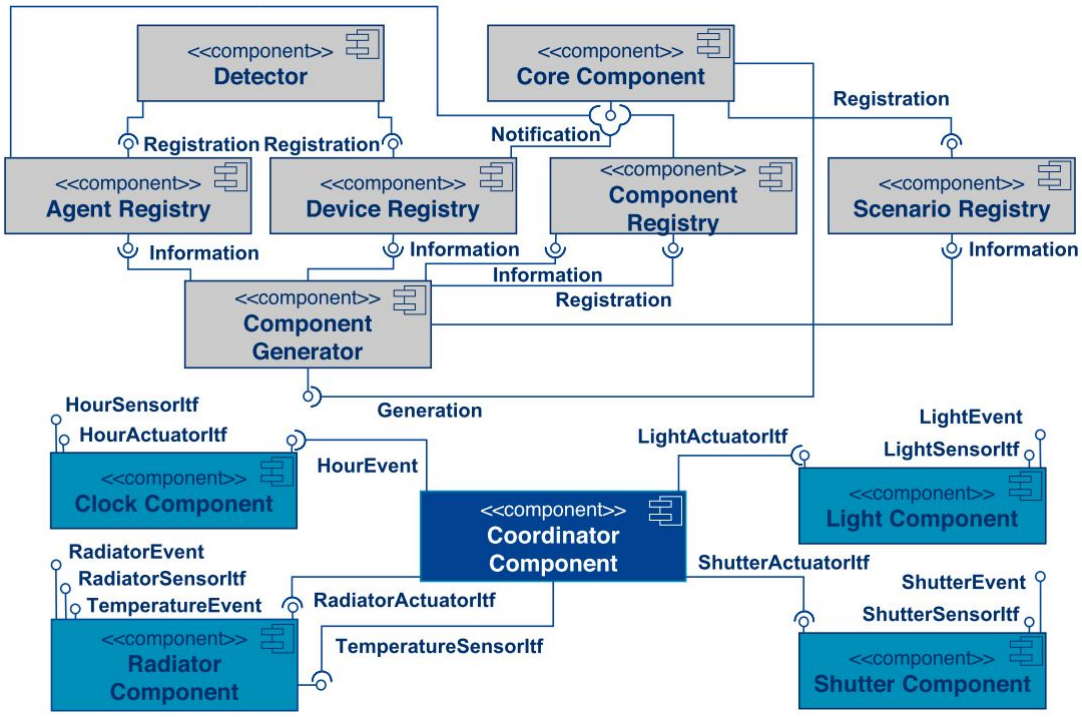


FIGURE 2.19 – Architecture interne des agents du framework Saasha

nement, par exemple en fonction de leur capacité à contrôler certains objets (localisation) ou de leur charge.

Nous avons prolongé ces travaux en travaillant sur le framework SAAS [54][55][53]. SAAS propose un langage d'orchestration, basé sur un ensemble classique de structures de contrôle, permettant de définir des scénarios plus élaborés (voir la figure 2.21). Ces orchestrations, encapsulées dans des composants orchestrateurs générés dynamiquement, peuvent être éventuellement réutilisées en tant que service à part entière. Ces orchestrations supportent directement la sensibilité au contexte en proposant des formes de quantification universelle ou existentielle dans la définition des invocations de services (invocation d'une instance quelconque d'un certain type de service ou de toutes les instances disponibles d'un certain type de service). On peut ainsi définir des scénarios du type "quand un service de type horloge indique qu'il est 19h, déclencher tous les services de type fermeture de volets disponibles" (la portée du scénario étant alors limitée par le réseau local sur lequel la détection des services est réalisée).

SAAS propose également un modèle de contrôle permettant l'exécution transactionnelle, pas-à-pas, des orchestrations. Ainsi, l'exécution des orchestrations peut-être suspendue en cas de problème d'invocation d'un service et reprise lorsque le service est disponible. SAAS supporte ainsi la mobilité des objets ou des utilisateurs mais également le partage des services et la distribution de l'exécution, autant d'éléments pouvant influencer la disponibilité des services invoqués. En fonction de la déclaration du service attendu, le service initialement associé à l'orchestration peut être remplacé par un service équivalent. En cas d'indisponibilité prolongée, dépassant une limite de temps fixée, l'exécution de l'orchestration est abandonnée et son échec est signalé.

Nous avons étendu ces mécanismes d'adaptation dans le système Arold [62]. Arold

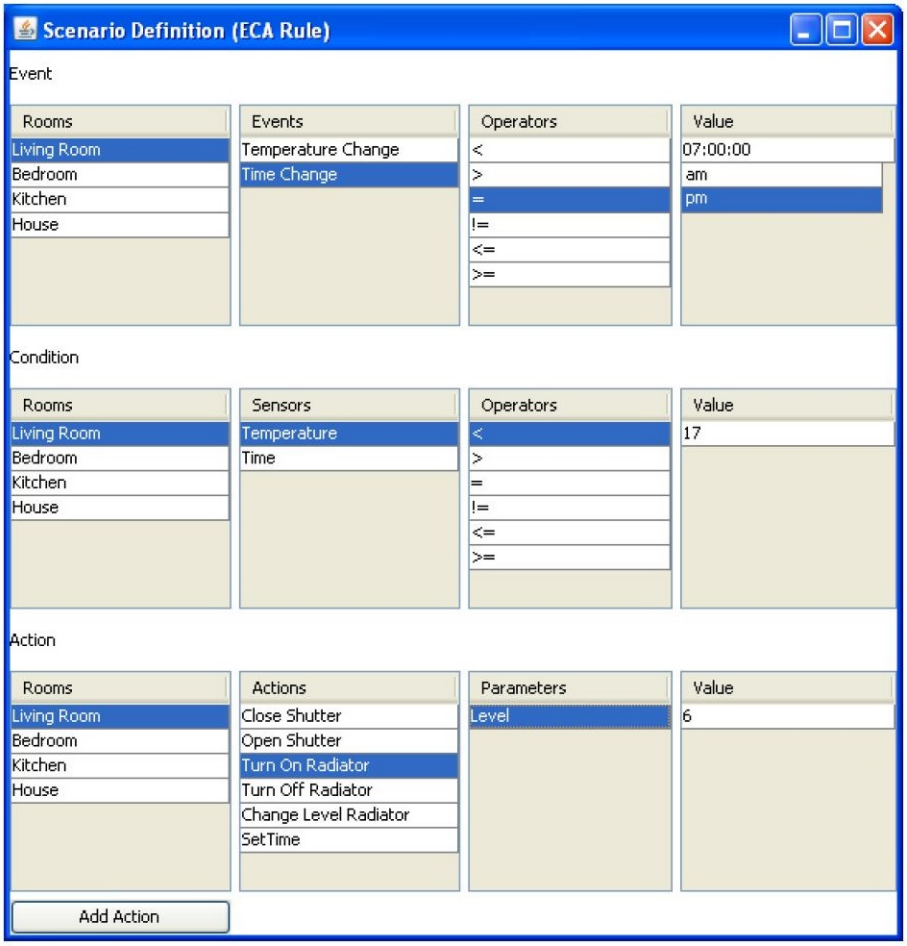


FIGURE 2.20 – Définition de scénario à l'aide de règles ECA dans Saasha

propose de définir le comportement du système à l'aide d'un langage de missions (voir la figure 2.22). Le comportement du système est structuré en un ensemble de modes. Chaque mode représente le comportement attendu du système pour gérer une situation donnée. Un ensemble de règles, déclenchées par des évènements et conditionnées par des gardes, gère les changements de modes. Ils constituent un premier mécanisme d'adaptation du comportement.

Chaque mode est défini par un ensemble de missions représentant les différentes activités que le système doit réaliser dans le mode courant. Les missions sont réparties en trois niveaux de priorité. En fonction des ressources disponibles, le système maintient de manière préférentielle les missions obligatoires (*mandatory*), importantes (*high*) puis les missions de routine (*normal*). Pour permettre un ajustement plus fin du déploiement des missions en fonction des ressources, chaque mission est décrite par un ensemble de stratégies. Une stratégie décrit non seulement une solution de déploiement de la mission, définie par comme un ensemble de tâches consommant des ressources matérielles, mais également l'efficacité de la solution, exprimée comme l'utilité du service rendu aux utilisateurs (en d'autres termes une mesure agrégée de qualité de service). Etant donné l'ensemble des missions à réaliser dans le mode courant, l'objectif du système Arold est de déployer l'ensemble des missions qui maximise l'utilité du système pour les utilisateurs, compte tenu de la priorité des missions du système et des ressources éventuellement limitées.

```
scenario NightScenario (description:close the main door and shutters,adjust
                  luminosity, change thermostat value and listen to radio;
                       creator: Antoine;
                       timeout: 4 hours) [
   MainDoor.DoorService.close();
   [ parallel:
      [
         all.ShutterService.close();
         while (LuminosityCaptor.LuminosityService.getValue() > ?) [
            RoomLight.DimmableLightService.decrease();
         ]
      ]
      if ( LivingRoomThermomether.ThermometerService.getTemperature() <= 18
           and HouseThermostat.ThermostatService.getValue() < 5 ) [
         HouseThermostat.ThermostatService.setValue(5);
      ]
      else [
         (creator=matt) any.AdjustTemperatureScenario.start(18);
      ]
   ]
   at (8pm) [
      PC.Player.play(WebRadio.RadioService.getChannel(InfoChannel));
   ]
]
```

FIGURE 2.21 – Le langage d'orchestration de services SAS SDL

Grâce au langage de missions, nous avons conçu le mécanisme de déploiement comme un problème d'optimisation combinatoire de type sac à dos (knapsack problem)[74]. Le mécanisme de déploiement est géré par un système multi-agents capable de calculer un (re)-déploiement des missions pour l'adapter aux changements du contexte d'exécution (disponibilité des agents et des ressources). Chaque agent est embarqué sur une plateforme matérielle et dispose d'un ensemble de composants lui permettant de contrôler les périphériques connectés à la plateforme (voir la figure 2.23). Par détection ou configuration, chaque agent dispose d'une liste des autres agents présents dans leur environnement. Lors de leur activation, les agents exécutent un protocole de formation de communauté et élisent pour leader l'agent disposant des meilleures ressources. Le leader collecte les informations sur les capacités (composants) et les ressources des autres agents de la communauté. Il calcule alors un déploiement optimisé des missions du système, à l'aide d'un algorithme glouton s'appuyant sur une estimation probabiliste de consommation de ressources. Ce mécanisme de déploiement, conçu de manière centralisés et gloutonne pour limiter le temps de calcul et la consommation de ressources, est adaptée aux systèmes embarqués et à une gestion dynamique de la reconfiguration du système en fonction des changements de contexte.

Une fois le calcul du déploiement réalisé, le leader indique à chaque agent de la communauté les différentes tâches qu'il doit exécuter pour participer à l'exécution des stratégies qui ont été sélectionnées. Le leader exécute alors une tâche de surveillance de la communauté et de l'exécution des missions. En cas de changement, le leader calcule un nouveau déploiement des missions. Ce redéploiement s'appuie éventuellement sur un modèle de tâches mobiles pouvant migrer entre les agents. Les agents de la communauté testent régulièrement la présence du leader. En cas de défaillance, le processus d'élection d'un nouveau leader est déclenché.

```
mode FloodAlert
{
  active mission SendFloodAlarms priority high
  active mission CollectFloodData priority normal
  active mission MonitorFloodAlert priority mandatory
}
mission CollectFloodData
{
  strategy CombinedFloodDataCollection utility 100
  strategy SingleFloodDataCollection utility 75
}
strategy SingleFloodDataCollection
{
  task WaterLevelCollector1
  task WaterLevelDatasource1
  connection WaterLevelCollector1.data=>WaterLevelDatasource1.store
}
strategy CombinedFloodDataCollection requires SingleFloodDataCollection
{
  task PressureCollector1
  task PressDatasource1
  connection PressureCollector1.data=>PressDatasource1.store
}
task WaterLevelCollector1
{
  type WaterLevelCollector
  resourceCost 10
  parameter sensorIP=192.168.1.30:100
  parameter rate=5mn
  parameter levelID=LeftBankRiverSite1
}
taskType WaterLevelCollector
{
implementation hydroguard.sensor.water.GenericWaterLevelSensor
parameter sensorIP : String
parameter rate : String
parameter levelID : String
out port data : WaterLevelDataSink
}
service WaterLevelDataSink
{
saveWaterLevel(levelID:String,timeStamp:Date,value:float)
}
```

FIGURE 2.22 – Exemple d'un descripteur de missions Arold

Arold propose ainsi différents mécanismes permettant aux agents de prendre en charge de manière autonome l'adaptation du comportement du système en fonction du contexte : localisation des périphériques, consommation des ressources matérielles (CPU, mémoire, disque, batterie, réseau, ...), communications réseau perturbées (apparition/disparition d'agents).

SAASHA et SAAS sont destinés à contrôler des objets connectés dans des environnements intelligents d'assistances aux personnes (Ambient Assisted Living). Arold est destiné à gérer un réseau de balises de surveillance de risques hydrologiques. Les implémentations prototypes de ces systèmes utilisent les technologies OSGi [64], pour la gestion des composants, et UPnP [71], pour la gestion des services. Nous nous sommes égale-

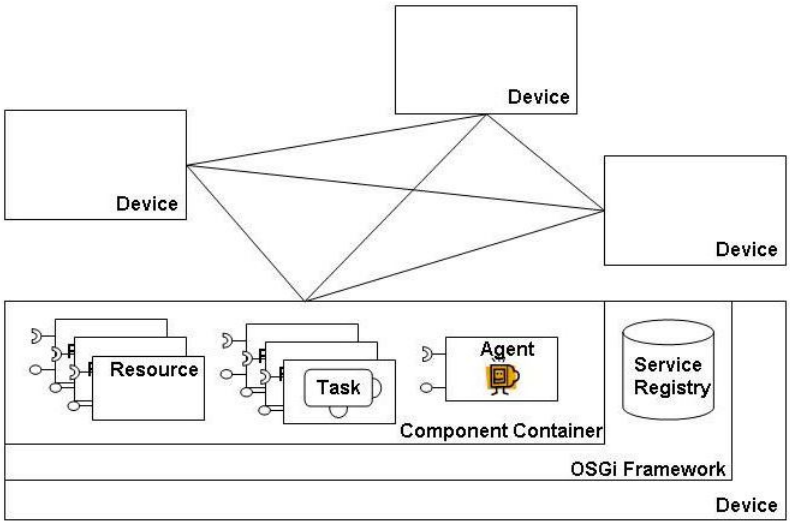


FIGURE 2.23 – Architecture du système Arold

ment appuyés sur les travaux que nous avons menés sur la fiabilisation des systèmes multi-agents, par la gestion des exceptions concurrentes dans les middlewares orientés messages [114][116][115][117][36][51] et les systèmes multi-agents répliqués [50].

Les travaux sur SAAS sont détaillés en partie par l'article présenté dans la section 4.2.5 et inclus dans l'annexe F.

2.4 Reverse engineering de lignes de produits logicielles à bases de composants

Nous avons également étudié la réutilisation sous l'angle de la rétro-ingénierie de codes sources orientés objets [6][7][9][10][8][13][12][11]. Ces travaux ont plus précisément porté la rétro-ingénierie de lignes de produits à partir des codes sources d'une famille de logiciels, implicitement obtenue, au fil du temps, par copie et modification.

Un premier objectif est l'identification et l'extraction de features, à partir d'éléments structurels similaires ou variables présents dans les codes sources. Nous avons combiné différentes techniques d'extraction d'information pour identifier les features (voir la figure 2.24). Une analyse formelle de concepts (FCA) [26][60](voir la section 2.2.3 pour une définition détaillée) est utilisée comme une première étape. FCA permet d'extraire automatiquement des blocs d'éléments structurels co-occurants, identifiés par les treillis de concepts construits à partir des éléments structurels présents dans les différents produits. On distingue ainsi les blocs de code communs, présents dans tous les produits, des blocs de code variables présents uniquement dans certains. Chaque bloc contient potentiellement l'implémentation d'un ou plusieurs features.

Une analyse du couplage structurel, basée sur les relations de dépendances entre éléments, et sémantique, basée sur une mesure de similarité lexicale (Latent Semantic Indexing) [44], est appliquée comme une seconde étape, pour procéder éventuellement au découpage des blocs en blocs élémentaires, dits atomiques. Chaque bloc atomique est alors considéré comme l'identification d'un feature.

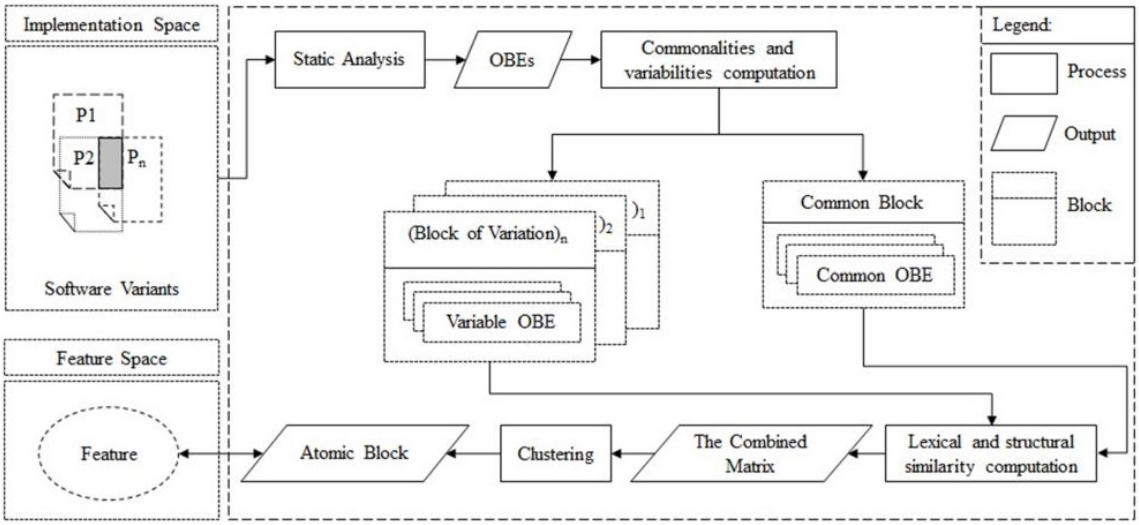
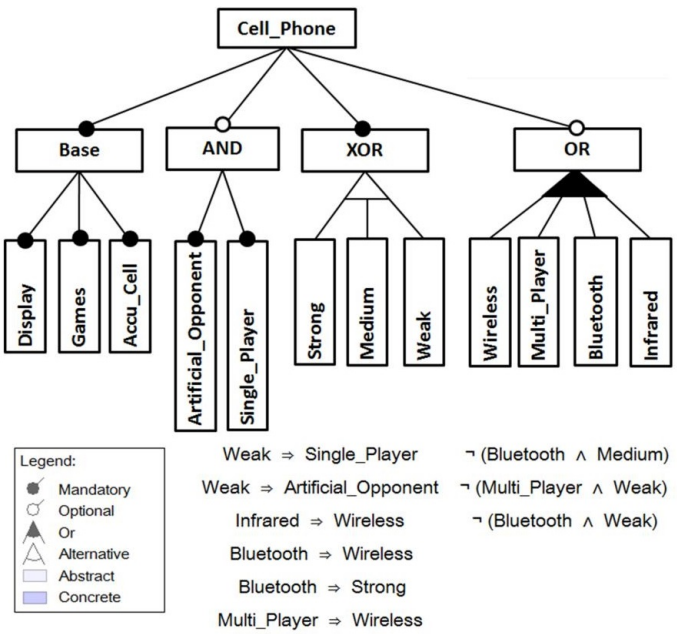


FIGURE 2.24 – Processus de rétro-ingénierie de features

Une FCA est de nouveau utilisée pour construire un treillis identifiant les combinaisons de features apparaissant dans la famille de produits. Nous avons proposé un algorithme transformant le treillis en un feature model [73] documentant la ligne de produit correspondante. La figure 2.25 présente le contexte binaire des features et des produits servant à la construction du treillis et le feature model correspondant, construit par notre algorithme de transformation. Le feature model propose une description structurée, sous forme d'arbre de décision, de l'ensemble des configurations de features, faisant apparaitre les dépendances, les variantes et les options possibles. Il permet de choisir une configuration de features cohérente et de sélectionner le produit correspondant.



	Cell_Phone	Wireless	Infrared	Bluetooth	Accu_Cell	Strong	Medium	Weak	Display	Games	Multi_Player	Single_Player	Artificial_Opponent
P-17	×				×	×			×	×			
P-18	×	×			×	×			×	×			
P-19	×	×	×		×	×			×	×			
P-20	×	×		×	×	×			×	×			
P-21	×	×	×	×	×	×			×	×			
P-22	×				×		×		×	×			
P-23	×	×			×		×		×	×			
P-24	×	×	×		×		×		×	×			
P-25	×	×			×	×			×	×	×		
P-26	×	×			×		×		×	×	×		
P-27	×	×			×	×			×	×		×	×
P-28	×	×			×		×		×	×		×	×
P-29	×	×			×			×	×	×		×	×
P-30	×	×			×	×			×	×	×	×	×
P-31	×	×			×		×		×	×	×	×	×

FIGURE 2.25 – Rétro-ingénierie d'un modèle de features à partir d'un treillis de features

Nous avons également étudié un processus de documentation automatique des features (voir la figure 2.26). Prenant en entrée un ensemble de features et un ensemble

d'éléments de documentation de la famille de produits (dans le cas de nos travaux des cas d'utilisation), nous utilisons une variante de FCA, appelée RCA [70], permettant de classifier des objets non seulement en fonction des attributs qui les définissent mais également de leurs relations. RCA nous permet de produire des regroupements de features et de cas d'utilisation, appelés blocs hybrides, en fonction de leurs co-occurrences dans les produits. Nous utilisons ensuite une mesure de similarité lexicale (toujours basée sur le Latent Semantic Indexing [44]) sur les différentes blocs produits pour associer les features avec les cas d'utilisation en fonction de leur proximité sémantique. On peut ainsi considérer que RCA est utilisée pour réduire la complexité des analyses lexicales en permettant un découpage de l'espace des documents à classifier (code des features et documentation des cas d'utilisation). Après seuillage, l'analyse lexicale produit une matrice binaire d'association entre les features et les cas d'utilisation. FCA est appliqué à cette matrice pour contruire un treillis de concepts associant les features aux cas d'utilisation.

Il est alors possible d'extraire des informations des cas d'utilisation pour documenter les features. Par exemple, la génération des noms des features à partir des noms des cas d'utilisation produit de bons résultats.

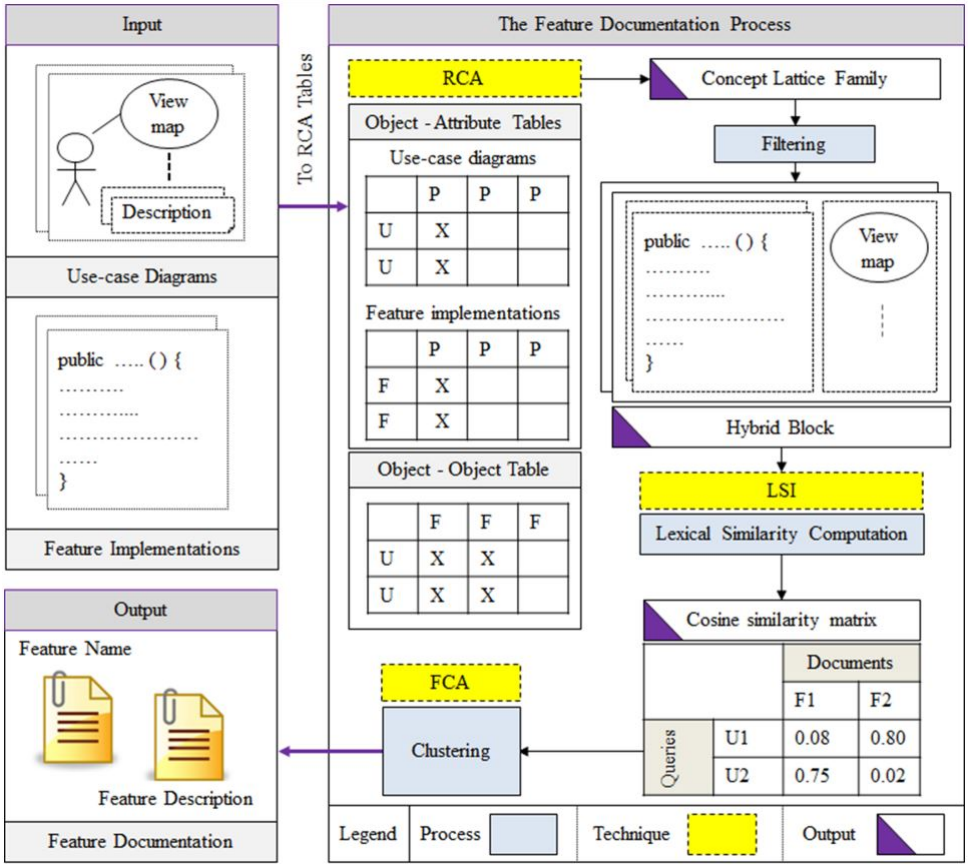


FIGURE 2.26 – Processus de rétro-ingénierie de documentation de features

Ces différents outils de rétro-ingénierie ont été implémentés dans le prototype REVPLINE et validés par un ensemble de cas d'études utilisant des projets opensource.

Ce travail est détaillé en partie par l'article présenté dans la section 4.2.2 et inclus dans l'annexe C.

Chapitre 3

Projet de recherche

« *On est toujours content quand les gens qui nous aiment relèvent nos travers comme des raisons supplémentaires de nous aimer.* »

Emmanuel Carrère - D'autres vies que la mienne (2009)

3.1 Automatisation du développement d'applications

3.1.1 Convergence entre modèles de features et ADLs

Les approches d'ingénierie de lignes de produits logiciels ont défini les bases conceptuelles des processus de développement basés sur la réutilisation. Elles distinguent deux processus : le processus d'ingénierie de domaine (appelé dans les travaux antérieurs processus de développement pour la réutilisation), visant à produire des bibliothèques d'artefacts de développement réutilisables ; le processus d'ingénierie des applications (appelé dans les travaux antérieurs processus de développement pour la réutilisation), visant à développer des applications par la réutilisation d'artefacts de développement. Nos travaux sur l'ADL Dedal, les bibliothèques de composants et les mécanismes de construction/évolution d'architectures, couvrent actuellement les étapes de conception, réalisation et déploiement de ces processus.

Une perspective est de couvrir les étapes amont du processus de développement. L'objectif serait d'assister et d'automatiser au maximum la réalisation d'applications sur la base de l'expression des besoins utilisateurs. Il réclame deux étapes initiales : traduire les besoins utilisateurs en exigences fonctionnelles puis identifier dans les bibliothèques disponibles les éléments (architectures, composants) pouvant être réutilisés pour réaliser une application conforme aux exigences.

Nous envisageons ainsi d'étudier comment indexer les bibliothèques d'architectures et de composants en combinant modèles de features et ADLs. Les ADLs, comme Dedal, fournissent des concepts permettant de bâtir une classification de composants et d'architectures basée sur leurs types. Les modèles de features fourniraient des concepts permettant une classification sémantique des composants et des architectures, basée sur leur capacité à remplir une exigence fonctionnelle donnée. Un ensemble de features, correspondant aux exigences fonctionnelles d'une application, permettrait alors de rechercher toutes les solutions de réalisation répertoriées dans les bibliothèque d'architectures disponibles. Ces mécanismes auraient deux applications possibles : permettre aux utilisateurs finaux de personnaliser des environnements (par exemple leurs installations domotiques) en construisant sans programmation des applications par la sélection de features (user empowerment) ; assister les professionnels dans la production de logiciels de qualité en systématisant la réutilisation d'architectures conçues et implémentées suivant les meilleures pratiques du génie logiciel (styles architecturaux, patrons de conception, patrons d'analyse,...).

Une solution serait non pas uniquement d'outiller un mapping entre features et architectures mais d'intégrer les concepts des modèles de features et des ADLs au sein d'un même langage afin de pouvoir produire des descriptions d'architectures adaptées à toutes les étapes du cycle de développement, y compris les plus amont (ingénierie des exigences).

3.1.2 Généralisation de la réutilisation d'architectures

Les mécanismes de réutilisation reposent sur les relations pouvant être établies entre les architectures. Une perspective consiste à étendre les types de relations de notre ADL Dedal. Les relations existantes ont été conçues pour permettre de réutiliser une spécification comme modèle de différentes configurations (relation d'implémentation) et une configuration comme modèle de différents déploiements (relation d'instanciation). L'étude de relations de spécialisation, de composition et de versionnement entre architectures permettra de réutiliser une architecture ou un ensemble d'architectures pour définir de

nouvelles architectures par spécialisation, composition ou dérivation. Comme abordé dans nos travaux précédents, la sémantique de ces relations sera définie dans un langage formel afin de pouvoir vérifier la validité des relations, de diagnostiquer les incohérences et de suggérer automatiquement, dans une approche de search-based software engineering, des modifications permettant d'établir ou de maintenir les relations souhaitées. En particulier, l'étude des relations de spécialisation entre architectures permettra de décrire des modèles d'architectures plus abstraits et génériques que les spécifications d'architectures actuellement proposées en Dedal. Ces architectures abstraites pourront documenter des patrons d'architectures (patrons d'analyse et patrons de conception) ou des styles architecturaux venant guider les architectes dans la conception d'architectures concrètes.

Nous pourrions par ailleurs étudier l'utilisation de ces relations pour structurer le contenu de bibliothèques d'architectures qui seraient l'outil central d'une ingénierie basée sur la réutilisation. Ces bibliothèques d'architectures viendraient étendre et compléter nos travaux sur la structuration des bibliothèques de composants et en reprendraient certains principes, comme la construction automatique de typologies d'architectures à l'aide d'analyses formelles de concepts. L'objectif de cette structuration serait de pouvoir supporter toutes les requêtes utiles au processus de développement : à haut niveau, par exemple, identifier les types abstraits d'architectures correspondant à un ensemble d'exigences fonctionnelles, besoins ou features ; à plus bas niveau, par exemple, identifier toutes les implémentations répertoriées d'un type d'architecture ou tous les raffinements possibles d'une implémentation donnée.

3.1.3 Retro-ingénierie d'architectures logicielles

Une perspective connexe est de développer la rétro-ingénierie d'architectures. La réutilisation est désormais massivement présente lors de l'étape de réalisation de la plupart des applications, par la mise en œuvre de frameworks ou de SDKs. Elle s'est également développée grâce aux approches orientées services et plus récemment au cloud computing (SaaS, PaaS et IaaS). Une grande partie du code des applications gère l'instanciation et la connexion d'éléments réutilisables, c'est-à-dire la définition d'architectures. Ces architectures sont encore dans de nombreux cas non explicitement documentées ou gérées à l'aide d'outils de bas niveau (descripteurs de déploiement), comme dans les frameworks Spring, OSGi, Maven, … .

La rétro-ingénierie de ces architectures vers un ADL permettrait d'adopter une démarche outillée d'ingénierie dirigée par les modèles pour gérer l'évolution de ces applications. Elle permettrait également d'alimenter des bibliothèques permettant la réutilisation d'architectures, par exemple dans une approche d'ingénierie de ligne de produits. La rétro-ingénierie d'architectures aurait également un intérêt en termes de recherche, la communauté « architectures » souffrant de l'absence de bases d'exemples pour valider ses résultats. La mise à disposition de bibliothèques d'architectures de référence, issus de projets réels, ouvrirait des perspectives d'extraction de patrons de conception et de styles architecturaux, de validation de métriques, de tests de passage à l'échelle et de mesures significatives de rappel et de précision pour l'extraction d'informations.

3.1.4 Généralisation des concepts architecturaux

Il serait intéressant d'étudier la généralisation des concepts de composant et d'architecture que nous utilisons dans notre ADL Dedal de manière à étendre son l'utilisation.

En effet, le principe de modélisation de systèmes à l'aide d'architectures à base de composants, est très versatile. Nous voudrions ainsi étudier les ADLs comme des outils génériques d'ingénierie, servant de base au développement d'outils spécifiques par intégration d'un DSL (Domain Specific Language) [59].

Le modèle de composants actuellement proposé par notre ADL Dedal a été conçu pour développer des logiciels, donc pour produire des architectures conceptuelles basées sur la définition de fonctions réparties sur des composants ayant des interactions de type client-serveur. Il n'est pas adapté, par exemple, à la définition des éléments physiques (fichiers de code, fichiers de ressources, serveurs, machines virtuelles, processeurs, ...) décrivant le déploiement d'un logiciel dans un environnement d'exécution concret. A une autre échelle, l'administration de logiciels ou de plateformes logicielles dans des environnements de (multi-)cloud computing [102] ou de edge computing [112] peut également être gérée à l'aide d'architectures à base de composants. Dans d'autres domaines, tels que les systèmes cyber-physiques [75] ou les systèmes de systèmes, la modélisation d'architectures fournit un support d'ingénierie pertinent.

Nous pourrions ainsi identifier et d'extraire le "noyau" conceptuel de Dedal pour proposer un outillage générique de construction, de réutilisation et d'évolution de descriptions architecturales, qui serait ensuite raffiné ou étendu pour supporter des activités d'ingénierie spécifiques.

3.2 Génie logiciel agile

3.2.1 Rétro-ingénierie de lignes de produits par exploitation de dépôts de code

La généralisation de l'adoption systèmes de gestion de version est un bon exemple de génie logiciel agile (pragmatique) : avec la possibilité de centraliser et de tracer les contributions de l'ensemble des développeurs d'un projet, de fusionner ces contributions en détectant d'éventuels conflits de cohérence, les systèmes de gestion de version ont permis d'élaborer des stratégies de développement collaboratifs adaptées à différentes tailles et types de projets, essentiellement sous forme de règles de bonne pratique (et de gestion de droits accordés aux modérateurs des projets) [37].

Aussi pertinentes que soient ces pratiques, elles pourraient bénéficier d'un support plus efficace si les relations de dérivation liant les versions successives avaient une sémantique bien définie, indiquant l'intention des modifications apportées. Nous voudrions ainsi distinguer les modifications architecturales qui maintiennent la compatibilité ascendante ou l'iso-fonctionnalité des modifications qui introduisent des ruptures pour confronter alors ces modifications à l'intention de créer une révision (une nouvelle version ayant pour vocation à remplacer les versions précédantes) ou une variante (une version supplémentaire adaptée à des besoins différents). Il se serait alors possible de proposer des outils d'assistance, vérifiant le respect des politiques de gestion des branches du dépôt ou suggérant des opérations de reconstruction des historiques pour s'y conformer.

En outre, la possibilité de dériver de nouvelles branches ou des dépôts entiers (forks) favorise la pratique du "clone and own" [57], c'est-à-dire le développement pragmatique d'une variante d'un logiciel par copie et modification. La généralisation de cette pratique engendre une perte d'efficacité qui n'est pas nécessairement perçue : des développements redondants sont réalisés ; des versions obsolètes de fonctionnalités ou de modules sont utilisés par manque de visibilité sur le contenu des différentes variantes ; de nouvelles configurations sont créées manuellement, par découpage et intégration manuelle

de parties de différentes variantes, au prix d'un effort d'adaptation et de débuggage importants.

Loin d'imposer des contraintes sur l'utilisation des dépôts de code, étudier des outils de rétro-ingénierie de lignes de produits logiciels à partir des historiques de versionnement permettrait d'identifier de manière agile les features, les configurations, ainsi que leurs révisions et leurs variantes pour construire des vues abstraites du contenu du dépôt, adaptées à des prises de décisions managériales ou architecturales.

3.2.2 Elaboration de métriques pour assister la prise de décisions architecturales

L'existence de vastes dépôts de projets open source, tels que GitHub et BitBucket, a permis le développement d'une approche du génie logiciel appelée « génie logiciel empirique » [27].

Cette approche consiste à exploiter les grandes masses d'informations disponibles dans les dépôts, en utilisant des techniques de data mining, pour découvrir ou valider des indicateurs (métriques) permettant d'estimer ou de prédire la qualité d'un logiciel ou du management d'un processus développement.

L'amélioration de la qualité des logiciels doit permettre à court terme de diminuer la quantité de bugs produits lors des développements et de faciliter leur correction. A plus long terme, une meilleure qualité doit faciliter l'évolution des logiciels. Il s'agit dans tous les cas de diminuer les coûts de développement en limitant la « dette technique » [58], c'est-à-dire les surcoûts engendrés par une mauvaise gestion de la qualité.

A l'image des outils de business intelligence destinés aux managers, les mesures de qualités alimenteraient des tableaux de bord permettant aux architectes (product owner, DSI, modérateur,...) de détecter les situations problématiques et d'être assisté dans la prise de décisions améliorant la qualité des logiciels.

Conforme à une approche d'ingénierie dirigée par les modèles, nous souhaitons déterminer l'impact sur la dette technique des décisions architecturales prises lors des étapes préliminaires de conception du cycle de développement, c'est-à-dire des choix d'organisation de la structure macroscopique d'un logiciel, définie par son découpage en modules.

Notre démarche consisterait d'une part à extraire, par rétro-ingénierie, une représentation de l'architecture d'un logiciel, pour chaque version sauvegardée dans un dépôt, de manière à suivre son évolution et identifier les décisions architecturales prises (modifications réalisées). Nous essaieront de classifier ces décisions architecturales pour élaborer un catalogue de patrons de décisions.

D'autre part, nous étudierons les corrélations entre métriques de qualité appliquées aux architectures et dette technique en exploitant les informations contenues dans les tickets (messages signalant toutes les tâches de développement à réaliser) stockés par les dépôts. Au besoin, nous élaborerons de nouvelles métriques de qualité pour mettre en évidence les corrélations potentielles entre caractéristiques des architectures et bonne maîtrise du développement (diminution du nombre de tickets ou diminution des coûts de résolution des tickets).

Nous espérons ainsi construire des indicateurs fournissant aux architectes une assistance permettant de détecter les situations nécessitant des prises de décisions architecturales (ré-ingénierie préventive ou corrective) et de choisir les meilleurs décisions pour limiter, à court et à plus long terme, la dette technique.

3.2.3 Convergence programmation/conception

Une perspective plus large est de faciliter la mise en œuvre des bonnes pratiques du génie logiciel par l'industrie. Mes contacts avec le secteur des ESN, principal employeur de nos étudiants ingénieurs, m'obligent à dresser un bilan très mitigé. Les ESNs et leurs clients raisonnent à très court terme, l'essentiel étant d'obtenir un logiciel fonctionnel dans les délais les plus courts et au moindre coût. Dans ce contexte, le génie logiciel est perçu comme une charge supplémentaire et non comme une opportunité d'économiser des ressources en évitant de cumuler une dette technique qui rend, à plus ou moins long terme, beaucoup plus coûteuses la maintenance et l'évolution des logiciels.

L'Ingénierie Dirigée par les Modèles fournit des solutions à cette problématique, en augmentant la proportion de code généré, donc mettant systématiquement en œuvre les meilleurs principes du génie logiciel. Mais les développeurs perçoivent la gestion conjointe des modèles et des codes sources comme une difficulté, une charge supplémentaire. Certains travaux ont cherché à répondre à cette question en proposant de remplacer les codes sources par des modèles exécutables. Mais il n'existe toujours pas, à notre connaissance, de proposition d'un langage de modélisation dont les concepts, la syntaxe et la sémantique opérationnelle constituent une alternative pratique aux langages de programmation. Autant les syntaxes graphiques sont pertinentes pour représenter les architectures abstraites, macroscopiques, autant elles deviennent lourdes à manipuler et complexes à interpréter quand il s'agit de définir de manière détaillées des fonctionnalités.

Les travaux de l'OMG sur fUML [109] (sous ensemble d'UML doté d'une sémantique opérationnelle permettant l'exécutabilité des modèles) illustrent parfaitement cette problématique. La syntaxe graphique permettant de définir l'implémentation des fonctionnalités, sous la forme de diagrammes d'activités exécutables, est redoutable de complexité. Aussi l'OMG propose également Alf [109], une syntaxe textuelle permettant de définir les activités sous forme de codes source facilement compréhensibles pour les praticiens car proches de C++. Cependant, il est peu probable que les bénéfices de ce langage (alignement sur les concepts d'UML) compensent suffisamment l'effort d'apprentissage d'un nouveau langage et d'adaptation des codes existants pour susciter une adoption massive.

Une idée alternative serait donc de doter les langages de programmation existants d'une syntaxe graphique, permettant de représenter la structure et le comportement des programmes à un niveau d'abstraction similaire à celui pratiqué dans les langages de conception. Il n'y aurait ainsi plus de dichotomie entre modèle conceptuel et programme (modèle exécutable), donc entre l'activité de conception et de programmation, puisqu'il s'agirait de pratiquer un même langage au travers de différentes syntaxes, adaptées aux différentes étapes du processus de développement.

Cette perspective est proche de celle de rétro-ingénierie d'architectures à partir de codes sources, puisqu'il s'agit là aussi d'identifier l'information décrivant le niveau architectural des logiciels et de la représenter explicitement avec une syntaxe (graphique) adaptée à ce niveau d'abstraction.

3.2.4 Ingénierie des exigences par prototypage des UI/UX

Dans un même esprit pragmatique, de co-conception des logiciels avec les clients/utilisateurs préconisée dans les démarches agiles, il serait intéressant d'étudier l'ingénierie des besoins/exigences fonctionnelles sous forme de maquettes d'interfaces graphiques et de scénarios d'utilisation (ce que permettent le langage Interaction Flow Modeling Language de l'OMG [32] ou des outils basés sur les processus métiers tels que Bonita BPM de Bonitasoft [28]). L'objectif serait là aussi de faciliter les activités d'ingénierie en brisant les

frontières entre modèle, programme et prototype.

Une solution consisterait en un langage de conception ayant une sémantique opérationnelle permettant au minimum d'animer la maquette (exécution des scénarios utilisateurs), de générer des squelettes d'implémentation de la logique métier de l'application ou des tests système. Ce langage serait par ailleurs conçu pour supporter différentes syntaxes graphiques permettant, pour une même logique métier sous-jacente et une même définition générique des interactions avec l'utilisateur, d'adapter les interfaces graphiques aux plateformes de développement choisies ou à des chartes graphiques spécifiques, afin que le modèle puisse être aussi proche que souhaité, visuellement, de l'application finale.

L'objectif serait ainsi de pouvoir réaliser l'ingénierie des besoins/exigences fonctionnelles par prototypage des interfaces utilisateurs et de la logique métier de l'application, avec la possibilité de réutiliser la logique métier pour pour concevoir des applications fonctionnellement proches ou pour faire évoluer une application existante.

Chapitre 4

Conclusion

« *La psychologie du complot nait de ce que les explications les plus évidentes de faits préoccupants ne nous satisfont pas car cela fait mal de les accepter.* »

Umberto Eco - Chroniques d'une société liquide (2016)

4.1 Synthèse et discussion

La question récurrente, lorsque l'on décide de passer son HDR, et plus encore lorsque l'on se décide tardivement, est de trouver un fil directeur permettant de synthétiser les différents travaux réalisés. Umberto Eco disait qu'un écrivain écrit toujours le même livre. Mais qu'il avait la chance de ne pas savoir lequel. En travaillant à la rédaction de cette HDR, j'ai eu la surprise de constater que mon travail de recherche, qui me semblait avoir exploré au fil des collaborations et des thèses des thèmes très divers, visait finalement à étudier une problématique abordée dès ma thèse [123] : le développement par la réutilisation de composants logiciels.

L'introduction proposée dans le chapitre 1 résumait cette problématique en énonçant cinq questions de recherche générales. Il est intéressant de les confronter, en conclusion, aux contributions (voir le chapitre 2) et aux perspectives (voir le chapitre 3) que nous avons présentées dans ce manuscrit.

4.1.1 Q1 - Quels artefacts réutilisables doit-on produire pour faciliter les développements ?

Notre proposition d'un modèle de ports composites (cf. section 2.1.1) améliore la sélection puis la mise en œuvre des composants. Ce travail est détaillé par l'article présenté dans la section 4.2.4 et inclus dans l'annexe E.

Les différents niveaux d'abstraction de notre ADL Dedal (cf. section 2.1.2) ont été conçus pour permettre la réutilisation des modèles d'architectures lors des transitions entre les différentes phases du cycle de développement. Dedal est détaillé dans l'article présenté dans la section 4.2.1 et inclus dans l'annexe B.

Nos travaux ont ainsi essentiellement porté, jusqu'à présent, sur des artefacts réutilisables facilitant la conception, l'implémentation et le déploiement des applications. Pour étendre notre couverture du processus de développement aux étapes d'ingénierie des besoins et des exigences, nous proposons d'étudier la complémentarité pouvant exister entre les ADLs et les modèles de features, qui pourraient être considérés comme des représentations abstraites, purement fonctionnelles, d'architectures (cf. section 3.1.1).

De manière connexe, nous envisageons d'étudier la généralisation de nos principes de réutilisation d'architectures (cf. section 3.1.2), qui reposent jusqu'à présent sur un schéma rigide, à trois niveaux. L'idée principale serait d'étudier une relation de spécialisation permettant de créer des hiérarchies de types d'architectures, depuis des modèles abstraits, décrivant des patrons d'architectures, jusqu'à des modèles spécifiques, décrivant des implémentations concrètes.

Une idée complémentaire est d'étudier la généralisation du concept de composant pour étendre ses utilisations et par extension celles des architectures (cf. section 3.1.4). En effet, nous avons travaillé jusqu'à présent avec un unique modèle de composants, adapté aux étapes de conception et d'implémentation de logiciels. Mais il serait nécessaire, pour couvrir d'autres étapes du cycle de développement, de représenter d'autres types de composants, tels que les éléments physiques (fichiers, processus, matériels), qui constituent l'environnement de déploiement et d'exécution, donc de proposer d'autres modèles de composants adaptés. Par ailleurs, les concepts de composants et d'architectures sont des concepts de modélisation génériques, applicables dans de nombreux domaines d'ingénierie. Il serait donc intéressant et complémentaire d'étudier un modèle générique de composants et d'architectures, qui servirait de noyau conceptuel au développement de

langages métier pour des domaines d'ingénierie spécifiques.

Sur un autre plan, la perspective de réaliser l'ingénierie des besoins/exigences par le maquettage d'interfaces et l'animation de scénarios utilisateurs (cf. section 3.2.4) vise à pouvoir réutiliser directement les éléments produits par l'analyse des besoins/exigences soit comme un prototype servant de base au développement soit comme un ensemble de tests systèmes permettant de vérifier la validité du logiciel vis-à-vis de son cahier des charges.

4.1.2 Q2 - Comment identifier et produire des artefacts réutilisables par rétro-ingénierie de logiciels existants ?

Nos travaux sur la rétro-ingénierie de features à partir du code source d'une famille d'applications étudient cette question (cf. section 2.4). La démarche que nous proposons permet d'identifier les éléments communs et variables présents dans les différentes applications, à l'aide d'une analyse structurelle et d'une analyse lexicale, puis de construire un modèle de features représentant les configurations présentes dans la famille, par extraction des contraintes d'exclusion et d'implication observées. Ce travail est détaillé par l'article présenté dans la section 4.2.2 et inclus dans l'annexe C.

Si ce travail permet d'identifier des features et leurs implémentations dans le code source d'applications, il n'étudie aucune solution permettant la réutilisation pratique et concrète des features, telle que la transformation de leurs implémentations en composants logiciels.

Nous souhaitons, de manière complémentaire, étudier la rétro-ingénierie d'architectures à base de composants à partir des codes sources des applications (cf. section 3.1.3). Notre idée est de procéder à une identification de l'architecture existante, en nous appuyant sur les concepts architecturaux génériques (modularité, encapsulation, découplage, abstraction) présents dans le paradigme de programmation utilisé. Des mécanismes complémentaires seront alors étudiés pour faire la réingénierie de l'architecture initialement extraite (analyse lexicale, métriques, classification, ...) et produire des composants et des architectures plus pertinents et de meilleure qualité. Il faudra vérifier, par l'expérimentation, que l'introduction d'un modèle d'architecture à base de composants, dès les premières étapes de la rétro-ingénierie, facilite et améliore le processus de rétro-ingénierie des composants, et au delà, des autres éléments réutilisables, tels que les features. L'identification des architectures utiliserait toutes les sources potentielles, telles que les descripteurs de déploiement proposés par certains frameworks (OSGi, Spring, Maven, ...).

Une autre perspective est de travailler sur l'exploitation des informations stockées dans les dépôts de code (cf. section 3.2.1). En effet, l'historique des différentes versions produit potentiellement, de manière implicite, une famille de produits logiciels sur laquelle nous pouvons appliquer notre approche d'identification de features. Mais pour assurer la pertinence des résultats, nous voulons travailler sur la sémantique de la relation de dérivation liant deux versions. Notre objectif est de pouvoir distinguer les versions qui contiennent des révisions d'implémentation de features des versions qui introduisent des variantes d'implémentation de features ou de nouveaux features, afin que les modèles de features extraits puissent proposer une vue abstraite du contenu du dépôt, aidant les architectes à comprendre et sélectionner les différentes configurations existantes.

D'une toute autre manière, rapprocher modélisation et programmation (cf. section 3.2.3), en dotant les langages de programmation d'une syntaxe graphique, est également une réponse à cette question. Cette perspective supprimerait en partie la nécessité d'iden-

tifier et d'extraire a posteriori la définition des architectures et des composants puisque ces éléments feraient partie intégrante du code des applications, en tant que vue abstraite de leurs structures et de leurs fonctionnalités. A l'image des langages orientés objets qui, au travers d'un nouveau concept de programmation, ont renforcé l'usage des bonnes pratiques de programmation en termes de modularité, d'encapsulation et de découplage, on peut espérer que la mise en relief des architectures et des composants aidera les développeurs à produire des des applications de meilleure qualité (ce qui rejoint notre perspective sur la conception et l'évaluation de métriques de qualité des architectures (cf. section 3.2.2).

4.1.3 Q3 - Comment indexer les artefacts réutilisables pour permettre leur recherche et leur sélection ?

Nos travaux ont proposé des solutions pour la construction automatique de hiérarchies de types de composants (cf. section 2.2.3). Ces mécanismes peuvent être utilisés pour indexer des bibliothèques de composants à partir de leurs descriptions syntaxiques, afin de répondre efficacement à des requêtes recherchant des composants pour une connexion ou une substitution dans une architecture. Ce travail est détaillé par l'article présenté dans la section 4.2.3 et inclus dans l'annexe D. Ces mécanismes d'indexation syntaxique sont à considérer en complément d'une indexation lexicale, aspect que nous avons exploré dans des travaux étudiant le remplacement dynamique dans des orchestrations de web-services.

Notre perspective principale sur cette question est d'étudier la mise en correspondance des modèles de features et des ADLs (cf. section 3.1.1), déjà évoquée comme une réponse à Q2 (cf. section 4.1.2). Les features fourniraient des définitions abstraites, sémantiques, de fonctionnalités ou de caractéristiques implémentées par des composants ou des architectures. La structuration du modèle de features serait alors exploitée pour explorer le contenu des bibliothèques de composants et d'architecture afin de rechercher et sélectionner des éléments correspondants aux besoins des utilisateur ou aux exigences fonctionnelles d'un cahier des charges.

Une autre perspective, connexe à la rétroingénierie de lignes de produits à partir de dépôts de code (cf. section 3.2.1) évoquée dans Q2 (cf. section 4.1.2), serait la réingénierie des historiques de version dans les dépôts de code. Une fois les features identifiés, ainsi que les différentes versions et variantes de leurs implémentations sous la forme de composants logiciels, une fois proposé un modèle de features correspondant, l'idée serait de reconstruire l'historique des versions afin que l'organisation des différentes branches soit conforme à ce modèle de features. En d'autres termes, il s'agirait d'étudier des stratégies de gestion des dépôts de code offrant un support efficace à une approche d'ingénierie de ligne de produits puis de proposer des outils pour assister l'application et faciliter le respect de ces stratégies (par exemple imposer que chaque feature possède sa propre branche de développement afin de pouvoir évoluer et être réutilisé indépendamment).

4.1.4 Q4 - Comment estimer l'effort d'adaptation d'artefacts répondant partiellement ou approximativement à un besoin ?

Cette question n'a jamais été directement abordée dans nos travaux. Elle est connexe à la question Q3 (cf. section 4.1.3)parce qu'elle aborde sous un autre angle la question de la recherche et de la sélection d'éléments réutilisables. Nous avons étudié des mécanismes de requête souples (cf. section 2.2.3), capables de proposer des éléments ne

correspondant pas directement et strictement aux critères de recherche. Cependant, ces mécanismes sont basés sur des hiérarchies de types syntaxiques. Les types des éléments proposés sont syntaxiquement compatibles (substituables) aux types recherchés : ils sont utilisables sans effort d'adaptation.

Nous avons étudié d'autres mécanismes, basés sur la proximité lexicale, pour rechercher des webservices similaires afin de gérer des substitutions de services dans des orchestrations. Mais nous n'avons pas abordé dans ces travaux l'évaluation de l'effort d'adaptation de l'interface des webservices pour permettre leur intégration concrète, ce qui aurait fourni un autre critère pertinent de classement ou de sélection des propositions.

Une perspective dans ce domaine consisterait à expérimenter des mécanismes de recherche étendus, exploitant les hiérarchies de types pour proposer non seulement des réponses strictement compatibles mais également des réponses proches de la compatibilité. A l'instar de la proximité sémantique, qui exploite la structure des ontologies, nous pourrions également suggérer des résultats correspondant à des types voisins (c'est-à-dire ayant un ancêtre commun avec le type de résultat recherché). Il faudra alors disposer d'une mesure de l'effort d'adaptation pour ordonner et filtrer les résultats. On pourrait imaginer qu'une distance syntaxique ou comportementale, à l'image de la distance sémantique, puisse fournir une estimation de cet effort. Cette mesure serait validée suivant une approche de génie logiciel empirique, en essayant d'extraire des dépôts de code open source une base d'expérimentation et de validation. La construction de cette base consisterait, par exemple, à repérer des modifications correspondant à des changements de types de composants, pouvant s'interpréter comme des adaptations, puis à mesurer le coût du développement correspondant, en analysant les commits. Il faudrait ensuite essayer d'établir un coût moyen par type de modification, en s'appuyant sur des classifications structurelles et lexicales (le coût moyen dépendant peut-être du domaine des composants). Nous rattachons ainsi cette perspective à celle d'élaboration de métriques pour assister la prise de décisions architecturales (cf. section 3.2.2).

4.1.5 Q5 - Comment assembler un ensemble d'artéfacts réutilisables pour produire la définition complète et cohérente d'un logiciel ?

Nous avons répondu à cette question en proposant un modèle de ports composites (cf. section 2.1.1). Ce travail est détaillé par l'article présenté dans la section 4.2.4 et inclus dans l'annexe E. Ce modèle de ports composites permet d'assurer, de manière structurelle, un premier niveau de vérification (une condition nécessaire) de complétude et de correction d'une architecture. Chaque connexion d'un port garantit que toutes les interfaces nécessaires à l'exécution du protocole de collaboration qui lui est associé ont été réalisées. Une architecture est correcte quand tous les ports sont correctement connectés (entièrement connectés ou déconnectés).

Ce travail nous a également permis d'aborder la construction automatique d'architectures dans une approches de search-based software engineering. Modélisé comme un problème de satisfaction de contraintes, ayant pour objectif la connexion d'un ensemble initial de ports correspondant aux exigences fonctionnelles à réaliser, notre algorithme explore l'espace des constructions d'architectures possibles, à l'aide des bibliothèques de composants disponibles, en s'appuyant sur un ensemble d'heuristiques de recherche.

On peut également rapprocher de cette question nos travaux sur l'adaptation autonomique de logiciels sensibles au contexte (cf. section 2.3). Ils sont illustrés par l'article présenté dans la section 4.2.5 et inclus dans l'annexe F. Nos avons étudié différentes solutions permettant la définition puis de l'exécution d'applications construites par réutilisa-

tion des services fournis par des environnements intelligents. Elles proposent différents mécanismes d'orchestration de services mis en œuvre par des frameworks SCA (SAASHA, SAS, AROLD). Les orchestrations sont ainsi transformées en architectures à bases de composants dans lesquelles des composants orchestrateurs sont connectés à des composants contrôlant l'utilisation des ressources fournies par l'environnement. Nos travaux sur la construction et l'évolution d'architectures à base de composants peuvent être ainsi appliqués à la gestion de ces applications.

L'une des perspectives est d'étendre nos travaux sur le typage des composants et par conséquent sur le typage des autres éléments les constituant (les ports, les interfaces, les fonctions). Cette perspective consisterait à baser le typage sur des descriptions comportementales, pour dépasser les limites d'un typage basé sur des descriptions syntaxiques. Cette perspective est évoquée dans nos travaux sur les ports composites (cf. section 2.1.1), au travers des différents protocoles d'interaction associés aux interfaces, aux ports et aux composants. Mais notre exploitation de ces protocoles dans le typage reste élémentaire et laisse un ensemble de questions ouvertes. Par exemple, comment vérifier la cohérence entre le protocole de comportement d'un composant composite et le comportement de son architecture interne ? Comment calculer la compatibilité entre deux protocoles, lister toutes les collaborations effectives possibles (séquences d'échanges de messages) et choisir celle qui répond le mieux aux objectifs fonctionnels ?

Cette perspective rejoint, sous un autre angle, l'étude d'une relation de spécialisation entre architectures (cf. section 3.1.2), évoquée pour résoudre Q1 (cf. section 4.1.1).

Cette perspective ne fournira qu'une réponse partielle à la problématique du typage des composants, en l'absence d'une spécification formelle permettant de comparer précisément la sémantique des fonctions, sous la forme, par exemple, d'assertions (pre-, post-conditions et invariants). Mais nous ne visons pas des domaines critiques, pour lesquels une vérification formelle est indispensable. Une autre voie serait d'exprimer la sémantique des fonctions et de vérifier la conformité des comportements à l'aide de tests. Mais cela entrainerait alors des coûts de développement supplémentaires. Nous cherchons des solutions simples, pragmatiques, pouvant être facilement adoptées par toutes les équipes de développement : c'est ce que viserait notre renforcement du typage à l'aide de protocoles d'interaction.

4.1.6 Q6 - Comment gérer l'évolution d'un logiciel pour éviter son obsolescence ?

La problématique de la gestion de l'évolution des logiciels a été premièrement abordée dans nos travaux sur la co-évolution des modèles d'architectures (cf. section 2.2.2). Nous avons proposé un méta-modèle formel de notre ADL Dedal, défini en langage B. Cette formalisation permet d'exprimer et de calculer des propriétés de correction intra-niveau d'abstraction et de cohérence inter-niveaux d'abstraction dans la définition d'une architecture. Lors d'une évolution, il est ainsi possible d'évaluer l'impact des modifications envisagées en termes de dérive et d'érosion des différents modèles de définition. L'utilisation d'un solver permet de générer automatiquement des plans d'évolution qui, par propagation des changements dans et entre les différents niveaux, permettent de maintenir les propriétés de correction et de cohérence de la définition d'une architecture. Ces travaux sont détaillés par l'article présenté dans la section 4.2.1 et inclus dans l'annexe B.

Ils sont actuellement poursuivi dans une thèse qui porte, entre autres, sur la gestion du versionnement des architectures [29, 92]. En ajoutant la dimension historique à la ges-

tion de l'évolution, nous voulons fournir un support discipliné et outillé à cette forme pragmatique de réutilisation, couramment utilisée dans l'industrie, par clonage et modification. Mais en renforçant la sémantique de la relation de dérivation, pour distinguer la création de révisions et de variantes et en outillant des stratégies adaptées de gestion des historiques, comme évoqué dans précédemment dans Q2 (cf. section 4.1.2), nous souhaitons également utiliser la gestion du versionnement comme un support à l'ingénierie de lignes de produits, les modèles de features fournissant des vues abstraites du graphe de versions.

Un deuxième aspect de la gestion de l'évolution a été abordé par nos travaux sur l'adaptation dynamique d'applications sensibles au contexte (cf. section 2.3). Comme évoqué dans Q5 (cf. section 4.1.5), nous avons étudié des mécanismes permettant le déploiement et l'exécution d'orchestrations de services. Tous ces mécanismes ont été conçus pour être sensibles au contexte afin de permettre l'adaptation dynamique des architectures à base de composants, qui, dans une approche SCA, implémentent les orchestrations. Une partie de ces adaptations sont non anticipées. Il s'agit, par exemple, des substitutions dynamiques de services, qui en fonction de leur disponibilité, permet d'assurer le maintien de l'exécution ou l'amélioration de la qualité de services. Il s'agit également de la distribution de l'exécution de l'orchestration, en fonction de la charge des agents gérant l'environnement et de la localisation des ressources. Une autre partie de ces adaptations est anticipée. Il s'agit, par exemple, de la définition de différents modes d'exécution ou de différentes stratégies d'implémentation pour une même fonctionnalité, permettant le maintien des missions d'un système en fonction des ressources disponibles dans son environnement. C'est une illustration du concept de "model at runtime" [100]. Les modèles des missions, des stratégies et des architectures associées sont embarqués et utilisés par un système multi-agents pour gérer de manière autonome l'évolution du système pendant son exécution.

Une perspective de ces travaux est de prolonger l'idée proposée par le langage de missions d'AROLD : mettre en relation une fonctionnalité avec un ensemble d'architectures définissant des variantes d'implémentation adaptées à des contextes spécifiques. Cette perspective rejoint celles évoquées dans la section 3.1.1. Le concept de mission est en effet proche de celui de feature, lorsque ce dernier décrit une fonctionnalité abstraite. Mais une mission possède une dimension opérationnelle supplémentaire, destinée à gérer le deploiement et l'évolution de l'exécution d'une fonctionnalité. Dans le continuum que nous souhaitons créer entre les modèles de features et les ADLs, un concept équivalent à celui des missions d'AROLD pourrait fournir un niveau de représentation et de structuration pertinent voire nécessaire. Les perspectives proposées sur le typage des architectures (cf. section 3.1.2) permettraient de vérifier la cohérence globale des différentes implémentations proposées pour une mission, voire de travailler sur la classification des missions elles-mêmes.

4.2 Articles sélectionnés

4.2.1 A formal approach for managing component-based architecture evolution

Cet article de journal international décrit le méta-modèle formel que nous avons conçu pour notre ADL Dedal. Une définition formelle des relations existant entre composants, au sein d'un niveau d'abstraction donné, et des relations existant entre les différents niveaux d'abstraction permet d'établir des propriétés de correction et de cohérence des

modèles d'architectures multi-niveaux écrits en Dedal. Ces propriétés permettent également d'analyser l'impact local ou global des changements réalisés sur un modèle d'architecture et, par l'utilisation de solveurs, dans une approche de search-based software engineering, de calculer automatiquement les propagations de changements devant être réalisées pour rétablir les propriétés de correction et de cohérence. Nous proposons ainsi une solution pour aider les architectes à gérer la co-évolution des différents niveaux de définition des modèles d'architectures Dedal.

4.2.2 Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach

Cet article de journal international décrit le processus que nous proposons pour documenter automatiquement un ensemble de features extraits du code source d'une famille de logiciels. Il combine des techniques similaires au processus d'extraction des features que nous avons proposé par ailleurs. Une première étape consiste à réaliser une classification des features et des éléments de documentation (cas d'utilisation) grâce à une analyse relationnelle de concepts afin de les regrouper en blocs représentatifs de leurs co-occurrences. Une deuxième étape consiste à utiliser une mesure de similarité lexicale (Latent Semantic Indexing) pour identifier, au sein de chaque bloc, les associations entre implémentation de features et cas d'utilisation les plus pertinentes. Une dernière étape, de classification par analyse formelle de concepts, structure l'ensemble des associations. Le nom des features est alors automatiquement extrait des cas d'utilisation qui leurs sont associés. La combinaison de ces techniques permet à la fois la performance (réduction de la complexité par découpage en blocs) et la précision (analyse sémantique et lexicale) du processus.

4.2.3 Formal concept analysis-based service classification to dynamically build efficient software component directories

Cet article de journal international décrit notre théorie du typage des composants logiciels. Elle propose une définition formelle de la relation de spécialisation entre types syntaxiques de composants reposant sur un principe de substitution inspiré des langages orientés objets (substitution possible d'une instance d'un type de composant par une instance d'un sous-type). Nous décrivons également comment cette relation de spécialisation peut être utilisée pour indexer de manière structurée un annuaire ou un dépôt de composants. Une analyse formelle de concepts est utilisée en plusieurs phases pour construire automatiquement une classification hiérarchique des types de composants. Une première étape consiste à construire, à partir d'une hiérarchie des types de données, une hiérarchie des signatures des fonctionnalités des composants, basée sur le type des paramètres d'entrée et de sortie. Une deuxième étape exploite cette première hiérarchie pour construire la hiérarchies des types d'interfaces requises et fournies. Enfin, la hiérarchie des types de composants est construire à l'aide de la hiérarchie des types d'interfaces. Cette hiérarchie de types de composants, basée sur le principe de substitution, permet de répondre efficacement à des requêtes de recherche de composants pour construire (connection à une liste d'interfaces) ou faire évoluer (remplacement d'un composant)

une architecture.

4.2.4 Search-based many-to-one component substitution

Cet article de journal international décrit notre proposition d'un modèle de ports composites venant étendre la description structurelle et le typage syntaxique des composants afin de fournir un moyen intuitif et efficace d'assister et d'automatiser la construction et l'évolution d'architecture logicielles à base de composants. Chaque port composite correspond à la documentation d'un usage précis du composant. Sa définition reflète structurellement et syntaxiquement le protocole de collaboration qu'il doit exécuter (dépendances de connexion d'interfaces et contraintes de connexion d'interfaces à un même composant). Notre modèle de ports composite permet de définir des propriétés de complétude et de correction d'une architecture, peu coûteuses en calcul, qui sont des conditions nécessaires à la vérification formelle de la correction de son comportement global. Nos propriétés de complétude et de correction peuvent être ainsi utilisées, dans une première phase, pour guider le travail d'un architecte et éviter certaines erreurs de construction avant d'engager de plus lourds calculs formels. Nous décrivons également comment ces propriétés peuvent être utilisées, dans une approche de search-based software engineering, pour construire automatiquement des architectures. A partir d'une sélection de composants et de ports composites, représentant les exigences fonctionnelles à satisfaire et d'une bibliothèque de composants disponibles, nous traduisons la vérification des propriétés de complétude et de correction de l'architecture en la résolution d'un problème de satisfaction de contraintes. La résolution de ce problème fournit toutes les solutions de construction d'architectures possibles. Nous décrivons l'ensemble des heuristiques étudiées pour optimiser l'exploration de l'espace de recherche et leur expérimentation sur des exemples générés aléatoirement.

4.2.5 User-defined scenarios in ubiquitous environments : creation, execution control and sharing

Cet article de conférence internationale est représentatif d'un domaine d'application que nous privilégions : les environnements intelligents construits par orchestration de fonctionnalités fournies par des objets connectés. Il présente, SAAS, un framework que nous proposons pour gérer la création, l'exécution et le partage de scénarios applicatifs. Conçu dans une approche SCA (Service Component Architecture), les scénarios sont définis à l'aide d'un langage d'orchestration et encapsulés dans des composants permettant de gérer leur déploiement et leur réutilisation en tant que services. SAAS assure un contrôle sensible au contexte, permettant l'interruption et la reprise de l'exécution des scénarios en fonction de la disponibilité des services requis (avec le remplacement possible des services par des services de même type). Il gère également différents modes d'exécution partagée des scénarios entre les utilisateurs présents dans même environnement.

Bibliographie

[1] La loi de Moore, http ://www.astrosurf.com/luxorion/loi-moore.htm, 2016. 1, 6

[2] eRCA, https ://code.google.com/archive/p/erca/, 2018. 36

[3] Sillitti A., Vernazza T., and Berlin Heidelberg Succi G. Lecture Notes in Computer Science, vol 2319. Springer. Service Oriented Programming : A New Paradigm of Software Reuse. In *Software Reuse : Methods, Techniques, and Tools*, volume 2319. Springer, Berlin, Heidelberg, 2002. 37

[4] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 2005. 31

[5] Jiri Adamek and Frantisek Plasil. Partial bindings of components - any harm ? In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 632–639, Busan, Korea, November 2004. IEEE. 18

[6] Rafat AL-Msie'deen, Marianne Huchard, Abdelhak Djamel Seriai, Christelle Urtado, and Sylvain Vauttier. Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach. *International Journal of Software Engineering and Knowledge Engineering*, 24(10) :1413–1438, December 2014. 8, 43

[7] Rafat AL-Msie'deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, and Sylvain Vauttier. Reverse engineering feature models from software configurations using Formal Concept Analysis. In Karell Bertet and Sebastian Rudolph, editors, *Proceedings of the 11th international conference on Concept Lattices and their Applications (CLA 2014)*, volume 1252 of *CEUR Workshop Proceedings*, pages 95–106, Košice, Slovakia, October 2014. 43

[8] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In Chengcui Zhang, James Joshi, Elisa Bertino, and Bhavani Thuraisingham, editors, *Proceedings of the 14th IEEE international conference on Information Reuse and Integration (IRI 2013)*, pages 586–593, San Francisco, USA, August 2013. IEEE. 43

[9] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Documenting the mined feature implementations from the object-oriented source code of a collection of software product variants. In *Proceedings of the 26th international conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, pages 138–143, Vancouver, Canada, July 2014. 43

[10] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Ahmad Khlifat. Concept lattices : a representation space to structure software variability. In *Proceedings of the 5th International Conference on Information and Communication Systems (ICICS 2014)*, page 6, Irbid, Jordan, April 2014. IEEE. 43

[11] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. An approach to recover feature models from object-oriented source code. In *Actes de la Journée Lignes de Produits 2012*, pages 15–26, Lille, France, Novembre 2012. 43

[12] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature location in a collection of software product variants using formal concept analysis. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Reuse - Proceedings of the 13th International Conference on Software Reuse*, number 7925 in LNCS, pages 302–307, Pisa, Italy, June 2013. Springer. 43

[13] Rafat AL-Msie'deen, Abdelhak Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing. In *Proceedings of the 25th international conference on Software Engineering and Knowledge Engineering (SEKE 2013)*, pages 244–249, Boston, USA, June 2013. 43

[14] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 187–197, Orlando, USA, May 2002. ACM. 11, 19

[15] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer-Verlag Berlin Heidelberg, 2013. 1, 7, 8, 29

[16] Gabriela Arévalo, Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Component and service farms. In Eric Cariou, Laurence Duchien, and Yves Ledru, editors, *Actes des deuxièmes journées nationales du GdR Génie de la Programmation et du Logiciel — Défis pour le Génie de la Programmation et du Logiciel*, pages 281–284, Pau, France, Mars 2010. 33, 37

[17] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Precalculating component interface compatibility using FCA. In Jean Diatta, Peter Eklund, and Michel Liquière, editors, *Proceedings of the 5th international conference on Concept Lattices and their Applications (CLA 2007)*, pages 241–252. CEUR Workshop Proceedings Vol. 331, ISSN 1613-0073, Montpellier, France, October 2007. 33, 36

[18] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services. In Y. Aït-Ameur, editor, *Actes de la 2ème Conférence francophone sur les Architectures Logicielles (CAL 2008)*, Revue des Nouvelles Technologies de l'Information (RNTI-L-2), ISSN 1764-1667, pages 123–138. Editions Cépaduès, Montréal, Canada, March 2008. Cet article a été accepté conjointement à LMO 2008. AR 53,6%. 33, 36

[19] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services utilisant l'analyse formelle de concepts. In Yves Ledru and Marc Pantel, editors, *Actes des journées nationales du GdR Génie de la Programmation et du Logiciel*, pages 130–131, Toulouse, France, Janvier 2009. ENSEEIHT-IRIT. 33, 36

[20] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Formal concept analysis-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, 38(4) :427–453, May 2009. 33, 36

[21] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things : A survey. *Computer Networks*, 54(15) :2787 – 2805, 2010. 7

[22] Zeina Azmeh, Fady Hamoui, Marianne Huchard, Nizar Messai, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Backing composite web services using Formal Concept Analysis. In Robert Jäschke and Petko Valtchev, editors, *Proceedings of the 9th International Conference on Formal Concept Analysis (ICFCA 2011)*, number 6628 in LNCS / LNAI, pages 26–41, Nicosia, Cyprus, May 2011. Springer. 33, 37, 38

[23] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. WSPAB : A tool for automatic classification & selection of web services using formal concept analysis. In *Proceedings of the 6th IEEE European Conference on Web Services (ECOWS 2008)*, pages 31–40, Dublin, Ireland, November 2008. IEEE. 33, 37, 38

[24] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Using concept lattices to support web service compositions with backup services. In *Proceedings of the 5th International Conference on Internet and Web Applications and Services (ICIW 2010)*, pages 363–368, Barcelona, Spain, May 2010. IEEE. 33, 37, 38

[25] K. A. Bohrer B. S. Rubin, A. R. Christ. Java and the IBM San Francisco project. *IBM Systems Journal*, 37 :365 – 371, 1998. 7

[26] Marc Barbut and Bernard Monjardet. *Ordre et classification : algèbre et combinatoire*. Hachette, 1970. 14, 36, 43

[27] Barry Boehm, Hans Dieter Rombach, and Marvin V. Zelkowitz. *Foundations of Empirical Software Engineering - The Legacy of Victor R. Basili*. Springer-Verlag, 2005. 51

[28] Bonitasoft. Bonita, https ://documentation.bonitasoft.com/bonita/7.6/. 52

[29] Alexandre Le Borgne, David Delahaye, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. A Version-Aware Three-Level Architecture Description Language. In *19th International Conference on Software Engineering and Knowledge Engineering*, 2017. 60

[30] Gilad Bracha and William Cook. Mixin-based Inheritance. *ECOOP*, 1990. 10

[31] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model Driven Software Engineering in Practice*. Morgan & Claypool, 2012. 10, 22

[32] Marco Brambilla and Piero Fraternali. *Interaction Flow Modeling Language : Model-Driven UI Engineering of Web and Mobile Apps with IFML*. Morgan Kaufmann, 2014. 52

[33] E. Bruneton, T. Coupaye, and J-B. Stefani. Fractal specification, v 2.0-3. http ://fractal.ow2.org/specification/index.html [Last checked 2009-04-27], February 2004. 12, 21, 28

[34] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP 2002)*, Malaga, Spain, 2002. 18

[35] Rajkumar Buyya, James Broberg, and Andrzej Goscinski. *Cloud Computing : Principles and Paradigms*. Wiley & Sons, 2010. 7

[36] Aurélien Campéas, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Distributed exception handling : ideas, lessons and issues with recent exception handling systems. In Nicolas Guelfi, editor, *International workshop on Rapid Integration of Software Engineering techniques (RISE 2004), revised selected papers, November 2004*, number 3475 in LNCS, pages 82–92. Springer, Luxembourg-Kirchberg, Luxembourg, 2005. 43

[37] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2014. 50

[38] Alistair Cockburn. *Agile Software Development : The Cooperative Game*. Addison Wesley, second edition, 2006. 8, 15

[39] Ivica Crnković, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, page 44, Papeete, French Polynesia, October 2006. IEEE. 8

[40] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5), Sept. - Oct. 2011. 11

[41] Ivica Crnković, Judith Stafford, and Clemens Szyperski. Software components beyond programming : From routines to services. *IEEE Software*, 28(3) :22–26, May / June 2011. 10

[42] Luca De Alfaro and Thomas A Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5) :109–120, 2001. 18

[43] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion : A survey. page 132–151, 2012. 30

[44] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6) :391–407, September 1990. 33, 43, 45

[45] Nicolas Desnos, Marianne Huchard, Guy Tremblay, Christelle Urtado, and Sylvain Vauttier. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution : Research and Practice, Special issue on Search-Based Software Engineering*, 20(5) :321–344, September / October 2008. 18, 27, 34

[46] Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay. Automated and unanticipated flexible component substitution. In H. W. Schmidt et al., editors, *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)*, volume 4608 of *LNCS*, pages 33–48, Medford, USA, July 2007. Springer. 18, 27

[47] Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. Assistance à l'architecte pour la construction d'architectures à base de composants. In Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors, *Actes de la 12ème conférence francophone sur les Langages et Modèles à Objets (LMO 2006)*, pages 37–52, Nîmes, France, March 2006. Hermès. 18, 27

[48] Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard. Automating the building of software component architectures. In Volker Gruhn and Flavio Oquendo, editors, *Proceedings of the 3rd European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications*, volume 4344 of *LNCS*, pages 228–235, Nantes, France, September 2006. Springer. 18, 27

[49] Edsger Dijkstra. The humble programmer (ewd340). *Communications of the ACM*, 10(15) :859–866, 1972. 6

[50] Christophe Dony, Chouki Tibermacine, Christelle Urtado, and Sylvain Vauttier. Specification of an exception handling system for a replicated agent environment. In *Proceedings of the 4th International Workshop on Exception Handling (WEH.08) at 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 24–31, Atlanta, USA, November 2008. ACM. 43

[51] Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Exception handling and asynchronous active objects : issues and proposal. In C. Dony, J.L. Knudsen, A.B. Romanovsky, and A. Tripathi, editors, *Advanced topics in exception handling techniques*, volume 4119 of *LNCS*, chapter 5, pages 81–100. Springer, 2006. 43

[52] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits : A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2) :331–388, March 2006. 10

[53] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Towards scenario creation by service composition in ubiquitous environments. In Stéphane Ducasse, Laurence Duchien, and Lionel Seinturier, editors, *Proceedings of the 9th edition of the BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, pages 145–155, Lille, France, December 2010. 39

[54] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. A service component framework for multi-user scenario management in ubiquitous environments. In *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA 2011)*, pages 155–160, Barcelona, Spain, October 2011. 39

[55] Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. User-defined scenarios in ubiquitous environments : creation, execution control and sharing. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, pages 302–307, Miami, USA, July 2011. 39

[56] Jacques Ferber. *Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence*. Addison Wesley, 1999. 38

[57] Stefan Fischer, Lukas Linsbauer, and Roberto Erick Lopez-Herrejon. Enhancing clone-and-own with systematic reuse for developing software variants. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014. 8, 29, 50

[58] Martin Fowler. Technical Debt, https ://martinfowler.com/bliki/TechnicalDebt.html. 51

[59] Martin Fowler. *Domain-Specific Language*. Addison-Wesley Professional, 2010. 15, 50

[60] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis : Mathematical Foundations*. Springer, 1st edition, 1997. 14, 36, 43

[61] Dimitra Giannakopoulou, Corina S. Päsäreanu, and Howard Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE international conference on Automated Software Engineering (ASE 2002)*, pages 3–12, Edinburgh, UK, September 2002. 18

[62] Guillaume Grondin, Matthieu Faure, Christelle Urtado, and Sylvain Vauttier. Mission-oriented autonomic configuration of pervasive systems. In *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA 2012)*, pages 685–690, Lisbon, Portugal, November 2012. 39

[63] Florian Hacklinger. JavaA - taking components into java. In *Proceedings of the ISCA 13th international conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)*, pages 163–168, Nice, France, July 2004. 11, 19

[64] Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in Action*. Manning Publications, 2011. 34, 38, 42

[65] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Specification of a component-based domotic system to support user-defined scenarios. In *Proceedings of 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, pages 597–602, Boston, USA, July 2009. 38

[66] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. SAASHA : a Self-Adaptable Agent System for Home Automation. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, pages 227–230, Lille, France, September 2010. IEEE. 38

[67] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Un système d'agents à base de composants pour les environnements domotiques. In Eric Cariou and Jean-Claude Royer, editors, *Actes de la 16ème conférence francophone sur les Langages et Modèles à Objets (LMO 2010)*, pages 35–49, Pau, France, Mars 2010. 38

[68] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987. 33

[69] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14) :833 – 839, 2001. 14, 27, 32

[70] M. Huchard, M. Rouane Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Annals of Mathematics and Artificial Intelligence*, 49(1) :39–76, Apr 2007. 45

[71] Michael Jeronimo and Jack Weast. *UPnP Design by Example : A Software Developer's Guide to Universal Plug and Play*. Intel Press, 2003. 38, 42

[72] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. 1(2), 1988. 10, 18

[73] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) : Feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990. 7, 44

[74] Hans Kellerer, Ulright Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004. 41

[75] Siddhartha Kumar Khaitan and James D. McCalley. Design Techniques and Applications of Cyber Physical Systems : A Survey. 2014. 50

[76] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 10

[77] Donald E. Knuth. *Art of Computer Programming, Volume 4B, Fascicle 5*. Addison Wesley, 2018. 27

[78] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(2-3) :259–284, 1998. 14

[79] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9) :1060–1076, September 1980. 29

[80] Meir M. Lehman and Juan Carlos Fernandez-Ramil. Towards a theory of software evolution - and its practical impact. In *Proceedings of the International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 2–11, November 2000. 29

[81] Barbara H. Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Notices*, 23 :17–34, January 1987. 33

[82] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16 :1811–1841, 1994. 33

[83] Dominique Luzeaux, Jean-René Ruault, and Jean-Luc Wippler. *Complex systems and systems of systems engineering*. John Wiley & Sons, 2013. 14

[84] M. W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 4(1) :267–284., 1998. 7

[85] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. 2nd edition, 2002. 11

[86] MDA. Omg model driven architecture. 23

[87] Nenad Medvidovic, Alexander Egyed, and Paul Grünbacher. Stemming architectural erosion by coupling architectural discovery and recovery, 2003. 29

[88] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000. 12, 22

[89] V. Mencl. Specifying component behavior with port state machines. *Electr. Notes Theor. Comput. Sci.*, 101 :129–153, 2004. 21

[90] Bertrand Meyer. Applying Design by Contract. 25(10) :40–51, 1992. 34

[91] George A. Miller. Wordnet : A Lexical Database for English. 38(11) :39–41, 1995. 33

[92] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, , and Sylvain Vauttier. A three-level versioning model for component-based software architectures. In *11th International Conference on Software Engineering Advances (ICSEA 2016)*, Roma, Italy, August 2016. ERA rank C. 60

[93] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Modélisation et vérification formelles en B des architectures logicielles à trois niveaux. In Catherine Dubois, Nicole Levy, Marie-Agnès Peraldi-Frati, and Christelle Urtado, editors, *Actes de la troisième Conférence en IngénieriE du Logiciel (CIEL 2014)*, pages 71–77, Paris, France, Juin 2014. 25

[94] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Towards automating the coherence verification of multi-level architecture descriptions. In *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA 2014)*, pages 416–421, Nice, France, October 2014. 25

[95] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. An evolution management model for multi-level component-based software architectures. In *Proceedings of the 27th international conference on Software Engineering and Knowledge Engineering (SEKE 2015)*, pages 674–679, Pittsburgh, USA, July 2015. 29

[96] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Formal rules for reliable component-based architecture evolution. In Ivan Lanese and Eric Madelaine, editors, *11th international symposium on Formal Aspects of Component Software (FACS 2014), September 2014, Bertinoro, Italy, Revised Selected papers*, volume 8997 of *LNCS*, pages 127–142. Springer, 2015. 29

[97] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. A three-level formal model for software architecture evolution. In Davide Di Ruscio and Vadim Zaytsev, editors, *Post-proceedings of the 7th Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE 2014)*, volume 1354 of *CEUR Workshop Proceedings*, pages 102–111, L'Aquila, Italy, July 2015. 29

[98] Abderrahman Mokni, Christelle Urtado, Sylvain Vauttier, Marianne Huchard, and Huaxi (Yulin) Zhang. A formal approach for managing component-based architecture evolution. *Science of Computer Programming, special issue of the 11th international symposium on Formal Aspects of Component Software*, 127 :24–49, October 2016. 29

[99] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, 1965. 6

[100] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models at Runtime to Support Dynamic Adaptation. IEEE Computer. October 2009. 61

[101] OMG. Unified modeling language, version 2.5, version 4.0 ,http ://www.omg.org/spec/uml/2.5/pdf/, 2015. 1, 11, 13, 19, 21, 33

[102] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. socloud : a service-oriented component-based paas for managing portability, provisioning, elasticity, and high availability across multiple clouds. *Computing*, 98(5) :539–565, 2016. 50

[103] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992. 29

[104] František Plášil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11) :1056–1076, November 2002. 18, 34

[105] Stefan Poslad. *Autonomous systems and Artificial Life*, page 317–341. Wiley, 2009. 38

[106] Ralf H. Reussner. Counter-constrained finite state machines : A new model for component protocols with resource-dependencies. In William I. Grosky and František Plášil, editors, *SOFSEM 2002 : Theory and Practice of Informatics : 29th Conference on Current Trends in Theory and Practice of Informatics Milovy, Czech Republic, November 22–29, 2002 Proceedings*, volume 2540 of *LNCS*, pages 20–40. Springer, Berlin, Heidelberg, 2002. 18

[107] Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning about software architectures with contractually specified components. In Alejandra Cechich, Mario Piattini, and Antonio Vallecillo, editors, *Component-Based Software Quality : Methods and Techniques*, volume 2693 of *LNCS*, pages 287–325. Springer, 2003. 18

[108] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IN PROCEEDINGS OF THE WORKSHOP ON MOBILE COMPUTING SYSTEMS AND APPLICATIONS*, pages 85–90. IEEE Computer Society, 1994. 38

[109] Ed Seidewitz. Uml with meaning : Executable modeling in foundational uml and the alf action language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 61–68, New York, NY, USA, 2014. ACM. 52

[110] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA applications with the frascati

platform. In *2009 IEEE International Conference on Services Computing (SCC 2009), 21-25 September 2009, Bangalore, India*, pages 268–275, 2009. 14, 37

[111] Nigel Shadbolt, Wendy Hal, and Tim Berners-Lee. The Semantic Web Revisited. 2006. 33

[112] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing : Vision and challenges. *IEEE Internet of Things Journal*, 3(5) :637–646, Oct 2016. 50

[113] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, second edition, 2000. 7

[114] Frédéric Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. A proposition for exception handling in multi-agent systems. In *Proceedings of the 2nd international workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003)*, pages 136–143, Portland, USA, May 2003. 43

[115] Frédéric Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Improving exception handling in multi-agent systems. In Carlos José Pereira de Lucena, Alessandro F. Garcia, Alexander B. Romanovsky, Jaelson Castro, and Paulo S. C. Alencar, editors, *Software engineering for multi-agent systems II, Research issues and practical applications*, volume 2940 of *LNCS*, pages 167–188. Springer, February 2004. Selected, extended and revised articles from SELMAS 2003. 43

[116] Frédéric Souchon, Christelle Urtado, Sylvain Vauttier, and Christophe Dony. Exception handling in component-based systems : a first study. In A. Romanovsky, C. Dony, JL. Knudsen, and A. Tripathi, editors, *Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003*, pages 84–91, Darmstadt, Germany, July 2003. also available as Technical Report TR 03-028, Department of computer science, University of Minnesota, Minneapolis. 43

[117] Frédéric Souchon, Sylvain Vauttier, Christelle Urtado, and Christophe Dony. Fiabilité des applications multi-agents : le système de gestion d'exceptions SaGE. In Olivier Boissier and Zahia Guessoum, editors, *Systèmes multi-agents défis scientifiques et nouveaux usages – Actes des Journées Francophones sur les Systèmes Multi-Agents 2004*, pages 121–134. Hermès, Paris, France, November 2004. 43

[118] Petr Spacek, Christophe Dony, and Chouki Tibermacine. A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In *Procedings the 17th International ACM SIGSOFT Conference on Component Based Software Engineering*, pages 13—23, Lille, France, July 2014. 11, 19

[119] Judith A. Stafford and Alexander L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 11(04) :431–451, 2001. 18

[120] Clemens Szypersky, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming (Second Edition)*. Addison-Wesley / ACM Press, second edition, 2002. 7, 10, 18

[121] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06)*, Vasteras, Sweden, June 2006. LNCS 4063, Springer-Verlag. 28

[122] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000. 18

[123] Sylvain Vauttier, Martine Magnan, and Chabane Oussalah. Extended Specification of Composite Objects in Uml. *Journal of Object-Oriented Programming*, 1999. 56

[124] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, second edition edition, 2009. 38

[125] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Connector-driven gradual and dynamic software assembly evolution. In Masoud Mohammadian, editor, *Proceedings of the International Conference on Innovation in Software Engineering (ISE 2008)*, pages 345–350, Vienna, Austria, December 2008. IEEE. 12

[126] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In Muhammad Ali Babar and Ian Gorton, editors, *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, August 2010. Springer. 22, 25

[127] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric development and evolution processes for component-based software. In *Proceedings of 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, pages 680–685, Redwood City, USA, July 2010. 22, 25

[128] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Dedal : un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants. In Khalil Drira, editor, *Actes de la 4ème Conférence francophone sur les Architectures Logicielles (CAL 2010), Revue des Nouvelles Technologies de l'Information (RNTI L-5)*, pages 15–27, Pau, France, Mars 2010. Cépaduès. 22

[129] Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. A three-level component model in component-based software development. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE 2012)*, Dresden, Germany, September 2012. ACM. 22, 25

Annexes

Annexe A

Publications

- **Revues internationales**

[Mok16] *A formal approach for managing component-based architecture evolution.* A. MOKNI, M. HUCHARD, C. URTADO, S. VAUTTIER and H. Y. ZHANG. Science of Computer Programming, special issue of the 11th international symposium on Formal Aspects of Component Software, 127:24–49, Elsevier, October 2016. IF 0,715 .

[ALM14d] *Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach. R. AL-MSIE'DEEN*, A. D. SERIAI, M. HUCHARD, C. URTADO and S. VAUTTIER. International Journal of Software Engineering and Knowledge Engineering. World Scientific Publishing. 24(10):1413–1438, December 2014. IF 0,362.

[Are09] *Formal concept analysis-based service classification to dynamically build efficient software component directories*. G. AREVALO, N. DESNOS, M. HUCHARD, C. URTADO and S. VAUTTIER. International Journal of General Systems, J. Diatta, P. Eklund and M. Liquière Editors, 38(4):427-453, Taylor and Francis, May 2009. ISSN 0308-1079. IF 0.551.

[Des08] *Search-based many-to-one component substitution.* N. DESNOS, M. HUCHARD, G. TREMBLAY, C. URTADO and S. VAUTTIER. Journal of Software Maintenance and Evolution: Research and Practice. Special issue on Search-Based Software Engineering, M. Di Penta, G. Antoniol and M. Harman Editors, 20(5):321-344, Wiley, September/October 2008. ISSN 532-060X. IF 0.765.

[Vau99a] *Extended Specification of Composite Objects in UML.* S. Vauttier, M. Magnan et C. Oussalah Journal of Object-Oriented Programming, SIGS Publications, Mai 1999.

- **Revues nationales**

[Mag98] *Modélisation et Gestion du Comportement des Objets Composites*. M. Magnan, S. Vauttier et C. Oussalah Networking and Information Systems Journal, Volume 1, Number 2-3, p 279-308. Editions Hermes, 1998.

- **Conférences internationales**

[Mok16b] *A three-level versioning model for component-based software architectures,* A. MOKNI, M. HUCHARD, C. URTADO and S. VAUTTIER. 11th International Conference on Software Engineering Advances (ICSEA 2016), Roma, Italy, August 2016. AR 30%.

[Mok15a] *An evolution management model for multi-level component-based software architectures,* A. MOKNI, M. HUCHARD, C. URTADO, S. VAUTTIER and H. Y. ZHANG. 27th International Conference on Software Engineering and Knowledge Engineering. Pittsburgh, USA, July 2015. AR 30%.

[Mok14d] *Fostering component reuse: automating the coherence verification of multi-level architecture descriptions*, A. MOKNI, M. HUCHARD, C. URTADO, S. VAUTTIER and H. Y. ZHANG. 9th International Conference on Software Engineering Advances. Nice, France, October 2014. AR 30%.

[Alm14c] *Reverse engineering feature models from software configurations using Formal Concept Analysis*, R. AL-MSIE'DEEN, A. D. SERIAI, M. HUCHARD, C. URTADO and S. VAUTTIER. 11th International Conference on Concept Lattices and their Applications. Košice, Slovakia, October 2014.

[Mok14c] *Formal rules for reliable component-based architecture evolution,* A. MOKNI, M. HUCHARD, C. URTADO, S. VAUTTIER and H. Y. ZHANG. 11th International Symposium on Formal Aspects of Component Software. Bertinoro, Italy, September 2014. AR 45%.

[ALM14b] *Documenting the mined feature implementations from the object-oriented source code of a collection of software product variants,* R. AL-MSIE'DEEN, A. D. SERIAI, M. HUCHARD, C. URTADO and S. VAUTTIER. 26th International Conference on Software Engineering and Knowledge Engineering, pages 138-143. Vancouver, Canada, July 2014. AR 29,6%.

[ALM14a] *Concept lattices: a representaion space to structure software variability,* R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado and S. Vauttier. 5th IEEE International Conference on Information and Communication Systems, Irbid, Jordan, April 2014. 72–77.

[ALM13c] *Mining features from the object-oriented source code of software variants by combining lexical and structural similarity,* R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado and S. Vauttier. 14th IEEE International Conference on Information Reuse and Integration, San Francisco, August 2013. 586–593.

[ALM13b] *Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing,* R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman. 25th International Conference on Software Engineering and Knowledge Engineering, Boston, June 2013. 244–249.

[ALM13a] *Feature location in a collection of software product variants using formal concept analysis,* R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman. 13th International Conference on Software Reuse, Pisa, June 2013. 302–307.

[GRO12] *Mission-oriented autonomic configuration of pervasive systems*. G. GRONDIN, M. FAURE, C. URTADO and S. VAUTTIER. 7th International Conference on Software Engineering Advances. Lisbon, Portugal, November 2012. AR 30%.

[Zha12b] *A three-level component model in component-based software development.* H. Y. ZHANG, L. ZHANG, C. URTADO, S. VAUTTIER and M. HUCHARD. 11th ACM International Conference on Generative Programming and Component Engineering. Dresden, Germany, September 2012.

[Zha12a] *Dedal-CDL: Modeling first-class architectural changes in Dedal*. H. Y. ZHANG, C. URTADO and S. VAUTTIER, L. ZHANG, M. HUCHARD and B. COULETTE. 10th Working IEEE/IFIP Conference

on Software Architecture & 6th European Conference on Software Architecture. Helsinki, Finland, August 2012.

[Cou12] *A formal support for incremental behavior specification in agile development*. A-L. COURBIS, T. LAMBOLAIS, H-V. LUONG, T-L. PHAN, C. URTADO and S. VAUTTIER. 24th International Conference on Software Engineering and Knowledge Engineering. Redwood City, USA, July 2012.

[Fau11b] *A service component framework for multi-user scenario management in ubiquitous environments.* M. FAURE, L. FABRESSE, M. HUCHARD, C. URTADO and S. VAUTTIER. 6th International Conference on Software Engineering Advances. Barcelona, Spain, October 2011. AR 30%.

[Fau11a] *User-defined scenarios in ubiquitous environments: creation, execution control and sharing*. M. FAURE, L. FABRESSE, M. HUCHARD, C. URTADO and S. VAUTTIER. 23rd International Conference on Software Engineering and Knowledge Engineering. Miami, USA, July 2011. AR 31%. 302-307.

[Azm11] *Backing composite web services using Formal Concept Analysis*. Z. AZMEH, F. HAMOUI, M. HUCHARD, C. TIBERMACINE, C. URTADO and S. VAUTTIER. 9th International Conference on Formal Concept Analysis. Nicosia, Cyprus, May 2011. AR 33%.

[Zha10c] *Architecture-centric component-based development needs a three-level ADL*. H. Y. ZHANG, C. URTADO and S. VAUTTIER. 4th European Conference on Software Architecture. Copenhagen, Denmark, August 2010. AR 25,3%. 295-310.

[Ham10b] *SAASHA: a Self-Adaptable Agent System for Home Automation*. F. HAMOUI, M. HUCHARD, C. URTADO and S. VAUTTIER. 36th EUROMICRO Conference on Software Engineering and Advanced Applications. Lille, France, September 2010.

[Zha10b] *Architecture-centric development and evolution processes for component-based software.* H. Y. ZHANG, C. URTADO and S. VAUTTIER. 22nd International Conference on Software Engineering and Knowledge Engineering. Redwood City, USA, July 2010. AR 33%.

[Azm10] *Using concept lattices to support web service compositions with backup services*. Z. AZMEH, M. HUCHARD, C. TIBERMACINE, C. URTADO and S. VAUTTIER. 5th International Conference on Internet and Web Applications and Services. Barcelona, Spain, May 2010.

[Abo09] *Automated architectural component classification using concept lattices.* N. A. ABOUD, G. AREVALO, J-R. FALLERI, M. HUCHARD, C. TIBERMACINE, C. URTADO and S. VAUTTIER. Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. Cambridge, UK, September 2009. AR 27,4%. 21-31.

[Ham09] *Specification of a component-based domotic system to support user-defined scenarios*. F. HAMOUI, M. HUCHARD, C. URTADO and S. VAUTTIER. 21st International Conference on Software Engineering and Knowledge Engineering. Boston, USA, July 2009. AR 38%. 597-602.

[Zha09] *Connector-driven process for the gradual evolution of component-based software.* H. Y. ZHANG, C. URTADO and S. VAUTTIER. 20th IEEE Australian Software Engineering Conference. Gold Coast, Australia, April 2009. AR 45,6%. 246-255.

[Zha08] *Connector-driven gradual and dynamic software assembly evolution*. H. Y. ZHANG, C. URTADO and S. VAUTTIER. IEEE International Conference on Innovation in Software Engineering. Vienna, Austria, December 2008. 345-350.

[Azm08] *WSPAB: A tool for automatic classification & selection of web services using formal concept analysis*. Z. AZMEH, M. HUCHARD, C. TIBERMACINE, C. URTADO and S. VAUTTIER. 6th IEEE European Conference on Web Services. Dublin, Ireland, November 2008. 31-40.

[Are07] *Precalculating component interface compatibility using FCA*. G. AREVALO, N. DESNOS, M. HUCHARD, C. URTADO and S. VAUTTIER. 5th International Conference on Concept Lattices and their Applications. Montpellier, France, October 2007. 241-252.

[Des07] *Automated and unanticipated flexible component substitution.* N. DESNOS, M. HUCHARD, C. URTADO, S. VAUTTIER, and G. TREMBLAY. 10th ACM SIGSOFT Symposium on Component-Based Software Engineering. Medford, USA, July 2007. AR 18,6%. 33-48.

[Ous97] *Managing the Global Behavior of Composite Objects*, C. Oussalah M. Magnan and S. Vauttier. 8th Int. Conf. on Database and Expert Systems Applications, DEXA'97, Toulouse, Septembre 1997.

[Mag97] *From Specification to Management of Composite Object Behavior*, M. Magnan, S. Vauttier and C. Oussalah. International Conference on Theory of Object Oriented Languages and Systems, TOOLS USA'97, Santa Barbara, July 97.

[Vau97a] *Behavior in Aggregation Hierarchies*, S. Vauttier, M. Magnan and C. Oussalah. 13th International Conference on Advanced Science and Technology, ICAST'97, Chicago, Avril 1997.

- **Workshops internationaux**

[Ayr11] *Hydroguard: Autonomous and automated system for monitoring, management of coastal risks and water resources.* P-A. AYRAL, C. GONZALEZ, C. LEQUETTE, I. LOVRIC, D. SALZE, C. URTADO and S. VAUTTIER. Workshop of the International Emergency Management Society, Nîmes, June 2011. 11 pages.

[Fau10] *Towards scenario creation by service composition in ubiquitous environments*. M. FAURE, L. FABRESSE, M. HUCHARD, C. URTADO and S. VAUTTIER. 9th BElgian-NEtherlands software eVOLution Seminar, S. Ducasse, L. Duchien and L. Seinturier editors. Lille, France, December 2010.

[Don08] *Specification of an exception handling system for a replicated agent environment*. C. DONY, C. TIBERMACINE, C. URTADO and S. VAUTTIER. 4th ACM International Workshop on Exception Handling, Atlanta, GA, USA, November 2008. 24-31.

[Des06b] Automating the building of software component architectures. N. DESNOS, S. VAUTTIER, C. URTADO and M. HUCHARD. 3rd European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications. Nantes, France, September 2006. AR 34%. 228-235.

[Cam04] *Distributed exception handling: ideas, lessons and issues with recent exception handling systems*. A. CAMPEAS, C. DONY, C. URTADO and S. VAUTTIER. International workshop on Rapid Integration of Software Engineering Techniques. Luxembourg, November 2004. 82-92.

[Sou03b] *Exception handling in component-based systems: a first study*. F. SOUCHON, C. URTADO, S. VAUTTIER and C. DONY. Exception Handling in Object-Oriented Systems Workshop at ECOOP 2003. Darmstadt, Germany, July 2003. 84-91

[Sou03a] *A proposition for exception handling in multi-agent systems*. F. SOUCHON, C. DONY, C. URTADO and S. VAUTTIER. 2nd International workshop on Software Engineering for Large-Scale Multi-Agent Systems at ICSE 2003. Portland, Oregon, May 2003. 136-143.

- **Conférences nationales**

[Mok14a] *Modélisation et vérification formelles en B des architectures logicielles à trois niveaux.* A. MOKNI, M. HUCHARD, C. URTADO, S. VAUTTIER and H. Y. ZHANG. 3ème Conférence en IngénieriE du Logiciel. Paris, Juin 2014. 71-77.

[Are10] *Component and service farms*. G. AREVALO, Z. AZMEH, M. HUCHARD, C. TIBERMACINE, C. URTADO and S. VAUTTIER. 2ème journées nationales du GDR GPL – Défis pour le Génie de la Programmaion et du Logiciel, pages 281-284. Pau, France, Mars 2010.

[Zha10a] *Dedal : un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants.* H. Y. ZHANG, C. URTADO and S. VAUTTIER. 4ème Conférence francophone sur les Architectures Logicielles. Pau, France, Mars 2010. AR 48%. 15-27.

[Ham10a] *Un système d'agents à base de composants pour les environnements domotiques*. F. HAMOUI, M. HUCHARD, C. URTADO and S. VAUTTIER. 16ème conférence francophone sur les Langages et Modèles à Objets. Pau, France, Mars 2010. AR 41%. 35-49.

[Are08a] *Construction dynamique d'annuaires de composants par classification de services.* G. AREVALO, N. DESNOS, M. HUCHARD, C. URTADO and S. VAUTTIER. 14ème conférence francophone sur les Langages et Modèles à Objets. Montréal, Canada, Mars 2008. AR 49%.

[Des06a] *Assistance à l'architecte pour la construction d'architectures à base de composants.* N. DESNOS, C. URTADO, S. VAUTTIER and M. HUCHARD. 12ème conférence francophone sur les Langages et Modèles à Objets. Nîmes, France, Mars 2006. AR 37,1%. 37-52.

[Sou04b] *Fiabilité des applications multi-agents : le système de gestion d'exceptions SaGE.* F. SOUCHON, S. VAUTTIER, C. URTADO and C. DONY. Journées Francophones sur les Systèmes Multi-Agents 2004. Paris, France, Novembre 2004. AR 17,9%. 121-134.

[Vau03] *La relation d'association dans les approches dirigées par les modèles : besoins et usages pratiques*. S. VAUTTIER, C. URTADO et E. MENDIZABAL. Colloque Agents logiciels, Coopération, Apprentissage et Activité humaine. Bayonne, France, Septembre 2003. 151-163.

[Vau00] *Improving statechart composition and reuse in UML*. S. VAUTTIER and C. URTADO. Conférence NîmesTIC 2000. Nîmes, France, September 2000. 287-296.

[Vau99b] *Une nouvelle approche de la spécification du comportement des objets composites en UML.* S. Vauttier, C. Oussalah et M. Magnan, 4ème conférence francophone sur les Langages et Modèles à Objets, Villefranche sur mer, Janvier 1999.

[Vau97c] *Extension d'UML à la Prise en Compte du Comportement d'Objets Composites*, S. Vauttier, C. Oussalah et M. Magnan. Journées du GDR Programmation, Rennes, Octobre 1997.

[Vau97b] *Comportement et Objets Composites*, S. Vauttier, M. Magnan et C. Oussalah 15ème Congrès d'INFormatique des ORganisations et Systèmes d'Information et de Décision, INFORSID'97, Toulouse, Juin 1997.

[Ous97] *Modélisation du comportement des objets composites*, C. Oussalah, M. Magnan et S. Vauttier 3rd International Symposium on Programming and Systems, ISPS '97, Alger, Avril 1997.

- **Chapitres dans ouvrages**

[Don06] *Exception handling and asynchronous active objects: Issues and proposal*. C. DONY, C. URTADO and S. VAUTTIER. Advanced Topics in Exception Handling Techniques. C. Dony, J. Knudsen, A. Romanovsky and A. Tripathi editors. LNCS 4119. Springer 2006. 81-100.

[Sou04a] *Improving exception handling in multi-agent systems*. F. SOUCHON, C. DONY, C. URTADO and S. VAUTTIER. Software engineering for multi-agent systems II, Research issues and practical applications. C. J. Pereira de Lucena, A. F. Garcia, A. B. Romanovsky, J. Castro and P. S. C. Alencar editors. LNCS 2940. Springer 2004. 167-188.

- **Edition d'actes**

[Rou06] *Actes de la 12ème conférence francophone sur les Langages et Modèles à Objets*. R. ROUSSEAU, C. URTADO and S. VAUTTIER Editors. Hermès, Nîmes, pp232 France, Mars 2006.

[Boi06] *Actes de la deuxième Journée Multi-Agents et Composants.* O. BOISSIER, N. BOURAQADI, J.-C. ROYER, D. SERIAI, C. URTADO, S. VAUTTIER and L. VERCOUTER Editors. Nîmes, France, Mars 2006.

[Boi04] *Actes de la première Journée Multi-Agents et Composants*. O. BOISSIER, N. BOURAQADI, J.-C. ROYER, D. SERIAI, C. URTADO, S. VAUTTIER and L. VERCOUTER Editors. 68 pages, Paris, France, Novembre 2004.

- **Manuscrit de thèse**

[Vau99c] *Une Etude du Comportement des Objets Composites*. Thèse de doctorat Université des Sciences et Techniques du Languedoc de Montpellier II, Montpellier, Décembre 99.

Annexe B

A formal approach for managing component-based architecture evolution

A formal approach for managing component-based architecture evolution

Abderrahman Mokni [a], Christelle Urtado[a], Sylvain Vauttier[a], Marianne Huchard[b], Huaxi Yulin Zhang[c]

[a] *LGI2P, Ecole Nationale Supérieure des Mines Alès, Nîmes, France*
[b] *LIRMM, CNRS and Université de Montpellier, Montpellier, France*
[c] *Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France*

Abstract

Software architectures are subject to several types of change during the software lifecycle (*e.g.* adding requirements, correcting bugs, enhancing performance). The variety of these changes makes architecture evolution management complex because all architecture descriptions must remain consistent after change. To do so, whatever part of the architectural description they affect, the effects of change have to be propagated to the other parts. The goal of this paper is to provide support for evolving component-based architectures at multiple abstraction levels. Architecture descriptions follow an architectural model named Dedal, the three description levels of which correspond to the three main development steps — specification, implementation and deployment. This paper formalizes an evolution management model that generates evolution plans according to a given architecture change request, thus preserving consistency of architecture descriptions and coherence between them. The approach is implemented as an Eclipse-based tool and validated with three evolution scenarios of a Home Automation Software example.

Keywords: architecture evolution, architecture analysis, evolution rules, formal models, abstraction level, evolution plans, MDE.

Email addresses: Abderrahman.Mokni@mines-ales.fr (Abderrahman Mokni), Christelle.Urtado@mines-ales.fr (Christelle Urtado), Sylvain.Vauttier@mines-ales.fr (Sylvain Vauttier), Marianne.Huchard@lirmm.fr (Marianne Huchard), yulin.zhang@u-picardie.fr (Huaxi Yulin Zhang)

1. Introduction

Component-based software development (CBSD) promotes a reuse-based approach to defining, implementing and composing loosely coupled independent software components into whole software systems [1]. While component reuse is crucial to shorten large-scale software systems development time, handling evolution in such processes is a significant issue [2]. Indeed, software systems have to evolve to extend their functionalities, correct bugs, improve performance and quality, or adapt to their environment. While unavoidable, software changes may engender several inconsistencies and system dysfunction if not analyzed and handled carefully. In turn, an ill-mastered evolution engenders software degradation, the loss of its evolvability and then its phase-out [3].

A famous problem of software evolution is software architecture erosion [4, 5]. It arises when modifications of the software implementation violate the design principles captured by its architecture. To increase confidence in reuse-centered, component-based software systems, all architecture descriptions must remain consistent and coherent with each other after every change.

While a lot of work has been dedicated to architectural modeling and evolution, there still is a lack of means and techniques to tackle architectural inconsistencies, and erosion in particular. Indeed, most existing approaches to architecture evolution hardly support the whole life-cycle of component-based software and only enable evolution of early stage models by propagating change impact to runtime models while evolution of runtime models are not fully dealt with, thus increasing the risks of architecture erosion.

This paper proposes an approach and its implementation to automatically manage component-based architecture evolution at multiple abstraction levels in a manner that preserves architecture consistency and coherence all along the software lifecycle. The approach is based on the Dedal [6, 7] architectural model that explicitly models architectures at three abstraction levels, each corresponding to one of the three major steps of CBSD – specification, implementation and deployment, thus granting a full evolution management process. Given a change request at any abstraction level, it transforms Dedal models into B formal models to analyze the requested change and generates an evolution plan that guarantees the consistency of architecture descriptions and the coherence between them. The proposed approach is centered on a formal evolution management model that includes the generated B mod-

els, the architecture properties to preserve and a set of evolution rules. It is implemented as an Eclipse-based tool that generates B models from diagrammatic Dedal models and uses our specific solver to resolve architecture evolution. The overall approach is illustrated with a Home Automation Software case-study.

The remainder of this paper outlines as follows: Section 2 presents the background of this work. Section 3 presents our proposal to tackle multi-level architecture evolution (*i.e.* the evolution of architecture definitions composed of multiple description levels) while Section 4 presents the implemented tool and experiments on three evolution scenarios. Section 5 discusses related work and finally, Section 6 concludes the paper and discusses future work.

2. Background

Our approach combines the use of Dedal to model software architectures and B to support automated analysis and verification. This section briefly introduces these languages.

2.1. The Dedal architecture model

2.1.1. Component-based software development by reuse

CBSD follows the *reuse-in-the-large* principle. Reusing existing (off-the-shelf) software components [8] therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [1]. Figure 1 illustrates our vision of such a development process which is classically divided in two:

- the component development process (referred to as software component development for reuse), which will not be detailed in the sequel. This development process produces components that are stored in repositories for later use by the software development process.
- the software development process (referred to as software development by component reuse) that describes how previously developed software components can be used for software development (and how this reuse impacts the way software is built).

Dedal is a novel architectural model and ADL [6, 7] that targets reuse-centered development. It covers the whole software development by component reuse life-cycle. The main idea of Dedal is to build a concrete architecture composed of stored and indexed components that are found in a

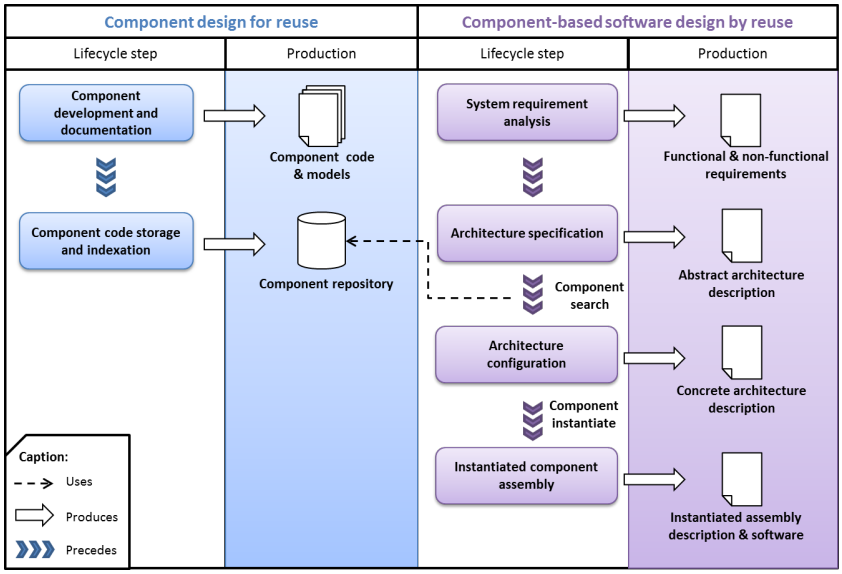


Figure 1: Dedal reuse-centered development process [7]

component repository as candidates to satisfy the design decisions specified in an intended architecture specification. The resulting concrete architecture can then be instantiated and deployed in multiple contexts. Therefore, Dedal proposes a three-step approach for specifying, implementing and deploying software architectures.

2.1.2. Dedal abstraction levels

To illustrate the concepts of Dedal, we propose to model a home automation software (HAS) that manages comfort scenarios, which automatically controls buildings' lighting and heating depending on time and ambient temperature. For this purpose, we propose an architecture with an orchestrator component that interacts with the appropriate devices to implement the desired scenario.

The *abstract architecture specification* is the first level of software architecture descriptions. It is abstract: it represents the architecture as imagined by the architect to meet the requirements of the future software. In Dedal, the architecture specification is composed of component roles, their connections and the expected global behavior. Component roles are abstract and partial component type specifications. Consequently, the provided interfaces of each role are to be connected to compatible required interfaces. Compo-

nent roles are identified by the architect in order to search for and select corresponding concrete components in the next step. Figure 2-a shows a possible HAS architecture specification. In this specification, five component roles are identified. A component playing the *HomeOrchestrator* role controls four components playing the *Light*, *Time*, *Thermometer* and *CoolerHeater* roles.

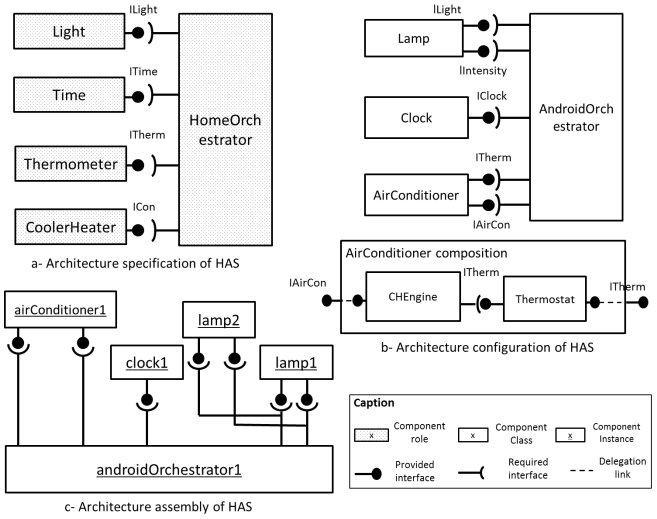


Figure 2: Architecture specification, configuration and assembly of the HAS

The *concrete architecture configuration* is an implementation view of software architectures. It results from the selection of existing component classes in component repositories. Thus, an architecture configuration lists and connects the concrete component classes that compose a specific version of the software. In Dedal, component classes can either be primitive or composite. A *primitive component class* encapsulates executable code. A *composite component class* encapsulates an inner architecture configuration (*i.e.* a set of connected component classes which may, in turn, be primitive or composite). A composite component class exposes a set of interfaces corresponding to the unconnected interfaces of its inner components. Figure 2-b shows a possible architecture configuration for the HAS example as well as an example of an *AirConditioner* composite component and its inner configuration. As illustrated in this example, a single component class may realize several roles from the architecture specification as with the *AirConditioner* component

class, which realizes both the *Thermometer* and *CoolerHeater* roles. Conversely, a component class may provide more services than those listed in (its role in) the architecture specification as with the *Lamp* component class which provides an extra service to control the intensity of light. These extra interfaces may be left unconnected.

The *instantiated architecture assembly* describes software at runtime and holds information about its internal state. The architecture assembly models an instantiation of its architecture configuration. It lists the instances of the component and connector classes that compose the deployed architecture at runtime and their assembly constraints (such as the maximum number of connected instances). *Component instances* document how the component classes from an architecture configuration are instantiated in the deployed software. Each component instance has an initial and a current state defined by a list of valued attributes. Figure 2-c shows an instantiated architecture assembly for the HAS example.

2.2. The B modeling language

B [9, 10] is a formal modeling language and a proof-based development method for software systems. The principle of such method is to start from a very abstract model of the system and then gradually refine it. Initially designed by Abrial in 1985 to specify critical systems, B was rapidly adopted by industry and used in many case studies such as the METEOR project [11] for controlling train traffic and the PCI protocol [12]. B is also widely used and studied in academia, mainly as a formal modeling language for verification, validation and model-checking.

2.2.1. Expressiveness and semantics

B is based on Zermelo-Fraenkel (ZF) set theory and first order logic language. The B notation is very similar to mathematical language and includes all standard logical connectors (*e.g.* $\wedge, \vee, \Rightarrow$), set-theoretic operations (*e.g.* $\in, \cup$), closure and specific relations like injective ($\rightarrowtail$), surjective ($\twoheadrightarrow$) and bijective ($\rightarrowtail\!\!\!\!\twoheadrightarrow$) functions. B also supports sequences and the basic boolean ($BOOL$), integer ($INTEGER$) and natural (NAT) types.

B specifications are composed of abstract machines similar to modules (*cf.* Figure 3). They are defined independently and can be reused as modules and refined to obtain more concrete models. An abstract machine is divided into a declarative part and a dynamic part. The declarative part

contains the declaration of sets (*SETS*), constants (*CONSTANTS*), variables (*VARIABLES*) which represent the state of the machine and invariant properties (*INVARIANT*) related to variables. Optionally, it is also possible to set definitions (*DEFINITIONS*) (like macros). Definitions are useful to define extensive sets and parametrized predicates and can be reused by invariants and operations. The dynamic part contains the initialization (*INITIALISATION*) of the machine as well as operations (*OPERATIONS*) over the state (variables) of the machine. The behavior of operations is explicitly defined in B using various constructs such as preconditions (*PRE P THEN S END*), bounded choice (*CHOICE S1 OR S2*) or non-determinism (*ANY v WHERE P THEN S END*). Post-conditions are expressed by substitutions that state the new assignments of the involved variables. Output variables may also be defined as values returned by operations.

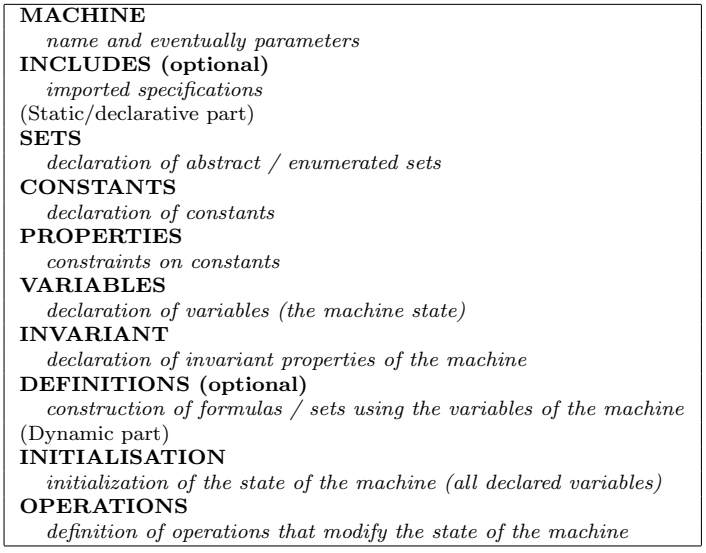


Figure 3: Structure of an abstract B machine

2.2.2. Tool support for B

B is supported by powerful tools like AtelierB [13], BToolkit [14] and the more recent Bware platform [15]. These tools focus on theorem-proving but they do not enable model-checking. ProB [16] was designed for this purpose. It is a model checker and animator for B models. It automatically generates counterexamples for given assertions by exhaustively exploring the model

(using state space exploration techniques). It also simulates the execution of operations on a given subset of the model and generates traces leading to some desired state. An API is also provided for developers to integrate the features of ProB in their tools.

2.3. Motivation and contribution

Component reuse helps decrease large-scale software systems time-to-market. Handling the evolution in such component-based software prevents architecture erosion and has long been identified and still remains an important thus difficult task [17, 18]. To tackle this issue, this paper proposes an approach to manage the evolution of component-based software architectures based on the three-level Dedal architecture model.

Dedal is tailored for reuse [6, 7] and provides as an original feature its three architecture definition levels. Indeed, specifications are the cornerstone of the concrete component search that is performed on component repositories to design, by reuse, the implementation of architectures. Along with Dedal configurations and assemblies, Dedal architecture definitions keep track of all the design decisions taken during the development process. This information is very useful to control evolution and evaluate its impact on the intentions of the architects. This is why Dedal is a choice ADL for architecture-based software evolution management.

The evolution process proposed here is driven by an evolution management model that captures changes initiated at any abstraction level, controls their impact to preserve / restore consistency and propagates them to other levels to maintain global coherence. This model is based on the B formal language which provides a rich and rigorous notation to formalize the architectural concepts and express properties over them. It supports automated analysis and model-checking thanks to the ProB tool.

In previous work [19, 20], we specified Dedal models using the B modeling language and proposed an evolution management model to enable the simulation, analysis and validation of evolution scenarios at any abstraction level using ProB. At that time, evolution was not yet automated since models were specified and evolved manually and separately. In the remainder, our approach integrating both Dedal and B to automatically manage component-based architecture evolution is presented. The automated Dedal to B transformation as well as a problem-specific B solver built on top of the ProB tool are the cornerstones of the contribution of this paper.

Using our problem-specific solver enables the automatic generation of evolution plans (sequences of change operations) to leverage the impact of a change request in a problem-specific manner and maintain the architecture descriptions coherent after change. The feasibility of our approach is demonstrated by experimenting on three evolution scenarios that each addresses change in a different abstraction level.

3. The formal evolution approach

This section presents our approach to formally handle the evolution of multi-level component-based architecture descriptions produced during software development. Its key idea is to use a B solver to automatically generate evolution plans that correspond to intended changes (*cf.* Figure 4).

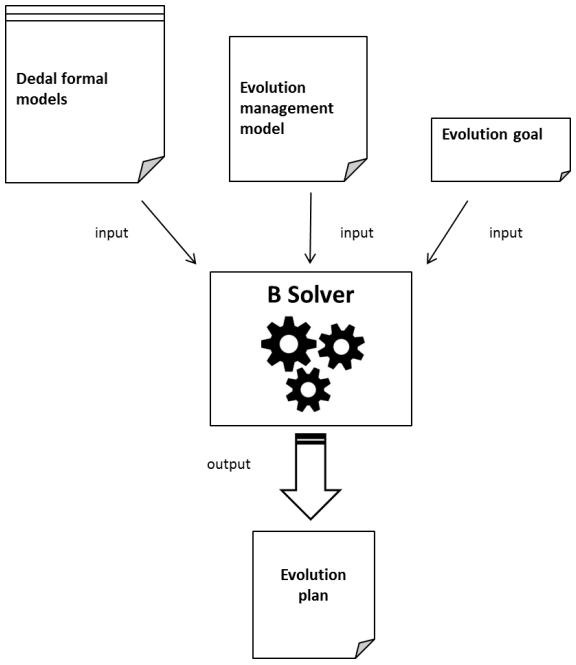


Figure 4: Evolution management approach

Given a model in an initial state, a set of state transition rules and a goal state, a B solver finds sequences of rules that reach the goal state or proves that the goal state cannot be reached (when it does not run out of time or resources because of high computational complexity).

A first requirement is thus to transform the Dedal models produced during development into B models that can be used as an input for the solver. The principles of this transformation are detailed in Section 3.1. Architecture evolution operations along with validation properties must also be expressed as a set of rules. The resulting *Evolution Management Model* is presented in Section 3.2. Finally, initiated architecture changes must be described as goal states, as explained in Section 3.3. With these inputs, a B solver can then find an evolution plan (a sequence of rules) that achieves the intended change (reaches the goal state) while preserving the coherence of the architecture definition (enforcing properties), as presented in Section 3.4.

3.1. Dedal to Formal Dedal transformation

Dedal models need to be translated into B models, so that a B solver can calculate modifications and evaluate properties on the resulting formal architecture descriptions. Defining this transformation amounts to formalize in B the concepts of the Dedal meta-model (*cf.* Figure 6). This way, any instance of the Dedal meta-model can be transformed into an equivalent instance of the Formal Dedal meta-model (*cf.* Figure 5).

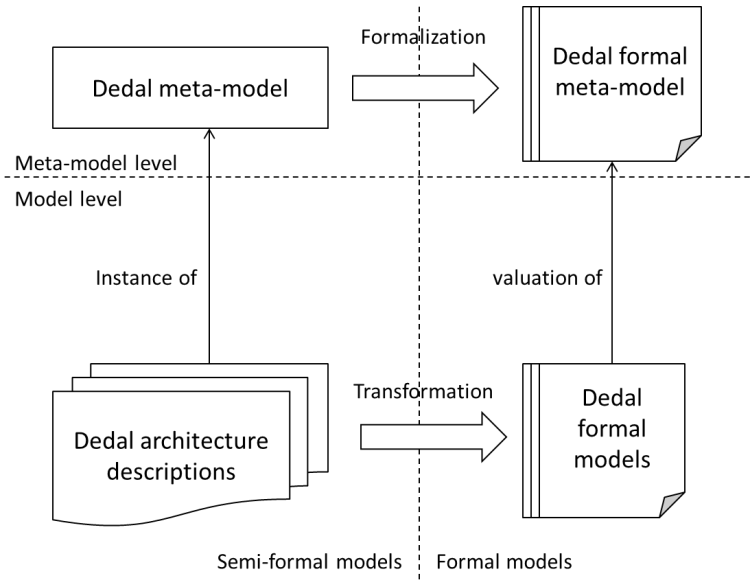


Figure 5: Dedal to Formal Dedal transformation

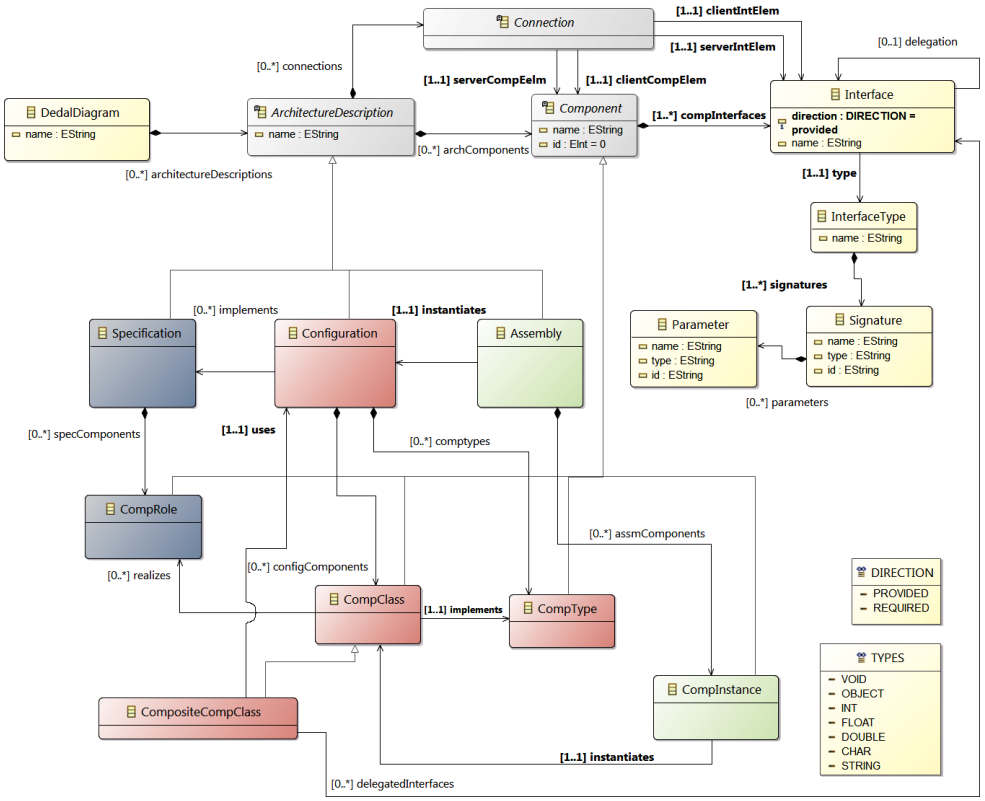


Figure 6: Dedal meta-model

A meta-class is usually mapped to a B variable typed by an abstract B set while an association relation is translated into a B relation. For instance, Figure 7 presents the formalizaton of the *Component* and *Interface* meta-classes and their *compInterfaces* association.

SETS
$COMPS; INTERFACES$
VARIABLES
$component, interface, comp\_interfaces$
INVARIANT
$component \subseteq COMPS \wedge$
$interface \subseteq INTERFACES \wedge$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}_1(interface)$

Figure 7: Formalization of meta-classes and associations in B

The *Component* and *Interface* meta-classes are respectively mapped to the *component* and *interface* variables and typed with the *COMPS* and *INTERFACES* abstract sets. Their *compInterfaces* association holding a one-to-many relation is translated into an injective function between the *component* variable and a non-empty set of interfaces: $\mathcal{P}_1(interface)$.

The whole Dedal meta-model formalization results in four main B machines (extracts of which are shown in Figure 8).

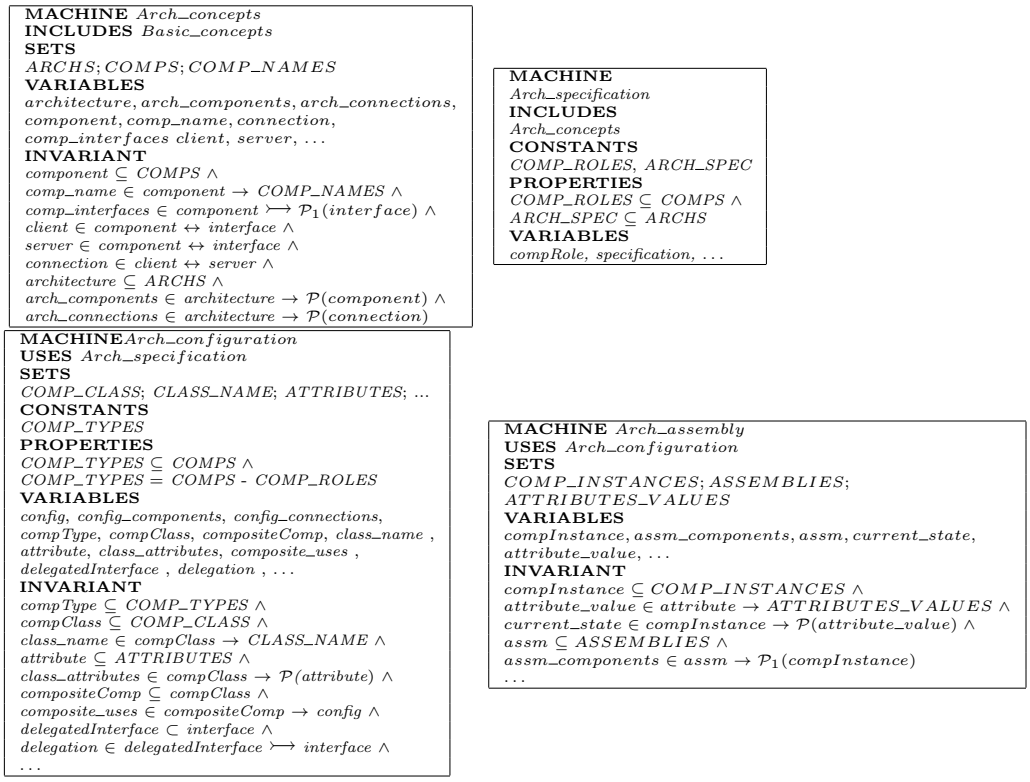


Figure 8: Overview of the Dedal formal meta-model

A generic *Arch_concepts* machine helps define the three specific *Arch_specification*, *Arch_configuration* and *Arch_assembly* machines that each correspond to one of the three architecture description levels of Dedal. *Arch_concepts* covers the generic concepts of a software architecture (corresponding to the abstract *Component*, *Connection*, and *ArchitectureDescription* meta-classes). It includes an inner *Basic_concepts* machine that contains definitions for the finer-grained architectural elements like *Interface*, *InterfaceType*, *Signature*

or *Parameter* meta-classes.

These generic definitions are reused in the three specific machines. For instance, in the *Arch_specification* machine, component roles are defined as a subset of components: $COMP\_ROLES \subseteq COMPS \wedge compRole \subseteq COMP\_ROLES$.

This corresponds to the inheritance relation between the *Component* and *CompRole* meta-classes. Consequently, all relations defined for the *component* set (such as *comp_interfaces*) also stand for the *compRole* set.

The abstract B machines define a formal meta-model that can be instantiated (concrete values are given to their variables) in order to generate a Formal Dedal model. The latter is then used as an input for the B solver.

3.2. The evolution management model

The evolution management model is composed of generic evolution rules that are used by the solver to find evolution plans satisfying given evolution goals. It consists in a B machine that defines the rules and properties that respectively enable the simulation and validation of architecture evolution at the three abstraction levels (*cf.* Figure 9).

MACHINE
EvolutionManager
INCLUDES
Arch_specification, *Arch_configuration*, *Arch_assembly*
SETS
/*Enumerated set to indicate the level of change*/
$CHANGE\_LEVEL = \{eLevel, specLevel, configLevel, asmLevel\}$
VARIABLES
/*Variable to control the level of change*/
changeLevel, ...
DEFINITIONS
/*Consistency and coherence properties*/
...
$global\_consistency == spec\_consistency \wedge config\_consistency \wedge assm\_consistency$
$global\_coherence == specConfigCoherence \wedge configAssmCoherence$
/*GOAL is the predicate given to the solver to find an evolution plan satisfying it*/
$GOAL == global\_consistency \wedge global\_coherence \wedge \ldots$
INITIALISATION
/*Initialization is used to set the initial level of change and
the initiated change*/
...
OPERATIONS
/*Initialization operations*/
...
/*Evolution rules(control the architecture manipulation operations) ***/**
...
END

Figure 9: The *EvolutionManager* machine

Its main elements are detailed in the following subsections.

3.2.1. Evolution rules

Evolution rules are operations that control the access and the impact of architecture manipulation operations in order to manage evolution and generate consistent evolution plans (*cf.* Figure 10). Each evolution rule embeds a corresponding architecture manipulation operation that handles the actual modification of the model, not taking into account the context of the current evolution plan.

output ← evolutionRuleName(targetArchitecture, artifacts) =
PRE
$initialization = true \wedge$
$changeLevel = currentChangeLevel \wedge$
$artifacts \notin addedArtifacts \cup deletedArtifacts \wedge$
$manipulationOperationPrecondition$
THEN
/* execute manipulationOperationName(targetArchitecture, artifacts),
update the sets of added artifacts and deleted artifacts,
set the value of output parameters */
END

Figure 10: Schema of an evolution rule

The evolution rule preconditions act as a primary filter for model manipulation operations. Initialization preconditions check that all the model initialization operations have completed before starting calculating evolution plans. Initialization includes calculating and checking relations between architecture elements, such as compatibility and substitution between components and interfaces. Change level preconditions restrict access to the operations related to the current level of change (evolution is managed on one level at a time). History preconditions prevent operations that may generate cycles and then decrease the efficiency of the solver. For instance, deleting and adding the same artifact several times is unnecessary during an evolution process. Similarly, removing an added artifact results in a null operation that may be avoided. History consists of two sets: one for added artifacts and the other for deleted ones. Evolution rules also inform the solver about the artifacts that have to be manipulated after the last executed change operation. This information is used as a heuristic to increase the efficiency of the solver. Heuristics are further discussed in Section 4.1.2.

Figure 11 gives the definition of the evolution rule that controls the role addition operation. This rule is enabled when evolution is handled at the

specification level, after initialization, provided that the role has not yet been added or previously removed. If so, the precondition of the role addition operation is checked and, when it is verified, the operation is executed. Finally, the set of added component roles (*addedRoles*) is updated and the output is set to the added component role (*newRole*).

output ← **mng_addRole**(*spec*, *newRole*) =
PRE
/*Initialization precondition*/
    initialisation = **TRUE** ∧
/*Change level precondition*/
    changeLevel = *specLevel* ∧
/*Precondition to avoid cycles (inverse operation)*/
    newRole ∉ (*deletedRoles* ∪ *addedRoles*) ∧
/*Precondition of the role addition operation*/
    roleAdditionPrecondition
THEN
/*Access to role addition operation*/
    addRole(*spec*, *newRole*) ||
    addedRoles := *addedRoles* ∪ {*newRole*} ||
    outpout := *newRole*
END;

Figure 11: The component role addition evolution rule

3.2.2. Model manipulation operations

A model manipulation operation is an operation that changes a target software architecture by the deletion, addition or substitution of one of its elements (components and connections). They are composed of three parts:

- the operation signature that defines the operation name and its arguments,
- preconditions that are related to the architectural model (*e.g.* a precondition that checks if substitutability between two component classes holds),
- actions (called substitutions in B) that update a set of variables related to the architectural model (*e.g.* the set of components of the architecture).

Architecture specification evolution. Evolving an architecture specification is usually a response to a new software requirement. For instance, the architect may need to add new functionalities to the system and hence add some new roles to the specification. Moreover, a specification may also be modified during the change propagation process to preserve coherence and keep an up-to-date specification description of the system that may be implemented in several ways. The proposed manipulation operations related to the specification level are the addition, deletion and substitution of a component role and the addition and deletion of connections. Figure 12 presents the definition of the role addition operation as an example of an architecture specification manipulation operation. Its precondition first checks that arguments are soundly typed and then that the chosen role does not already belong to the architecture specification and will not name clash. Its actions update the set of component roles of the architecture specification, along with the sets of connected provided and required interfaces (respectively *spec_components*, *spec_servers* and *spec_clients*). Indeed, as only effectively used elements are defined at specification level, every interface must be connected.

addRole(*spec*, *newRole*) =
PRE
$spec \in arch\_spec \wedge newRole \in compRole \wedge newRole \notin spec\_components(spec) \wedge$
/* *spec* does not contain a role with the same name*/
$\forall\ cr.(cr \in compRole \wedge cr \in spec\_components(spec)$
$\Rightarrow comp\_name(cr) \neq comp\_name(newRole))$
THEN
    spec_servers(*spec*) := *spec_servers*(*spec*) ∪ servers(newRole) ||
    spec_clients(*spec*) := *spec_clients*(*spec*) ∪ clients(newRole) ||
    spec_components(*spec*) := *spec_components*(*spec*) ∪ {*newRole*}
END;

Figure 12: The component role addition manipulation operation

Architecture configuration evolution. Change can be initiated at the configuration level, for example when new versions of software component classes are released or when component classes are not available anymore. Otherwise, an implementation may also be impacted by change propagation either from the specification level, in response to new requirements, or from the assembly level, in response to a dynamic change of the system. Indeed, a configuration may be instantiated several times and deployed in multiple contexts. Figure 13 presents the component class substitution operation as an example of an architecture configuration manipulation operation.

replaceClass(*config*, *oldClass*, *newClass*) =
PRE
oldClass ∈ *compClass* ∧ *newClass* ∈ *compClass* ∧ *config* ∈ *configuration* ∧
oldClass ∈ *config_components*(*config*) ∧
/* The old component class can be substituted for the new one
(verified by the component substitution rule)*/
newClass ∉ *config_components*(*config*) ∧ (*oldClass*, *newClass*) ∈ *class_substitution*
THEN
config_components(*config*) := (*config_components*(*config*) - {*oldClass*}) ∪ {*newClass*}
END

Figure 13: The component class substitution manipulation operation

Besides checking the type of the arguments, its precondition verifies that the new component class does not already belong to the configuration and can be a substitute for the old component class (using the relations calculated during initialization). When the precondition is verified, the set of component classes composing the configuration is updated. As compared to the role addition operation presented in previous section, there is no need to update the sets of client and server interfaces (connected required and provided interfaces) here, as substitution must preserve the connections of the replaced component class (*see* § 3.2.3 for deeper insight about substitution rules).

Architecture assembly evolution. Since the architecture assembly represents the software at runtime, managing the assembly level relates to dynamic evolution issues. Indeed, some software systems have to be self-adaptive to keep providing their functions despite environmental changes (*e.g.* lack of resources, failures, user requests). Dealing with unanticipated changes is one of the most important issues in software evolution. This issue is handled by the evolution manager which monitors the execution state of the software through its corresponding formal model. It then triggers the assembly evolution rules to restore consistency and coherence when needed. The assembly manipulation operations include component instance addition, component instance removal, component instance substitution and component instance connection / disconnection. Figure 14 gives the definition of the component instance addition as an example of an assembly manipulation operation. After checking the types of the arguments, the precondition verifies that the instance corresponds to the chosen component class, that it does not already belong to the assembly and that another instance of the class can be added in the assembly. It also verifies that the chosen initial state is valid.

deployInstance(asm, $inst$, $class$, $state$) =
PRE
$asm \in assembly \wedge class \in compClass \wedge$
/* The instance is a valid instantiation of the chosen component class*/
$inst \in compInstance \wedge class = comp\_instantiates(inst) \wedge inst \notin assm\_components(asm) \wedge$
/* The state given to the instance is a valid value assignment of its attributes of the instantiated component class*/
$state \in \mathcal{P}\ (attribute\_value) \wedge$ **card**$(state) =$ **card**$(class\_attributes(class)) \wedge$
/* The maximum number of allowed instances of the given component class is not already reached*/
$nb\_instances(class) < max\_instances(class)$
THEN
/*initial and current state initialization*/
initial_state$(inst) := state \ ||$
current_state$(inst) := state \ ||$
/*updating the number of instances and the assembly architecture*/
nb_instances$(class) := nb\_instances(class) + 1 \ ||$
assm_components$(asm) := assm\_components(asm) \cup \{inst\} \ ||$
assm_clients$(asm) := assm\_clients(asm) \cup clients(inst)$
END;

Figure 14: The component instance deployment manipulation operation

When executed, the operation adds the instance in the assembly, updates the count of instances of the component class and updates the set of client interfaces. The set of server interfaces will be updated later, as client interfaces are automatically connected by the evolution manager to maintain the consistency of the assembly (*see* § 3.2.3).

Manipulation operations constitute the dynamic aspect of the architectural formal models. They enable to change the state of a model which must therefore be validated thanks to consistency and coherence properties exposed in the following sections.

3.2.3. Consistency properties

Consistency properties maintain the correctness of each architecture description level during the evolution process. Taylor *et al.* [21] define consistency as an internal property intended to ensure that different elements of an architecture model do not contradict one another. They point out five kinds of inconsistencies that may occur in architecture models: name, interface, behavior, interaction and refinement. Our consistency properties deal with the following inconsistencies:

- *Name consistency* ensures that each component holds a unique name to avoid conflicts when selecting components.

- *Interface consistency* ensures that all architecture connections are correct (*i.e.* a required interface is always connected to a compatible provided interface).

- *Interaction consistency* ensures that the architecture realizes its functional objectives (components are able to soundly cooperate through their connected interfaces). In our approach, this property is implemented as a verification that each required interface is connected to a compatible provided one. Moreover, in architecture specifications, all server interfaces must also be connected (no unused feature is described at this level). Besides, every architecture definition must be composed of a connected graph, so that no part of the architecture is isolated.

Behavior consistency is out of the scope of the work presented in this paper which only considers static type definitions, for now. *Refinement consistency* is handled separately by our coherence properties (*cf.* Section 3.2.4).

As an example, the formalization of our interface consistency property is presented in Figure 15.

$$
\begin{array}{l}
\forall\ (cl,\ se).(cl \in client \wedge se \in server \Rightarrow \\
\quad ((cl,\ se) \in connection \Rightarrow \\
\quad \exists\ (C_1\ ,\ C_2\ ,\ int_1\ ,\ int_2\ ).(C_1 \in component \wedge C_2 \in component \wedge C_1 \neq C_2 \wedge \\
\quad int_1 \in interface \wedge int_2 \in interface \wedge cl = (C_1\ ,\ int_1) \wedge se = (C_2\ ,\ int_2) \wedge \\
\quad (int_1\ ,\ int_2) \in int\_compatible)))
\end{array}
$$

Figure 15: Interface consistency property

This property states that a required (client) interface is properly connected to a provided (server) interface when these two interfaces belong to different components and have compatible types.

Consistency properties are based on commonly adopted syntactic typing rules that state compatibility and substitution between finer grained entities such as components and interfaces. These rules transpose the well studied typing principles used in the object-oriented paradigm to the component-oriented paradigm. As usual, the main principle is that a component that belongs to a subtype can substitute for a component that belongs to a supertype (*i.e.* be connected at the same place in the same architecture). This entails that a component subtype must define a set of interfaces that can replace all the interfaces defined in its supertype (identical interfaces

or interfaces belonging to subtypes). Moreover, a component subtype cannot define extra required interfaces, as they correspond to extra connection requirements that break the substitution guarantee with the supertype. Conversely, extra provided interfaces can be defined in a subtype as they do not imply mandatory extra connections.

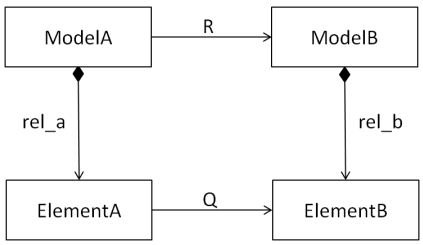
Comparing component types thus amounts to comparing interface types. Interface type hierarchies are built with respect to the same substitution principle: an interface subtype must define a set of operations that can replace those of its supertypes. Usual specialization rules are applied to provided interface types, that are comparable to object types. A provided interface subtype must define at least the same operations as its supertypes or specialized operations that can replace them. Classically, an operation specializes another one when it has the same name, a contravariant set of input parameters (at most as many parameters, with identical or more generic data types) and a covariant set of output parameters (at least as many parameters, with identical or more specific data types). With these rules, it is always possible to call a more specialized operation with the input values of a more generic one and then to use the output values of the more specialized operation in place of the output value of the more generic one.

Regarding required interfaces, opposite specialization rules are used. Indeed, a required interface corresponds to dependencies. Thus, a required interface subtype cannot define more operations than its supertypes, in order not to add extra dependencies. It cannot define less operations either, as this can impair interactions with other components. A required interface subtype must then implement the same operations as its supertypes, or more generic operations (*i.e.* operations with the same name, at least as many input parameters of identical or more specific data types and at most as many output parameters with identical or more generic data types). Requiring more generic operations than its supertypes, a more specialized required interface can replace a more generic required interface. Dedal typing rules are discussed and detailed in previous work [22].

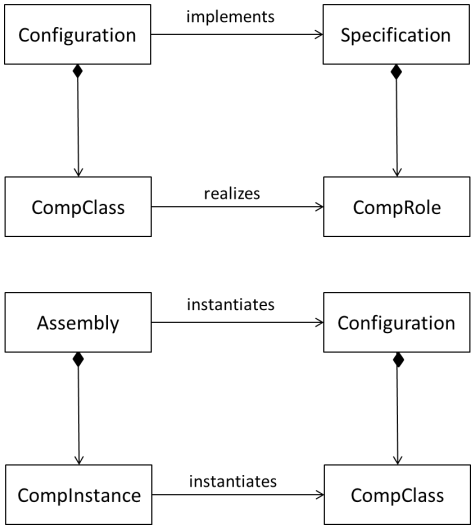
Compatibility is calculated thanks to the aforementioned typing rules. Basically, a required interface is compatible with a provided interface when they have the same type (*i.e.* are defined by the same set of operations). The required interface is also compatible with a provided interface that belongs to a subtype of its type (because of the substitution principle). Compatibility rules are also detailed in [22].

3.2.4. Coherence properties

Coherence properties prevent architecture erosion (mismatches between the different description levels) so as to maintain the global correctness of architecture definitions. Coherence properties maintain the relations that must exist between the specification, configuration and assembly defining an architecture (*cf.* Figure 16-b): its configuration must be a valid implementation of its specification; its assembly must be a valid instance of its configuration.These relations between description levels rely on typing relations between their composing elements. The component classes composing the configuration of an architecture must implement the component roles of its specification. In the same way, the component instances composing its assembly must be valid instances of the component classes of its configuration. This relates to a generic principle (*cf.* Figure 16-a) that a relation between two kinds of models implies a relation between their composing elements (and possibly reciprocally under restrictive conditions). For instance, a model can be considered as a specialization of another model only when its composing elements specialize the elements of the other model.



(a) inter-model coherence relation generic principle



(b) coherence relations between the Dedal architecture levels

Figure 16: Coherence relations between architecture levels

The generic principle can be formalized by the generic coherence rule

depicted in Figure 17.

$$
\begin{array}{l}
coherence(model_A, elem_A, model_B, elem_B, rel_a, rel_b, R, Q) == \\
\forall (M_a, M_b).(M_a \in model_A \wedge M_b \in model_B \Rightarrow ((M_a, M_b) \in R \\
\quad \Leftrightarrow \\
(\forall\, e_b.(e_b \in elem_B \wedge (M_b, e_b) \in rel_b \Rightarrow \\
\quad \exists\, e_a.(e_a \in elem_A \wedge (M_a, e_a) \in rel_a \wedge (e_a, e_b) \in Q)))))
\end{array}
$$

Figure 17: Generic coherence rule

In our work, two properties are defined in the *Evolution Management Machine* to assert the coherence of an architecture definition : coherence between configuration and specification and coherence between assembly and configuration.

Coherence between configuration and specification. A specification is a formal description of software requirements that is used to guide the search for suitable concrete component classes to implement the software. An architecture configuration is coherent with a specification when two properties hold:

- all component roles from the specification are realized by component classes in the configuration. This results in a many-to-many relation as several component roles may be realized by a single component class while, conversely, several component classes may be needed to realize a single role. Using the generic coherence rule (*cf.* Figure 17), this first property can be expressed as shown in Figure 18.

$$
\begin{array}{l}
implements \in configuration \leftrightarrow specification \wedge \\
coherence(configuration, compClass, specification, compRole, \\
\qquad config\_components, spec\_components, implements, realizes)
\end{array}
$$

Figure 18: Implementation coherence property using the generic rule

  To illustrate the instantiation of the generic coherence rule, we give the expansion of the implementation coherence property in Figure 19. In the remainder (Figure 20 and Figure 21), only the generic coherence rule is used.

- each connected provided (*server*) interface in the configuration is defined in the specification. This prevents having a configuration that implements extra functions not specified at the higher level which leads to architectural drift or erosion (*cf.* Figure 20).

$implements \in configuration \leftrightarrow specification \wedge$
$\forall\ (Conf,\ Spec).(Conf \in configuration \wedge Spec \in specification \Rightarrow$
$(Conf,\ Spec) \in implements$
$\Leftrightarrow$
$\forall\ CR.(CR \in compRole \wedge CR \in spec\_components(Spec) \Rightarrow$
$\exists\ CL.(CL \in compClass \wedge CL \in config\_components(Conf) \wedge$
$(CL, CR) \in realizes)))$

Figure 19: Implementation coherence property (expanded)

$conform \in specification \leftrightarrow configuration \wedge$
$coherence(configuration, server, specification, server,$
$\quad config\_servers, spec\_servers, conform, int\_substituion')$
where :
$(s, s') \in int\_substitution' \Leftrightarrow (serverInterfaceElem(s), serverInterfaceElem(s')) \in int\_substitution$

Figure 20: Provided interface connection coherence property

Coherence between assembly and configuration. As the definition of an assembly is not obtained from a configuration by an instantiation process (assemblies are defined at design-time), coherence between assembly and configuration descriptions must be checked *a posteriori* explicitly. An assembly is coherent with a configuration when every class of the configuration is instantiated at least once in the assembly and, conversely, every component instance in the assembly is a valid instance of a component class of the configuration (*cf.* Figure 21).

$instantiates \in assembly \rightarrow configuration \wedge$
$coherence(assembly, compInstance, configuration, compClass,$
$\quad assm\_components, config\_components, instantiates, comp\_instantiates)$
$\wedge$
$coherence(configuration, compClass, assembly, compInstance,$
$\quad config\_components, assm\_components, instantiates^{-1}, comp\_instantiates^{-1})$
where:
$instantiates^{-1}$ and $comp\_instantiates^{-1}$ are the respective reverse relations of $instantiates$ and $comp\_instantiates$

Figure 21: Configuration instantiation coherence property

3.3. Evolution goal

The evolution goal (GOAL) consists in a predicate definition that the solver will attempt to satisfy by searching for a valid sequence of evolution

rules (evolution plan) to execute on the architecture. The evolution goal consists of a static and a variable part. The static part contains all the consistency (*global_consistency*) and coherence (*global_coherence*) properties: the calculated evolution plan must maintain the validity of the architecture. The variable part contains the arguments of the initiated change: the evolution plan must achieve the intended change. For example, if the initiated change consists in the addition of a component role *cr* in a specification *spec*, the evolution goal would be the following:

$$GOAL == global\_consistency \wedge global\_coherence \wedge cr \in spec\_components(spec)$$

3.4. Evolution plan generation

Our evolution process distinguishes two kinds of change: initiated change and triggered change. *Initiated changes* have an external source: they originate from a user action or from the execution environment. *Triggered changes* are induced by the evolution manager to restore architecture consistency at each level (they are called *local changes*) and / or global architecture coherence (they are called *propagated changes*), after they have been impacted by an initiated change.

Evolution is handled as a three step process (*cf.* Figure 22).

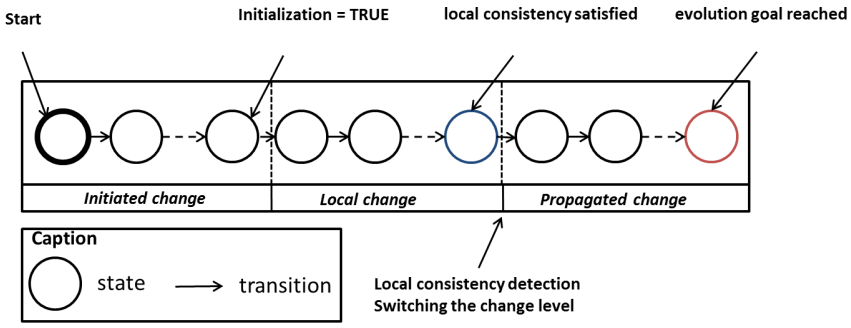


Figure 22: Evolution plan generation process

First, the initiated changes that compose a change request are all processed. These changes all affect a given level of architecture description (called the changed architecture level). In a second step, the impact of these initiated changes are calculated at the changed architecture definition level, thanks to the consistency properties. Maintaining consistency may imply additional (triggered) changes. Finally, the impact of these changes on the

other architecture definition levels are calculated thanks to the coherence properties. Maintaining coherence may also imply additional (propagated) changes on the other architecture definition levels.

4. Implementation and experimentation

To support our approach, we have implemented DedalStudio, a CASE tool which provides a Dedal modeler, a Formal Dedal generator and an evolution manager based on a solver. Three experiments are then presented in this section to assert the feasibility of our formal evolution approach. Each evolution scenario illustrates a change propagation issue that starts at a different abstraction level, in order to cover the three kinds of multi-level evolution: top-down, bottom-up and mixed. Finally, we evaluate the performance of our solver on the basis of the three experiments.

4.1. DedalStudio

4.1.1. Architecture of the tool suite

To validate our approach, we have implemented DedalStudio, an Eclipse-based modeling and evolution management environment for Dedal. The tool architecture is shown in Figure 23.

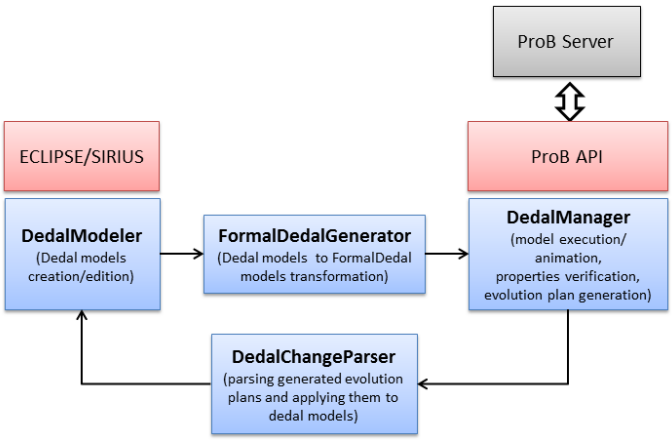


Figure 23: Architecture of the Dedal modeling and evolution management environment

DedalStudio enables the creation of architecture definitions, using a graphical concrete syntax designed for the Dedal meta-model, composed of Specification Diagrams (SD), Configuration Diagrams (CD) and Assembly Diagrams (AD). The diagram editor (*DedalModeler*), shown in Figure 24 is

based on SIRIUS [1], a generic platform that enables the creation of graphical modeling tools on top of EMF (Eclipse Modeling Framework) [2]. The *FormalDedalGenerator* creates Formal Dedal models corresponding to Dedal diagrams. The *DedalManager* handles the evolution process and the generation of evolution plans. It implements a customized solver built upon the ProB API [3] that enables the animation and model-checking of B models. Finally, the *DedalChangeParser* parses the generated evolution plans and apply the manipulation operations on the Dedal models. All theses tools, except for *DedalModeler* which is targeted to the architect, are fully automatic.

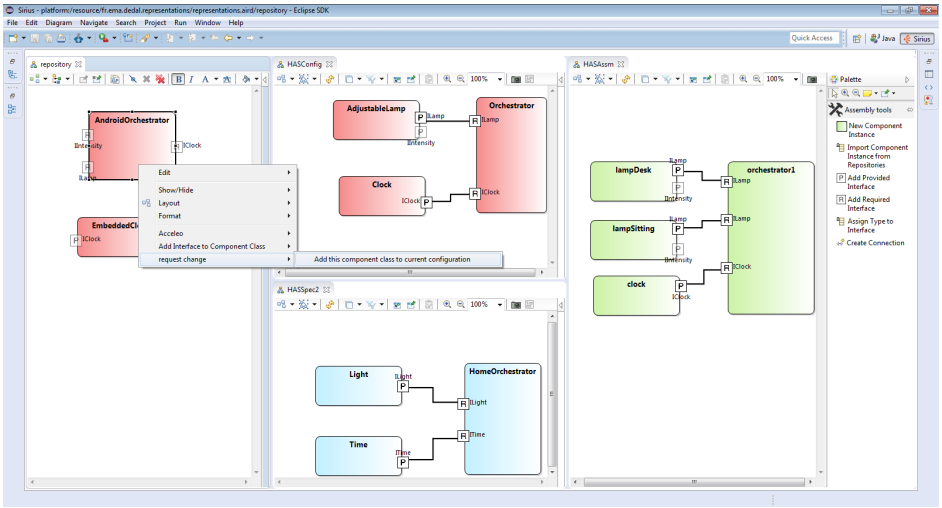


Figure 24: The DedalModeler tool

4.1.2. The DedalManager solver

Evolution management starts when a change to the architecture model is requested (for instance, a component class addition is requested in the configuration). The *DedalManager* receives the request, identifies the change level and deduces the evolution goal. It then invokes its solver, that conforms to the design principles presented in Section 3. The resolution algorithm implemented in the solver explores the search space to find a sequence of evolution

[1] https://eclipse.org/sirius/
[2] https://eclipse.org/modeling/emf/
[3] http://stups.hhu.de/ProB/w/ProB_Java_API

rules leading to the chosen goal. If a solution is found, the *DedalManager* generates an evolution plan that can then be committed by user. Otherwise (*i.e.* in case of failure), the *DedalManager* rejects the change request.

In previous work [20], we have made an evaluation of the performances of the ProB solver to generate evolution plans by state space exploration. The tested strategies were Depth-First (DF), Breadth-First (BF) and mixed (DF/BF) [23]. In most cases DF performed best, better than DF/BF and BF. The ProB solver, however, is general-purpose and increasing resolution time (over 3 minutes) is necessary when models become complex. To try and overcome this problem, this paper proposes an alternative: the implementation of a customized solver, using the API provided with ProB. It also consists in a depth-first search algorithm but enhanced with two specific heuristics: the artifact-oriented heuristic and the operation-oriented heuristic.

The artifact-oriented heuristic. The idea of artifact-oriented heuristic is to prioritize the operations manipulating the artifacts that are more likely to satisfy the evolution goal (thereafter called the main artifacts). For instance, adding a new component usually entails several connection operations on that component to restore architecture consistency. Main artifacts are determined at each iteration of the search process by the output of last executed evolution rule.

The operation-oriented heuristic. The operation-oriented heuristic adopts an opposite point of view. It delays the use of operations that engender unsatisfied dependencies between the components of the architecture and hence more evolution operations to be found in order to reestablish architecture consistency. Addition operations are the most concerned ones. They are therefore ordered as the least priority operations while performing the search process.

The search algorithm. Listing 1 describes the search algorithm of our customized solver. Lines 1–14 define and initialize the main variables of the algorithm. `Transitions` refers to the set of all the evolution rules instances in the current state of the architecture model. The set of already explored transitions is stored in `visited`, in order to avoid cycles in the search process. The current sequence of executed transitions is stored in `pl`, to collect the candidate evolution plan. The traversal of the search graph is handled by `stack`. At each step of the search process, the set of all the enabled transitions (*i.e.* the evolution rule instances whose preconditions are verified) is

pushed on the stack in order to explore them in the next steps. Transitions are pushed on the stack along with the current state of the architecture model and the current evolution plan. This enables to backtrack to previous nodes in the search graph and explore other paths when dead ends are reached. The main artifact `a` is used in the evaluation of the artifact-oriented heuristics. The `initialMainArtifact` references the artifact modified by the initiated change. It is calculated from the post-conditions of the corresponding operations.

```
1  // initialisation step
2  s = initialState;
3  a = initialMainArtifact;
4  pl = null;
5  stack = null;
6  visited = ∅;
7  enabledTransitions = {e_i ∈ Transitions where pre(e_i) == true};
8  priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
9  lowpriorTransitions = ∅;
10 enabledTransitions = enabledTransitions - priorTransitions;
11
12 // organizing stack
13 stack.push(s, pl, enabledTransitions);
14 stack.push(s, pl, priorTransitions);
15
16 // starting forward, DF search
17 while (stack ≠ ∅)
18  {
19  (s, pl, e_i) = stack.pop();
20  if ((s, e_i) ∉ visited)
21   {
22   visited = visited ∪ {(s, e_i)};
23   s = execute(e_i);
24   pl = pl+e_i;
25   if (goal == true) return pl;
26   a = output(e_i);
27   enabledTransitions = {e_i ∈ Transitions where pre(e_i) == true};
28   priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
29   lowpriorTransitions = {e_i ∈ enabledTransitions where h2(e_i) == true};
30   enabledTransitions = enabledTransitions - (priorTransitions ∪
         lowpriorTransitions);
31   stack.push(s, pl, lowpriorTransitions);
32   stack.push(s, pl, enabledTransitions);
33   stack.push(s, pl, priorTransitions);
34   }
35  }
36 return null; // no solution for this change request
```

Listing 1: Search algorithm of our specific solver

At each iteration of the search process (lines 17–33), the top of the `stack` is popped (line 19), setting a context consisting of an architecture model state (`s`), an evolution plan (`pl`) and an enabled transition (e_i). If the transition

has already been visited from this state (line 20), another context is popped from the `stack` (this happens when a state can be reached by several paths of the search tree). If the transition has not been explored, it is listed as `visited` (line 22) and executed (line 23), updating the state of the architecture model. The last executed transition is appended to the evolution plan (line 24). If the `goal` is satisfied, an evolution plan has been found and it is returned (line 25). Otherwise, the set of the enabled transitions in the current state is calculated (line 27) as is the set of higher priority enabled transitions (line 28) based on the artifact-oriented heuristic (`h1`). This uses the main artifact defined as the output of the last executed transition (line 26). The set of lower priority enabled transitions is also calculated (line 29), based on the operation-oriented heuristic (`h2`). This enables to push on the `stack` the enabled transitions to be explored depending on the priority determined by our heuristics (lines 31–33). The use of a `stack` enables a DF traversal of the graph: the next iteration of the search process will pop one of the currently enabled transitions, from the current architecture state, trying to extend the search path down to the `goal`. When a dead end is reached (no transitions are enabled in the current state), the search process implicitly backtracks to a previous graph node by popping from the top of the `stack` a previously pushed context. This enables the complete traversal of the search graph (breadth search). The search process is iterated until the `goal` is reached or there is no more transition to explore (line 17). In this latter case, the requested change is rejected (line 36).

Three examples of evolution plans calculated by our solver are presented in the next sections.

4.2. First experiment: requirement change

The first scenario addresses a requirement change. The initial HAS architecture enables to switch on / off the lights at specific hours (*cf.* Figure 25). However, it does not enable any control on light intensity. To add this new functionality, an architect should modify the HAS specification. This corresponds to a top-down evolution since the change starts at the highest abstraction level. A solution is to replace the *Light* component role by a new one (*Luminosity*) that enables intensity control. Figure 26 presents the initial architecture specification and the evolved one.

An extract of the instantiation of the *Arch_specification* machine corresponding to the HAS is presented in Figure 27.

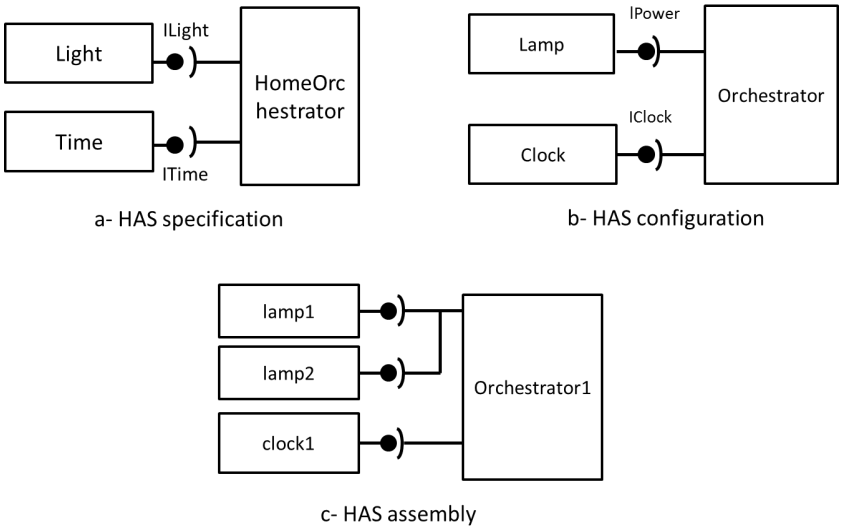


Figure 25: Architecture definitions of the HAS

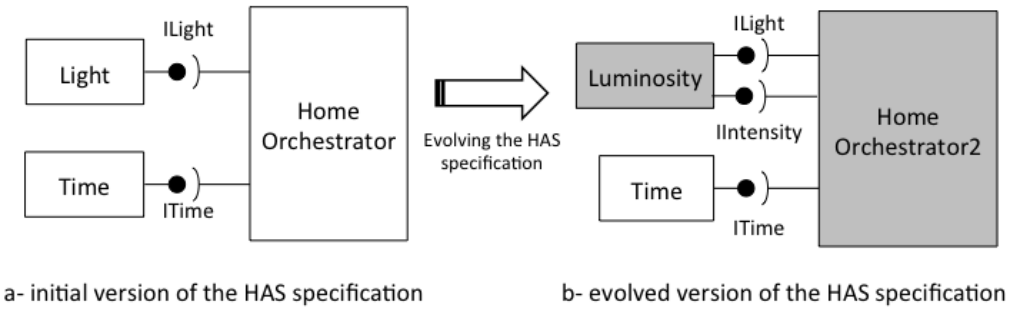


Figure 26: Evolving the HAS specification by role replacement

4.2.1. Evolution goal and initiated change

The initiated change consists in replacing the *Light* component role (*cr1*) by the *Luminosity* component role (*cr1a*). This corresponds to the execution of the role substitution operation on the HAS specification:

| **spec_replaceRole**$(HAS\_spec, cr1, cr1a)$

The following goal is thus given to the solver, based on the post-conditions of the substitution operation, defining the change that must be achieved by the evolution process:

$GOAL == global\_consistency \wedge global\_coherence \wedge cr1a \in spec\_components(HAS\_spec) \wedge$
$cr1 \notin spec\_components(HAS\_spec)$

The solver then calculates an evolution plan that can restore the consistency and coherence of the architecture that may have been altered by the initial change.

compRole $:= \{cr1, cr1a, cr2, cr3, cr3a\}||$
comp_name $:= \{cr1 \mapsto Light, cr1a \mapsto Luminosity, cr2 \mapsto Time,$
$cr3 \mapsto HomeOrchestrator,$
$cr3a \mapsto HomeOrchestrator2\}||$
arch_spec $:= \{HAS\_spec\}||$
spec_components $:= \{HAS\_spec \mapsto \{cr1, cr2, cr3\}\}||$
spec_connections $:= \{HAS\_spec \mapsto \{$
$((cr3, rintILight) \mapsto (cr1, pintILight)),$
$((cr3, rintITime) \mapsto (cr2, pintITime)), \}\}||$
spec_clients $:= \{(HAS\_spec \mapsto \{(cr3, rintILight), (cr3, rintITime)\}\}||$
spec_servers $:= \{(HAS\_spec \mapsto \{(cr1, pintILight), (cr2, pintITime)\}$

Figure 27: Instantiation of the *Arch_specification* machine for the HAS

4.2.2. Triggered change

The intended role substitution entails the addition of a new server interface (the *IIntensity* provided interface) which must be connected to restore the consistency of the HAS specification (all interfaces must be connected at specification level). The solver generates the plan presented in Figure 28 to restore the consistency of the HAS specification.

$spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1a, pintILight))$
$spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime))$
$spec\_deleteRole(HAS\_spec, cr3)$
$spec\_addRole(HAS\_spec, cr3a)$
$spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1a, pintILight2))$
$spec\_connect(HAS\_spec, (cr3a, rintITime2), (cr2, pintITime))$
$spec\_connect(HAS\_spec, (cr3a, rintIIntensity), (cr1a, pintIIntensity))$

Figure 28: HAS specification consistency restoration plan

Change entails the disconnection of all the required interfaces, the deletion of the initial orchestrator (*cr3*), the addition of a new orchestrator (*cr3a*) and finally the connection of all the required interfaces (this is enough to get all the interfaces connected and satisfy the interaction consistency property at specification level).

After consistency is verified for specification, change is propagated to the configuration in order to restore the coherence of the architecture definition.

4.2.3. Change propagation to the configuration

Coherence is altered due to the new requirement defined by the specification. Indeed, the initial HAS configuration (*cf.* Figure 25) does not correctly implement all the roles of the evolved HAS specification. Figure 29 details the instantiation of the *Arch_configuration* machine corresponding to the initial HAS configuration.

$$
\begin{array}{l}
\textbf{compClass} := \{cl1, cl1a, cl2, cl3, cl3a, cl2a\}|| \\
\textbf{comp\_name} := \{cl1 \mapsto Lamp, cl1a \mapsto AdjustableLamp, cl2 \mapsto Clock, \\
\quad cl3 \mapsto Orchestrator, cl3a \mapsto AndroidOrchestrator, \\
\quad cl2a \mapsto AndroidClock\}|| \\
\textbf{configuration} := \{HAS\_config\}|| \\
\textbf{config\_components} := \{HAS\_config \mapsto \{cl1, cl2, cl3\}\} \\
\textbf{config\_connections} := \{HAS\_config \mapsto \{ \\
\quad ((cl3, rintIPower) \mapsto (cl1, pintIPower)), \\
\quad ((cl3, rintIClock) \mapsto (cl2, pintIClock))\}
\end{array}
$$

Figure 29: Initial HAS configuration in Formal Dedal

Change propagation is therefore needed to restore coherence. The restoration plan found by the solver is presented in Figure 30.

$$
\begin{array}{l}
config\_replaceClass(HAS\_config, cl1, cl1a) \\
config\_disconnect(HAS\_config, (cl3, rintILamp), (cl1, pintILamp)) \\
config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock)) \\
config\_deleteClass(HAS\_config, cl3) \\
config\_addClass(HAS\_config, cl3a) \\
config\_connect(HAS\_config, (cl3a, rintILamp2), (cl1a, pintILamp2)) \\
config\_connect(HAS\_config, (cl3a, rintIClock2), (cl2, pintIClock)) \\
config\_connect(HAS\_config, (cl3a, rintIIntensity), (cl1a, pintIIntensity))
\end{array}
$$

Figure 30: Coherence restoration plan for the HAS configuration

It first consists in replacing the *Lamp* component class by the *AdjustableLamp* component class. This operation does not require any modification of the connections, as it is based on the substitution principle between the two component classes (the *AdjustableLamp* class is a specialization of the *Lamp* class). The situation is different regarding the *Orchestrator* component class. It cannot be simply replaced by the existing *AndroidOrchestrator* component class, which is a valid implementation of the *HomeOrchestrator2* role. Indeed, as it holds an extra required interface, the *AndroidOrchestrator* component class is not a specialization of the *Orchestrator* component class. Nonetheless, the solver is able to find a suitable plan to restore consistency

in this more difficult situation. The *Orchestrator* component class is disconnected and removed. The *AndroidOrchestrator* component class is then added and connected. This way, the configuration is consistent (all required interfaces are connected and the configuration is composed of a unique connected graph of components) and coherent with the specification (every role is implemented in the configuration).

4.2.4. Change propagation to the assembly

After coherence is reached in the configuration, change is propagated to the architecture assembly. Here again, coherence is altered because the current HAS assembly is not a valid instantiation of the evolved HAS configuration. Figure 31 details the initial state of the corresponding *Arch_assembly* machine.

$$
\begin{array}{l}
\textbf{compInstance} := \{ci11, ci12, ci1a1, ci1a2, ci2, ci2a, ci3, ci3a\}|| \\
comp\_instantiates := \{ci11 \mapsto cl1, ci12 \mapsto cl1, ci1a1 \mapsto cl1a \\
\quad ci1a2 \mapsto cl1a, ci2 \mapsto cl2, ci2a \mapsto cl2 \\
\quad ci3 \mapsto cl3, ci3a \mapsto cl3\}|| \\
\textbf{compInstance\_name} := \{ci11 \mapsto lamp1, ci12 \mapsto lamp2, ci1a1 \mapsto adjustableLamp1, \\
\quad ci1a2 \mapsto adjustableLamp2, ci2 \mapsto clock1 \\
\quad ci3 \mapsto orchestrator1, ci3a \mapsto androidOrchestrator1, \\
\quad ci2a \mapsto androidClock1\}|| \\
\textbf{assembly} := \{HAS\_assembly\}|| \\
\textbf{assm\_components} := \{HAS\_assembly \mapsto \{ci11, ci12, ci2, ci3\}\} \\
\textbf{assm\_connections} := \{HAS\_assembly \mapsto \{ \\
\quad ((ci3, rintIPowerInst) \mapsto (ci11, pintIPowerInst)), \\
\quad ((ci3, rintIClock) \mapsto (ci2, pintIClockInst)), \ldots\}
\end{array}
$$

Figure 31: Initial HAS architecture assembly

The coherence restoration plan presented in Figure 32 is generated by the solver to propagate changes. First, the client interfaces of the *Orchestrator* component instance are disconnected. Then, the two *Light* component instances are replaced by *AdjustableLight* component instances (as allowed by the substitution principle). The *Orchestrator* component instance is removed and an *AndroidOrchestrator* component instance is added. As explained for the configuration coherence restoration, substitution is not possible because of the extra required interfaces of the *AndroidOrchestrator* component. Fortunately, an evolution plan can still be found so that every component class in the configuration is instantiated at least once in the assembly. Finally, all the required interfaces are connected to compatible provided interfaces, maintaining a consistent assembly.

$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))$
$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci2, pintILampInst2))$
$assm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci12, pintIClockInst))$
$assm\_replaceInstance(HAS\_assembly, ci1, ci1a1)$
$assm\_replaceInstance(HAS\_assembly, ci12, ci1a2)$
$assm\_removeInstance(HAS\_assembly, ci3)$
$assm\_deployInstance(HAS\_assembly, ci3a)$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a1, pintILampInst1))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a1, pintIIntensityInst1))$
$assm\_bind(HAS\_assembly, (ci3a, rintIClockInst), (ci2, pintIClockInst))$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a2, pintILampInst2))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a2, pintIItensityInst2))$

Figure 32: Coherence restoration plan for the HAS architecture assembly

4.3. Second experiment: implementation change

The second scenario addresses an implementation change. The objective is to enable the control of the building through a mobile device (running Android OS for example). To adapt the current implementation to Android, the *Orchestrator* component class (*cl3*) should be removed and replaced with an Android compatible one (*cl3a*). Change is initiated at the configuration level, which entails a mixed evolution: bottom-up because the change has to be propagated to the higher level specification and top-down because it has to be propagated also to the lower assembly level. Figure 33-a shows the initial implementation of the HAS while Figure 33-b shows the evolved one.

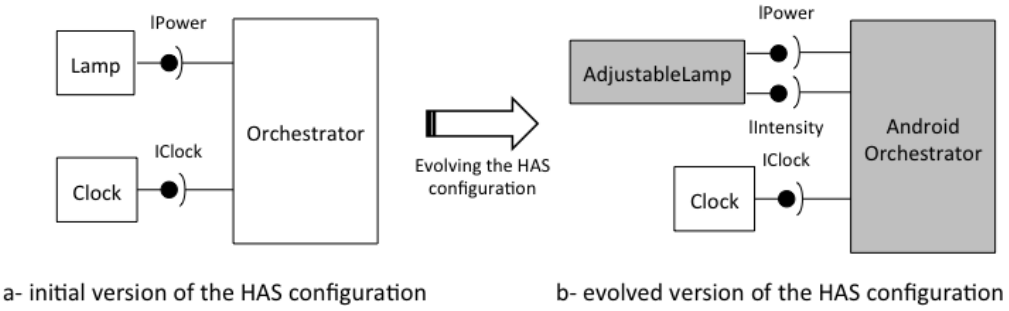


Figure 33: Evolving the HAS configuration by component substitution

4.3.1. Initiated change

Change is initiated by deleting the initial orchestrator (*cl3*) and adding the Android compatible one (*cl3a*). This is processed by the following sequence of operations:

$config\_disconnect(HAS\_config, (cl3, rintILamp), (cl1, pintILamp))$
$config\_disconnect(HAS\_config, (cl3, rintIClock), (cl2, pintIClock))$
$config\_deleteClass(HAS\_config, cl3)$
$config\_addClass(HAS\_config, cl3a)$

To start the evolution process, the following goal is given to the solver:

$GOAL == global\_consistency \wedge global\_coherence \wedge cl3a \in config\_components(HAS\_config) \wedge$
$cl3 \notin config\_components(HAS\_config)$

4.3.2. Triggered change

The generated triggered change is listed in Figure 34. To restore consistency, all component classes must be correctly connected. The *AndroidOrchestrator* component class requires an additional server interface to control the intensity of light. The *Lamp* component class (*cl1*) is suitably replaced with *AdjustableLamp* (*cl1a*) that provides the *IIntensity* server interface. This is another illustration of the solving capabilities of our approach.

$config\_connect(HAS\_config, (cl3a, rintIClock2), (cl2, pintIClock))$
$config\_replaceClass(HAS\_config, cl1, cl1a)$
$config\_connect(HAS\_config, (cl3a, rintIPower2), (cl1a, pintIPower2))$
$config\_connect(HAS\_config, (cl3a, rintIIntensity), (cl1a, pintIIntensity))$

Figure 34: HAS configuration consistency restoration plan

After configuration consistency is verified, change is propagated to the architecture specification.

4.3.3. Change propagation to the specification

The current HAS specification is not any more a good design model of the new version of the HAS configuration. This corresponds to an erosion problem as light intensity control is not included in the current specification. Hence, a new specification version is required to keep architecture descriptions coherent.

Change is propagated to the HAS specification (*cf.* Figure 35) by replacing the *HomeOrchestrator* role (*cr3*) with the *HomeOrchestrator2* (*cr3a*). To do so, the *HomeOrchestrator* role is disconnected and deleted. Then the *HomeOrchestrator2* role is added. On the other way, the *Luminosity* role (*cr1a*) can be directly substituted for the *Light* role (*cr1*). This enforces coherence between the specification and the configuration. Finally, the connection of all client interfaces is sufficient to restore the consistency of the specification (no pending interfaces; a unique connected component graph).

$spec\_disconnect(HAS\_spec, (cr3, rintILight), (cr1, pintILight))$
$spec\_disconnect(HAS\_spec, (cr3, rintITime), (cr2, pintITime))$
$spec\_deleteRole(HAS\_spec, cr3)$
$spec\_addRole(HAS\_spec, cr3a)$
$spec\_replaceRole(HAS\_spec, cr1, cr1a)$
$spec\_connect(HAS\_spec, (cr3a, rintILight2), (cr1a, pintILight2))$
$spec\_connect(HAS\_spec, (cr3a, rintIIntensity), (cr1a, pintIIntensity))$
$spec\_connect(HAS\_spec, (cr3a, rintITime2), (cr2, pintITime))$

Figure 35: HAS specification coherence restoration plan

4.3.4. Change propagation to the assembly

The current version of the HAS assembly is no more a valid instantiation of the evolved HAS configuration. Change has to be propagated at assembly level to restore coherence (*cf.* Figure 36).

$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))$
$assm\_unbind(HAS\_assembly, (ci3, rintILampInst), (ci12, pintILampInst2))$
$assm\_unbind(HAS\_assembly, (ci3, rintIClockInst), (ci2, pintIClockInst))$
$assm\_removeInstance(HAS\_assembly, ci3)$
$assm\_deployInstance(HAS\_assembly, ci3a, cl3a)$
$assm\_replaceInstance(HAS\_assembly, ci11, ci1a1)$
$assm\_replaceInstance(HAS\_assembly, ci12, ci1a2)$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a1, pintILampInst1a))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a1, pintIItensityInst1))$
$assm\_bind(HAS\_assembly, (ci3a, rintIClockInst), (ci2, pintIClockInst))$
$assm\_bind(HAS\_assembly, (ci3a, rintILamp2Inst), (ci1a2, pintILampInst2a))$
$assm\_bind(HAS\_assembly, (ci3a, rintIIntensity2Inst), (ci1a2, pintIItensityInst2))$

Figure 36: HAS assembly coherence restoration plan

In a similar way to specification coherence restoration, the *Orchestrator* component instance (*ci3*) is disconnected and deleted. An *AndroidOrchestrator* component instance (*ci3a*) is added to the assembly. Two *AdjustableLight* component instances (*ci1a1*) and (*ci1a2*) are substituted for the existing *Light* component instances (*ci11*) and (*ci12*). This restores the coherence of the assembly with the configuration. The server interfaces of the components are then bound to compatible provided interfaces, so that the assembly remains consistent (no pending server interfaces; a unique connected component graph).

4.4. Third experiment: runtime change

The third scenario addresses a runtime change. It corresponds to a bottom-up evolution since the change is initiated at the lowest abstraction

level. Because of a dry battery, the clock device in the building is out of service. This environmental change induces the dysfunction of the *clock1* driver (*ci2*). The objective is to find a solution to dynamically repair the architecture in order to maintain the functionalities of the system.

Figure 37 shows the initial and evolved version of the HAS assembly.

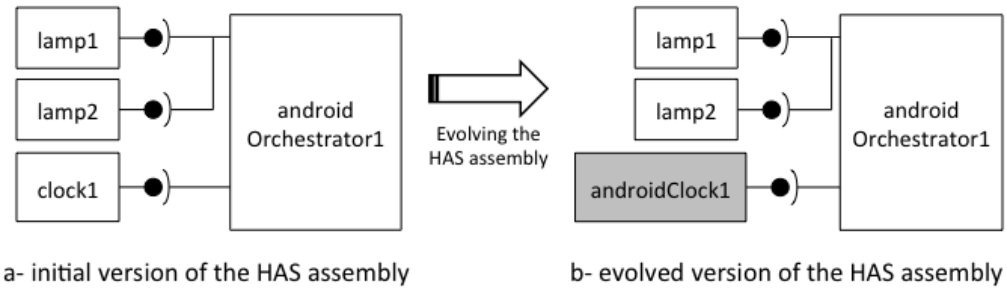


Figure 37: Evolving the HAS assembly by component instance substitution

4.4.1. Initiated change

clock1 (*ci2*) must be replaced by another component instance that provides the same services. An instance of the *AndroidClock* component class, *androidClock1* (*ci2a*), is thus chosen to replace *clock1*. The initiated change is handled by the following operations:

$$| \; replaceInstance(HAS\_assembly, ci2, ci2a)$$

The solver then searches an evolution plan that reaches the following goal:

$$| \; GOAL == global\_consistency \wedge global\_coherence \wedge ci2a \in assm\_components(HAS\_assembly) \wedge ci2 \notin assm\_components(HAS\_assm)$$

4.4.2. Triggered change

The component instance replacement does not alter the consistency of the assembly architecture. However, coherence with the configuration architecture has to be reestablished. Indeed, the evolved assembly architecture is not a valid instantiation of the current configuration architecture since the *ci2a* component instance does not instantiate the *cl2* component class.

4.4.3. Change propagation to the configuration

Change propagation induces the substitution of the *AndroidClock* component class (*cl2a*) for the *Clock* component class (*cl2*), which amounts to the following evolution plan:

$| \ replaceClass(HAS\_config, cl2, cl2a)$

As connections are preserved by the substitution operation, the consistency of the configuration is also preserved. The evolution plan thus includes no other operation.

4.4.4. Change propagation to the specification

The component class substitution preserves the coherence between the specification and the configuration. Indeed, when a component class implements a given role, any component subclass, as a substitute, also implements the role. As a consequence, no change needs to be propagated to the specification.

4.5. Performance evaluation

The performance of the solver has been measured during the three experiments, in order to evaluate the influence of our proposed heuristics. Tests were run on a standard PC (2.5 GHZ Intel Core i5, 8 GB SDRAM) under Windows 7. Test of the three evolution scenarios are then performed first using *DF* and then using *DF* enhanced with heuristics (*H-DF*) to compare the results. Table 1 shows the average time in milliseconds of 5 runs for each evolution scenario, using depth-first search without heuristics (*DF*) and with heuristics (*H-DF*).

	Change level	**DF (ms)**	**H-DF (ms)**
Exp 1	specLevel (initial)	3260	2100
	configLevel	3254	1393
	asmLevel	26738	1926
Exp 2	configLevel (initial)	4712	2537
	specLevel	8733	1896
	asmLevel	TIME-OUT	1927
Exp 3	asmLevel (initial)	4747	1184
	configLevel	TIME-OUT	2351
	specLevel (not affected)	–	–

Table 1: Performance evaluation

Timeout is set to 3 minutes. Results doubtlessly show the benefits of a custom solver that integrates specific heuristics. The order and number of

evolution rules may differ from a generated evolution plan to another (our algorithms are not deterministic as they make random choices when sets of equivalent elements are considered, such as a set of candidate main artifacts) but all generated plans are valid and lead to the same goal state.

A more precise performance evaluation, based on a larger set of experiments and a theoretical study of the combinatorial complexity of the search space is needed. Performance is indeed an inherent limitation for search-based software engineering, as the resolution time of solvers generally grows exponentially depending on the size of the problems. Designing and integrating new heuristics to cut down resolution time is promising (we can for instance preferentially choose transitions that generate no or little incoherence in the architecture model).

5. Related work

This section presents three areas of related work. The first area is that of software architecture evolution which is the main theme of this work. It presents a survey of the main state-of-the-art evolution approaches our work can be compared to. The second area is that of formal modeling languages. It presents a brief comparison of seven formal modeling languages including B. The third area describes other approaches based on model transformation and integration of semi-formal and formal methods. These approaches do not necessarily focus on architecture evolution but they present interesting alternatives from the technical point of view.

5.1. Software architecture evolution

Most of the approaches dealing with architecture evolution adopt an ADL to model architectures and propose a mapping between the ADL and a runtime framework in order to implement the change and enable dynamic evolution. C2-SADEL [24], Darwin [25], ArchWare [26] and Plastik [27] fall into this category. C2-SADEL models architectures in the C2 style [28] and provides multiple component subtyping mechanisms to favor reuse and enable architecture evolution. Its tool support is Dradel, an environment that enables the mapping between architectural description and the implementation by translating them into Java code. The tool supports static evolution by applying changes on architectural descriptions first and then implementing them. The architecture analysis however is limited since no powerful analysis techniques were integrated. Darwin and ArchWare (which provides π-

ADL [29] as an ADL) focus on modeling dynamic structures. They both rely on π-calculus to define the semantics of architecture constructs and guarantee a reliable interaction between components and compile architecture descriptions into code. ArchWare also proposes π-ARL [30] an architecture refinement language to evolve architecture descriptions by stepwise refinement. Plastik was also proposed to deal with dynamic reconfigurations. It relies on Armani, an extension of the ACME [31] ADL to enable invariants expression and reconfigurations properties. Compared to the previous approaches, Plastik has the advantage to map its ADL to OpenCOM [32], a runtime component model dedicated to component-based programming and proposing built-in reconfiguration operations. The main shortcoming of these approaches is that they don't consider changes as first-class elements and focus more on how to implement architecture evolution rather than specify, analyze and propagate it. Moreover, adopted ADLs hardly cover the entire CBSD process. The specification level (necessary to guide reuse) and assembly level (that describes the software at runtime) are often missing. Finally, the coherence between architectural descriptions and implementation is not guaranteed since evolution is processed top-down only.

Recent work by Sanchez *et al.* [33] proposes an architecture-based re-engineering approach to evolve and maintain legacy software. The principle is to produce a high level architecture description of the legacy system so that it becomes easy to reason about change and then reversely use the produced knowledge to modify source code. The approach is guided through a bidirectional transformation and relies on Archery [34], an ADL for modeling architecture patterns corresponding to translated code parts. Targetted at legacy system re-engineering, this work is different from our proposal on the evolution of component-based software systems developed by a reuse-based process.

Other recent approaches show a particular interest to specifying architecture evolution as first-class entities. A first example is the work of Tamazalit, Le Goaer *et al.* [35, 36]. The authors introduce the notion of *evolution styles*, first-class entities that can be specified and classified for reuse to evolve a particular family of systems. Evolution styles include evolution operations that can be specialized, composed and instantiated to deal with change. Barnes *et al.* [37] adopt a wider definition of evolution styles and introduce the concept of evolution paths as a way to plan the evolution of domain-specific software systems. A path is an evolution trace leading from an initial architecture to a desired target architecture. An evolution style refers to a family

of evolution paths sharing common properties. It includes operations, constraints and functions to evaluate paths according to quality metrics. Path constraints can be formally specified using the *path constraint language*, a specific extension of LTL (Linear Temporal Logic). While the computability of the language was proved, as far as we know, there is no existing model checker to support the automated analysis of path constraints. The authors also propose a solution [38] to automate evolution planning using PDDL [39] (the Planning Domain Definition Language). However, this approach still lacks automation since no translation from any ADL to PDDL specification was proposed. Moreover, the evolution is specified and planned beforehand. In our approach, changes are not necessarily expected and the architect intervenes only to validate the work of the evolution manager.

Another closely related work is the one of Hansen, Ingstrup and others [40, 41]. The authors propose an approach to model and analyze runtime architectural change. They opt for a runtime architecture model that closely maps to the OSGi [4] platform to facilitate implementation and for Alloy [42] as a relational first-order logic modeling language to formalize the static and dynamic (operations) concepts of the architecture model. The choice of Alloy is motivated by its support for object-oriented modeling and its accompanying analyzer that enables automated verification. The objective is to apply architectural changes without violating some predefined properties. For this purpose, the authors model the reconfiguration planning as a predicate satisfaction problem with pre- and post-conditions. Then, they run the Alloy SAT solver to find sequences of the model instances satisfying the problem where the first instance satisfies the pre-conditions and the last instance satisfies the post-conditions. This work is similar to ours in the sense that both aim to provide a reliable and automated way to handle architectural changes. It proposes an interesting alternative for resolving evolution using the constraint-solving technique. However, this work focuses only on one level of change which is runtime. Moreover, the formalized architecture model is dependent on OSGi. Finally, the work lacks automation, since no automatic translation from ADL models to Alloy models was proposed.

[4] http://www.osgi.org/Main/HomePage

5.2. Comparison of formal modeling languages

Formal modeling brings abstraction, precision and rigor to software systems. It intervenes at the very early stages of software development to give a formal specification of system requirements. Resulting models constitute unambiguous descriptions that enable software analysis, verification and validation. Several languages and methods were proposed to aid formal modeling. Formal languages provide abstractions to represent concepts, properties over them and possibly behavior. However, they differ in expressiveness, underlying semantics and purpose. Some languages focus more on descriptions and how to make formal modeling more accessible whereas others focus more on automated analysis neglecting expressiveness. A good formal language must be a compromise between both aspects. In the following, we compare seven formal modeling languages. These languages are B [9], Z [43], OCL [44], Alloy [42], VDM [45], Coq [46] and Agda [47].

B, Z and VDM are quite similar in term of expressiveness since they were basically designed for theorem-proving. All of them enable to express properties practically in the same way and support almost the same types (In addition, VDM supports real numbers). However there are some subtle differences between them. Z is more abstract while VDM and B are more low level and intended to be refined into code. Both VDM and B adopt a similar structure that realizes abstract state machines. They explicitly separate the declarative (structure) from the dynamic (operations) part and, unlike Z, they separate pre-conditions from post-conditions. B has the particularity to modify variables by assignments like in programming languages while in VDM and Z, pre and post states must be explicit.

Coq and Agda are proof assistants designed for the verification of functional programs. Unlike the previously mentioned formal modeling languages, Coq and Agda are implementations of type theories rather than set theory. They support higher order logic, polymorphism, dependent types, as well as inductive types. Set theoretic operators (*e.g.* $\cup$, $\cap$), for instance, are not directly predefined in such systems. Unlike B and VDM, these languages do not implement state machines. Therefore, there is no built-in structure that explicitly defines variables, invariants and operations.

OCL and Alloy are different and were designed for different purposes. OCL was basically developed to express constraints that can not be expressed using graphical notations on UML diagrams. It has an object-oriented notation and heavily relies on navigation. Hence predicate expressions are sometimes

verbose comparing to the mathematical notation adopted by the other languages. Alloy is a structural modeling language inspired by Z. It was designed for supporting fully automated analysis. Being strictly first-order, Alloy is less expressive than the other languages [48]. For instance, set of sets and predicates over relations are not directly expressible with Alloy.

Regarding analysis support, all these languages are typed and hence support type-checking. Theorem-proving is supported by Coq, Agda, Z, B and VDM which were basically designed for software correctness. Model-checking and constraint solving is only supported by B, with the ProB tool, and Alloy, with the Alloy analyzer. To some extent, Jaza [49], an animator for Z, enables constraint-solving on small domains. However, Z is limited in terms of model-checking capabilities. This is due to the high abstract nature of the Z language making its handling challenging [50]. Nevertheless, continuous attempts to build a model checker for Z are undertaken [51].

B seems to be the best compromise between expressiveness and analysis support. Alloy could also be a good alternative in our case. However, regardless its expressiveness, it presents another shortcoming. As witnessed in Torlak *et al.* [52], Alloy lacks support of partial instances. Partial instances are explicit representations of instances included in the specification of the model. This is central in our approach since instances are generated automatically from graphical models and injected in B specifications (so-called deep embedding technique [53]). Montaghami *et al.* [54] argued that this feature enables a number of capabilities such as test-driven development, regression testing, modeling by example, and combined modeling and meta-modeling. The authors also proposed a syntax extension of Alloy to support partial instance definition but, as far as we know, this feature is not yet integrated in the last version of Alloy [55].

5.3. Alternative formal approaches

Integration between semi-formal and formal methods is gaining more and more interest in software engineering. On the one hand, semi-formal languages, such as UML [56], offer graphical notations that significantly ease modeling. On the other hand, formal modeling languages provide a strong support for automated software analysis. Several works benefit from combining both kinds of notation to validate their approaches.

Ledru *et al.* [57] propose an approach based on the transformation of UML into B to validate security policies for information systems. They use their

B4MSecure [5] tool to generate B specifications corresponding to a security model. Conjointly, they use ProB to validate security policy scenarios.

Keznikl *et al.* propose the ARCAS method [58], an automated approach to generate connections solutions for middleware architectures. Given a connector specification, the approach translates it into a corresponding Alloy model and performs constraint-solving to find connector instances that realize the specification.

Macedo *et al.* propose Echo [59], an Eclipse-based tool for model repair and transformation using model finding. Given a set of meta-models with internal constraints (specified using OCL) and a set of inter-model consistency rules (specified using QVT-R [60] transformations), Echo can detect inconsistencies on derived models and keep them consistent with their corresponding meta-models and between them as well. The detection and repair mechanism is based on translating MDE [61] artifacts (meta-models and their annotations with OCL and QVT-R rules) to Alloy. The output is then analyzed using a procedure built on top of Alloy solver that generates consistent models as close as possible to the original ones.

6. Conclusions and future work

Managing software architecture evolution throughout the whole software lifecycle is a significant issue. This paper proposes an approach to manage the evolution of component-based software architectures. Thanks to the three-level Dedal architecture model, our approach handles change at three abstraction levels of software architectures: specification, implementation and deployment. The evolution process is driven by an evolution management model that captures changes initiated at any abstraction level, controls their impact to preserve / restore consistency and propagates them to other levels to maintain global coherence.

The proposed evolution management model is based on the B formal language. Using our solver built on top of the ProB tool, it enables the generation of reliable evolution plans as sequences of change operations. The feasibility of our approach is demonstrated by experimenting on three evolution scenarios that each addresses change in a different abstraction level.

The limitation of this work is its scalability. This limitation is classical in comparable works as architecture descriptions can be considered as graphs

[5] http://b4msecure.forge.imag.fr/

(of connected software components) the size of which can theoretically be arbitrarily big. Establishing evolution plans therefore amounts to exploring all possible change action combinations on these graphs to restore properties that can be seen as (local or global) constraints on these graphs. Scalability issue is an inherent limitation for search-based software engineering problems. However, such limitation is mitigated by two factors. First, architecture descriptions are often limited in size as architects prefer to split them in intelligible parts of moderate size using hierarchical composition, an asset of CBSD [1]. Secondly, instead of using an off-the-shelf agnostic B solver, we proposed our own solver that integrates problem-specific heuristics that decrease the calculation time.

Threats to the validity of our approach lie in the example scenarios that we have considered for experimental validation. Although, the examples cover all kinds of scenarios, experimenting with real architecture descriptions might reveal unforeseen issues (scalability, efficiency of heuristics, *etc.*). Further experiments on real case studies is therefore necessary to fully validate our approach.

As future work, we would like to extend our definition of the consistency property in order to include behavioral consistency as described in Taylor *et al.* [21] and thus cover all their identified five kinds of consistency. This would amount in considering architectural protocols and component behavior.

Another interesting research direction would be to integrate the notion of evolution style [36] in our evolution management model. The idea is to enable the generation of multiple candidate evolution plans that can be evaluated considering non-functional properties (*e.g.* quality, cost, time) as proposed by Barnes *et al.* [37].

Regarding the technical aspect, we are investigating new heuristics to improve the performance of our solver and reduce complexity.

7. Acknowledgements

The authors warmly thank the anonymous reviewers for their in-depth reading of the paper and their helpful comments that made it possible to greatly improve its quality.

References

[1] H. V. Vliet, Software Engineering: Principles and Practice, 3rd Edition, Wiley Publishing, 2008.

[2] H. P. Breivold, I. Crnkovic, M. Larsson, A systematic review of software architecture evolution research, Information and Software Technology 54 (1) (2012) 16 – 40. doi:http://dx.doi.org/10.1016/j.infsof.2011.06.002.
URL http://www.sciencedirect.com/science/article/pii/S0950584911001376

[3] T. Mens, S. Demeyer, Software Evolution, Springer, 2008.

[4] D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.

[5] L. de Silva, D. Balasubramaniam, Controlling Software Architecture Erosion: A Survey, Journal of Systems and Software 85 (1) (2012) 132–151.

[6] H. Y. Zhang, C. Urtado, S. Vauttier, Architecture-centric component-based development needs a three-level ADL, in: Proceedings of the 4th European Conference of Software Architecture, Vol. 6285 of Lecture Notes in Computer Science, Springer, Copenhagen, Denmark, 2010, pp. 295–310.

[7] H. Y. Zhang, L. Zhang, C. Urtado, S. Vauttier, M. Huchard, A three-level component model in component-based software development, in: Proceedings of the 11th of the International Conference on Generative Programming: Concepts and Experiences, ACM, Dresden, Germany, 2012, pp. 70–79.

[8] I. Crnkovic, S. Sentilles, A. Vulgarakis, M. Chaudron, A classification framework for software component models, IEEE Transactions on Software Engineering 37 (5) (2011) 593–615.

[9] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, USA, 1996.

[10] D. Cansell, D. Méry, Foundations of the B method, Computers and Informatics 22 (2003) 1–31.

[11] P. Behm, P. Benoit, A. Faivre, J.-M. Meynadier, Meteor: A Successful Application of B in a Large Project, in: J. Wing, J. Woodcock, J. Davies

(Eds.), FM99 Formal Methods, Vol. 1708 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1999, pp. 369–387.

[12] D. Cansell, G. Gopalakrishnan, M. Jones, D. Mry, A. Weinzoepflen, Incremental proof of the producer/consumer property for the PCI protocol, in: D. Bert, J. Bowen, M. Henson, K. Robinson (Eds.), ZB 2002:Formal Specification and Development in Z and B, Vol. 2272 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2002, pp. 22–41.

[13] Atelier B, ClearSy, Aix-en-Provence (F) http://www.atelierb.eu/ accessed 09/01/2015.

[14] The B-Toolkit User's Manual, B-Core (UK) Limited.

[15] D. Delahaye, C. Dubois, C. March, D. Mentr, The BWare project: Building a Proof Platform for the Automated Verification of B Proof Obligations, in: Y. Ait Ameur, K.-D. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, Vol. 8477 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2014, pp. 290–293.

[16] M. Leuschel, M. Butler, ProB: An Automated Analysis Toolset for the B Method, International Journal on Software Tools for Technology Transfer 10 (2) (2008) 185–203.

[17] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: why reuse is so hard, IEEE Software 12 (6) (1995) 17–26. doi:10.1109/52.469757.

[18] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: Why reuse is still so hard, IEEE Software 26 (4) (2009) 66–69. doi:10.1109/MS.2009.86.

[19] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H. Y. Zhang, Formal rules for reliable component-based architecture evolution, in: Formal Aspects of Component Software - 11th International FACS Symposium revised selected papers, Bertinoro, Italy, 2014, pp. 127–142.

[20] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H. Y. Zhang, An evolution management model for multi-level component-based software architectures, in: The 27th International Conference on Software Engineering

and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015, 2015, pp. 674–679. doi:10.18293/SEKE2015-172.
URL http://dx.doi.org/10.18293/SEKE2015-172

[21] R. Taylor, N. Medvidovic, E. Dashofy, Software architecture: Foundations, Theory, and Practice, Wiley, 2009.

[22] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H. Y. Zhang, Towards automating the coherence verification of multi-level architecture descriptions, in: Proceedings of the 9th ICSEA, Nice, France, 2014, pp. 416–421.

[23] M. Leuschel, J. Bendisposto, Directed model checking for B: An evaluation and new techniques, in: J. Davies, L. Silva, A. Simao (Eds.), Formal Methods: Foundations and Applications, Vol. 6527 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 1–16. doi:10.1007/978-3-642-19829-8_1.

[24] N. Medvidovic, D. S. Rosenblum, R. N. Taylor, A Language and Environment for Architecture-based Software Development and Evolution, in: Proceedings of the 21st ICSE, 1999, pp. 44–53.

[25] J. Magee, J. Kramer, Dynamic structure in software architectures, ACM SIGSOFT Software Engineering Notes 21 (6) (1996) 3–14.

[26] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti, ArchWare: Architecting Evolvable Software, in: F. Oquendo, B. Warboys, R. Morrison (Eds.), Software Architecture, Vol. 3047 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 257–271. doi:10.1007/978-3-540-24769-2_23.
URL http://dx.doi.org/10.1007/978-3-540-24769-2_23

[27] A. Joolia, T. Batista, G. Coulson, A. T. A. Gomes, Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform, in: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE, Washington, USA, 2005, pp. 131–140.

[28] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, A Component- and Message-based Architectural Style

for GUI Software, in: Proceedings of the 17th International Conference on Software Engineering, ICSE '95, ACM, New York, USA, 1995, pp. 295–304.

[29] F. Oquendo, Pi-ADL: An Architecture Description Language Based on the Higher-order Typed Pi-calculus for Specifying Dynamic and Mobile Software Architectures, SIGSOFT Software Engineering Notes 29 (3) (2004) 1–14.

[30] F. Oquendo, Pi-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures, SIGSOFT Software Engineering Notes 29 (5) (2004) 1–20. doi:10.1145/1022494.1022517.
URL http://doi.acm.org/10.1145/1022494.1022517

[31] D. Garlan, R. Monroe, D. Wile, ACME: An Architecture Description Interchange Language, in: Proceedings of Centre for Advanced Studies Conference, IBM Press, 1997, p. 7.

[32] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, A component model for building systems software, in: Software Engineering and Applications (SEA '04), Cambrigde, 2004, pp. 684–689.

[33] A. Sanchez, N. Oliveira, L. S. Barbosa, P. Henriques, A perspective on architectural re-engineering, Science of Computer Programming 98, Part 4 (2015) 764 – 784. doi:http://dx.doi.org/10.1016/j.scico.2014.02.026.
URL http://www.sciencedirect.com/science/article/pii/S0167642314000938

[34] A. Sanchez, L. Barbosa, D. Riesco, Bigraphical Modelling of Architectural Patterns, in: F. Arbab, P. lveczky (Eds.), Formal Aspects of Component Software, Vol. 7253 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 313–330. doi:10.1007/978-3-642-35743-5_19.
URL http://dx.doi.org/10.1007/978-3-642-35743-5_19

[35] D. Tamzalit, M. Oussalah, O. L. Goaer, A. Seriai, Updating Software Architectures : A Style-Based Approach, in: Proceedings of the International Conference on Software Engineering Research and Practice &

Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1, 2006, pp. 336–342.

[36] O. L. Goaer, D. Tamzalit, M. Oussalah, A. Seriai, Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures, in: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland, 2008, pp. 311–318. doi:10.1109/COMPSAC.2008.104.
URL http://dx.doi.org/10.1109/COMPSAC.2008.104

[37] J. M. Barnes, D. Garlan, B. Schmerl, Evolution styles: foundations and models for software architecture evolution, Software and Systems Modeling 13 (2) (2014) 649–678.

[38] J. M. Barnes, A. Pandey, D. Garlan, Automated planning for software architecture evolution, in: Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pp. 213–223. doi:10.1109/ASE.2013.6693081.
URL http://dx.doi.org/10.1109/ASE.2013.6693081

[39] D. McDermott, PDDL-The Planning Domain Definition Language, Tech. rep., Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998).

[40] M. Ingstrup, K. M. Hansen, Modeling architectural change: Architectural scripting and its applications to reconfiguration, in: Joint Working IEEE/IFIP Conference on Software Architecture, WICSA/ECSA, 2009, pp. 337–340. doi:10.1109/WICSA.2009.5290670.

[41] K. M. Hansen, M. Ingstrup, Modeling and Analyzing Architectural Change with Alloy, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 2257–2264. doi:10.1145/1774088.1774560.
URL http://doi.acm.org/10.1145/1774088.1774560

[42] D. Jackson, Alloy: A Lightweight Object Modelling Notation, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 256–290.

[43] J. M. Spivey, The Z Notation: A Reference Manual, Prentice Hall International (UK) Limited, 1992.

[44] OCL, 2.3.1 specification, http://www.omg.org/spec/OCL/2.3.1/ accessed 09/01/2015.

[45] C. B. Jones, Systematic Software Development Using VDM (2nd Ed.), Prentice-Hall, 1990.

[46] Y. Bertot, P. Castran, G. Huet, C. Paulin-Mohring, Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions, Texts in theoretical computer science, Springer, Berlin, New York, 2004, donnes complmentaires http://coq.inria.fr.
URL http://opac.inria.fr/record=b1101046

[47] U. Norell, Dependently Typed Programming in Agda, in: Proceedings of the 6th International Conference on Advanced Functional Programming, AFP'08, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 230–266.
URL http://dl.acm.org/citation.cfm?id=1813347.1813352

[48] D. Jackson, Software Abstractions: Logic, Language, and Analysis, Appendix E: Alternative Approaches, The MIT Press, 2006.

[49] M. Utting, Jaza user manual and tutorial, http://www.cs.waikato.ac.nz/ marku/jaza/ accessed 03/08/2015.

[50] G. Smith, L. Wildman, Model Checking Z Specifications Using SAL, in: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB 2005: Formal Specification and Development in Z and B, Vol. 3455 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 85–103. doi:10.1007/11415787_6.
URL http://dx.doi.org/10.1007/11415787_6

[51] J. Derrick, S. North, A. Simons, Z2SAL: a translation-based model checker for Z, Formal Aspects of Computing 23 (1) (2011) 43–71. doi:10.1007/s00165-009-0126-7.
URL http://dx.doi.org/10.1007/s00165-009-0126-7

[52] E. Torlak, D. Jackson, Kodkod: A Relational Model Finder, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4424 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 632–647. doi:

10.1007/978-3-540-71209-1_49.
URL http://dx.doi.org/10.1007/978-3-540-71209-1_49

[53] J. Svenningsson, E. Axelsson, Combining Deep and Shallow Embedding for EDSL, in: H.-W. Loidl, R. Pea (Eds.), Trends in Functional Programming, Vol. 7829 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 21–36. doi:10.1007/978-3-642-40447-4_2.
URL http://dx.doi.org/10.1007/978-3-642-40447-4_2

[54] V. Montaghami, D. Rayside, Extending alloy with partial instances, in: J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, E. Riccobene (Eds.), Abstract State Machines, Alloy, B, VDM, and Z, Vol. 7316 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 122–135. doi:10.1007/978-3-642-30885-7_9.
URL http://dx.doi.org/10.1007/978-3-642-30885-7_9

[55] Alloy lanaguge reference, http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf accessed 25/07/2015.

[56] UML, 2.5 specification, http://www.omg.org/spec/UML/2.5/Beta2/ accessed 09/01/2015.

[57] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, M.-A. Labiadh, Taking into Account Functional Models in the Validation of IS Security Policies, in: C. Salinesi, O. Pastor (Eds.), Advanced Information Systems Engineering Workshops, Vol. 83 of Lecture Notes in Business Information Processing, Springer, Berlin, Heidelberg, 2011, pp. 592–606.

[58] J. Keznikl, T. Bure, F. Plil, P. Hntynka, Automated resolution of connector architectures using constraint solving (ARCAS method), Software & Systems Modeling 13 (2) (2014) 843–872. doi:10.1007/s10270-012-0274-8.
URL http://dx.doi.org/10.1007/s10270-012-0274-8

[59] N. Macedo, T. Guimaraes, A. Cunha, Model repair and transformation with Echo, in: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013, pp. 694–697. doi:10.1109/ASE.2013.6693135.

[60] OMG, MOF 2.0 Query/View/Transformation (QVT), version 1.1, http://www.omg.org/spec/QVT/1.1/ accessed 24/07/2015.

[61] D. C. Schmidt, Guest Editor's Introduction: Model-Driven Engineering, Computer 39 (2) (2006) 25–31.

Annexe C

Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach

International Journal of Software Engineering and Knowledge Engineering

Automatic documentation of [mined] feature implementations from source code elements and use case diagrams with the REVPLINE approach

R. AL-msie'deen, M. Huchard and A.-D. Seriai
LIRMM / CNRS and Montpellier University
Montpellier - France
{*al-msiedee, huchard, seriai*}*@lirmm.fr*

C. Urtado and S. Vauttier
LGI2P / Ecole des Mines d'Alès
Nîmes, France
{*Christelle.Urtado, Sylvain.Vauttier*}*@mines-ales.fr*

Received (31 August 2014)
Accepted (16 October 2014)

Companies often develop a set of software variants that share some features and differ in other ones to meet specific requirements. To exploit the existing software variants as a Software Product Line (SPL), a Feature Model of this SPL must be built as a first step. To do so, it is necessary to define and document the optional and mandatory features that compose the variants. In our previous work, we mined a set of feature implementations as identified sets of source code elements. In this paper, we propose a complementary approach, which aims to document the mined feature implementations by giving them names and descriptions, based on the source code elements that form feature implementations and the use-case diagrams that specify software variants. The novelty of our approach is its use of commonality and variability across software variants, at feature implementation and use-case levels, to run Information Retrieval methods in an efficient way. Experiments on several real case studies (Mobile media and ArgoUML-SPL) validate our approach and show promising results.

Keywords: Software variants; Software Product Line; Feature documentation; Code comprehension; Formal Concept Analysis; Relational Concept Analysis; Use-case diagram; Latent Semantic Indexing; Feature Models.

1. Introduction

Similarly to car developers who propose a full range of cars with common characteristics and numerous variants, software developers may cater to various needs and propose as a result a software family instead of a single product. Such a software family is called a Software Product Line (SPL) [1].

Software variants often evolve from an initial product, developed for and successfully used by the first customer. These product variants usually share some common features and differ regarding others. As the number of features and the

number of software variants grow, it is worth re-engineering them into a SPL for systematic reuse [2].

The first step towards re-engineering software variants into SPL is to mine a Feature Model (FM). To obtain such a FM, the common and optional features that compose these variants have to be identified and documented. This consists in (step 1) identifying, among source code elements, groups that implement candidate features and (step 2) associating them with their documentation (*i.e.*, a feature name and description). In our previous work [3], we proposed an approach for step 1 which consists in mining features from the object-oriented source code of software variants (the REVPLINE approach[a]). REVPLINE mines functional features as sets of Source Code Elements (SCEs) (*e.g.*, packages, classes, attributes, methods or method bodies).

In this article, we address step 2. To assist a human expert in documenting the mined feature implementations, we propose an automatic approach which associates names and descriptions using the source code elements of feature implementations and the use-case diagrams of software variants. Compared with existing work that documents source code (*cf.* Section 2), the novelty of our approach is that we exploit commonality and variability across software variants at feature implementation and use-case levels in order to apply Information Retrieval (IR) methods in an efficient way.

Considering commonality and variability across software variants enables to group use-cases and feature implementations into *disjoint* and *minimal* clusters based on Relational Concept Analysis (RCA). Each cluster consists of a subset of feature implementations and their corresponding use-cases. Then, we use Latent Semantic Indexing (LSI) to measure similarities and identify which use-cases best characterize the name and description of each feature implementation by using Formal Concept Analysis (FCA). In the cases where use case diagrams or documentation are not available, we propose an alternative approach, based on the names of the source code elements that implement features.

The remainder of this paper is structured as follows: Section 2 presents the state of the art and motivates our work. Section 3 briefly describes the technical background which is used in our work. Section 4 outlines the feature documentation process. Section 5 details the feature documentation process based on the use case diagrams of variants step by step. Section 6 presents the feature documentation process based on SCE names. Section 7 reports the experimentation and discusses threats to the validity of our approach. Finally, Section 8 concludes and provides perspectives for this work.

[a]REVPLINE stands for RE-engineering software Variants into a software Product LINE.

2. State of the art

In our approach, we aim at documenting groups of source code elements that are the result of a mining process. These groups of source code elements correspond to feature implementations. The comprehension or documentation of feature implementation is a complex problem-solving task [4]. We consider here that documentating a feature is the process of analyzing the implementation of a feature to provide it with either a name or a more detailed description based on software artifacts such as use-case diagrams or identifier names. Related approaches are documentating the source code of a single software, finding traceability links between source code and documentation, and documentating mined features in software product lines.

This section presents research papers in these three fields. We then conclude this state of the art by a synthesis which introduces the main objectives of our approach.

2.1. *Source code documentation in a single software*

Kebir *et al.* [5] propose to identify components from the object-oriented source code of a single software. Their approach assigns names to the components in three steps: extracting and tokenizing class names from the identified implementation of a component, weighting words and building the component name by using the strongest weighted tokens.

Kuhn [6] presents a lexical approach that uses the log-likelihood ratio of word frequencies to automatically retrieve labels from source code. This approach can be applied to compare several components (*i.e.*, describing their differences as well as their commonalities), a component against a normative corpus (*i.e.*, providing labels for components) and different versions of the same component (*i.e.*, documenting the history of a component). In Kuhn *et al.* [7], information retrieval techniques are used to exploit linguistic information found in source code, such as identifier names and comments, in order to enrich software analysis with the developers' knowledge that is hidden in the code. They introduce semantic clustering, a technique based on LSI and clustering to group source artifacts (*i.e.*, classes) that use similar vocabulary. They call these groups semantic clusters and they interpret them as linguistic topics that reveal the intention of the code. They compare the topics, identify links between them, provide automatically retrieved labels (using LSI again to automatically label the clusters with their most relevant terms). They finally use distribution maps to illustrate how the semantic clusters are distributed over the system. Their work is language independent as it works at the level of identifier names.

De Lucia *et al.* [8] propose an approach for source code labeling, based on IR techniques, that identifies relevant words in the source code of a single software. They apply various IR methods (such as VSM, LSI and Latent Dirichlet Allocation (LDA)) to extract terms from class names by means of some representative words, with the aim of facilitating code comprehension or improving visualization. This work investigates to what extent IR-based source code labeling would identify relevant words in the source code, compared to the words a human would manually

select during a program comprehension task.

A technique for automatically summarizing source code by leveraging the lexical and structural information in the code is proposed in Haiduc *et al.* [9]. Summaries are obtained from the content of a document by selecting the most important information in that document. The goal of this approach is the automatic generation of summaries for source code entities.

Falleri *et al.* [10] propose a wordNet-like approach to extract the structure of a single software using relationships among identifier names. The approach considers natural language processing techniques which consist of a tokenization process (straightforward decomposition technique by word markers, *e.g.*, case changes, underscore, *etc.*), part of speech tagging and rearranging order of terms by term dominance order rules based on part of speech information.

Sridhara *et al.* [11] present a novel technique to automatically generate comments for Java methods. They use the signature and body of a method (*i.e.*, method calls) to generate a descriptive natural language summary of the method. The developer remains responsible for verifying the accuracy of generated summaries. The objective of this approach is to ease program comprehension. Authors use natural language processing techniques to automatically generate leading method comments. Studies have shown that good comments help programmers understand quickly what a method does, thus assisting program comprehension and software maintenance.

2.2. *Source code-to-documentation traceability links*

Grechanik *et al.* [12] propose a novel approach for partially automating the process of recovering traceability links (TLs) between types and variables in Java programs and elements of use-case diagrams (UCDs). Authors evaluate their prototype implementation on open-source and commercial software, and their results suggest that their approach can recover many traceability links with a high automation degree and precision. As UCDs are widely used to describe the functional requirements of software products, these traces help programmers understand the code that they maintain and modify.

Marcus *et al.* [13] use LSI to recover traceability links between source code and documentation. The documentation consists of requirement documents which describe elements of the problem domain such as manuals, design models or test suites. This documentation is supposed to have been written before implementation and does not include any parts of the source code.

Diaz *et al.* [14] capture relationships between source code artifacts to improve the recovery of traceability links between documentation and source code. They extract the author of each source code component and, for each author, identify the "context" she/he worked on. Thus, to link documentation and source code artifacts (*i.e.*, use cases and classes), they compute the similarity between these use cases and the authors' contexts. When retrieving related classes using a standard IR-based

approach (*e.g.*, LSI or VSM) they reward all the classes developed by authors whose contexts are most similar to use cases.

Xue *et al.* [2] automatically identify traceability links between a given collection of features and a given collection of source code variants. They consider feature descriptions as an input.

2.3. *Documentation of mined features in SPL*

Braganca and Machado [15] describe an approach for automating the transformation of UML use-cases into FMs. In their work, each use-case is mapped to a feature. Their approach explores the *include* and *extend* relationships between use-cases to discover relationships between features. Their work assumes that the feature name is given by that of the use-case.

Yang *et al.* [16] analyze open source applications for multiple domains with similar functionalities. They propose an approach to recover domain feature models using data access semantics, FCA, concept pruning/merging, structure reconstruction and variability analysis. After concept pruning/merging, analysts examine each of the generated candidate feature (*i.e.*, concept cluster) to evaluate its relevance. Meaningless candidate features are removed, whilst meaningful candidate features are chosen as domain features. Then analysts name each domain feature with the help of the corresponding concept intent and extent. After these manual examination and naming, all domain features bear significant names denoting their business functions.

Paškevičius *et al.* [17] present a framework for an automated derivation of FMs from existing software artifacts (*e.g.*, classes, components, libraries, *etc.*), which includes a formal description of FMs, a program-feature relation meta-model and a method for FM generation based on feature dependency extraction and clustering. FMs are generated as Feature Description Language (FDL) descriptors and as Prolog rules. They focus on reverse engineering of source code to FMs and assume that feature names are provided by that of the class or component.

Ziadi *et al.* [18] propose a semi-automatic approach to identify features from object-oriented source code. Their approach takes the source code of a set of product variants as its input. They *manually* assign names to the identified feature implementations by relying on the feature names that are used in the original FM.

In our previous work [19, 3, 20], we manually propose feature names for the mined feature implementations, based on the code elements of each feature implementation.

Davril *et al.* [21] build FMs from product descriptions. Extracting FMs from these informal data sources includes mining feature descriptions from sets of product descriptions, naming the features in a way that is understandable to human users and then discovering relations between features in order to organize them hierarchically into a comprehensive model. The identified features correspond to clusters of descriptions. Authors propose a method to name a cluster using the

most frequently and less verbose occurring phrase in the descriptors of this cluster.

2.4. *Synthesis*

Most existing approaches are designed to extract labels, names, topics or to identify traceability links between code and documentation artifacts in a single software system. To document mined features, most existing approaches manually assign feature names to feature implementations (without any further description) and they often rely on atomic source code element names (*e.g.*, class or component names). The most advanced approach for automatic feature description extraction is that of [21] which works on informal product descriptions.

Our work addresses the problem of documenting features mined from several variants of a software system. Our mined feature implementations are sets of source code elements that are more formal artifacts than the product descriptions used in [21] and give complementary information as compared to use case diagrams [15]. Our approach relies on commonality and variability across the variants to apply information retrieval methods more efficiently than in a single software system. Our input data are the source code and use case diagrams of the variants. We do not consider any prior knowledge about features contrarily to [2]. We aim at automatically assigning a name and a description to each mined feature implementation using several techniques (Formal Concept Analysis, Relational Concept Analysis and Latent Semantic Indexing), whereas several approaches manually assign names [16, 18, 3]. Feature documentation consists in use case names, tokens from the source code elements and use case descriptions.

3. Technical background

This section provides a glimpse on FCA, RCA and LSI. It also shortly describes the example that illustrates the remaining sections of the paper.

3.1. *Formal and Relational Concept Analysis*

Formal Concept Analysis (FCA) is a classification technique that takes as an input data sets describing objects and their attributes and extracts concepts that are maximal groups of objects (concept extents) sharing maximal groups of attributes (concept intents) [22]. It has many applications in software engineering [23, 24, 25]. The extracted concepts are linked by a partial order relation, which represents concept specialization, as an order which has a lattice structure (called the concept lattice). In the concept lattice, each attribute (*resp.* each object) is introduced by a unique concept and inherited by its sub-concepts (*resp.* by its super-concepts). In figures, objects and attributes are often represented only in their introducing concept for the sake of simplicity. Rather than using the whole concept lattice, which is often a large and complex structure (exponential number of concepts, considering objects and attributes, in the worst case), we use a sub-order called the

AOC-poset, which is restricted to the concepts that introduce at least one object or one attribute. The interested reader can find more information about our use of FCA in [3].

Relational Concept Analysis (RCA) [26] is an iterative version of FCA in which objects are classified not only according to attributes they share, but also according to relations between them (*cf.* Section 5.1). Other close approaches are [27, 28, 29].

In the RCA framework, data are encoded into a *Relational Context Family* (RCF), which is a pair (K, R), where K is a set of formal (object-attribute) contexts $K_i = (O_i, A_i, I_i)$ and R is a set of relational (object-object) contexts $r_{ij} \subseteq O_i \times O_j$, where O_i (domain of r_{ij}) and O_j (range of r_{ij}) are the object sets of the contexts K_i and K_j, respectively (*cf.* Table 3). A RCF is iteratively processed to generate, at each step, a set of concept lattices. As a first step, concept lattices are built using the formal contexts only. Then, in the following steps, a scaling mechanism translates the links between objects into conventional FCA attributes and derives a collection of lattices whose concepts are linked by relations (*cf.* Figure 5).

To apply FCA and RCA, we use the Eclipse eRCA platform[b].

3.2. *Latent Semantic Indexing*

Information Retrieval (IR) refers to techniques that compute textual similarity between documents. Textual similarity is computed based on the occurrences of terms in documents [30]. When two documents share a large number of terms, those documents are considered to be similar. Different IR techniques have been proposed, such as Latent Semantic Indexing (LSI) and Vector Space Model (VSM), to compute textual similarity.

As proposed by [2], we use LSI to group together software artifacts that pertain to the implementation or the documentation of a similar, thus considered common, concept[c].

To do so, software artifacts are regarded as textual documents. Occurrences of terms are extracted from the documents in order to calculate similarities between them and then classify together similar documents (*cf.* Section 5.2).

The heart of LSI is the singular value decomposition technique. This technique is used to mitigate noise introduced by stop words (like "the", "an", "above") and overcome two issues of natural language processing: *synonymy* and *polysemy*.

The effectiveness of IR methods is usually measured by metrics including *recall*, *precision* and *F-measure* (*cf.* Equations 1, 2 and 3). In this work, for a given use-case (query), recall is the percentage of correctly retrieved feature implementations (documents) to the total number of relevant feature implementations, while precision is the percentage of correctly retrieved feature implementations to the total number of retrieved feature implementations. F-Measure defines a trade-off between

[b]The eRCA : `http://code.google.com/p/erca/`

[c]To set our approach up, we developed our own LSI tool, available at `https://code.google.com/p/lirmmlsi/`

precision and recall, that gives a high value only when both recall and precision are high.

$$Recall = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{relevant\ documents\}|} \quad (1)$$

$$Precision = \frac{|\{relevant\ documents\} \bigcap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \quad (2)$$

$$F - Measure = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} \quad (3)$$

All measures have values in [0%, 100%]. When recall equals 100%, all relevant feature implementations are retrieved. However, some retrieved feature implementations may not be relevant. If precision equals 100%, all retrieved feature implementations are relevant. Nevertheless, all relevant feature implementations may not be retrieved. When F-Measure equals 100%, all relevant feature implementations are retrieved.

The interested reader can find more information about our use of LSI in [3].

3.3. *The Mobile Tourist Guide Example*

In this example, we consider four software variants of the Mobile Tourist Guide (MTG) application. These variants enable users to inquire about tourist information on mobile devices. MTG_1 supports core MTG functionalities: *view map*, *place marker on a map*, *view direction*, *launch Google map* and *show street view*. MTG_2 has the core MTG functionalities and a new functionality called *download map from Google*. MTG_3 has the core MTG functionalities and a new functionality called *show satellite view*. MTG_4 supports *search for nearest attraction*, *show next attraction* and *retrieve data* functionalities, together with the core ones. Table 1 describes the sets of functionnalities (use-cases) implemented by the different MTG software variants. Figure 1 shows the corresponding use-case diagrams.

In this example, we can observe that the use-case diagrams of software variants show commonality and variability at use-case level (*i.e.*, functionalities). This prompts to extract feature documentation from the use-case diagrams of software variants. Table 2 shows the mined feature implementations from MTG software variants. In the examples, mined feature implementations are named using the same names as that of the corresponding use-case for the sake of clarity. But as mentioned before, mined feature names are not known beforehand. We only use the mined feature implementations composed of Source Code Elements (SCEs) and the use-case diagrams of software variants as inputs for the documentation process.

Table 1. The use-cases of MTG software variants.

	View map	Place marker on a map	View direction	Launch Google map	Show street view	Download map from Google	Show satellite view	Search for nearest attraction	Show next attraction	Retrieve data
Mobile Tourist Guide 1	×	×	×	×	×					
Mobile Tourist Guide 2	×	×	×	×	×	×				
Mobile Tourist Guide 3	×	×	×	×	×		×			
Mobile Tourist Guide 4	×	×	×	×	×			×	×	×

Product-by-use case matrix
(× use-case is in the product)

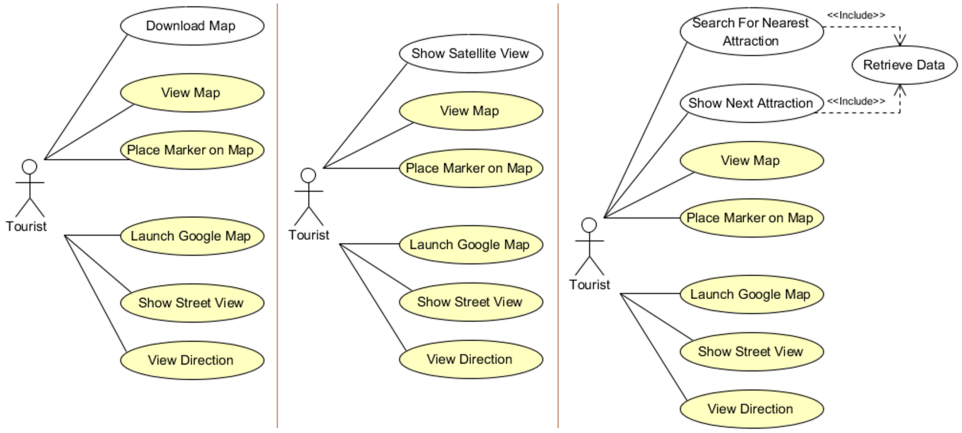


Fig. 1. The use-case diagrams of the MTG software variants.

4. The Feature Documentation Process

As previously mentioned, we aim at documenting mined feature implementations by using use-case diagrams that document a set of software variants. We rely on the same assumption as in the work of [15] stating that each use-case corresponds to a feature. The feature documentation process uses lexical similarity to identify which use-case best characterizes the name and description of each feature implementation. As performance and efficiency of the IR technique depend on the size of the search space, we take advantage of the commonality and variability between software variants to group feature implementations and the corresponding use-cases in the software family into disjoint, minimal clusters (*e.g.*, Concept_1 of Figure 5). We call each disjoint minimal cluster a *Hybrid Block* (HB). After reducing the search

Table 2. The mined feature implementations from MTG software variants.

	Feature Implementation_1: View map	Feature Implementation_2: Place marker on a map	Feature Implementation_3: View direction	Feature Implementation_4: Launch Google map	Feature Implementation_5: Show street view	Feature Implementation_6: Download map from Google	Feature Implementation_7: Show satellite view	Feature Implementation_8: Search for nearest attraction	Feature Implementation_9: Show next attraction	Feature Implementation_10: Retrieve data
Mobile Tourist Guide 1	×	×	×	×	×					
Mobile Tourist Guide 2	×	×	×	×	×	×				
Mobile Tourist Guide 3	×	×	×	×	×		×			
Mobile Tourist Guide 4	×	×	×	×	×			×	×	×

Product-by-feature implementation matrix
(× feature implementation is in the product)

space to a set of hybrid blocks, we measure textual similarity to identify, within each hybrid block, which use-case may provide a name and a description of each feature implementation.

For a product variant, our approach takes as inputs the set of use-cases that documents the variant and the set of mined feature implementations that are produced by REVPLINE. Each use-case is defined by its name and description. This information represents domain knowledge that is usually available as software documentation (*i.e.*, requirement model). In our work, the use-case description consists of a short paragraph in a natural language. For example, the *retrieve data* use-case of Figure 1 is described by the following paragraph, "*the tourist can retrieve information and a small image of the attraction using his/her mobile phone. In addition, the tourist can store the current view of the map in the mobile phone*".

Our approach provides a name and a description for each feature implementation based on a use-case name and description. Each use-case is mapped onto a functional feature thanks to our assumption. If two or more use-cases have a relation with the same feature implementation, we consider them all as the documentation for this feature implementation.

Figure 2 shows an overview of our feature documentation process. The first step of this process identifies hybrid blocks based on RCA (*cf.* Section 5.1). In the second step, LSI is applied to determine similarity between use-cases and feature implementations (*cf.* Section 5.2). FCA is then used to build use-case clusters. Each

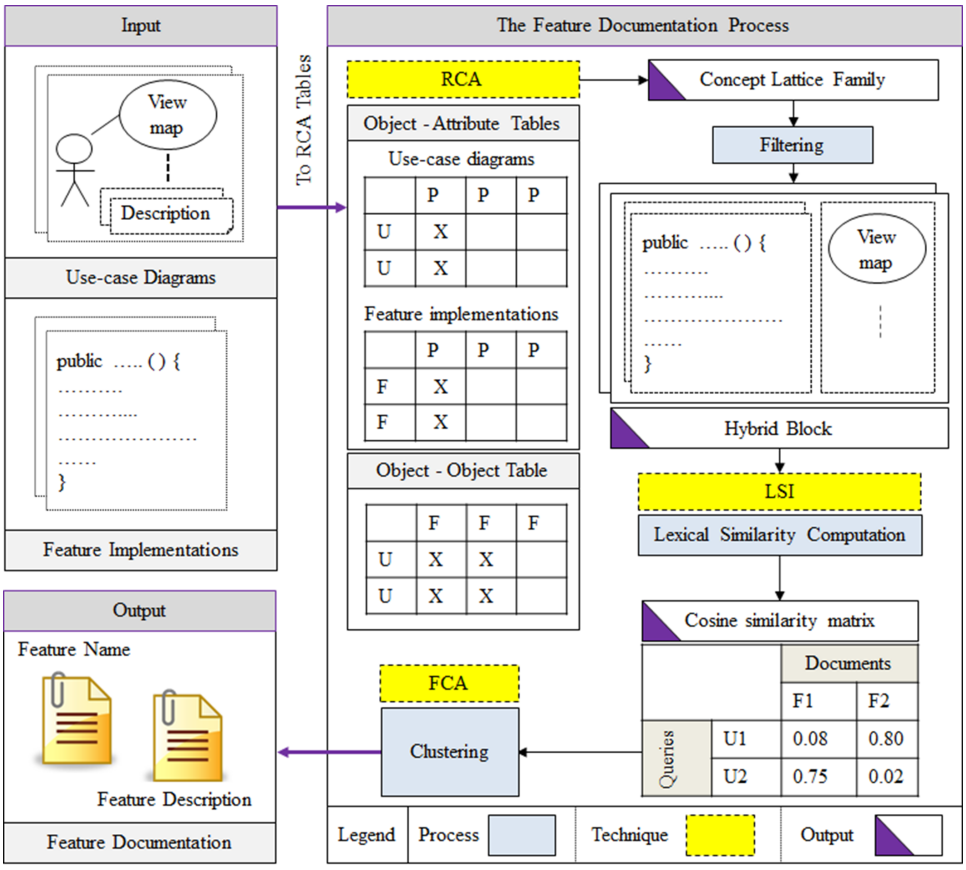


Fig. 2. The feature documentation process.

cluster identifies a name and a description for feature implementation (*cf.* Section 5.3).

5. Feature Documentation Step by Step

In this section, we describe the feature documentation process step by step. Feature name and description are identified through three steps as detailed in the following.

5.1. *Identifying Hybrid Blocks of Use-cases and Feature Implementations via RCA*

Mined feature implementations and use-cases are clustered into disjoint minimal clusters (*i.e.*, hybrid blocks) to apply LSI on reduced search spaces. RCA is used to build the clusters, based on the commonality and variability in software variants: use-cases and feature implementations that are common to all software variants;

use-cases and feature implementations that are shared by a set of software variants, but not all variants; use-cases and feature implementations that are held by a single variant (*cf.* Figure 3).

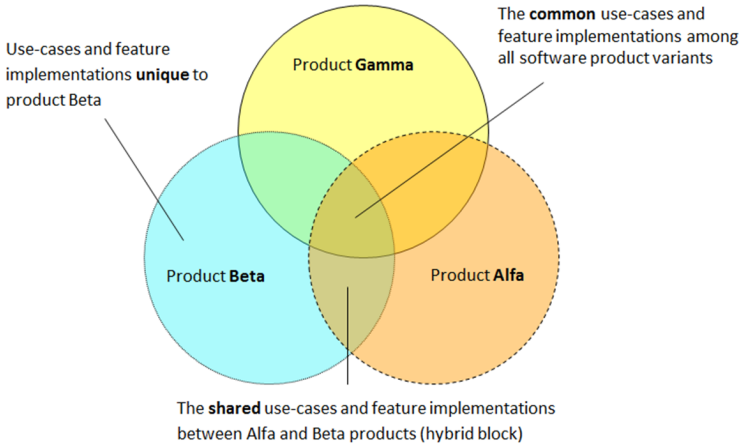


Fig. 3. The common, shared and unique use-cases (*resp.* feature implementations) across software product variants.

An RCF is automatically generated from the use-case diagrams and the mined feature implementations associated with software variants[d]. The RCF used in our approach contains two formal contexts and one relational context, as illustrated in Table 3. The first formal context represents the *use-case diagrams*: objects are use-cases and attributes are software variants. The second formal context represents *feature implementations*: objects are feature implementations and attributes are software variants. The relational context (*i.e.*, *appears-with*) indicates which use-case appears in the same software variant as a feature implementation.

For the RCF in Table 3, the two lattices of the *Concept Lattice Family* (CLF) are represented in Figure 4 and a close-up view is represented in Figure 5. An example of a hybrid block is given in Figure 5 (the left dashed block), which gathers a set of use-cases (from the extent of $Concept\_1$ in the *Use_case_Diagrams* lattice) that always appear with a set of feature implementations (from the extent of $Concept\_6$ in the *Feature_Implementations* lattice). The relation between the two concepts is represented in $Concept\_1$ via the relational attribute $appears\text{-}with{:}Concept\_6$. As shown in Figure 5, RCA enables to reduce the search space by exploiting commonality and variability across software variants. In our work, we filter the CLF from bottom to top to get a set of hybrid blocks[e].

[d]Source code : `https://code.google.com/p/rcafca/`
[e]Source code : `https://code.google.com/p/fecola/`

Table 3. The RCF for features documentation.

Use_case_Diagrams	MTG_1	MTG_2	MTG_3	MTG_4
View Map	×	×	×	×
Launch Google Map	×	×	×	×
View Direction	×	×	×	×
Show Street View	×	×	×	×
Place Marker on Map	×	×	×	×
Download Map		×		
Show Satellite View			×	
Show Next Attraction				×
Search For nearest attraction				×
Retrieve Data				×

Feature_Implementations	MTG_1	MTG_2	MTG_3	MTG_4
Feature Implementation_1	×	×	×	×
Feature Implementation_2	×	×	×	×
Feature Implementation_3	×	×	×	×
Feature Implementation_4	×	×	×	×
Feature Implementation_5	×	×	×	×
Feature Implementation_6		×		
Feature Implementation_7			×	
Feature Implementation_8				×
Feature Implementation_9				×
Feature Implementation_10				×

Relational context: appears-with	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5	Feature Implementation_6	Feature Implementation_7	Feature Implementation_8	Feature Implementation_9	Feature Implementation_10
View Map	×	×	×	×	×					
Launch Google Map	×	×	×	×	×					
View Direction	×	×	×	×	×					
Show Street View	×	×	×	×	×					
Place Marker on Map	×	×	×	×	×					
Download Map						×				
Show Satellite View							×			
Show Next Attraction								×	×	×
Search For Nearest Attraction								×	×	×
Retrieve Data								×	×	×

5.2. *Measuring the Lexical Similarity Between Use-cases and Feature Implementations via LSI*

Each hybrid block created during previous step consists of a set of use-cases and a set of feature implementations. Which use-cases characterize the name and description of each feature implementation needs to be identified. To do so, we use textual similarity between use-cases and feature implementations. This similarity measure is calculated using LSI. We consider that a use-case corresponding to a feature implementation should be lexically closer to this feature implementation than to others. Similarity between use-cases and feature implementations in the hybrid blocks is computed in three steps as detailed below.

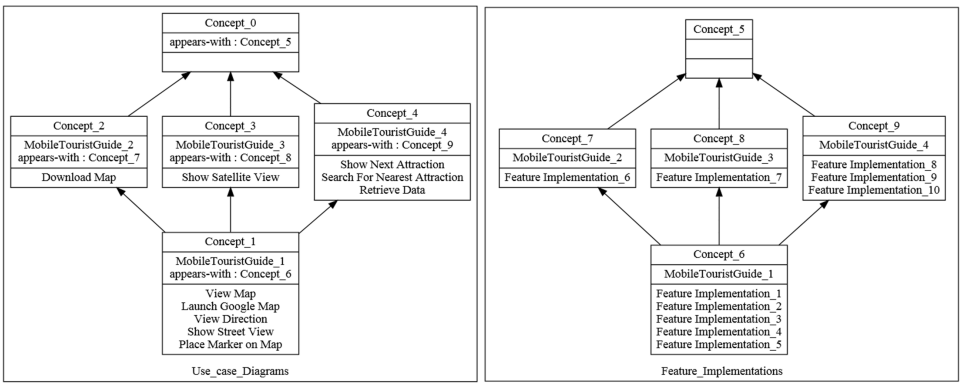


Fig. 4. The concept lattice family of relational context family in Table 3.

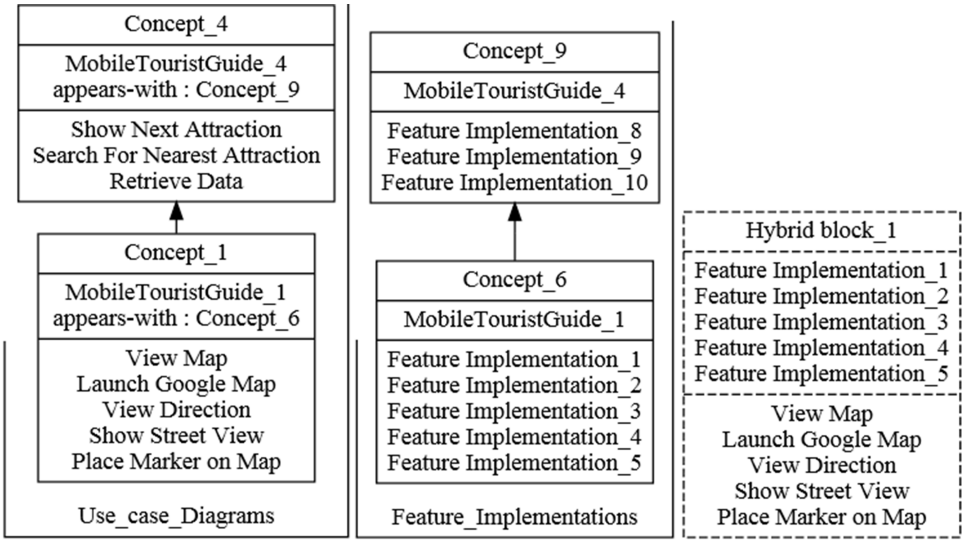


Fig. 5. Parts of the CLF deduced from Table 3.

5.2.1. *Building the LSI Corpus*

In order to apply LSI, we build a corpus that represents a collection of documents and queries (*cf.* Figure 6). In our work, each *use-case* name and description in the hybrid block represents a *query* and each *feature implementation* represents a *document*.

This document contains all the words extracted from the SCE names in the feature implementation as a result of splitting them using a tokenization scheme (Camel-case). Regardless of their location in the SCE names, we store all the words in the document. For example, in the SCE name *ManualTestWrapper* all words are

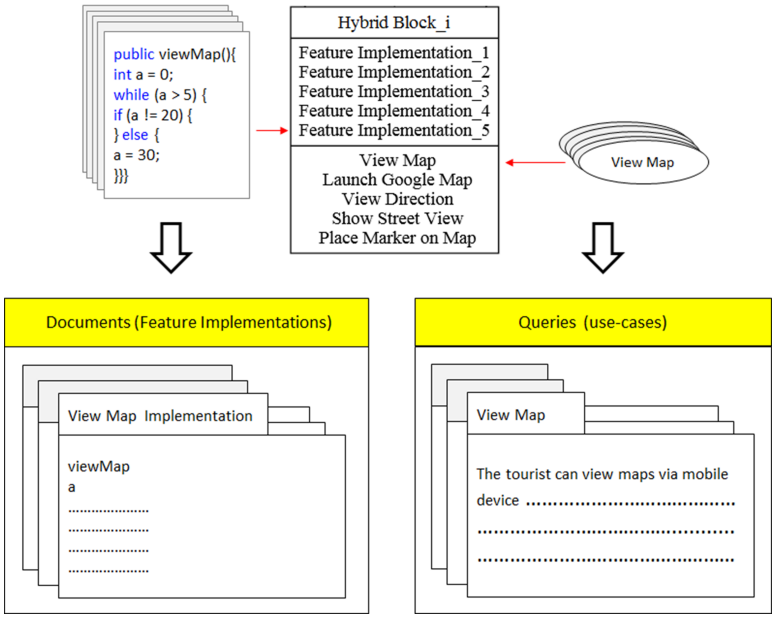


Fig. 6. Constructing a raw corpus from hybrid block.

important: *manual*, *test* and *wrapper*. This process is applied to all feature implementations. Our approach creates a query for each use-case. This query contains the use-case name and its description.

To be processed, documents and queries must be normalized as follows: stop words, articles, punctuation marks, or numbers are removed; text is tokenized and lower-cased; text is split into terms; stemming is performed (*e.g.*, removing word endings); terms are sorted alphabetically. We use WordNet[f] to do part of the preprocessing (*e.g.*, stemming and removal of stop words).

The most important parameter of LSI is the number of term-topics (*i.e.*, *k-Topics*) chosen. A term-topic is a collection of terms that co-occur often in documents of the corpus, for example {*user, account, password, authentication*}. In our work, the number of *k-Topics* is equal to the number of feature implementations for each corpus.

5.2.2. *Building the Term-document and the Term-query Matrices for each Hybrid Block*

The *term-document matrix* is of size $m \times n$, where m is the number of terms extracted from feature implementations and n is the number of feature implementations (*i.e.*, documents) in a corpus. The matrix values indicate the number of oc-

[f] `http://wordnet.princeton.edu/`

currences of a term in a document, according to a specific weighting scheme. In our work, terms are weighted using the *TF-IDF* function (the most common weighting scheme) [2]. The *term-query matrix* is of size $m \times n$, where m is the number of terms that are extracted from use-cases and n is the number of use-cases (*i.e.*, queries) in a corpus. An entry in the term-query matrix refers to the weight of the i^{th} term in the j^{th} query.

Table 4. The term-document and the term-query matrices of *Concept_1* in Figure 5.

	Feature Implement._1	Feature Implement._2	Feature Implement._3	Feature Implement._4	Feature Implement._5
device	1	0	0	0	1
direction	0	0	0	6	0
google	1	0	0	0	0
launch	4	0	0	0	0
map	1	2	0	0	4
marker	0	6	0	0	0
mobile	1	0	0	0	1
place	0	3	0	0	0
show	0	0	2	0	0
street	0	0	5	0	0
tourist	1	1	1	1	1
view	0	0	1	2	5

The term-document matrix

	Launch Google Map	Place Marker on Map	Show Street View	View Direction	View Map
device	1	0	0	0	1
direction	0	0	0	8	0
google	3	0	0	0	0
launch	3	0	0	0	0
map	2	2	1	1	5
marker	0	3	0	0	0
mobile	1	0	0	0	1
place	0	3	0	0	0
show	0	0	3	0	0
street	0	0	5	0	0
tourist	1	1	1	1	1
view	0	0	1	3	5

The term-query matrix

In the term-document matrix (in left-hand side of Table 4), the *direction* term appears 6 times in the *Feature Implementation_4* document. In the term-query matrix (in right-hand side of Table 4), the *direction* term appears 8 times in the *view direction* query.

5.2.3. *Building the Cosine Similarity Matrix*

Similarity between documents in a corpus is measured by the cosine of the angle between their corresponding term vectors [31], as given by Equation 4, where d_q is a query vector, d_j is a document vector and $W_{i,q}$ and $W_{i,j}$ go over the weights of terms in the query and document respectively.

$$cosine\ similarity\ (d_q, d_j) = \frac{\overrightarrow{d_q} \cdot \overrightarrow{d_j}}{|\overrightarrow{d_q}||\overrightarrow{d_j}|} = \frac{\sum_{i=1}^{n} W_{i,q} * W_{i,j}}{\sqrt{\sum_{i=1}^{n} W_{i,q}^2}\sqrt{\sum_{i=1}^{n} W_{i,j}^2}} \quad (4)$$

Similarity between use-cases and feature implementations in each hybrid block is thus defined by a *cosine similarity matrix* which columns (documents) represent feature implementations and rows (queries) use-cases [3].

Table 5. The cosine similarity matrix of *Concept_1* in Figure 5.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	0.861933577	0.0137010	0	0	0.152407
Place Marker on Map	0.01114798	0.9480070	0	0	0.085939
Show Street View	0.004088722	0.0051128	0.98581691	0.00571	0.070920
View Direction	0.00296571	0.0037085	0.0069484	0.999139665	0.108597
View Map	0.114676597	0.0627020	0.039159941	0.070025418	0.993111

Results are represented as a directed graph. Use-cases (*resp.* feature implementations) are represented as vertices and similarity links as edges. The degree of similarity appears as weights on the edges (*cf.* Figure 7). This graph is only used for visualization purposes.

5.3. *Identifying Feature Name via FCA*

Having the cosine similarity matrix, we use FCA to identify, in each hybrid block, which use-cases and feature implementations are related. To transform a (numerical) cosine similarity matrix into a (binary) formal context, we use a 0.70 threshold (after having tested many threshold values). This means that only pairs of use-cases and feature implementations having similarity greater than or equal to 0.70 are considered related. Table 6 shows the formal context obtained by transforming the cosine similarity matrix corresponding to the hybrid block of *Concept_1* from Figure 5.

For the *Concept_1* hybrid block of Figure 5 the number of term-topics of LSI is equal to 5. In the formal context associated with this hybrid block, the *"Launch Google Map"* use-case is linked to the *"Feature Implementation_1"* feature implementation because their similarity equals 0.86, which is greater than the threshold.

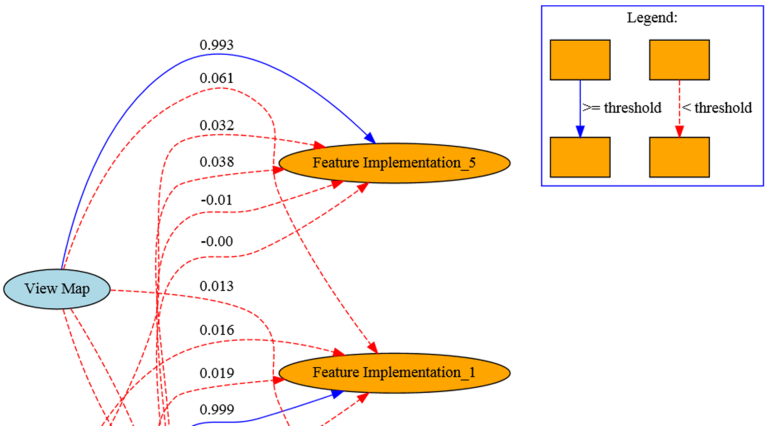


Fig. 7. The lexical similarity between use-cases and feature implementations as a directed graph.

Table 6. Formal context of *Concept_1* in Figure 5.

	Feature Implementation_1	Feature Implementation_2	Feature Implementation_3	Feature Implementation_4	Feature Implementation_5
Launch Google Map	×				
Place Marker on Map		×			
Show Street View			×		
View Direction				×	
View Map					×

On the contrary, the *"View Direction"* use-case and the *"Feature Implementation_5"* feature implementation are not linked because their similarity equals 0.10, which is less than the threshold. The resulting AOC-poset is composed of concepts whose *extent* represents the use-case name(s) and *intent* represents the feature implementation(s). In this simple example, there is a one-to-one association between use case names and feature implementations, but in the general case, we may have several use case names associated with several feature implementations.

For the MTG example, the AOC-poset of Figure 8 shows five non comparable concepts (that correspond to five distinct features) mined from a single hybrid block (*Concept_1* from Figure 5). The same feature documentation process is used for each hybrid block.

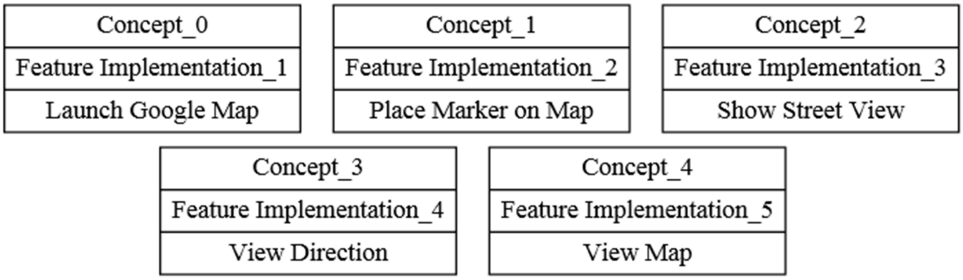


Fig. 8. The documented features from *Concept_1*.

6. Naming Feature Implementation Based on SCE Names

In our approach, we consider that use-case diagrams or other kinds of documentation (*i.e.*, design documents) are not always available. In case they are not, we use the source code of the mined features to automatically generate feature names and documentations.

We adapt the process proposed in [5]. Their work identifies component names based on *class names* in a single software. In our work, we extract a *name* for each *feature implementation* from the names given to its *SCEs*. We identify the name in three steps:

1. Extracting and tokenizing SCE names.
2. Weighting tokens.
3. Composing the feature name.

Our approach can be applied at any code granularity level (package, class, attribute, method, local variable, method invocation or attribute access).

- **Extracting and tokenizing SCE names.** At this step, the names of all the SCEs found in the feature implementation are extracted. Then, each SCE name is split into tokens, using a camel-case scheme. For example *getMinimumSupport* is split into *get*, *Minimum* and *Support*. This is a simple but common identifier splitting algorithm [32], conforming to programming best practices.
- **Weighting tokens.** At this step, a weight is assigned to each extracted token. A *high weight* (1.0) is given to the first word in a SCE name. A *medium weight* (0.7) is given to the second word in a SCE name. Finally a *small weight* (0.5) is given to the other words.
- **Composing the feature name.** In this step, a feature name is built using the highest weighted words.

The number of words used in the feature name is chosen by an *expert*. For example, the expert can select the top two words to construct the feature name. When many tokens have the same weight, all possible combinations are presented to the expert and he can choose the most relevant one. Table 7 shows an example

of feature name proposals for the *show street view* feature implementation. In this example, the expert assigns a feature name based on the top three tokens. The assigned name for this feature implementation is eventually *StreetShowView*.

Table 7. SCE names, tokens, weight and highest weighted tokens for the *show street view* feature implementation.

SCE Name	Token/Weight			
	T1/ w = 1.0	T2/ w = 0.7	T3/ w = 0.5	T4/ w = 0.5
ShowStreetView	show	Street	View	
StreetPosition	Street	Position		
ChangeStreetSettings	Change	Street	Settings	
getStreetAddress	get	Street	Address	
setStreetAddress	set	Street	Address	
ShowNearestStreet	show	Nearest	Street	
ShowNextStreet	show	Next	Street	
retrieveStreetData	retrieve	Street	Data	
ShowStreet	show	Street		
updateStreetInfo	update	Street	Info	
ViewStreetMap	View	Street	Map	
ViewStreetPositionInfo	View	Street	Position	Info

Token	Total Weight	Top 3	Top 4
Show	4	×	×
Street	8	×	×
View	2.5	×	×
Position	1.2		×
Change	1		
Settings	1		
get	1		
Address	1		
set	1		
Nearest	0.7		
Next	0.7		
retrieve	1		
Data	0.5		
update	1		
Info	1		
Map	0.5		

7. Experimentation

7.1. *Experimental setup*

To validate our approach, we ran experiments on two Java open-source applications: Mobile media software variants (small systems) [33] and ArgoUML-SPL (large systems) [34]. We used 4 variants for Mobile media and 10 for ArgoUML. These two case studies interestingly implement variability at different levels: class and method levels. In addition, these case studies are well documented: their use-case diagrams and FMs are available for comparison and validation of our results[g]. Table 8 summarizes the obtained results.

7.2. *Results*

In these two case studies, we observe that the *recall* values are 100% for all features: this means that our approach efficiently associates uses cases with their implementations in the software. Precision values are in [50% - 100%]: similarity between a use-case and several features implementation may be high, when they pertain to the same application domain and thus use common vocabulary, leading to ill associations. F-Measure values are consequently in [66% - 100%]. In most cases, the

[g]Case studies and code : `http://www.lirmm.fr/CaseStudy`

Table 8. Features documented from case studies.

#	Feature Name	Hybrid block #	k-Topics	Evaluation Metrics		
				Recall	Precision	F-Measure
Mobile Media						
1	Delete Album	HB_1	4	100%	100%	100%
2	Delete Photo	HB_1	4	100%	50%	66%
3	Add Album	HB_1	4	100%	100%	100%
4	Add Photo	HB_1	4	100%	50%	66%
5	Exception handling	HB_2	1	100%	100%	100%
6	Count Photo	HB_3	3	100%	50%	66%
7	View Sorted Photos	HB_3	3	100%	50%	66%
8	Edit Label	HB_3	3	100%	100%	100%
9	Set Favourites	HB_4	2	100%	50%	66%
10	View Favourites	HB_4	2	100%	50%	66%
ArgoUML-SPL						
1	Class diagram	HB_1	1	100%	100%	100%
2	Logging	HB_2	2	100%	50%	66%
3	Cognitive support	HB_2	2	100%	100%	100%
4	Deployment diagram	HB_3	1	100%	100%	100%
5	Collaboration diagram	HB_4	2	100%	50%	66%
6	Sequence diagram	HB_4	2	100%	50%	66%
7	State diagram	HB_5	1	100%	100%	100%
8	Activity diagram	HB_6	2	100%	100%	100%
9	Use case diagram	HB_6	2	100%	100%	100%

contents of hybrid blocks are in the range of $[1-4]$ use-cases and feature implementations: this validates RCA as a valid technique for building small search spaces in order to efficiently compute lexical similarity. Lexical similarity also proves to be a suitable tool, as shown by high recall, which confirms that a common vocabulary is used in use-case descriptions and feature implementations.

In our work, we cannot use a fixed number of topics for LSI because hybrid blocks (clusters) have different sizes. The column (*k-Topics*) in Table 8 represents *the number of term-topics*.

All feature names produced by our approach are presented in the column (*Feature Name*) of Table 8, as built from names of use-cases. For example, in the FM of Mobile media [33] there is a feature called *sorting*. The name proposed by our approach for this feature is *view sorted photos* and its description is "*the device sorts the photos based on the number of times photo has been viewed*".

7.3. *Threats to validity*

There is a limit to the use of FCA as a clustering technique. Cosine similarity matrices are transformed into formal (binary) contexts thanks to a fixed threshold.

So if similarity between the query and the document is greater than or equals 0.70 the two documents are considered similar. By contrast, if similarity is less than the threshold (*i.e.*, 0.69) the two documents are considered dissimilar. This sharp threshold effect may affect the quality of the result, since a similarity value of 0.99 (*resp.* 0.69) is treated as a similarity value 0.70 (*resp.* 0). Adaptive and fuzzy threshold schemes should be studied to improve precision without impacting the high recall of our approach.

In our approach we consider that each use-case corresponds to a functional feature. However, several use-cases may be implemented by a single feature. In this case all these use-cases should be considered as relevant documentation for this feature. Our approach should be improved with other techniques to extract a unique name and a compound description.

Naming a feature using the names of the SCEs in its implementation is not always reliable. In our approach, we rely on the top word frequencies to compose the proposed name. However, top words may be not relevant to depict the function of the feature. The weighting scheme should take into account the different roles and importance of SCEs in the implementation in order to select the most relevant words, if they are not the most frequent ones.

8. Conclusion and Perspectives

In this paper, we propose an approach for documenting automatically a set of feature implementations mined from a set of software variants. We exploit commonalities and variabilities between software variants at feature implementation and use-case levels in order to apply IR methods in an efficient way. We have implemented our approach and evaluated its results on two case studies. The good results (high recall) validates the main principles of our approach. Regarding future work, we would like to improve the clustering into hybrid blocks using other techniques. We would also like to improve the precision and relevance of our results thanks to adaptive similarity thresholds and semantic weighting schemes. We also plan to extract relations between the mined and documented features and automatically build a FM in order to support a complete reverse engineering process from source code to a SPL.

Acknowledgements

This work has been supported by project CUTTER ANR-10-BLAN-0219.

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. The SEI series in software engineering. Addison Wesley Professional, 2002.

[2] Y. Xue, Z. Xing, and S. Jarzabek, "Feature location in a collection of product variants," in *WCRE*. IEEE, 2012, pp. 145–154.

[3] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing," in *Proceedings of The 25th International Conference on Software Engineering and Knowledge Engineering*, 2013, pp. 244–249.

[4] H. A. Müller, S. R. Tilley, and K. Wong, "Understanding software systems using reverse engineering technology perspectives from the rigi project," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, ser. CASCON '93. IBM Press, 1993, pp. 217–226.

[5] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," in *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, ser. WICSA-ECSA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 181–190.

[6] A. Kuhn, "Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code," in *MSR*, M. W. Godfrey and J. Whitehead, Eds. IEEE, 2009, pp. 175–178.

[7] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[8] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *ICPC*, D. Beyer, A. van Deursen, and M. W. Godfrey, Eds., 2012, pp. 193–202.

[9] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 223–226.

[10] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a wordnet-like identifier network from software," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 4–13.

[11] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.

[12] M. Grechanik, K. S. McKinley, and D. E. Perry, "Recovering and using use-case-diagram-to-source-code traceability links," in *ESEC/SIGSOFT FSE*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 95–104.

[13] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[14] D. Diaz, G. Bavota, A. Marcus, R. Oliveto, S. Takahashi, and A. D. Lucia, "Using code ownership to improve ir-based traceability link recovery," in *ICPC*, 2013, pp. 123–132.

[15] A. Bragança and R. J. Machado, "Automating mappings between use case diagrams and feature models for software product lines," in *SPLC*. IEEE, 2007, pp. 3–12.

[16] Y. Yang, X. Peng, and W. Zhao, "Domain feature model recovery from multiple applications using data access semantics and Formal Concept Analysis," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 215–224.

[17] P. Paškevičius, R. Damaševičius, E. karčiauskas, and R. Marcinkevičius, "Automatic extraction of features and generation of feature models from Java programs," *Information Technology and Control*, pp. 376 – 384, 2012.

[18] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *CSMR*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE, 2012, pp. 417–422.

[19] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Feature location in a collection of software product variants using Formal Concept Analysis," in *ICSR*, ser. Lecture Notes in Computer Science, J. M. Favaro and M. Morisio, Eds., vol. 7925. Springer, 2013, pp. 302–307.

[20] R. Al-Msie'deen, A.-D. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity," in *IRI*. IEEE, 2013, pp. 586–593.

[21] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *ESEC/SIGSOFT FSE*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 290–300.

[22] B. Ganter and R. Wille, *Formal concept analysis - mathematical foundations*. Springer, 1999.

[23] T. Tilley, R. Cole, P. Becker, and P. W. Eklund, "A survey of formal concept analysis support for software engineering activities," in *Formal Concept Analysis*, ser. Lecture Notes in Computer Science, B. Ganter, G. Stumme, and R. Wille, Eds., vol. 3626. Springer, 2005, pp. 250–271.

[24] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux, "Formal Concept Analysis enhances fault localization in software," in *ICFCA*, ser. Lecture Notes in Computer Science, R. Medina and S. A. Obiedkov, Eds., vol. 4933. Springer, 2008, pp. 273–288.

[25] M. U. Bhatti, N. Anquetil, M. Huchard, and S. Ducasse, "A catalog of patterns for concept lattice interpretation in software reengineering," in *SEKE*. Knowledge Systems Institute Graduate School, 2012, pp. 118–123.

[26] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev, "Relational concept discovery in structured datasets," *Ann. Math. Artif. Intell.*, vol. 49, no. 1-4, pp. 39–76, 2007.

[27] S. Prediger and R. Wille, "The lattice of concept graphs of a relationally scaled context," in *ICCS*, ser. Lecture Notes in Computer Science, W. M. Tepfenhart and W. R. Cyre, Eds., vol. 1640. Springer, 1999, pp. 401–414.

[28] S. Ferré, O. Ridoux, and B. Sigonneau, "Arbitrary relations in Formal Concept Analysis and logical information systems," in *ICCS*, ser. Lecture Notes in Computer Science, F. Dau, M.-L. Mugnier, and G. Stumme, Eds., vol. 3596. Springer, 2005, pp. 166–180.

[29] F. Baader and F. Distel, "A finite basis for the set of EL-implications holding in a finite model," in *ICFCA*, ser. Lecture Notes in Computer Science, R. Medina and S. A. Obiedkov, Eds., vol. 4933. Springer, 2008, pp. 46–61.

[30] D. Grossman and O. Frieder, *Information Retrieval: Algorithms and Heuristics*, ser. Kluwer international series in engineering and computer science. Springer, 2004.

[31] M. Berry and M. Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval*, ser. ITPro collection. Society for Industrial and Applied Mathematics, 1999.

[32] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ser. ICPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 11–20.

[33] L. P. Tizzei, M. O. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: Assessing design stability of a software product line," *Information & Software Technology*, vol. 53, no. 2, pp. 121–136, 2011.

[34] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *CSMR*, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE, 2011, pp. 191–200.

[35] R. Medina and S. A. Obiedkov, Eds., *Formal Concept Analysis, 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4933. Springer, 2008.

Annexe D

Formal concept analysis-based service classification to dynamically build efficient software component directories

International Journal of General Systems
Vol. 00, No. 00, October 2008, 1–23

Research paper

FCA-based service classification to dynamically build efficient software component directories

Gabriela Arévalo[a*], Nicolas Desnos [b], Marianne Huchard [c],

Christelle Urtado [b] and Sylvain Vauttier [b]

[a] *LIFIA - Facultad de Informática (UNLP) - La Plata - Argentina*;
[b] *LGI2P / Ecole des Mines d'Alès - Nîmes - France*;
[c] *LIRMM - CNRS and Univ. Montpellier - France*

(*released August 2008*)

Component directories index components by the services they offer thus enabling us to rapidly access them. Component directories are also the cornerstone of dynamic component assembly evolution when components fail or when new functionalities have to be added to meet new requirements. This work targets semi-automatic evolution processes. It states the theoretical basis of on-the-fly construction of component directories using Formal Concept Analysis based on the syntactic description of the services that components require or provide. In these directories, components are more clearly organized and new abstract and highly reusable component external descriptions suggested. Moreover, this organization speeds up both automatic component assembly and automatic component substitution.

Keywords: Component-Based Software Engineering, Component directories, Formal Concept Analysis, Component classification

1. Introduction

Component-based software engineering (CBSE) enables software applications to be built by assembling off-the-shelf components. To ease this process, components expose their external description: a component's set of required and provided interfaces corresponds to the syntactical description of the services the component provides to other components in its environment or requires from other components of its environment to execute itself. Previous work on automatic component assembly and dynamic component assembly evolution (Desnos *et al.* 2006, 2007, 2008) convinced us that an efficient component directory is needed. Indeed, searching in a directory for a component from a given repository that is compatible with, or substitutable for, a given component is a non-trivial task. Additionally, white-page-like directories, which represent the mostly used category of directories, are not suitable because they are not structured to enable the search for compatible or substitutable components.

The idea of this paper is to propose mechanisms to semi-automatically index software components through a yellow-page-like component directory that supports

*Corresponding author. Email: Gabriela.Arevalo@lifia.info.unlp.edu.ar

ISSN: 0308-1079 print/ISSN 1563-5104 online

DOI: 10.1080/0308107YYxxxxxxxx
http://www.informaworld.com

efficient search for components that are compatible or substitutable to a given component. Our approach relies on Formal Concept Analysis (FCA) that enables us to pre-calculate three categories of lattices:

- *Functionality signature lattices* order functionality signatures in a way that naturally eases their search and can be used for required and provided functionality connection or for required or provided functionality substitution. This category of lattices serves as the basis for building interface lattices.
- *Interface lattices* are more abstract than functionality signature lattices; they code information on functionality specialization that has been modeled in functionality signature lattices. They order component interfaces — organize service descriptions — in a way that naturally eases their search and can be used for required and provided interface connection or for required or provided interface substitution. This category of lattices serves as the basis for building component type lattices.
- *Component type lattices* are more abstract than interface lattices; they code the information on interface specialization that has been modeled in interface lattices. They order component types in a way that naturally eases their search and can be used for component connection or component substitution.

These lattices provide the architect or developer with intelligible classifications for functionality signatures, interfaces and component types. They enable us to separate the service compatibility calculus from the component search itself during the processes of assembly or component assembly evolution (component substitution).

Indeed, a component type lattice can be used as an index for the search of a compatible component (in order to build an assembly) or of a comparable component (in order to find a substitute). Furthermore, FCA creates new component external descriptions (new component types) that do not exist in the component repository but are more abstract and reusable than existing components. These new abstractions can be an opportunity for component developers to be guided during their engineering or re-engineering process. They can also enrich the repository.

The remainder of this paper is organized as follows. Section 2 shows an extension of object-oriented type theory to component types. Then, after recalling the basics of FCA and describing the example used in the paper, Section 3 shows how to build a lattice of functionality signatures and how to use it as a basis for component assembly or component substitution. Section 4 generalizes these results to entire interfaces and shows how to use the resulting interface lattice. Section 5 goes one step further in proposing a methodology to build and interpret a component lattice. To finish, Section 6 compares our approach to related existing work and Section 7 concludes and presents future research directions.

2. Functionality signatures and interface syntactical compatibility

This section explains how the syntactical compatibility of component interfaces can be calculated from functionality signatures which define the syntactical type of interfaces. The syntactical compatibility of interfaces is used to check the validity of connection and substitution operations on component assemblies. It statically asserts a certain level of coherence in a component assembly that, before semantic analysis or execution, provides early error detection and correction.

2.1 *Functionality signature compatibility in object-oriented programming*

In strongly-typed object-oriented programming languages (Cardelli 1984), method signature overriding is allowed in subclasses but constrained by rules that enforce

the substitutability of subclass instances towards superclass instances. Thus, a method signature in a subclass must have contravariant argument types and a covariant return type: argument types must be generalized and the return type must be specialized. Intuitively, a method implements a service provided by an object: when the method is called, assuming that sufficient information is received (as specified by argument types), a result of the defined return type is sent back. This corresponds to the concept of software contract, introduced by Meyer (1991) to reason about interactions between objects. Following the above rules, an instance of a class can replace an instance of one of its superclasses because it provides at least the same services, but is allowed to require less invocation information and to return a richer result.

These principles are also used to define relaxed matching schemes used to retrieve a class or a functionality from a repository (Zaremski and Wing 1995). A request is expressed as the signature of the functionality that is searched for. Any functionality the signature of which specializes (overrides) the requested signature is returned as an approximate but still (type-) compatible answer.

2.2 *Functionality signatures and component interface specification*

An interface is a type that collects functionality signatures; it is used to qualify the collaborations a component can establish with other components. An interface is also a communication point through which a component exchanges service request and response messages with another component. Messages are sent and received along connections linking the interfaces of a component to compatible interfaces of other components (Szypersky *et al.* 2002). Comparing the syntactical types of two interfaces amounts to compare pairs of functionality signatures from both interfaces (Zaremski and Wing 1997). But in contrast with object models, a direction is added to the definition of interfaces in order to specify whether a component is a client (*i.e.,* uses the interface to require a service) or a server (*i.e.,* uses the interface to provide a service). Thus, two kinds of compatibilities can be verified between interfaces: a connection compatibility between a client interface and a server interface or a substitution compatibility between interfaces that have the same direction. The connection or substitution compatibility of two components can in turn be determined by verifying the connection or substitution compatibility of pairs of interfaces from both components.

In this paper, functionality signatures are defined by a name, a list of argument types and a return type. As in classical programming languages, names are used as the primary semantic element to match functionalities. Then, the types of the IN-parameters and OUT-parameters of homonymic functionalities are considered. For the sake of simplicity, only a single OUT-parameter (the functionality result) is used in this paper. But the same principles can be applied to any OUT-parameter when multiple OUT-parameters are used in a functionality signature.

Figure 1 shows an example of different signatures for homonymic functionalities named `create`, associated with both required and provided component interfaces. The data type hierarchy used to define parameter types is presented in Figure 1(d). The different cases of functionality signature specialization are illustrated: argument type specialization (*cf.* Figure 1(a)), result type specialization (*cf.* Figure 1(b)) and argument addition into the IN-parameter set (*cf.* Figure 1(c)).

When associated with a provided interface, a functionality signature has the same semantics as in object-oriented programming: the argument types define what the server component requires to receive in order to execute its service and the return type defines what result it commits to provide. When associated with a required

interface, a functionality signature specifies the service that is searched for by a client component: the argument types define the invocation information that the client component will send to a server component and the return type defines the type of the result it requires.

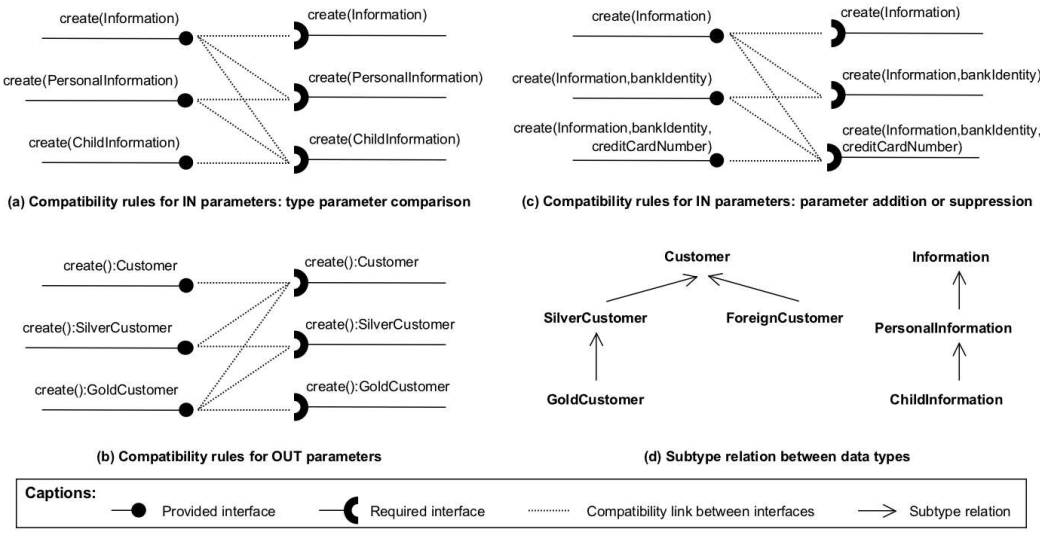


Figure 1. Interface compatibility when types and number of parameters vary.

2.3 *Functionality signature specialization and provided interface substitution*

Zaremski and Wing (1997) present functionality signature matching based on pre- and post- conditions. Consider a provided interface I_1, which holds a functionality of signature $S = f(X\ x) : Z$. As informally stated above, its software contract corresponds to the following pre-condition and post-condition:

$$S_{pre}(x) : Type(x) \leq X$$
$$S_{post}(x) : Type(f(x)) \leq Z$$

Let us consider another provided interface I_2, which holds a functionality of signature $T = f(L\ l) : M$, along with its pre-condition and post-condition:

$$T_{pre}(l) : Type(l) \leq L$$
$$T_{post}(l) : Type(f(l)) \leq M$$

To soundly substitute I_2 to I_1 in an assembly, the following predicate must hold:

$$Substitution_{provided}(I_2, I_1) = S_{pre}(x) \Rightarrow T_{pre}(x) \wedge T_{post}(x) \Rightarrow S_{post}(x)$$

This verifies that f in I_2 can execute the same invocations as f in I_1; second, it verifies that the results returned by f in I_2 can be used instead of the results returned by f in I_1.

To be true, the predicate entails that:

$$X \leq L \text{ (indeed, } Type(x) \leq X \wedge X \leq L \Rightarrow Type(x) \leq L),$$
$$M \leq Z \text{ (indeed, } Type(f(x)) \leq M \wedge M \leq Z \Rightarrow Type(f(x)) \leq Z).$$

This respectively corresponds to a contravariant specialization of the argument types and to a covariant specialization of the result type between the two functionality signatures, as previously presented for object-oriented languages. A provided

interface can be replaced by another provided interface with more specific functionality signatures, following the above specialization rules.

For example (*cf.* Figure 1(a)), a provided interface holding the `create(Information)` signature can be substituted to a provided interface holding the `create(PersonnalInformation)` signature (contravariant specialization of the argument type). Similarly (*cf.* Figure 1(b)), a provided interface holding the `create():GoldCustomer` signature can be substituted to a provided interface holding the `create():SilverCustomer` signature (covariant specialization of the result type).

2.4 *Functionality signature specialization and required interface substitution*

Let us now consider a required interface I_3, which holds a functionality of signature $S = f(X\ x) : Z$. The pre-condition and post-condition corresponding to its software contract are the same as for a provided interface but, as discussed above, their semantics are converse. Indeed, x now represents the data the client component commits to send and $f(x)$ the data the client expects to receive:

$$S_{pre}(x) : Type(x) \leq X$$
$$S_{post}(x) : Type(f(x)) \leq Z$$

Let us also consider another required interface I_4, which contains a functionality of signature $T = f(L\ l) : M$. This corresponds to the same pre-condition and post-condition as above:

$$T_{pre}(l) : Type(l) \leq L$$
$$T_{post}(l) : Type(f(l)) \leq M$$

To soundly substitute I_4 to I_3 in an assembly, the following predicate must hold:

$$Substitution_{required}(I_4, I_3) = T_{pre}(x) \Rightarrow S_{pre}(x) \wedge S_{post}(x) \Rightarrow T_{post}(x)$$

This firstly verifies that the client component holding I_4 will call f in the same way as the client component holding I_3 (to have the guarantee that the connected server component can execute all invocations); secondly, this verifies that the results received by I_3 will also satisfy the requirements of the client component holding I_4.

To be true, the predicate entails that:

$$L \leq X \text{ (indeed, } Type(x) \leq L \wedge L \leq X \Rightarrow Type(x) \leq X),$$
$$Z \leq M \text{ (indeed, } Type(f(x)) \leq Z \wedge Z \leq M \Rightarrow Type(f(x)) \leq M).$$

This respectively corresponds to a covariant specialization of the argument types and a contravariant specialization of the result type between the two functionality signatures. Unsurprisingly, the specialization rules for functionality signatures in required interfaces are the opposite of those which apply to provided interfaces. Here again, following the above rules, a required interface can be replaced by another required interface with more specific functionality signatures.

For example (*cf.* Figure 1(a)), a required interface holding the `create(ChildInformation)` signature can be substituted to a required interface holding the `create(PersonnalInformation)` signature (covariant specialization of the argument type). Similarly (*cf.* Figure 1(b)), a required interface holding the `create():Customer` signature can be substituted to a required interface holding the `create():SilverCustomer` signature (contravariant specialization of the result type).

2.5 *Functionality signature specialization and interface connection*

Finally, let us again consider the provided interface I_1 and the required interface I_4. To soundly connect I_1 to I_4, the following predicate must hold:

$$Connection(I_4, I_1) = T_{pre}(x) \Rightarrow S_{pre}(x) \wedge S_{post}(x) \Rightarrow T_{post}(x)$$

This firstly verifies that any data sent by the client component holding I_4 can effectively be used by the server component holding I_1 to execute f; secondly, this verifies that the data sent by the server component holding I_1 corresponds to the result expected by the client component holding I_4.

To be true, the predicate entails that:

$$L \leq X \text{ (indeed, } Type(x) \leq L \wedge L \leq X \Rightarrow Type(x) \leq X),$$
$$Z \leq M \text{ (indeed, } Type(f(x)) \leq Z \wedge Z \leq M \Rightarrow Type(f(x)) \leq M).$$

This corresponds to a contravariant specialization of argument types and a covariant specialization of the result type between the two functionality signatures. The functionality signatures associated with a required interface of the client component must be more generic than the functionality signature associated with the provided interface of the server component.

For example (*cf.* Figure 1(a)), a required interface holding the `create(PersonalInformation)` signature can be connected to a provided interface holding the `create(Information)` signature (contravariant specialization of the argument type). Similarly (*cf.* Figure 1(b)), a required interface holding the `create():Customer` signature can be connected to a provided interface holding the `create():SilverCustomer` signature (covariant specialization of the result type).

2.6 *Functionality signature specialization and parameter addition or suppression*

A special case of parameter type generalization is now considered. When a parameter type is generalized in a functionality signature, it conceptually means that the specification becomes less demanding on parameters. The objects of the `Object` type (root of the object type hierarchy) are the objects which contain the least data. We extend the generalization principle by stating that `void` is the root type in our system and that it further generalizes the `Object` type.

This way, a special case of parameter type generalization is to set a parameter type to `void`. Any data, including no data, becomes suitable for this parameter. As this parameter is optional, it is possible to remove the parameter from the functionality signature. We therefore consider suppressing a parameter as a special case of parameter type generalization.

Conversely, it is possible to add an extra parameter of type `void` to a functionality signature without changing its semantics (this additional parameter can always be ignored). The type of such a parameter can then be specialized in the process of functionality signature specialization, thus becoming a parameter of a concrete type. We therefore consider parameter addition as a special case of parameter type specialization.

For example (*cf.* Figure 1(c)), a provided interface holding the `create(Information)` signature can be substituted to a provided interface holding the `create(Information,BankIdentity)` signature, as the former signature is obtained by removing the second parameter of the latter signature (contravariant specialization of the parameter type). Similarly, a required interface holding the `create(In-`

`formation,BankIdentity)` signature can be substituted to a required interface holding the `create(Information)` signature, as the former signature is obtained by adding a second parameter to the latter signature (covariant specialization of a virtual second parameter of type `void`).

2.7 *Discussion*

In Zaremski and Wing (1997), which proposes an extensive study and classification of functionality signature matching, the above predicates correspond to a kind of functionality signature matching called "plug-in" matching. It is used to verify that the code of a functionality can be plugged into some other code, to handle some expected behavior, as specified by a syntactical signature. We have adapted this generic functionality signature matching principle to the specific concepts of component models, namely the syntactical coherence of interface connection and substitution.

Our formalization shows that checking the coherence of these operations amounts to verifying the existence of specialization relations between functionality signatures. Thus, we studied how to build specialization hierarchies of functionality signatures, interfaces and component types. We intend to use these hierarchies as a practical, systematic and efficient means to set up and structure a component directory, where components are indexed by the type of services they provide and require, in other words, a trading service for component-based platforms (Iribarne *et al.* 2004)).

The next sections describe how an FCA-based approach to this problem can be used to build the necessary specialization lattices. It is to be noticed that, at any step, a single lattice is sufficient to compare both required and provided elements for both substitution and connection. Indeed, as shown previously, only two specialization rules are used, which are converse.

3. Lattice of functionality signatures

The substitutability rules presented in the previous section can be considered as the basis of a specialization relationship among functionalities: a functionality that can substitute for another can be considered as its specialization. Existing functionalities can thus be organized — classified — in a hierarchy based on their substitutability relationships. Furthermore, this section will show that FCA provides a finer-grained classification. After recalling the basics of Formal Concept Analysis, we show how it can be used to build a lattice of functionality signatures and how the lattice can then be interpreted and used.

3.1 *A survival kit for Formal Concept Analysis*

The classification we build is based on the partially ordered structure known as *Galois connection-based lattice* (Birkhoff 1940, Davey and Priestley 1991) or *concept lattice* (Wille 1982) which is induced by a context K, composed of a binary relation R over a pair of sets O (*objects*) and A (*attributes*) (Table 1). A formal concept C is a pair of corresponding sets (E, I) such that:

$$E = \{\ \mathrm{e} \in O |\ \forall\ \mathrm{i} \in \mathrm{I},\ (\mathrm{e},\ \mathrm{i}) \in R\} \quad \text{is called } \textit{extent} \text{ (covered objects)},$$
$$I = \{\ \mathrm{i} \in A |\ \forall\ \mathrm{e} \in \mathrm{E},\ (\mathrm{e},\ \mathrm{i}) \in R\} \quad \text{is called } \textit{intent} \text{ (shared features)}.$$

For example, $(\{1,2\},\{b,c\})$ is a formal concept because objects 1 and 2 exactly share attributes b and c (and vice-versa). On the contrary, $(\{2\},\{b,c\})$ is not a formal concept.

Furthermore, the set of all formal concepts $\mathcal{C}$ constitutes a lattice $\mathcal{L}$ when provided with the following specialization order based on intent / extent inclusion:

$$(E_1, I_1) \leq_{\mathcal{L}} (E_2, I_2) \Leftrightarrow E_1 \subseteq E_2 \text{ (or equivalently } I_2 \subseteq I_1).$$

Figure 3.1 shows the Hasse diagram of $\leq_{\mathcal{L}}$.

Table 1. Binary relation of $K = (O, A, R)$ where O=$\{1,2,3,4,5,6\}$ and A= $\{a,b,c,d,e,f,g,h\}$.

	a	b	c	d	e	f	g	h
1		×	×	×	×			
2	×	×	×				×	×
3	×	×				×	×	×
4				×	×			
5			×	×				
6	×							×

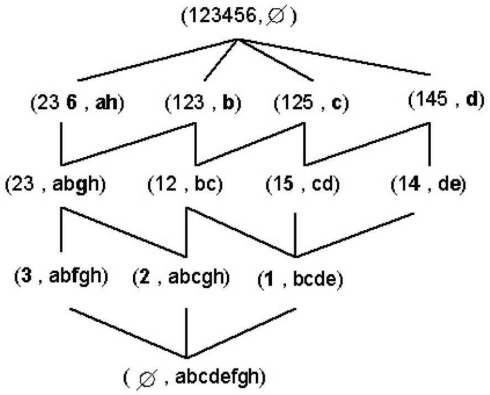


Figure 2. Hasse diagram of the concept lattice $\mathcal{L}$.

3.2 *Example of an online bookstore application*

In the rest of this article, we will use, as an illustration, the example of an online bookstore application that targets both the adult and children audiences (*cf.* Figure 3(a) to see the hierarchy of product types). Two categories of customers can interact with this application. Adults can save favorite book lists (as wish lists) through the application or shop for books following various protocols defined according to a client typology (*cf.* Figure 1(d)). Children can establish children book wish lists that constitute virtual orders that adults can offer them as soon as their parents obtain the `SilverCustomer` client category. For this online bookstore application, we have a component repository (*cf.* Figure 3(b)) in which we can see various components to manage orders (by adults or children) and various components to manage customer lists. These components each expose an interface list the types of which are enumerated in Figure 3(c).

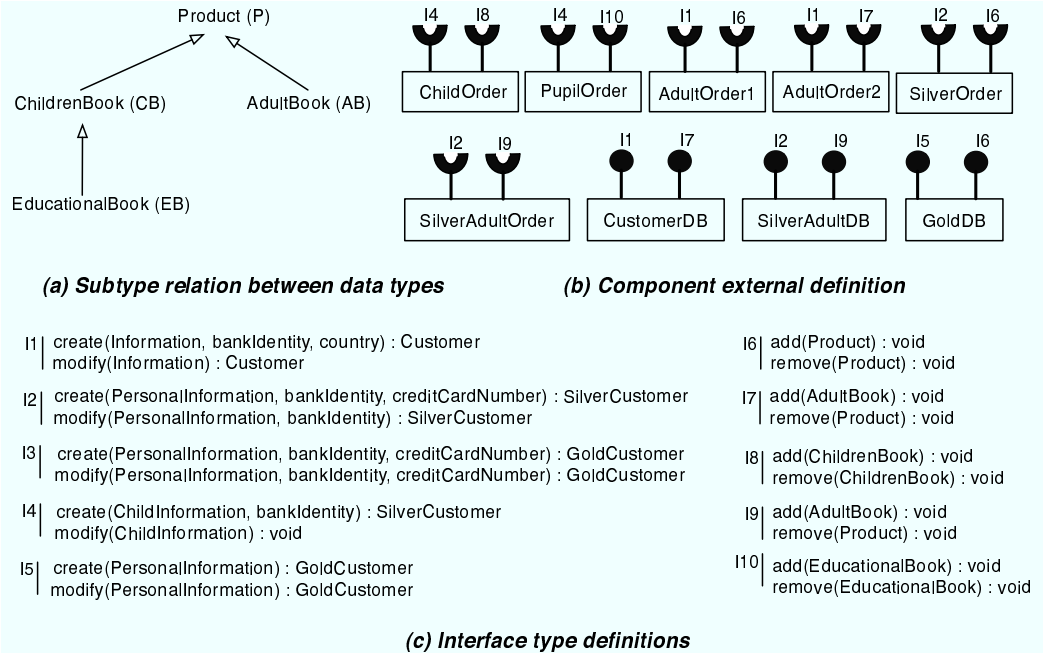


Figure 3. Data types, interfaces and components of an online bookstore application.

3.3 *Building the functionality signature lattice*

We explain here the construction of the required functionality signature lattice. As provided functionality signatures are reversely ordered, the lattice we obtain can also be used to deal with them, when considered upside down.

We illustrate our explanation considering the required functionality `create(PI, BI,CCN):SC` as it is described by Table 2. At first, for each `create` functionality whose signature is held by one of the interfaces of Figure 3, attributes are deduced from IN and OUT parameter types that explicitly appear in the signature. These attributes are marked using the $\times$ symbol in Table 2: `create(PI,BI,CCN):SC` is thus described explicitly by attributes `IN:PI`, `IN:BI`, `IN:CCN` and `OUT:SC`. Then, we infer attributes (marked with a $\otimes$ symbol in Table 2) when their types are compatible, regarding specialization of signatures. Here are our inference rules:

- IN parameters. As explained previously, if a required functionality sends a parameter of some type, it implicitly sends a parameter of any more general type. For example, the `IN:I` attribute is inferred when the `IN:PI` attribute is already present.
- OUT parameters. If a required functionality expects to receive a return value of a type, any return value of a more specific type is also suitable. For example, the `OUT:GC` attribute is inferred when the `OUT:SC` attribute is already present.

Figure 4 depicts the concept lattice corresponding to the binary relation shown in Figure 2, built with the GaLicia FCA tool (GaLicia 2002). Concepts are presented using reduced intents and extents (resp. denoted by *ReducedI* et *ReducedE*) for readability sake: an object (signature) that belongs to the reduced extent of a concept is inherited by all concepts that are above (down-to-up inheritance); similarly, a property (IN or OUT parameter type) that belongs to the reduced intent of a concept is inherited by all concepts that are below (up-to-down inheritance).

Table 2. R_{create} context describing signatures of the required `create` functionality through its parameters.

	IN parameters						OUT param.		
	I	PI	CI	BI	CCN	Co	C	SC	GC
create(I,BI,Co):C	×			×		×	×	⊗	⊗
create(PI,BI,CCN):SC	⊗	×		×	×			×	⊗
create(PI,BI,CCN):GC	⊗	×		×	×				×
create(CI,BI):SC	⊗	⊗	×	×				×	⊗
create(PI):GC	⊗	×							×

I	Information
PI	PersonalInfo.
CI	ChildInfo.
BI	BankIdentity.
CCN	CreditCardNb
Co	Country
C	Customer
SC	SilverCustomer
GC	GoldCustomer
FC	ForeignCustomer

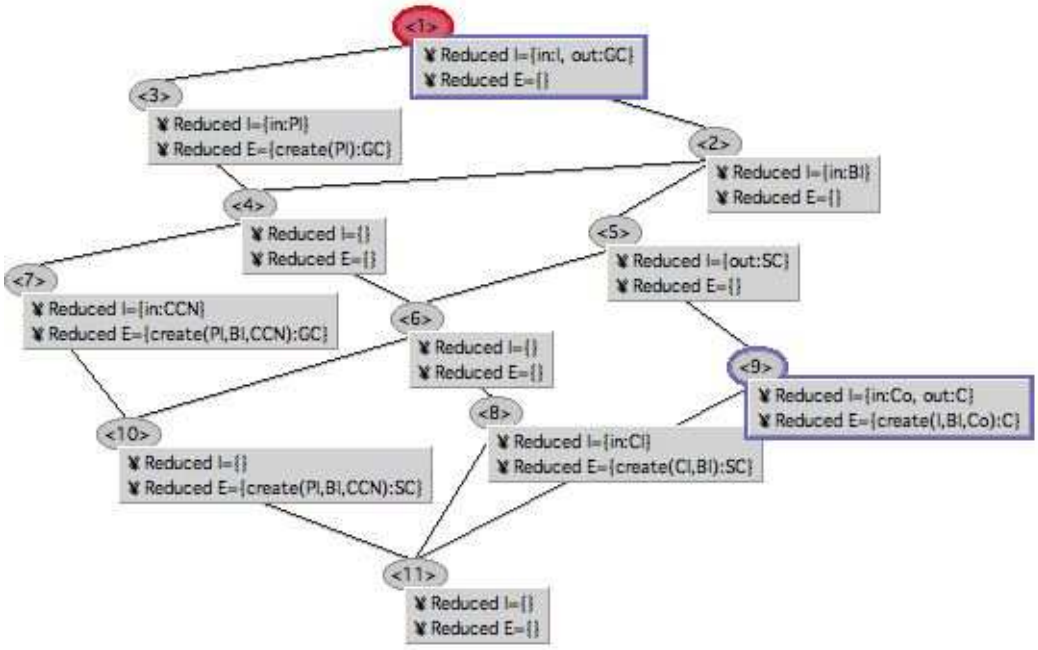


Figure 4. Signature lattice $\mathcal{L}_{create}$ for the `create` functionalities.

3.4 *Using the functionality signature lattice*

The functionality signature lattice can be used in various types of situations related to component connection or substitution.

Let us consider the lattice of Figure 4 with the viewpoint of required functionalities. In this lattice, `create(PI):GC` is represented by concept C_3 while `create(CI,BI):SC` is represented by concept C_8. Concept C_3 is more general than concept C_8 which can be interpreted as: concept C_8 can replace concept C_3. In a component assembly, a connection to a required functionality corresponding to concept C_3 can be replaced by a connection to a required functionality corresponding to concept C_8. In the general case, when there is a path between two concepts, the more specific (which has more properties) can replace the more general (which has a subset of properties) when the more general concept is connected (*cf.* Figure 5(a)). The same lattice can also be used to substitute a provided functionality when read upside down (*cf.* Figure 5(b)). This generalizes as follows.

Property 3.1 Functionality substitution. Let C_{father}, C_{son} be two concepts of the signature lattice of functionality f, such that $C_{son} \leq_{\mathcal{L}_f} C_{father}$. Functionalities of C_{son} can replace functionalities of C_{father} when the functionalities are required. Opposite replacement applies when the functionalities are provided.

Both provided and required points of view can be combined to address com-

ponent connection. Let us consider the `create(PI,BI,CCN):GC` signature (concept C_7). The corresponding required functionality can obviously connect to the provided functionality that has the same signature (`create(PI,BI,CCN):GC`). Given the substitution rule, provided functionalities which are upper in the lattice, such as provided `create(PI):GC` (concept C_3), can be connected to required `create(PI,BI,CCN):GC` (*cf.* Figure 5(c)). Using the same rule in the symmetric way, required functionalities which are below in the lattice, such as required `create(PI,BI,CCN):SC` (concept C_{10}), can be connected to provided `create(PI,BI,CCN):GC`. By transitivity, we can deduce that required `create(PI, BI,CCN):SC` can be connected to provided `create(PI):GC`. This is expressed in the following connection rule that formalizes how valid functionality connection can be deduced from the lattice.

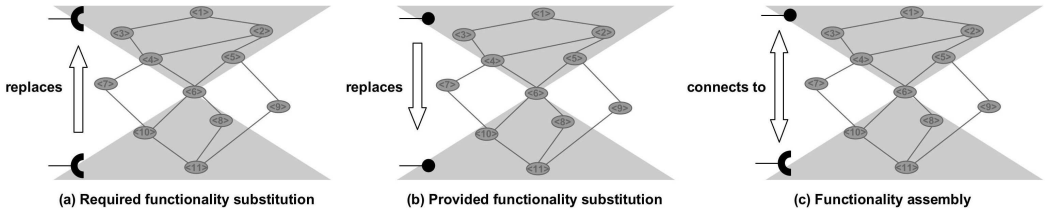


Figure 5. Interpretation of the lattice of functionality signatures.

Property 3.2 Functionality connection rule. Let C, C_{father}, C_{son} be three concepts of the signature lattice of functionality f such that $C_{son} \leq_{\mathcal{L}_f} C \leq_{\mathcal{L}_f} C_{father}$, required functionalities of C_{son} can be connected to provided functionalities of C_{father}.

4. Interface lattice

Components are reusable software entities that are chosen off-the-shelf and fulfill high-level goals (database component, planning component, and so on). Interfaces play an important role to achieve these goals by grouping functionalities that have close semantics and may participate together in potential collaborations. Component assembly is based mainly on the connection of compatible interfaces in a higher abstraction level than simple functionalities.

Considering included functionalities, the interfaces can be provided with a specialization order in a natural way. This "natural" classification simply uses the inclusion relation between sets of functionalities in the interfaces and can equally benefit from FCA to look for factorizable functionalities (in our case `remove(P)` can be factored out).

Then, if we consider substitution or connection, we can improve our search and discover more pertinent abstractions when using the abstractions discovered in the functionality signature lattice. Lattices of the `modify`, `add` and `remove` functionalities of our example are built similarly to the lattice of the `create` functionality. Tables 3 and 4 detail the contexts, while Figure 6 and 7 show the corresponding lattices. As we have observed, these abstractions on the signatures are the concepts the extent of which has a set of signatures (the signatures covered by the concept) and the intent of which has a set of attributes describing the signature (IN and OUT parameters). For each concept, we can calculate a corresponding canonical signature. We show an example before giving the general definition.

Figure 4 shows the concepts built using the binary relation described in Table 2. A concept the reduced extent of which has an original signature (*e.g.,* concept

Table 3. Context R_{modify} describing the signatures of the required `modify` functionalities.

	In parameters					Out param.			
	I	PI	CI	BI	CCN	void	C	SC	GC
modify(I):C	×						×	⊗	⊗
modify(PI,BI):SC	⊗	×		×				×	⊗
modify(PI,BI,CCN):GC	⊗	×		×	×				×
modify(CI):void	⊗	⊗	×			×	⊗	⊗	⊗
modify(PI):GC	⊗	×							×

Table 4. Context R_{add} describing the signatures of the required `add` functionalities. The context R_{remove} is identical.

	In parameters				Out param.
	P	AB	CB	EB	void
add(P):void	×				×
add(AB):void	⊗	×			×
add(CB):void	⊗		×		×
add(EB):void	⊗		⊗	×	×

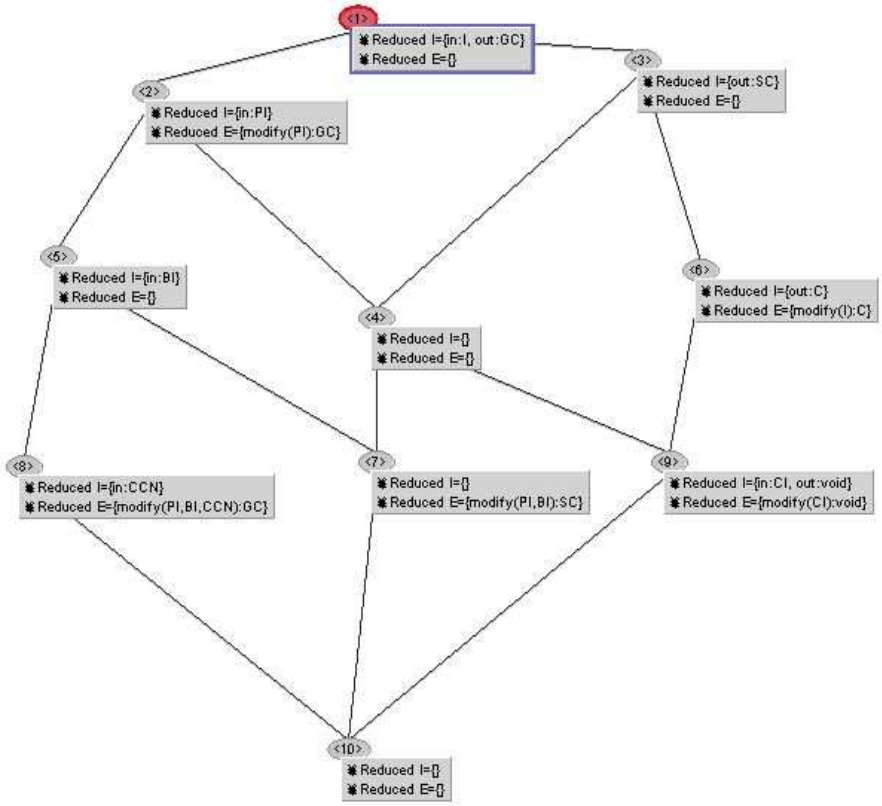


Figure 6. Signature lattice $\mathcal{L}_{modify}$ for the `modify` functionalities

C_9) exactly represents that signature (*e.g.,* `create(I,BI,Co):C`). A concept the reduced extent of which is empty can be interpreted as a new signature that we can infer starting from the attributes inherited by the concept, and considering only the more specific ones. For example, concept C_6 of Figure 4 inherits attributes `in:I`, `in:PI`, `in:BI`, `out:GC`, `out:SC`. In case of required signatures, `in:PI` is more specific than `in:I` meanwhile `out:SC` is more specific than `out:GC`. Concept C_6 can be then interpreted as signature `create(PI,BI):SC` which we call the canonical signature of the concept. This enables us to build an interface description based on the set of original signatures completed by all the signatures created in the generalization process (*cf.* Table 5).

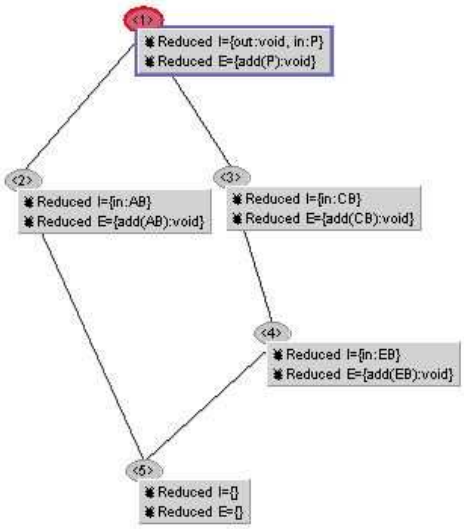


Figure 7. Signature lattice $\mathcal{L}_{add}$ for the `add` functionalities. The lattice $\mathcal{L}_{remove}$, isomorphic to $\mathcal{L}_{add}$, is not represented.

Definition 4.1 Canonical functionality signature of a concept**:** Let C be a concept in a signature lattice $\mathcal{L}_f$ which describes functionality f and $\leq_{Types}$, the specialization partial order on parameter types. $\sigma(C)$, the canonical signature of C, is defined as follows:

- If $ReducedE(C) = \{s\}$, then $\sigma(C) = s$.
- If $ReducedE(C) = \varnothing$, then $\sigma(C) = f(i) : o$, where $i = min_{\leq_{Types}}\{T|\text{IN} : T \in Intent(C)\}$ and where $o = max_{\leq_{Types}}\{T|\text{OUT} : T \in Intent(C)\}$.

This exact description enables us to build more pertinent interface generalizations than those we obtained with the "natural" classification of interfaces. It is used as follows to build interface descriptions within the new context $R_{IntSigCar}$.

- The canonical signatures are used as attributes in the formal context.
- When an interface I has a signature s in a functionality f in its original description, if we denote by C the concept such that $\sigma(C) = s$, we associate to the interface the attribute s and all the canonical signatures of the concepts that are upper of C in the lattice:
$R_{IntSigCar} = \{(I, sc)|s \textit{ belongs to the definition of } I, sc = \sigma(C_{father}),$
$C_{father} \geq_{\mathcal{L}_f} C \textit{ with } s = \sigma(C)\}$.

For example, interface `I1` holds the signature `create(I,BI,Co):C`. This signature is the canonical signature of concept C_9 in lattice $\mathcal{L}_{create}$. In Tab. 5, we associate `I1` to `create(I,BI,Co):C` (marked with symbol $\times$) and we equally associate to `I1` the canonical signatures of all concepts of $\mathcal{L}_{create}$ that are upper of C_9. That results in the following signatures (marked with symbol $\otimes$) : `create(I,BI):SC` (concept C_5), `create(I,BI):GC` (concept C_2), and `create(I):GC` (concept C_1). From required functionality viewpoint, these signatures are generalizations of the original signature `create(I,BI,Co):C` (with the semantics of substitutability).

The built lattice $\mathcal{L}_I$ (*cf.* Figure 8) shows specialization relations between interfaces. These relations show possible connections or substitutions which are deduced from the previously mentioned rules on functionality signatures that are extended to interfaces (repeatedly applied to all signatures that constitute these interfaces).

For example, the required interface `I10` can be connected to provided interface

Table 5. Context $R_{IntSigCar}$ encoding required interfaces using signature generalizations. Rows: interfaces. Columns: canonical signatures and concepts.

	create											modify									
	create(I):GC — C_1	create(I,BI):GC — C_2	create(PI):GC — C_3	create(PI,BI):GC — C_4	create(I,BI):SC — C_5	create(PI,BI):SC — C_6	create(PI,BI,CCN):GC — C_7	create(CI,BI):SC — C_8	create(I,BI,Co):C — C_9	create(PI,BI,CCN):SC — C_{10}	create(CI,BI,CCN,Co):C — C_{11}	modify(I):GC — C_1	modify(PI):GC — C_2	modify(I):SC — C_3	modify(PI):SC — C_4	modify(PI,BI):GC — C_5	modify(I):C — C_6	modify(PI,BI):SC — C_7	modify(PI,BI,CCN):GC — C_8	modify(CI):void — C_9	modify(CI,BI,CCN):void — C_{10}
I1	⊗	⊗			⊗				×			⊗		⊗			×				
I2	⊗	⊗	⊗	⊗	⊗	⊗	⊗			×		⊗	⊗	⊗	⊗	⊗		×			
I3	⊗	⊗	⊗	⊗			×					⊗	⊗			⊗			×		
I4	⊗	⊗	⊗	⊗	⊗	⊗		×				⊗	⊗	⊗	⊗		⊗			×	
I5	⊗		×									⊗	×								
I6																					
I7																					
I8																					
I9																					
I10																					

	add					remove				
	add(P):void — C_1	add(AB):void — C_2	add(CB):void — C_3	add(EB):void — C_4	add(AB,EB):void — C_5	remove(P):void — C_1	remove(AB):void — C_2	remove(CB):void — C_3	remove(EB):void — C_4	remove(AB,EB):void — C_5
I1										
I2										
I3										
I4										
I5										
I6	×					×				
I7	⊗	×				×				
I8	⊗		×			⊗		×		
I9	⊗	×				×				
I10	⊗		⊗	×		⊗		⊗	×	

`I6`. Still, required interface `I10` (C_{10}) can replace required interface `I6` (C_2) . We see that a manual or automatic search of components is faster with this lattice that defines a search index. We thus avoid looking at all components in the repository since we only look for relevant branches. Let us imagine the case in our example where component `SilverAdultOrder` searches, logically, to be connected to component `SilverAdultDB` usually present in the system that is temporarily unavailable. The relation in the lattice, starting from the expected required interface I_9 (C_5), enables us to immediately find (just traversing the edge that goes from concept C_5 to concept C_2, that possesses the I_6 interface) that component `GoldDB` could be used as a replacement. Temporarily the user will benefit of a higher service in replacement of a missing service.

In the lattice, we also find new interfaces, obtained using the existing interface generalization. Starting from functionalities discovered in the first lattice, the technique can then infer a new interface, including at least this shared functionality. Here we see one of the main advantages of FCA-based techniques compared to simple calculation of signature comparison: new signatures appear, and thus we have new interfaces more abstract than existing ones. The following generalization step is to use this lattice to build a component lattice. This latter lattice is more interesting for designers who can be guided when creating more general new components, as well as for assemblers who can consult an organized library rather than

just a flat set of artifacts.

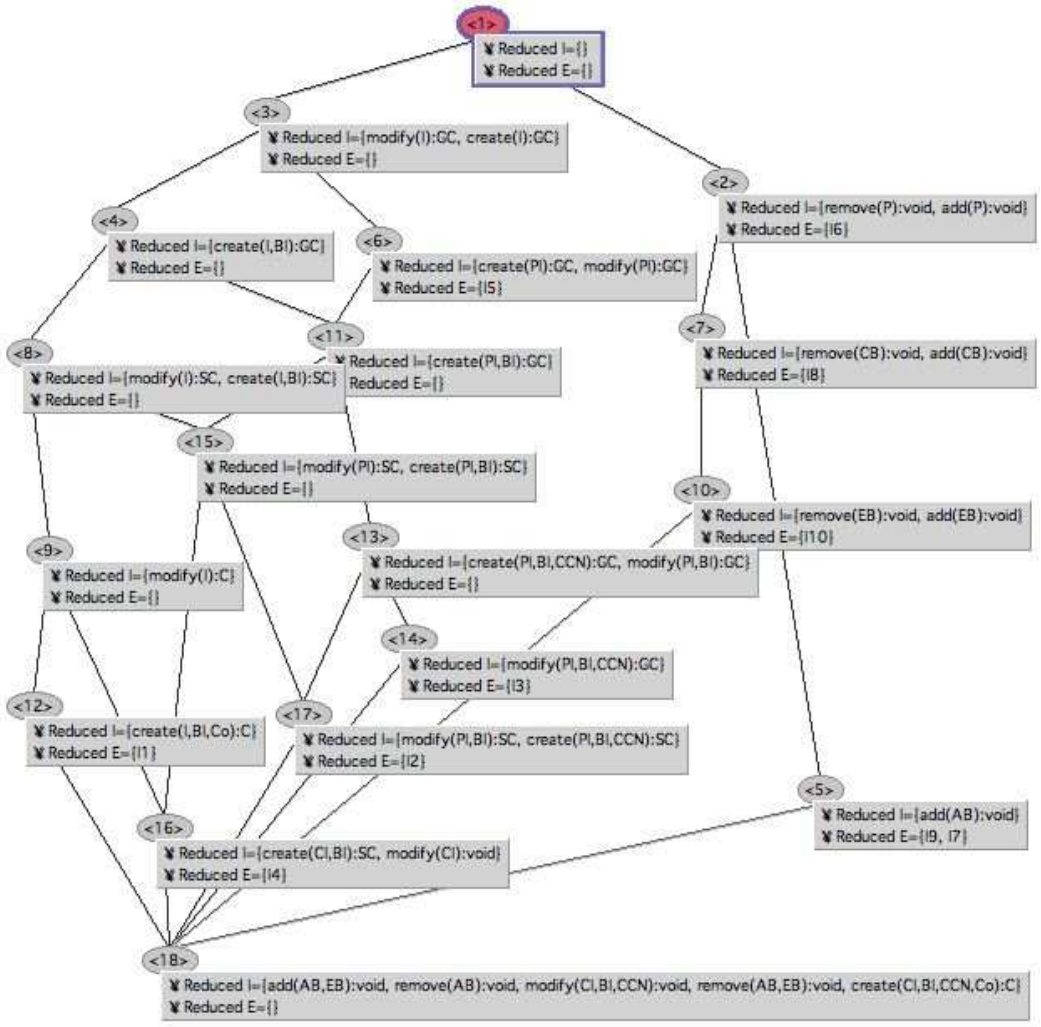


Figure 8. Interface lattice $\mathcal{L}_I$ using the functionality signature lattice.

5. Lattice of component types

In this section, we first propose a solution to build the lattice of component types. The technique used to do so is the same as the one previously used for interfaces: the interface lattice helps enrich the description of the formal context that will be used to build the component type lattice. Then, the remainder of the section shows possible uses of this lattice.

5.1 *Definition of the lattice of component types*

Component types are described by their required and provided interfaces. This information can be organized by specialization, but, similarly to that done with interfaces, component types can benefit from both the specialization relationships between interfaces and the discovered interfaces obtained from the interface lattice. We thus get an enrichment of the description of components and a more precise classification, offering more abstractions.

The first phase of the building process entailed the introduction of the notion of a "canonical interface" associated to an interface concept. This notion is similar to the canonical functionality signature corresponding to a signature concept that we defined above. Let us just mention that our analysis is still based on the case of required interfaces.

Definition 5.1 Required canonical interface corresponding to an interface concept: Let C be a concept in the interface lattice $\mathcal{L}_I$. The corresponding canonical interface $I(C)$ is defined as follows:

- If $ReducedE(C) \supseteq \{I\}$, then $I(C) = I$. We can choose any interface in the reduced extent because they are all equivalent.
- If $ReducedE(C) = \varnothing$, then $I(C) = min_{\leq_{SigCar}}\{s \in Intent(C)\}$. The canonical interface gathers more specialized signatures from the set of canonical signatures that forms the intent. The order $\leq_{SigCar}$ between canonical signatures is naturally inferred from the specialization relationship between concepts of the lattice $\mathcal{L}_f$: $s_{son} \leq_{SigCar} s_{father}$ iff $s_{son} = \sigma(c_{son})$, $s_{father} = \sigma(c_{father})$ and $c_{son} \leq_{\mathcal{L}_f} c_{father}$.

Canonical interfaces found in the lattice are all the original interfaces (`I1` to `I6`, `I8` and `I10`, and a single interface corresponding to the {`I7`,`I9`} interface pair) to which new abstract interfaces are added by the classification process. These new abstract interfaces are described by their signature set (*cf.* Tab. 6).

Table 6. New canonical interfaces.

Int. name	Signature set	Concept
I11	{}	C_1
I12	$\{create(I) : GC; modify(I) : GC\}$	C_3
I13	$\{create(I, BI) : GC; modify(I) : GC\}$	C_4
I14	$\{create(I, BI) : SC; modify(I) : SC\}$	C_8
I15	$\{create(I, BI) : SC; modify(I) : C\}$	C_9
I16	$\{create(PI, BI) : GC; modify(PI) : GC\}$	C_{11}
I17	$\{create(PI, BI) : SC; modify(PI) : SC\}$	C_{15}
I18	$\{create(PI, BI, CCN) : GC; modify(PI, BI) : GC\}$	C_{13}
I19	$\{create(CI, BI, CCN, Co) : C; modify(CI, BI, CCN) : void;$ $add(AB, EB) : void; remove(AB, EB) : void\}$	C_{18}

We then set up a relation $R_{CompCanInt}$ between component types and canonical interfaces including their orientation (required or provided) (*cf.* Tab. 7). The rows represent components, the columns interfaces. Interface identification (in column heads) combines the two interface orientations (noted `req:` and `pro:`) with each canonical interface name and is followed by their concept number in the interface lattice. For example, column 1 corresponds to the canonical required interface `I1`, associated to concept C_{12} (as `I1` is member of its reduced extent). Column 11 corresponds to the canonical required interface `I12`, associated to concept C_3.

Definition 5.2 Component relation $R_{CompCanInt}$:
Component types are the formal objects while canonical interfaces are the formal attributes. Let C be a component and I an interface, $(C, I) \in R_{CompCanInt}$ iff one of the following properties is true:

- I is declared by C,
- $I \geq_{\mathcal{L}_I} J$ and J is declared by C.

Figure 9 shows lattice $\mathcal{L}_C$ of component types. The following section will show how it can be used.

Table 7. The component context $R_{CompCanInt}$.

	req:I1 - c_{12}	req:I2 - c_{17}	req:I3 - c_{14}	req:I4 - c_{16}	req:I5 - C_6	req:I6 - C_2	req:I7,I9 - C_5	req:I8 - C_7	req:I10 - C_{10}	req:I11 - C_1	req:I12 - C_3	req:I13 - C_4	req:I14 - C_8	req:I15 - C_9	req:I16 - C_{11}	req:I17 - C_{15}	req:I18 - C_{13}	req:I19 - C_{18}
CO				×	⊗	⊗		×		⊗	⊗	⊗	⊗	⊗	⊗	⊗		
PO				×	⊗	⊗		⊗	×	⊗	⊗	⊗	⊗	⊗	⊗	⊗		
AO1	×					×				⊗	⊗	⊗	⊗	⊗				
AO2	×					⊗	×			⊗	⊗	⊗	⊗	⊗				
SO		×			⊗	×				⊗	⊗	⊗	⊗		⊗	⊗	⊗	
SAO		×			⊗	⊗	×			⊗	⊗	⊗	⊗		⊗	⊗	⊗	
CDB																		
SDB																		
GDB																		

	pro:I1 - C_{12}	pro:I2 - C_{17}	pro:I3 - C_{14}	pro:I4 - C_{16}	pro:I5 - C_6	pro:I6 - C_2	pro:I7,I9 - C_5	pro:I8 - C_7	pro:I10 - C_{10}	pro:I11 - C_1	pro:I12 - C_3	pro:I13 - C_4	pro:I14 - C_8	pro:I15 - C_9	pro:I16 - C_{11}	pro:I17 - C_{15}	pro:I18 - C_{13}	pro:I19 - C_{18}
CO																		
PO																		
AO1																		
AO2																		
SO																		
SAO																		
CDB	×						×											⊗
SDB		×					×											⊗
GDB		⊗	⊗	⊗	×	×	⊗	⊗	⊗						⊗	⊗	⊗	⊗

5.2 *Usage of the lattice of component types*

While interfaces represent parts of collaborations, component types introduce consistent units dedicated to the provision of a consistent set of services. As in the previous lattices, but at a higher level in the structure of software artifacts, the lattice of component types offers both a specialization relation between component types and new abstract component types. This lattice has several applications in component assembly and software application re-engineering.

5.2.1 Emergence of new component types

The concepts in the lattice of component types can be interpreted as component types that we define as "canonical component types" to remain coherent with the previous definitions. Some of these canonical component types correspond to the original components: they are associated with concepts the reduced extent of which contains an original component. When the reduced extent of a concept is empty, we explore the intent of the concept to build the corresponding canonical component type. Thus, we consider symmetrically the required and provided interfaces from the intent. In the case of required interfaces, we consider those that have the smallest (more specific) type as shown in the interface lattice. In the case of provided interfaces, we consider those that have the largest (more general) type. These rules are a transcription of the substitution rules for functionality signatures, extended to interfaces.

Definition 5.3 Canonical component type corresponding to a component type concept**:** Let C be a concept in the component type lattice $\mathcal{L}_C$. The canonical component type $T_c(C)$ is defined as follows:

- If $ReducedE(C) \supseteq \{T\}$, then $T_c(C) = T$. We can choose any component type from the reduced extent because they are all equivalent.
- If $ReducedE(C) = \varnothing$, then $T_c(C) = \{pro : I, I \in max_{\leq_{\mathcal{L}_I}}\{J|pro : J \in Intent(C)\}\} \cup \{req : I, I \in min_{\leq_{\mathcal{L}_I}}\{J|req : J \in Intent(C)\}\}$.

In the case where an original component type appears in the reduced extent, the proposed construction finds an identical canonical component type. For example, concept C_{15} of lattice $\mathcal{L}_C$ has {`pro:I5, pro:I6`} as its canonical component type

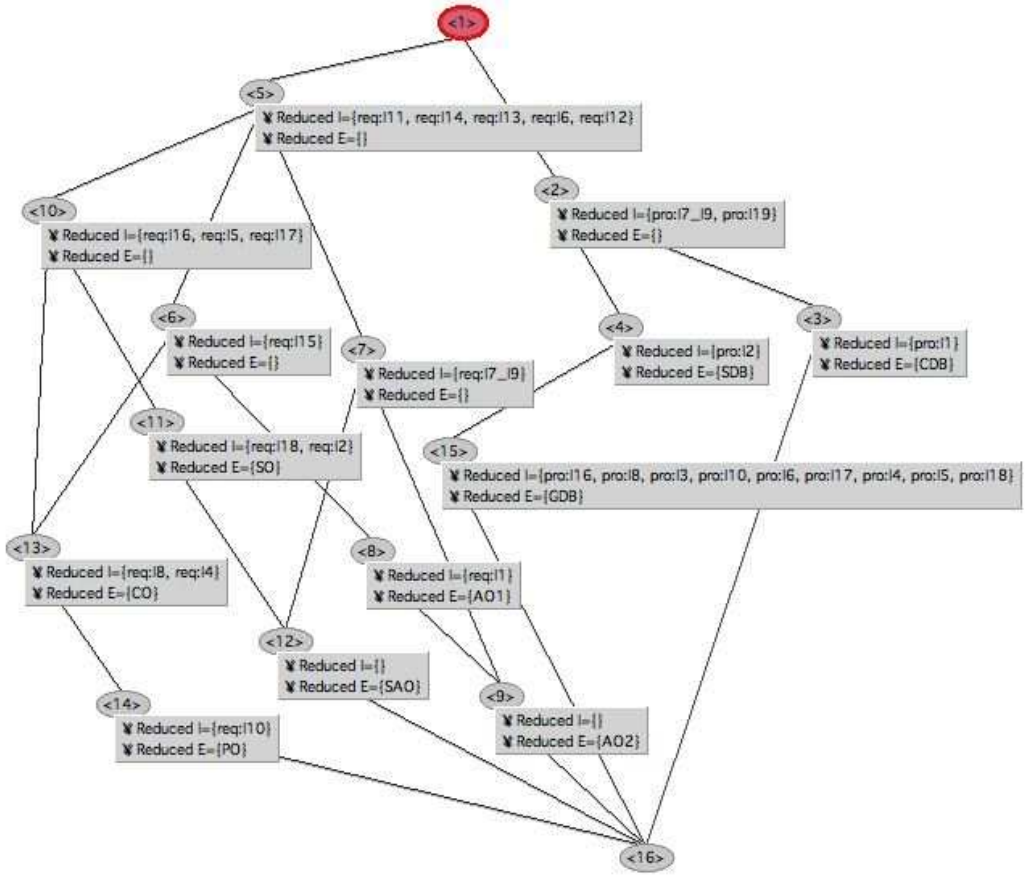


Figure 9. Lattice $\mathcal{L}_C$ of component types using the interface lattice.

because `I5` and `I6` are the maximum of $Intent(C_{15})$ (we do not make a distinction between required and provided interfaces because there are only provided interfaces in this intent). The reader will also notice that {`pro:I5, pro:I6`} is exactly the component type `GDB` that is found in the reduced extent of C_{15}.

5.2.2 Substitution and connection

The specialization relation we have built between concepts is tailored for substitution. Component substitution can be necessary in the event an entirely connected component fails. For example, let us suppose that an assembly is formed by component `CO` of type {`req:I8,req:I4`} entirely connected to component `GDB` of type {`pro:I5,pro:I6`}. Firstly, we can convince ourselves about the syntactical validity of the assembly that is ensured by two properties: required `I8` specializes provided `I6` and required `I4` specializes provided `I5` (as we generalize to interfaces the property described on Figure 5). Let us now imagine that component `CO` fails. Specialization in the lattice enables us to efficiently find a potential replacement. Component `PO` of type {`req:I10, req:I4`} will be a good candidate. The assembly remains valid because required `I10` specializes provided `I6`. The user will have access to a partial service because it is now only possible, among child books (`ChildBook` type), to ask for educational books (`EducationalBook` type), but the service may also perform better because it specializes about educational books.

Let us now analyze the connection problem. We note that two complementary components are not necessarily related to each other in the lattice: for example, there is no link between the components `AO2` of type {`req:I1,req:I7`} (concept C_9) and `CDB` of complementary type {`pro:I1,pro:I7`} (concept C_3). Indeed `req:I`

and `pro:I` are considered independent attributes. Given a component (*e.g.,* `AO2` of type {`req:I1,req:I7`}), it is nonetheless possible to find components that it can be connected to. A solution firstly consists in classifying the type of its complementary component (*e.g.,* {`pro:I1,pro:I7`}) applying the inferences. In our example, we obtain {`pro:I1,pro:I7,pro:I19`}. In this case, the classification enables us to reach concept C_3. C_3 and all smaller (more specific) concepts define, by the mean of their corresponding canonical component type, the types of components that can entirely connect to `AO2`.

5.2.3 Reingeneering and building generic architectures

We have previously described how the lattice discovers new component types. For example, concept C_5 of canonical type {`req:I14,req:I6`} has an empty extent. It indicates that the concept does not precisely correspond to an original component. However, it is an abstraction of all component types corresponding to lower (more specific) concepts. This canonical type, {`req:I14,req:I6`}, abstracts components relative to product orders in the example. It can be replaced by any of the more specific components. If a component of this canonical type participates in a component architecture, this architecture will have the capability of being instantiated using an important variety of concrete components. The discovery of such new abstract components into the classification can be interpreted as reengineering the set of existing components, and can help the developer design more generic architectures.

5.2.4 Architecture abstraction

The component lattice shows both specialization relationships among component types and newly discovered abstract component types. This can serve as the basis of whole architecture classification. This new objective is a little less direct to reach than the other generalization steps we have described in the paper because, in an architecture, components are not only described by binary attributes but also by their interconnections. Several ideas can be explored to take into account these connections such as Relational Concept Analysis (Huchard *et al.* 2007) or relations in Logical Information Systems (Ferré *et al.* 2005).

6. Related work

Few of the related approaches use a syntactical type hierarchy to structure component indexes and help component search. Zaremski and Wing (1995) suggest such a mechanism but in the more general context of functionality signature matching. The functionality hierarchy lies on the partial order relationship defined by the signature matching operator used, whether it be exact or relaxed. Module matching (component matching) is deduced from functionality matching: a component is comparable with another if each of its functionalities match a functionality of the other.

Existing yellow page-based service directories, also called service traders (Iribarne *et al.* 2004), such as CORBA Trading Object Service (OMG 2000), conform to the principles of the ODP standard (Information Technology Open Distributed Processing 1998). A component exports an advertisement into the component directory in order to be registered as the provider of some service. The service advertisement conforms to an existing service type that lists the properties and syntactical interfaces the components must have to provide the service. Service types can be ordered in a specialization hierarchy which is static and manually built. As opposed to our approach, these models use statically defined service

hierarchies (Marvie *et al.* 2001). This kind of indexing and the corresponding directories are not adapted to dynamical, evolving and open environments.

Works based on FCA propose to semi-automatically index components (Lindig 1995) in order to be able to help the developer identify adequate components from all the components stored in a component repository. Component search lies on groups of names and keywords and on incremental queries that help focus the search, diminishing the number of potential results as the search gets more precise. Fischer (1998), Sigonneau and Ridoux (2004) both aim at building such browsable functionality directories. Concepts are used to handle the iterative selection of attributes that define the user request as a traversal of the concept hierarchy. Thus, in these approaches, concept hierarchies do not directly reflect specialization relations between the syntactical types of functionality signatures. Fischer (1998) uses attributes which represent fragments of the formal specifications of functionalities (elementary pre- and post- conditions). Sigonneau and Ridoux (2004) use syntactical types of input and output parameters, along with covariant and contravariant specialization rules. In the context of web service search, machine learning techniques are used for service classification and annotation (Bruno *et al.* 2005, Corella and Castells 2006). Starting from textual documentation, services are automatically clustered using support vector machines or ontologies. FCA is then used in a second step to drive the matching between textual information and searched services.

As compared to these proposals, the originality of our work is to study directories of components described by sets of required and provided interfaces. Different specialization relations are defined to take into account not only parameter but also functionality directions. Moreover, we propose an iterative process to build lattices of component types which are composed of interfaces of both directions, which are in turn composed of functionalities. This iterative nature strongly differs from other works that use FCA which only build lattices of functionality types.

7. Conclusion and future work directions

In this article, we proposed to build component directories using FCA. The directory relies on the last built lattice that organizes components in order to speed up their retrieval, for either assembly or substitution. This component lattice is built upon some related lattices: an interface lattice which itself uses a classification provided by a functionality signature lattice. Beyond its usefulness for component assembly or component substitution, this classification also discovers new abstractions (new functionality signatures, new interface types and new component types), providing developers with valuable information about highly reusable elements. The developer can use this information as a guide along the development process or as re-engineering information.

The work presented in this article raises new research issues. Firstly, we want to study how our system can be implemented and integrated into an IDE to assist the management of component-oriented applications. This task comprises four steps:

- Extracting information about the component interfaces. We want to use the introspection capabilities of components to extract and dynamically maintain information on interfaces as the components enter or leave the system.
- Encoding the information in formal contexts, taking into account the identified inference rules and the type hierarchy of the system.

- Building lattices. Kuznetsov and Obiedkov (2002) present several incremental algorithms that enable new concepts to be added to an existing lattice. Several of these algorithms are implemented in GaLicia (Valtchev *et al.* 2003). These algorithms could be used to calculate the different lattices and also maintain them dynamically as components enter or leave the system.
- Using lattices. The obtained lattices can be used not only as a component index to ease search, but also as a way of visualizing the content of component libraries using the graphical interface of GaLicia or a similar FCA tool like TOSCANA (Vogt and Wille 1994) or CONEXP (Yevtushenko 2000).

This will enable us to systematically experiment with our approach on large component repositories, considering various component or interface granularity and function signature complexity.

We also plan to study complementary features of components, interfaces and signatures, such as ports, protocols or exceptions. For instance, ports (Desnos *et al.* 2006, 2007) would enable specifications of the dynamic behavior of components to be considered, providing more accurate component indexing and retrieval.

Another extension is inspired by Web Services directories (Klusch 2008). Contrary to component directories, they mainly use semantic information (names, descriptions) in their search mechanism. We can experiment with these techniques to refine classification considering the name of the parameters in the functionality signatures. Conversely, it is interesting to analyze how our approach could be used to improve the calculation of syntactical compatibility in Web Services.

Acknowledgments

Authors which to thank Nour Alhouda Aboud for her careful reading of a draft version of this paper. They also want to thank Peter W. Eklund, the guest editors and the anonymous reviewers of the paper for their helpful comments.

References

Birkhoff, G., 1940. *Lattice theory.* AMS Colloquium Publication, vol. XXV.

Bruno, M., Canfora, G., Penta, M.D. and Scognamiglio, R., 2005. An Approach to support Web Service Classification and Annotation. *In*: W.K. Cheung and J. Hsu, eds. *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)* Washington, DC, USA: IEEE Computer Society, 138–143.

Cardelli, L., 1984. A Semantics of Multiple Inheritance. *In*: G. Kahn, D.B. MacQueen and G.D. Plotkin, eds. *Semantics of Data Types*, LNCS 173 Springer, 51–67.

Corella, M.Á. and Castells, P., 2006. Semi-automatic Semantic-Based Web Service Classification. *In*: J. Eder and S. Dustdar, eds. *Business Process Management Workshops*, LNCS 4103 Springer, 459–470.

Davey, B.A. and Priestley, H.A., 1991. *Introduction to lattices and orders.* second Cambridge University Press.

Desnos, N., Huchard, M., Tremblay, G., Urtado, C. and Vauttier, S., 2008. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution: Research and Practice. Special Issue on Search-Based Software Engineering*, 20 (5), 321–344.

Desnos, N., Huchard, M., Urtado, C., Vauttier, S. and Tremblay, G., 2007.

Automated and unanticipated flexible component substitution. LNCS 4608 Springer, 33–48.

Desnos, N., Vauttier, S., Urtado, C. and Huchard, M., 2006. Automating the Building of Software Component Architectures. LNCS 4333 Springer, 228–235.

Ferré, S., Ridoux, O. and Sigonneau, B., 2005. Arbitrary Relations in Formal Concept Analysis and Logical Information Systems. *In*: F. Dau, M.L. Mugnier and G. Stumme, eds. *ICCS 2005*, LNCS 3596 Springer, 166–180.

Fischer, B., 1998. Specification-based Browsing of Software Component Libraries. *In*: D.F. Redmiles and B. Nuseibeh, eds. *Proceedings of the 13th IEEE international conference on Automated Software Engineering (ASE'98)*, October. Honolulu, Hawaii, USA: IEEE Computer Society, 74–83.

GaLicia, Galois lattice interactive constructor. : Université de Montréal. http://www.iro.umontreal.ca/∼galicia - accessed on Sept. 22, 2008 [2002].

Huchard, M., Hacene, M.R., Roume, C. and Valtchev, P., 2007. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49 (1-4), 39–76.

Information Technology Open Distributed Processing, ODP Trading Function Specification ISO/IEC 13235-1:1998(E). : International Organization for Standardization and International Telecommunication Union. http://webstore.iec.ch/preview/info_isoiec13235-1%7Bed1.0%7Den.pdf - accessed on Sept. 22, 2008 [1998].

Iribarne, L., Troya, J.M. and Vallecillo, A., 2004. A Trading Service for COTS Components. *The Computer Journal*, 47 (3), 342–357.

Klusch, M., 2008. Semantic service coordination. *In*: M. Schumacher, H. Helin and H. Schuldt, eds. *CASCOM - Intelligent Service Coordination in the Semantic Web*. Birkhaeuser Verlag, Springer, chap. 4.

Kuznetsov, S.O. and Obiedkov, S.A., 2002. Comparing performance of algorithms for generating concept lattices.. *Journal of Experimental & Theoretical Artificial Intelligence*, 14 (2-3), 189–216.

Lindig, C., 1995. Concept-Based Component Retrieval. *In*: J. Köhler *et al.*, eds. *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 21–25.

Marvie, R., Merle, P., Geib, J.M. and Leblanc, S., 2001. Type-Safe Trading Proxies Using TORBA. Fifth International Symposium on Autonomous Decentralized Systems, ISADS, IEEE Computer Society, 303–310.

Meyer, B., 1991. Design by contract. *In*: D. Mandrioli and B. Meyer, eds. *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1–50.

OMG, Trading Object Service Specification (TOSS) v1.0. : Object Managment Group. http://www.omg.org/cgi-bin/doc?formal/2000-06-27 - accessed on Sept. 22, 2008 [2000].

Sigonneau, B. and Ridoux, O., 2004. Indexation multiple et automatisée de composants logiciels orientés objet. *In*: J. Julliand, ed. *AFADL — Approches Formelles dans l'Assistance au Développement de Logiciels*, Juin. Besançon, France: RTSI, Lavoisier.

Szypersky, C., Gruntz, D. and Murer, S., 2002. *Component Software - Beyond Object-Oriented Programming (Second Edition)*. Second Addison-Wesley / ACM Press.

Valtchev, P., Grosser, D., Roume, C. and Hacene, M.R., 2003. GaLicia: An Open Platform for Lattices. *In*: A. de Moor, W. Lex and B. Ganter, eds. *Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03)*, Dresde (DE), July. Http://www.iro.umontreal.ca/∼galicia Shaker Verlag, 241–254.

Vogt, F. and Wille, R., 1994. TOSCANA - a Graphical Tool for Analyzing and

Exploring Data. *In*: R. Tamassia and I.G. Tollis, eds. *Graph Drawing*, LNCS 894 Springer, 226–233.

Wille, R., 1982. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, 83, 445–470.

Yevtushenko, S.A., ConExp, http://conexp.sourceforge.net/index.html. : SourceForge. [2000].

Zaremski, A.M. and Wing, J.M., 1995. Specification matching of software components. *In*: G.E. Kaiser, ed. *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, Washington, D.C., United States, October. New York, NY, USA: ACM Press, 6–17.

Zaremski, A.M. and Wing, J.M., 1997. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6 (4), 333–369.

Annexe E

Search-based many-to-one component substitution

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **00**:1–18 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Search-based many-to-one component substitution

Nicolas Desnos[1], Marianne Huchard[2], Guy Tremblay[3], Christelle Urtado[1,*] and Sylvain Vauttier[1]

[1] *LGI2P, Ecole des Mines d'Alès, Nîmes, France*
[2] *LIRMM, UMR 5506, CNRS and Univ. Montpellier 2, Montpellier, France*
[3] *Département d'informatique, UQAM, Montréal, Que., Canada*

SUMMARY

In this paper, we present a search-based automatic many-to-one component substitution mechanism. When a component is removed from an assembly to overcome component obsolescence, failure or unavailability, most existing systems perform component-to-component (one-to-one) substitution. Thus, they only handle situations where a specific candidate component is available. As this is not the most frequent case, it would be more flexible to allow a single component to be replaced by a whole component assembly (many-to-one component substitution). We propose such an automatic substitution mechanism, which does not require the possible changes to be anticipated and which preserves the quality of the assembly. This mechanism requires components to be enhanced with *ports*, which provide synthetic information on components' assembling capabilities. Such port-enhanced components then constitute input data for a search-based mechanism that looks for possible assemblies using various heuristics to tame complexity.

KEY WORDS: component substitution, component assembly evolution, search-based building process, many-to-one component replacement, heuristics, dead components

Introduction

Nowadays, software systems have to meet the needs of long life, autonomous and ubiquitous applications and must therefore be flexible, dynamic, and adaptable like never before.

*Correspondence to: LGI2P, Ecole des Mines d'Alès, Parc scientifique Georges Besse, F30035 Nîmes cedex, France
†E-mail: Nicolas.Desnos@ema.fr, huchard@lirmm.fr, tremblay.guy@uqam.ca, Christelle.Urtado@ema.fr, Sylvain.Vauttier@ema.fr

Received 31 August 2007
Revised 28 February 2008
Accepted 17 June 2008

Component-based software engineering (CBSE) [1] is a good solution to optimize software reuse and dynamic evolution while guaranteeing the quality of the software. Typically, a component is seen as a black box which provides and requires services through its interfaces. An architecture is built to fulfill a set of functional objectives (its functional requirements) while enforcing a set of properties (its non-functional requirements) and is described as a static interconnection of software component classes. A component assembly is a runtime instantiation of an architecture composed of linked component instances.

In long life applications or evolving environments, component substitution is a necessary mechanism for software evolution: it is a response to such events as component obsolescence, failure or unavailability. Anticipating valid component substitutions while designing some software is not always possible as the various contexts in which that software may run are not known in advance. Repairing a component assembly after a component has been removed while still preserving its whole set of functionalities is thus difficult. When a component is removed from an assembly, most existing approaches perform component-to-component (one-to-one) substitution [2, 3, 4, 5]. However, these approaches rely on the fact that an appropriate component, candidate for substitution, is available. This situation can hardly happen because it is difficult to find a component that has the same capabilities as the removed one. When such a component does not exist, allowing a single component to be replaced by a whole component assembly would permit more flexibility.

In this article, we propose an automatic substitution mechanism such that the possible changes do not need to be anticipated. Our approach uses primitive and composite ports for ensuring that a component can be replaced by a group of components while preserving the quality of the whole assembly. Such many-to-one component replacements are allowed by a search-based building algorithm that combines backtracking and branch and bound techniques to examine candidate assemblies. This algorithm is optimized using various search strategies and heuristics.

The rest of this paper proceeds as follows. The first two sections set up the context of this work. First, we briefly recall the typical CBSE process, in order to define correctness and completeness of an architecture. Then, we analyze the needs and limits of state-of-the-art practices for dynamic architecture reconfiguration. The following sections introduce our contribution. First, we present how ports allow us to automatically build valid assemblies [6] and how the assembly building process can be seen as a search-based problem, more precisely as a CSP. We then show how our building algorithm can be used as part of a four step component substitution process, and discuss how the complexity of the algorithm can be tamed using various search strategies and heuristics. Next, we discuss our implementation as well as some experiments we performed. Finally, we conclude and discuss some possible future work.

Software Architecture Correctness and Completeness in CBSE

This section discusses the issues of correctness and completeness of software architectures that result from a component reuse-based development process.

Typical CBSE process. CBSE [7] makes it possible to build large systems by assembling reusable components. The life cycle of a component-based architecture can be divided into three phases: design-time, deployment-time and runtime.

At design-time, the system is analyzed, designed and the design validity is checked. An architecture is built to fulfill a set of functional objectives (its functional requirements) [8, 9]. Functional objectives are deduced from problem analysis and defined as a set of functionalities to be executed on selected components. Then, the structure of the architecture is described as a static interconnection of software component classes through their interfaces. This requires both selecting and connecting the software components to be reused[†]. This description, typically written in an architecture description language [10], expresses both the functional and non-functional capabilities of the architecture, as well as both the structural and behavioral dependencies between components. For simplicity's sake, this work only focuses on preserving functional requirements while the software evolves. Non-functional properties are also important but can be handled only after the functional constraints have been satisfied. Once the architecture is described, its validity is statically checked. Most systems verify the correctness of the architecture, while some also guarantee its completeness—both notions are described below. Once the validity of the architecture is checked, it can be deployed. Deployment requires instantiating the architecture, configuring its physical execution context and dispatching the components in this physical context. One of the results of deployment is a component assembly: a set of linked component instances that conforms to the architectural description. At runtime, the component assembly executes.

The evolution of this assembly is an important issue for the application to adapt to its environment in situations such as maintenance, evolution of the requirements, fault-tolerance, component unavailability, etc. In this context, an important question is: What are the possible dynamic evolutions that can be supported by the component assembly and by the architecture itself? The remaining of this paper is a tentative answer to this question.

Correctness. Verifying the correctness of an architecture amounts to verifying the connections between components and checking whether they correspond to a possible collaboration [9]. These verifications use various kinds of meta-information (types, protocols, assertions, etc.) associated with various structures (interfaces, contracts, ports, etc.). The most precise checks are done by protocol comparison, which is a hard combinatorial problem [11, 12, 13, 14, 15].

Completeness. An architecture must guarantee that all its functional objectives will be met. In other words, the connections of an architecture must be sufficient to allow the execution of collaborations that reach (include) all its functional objectives. We call this **completeness** of the architecture [6]. Indeed, the use of a component functionality (modeled by the connection

[†]We assume that the selected components need no adaptation (or have already been adapted) as it is the only situation in which the components' external definitions are sufficient to *match* (whatever the definition of matching is) with other components' needs. Complementary approaches, interested in automating the assembly building process and performing component adaptation, must necessarily rely on additional information (*e.g.*, domain specific semantics, data or usage patterns) either provided by designers or collected through a reingeneering process. Thus, our approach is lighter.

of an interface) can require the use of other functionalities which, in turn, entail new interface connections. Such functionalities (or interfaces) are said to be **dependent**. This information is captured in the description of component behavior and depends on the context in which the functionality is called (execution scenario). There are various ways to ensure completeness:

- For a first class of systems [16], completeness of an architecture is characterized by the fact that all interfaces of any of the architecture's component are connected. This notion of completeness is simple to check but over-constrains the assembly, thus diminishing both the capability of individual components to be reused in various contexts and the possibilities of building a complete architecture given a set of predefined components.
- A second class of systems [3] defines two categories of interfaces, namely, mandatory and optional. An architecture is then considered complete if all mandatory interfaces are connected, while optional ones can be left pending. This does not complicate completeness checking, yet increases the opportunities of building a complete architecture given a set of predefined components. However, associating the mandatory / optional property to an interface regardless of the assembly context does not increase the capability of individual components to be reused in various contexts.
- The third, more relaxed view of completeness, requires connecting only the interfaces that are strictly necessary [12, 17, 18] by exploiting the component behavior's description. This is typically done by analyzing protocols which makes completeness checking less immediate.

In order to build correct and complete component assemblies we consider having as precise correction checking as possible and adopt the third vision on completeness while trying to limit the costs of protocol comparison by dismissing the less useful information. To achieve this, we define a port-enhanced component model.

Example. Figure 1 illustrates that completeness of an assembly can be ensured while connecting only the strictly necessary interfaces. For simplicity's sake, in the example, compatible operations and interfaces have the same name whereas, in the general case, interface types only need to be substitutable. The *Dialogue* interface from the *Client* component represents a functional objective and must therefore be connected. As can be deduced by analyzing the execution scenario that has to be supported, all the dependent interfaces (grayed on Figure 1) must also be connected in order to reach completeness. For example, as shown in line 12 of the execution scenario, the *Control* interface from the *MemberBank* component must be connected whereas the *Question* interface from the *Client* component, which is not used in the scenario, does not need to be connected.

Dynamic Architecture Reconfiguration

This section discusses correctness and completeness issues for evolving software assemblies. To ensure that an evolving valid component assembly remains valid at runtime, all systems try to control how the assembly evolves. Different evolution policies exist:

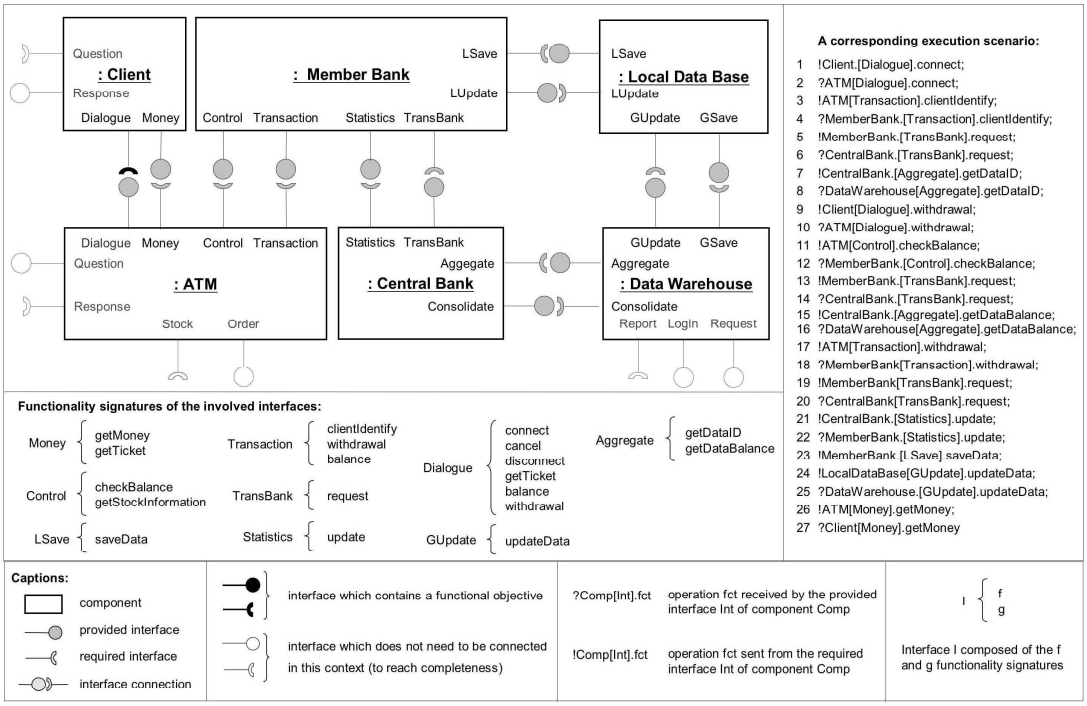


Figure 1. A complete assembly and a possible corresponding execution scenario

- Some systems simply forbid dynamic reconfigurations [19, 20], so assemblies cannot evolve at runtime. This, of course, is an unsatisfactory policy.
- Some systems [21, 3] allow the architectural structure to be violated when modifying component assemblies at runtime. They allow component and connection modifications (addition, suppression) based on local interface type comparisons. The result is a lack of control on the assembly: its validity is not guaranteed anymore.
- Other systems ensure that component assemblies always conform to the architectural structure. All possible evolutions must therefore be anticipated at design-time and described in the architecture itself [10]. Different techniques can be used. For example, [22, 5] use patterns to specify which interfaces can be connected or disconnected and which components can be added or removed. [23, 24] use logical rules, a more powerful means to describe the possible evolutions. These solutions, however, complicate the design process and make anticipation necessary, which is not always possible [5, 25].

Dynamic Component Removal. Among the situations that must be handled to enable component assembly evolution is dynamic component removal. Other facets of interest in the

change process are related to identifying changes, interrupting components' execution, saving the removed components' state, deploying the new components, and migrating the saved states to the new components[‡]. When removing a component from an assembly, the main constraint is to ensure that there will be no functional regression. The third category of systems mentioned above typically allow a removed component to be replaced by a component which provides compatible services in order for the assembly to still conform to the architecture. Anticipating possible evolutions allows these systems to ensure that the new assembly will still be valid, as it has been statically checked on the architecture at design-time.

There are two major interpretations of component compatibility. In most systems [26, 2, 5, 3], components must strictly be compatible: the new component must provide at least the provided interfaces of the removed component and cannot require more required interfaces. In [27], compatibility is less restrictive and context-dependent. If a provided interface from the removed component is not used by another component in the assembly (not used in this context), the new component is not required to provide this interface (as it is not necessary in this context). On the other hand, the new component can have additional required interfaces as soon as they find a compatible provided interface among the assembly's components. This context-dependent definition of compatibility allows better adaptability of assemblies.

Discussion. There are two main restrictions to the state of the art solutions to completing a component assembly after a component has been dynamically removed:

1. Anticipating all possible evolutions to include their description in the initial, design-time, architecture description is not always possible because it requires knowing all situations that may occur in the system's future evolution. Ideally, it would be better to manage the evolution of software assemblies in an unanticipated way.
2. Replacing the removed software component by a single component is not always possible because it is generally unlikely that a component having compatible interfaces exist among the potential candidates for substitution. Yet, in the (more frequent) case when an adequate component does not exist, it might be possible to replace the removed component by a set of linked components that, together, provide the required services.

The problem of replacing a removed component by an assembly of components in an unanticipated way while guaranteeing, as much as possible, the quality (executability) of the whole assembly is the initial motivation for the work presented in this paper.

[‡]Even though we have not yet studied the deployment process itself, our system could help identify which components might be impacted when removing some components, thus minimizing the number of components that need to be interrupted. Moreover, as far as state consistency is concerned, we assume, as in all CBSE approaches, that no assumption can be made on the components' implementation. This assumption thus makes state migration impossible, as it would constrain the internal structure of components. If some change occurs, a robust implementation of our sytem would rollback the states of all components so that they all are in some initial state that enables them to be restarted safely.

Port Enhanced Components for Incremental Assembly Completeness Checking

This section describes how adding ports to components provides a means to automatically build complete component assemblies [6, 28]. Existing approaches usually statically describe architectures in a top-down manner. Once the architecture is defined, they verify its validity using costly checking algorithms [11, 12, 13, 14, 15]. Our building of assemblies from components obeys an iterative (bottom-up) process. This makes the combinatorial cost of these algorithms critical and prevents us from using them repeatedly, as a naive approach would require. To reduce the complexity, we chose to simplify the information contained in protocols—more precisely, behavior protocols such as those used in SOFA [29], which are regular expressions that express the various possible sequences of events (traces) allowed by a component—and represent this information in a more *abstract* and usable manner through primitive and composite ports. Ports allow us to build a set of interesting complete assemblies from which it is possible to choose and check the ones that are best adapted to the architect's needs.

Primitive and Composite Ports. The underlying idea for building a complete component assembly is to start from the functional objectives, select the suitable components, and then establish the necessary links between them. Completeness is a global property that we will guarantee locally, in an incremental way, all along the building process. The local issue is to determine which interfaces to connect and where (to which component) to connect them. This information is hidden into behavior protocols where it is difficult to exploit in an incremental assembly process. We thus enhance the component model with the notion of port, to model the information that is strictly necessary to guarantee completeness in an abstract way. Primitive and composite ports will represent two kinds of connection constraints on interfaces, so that the necessary connections can be correctly determined. In some way, ports express the different usage contexts of a component, making it possible to connect only the interfaces which are useful for completeness. Primitive ports are used to model simple usage contexts (possible collaboration between two components) and composed into composite ports that model more complex contexts (complex collaborations). As in UML 2.0 [30], one can also consider that the functional objectives of an architecture are represented by use cases, that collaborations concretely realize use cases and contain several entities that each play a precise role in the collaboration. Primitive and composite ports can be considered as the part of a component that enables the component to play a precise role with respect to a given use case.

Primitive ports are composed of interfaces, as in many other component models [30, 31]. Ports are introduced as a kind of structural meta-information, complementary to interfaces, that group together the interfaces of a component corresponding to a given usage context. More precisely, a primitive port can be considered as the expression of a constraint to connect a set of interfaces both at the same time and to a unique component.

In Figure 2, the *Money_Dialogue* primitive port gathers the *Dialogue* and the *Money* interfaces from the *Client* component. It expresses a particular usage context for this component: here, a collaboration the *Client* component can establish with another (yet unknown) component to withdraw money. Connecting two primitive ports is an atomic operation that connects their interfaces: two primitive ports are connected together when all

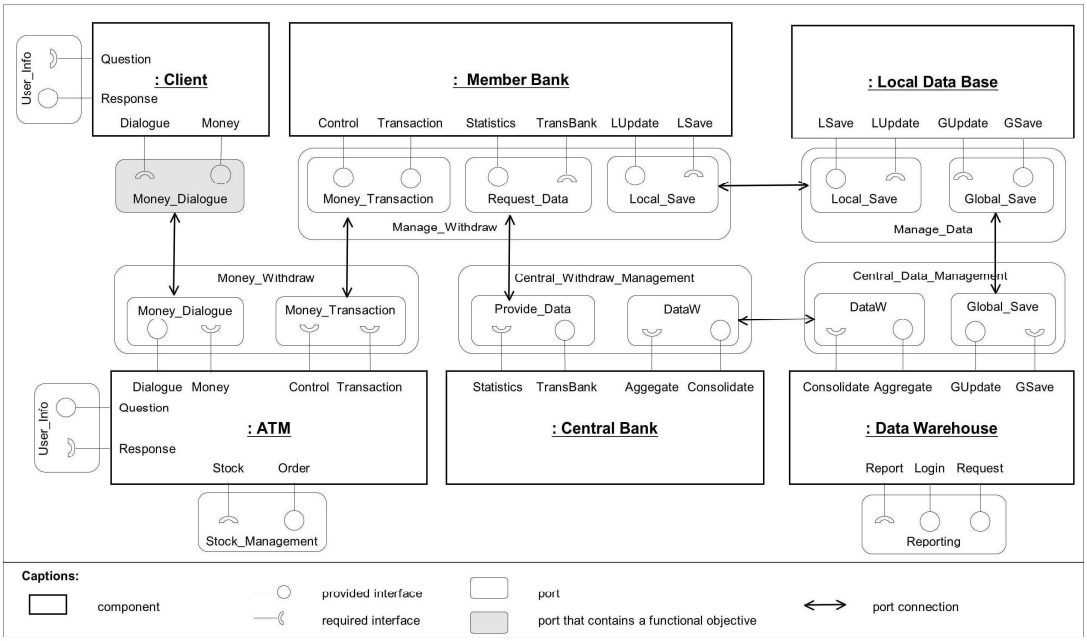


Figure 2. Example of components with primitive and composite ports

the interfaces of the first port are connected to interfaces of the second port (and reciprocally). Thus, port connections make the building process more abstract (port-to-port connections) and more efficient (no useless connections). In this example, the *Money_Dialogue* primitive port from the *Client* component is connected to the *Money_Dialogue* primitive port from the *ATM* component.

Composite ports are composed of other ports. They model complex collaborations that are composed of finer grained ones (modeled by the sub-ports). Indeed, they provide an abstract description of a part of the component behavior—less information than in component behavior protocols but more information than the syntactic description of component capabiliies modelled by interfaces. A composite port expresses a constraint to connect a set of interfaces at the same time but possibly to different components. In Figure 2, the *ATM* component has a composite port named *Money_Withdraw* which is composed of the *Money_Dialogue* and *Money_Transaction* primitive ports.

Much like a designer must do with protocols, ports have to be manually added to document the design of components; however, we are currently working on their automatic generation from behavior protocols.

Completeness of an Assembly as Local Coherence of its Components. Calculating the completeness of an already built component assembly is of no interest in an incremental building approach. Our idea is to better consider a local property of components that, if true, guarantees that the component is adequately connected to its immediate neighbors, and then to aggregate these local values into a global completeness property. We call this local property **coherence** and have shown [6] that it is a necessary condition for validity. Intuitively, we will see that when all components of an assembly are coherent, the assembly is complete. A component is said to be coherent if all its exposed (top-level) composite ports are and these latter are coherent if their primitive ports are connected in a coherent way (see below).

To determine the completeness of an assembly, we need to know if the interfaces that must be connected are indeed connected. The main idea is to check the coherence of each composite port. Two cases must be checked: when the composite port does not share any primitive port with another unrelated composite port and when it does share some primitive ports.

An exposed composite port is said to be coherent if one of these three mutually exclusive cases holds:

1. All its primitive ports are connected.
2. None of its primitive ports is connected.
3. Some, but not all, of its primitive ports are connected. In this case, the composite port can still be coherent if it shares the connected primitive ports with another unrelated composite port (of the same component) which is itself entirely connected. Indeed, sharing of primitive ports represents alternative connection possibilities [6]. A partially connected composite port can represent a role which is useless for the assembly as long as its shared primitive ports are connected in the context of another (significant) composite port.

A component is said to be coherent if all its exposed composite ports are coherent. An assembly of components is said to be complete if *i*) all the primitive ports which represent functional objectives are connected; *ii*) all its components are coherent.

In the next section, we provide more formal definitions in order to show that the building of all complete component assemblies can be seen as a search-based problem.

Building Complete Component Assemblies: a Search-Based Problem

Formal Definition of Completeness. More formally, completeness can be described after setting some preliminary definitions.

- We define a **component** C as a quintuple:

$$C = (Prv_C, Req_C, Prim_C, Comp_C, TopComp_C)$$

Prv_C is the set of C's provided interfaces and Req_C its set of required interfaces.
$Prim_C$ is the set of all C's primitive ports, $Comp_C$ its whole set of composite ports and $TopComp_C \subseteq Comp_C$ its set of exposed (top-level) composite ports.

- We denote by $Int_C = Prv_C \cup Req_C$ the whole set of C's interfaces and by $Ports_C = Prim_C \cup Comp_C$ the whole set of C's ports.
- An **interface** is characterized by a set of operation signatures (its interface type) and a direction (provided or required). We assume, as in most object-oriented languages (*e.g.*, Java) and modeling languages (*e.g.*, UML), that interface types are partially ordered in a specialization hierarchy. If not, or if a finer definition is required, it is always possible to (re)define such a specialization relation as we have done in [32].
- A **primitive port** ρ is a set of interfaces. Let $\rho \in Prim_C$ be a primitive port of C, $\rho \subseteq Int_C$.
- A **composite port** γ of C is a set of ports, primitive or composite, from C, subject to some restrictions described below.
- Let $\gamma \in Comp_C$ be a composite port of C, where $\gamma \in 2^{Ports_C}$. We define $PrimPorts^*(\gamma)$, resp. $CompPorts^*(\gamma)$, as the set of all primitive, resp. composite, ports that are directly or indirectly contained in γ:

$$\begin{aligned} PrimPorts^*(\gamma) &= \{\rho \in \gamma \cap Prim_C\} \cup \bigcup_{\gamma' \in \gamma \cap Comp_C} PrimPorts^*(\gamma') \\ CompPorts^*(\gamma) &= \{\gamma' \in \gamma \cap Comp_C\} \cup \bigcup_{\gamma' \in \gamma \cap Comp_C} CompPorts^*(\gamma') \end{aligned}$$

  Note that for γ to be well-defined, γ cannot be a (direct or indirect) sub-port of itself, that is, $\gamma \notin CompPorts^*(\gamma)$.
  For component C to be well-defined, each of its composite ports must either itself be or be a sub-port of an exposed composite port of C. This can be expressed as:

$$\forall \gamma \in Comp_C \cdot \gamma \in TopComp_C \ \vee \ \exists \gamma' \in TopComp_C \cdot \gamma \in CompPorts^*(\gamma')$$

- Let i be an interface. We denote by $Dir(i) \in \{pro, req\}$ the direction of interface i and by $Type(i)$ its type. We denote by $\preceq$ the specialization relation between interface types. An interface i is said to be **compatible** with an interface i' iff the provided interface type is equal to or more specific than the required interface type:

$$Compat(i, i') = \oplus \begin{cases} Dir(i) = pro \wedge Dir(i') = req \wedge Type(i) \preceq Type(i') \\ Dir(i) = req \wedge Dir(i') = pro \wedge Type(i') \preceq Type(i) \end{cases}$$

- A primitive port ρ is said to be **compatible** with another primitive port ρ', noted $(\rho, \rho') \in \mathcal{R}_{comp}$, iff there is a bijection from one's set of interfaces to the other's set of interfaces such that corresponding interfaces are compatible. Primitive port compatibility is symmetric.

$$(\rho, \rho') \in \mathcal{R}_{comp} = \exists f : \rho \to \rho' \cdot \forall\, i' \in \rho' \cdot \exists!\, i \in \rho \cdot f(i) = i' \wedge Compat(i, i')$$

Let us now consider a component assembly that involves a set of components and a set of primitive port connections.

- We denote by $\widehat{\rho}$ the fact that, with respect to a set of components, ρ is *connected*—i.e., any required (resp. provided) interface of ρ is correctly linked with a provided (resp. required) interface of another (primitive) port.

- We denote by $\widehat{\gamma}$ when all primitive ports contained in γ are connected[§]:

$$\widehat{\gamma} = \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \cdot \widehat{\rho}$$

- Let $\gamma \in \mathit{TopComp}_C$ be a top-level composite port of component C. $\mathit{Shared}_C(\gamma)$ is the set of primitive ports shared by γ and by any other top-level composite port of C:

$$\mathit{Shared}_C(\gamma) = \{\rho \in \mathit{PrimPorts}^*(\gamma) \mid \exists\, \gamma' \in \mathit{TopComp}_C \cdot \gamma \neq \gamma' \wedge \rho \in \mathit{PrimPorts}^*(\gamma')\}$$

Given an exposed composite port $\gamma \in \mathit{TopComp}_C$, three mutually exclusive cases are possible for γ to be coherent as argued in the previous section.

- $\gamma \in \mathit{TopComp}_C$ is **coherent** (with respect to component C) if the following holds:

$$\oplus \begin{cases} \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \cdot \widehat{\rho} & \text{(which is equivalent to } \widehat{\gamma}) \\ \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \cdot \neg\widehat{\rho} & \\ \wedge \begin{cases} \forall\, \rho \in \mathit{Shared}_C(\gamma) \cdot & \\ \quad \widehat{\rho} \Rightarrow \exists\, \gamma' \in \mathit{TopComp}_C \cdot \gamma \neq \gamma' \wedge \rho \in \mathit{PrimPorts}^*(\gamma') \wedge \widehat{\gamma'} \\ \forall\, \rho \in \mathit{PrimPorts}^*(\gamma) \setminus \mathit{Shared}_C(\gamma) \cdot \neg\widehat{\rho} \end{cases} \end{cases}$$

- A component C is **coherent** iff: $\forall\, \gamma \in \mathit{TopComp}_C \cdot \gamma$ is coherent

Building All Complete Assemblies as a Constraint Satisfaction Problem. The inputs of our problem are:

- A component repository. This repository is characterized by the set Π of all primitive ports from all the components in the repository, and by the set $\mathit{TopComp}$ of all the exposed (top-level) composite ports from all the components in the repository.
- Functional objectives. These functional objectives are defined through $\mathcal{O} \subseteq \Pi$, the set of primitive ports which match the functional objectives.

Let us now define $\mathit{Role}(\rho)$ as the set of all the exposed composite ports to which a primitive port ρ belongs.

$$\mathit{Role}(\rho) = \{\gamma \in \mathit{TopComp} \mid \rho \in \mathit{PrimPorts}^*(\gamma)\}$$

We also note $\mathit{Compatible}(\rho)$ the set of all primitive ports in Π that are compatible with a primitive port ρ.

Let $\mathit{Connections}_\rho = \{x_\gamma^\rho\}_{\rho \in \Pi, \gamma \in \mathit{Role}(\rho)}$ be the set of variables that represent the connections of a primitive port ρ.

Each variable represents the connection of a primitive port in the context of one of the exposed composite ports it belongs to. The connection of a shared primitive port is thus represented by several variables. Each variable enables to distinguish the different connection contexts, in which a shared primitive port is considered at the same time as connected, when

[§]As in VDM [33] and B [34], "$\cdot$" separates the (typed) variable introduced by the quantifier and the associated predicate.

it participates to the connection of a connected exposed composite port, or unconnected, when it belongs to another unconnected exposed composite port.

Let $Connections = \bigcup_{\rho \in \Pi} Connections_\rho$ be the set of variables that are used to describe all connections between all existing components. *Connections* is thus the set of variables of the CSP we have to solve. The value domains of these variables are:

$$\forall\, x_\gamma^\rho \in Connections \cdot Dom(x_\gamma^\rho) = Compatible(\rho) \cup \{nil\}$$

Given those value domains, each variable, which represents a given primitive port, can be assigned as value the primitive port to which it is connected—*nil* is a special value indicating that the primitive port is unconnected.

Building a component assembly then amounts to assigning values for the various variables of *Connections*, with respect to a set of constraints that guarantee the consistency of the architecture:

1. *Constraints on functional objectives.* All primitive ports selected as functional objectives must be connected in the solution.

$$\forall\, \rho \in \mathcal{O} \cdot \exists\, x_\gamma^\rho \cdot x_\gamma^\rho \neq nil$$

2. *Constraints on port connection symmetry.* When a primitive port is connected to another primitive port, then the latter primitive port must be connected to the former.

$$x_\gamma^\rho = \rho' \Rightarrow \exists\, x_{\gamma'}^{\rho'} \cdot x_{\gamma'}^{\rho'} = \rho$$

3. *Constraints on exposed composite port coherence.* The variables that correspond to connections of primitive ports of an exposed composite port must either all be set to *nil* or all be set to some *non-nil* value.

$$\forall\, \gamma \in TopComp \cdot \forall\, \rho, \rho' \in PrimPorts^*(\gamma) \cdot x_\gamma^\rho \neq nil \Rightarrow x_\gamma^{\rho'} \neq nil \;\oplus\; x_\gamma^\rho = nil \Rightarrow x_\gamma^{\rho'} = nil$$

4. *Constraints on shared primitive port connection well-formedness.* When a shared primitive port belongs to several connected exposed composite ports, it must be connected to the same primitive port in every context.

$$\forall\, \gamma, \gamma' \in TopComp \cdot \forall\, \rho \in Shared_C(\gamma) \cap Shared_C(\gamma') \cdot x_\gamma^\rho \neq nil \;\wedge\; x_{\gamma'}^\rho \neq nil \Rightarrow x_\gamma^\rho = x_{\gamma'}^\rho$$

When there is no functional objective, a trivial solution that satisfies all the constraints is an assembly with no connection (every variable in *Connections* is *nil*). Every defined functional objective adds a constraint that excludes *nil* from the domain of the associated variable: the corresponding port must be connected. A non-trivial solution must then be found thanks to a combination of different problem solving techniques. We propose a backtracking algorithm that enumerates the possible variable assignment combinations, optimized with strategies that prune the search tree and heuristics that speed up its traversal—thus effectively resulting in a branch-and-bound strategy. These search techniques are combined with constraint propagation (arc consistency) that filter inconsistent values from variable domains to reduce the search space. This algorithm, its optimizations and its results are presented next.

Overview of the Incremental Building Process. The principle of our automatic building process is first to connect all the primitive ports representing functional objectives and then to iteratively list and connect all the primitive ports that must be connected to maintain the coherence of the components' exposed composite ports. This process is implemented as a depth-first traversal of a construction tree. Backtracking allows a complete exploration of every construction paths (alternative connection choices), thus ensuring that all possible solutions are found.

The building algorithm uses a set (FO-set) that always contains a list of the ports that still have to be connected. This FO-set contains only primitive ports: when a composite port (γ) has to be connected, it is decomposed into the set of primitive ports it contains, directly or indirectly ($PrimPorts^*(\gamma)$), and these primitive ports are added to the FO-set. The FO-set is initialized with the primitive ports that correspond to the functional objectives. The building process can be decomposed into three steps:

1. *Choice of the primitive port.* One of the primitive ports is selected from the FO-set.
2. *Choice of a compatible unconnected primitive port and connection.* Compatible primitive ports are searched for amongst the ports of components from the repository or from the already built sub-assembly. If compatible unconnected ports are found, one of them is selected. If the chosen port belongs to a component that does not yet belong to the assembly, the component is added to the assembly. The two ports are then connected together.
3. *Choice of a collaboration context and update of the dependency set.* If the chosen compatible port belongs to a single exposed composite port, all other primitive ports of that composite port are added to the FO-set. If the chosen compatible port is shared by several exposed composite ports, one of those exposed composite ports (defining one of the possible collaboration contexts) is chosen as a collaboration context and its primitive ports are added to the FO-set. The other exposed composite ports may in turn be chosen when the building process backtracks to explore another solution. In any case, no port dependencies—and therefore no interface dependencies—are left unsatisfied.

These three steps are iterated until the FO-set is empty. All the initial primitive ports that represent functional objectives are then connected along with all ports they are recursively dependent upon: the resulting assembly is thus complete. During the whole process, backtracking allows to both rollback unsuccessful connection attempts—past connection choices lead to a situation where there is no available primitive port where to connect a primitive port from the FO-set—and build all possible complete assemblies. This enumerative building process, of which the basic principle is presented here, is highly combinatorial. Optimization strategies and heuristics have been used to speed up the traversal of the construction space as presented below.

As a result, the building algorithm provides a set of complete architectures. Since architecture completeness is a necessary condition for architecture validity, the resulting set of complete architectures provides preselected assemblies on which classical correctness checkers, such as [5], can then be used.

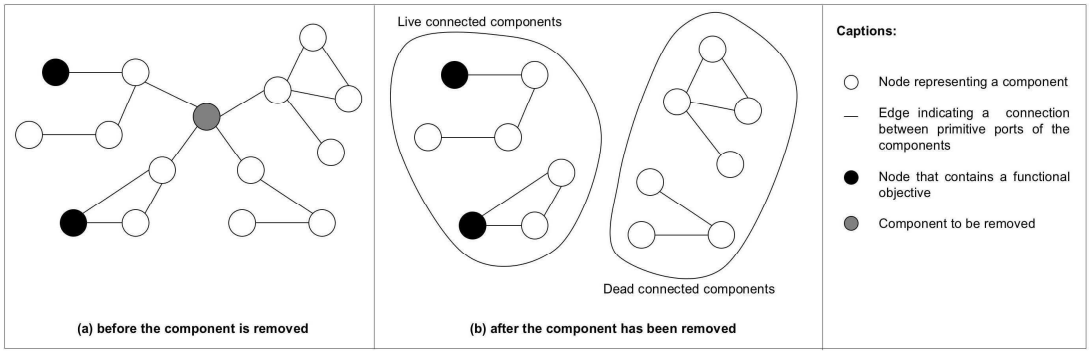


Figure 3. An assembly can be seen as (a) an abstract graph which is divided (b) in two sets of connected components when a component has been removed

Many-to-one Component Substitution Using the Automatic Building Process

To react to the dynamic removal of a software component, we propose a two step process that allows a flexible replacement of the missing component:

1. Analyze the assembly from which the component has been removed and remove the now useless (dead) components;
2. Consider the incomplete component assembly as an intermediate result of our iterative building algorithm and therefore run the building algorithm on this incomplete assembly to re-build a complete assembly.

Removing Dead Components. When a component has been removed from a complete assembly, some parts of the assembly may become useless. Indeed, some of the components and connections in the original assembly might have been there to fulfill needs of the removed component. To determine which parts of the assembly have become useless, let us define a graph which provides an abstract view of the assembly.

An assembly can be represented as a graph where each node represents a component and each edge represents a connection between two (primitive) ports of two of its components. We also distinguish two kinds of components: those which fulfill a functional objective—i.e., the components which contain a port which contains an interface which contains a functional objective—and those which do not (cf. Figure 3).

An **assembly** A can then be seen as a graph along with a set of functional objectives:

$$A = (G_A, FO_A)$$

Here, $G_A = (Cmps_A, Conns_A)$ is a graph, with $Cmps_A$ the set of nodes—each node being a component—, $Conns_A$ the set of edges—each edge indicating the existence of some primitive

port connection between the components—, and $FO_A \subseteq \bigcup_{C \in Cmps_A} Prim_C$ the set of primitive ports that contain some functional objectives[¶].

If we consider the graph that results from the removal of the node representing the removed component, we can partition the graph in two parts: the connected components[‖] that have at least a node which contains a functional objective and the connected components without any node that contains a functional objective. The second part of the graph is no longer useful because the associated components were not in the assembly to fulfill some functional objectives but rather to fulfill some of the removed component's needs. Removing this part of the graph amounts to removing now useless parts of the assembly before trying to re-build the missing part with new components and connections.

Let $A = (G_A, FO_A)$ be an assembly and let $C \in Cmps_A$ be the component to remove. We define $H_{A,C}$ as the graph G_A from which we removed component C and all the edges (denoted by $Conns_C$) corresponding to primitive port connections between C and another component of G_A:

$$H_{A,C} = (Cmps_A \setminus \{C\}, Conns_A \setminus Conns_C)$$

We define $\mathcal{L}_{A,C}$ the live connected components of $H_{A,C}$ as the graph composed of all the connected components of $H_{A,C}$ that have at least a node which contains a functional objective.

We also define $\mathcal{D}_{A,C}$ the dead connected components of $H_{A,C}$ as the graph composed of all the connected components of $H_{A,C}$ that have no node which contains a functional objective.

Let us just notice that:

$$H_{A,C} = \mathcal{L}_{A,C} \cup \mathcal{D}_{A,C}$$

Figure 3 illustrates the definitions of $\mathcal{L}_{A,C}$ and $\mathcal{D}_{A,C}$. When a component is removed from the assembly, all components which do not participate anymore in the assembly's completeness can be removed. Components from the dead connected components set $\mathcal{D}_{A,C}$ can be removed from the assembly because they only participated in the removed component's coherence. Indeed, as dependencies are modelled by edges of the graph, if there are unconnected subgraphs that are not needed to implement the functional objectives (which we call subgraphs of dead components), these subgraphs are useless (no dependency links them to the parts of the graphs that contain functional objectives).

Removing the dead components is a necessary step because keeping useless components add useless dependencies that make the resulting assembly considerably larger, thus complicating the building process, making the validity checks more difficult and making the assembly more subject to failures, less open for extensions, etc. Let us just also note that the components in $\mathcal{D}_{A,C}$ are dead components but that there still might be useless components in $\mathcal{L}_{A,C}$ (those we keep). We are considering future improvements that would exploit the protocols to improve the detection of dead components.

[¶]Recall that a functional objective is simply an operation defined in one of the provided interfaces.
[‖]In this subsection of the paper, a *connected component* refers to a subgraph that is connected, meaning that there exists a path between any of its two nodes.

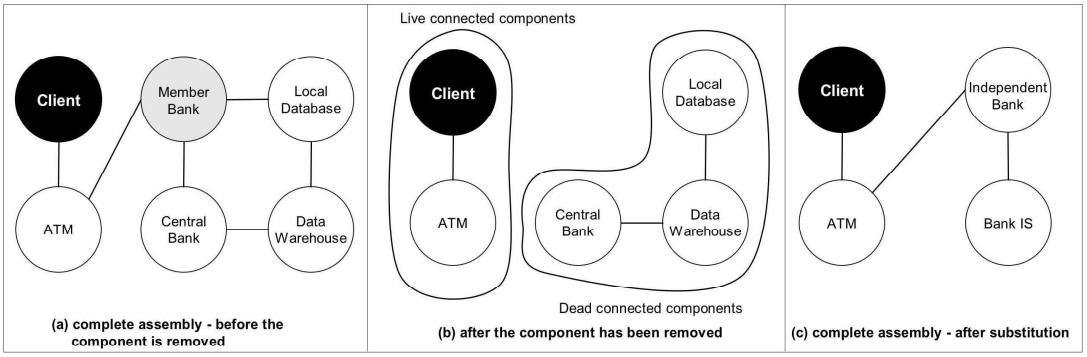


Figure 4. Evolution scenario on the ATM example: removal of the *MemberBank* component

Re-building the Incomplete Assembly. Once the dead components have been removed from the component assembly, the assembly contains all the components necessary to ensure completeness except for one component (the removed one) along with its dependent components. Some of the remaining components' dependencies are not yet satisfied. The goal is then to find a single component (like other systems do) or a series of assembled components that can fulfill the same unsatisfied dependencies as the removed component did. We suppose that it is quite unlikely that there exists a component that exactly matches the role the removed component had in the assembly. It is more likely (more flexible) to have the possibility of replacing the removed component by a set of assembled components that, together, can replace the removed component.

The partial assembly in $\mathcal{L}_{A,C}$ is the re-building process starting point. It is considered an intermediate result of the global building process described above. The partial assembly is not complete yet: there still exist unsatisfied dependencies that were previously fulfilled by the removed component. These dependencies are identified, and the building process we described above is run to complete the architecture. In this case, the initial FO-set contains the primitive ports that correspond to unsatisfied dependencies, to which is added, if applicable, the removed component's primitive ports that were part of the assembly's initial functional objectives.

Evolution Scenario. For our *ATM* example, Figure 4(a) represents the graph corresponding to the example of Figure 2. The *Client* node represents the *Client* component which contains a functional objective. The other nodes (*MemberBank*, *ATM*, *CentralBank*, *LocalDatabase* and *DataWarehouse*) represent components which do not contain any functional objective. We also assume there is a component repository, which will be searched for possible replacement components. Figure 4(b) shows that the partial component assembly from $\mathcal{L}_{ATMexample,MemberBank}$ is not complete because the *ATM* component has become incoherent after the *MemberBank* component and the three now consequently dead components ($\mathcal{D}_{ATMexample,MemberBank} = \{CentralBank, LocalDatabase, DataWarehouse\}$)

have been removed. To complete the assembly, new components must be added. Figure 4(c) sketches the result of this re-building process: The *IndependentBank* component is connected to the *BankIS* component and they both replace the components that had been removed to complete the *ATM* example assembly.

Figure 5 details the resulting architecture. In this example, *MemberBank* is the component to remove. When it is removed, completeness of the architecture is lost. Indeed, the *ATM* component is not locally coherent anymore. Its *Money_Withdraw* composite port is not coherent because the primitive port *Money_Transaction* is not connected while the *Money_Dialogue* primitive port is not shared and still connected. The *CentralBank*, *LocalDatabase* and *DataWarehouse* components constitute the $\mathcal{D}_{ATMexample,MemberBank}$ graph and can also be removed. Completeness is reached by selecting and connecting new components. In this example, an *IndependentBank* component is connected to the *ATM* component through its *Money_Transaction* primitive port. At this step, the assembly is not yet complete because all the components are not yet coherent. Indeed, the *IndependentBank* component is not coherent because its *Manage_Withdraw* composite port is not coherent. Another component is thus added to the assembly: the *BankIS* component is connected to the *IndependentBank* component through its *Request_Data* primitive port. At this point, the assembly is complete. As a result, one can then consider that the removed component has been replaced by an assembly composed of the *IndependentBank* and the *BankIS* components.

Optimization of the Re-building Process using Strategies and Heuristics

As described previously, the optimization problem is defined as a CSP. Our search space is the set of possible assemblies, considering only syntactical compatibility rules to connect ports. Assemblies that satisfy a set of functional objectives and consistency properties (connection dependencies) are searched for in this search space. Our solution strategy classically uses backtracking [35] to enumerate all possible connections and incrementally build all possible assemblies. Backtracking is combined with a branch-and-bound (B&B) strategy [36] that prunes the solution exploration tree. The objective function to be minimized is the number of connections in the assembly. For a given assembly, this amounts to minimizing the number of *non-nil* valuations for the $Connections_\rho$ variables. As quoted in [37], B&B techniques have little been used in SBSE although there are some exceptions: B&B is used to deal with *the next release problem* where requirements are chosen under some resource and dependency constraints [38], and for solving *the staffing problem* expressed as a CSP.

We measured the performance of the building algorithm. We chose to test the whole building algorithm instead of its restriction to the re-building of an incomplete assembly after the removal of a component. In other words, we started building assemblies from scratch instead of starting from an incomplete sub-assembly. For this purpose, we implemented a test environment that generates random component sets, thus providing various building contexts, differing in both size and complexity. Once a component set is generated, an arbitrary number of ports can be chosen as functional objectives and the building algorithm can be launched. Our experiments show that the combinatorial complexity of the building process is quite high, as illustrated in the next section. To use our approach in highly demanding situations, such

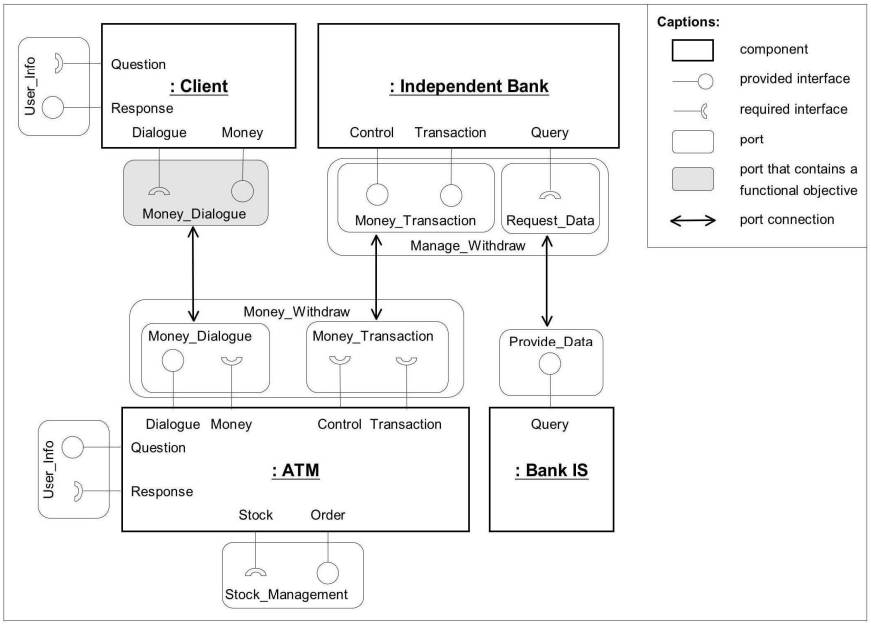


Figure 5. Dynamic reconfiguration of the assembly

as runtime deployment and configuration of components, we studied various heuristics that speed up the building process.

B&B strategy to build minimal assemblies. A first strategy is to try to find not all the possible assemblies but only the most interesting ones. Minimality is an interesting quality metrics for an assembly [39]. More precisely, we try to minimize the number of primitive port connections: fewer connections entail fewer semantic verifications, fewer interactions and, therefore, fewer conflict risks. Fewer connections also entail more evolution capabilities (free ports). To efficiently search for minimal assemblies, we added a branch-and-bound strategy to our building algorithm. The bound is the maximum number of primitive port connections allowed for the construction of the assembly. When this maximum is reached while exploring a branch of the construction tree, the rest of the branch can be discarded as any new solution will be suboptimal relative to any previously found solution (pruning).

Look-ahead (LA) Strategy. An estimate can be used to predict if traversing the current construction branch can lead to a minimal solution. This estimate is based on the minimum number of primitive port connections required to complete the building process. As soon as the sum of the estimate and the number of already existing connections is larger than the current bound, the branch can be pruned. A simple example of such an estimate is the number

of ports in the FO-set divided by two, which corresponds to a lower bound of the number of connections needed to connect the ports that already are in the FO-set (in the most optimistic case, each port from the FO-set can connect to another port from the FO-set thus adding no new dependency and, moreover, satisfying two dependencies at once). A more selective and realistic estimate consists in calculating how many of primitive port pairs from the FO-set can be connected with one another. The number of remaining connections is higher than the cardinality of the FO-set minus the number of primitive port pairs.

Min Domain (MD) Heuristic. This heuristic is used to efficiently choose primitive ports from the FO-set in step 1 of the building algorithm. To each port can be associated the set of primitive ports it can be connected to. Amongst these, the port with the fewest free compatible ports is chosen first. This minimizes the effort to try all the connection possibilities: in case of repeated failures, impossible constructions can be detected sooner.

Min Effort (ME) Heuristic. In the branch-and-bound strategy, every time the bound is lowered, traversal of the tree is speeded up. During step 2 of the building algorithm, when choosing the compatible primitive port to connect to, the free compatible primitive port that belongs to the composite port (γ) that contains the fewest primitive ports (smallest $Card(PrimPorts^*(\gamma))$) is chosen first. This corresponds to choosing the primitive port that adds the fewest dependencies, thus minimizing future connection efforts. Another similar situation occurs during step 3 of the building algorithm: when the primitive port to connect to is chosen, if it is shared by several composite ports, then the composite port that contains the fewest primitive ports is chosen as the first collaboration context to explore.

No New Dependency (NND) Heuristic. In step 2 of the building algorithm, compatible ports are first searched for in the FO-set itself. When a compatible port can be found in that set, it is preferred to others because its connection will add no new dependency and, furthermore, will satisfy two dependencies at once—indeed, when a port belongs to the FO-set, its related primitive port is already in the FO-set.

Implementation and Experimentation

The two processes presented above (automatic component assembly building and dynamic substitution after a component removal) have both been implemented as an extension of the open-source Julia implementation[**] of the Fractal component model [3].

Experimentation framework. To evaluate the applicability and usefulness of the built assemblies and the optimization techniques, we needed a test environment. We were not able to experiment on real components because real-world component repositories, with properly documented behavior, are not yet available. Indeed, to (manually or semi-automatically) add

[**] `http://www.objectweb.org`

	Build 1	Build 2	Build 3	Subst 1
Number of components in a base	15	20	30	40
Max. number of primitive ports by component	10	10	10	29
Max. number of composite ports by component	4	10	10	3
Max. number of primitive ports by composite port	5	6	6	6

Table I. Variable values defining experimentation contexts: three building contexts of growing complexity and a substitution context

ports to components, component behavior must be described in an abstract way (for example, with protocols). We expect that research aiming at facilitating component reuse will encourage the building of such component repositories, thus providing better frameworks for future experimentation. To overcome this lack of real repositories, instead we simulated component repositories, aiming to define components as complex as real ones. Moreover, as a meantime alternative, we also plan to contribute to standardizing benchmarks in SBSE by providing our simulated repositories data online. As it is already the case in other applications of search-based methods, this will contribute to enabling comparisons and increasing reproducibility.

We implemented a test environment that generates random component sets, thus providing various building contexts, differing in both size and complexity. Once a component set is generated, an arbitrary number of ports can be chosen as functional objectives and the building algorithm can be launched. In this environment, a test repository has the following characteristics:

- *Fixed parameters.* The number of randomly generated method names is set to 5000, the number of randomly generated interfaces to 150, the maximum number of methods in an interface to 5, the maximum number of interfaces in a primitive port to 5, and the number of initial functional objectives to 3.
- *Variable parameters.* Depending on the experiments, we tried various values for some of the other characteristics. For example, the number of components in a component repository, the maximum number of primitive or composite ports by component, and the maximum number of primitive ports by composite port were variable parameters. This allowed us to have problem instances of various complexities.

Evaluation of the building algorithm. To evaluate the building algorithm, we empirically defined 3 building contexts that allowed to increase complexity and see how robust our heuristics were (see Table I). More precisely, for each context, we generated 3 different component repositories, and for each repository, we randomly chose 3 different initial functional objective sets. Then, for each FO-set, we ran the building algorithm, to build minimal complete assemblies, 5 times. Results are synthesized in Table II which shows how the algorithm behaves when various sets of strategies and heuristics are used. The table records both the percentage of runs that did not exceed 2700 seconds (45 minutes), and, when applicable, the average execution times (in seconds) for such runs. A run is a complete search for minimal solutions.

	No heuristic		B&B		B&B+LA		B&B+LA+MD		B&B+LA+ MD+NND+ME	
Build 1	55%	1347	100%	8	100%	2	100%	2	100%	1
Build 2	0%		0%		89%	22	100%	57	100%	9
Build 3	0%		16%	106	100%	12	100%	5	100%	3

Table II. Comparison of the percentage of completed runs and average execution time of the building algorithm while varying strategies and heuristics

% solved cases	80
% one-to-one substitution among solved cases	19
% reused dead components	20

Table III. Synthetic view of results for reconfiguration experiments

As execution is interrupted after 45 minutes, 0% means that all runs have been interrupted before the search for minimal solution was completed. 100% means that all runs succeeded in founding all minimal solutions. Execution times lower than one second are simply noted as 1 second. Results show how the whole set of strategies and heuristics are necessary for and efficient at taming the building process complexity. Minimal solutions vary in size from 3 to 35 connections and from 4 to 18 components. As the simulated situations corresponding to the third context seem to be more complex than any typical component assembly found in the literature, we decided it was not worth trying further heuristics or switching to an incomplete search method.

Evaluation of the dynamic reconfiguration approach. Our dynamic reconfiguration approach has been tested in the same environment used to test the building process. The experimentation context is defined by the variable values shown in last column of Table I. We generated 10 component repositories for this context. Then, to test our solution for evolution, we started from a generated complete component assembly from which a randomly chosen component was removed. The substitution process was then triggered by considering that the removed component was not available anymore. We ran the dynamic reconfiguration process 40 times, each time with a newly generated assembly (varying the set of functional objectives) and a new component to remove. Results are synthesized in Table III. Those experiments showed that our solution provides alternative substitution possibilities (compared to existing one-to-one substitution mechanisms), thus is more flexible because it does not depend on the presence of a component that can exactly match (the role of) the removed one. In some situations (20%), no solution exists—the repository does not contain components that can be combined to be substituted to the removed one— but among the solved situations, 81% are solved

thanks our many-to-one substitution proposal (compared to only 19% that can be solved with usual one-to-one substitution techniques). Furthermore, the resulting substitution was usually many-to-one. Also, we noticed that the complexity of the mechanism exposed here is not higher than the complexity of the complete building process—which was efficient thanks to the optimization strategies and heuristics.

Conclusion

To strengthen the ability of component-based software to dynamically evolve, we presented a solution for the dynamic replacement of a component from an assembly. Its originality lies in the fact that it is not restricted to component-to-component (one-to-one) substitution. Our approach requires that components carry information on the possible collaborations they can establish with other components, embodied by primitive and composite ports (similar to complex plugs). Using this information, a search-based mechanism builds a minimal sub-assembly in order to replace the removed component while guaranteeing there is no functional regression. A cleaning step then removes the useless components. The advantage of this approach is that it increases the number of reconfiguration possibilities by being less constraining. As the problem of assembly (re-)building is highly combinatorial, optimization strategies and heuristics have been proposed, implemented, and compared. The whole solution is implemented as an extension of an existing open source implementation of the Fractal component model and successfully tested on generated components.

Next steps will consist in experimenting our approach using real-world software components: our experimentation framework allowed us to validate our approach and be confident that it can deal with realistic situations. Another open issue is component documentation with primitive and composite ports. We are currently investigating strategies to automatically generate ports from protocols (in a design for reuse process) or from execution traces obtained by executing component assemblies (in a design by reuse approach). Run-time replacement of a component also raises the problem of identifying the minimal (yet sufficient) set of components that have to be stopped. We plan to investigate how port connections could help provide an efficient solution to this problem.

Acknowledgements. The authors thank the anonymous reviewers for the usefulness and precision of their comments that allowed to greatly increase the quality of this paper.

REFERENCES

1. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
2. Frantisek Plásil, Dusan Balek, and Radovan Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. of the Int. Conf. on Configurable Distributed Systems*, pages 43–52, Washington, DC, USA, 1998. IEEE Computer Society.
3. E. Bruneton, T. Coupaye, and JB. Stefani. Fractal specification - v 2.0.3, February 2004. `http://fractal.objectweb.org/specification/index.html` [3 July 2008].
4. Bart George, Régis Fleurquin, and Salah Sadou. A substitution model for software components. In *Proc. of the 2006 ECOOP Workshop on Quantitative Approaches on Object-Oriented Software Engineering (QaOOSE'06)*, Nantes, France, July 2006.

5. Tomas Bures, Petr Hnetynka, and Frantisek Plásil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA*, pages 40–48. IEEE Computer Society, 2006.
6. Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard. Automating the building of software component architectures. In Volker Gruhn and Flavio Oquendo, editors, *Software Architecture: 3rd European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA)*, volume 4344 of *LNCS*, pages 228–235. Springer, 2006.
7. Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.
8. Ivica Crnkovic. Component-based software engineering—new challenges in software development. *Software Focus, John Wiley & Sons*, 2(4):127–133, December 2001.
9. Remco M. Dijkman, Joao Paulo Andrade Almeida, and Dick A.C. Quartel. Verifying the correctness of component-based applications that support business processes. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proc. of the 6th Workshop on CBSE: Automated Reasoning and Prediction*, pages 43–48, Portland, Oregon, USA, May 2003.
10. Nenad Medvidovic and Richard N.Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.
11. Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Trans. Softw. Eng. Methodol.*, 9(3):239–272, 2000.
12. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. of the 8th European software engineering conference*, pages 109–120, New York, NY, USA, 2001. ACM Press.
13. Martin Mach, Frantisek Plásil, and Jan Kofron. Behavior protocols verification: Fighting state explosion. *International Journal of Computer and Information Science, ACIS*, 6(1):22–30, March 2005.
14. Viliam Holub and Frantisek Plásil. Reducing component systems' behavior specification. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 63–72, Washington, DC, USA, 2007. IEEE Computer Society.
15. Viliam Holub and Petr Tuma. Streaming state space: A method of distributed model verification. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 356–368, Shanghai, China, June 2007. IEEE Computer Society.
16. Kurt C. Wallnau. Volume III: A technology for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon University, Pittsburgh, OH, USA, April 2003.
17. Jiri Adamek and Frantisek Plásil. Partial bindings of components - any harm? In *APSEC '04: Proc. of the 11th Asia-Pacific Software Engineering Conference*, pages 632–639, Washington, DC, USA, 2004. IEEE Computer Society.
18. Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*, pages 287–325. Springer, 2003.
19. David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.
20. Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4):314–335, 1995.
21. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proc. of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.
22. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proc. of ICSE*, pages 187–197, Orlando, FL, USA, May 2002. ACM Press.
23. Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Softw. Eng.*, 21(4):373–386, 1995.
24. Robert John Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
25. Jasminka Matevska-Meyer, Wilhelm Hasselbring, and Ralf H. Reussner. A software architecture description supporting component deployment and system runtime reconfiguration. In *Proc. of the 9th Int. Workshop on Component-Oriented Programming (WCOP '04)*, Oslo, Norway, June 2004.
26. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of 2Oth Intl. Conf. on Software Engineering*, pages 177–187, Kyoto, Japan, April 1998. IEEE Computer Society.
27. P. Brada. Component change and version identification in SOFA. In *SOFSEM '99: Proc. of the 26th Conf. on Current Trends in Theory and Practice of Informatics*, pages 360–368, London, UK, 1999.

Springer-Verlag.
28. Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay. Automated and unanticipated flexible component substitution. In Helnz W. Schmidt, Ivica Crnkovic, Georges T. Heineman, and Judith A. Stafford, editors, *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE2007)*, volume 4608 of *LNCS*, pages 33–48, Medford, MA, USA, July 2007. Springer.
29. F. Plásil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.
30. OMG. Unified modeling language: Superstructure, version 2.0, 2002. `http://www.omg.org/uml`.
31. Ana Elisa Lobo, Paulo Asterio de C. Guerra, Fernando Castor Filho, and Cecilia Mary F. Rubira. A systematic approach for the evolution of reusable software components. In *ECOOP'2005 Workshop on Architecture-Centric Evolution*, Glasgow, july 2005. `http://wi.wu-wien.ac.at/home/uzdun/ACE2005/04-lobo.pdf` [4 July 2008].
32. Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Precalculating component interface compatibility using FCA. In Jean Diatta, Peter Eklund, and Michel Liquière, editors, *Proceedings of the 5^{th} international conference on Concept Lattices and their Applications (CLA 2007)*, pages 241–252, Montpellier, France, October 2007.
33. C.B. Jones. *Systematic Software Development using VDM (2nd Edition)*. Prentice-Hall, 1990.
34. J.-R. Abrial. *The B-Book, Assigning programs to meanings*. Cambridge University Press, 1996.
35. Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
36. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimisation, Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, 1982.
37. Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE '07)*, pages 342–357, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society.
38. Anthony J. Bagnall, Victor J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information & Software Technology*, 43(14):883–890, 2001.
39. Alejandra Cechich, Mario Piattini, and Antonio Vallecillo, editors. *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *LNCS*. Springer, 2003.

AUTHORS' BIOGRAPHIES

Nicolas Desnos has recently (2008) obtained his PhD from Montpellier 2 University. He worked on defining an autonomic process to build complete software component assemblies. His research interests are component-based software engineering and heuristic search techniques.

Marianne Huchard is a fulltime professor at the LIRMM laboratory (CNRS and Université Montpellier 2) since 2004. Her research interests include Formal Concept Analysis (FCA) and Relational Concept Analysis (RCA) for software engineering (class model reengineering, component directories), model driven engineering for FCA and RCA applications as well as component-based software engineering.

Guy Tremblay is a fulltime professor in the computer science department of the Université du Québec à Montréal (UQAM) since 1985. He is also a member of the LAboratory for research on Technology for E-commercE (LATECE). His research interests include parallel programming, model checking techniques, business process modeling languages and formal methods for (web) service composition and software components.

Christelle Urtado is a fulltime assisant-professor at the Ecole des Mines d'Alès since 1999. Her resarch interests include self-* approaches for component-based software engineering (component self-assembly, component evolution, component directories) and fault tolerance (exception handling) in component-based or agent-based software systems. She also interests in helping designers build and maintain complex software systems.

Sylvain Vauttier is a fulltime assistant-professor at Ecole des Mines d'Alès since 2000. His research interests encompass component and agent-based software engineering techniques. His work, focused on behavior composition mechanisms, is more recently applied on the autonomous construction and evolution of software deployed on ambiant intelligence environments.

Annexe F

User-defined scenarios in ubiquitous environments : creation, execution control and sharing

User-defined Scenarios in Ubiquitous Environments: Creation, Execution Control and Sharing

Matthieu Faure, Luc Fabresse
Ecole des Mines de Douai, Douai, France
{Matthieu.Faure, Luc.Fabresse}@mines-douai.fr

Marianne Huchard
LIRMM - UMR 5506, CNRS and Univ. Montpellier 2, Montpellier, France
huchard@lirmm.fr

Christelle Urtado and Sylvain Vauttier
LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

***Abstract*—Ubiquitous computing provides a dynamic access to different functionalities of networkable electronic devices. Whereas basic services have limited use, predefined complex services cannot encompass every end-user's needs nor be adapted to a set of services that are dynamically discovered in an open environment. Alternatively, users need to be provided with means to express their requirements, choosing precisely which services to compose into a scenario of their own. In service-oriented computing, some systems propose mechanisms to develop tailored components that provide composite services; however they are not adapted to end-users, have limited composition capabilities and/or do not consider several characteristics of ubiquitous environments (such as multiple users and devices).**

This paper presents a novel user-centric system called SaS for mobile personal devices. SaS provides end-users with an easy access to services and a simple GUI to combine them into complex scenarios. A new architectural description language is used to specifically support scenario creation by service composition. Scenario may be shared among users and devices. SaS offers scenario execution control for example to start and stop it but also to query the current state of a scenario. In addition, SaS proposes some mechanisms to maintain scenario availability in case of service/device unavailability. SaS is currently implemented in a proof-of-concept prototype on top of OSGi.

Keywords*—*Ubiquitous computing, service-oriented computing, user-centric system, service composition, scenario creation.

I. INTRODUCTION

With the rise of ubiquitous computing [1], [2], we are surrounded by electronic devices (such as smart phones or TVs) that propose a huge amount of services through public or private networks. According to the OASIS organization, *"a service is a mechanism to enable access to one or more capabilities"* [3]. In service-oriented computing (SOC) [4], and specially in home automation [5], [6], [7], efforts have been made to facilitate the use of these electronic devices through their services. As shown in Figure 1, the more complex user requirements can only be satisfied by compositions of multiple services provided by multiple devices. Different service composition means have been studied and proposed [8], [9], [10], [11]. However, they are not designed for end-users without technical knowledge.

Enabling end-users to describe their own *scenarios* is a first improvement and a step towards ambient intelligence [12]. In addition, users should easily manage the created scenarios and have access to them from several control devices (PDA, mobile phone or laptop). Moreover, ubiquitous environments imply that several users might be eager to share scenarios. Scenarios should therefore be exported in the environment to be shared and reused.

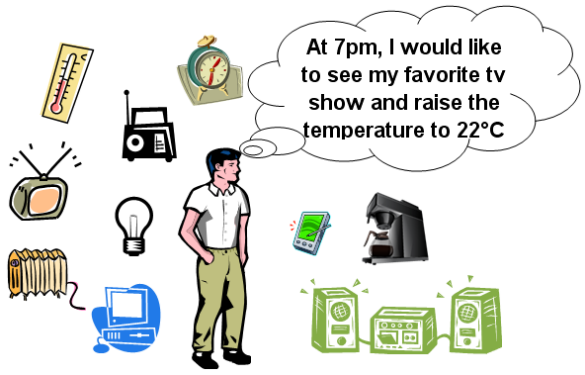


Figure 1. User's main issue

In this paper, we propose the SaS (Scenarios as Services) system specifically designed for end-users to describe, use and share *scenarios* in a high level manner. SaS comprises a new architecture description language (ADL) [13], [14], [15] dedicated to user scenario creation. SaS also integrates a graphical user interface (GUI) based on this ADL. This GUI presents a classified and filtered view of the services available from a set of widely spread devices and provides tools to easily *compose* selected services. SaS integrates scenarios as *regular services*. This enables easy scenario execution control, scenario sharing among users and hierarchical composition of scenarios. The SaS system has been prototyped. Its current implementation is on top of OSGi [16].

The remainder of this paper is structured as follows. Section II introduces the context of this work, presents the requirements for open and distributed environments and discusses the state of the art. Section III presents the first part of our proposition: scenario creation. Section IV is dedicated to scenario execution control and sharing. Section V presents the architecture of the prototype implementation. Finally, section VI evaluates our proposition, concludes and draws some perspectives to this work.

II. User-Centric Systems in Ubiquitous Environments

This section first describes the terminology of ubiquitous environments and especially that of user-centric systems. It then presents the requirements that are mandatory for that kind of systems. It finally compares some of the main state of the art approaches regarding these requirements.

A. Terminology

Ubiquitous systems involve multiple *users* and multiple *devices* that each provide a set of *services*. A device is an electronic object (such as a clock). Devices publish services (such as `Time`). Each service provides one or more *operations* (such as `getTime` or `setTime`). They are called "capabilities" in the SOA norm [3] by the OASIS consortium. End-users use these operations as an access to functionalities of devices.

Devices can interoperate but the overall goal that the system has to achieve always comes from users. Users can be simple consumers or technical experts that command devices, their needs can always be considered as *scenarios* which are combinations of operations. However, we choose to name this combination a *service composition* because services are not stand-alone elements and to stick to the terminology used in SOC.

A SaS system is deployed on a mobile personal device. The system and the device on which it is deployed together define a SaS *platform*. A SaS platform participate in one or more networks which constitute the platform's environment. The global environment is the union of all the environment of its constituting SaS platforms.

B. Requirements for user-centric systems

User needs always constitute a scenario. User-centric system must therefore enable *scenario creation*. As seen in Figure 1, user scenarios are not always simple service aggregations but can imply conditions, control statements and logical operators. Users should thus be able to *compose services* according to their needs. Most of the users are not technical experts. Scenario creation should therefore be *user-friendly* and adapted to devices. Ubiquitous environments imply multiple users and devices. So, created scenarios should be available into the environment and *shared* among users. Users must be able to easily start and stop created scenarios and check scenario status and execution advancement. Thus, the system should control the *scenario life-cycle*. Moreover, already created scenarios should be easily modified and *recomposed* into other ones. In addition, devices that provide services and/or scenarios can disappear. The system must therefore *maintain* scenario execution and availability in case of device disappearance.

C. State of the art

With the requirements established previously, we can analyze some of the main systems that provide a solution for ubiquitous environments and enable end-users to create scenarios.

- **SLCA** [17] provides developers with the capability to compose web services. A composite service contains proxy components attached to involved web services. SLCA enables hierarchical service composition. In addition, it is an event-based system which adapts to environment changes. In case of service unavailability the composite service replaces it if an appropriate service is found. If not, the composite service removes the proxy component attached to this service.
- **MASML** [6] is a multi-agent system for home automation. Scenarios are defined with an XML syntax and consist of a sequence of service operation invocations. MASML XML documents can embed ECMA scripts [18] to add logic elements. A mobile agent is in charge of scenario execution. It is moving to each appropriate device with the scenario description file to execute it. This enables scenario advancement tracking but not parallel execution.
- **SODAPOP** [19] proposes an innovative approach based on the same observation than us: user needs are scenarios. What is important is the goal to achieve. The main hypothesis is that each service contains informations about its initial conditions and its effects. SODAPOP automatically classifies new services with these informations. Then, it can combine some of them to reach the user's goal.
- **SASHAA** [20], [21] is one of our previous work, focused on ubiquitous systems for home automation. It enables end-users to create scenarios with Event - Conditions - Action rules through an appropriate GUI. It is an adaptive system which creates a new component for each new device.

Table I compares these systems with respect to the requirements that we identified in II-B. Symbol √ means that the requirements is fulfilled, - signifies that it is partially accomplished and X represents an absence of solution.

Systems	Scenario Creation	Advanced Service Composition	User firendliness	Scenario sharing	Scenario Lifecycle	Hierarchical Composition	Scenario Maintenance
SASHAA	√	-	√	X	-	X	√
SLCA	√	√	X	X	X	√	-
MASML	√	√	X	X	-	X	√
SODAPOP	√	X	-	X	X	X	X

Table I
SYSTEM COMPARISON WITH OUR REQUIREMENTS

Except for SAASHA, we can notice in Figure I that all these works propose programming tools for developers. They are not directed to end-users. In addition, scenario sharing is never took into consideration. For scenario life-cycle control, SASHAA only enables to start and stop scenarios whereas MASML just allows users to check scenario advancement. Because of this, we decided to propose a new system which best meets all the expectations of user-centric systems for ubiquitous environments.

III. The SaS System: Scenario Creation

A. Overview of SaS

The purpose of SaS which stands for *Scenarios As Services* is threefold:

1) help end-users create scenarios by service composition,
2) monitor scenario execution on a given platform,
3) export scenarios into the environment for future use or sharing.

To do so, several steps are necessary that define a user-centric cycle, as illustrated by Figure 2:

0. The system (placed into a user device) discovers the services available in its neighboring environment.
1. SaS classifies service operations depending on their providers (devices) and services. It then displays them.
2. Users can compose several available services to create a scenario. This is possible through a dynamically adaptive graphical user interface based on our ADL.
3. The created scenario is translated into a descriptor file. It therefore becomes easily transmissible and can be shared with other platforms and users.
4. Next, SaS analyzes the scenario descriptor. It extracts information about the different services involved and how they are composed.
5. The system creates a composite with the involved services and a generated *manager*. This manager handles the services according to the previously made user choices.

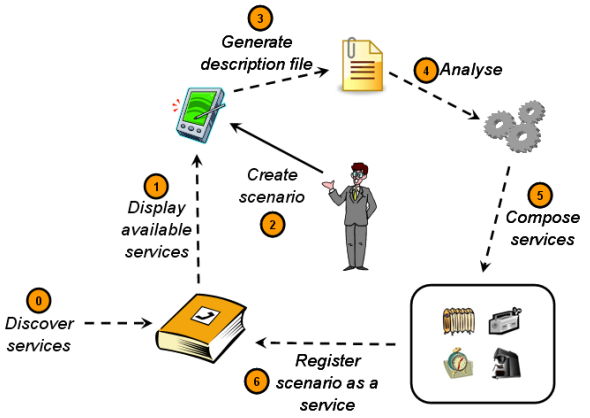


Figure 2. Overview of the proposed SaS scenario creation and reuse cycle

6. Finally, the manager, which is in the composite, registers the scenario as a new service into the environment. It becomes accessible from other devices and shared among end-users. Moreover, it can be composed into a new scenario.

B. Scenario creation

This section describes scenario creation, which is the main part of our system. It consists in three steps: service selection, scenario construction by service composition and scenario export. As described in Section III-A, the first functionality of the SaS system is service discovery. Some protocols already exist that do so (*e.g.* UPnP [22], SLP [23], Jini [24]) along with different extra functionalities. To be as interoperable as possible, our SaS system does not prescribe the use of a particular discovery protocol. Once the available services are discovered, they can be listed and ordered by SaS to enable service selection.

1) Service Selection: Every service proposes one or more operations (for example, the *light* service might offer two operations: `getValue` and `setValue`). To define a scenario, users always select operations, but the name of the service and the identity of the provider device do not always matter. For example, to print a document a person generally chooses his favorite printer, accesses the specified service and selects the appropriate operation. Alternatively, if he needs to know what time it is, he directly selects the `getTime` operation, no matter which clock or service provides it.

Service selection in SaS sticks to this requirement. SaS proposes three filtered views to select available operations: by device type (*e.g.* the list of available printers), by service name (*e.g.* the *printService* service) or directly by operation name (*e.g.* the `print` operation). If users select a device, services provided by this device are then proposed to choose from. If users select a service, operations that compose this service are then proposed to choose from. Moreover, distinct devices can propose services with the same name, sometimes with additional operations. SaS groups these services together and displays all the available operations collectively.

2) Service Composition: SaS enables service composition thanks to an ADL and its GUI. Depending on user choices, created scenarios can be then exported into the environment.

a) Presentation of the ADL: In order to help end-users create scenarios that correspond to their needs, we propose a new ADL. It is simple and tailored to scenario creation. Compared to other programming languages for service composition (like BPEL [25]) which are imperative and designed for executable process, our ADL is a high level language, declarative and destined to end-users. With this ADL, one can declare both services and scenarios.

- Service declaration
  We define a service by a device (its provider), a name and an operation list. This list cannot be empty. Operations have a return type (which can be *void*) and can have typed parameters. We represent only the main elements of the grammar in Listing 1.

```
<service> ::= service <device> <service_name> <op_list>

<op_list> ::= ( <operation> ; )*

<operation> ::= operation <operation_name> (
   [<parameter_list>] ) : <return_type>

<parameter_list> ::= <parameter_type> (,<parameter_type>)*

<return_type>::= <type>
<parameter_type> ::= <type>
```

Listing 1. Service declaration with the Backus–Naur Form (BNF)

- Scenario declaration
  By definition, a scenario has a name and an action list. An action can be:
  - **an operation invocation**: a service operation is invoked, with its parameter values. Users can directly enter parameter values or invoke another service operation to create the desired value (operation composition). SaS checks if parameter types conform to the service definition.
  - **an alternative (*if - else*)**: conditions compare the result of two service operations or the result of a service operation and a value chosen by the user.
  - **a repetition loop**: enables `while` loops iterations while a condition remains satisfied. Alternatively, it is possible to precise how many `times` a series of actions should be invoked.

Listing 2 describes the main elements of a scenario declaration using the BNF notation.

```
<scenario> ::= scenario <scenario_name> <action_list>

<action_list> ::= { <action> + }
<action> ::= <op_invocation> ; | <alternative> | <repeat>

<op_invocation> ::= [<device>] <service_name>.
  <operation_name>([<parameter_list>])
<parameter_list> ::= (<op_invocation>|<parameter_value>)
                     (, (<op_invocation>|<parameter_value>))*

<alternative> ::= if <cplx_cond><action_list> [<else_clause>]
<else_clause> ::= (else <action_list> )*
<cplx_cond> ::= (<condition> (<log_operator><condition>)*)
<condition> ::= <op_invocation> <comp_operator>
                (<op_invocation> | <compare_to_value>)
<repeat> ::= (while <cplx_cond> | <repeat_value> times)
             <action_list>

<log_operator> ::= and|or|not
<comp_operator> ::= < | <= | > | >= | ==
```

Listing 2. Grammar of the scenario declaration using the BNF notation

b) The graphical user interface: This ADL syntax is simple and declarative as we can see in the example on the right of Figure 3. Nevertheless, SaS proposes a more user-friendly option to create scenarios through a graphical representation of the ADL. Users therefore do not manipulate the ADL anymore but compose service operations with basic instructions (based on the *operators* of our ADL): `if`, `else`, `while`, `times`, `and`, `or`, `not`, `<`, `>`, `≤`, `≥`, `==`. For parameters entries, users can select an operation result or choose fixed values and apply an arithmetic operation (such as `+`, `−`, `*`, `/`).

Once the scenario is defined, users can choose to export it into the environment. They also have to specify if the scenario can be redeployed into another SaS platform. Thanks to a transformation process, the scenario is then automatically transcribed into our ADL. Figure 3 (right) shows the scenario transcription with a simplified version of the GUI (left).

3) Composite service creation: After the scenario is created, SaS analyzes its description file to create a composite that manages

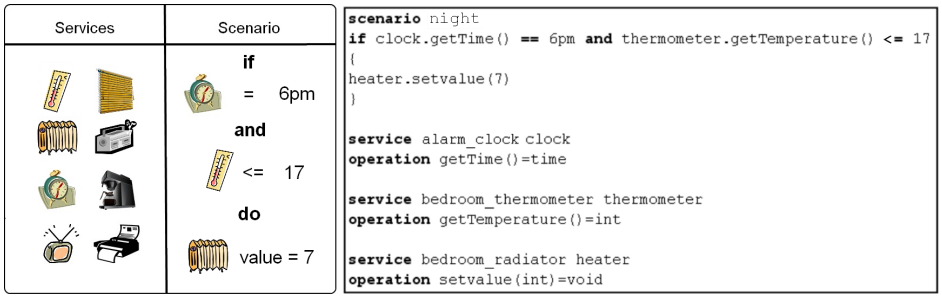


Figure 3. Scenario Transcription: from our ADL

its deployment. This composite includes references to the services chosen in the scenario and a *Scenario Manager*. The manager has two roles: manage the different services and export the scenario as a new service in the system according to users preferences.

Depending on user choices, services instantiated inside the composite are specific to a device or come from any of its available providers[1]. In this last case, if the service provider disappears, SaS dynamically recomposes the composite that implements the scenario to integrate another implementation of the same service (if available). Figure 4 illustrates composite services with an example. The scenario is simple and placed in a home automation environment: at 6pm, close the main door and set the thermostat at 7. There are three services: only one is defined from a specific device (the main door). Others are instantiated from any devices that provide these services.

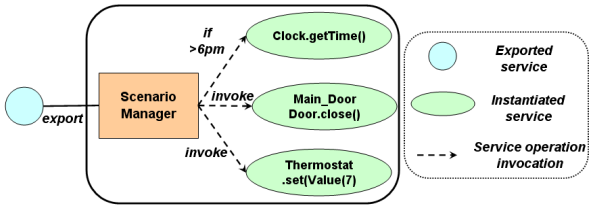


Figure 4. A Composite Service

IV. Scenario Execution Control and Sharing

Once scenarios are created, they can be shared among users. In addition, scenarios are easily manageable and should stay available into the environment in case of service or device unavailability.

A. Scenario sharing

As seen in subsection III-B3 the Scenario Manager registers the scenario as a new service. The scenario can then be used as a service and, as such, composed into a new coarser grained scenario (scenario hierarchical composition). This service has four operations: `start`, `stop`, `getScenarioState` and `getDescriptor`. It does not describe the functioning of the scenario: it hides services and their interactions inside the composite. This guarantees encapsulation. Reflexion is nonetheless provided (composites are not black boxes but gray boxes) thanks to a service operation: `getDescriptor`. This operation provides access to the scenario descriptor file. Users can directly read this file or obtain a visual transcription of the scenario on his GUI.

Figure 5 illustrates scenario sharing. The user of the SaS platform named A creates and exports a scenario. This scenario is registered as a new service. It is then discovered by platforms B and C. The user of platform B recomposes this scenario into a new one, whereas the user of platform C just gets an overview of the scenario on its GUI.

[1]An optimized selection scheme is a perspective.

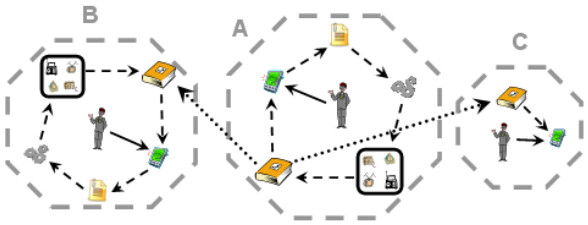


Figure 5. Overview of SaS scenario sharing

B. Scenario execution control

We define scenarios as service compositions. We can see a scenario as an active entity, which evolves, changes from one state to another. Moreover, the execution contains several steps which can fail or succeed. For example, users that discover a scenario might want to know if this scenario is currently in execution. If so, it is important to check which steps have been executed, which succeeded and what are the next steps. This is why, SaS manages scenarios' life-cycles and enables to check scenario execution status.

1) Scenario life-cycle: Scenarios are dynamic. They can be executed, stopped, have a missing service... The state diagram of Figure 6 illustrates the different states of scenario life-cycle which are:

- **Installed,** the scenario has been deployed and registered (and so discovered) as a new service. SaS automatically checks if the different involved services are present.
- **Ready to launch,** all involved services are available. If a service disappears, the scenario goes to the previous state.
- **Running,** the scenario has been launched and is currently executed. The scenario could finish and come back to the previous state or be interrupted.
- **Stopped,** the scenario has been stopped by a user or a service inside the scenario disappeared. The scenario is paused waiting to be restarted or to be executable again by the appearance of an appropriate service.

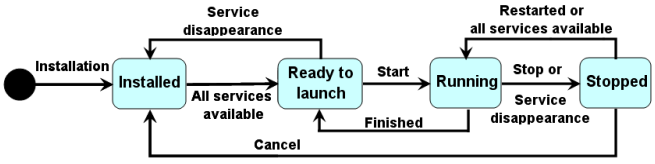


Figure 6. State diagram of scenario life-cycle

2) Scenario running state advancement: Users must know which scenario operations are running and which have already been executed. This is why, SaS registers scenario execution progress. To do so, SaS considers the *Running* state of the scenario life-cycle as

a succession of stages: the operation invocations. These stages are evaluated depending on their types:

- **Functions**, if an operation is invoked to obtain a result, SaS logs this operation as done when we obtain the result. If an error occurs, SaS continues to execute the scenario if possible (*i.e.* if the operation result is not needed) and displays a warning to the user.
- **Procedures**, if the operation does not return a result, SaS logs it as *executed* when the operation is invoked.

In addition, SaS logs the execution times of the different scenario steps. Users can see when the scenario began and how long every operation took. So, SaS enables users to check the current scenario position and control its correct advancement. Users can get these informations thanks to the `getScenarioState` operation.

C. Scenario availability

With scenario export as new services, users can have access to the same scenario on several devices, however, they might want access to it even if the original provider is off. This is why, scenario access should be maintained if the original provider disappears.

To do so, SaS enables scenario redeployment on other platforms. This is possible because SaS differentiates scenarios from available services. When a scenario appears, every SaS platform downloads the scenario description. Thus, users can directly have an overview of the scenario definition and platforms can redeploy the scenario if the original scenario provider disappears.

V. System Design and Implementation

This section describes the design and implementation of the SaS prototype. It is an ongoing work implemented in Java over OSGi [16], [26] with iPOJO [27]. OSGi is a popular SOC framework that is widely adopted by industry for developers to create *bundles* (Java components). iPOJO is based on OSGi and follows the Service-Oriented Component Model [28]. The main idea is that a component should only contain business logic as in EJB 3.0 [29] (*EJB entities*); SOC mechanisms should seamlessly be handled by the component container as container-managed cross-cutting services.

A. Model

As shown in Figure 7, four components compose SaS. Each of them is packaged as an OSGi bundle because it is safer and easier to update.

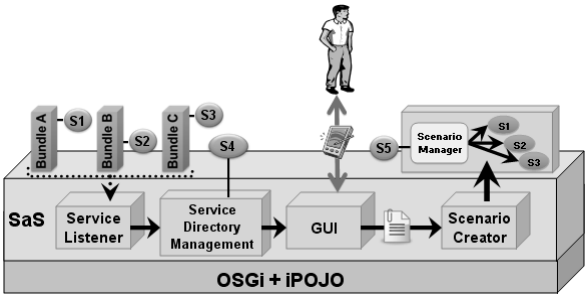


Figure 7. The SaS prototype implemented over OSGi and iPOJO

- **Service Listener**. This bundle obtains, orders and dynamically updates the list of available services from the OSGi context.
- **Service Directory Management (SDM)**. It is the intermediate between the *Service Listener* and the GUI responsible of managing (adding, removing) services and scenarios in the *Service Directory*.
- **GUI**. The GUI is platform and operating system specific (PDA, mobile phone, Android, iOS,...). It provides a graphical representation of our ADL which provides users with the capability to see the available services and compose them.
- **Scenario Creator**. It creates scenario bundles which are composed of the selected services and a scenario manager. This latter manages services inside the composite and exports the scenario as a service.

B. Insights into the SaS prototype

This subsection presents the implementation of the main functionalities of the SaS system.

The latest version (4.2) of OSGI now supports distribution (RFC 119 [26]). So, for service discovery, SaS uses the API of distributed OSGi which can be implemented by many discovery protocols. Then, the *Service Listener* retrieves the list of available services from the directory provided by the OSGi framework and sends it to the *SDM*. This latter orders and classifies the service list. The GUI[2] displays available devices, services and operations. It filters the displayed services (*resp.* operations) if a specific device (*resp.* service) is selected. A user creates a scenario through the GUI which stores it in an XML-based description file. This description becomes easily transmissible and promptly interpretable and analyzable since XML is a standard as an exchange format. Using this description, the *Scenario Creator* of SaS automatically (i) generates a scenario manager which (ii) exports and manages the scenario. Finally, the scenario (iii) is redeployed on other SaS platforms according to user preferences. The remainder of this section describes more deeply this three important steps.

1) Scenario Manager generation: Starting from the XML-based description of a scenario, SaS generates a Java class that represents the manager of this scenario (*Scenario Manager*). To do so, the first step consists to parse the XML description file using a *SAX* parser [30]. SAX translates XML elements into a sequence of Java instructions. Then, SaS generates a *Scenario Manager* class with the Javassist [31] library that enables dynamic byte code edition such as creating classes or modifying existing classes. The *Scenario Manager* is generated as a class that implements the *ScenarioManagerInterface* interface. This interface declares four public methods including a `start` method which is automatically filled in the *Scenario Manager* class with the Java instructions resulting from the SAX parsing.

2) Scenario export and execution control: SaS uses the iPOJO API [32] to dynamically create an OSGi composite bundle that packs together the generated *Scenario Manager* and the involved services. This composite bundle is then installed and started into the OSGi platform. Finally, the *Scenario Manager* registers a new service inside the OSGi directory, specifying its capability to execute four public methods (`start`, `stop`, `getScenarioState` and `getDescriptor`).

For scenario execution control, the *Scenario Manager* creates a log file every time the `start` service operation is invoked with the invoker platform *id* and the current time. Then, the *Scenario Manager* logs in this file every service invocation success (or failure) with time. With this log file, SaS knows at every moment if the scenario is currently in execution, when it began, who launched it and which steps are already executed. These informations are available through the service operation `getScenarioState`.

3) Scenario automatic deployment: As seen in V-B1, all scenarios implement the same Java interface (*ScenarioManagerInterface*). So, SaS can easily recognize them. Thus, when a scenario is discovered as a new service, the *Service Directory Management* automatically gets the scenario description file thanks to the `getDescriptor` operation provided by the service. The scenario is not deployed again but SaS keeps the XML description file. *SDM* sends the scenario description to the GUI. If the original provider disappears, another SaS platform may redeploy the scenario if its directory contains all the involved services.

[2]which is still under development.

VI. Evaluation and Conclusion

With the SaS system, we propose a newly user-centric system that meets the expectations of ubiquitous environments. First, we provide *scenario creation* by service composition. Users can create *complex scenarios* that correspond to their needs thanks to an appropriate ADL. This ADL is simple, user-oriented and proposes an alternative graphical view to be *accessible for everyone*. SaS exports scenarios as new services into its environment, thus, users can easily *share* their scenarios. Moreover, SaS manages *scenario life-cycle*: it enables users to start and stop scenarios, check scenarios status and scenario execution advancement. Moreover, users can get an overview of a scenario specification (scenario introspection capability) thanks to a descriptor file and *reuse a scenario* as a service being part of a new encapsulating scenario composition (scenario hierarchical composition). SaS also tries to *maintain scenario availability*. Locally, if a service involved into a scenario disappears, SaS tries to replace it into the composite. Globally, when a SaS platform disappears, scenarios exported by this platform are redeployed on other ones to remain available. In conclusion, the SaS system presented in this paper satisfies all the requirements defined is section II-B. A prototype of SaS in Java over OSGi and iPOJO is currently under development.

We have three major perspectives. First, we want to evaluate SaS to show its simplicity for non technical end-users by comparing it with existing tools such as Yahoo Pipes [33], Automator [34] and Scratch [35]. Such tools provide non-technical end-users of the capability to graphically develop small applications by composing elements. Then, we plan to define some recovery strategies to anticipate service loss such as *caching* and *hoarding*. Finally, we want to improve scenario distribution and propagate scenarios into the network.

Acknowledgements

This work is partially supported by a grant from the CARNOT M.I.N.E.S Institute (*http://www.carnot-mines.eu/*).

References

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, pp. 78–89, 1995.

[2] H. Schulzrinne, X. Wu, S. Sidiroglou, and S, "Ubiquitous computing in home networks," *IEEE Communications*, pp. 128–135, nov 2003.

[3] OASIS, "Reference Model for Service Oriented Architecture 1.0," pp. 12 – 13, oct 2006. [Online]. Available: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html

[4] M. P. Papazoglou, "Service-Oriented Computing : Concepts , Characteristics and Directions," in *Proc. of the 4th International Conference on Web Information Systems Engineering*. IEEE Computer Society, 2003, pp. 3–12.

[5] A. Bottaro, A. Gérodolle, and P. Lalanda, "Pervasive service composition in the home network," in *Proc. of the 21st International Conference on Advanced Networking and Applications*, 2007, pp. 596–603.

[6] C.-L. Wu, C.-F. Liao, and L.-C. Fu, "Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology," *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 2, pp. 193–205, mar 2007.

[7] D. Valtchev and I. Frankov, "Service gateway architecture for a smart home," *Communications Magazine, IEEE*, pp. 126–132, 2002.

[8] M. Bakhouya and J. Gaber, *Agent Systems in Electronic Business*. IGI Publishing, 2007, ch. Service Composition Approaches for Ubiquitous and Pervasive Computing Environments: A Survey, pp. 323–350.

[9] J. Bronsted, K. M. Hansen, and M. Ingstrup, "Service composition issues in pervasive computing," *IEEE Pervasive Computing*, vol. 9, pp. 62–70, 2010.

[10] A. Urbieta, G. Barrutieta, J. Parra, and A. Uribarren, "A survey of dynamic service composition approaches for ambient systems," in *Proceedings of the 2008 Ambi-Sys workshop on Software Organisation and MonIToring of Ambient Systems*, ser. SOMITAS '08, 2008, pp. 1–8.

[11] N. Ibrahim and F. Le Mouël, "A Survey on Service Composition Middleware in Pervasive Environments," *International Journal of Computer Science Issues (IJCSI)*, vol. 1, pp. 1–12, 2009. [Online]. Available: http://hal.inria.fr/inria-00414117/en/

[12] E. Aarts and B. de Ruyter, "New research perspectives on Ambient Intelligence," *Journal of Ambient Intelligence and Smart Environments*, vol. 1, pp. 5–14, 2009.

[13] P. Clements, "A survey of architecture description languages," in *Proc. of the 8th international workshop on software specification and design*. IEEE Computer Society, March 1996, pp. 16–25.

[14] S. Vestal, "A Cursory Overview and Comparison of Four Architecture Description Languages," Honeywell, Tech. Rep., February 1993.

[15] P. Mishra and N. Dutt, "Architecture description languages," *IEEE proc. - Computers and Digital Techniques*, vol. 152, no. 3, p. 285, 2005.

[16] OSGi Alliance, "OSGi Service Platform Core Specification Release 4," 2005. [Online]. Available: http://www.osgi.org/download/r4v40/r4.core.pdf

[17] V. Hourdin, J. Tigli, S. Lavirotte, G. Rey, and M. Riveill, "SLCA, composite services for ubiquitous computing," in *Proc. of the International Conference on Mobile Technology, Applications, and Systems*. New York, New York, USA: ACM Press, 2008, pp. 1–8.

[18] Ecma International, "ECMA-262: ECMAScript Language Specification," December 2009. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[19] J. Encarnaçao and T. Kirste, "Ambient intelligence: Towards smart appliance ensembles," *From Integrated Publication and Information Systems to Information and Knowledge Environments*, no. December, pp. 261–270, 2005.

[20] F. Hamoui, M. Huchard, C. Urtado, and S. Vauttier, "Specification of a component-based domotic system to support user-defined scenarios," in *Proc. of 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, July 2009.

[21] ——, "Un système d'agents à base de composants pour les environnements domotiques," in *Actes de la 16ème conférence francophone sur les Langages et Modèles à Objets (LMO 2010)*, Mars 2010, pp. 35–49.

[22] UPnP Forum, "Understanding UPnP: A White Paper," 2000. [Online]. Available: http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc

[23] C. Bettstetter and C. Renner, "A comparison of service discovery protocols and implementation of the service location protocol," in *Proc. of the 6th EUNICE Open European Summer School: Innovative Internet Applications*. Citeseer, 2000, pp. 13–15.

[24] G. Aschemann, R. Kehr, and A. Zeidler, "A Jini-based Gateway Architecture for Mobile Devices," *In Proc. of the Java-Informations-Tage (JIT99)*, p. 203–212, September 1999.

[25] OASIS, "Web services business process execution language version 2.0," april 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[26] OSGi Alliance, "OSGi Service Platform Enterprise Specification," pp. 15 – 27, march 2010. [Online]. Available: http://www.osgi.org/download/r4v42/r4.enterprise.pdf

[27] C. Escoffier and R. Hall, "Dynamically adaptable applications with iPOJO service components," *Proc. of the 6th international conference on Software composition*, pp. 113–128, 2007.

[28] H. Cervantes and R. Hall, "Autonomous adaptation to dynamic availability using a service-oriented component model," in *International Conference on Software Engineering (ICSE)*. IEEE, 2004, pp. 614–623.

[29] Sun Microsystems, "Enterprise javabeans specifications," may 2006. [Online]. Available: http://java.sun.com/products/ejb/docs.html

[30] S. Means and M. A. Bodie, *Book of SAX: The Simple API for XML*. No Starch Press, 2002.

[31] S. Chiba and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," *Proc. of the 2nd international conference on Generative programming and component engineering*, pp. 364–376, 2003.

[32] Apache Foundation, "ipojo api," 2010. [Online]. Available: http://felix.apache.org/site/apache-felix-ipojo-api.html

[33] Yahoo, "Rewire the Web." [Online]. Available: http://pipes.yahoo.com/pipes

[34] Apple, "Automator: Your personal Automation Assistant." [Online]. Available: http://www.macosxautomation.com/automator

[35] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for Everyone," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.