Un processus de sélection de composants logiciels multi-niveaux

Bart George

N^o d'ordre: XX

THÈSE

soutenue

devant l'Université de Bretagne Sud

pour l'obtention du grade de :
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD

Mention :
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

par

Bart GEORGE

Équipe d'accueil : Laboratoire VALORIA

Titre de la thèse :

Un processus de sélection de composants logiciels multi-niveaux

Présentée le 13 novembre 2007 devant la commission d'examen composée de :

Professeur	Pierre-François	MARTEAU	Université de Bretagne Sud	Président
Professeur	Houari	SAHRAOUI	Université de Montréal	Rapporteur
MCF HDR	Antoine	BEUGNARD	ENST Bretagne	Rapporteur
Professeur	Christophe	DONY	Université de Montpellier II	Examinateur
MCF HDR	Salah	SADOU	VALORIA,Université de Bretagne Sud	Directeur
MCF	Régis	FLEURQUIN	VALORIA,Université de Bretagne Sud	Encadrant

À la mémoire de ma grand-mère,
À mon grand-père,
À tous les autres,

Ce travail est aussi le fruit de leurs efforts.

Remerciements

Je tiens à adresser mes vifs remerciements à Houari Sahraoui, non seulement pour avoir bien voulu rapporter cette thèse, mais aussi pour m'avoir si bien accueilli à l'Université de Montréal pendant ces deux mois déterminants pour ma thèse. J'adresse également de vifs remerciements à Antoine Beugnard pour avoir accepté de rapporter ce mémoire, ainsi qu'à Christophe Dony pour avoir accepté de l'examiner.

Je tiens aussi à remercier Pierre François Marteau, non seulement en tant que directeur du laboratoire VALORIA, mais aussi pour avoir accepté de présider ce jury et pour nous avoir aidés, moi et les autres membres de l'association étudiante AECUBS, à organiser la conférence MajecSTIC 2006, dont il a bien voulu présider le comité de programme.

Je souhaite exprimer ma reconnaissance infinie envers Salah Sadou, mon directeur de thèse, et Régis Fleurquin, mon encadrant et tuteur. C'est parce qu'ils ont cru en moi que j'ai pu faire cette thèse, et c'est parce qu'ils n'ont jamais cessé pendant toutes ces années de croire en moi, de me faire confiance, de me soutenir, de me conseiller et de me supporter, que j'ai pu la terminer et présenter ce mémoire.

Je remercie l'ensemble du laboratoire VALORIA et l'équipe GEODES du laboratoire DIRO de l'Université de Montréal, pour m'avoir si bien accueilli et pour m'avoir permis de travailler dans des conditions idéales. Je remercie du fond du cœur mes compagnons de l'équipe SE pour m'avoir si bien conseillé et épaulé. Je remercie très amicalement mes compagnons de bureau successifs, que ce soit au VALORIA ou au GEODES. Je remercie également tous mes compagnons d'AECUBS et de MajecSTIC 2006.

Enfin, j'adresse ma reconnaissance et mes remerciements éternels à mon grand-père et à ma défunte grand-mère, car je ne serais jamais arrivé jusque-là sans leur soutien indéfectible, leur gentillesse, leur amour et leur patience sans limites. Cet accomplissement est aussi le leur. Je remercie aussi chaleureusement le reste de ma famille et de mes amis pour leur soutien.

Résumé

Le paradigme composant propose de construire un système à partir d'éléments faiblement couplés et pour la plupart déjà existants. Le taux de réutilisation ainsi atteint entraîne une diminution des délais et des coûts de développement. Pour faire face à la complexité croissante des applications, les entreprises sont de plus en plus obligées d'avoir recours à des composants commerciaux "sur étagère", fournis par des tierces personnes, et dont la nature même impose de repenser profondément le cycle de développement d'un logiciel. Il n'est plus possible de spécifier un besoin ou une architecture sans se demander s'il existe sur le marché un composant capable de satisfaire le premier ou de s'intégrer dans la seconde.

Dans ce contexte, une activité voit son importance renforcée : la sélection de composants. Cette activité est sensible : une mauvaise définition des besoins associée à une mauvaise sélection des composants peut conduire à des catastrophes financières, voire même humaines dans certains cas. Elle est de plus très coûteuse car elle impose le parcours de marchés comportant des milliers de composants, décrits avec des formats potentiellement très différents. La sélection devient au final très consommatrice en temps, au point de menacer les gains que conférait à l'origine ce type d'approche. La seule solution pour espérer maintenir ces gains est de disposer d'un mécanisme de sélection qui soit autant que possible automatisé.

Dans cette thèse je propose un mécanisme qui permet de sélectionner, parmi une vaste bibliothèque de composants, le candidat qui répond le mieux à un besoin spécifique, aussi bien sur le plan fonctionnel que non-fonctionnel. L'originalité de cette approche est de permettre une sélection itérative en s'appuyant sur des niveaux de description des besoins de plus en plus détaillés. À cette fin, ce mécanisme intègre des résultats de travaux provenant de domaines variés tels que la recherche de composants, le sous-typage et les métriques de qualité, au sein d'un unique concept : le composant recherché.

Abstract

Component-Based Software Engineering allows to build a system from reusable pre-existing commercial off-the-shelf (COTS) components. The two immediate potential benefits for such an approach are reduced development costs and shorter time-to-market. For this reason, more and more software applications are built using COTS rather than being developed from scratch, as this is something that fewer and fewer companies can afford. However, due to the intrisic nature of COTS as "black-box" units put into markets by third party publishers, we must rethink software development life-cycle in depth. In fact, it becomes impossible to specify requirements without asking if the marketplace provides COTS that can satisfy them. And one cannot specify an architecture without asking if there exist components to integrate it.

In such a context, component selection becomes particularly important. So important that a bad requirements definition associated to a poor selection of components can lead to major financial failure, and sometimes human ones. There are also extra costs due to the investigation of hundreds of candidates disseminated into several different markets and libraries, not to mention the diversity of components' description formats. Finally, this phase can become so time-consuming that it may annihilate the initial promise of costs and time reductions. Therefore, the only solution to maintain these gains is to have a selection process that would be well-defined, repeatable, and as much automated as possible.

In this thesis I propose a mechanism that allows to select, among a vast library of candidates, the one that is the "best" for a specific need, functionally as well as non-functionally. The originality of this approach is to allow an iterative selection by using more and more precise requirements' description levels. To achieve this goal, this mechanism incorporates the results of works from various domains such as component search and retrieval, subtyping and quality metrics, into a unique concept : target component.

Table des matières

Table des figures

Liste des tableaux

Première partie

Prologue

Chapitre 1

Introduction

Sommaire

1.1 Le paradigme composant

En 1968, face à la crise que traversait l'industrie du logiciel, M. McIllroy lança un appel pour le développement massif de composants logiciels réutilisables [184]. Selon lui, *"l'industrie du logiciel repose sur des bases fragiles et l'un des aspects de cette fragilité est l'absence d'une sous-industrie du composant logiciel"*. Depuis cet appel, plusieurs concepts ont vu le jour, tels que ceux de module [78, 76, 220], de type abstrait de données [141, 173], d'objet [243] et plus récemment de composant. L'idée générale de ce nouveau paradigme de développement est de pouvoir construire son application comme on assemble un puzzle ou un Lego [239]. Les pièces de ce puzzle seraient des unités logicielles déjà développées, prêtes à l'emploi et réutilisables [273]. Cette approche rend possible une diminution drastique des délais et des coûts de développement [288]. Cela explique son succès croissant. Dans cette section, nous

allons nous pencher sur la notion de composant logiciel. Nous allons voir en quoi consiste un composant d'un point de vue fonctionnel et non-fonctionnel, et nous allons examiner les différentes étapes d'un cycle de développement à base de composants.

1.1.1 Définition d'un composant

Parmi toutes les définitions des composants logiciels qui ont été proposées, une commence à faire consensus : celle de C. Szyperski [273], selon laquelle *"un composant est une unité de composition qui spécifie par contractualisation ses interfaces, et qui explicite ses dépendances de contexte uniquement. Un composant logiciel peut être déployé indépendamment et est sujet à une composition par de tierces entités"*. La spécification UML 2.0 de l'OMG donne elle aussi une définition du terme composant [215]. Dans cette définition, un composant logiciel est *"une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en terme d'interfaces fournies et requises. En tant que tel, un composant sert comme un type, dont la conformité est définie par ces interfaces fournies et requises (incluant à la fois leur sémantique statique et dynamique)"*.

Dans la première définition, on se concentre sur la composition, les interfaces contractuelles et le déploiement. On se place dans un contexte d'assemblage où ces entités (composants) ont déjà été développées et où l'on voudrait les assembler pour former une application. Les composants spécifient des interfaces sous forme de contrats, et peuvent être déployés séparément. Dans la seconde définition, on met l'accent sur la modularité, la substitution et le typage. Ici, on se place dans un contexte de modélisation architecturale où l'on perçoit le système comme étant un ensemble d'unités modulaires et substituables. Chaque unité (composant) est typée par ses interfaces, qui spécifient les services requis et fournis par ce composant. On retrouve cette notion d'interfaces fournies et requises dans toutes les technologies à base de composants telles que CCM, où l'on parle de facettes et de réceptacles [214] ou Fractal, où l'on parle d'interfaces client et serveur [45].

1.1.2 Structure d'un composant

On a vu qu'un composant, selon la définition communément admise utilisée, est caractérisée par un ensemble d'interfaces fournies et requises. Les interfaces fournies regroupent l'ensemble des services offerts par un composant pour les autres composants. Les interfaces requises, quant à elles, contiennent ce dont un composant a besoin pour fonctionner. Dans les deux cas, une interface de composant contient un ensemble de propriétés fonctionnelles et non-fonctionnelles qui sont visibles depuis l'environnement extérieur de ce composant.

Les propriétés fonctionnelles des interfaces prennent la forme d'opérations dont on donne la signature et parfois la sémantique. La signature d'une opération est de la forme Nom_{Op}=($param_1$, $param_2$, ..., $param_n$) $\rightarrow$ *result*, où Nom_{Op} est le nom de l'opération, ($param_1$, $param_2$, ..., $param_n$) est l'ensemble de ses paramètres ou arguments, et *result* est son résultat. Les aspects sémantiques prennent la forme de prédicats associés aux interfaces et aux opérations. Ces prédicats représentent des spécifications comportementales regroupées sous la forme d'invariants, de pré-conditions et de post-conditions. Les pré-conditions assurent que l'opération

possède certaines propriétés avant d'être exécutée, tandis que les post-conditions garantissent qu'elle en possédera d'autres après son exécution. L'invariant, quant à lui, garantit que certaines propriétés seront respectées avant et après l'exécution.

Si les propriétés fonctionnelles décrivent les aspects syntaxiques et sémantiques des services fournis et requis par un composant, alors les propriétés non-fonctionnelles, ou propriétés de qualité[1] représentent la manière de rendre ou d'attendre ces services. Le standard ISO-8402 [129] définit la qualité comme *"la totalité des caractéristiques d'un produit ou d'un service qui définissent son abilité à satisfaire des besoins établis ou implicites"*. Un sous-ensemble particulier de la qualité logicielle est la qualité de service. Le standard ISO-9309 [130] la définit comme *"un ensemble de qualités liées au comportement collectif d'un ou plusieurs objets"*. D'un point de vue plus général, il s'agit de *"l'effet combiné des performances d'un service qui déterminent le degré de satisfaction d'un utilisateur de ce service"* [56].

En considérant la qualité logicielle dans son ensemble, on appelle caractéristique de qualité toute propriété observable ou mesurable, statique ou dynamique d'un produit ou un processus logiciel. Il peut y avoir plusieurs niveaux de description de ces propriétés. C'est pourquoi une caractéristique de qualité peut être décomposée en sous-caractéristiques, lesquelles peuvent être à leur tour décomposées en attributs, jusqu'à ce qu'on puisse associer des métriques aux attributs qualité de granularité assez fine. Une métrique est une mesure qualitative ou quantitative pour observer ou mesurer la qualité. Un modèle de qualité est une taxonomie des propriétés non-fonctionnelles (caractéristiques, sous-caractéristiques, attributs...). Il est communément admis que c'est l'axe non-fonctionnel qui détermine l'architecture d'un logiciel [18], et que la qualité de cette architecture dépend étroitement de la qualité des composants qui l'intègrent [260].

1.1.3 Développement à base de composants

L'ingénierie logicielle à base de composants ou CBSE (*Component-Based Software Engineering*) est dérivée de l'ingénierie logicielle classique [117]. Cependant, la CBSE se distingue d'une approche non-orientée composants dans le sens où un système logiciel à base de composants est construit à partir de composants pré-existants. On doit donc faire la distinction entre deux cycles de vie : un pour le développement de composants réutilisables (*Design for Reuse*), et un pour le développement d'applications à partir de ces composants réutilisables (*Design by Reuse*). Les différentes phases de ce second processus telles qu'elles ont été identifiées par I. Crnkovic *et al* [66], sont : l'analyse et la spécification des besoins, la conception générale du système logiciel qui va intégrer les composants, la sélection et le tests des composants qui vont être intégrés, l'intégration proprement dite des composants dans le système, la vérification et la validation du système complet, et enfin sa maintenance. Ainsi, la phase d'implémentation, qui accaparait l'essentiel de l'effort de développement dans les approches non-orientées composants, se voit remplacée dans la CBSE par la phase de sélection des composants.

Cependant, face au nombre croissant et à la complexité accrue des logiciels, ainsi qu'à des impératifs de plus en plus forts de mise sur le marché, les entreprises peuvent de moins en moins s'offrir le luxe de construire systématiquement leurs propores composants. Elles sont de plus en plus obligées d'avoir recours à des composants pré-développés et achetés à des tierces

[1] Les termes "non-fonctionnel" et "qualité" sont amalgamés, c'est pourquoi dans cette thèse l'un ou l'autre de ces termes seront employés indifféremment dans la suite de ce mémoire.

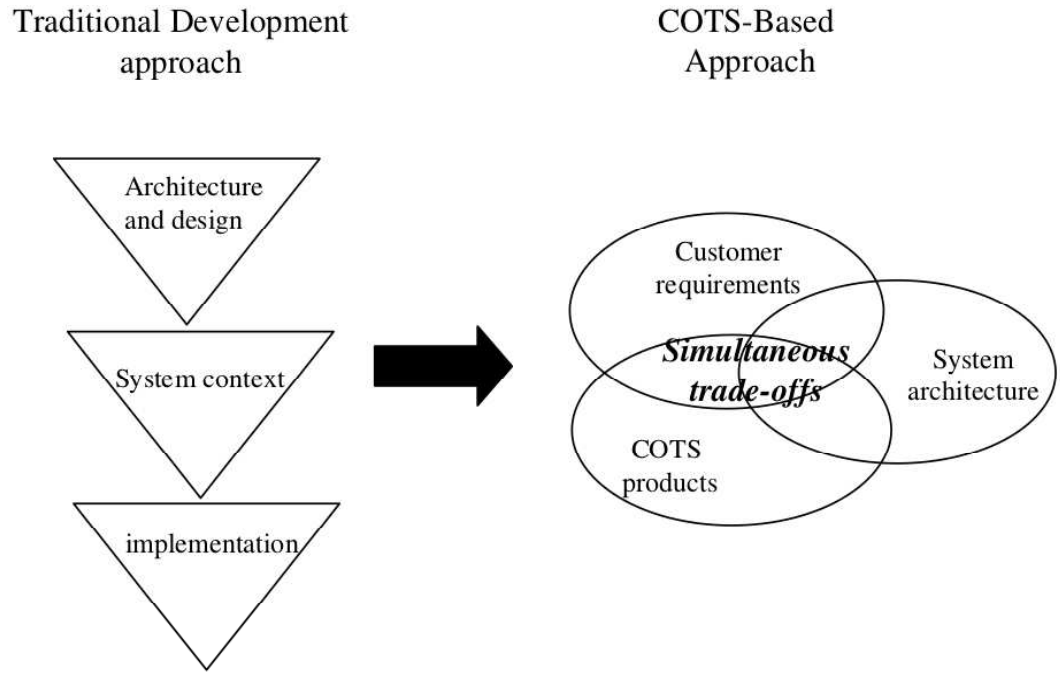

FIG. 1.1 – Comparaison entre approches de développement (d'après [36, 206])

personnes [296]. Le plus souvent, les composants intégrés dans les applications sont donc des "composants sur étagère" [282], aussi appelés COTS (*Commercial-Off-The-Shelf*). Selon la définition donnée par le SEI [36] et faisant consensus dans le domaine de la sélection de composants, un COTS est un logiciel *"vendu ou distribué sous licence au grand public, disponible par le biais de plusieurs copies identiques, utilisé sans modification de sa structure interne (c'est-à-dire son code source), et maintenu à jour par le vendeur"*. De tels COTS sont vendus sur des marchés aux composants [280], tels que ComponentSource [63] ou eCots [190] (on peut citer dans une moindre mesure SourceForge [265], qui est dédié aux "composants" *Open Source*). Ces COTS prennent de plus en plus de place dans le développement d'applications logicielles [283]. Leur nature même impose de repenser profondément le cycle de développement d'un logiciel [282].

En effet, comme le montre la figure 1.1, contrairement à une approche non-orientée composant, qui est strictement séquentielle et dont l'effort se concentre sur l'implémentation, une approche à base de composants de type COTS consiste en une série d'allers-retours entre les différentes phases d'analyse des besoins, de conception de l'architecture globale de l'application, et de sélection des COTS qu'on va intégrer dans cette application [36]. Dans une approche à base de COTS, on ne peut pas poser des besoins sans se demander s'il existe sur le marché des COTS capables de les satisfaire, et on ne peut pas spécifier une architecture sans se demander s'il existe sur le marché des COTS capables de s'y intégrer. Certaines activités propres à la sélection de COTS, comme l'identification des candidats, leur évaluation, la définition des critères d'évaluation, ou l'adaptation des candidats choisis avant leur intégration dans l'appli-

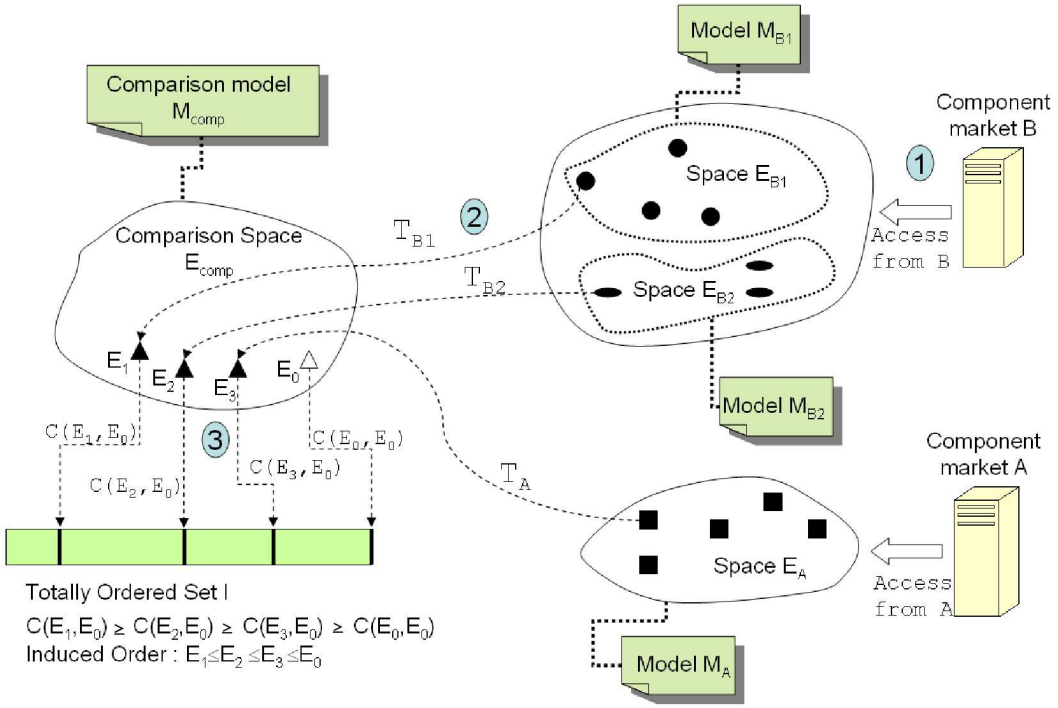

FIG. 1.2 – Problèmes liés à l'identification et la sélection de composants

cation, n'ont pas d'équivalent dans une approche non-orientée composants. Enfin, par rapport à un cycle de développement classique, l'utilisation de COTS entraîne de nouveaux risques et problèmes [103, 287], en particulier celui de ne pas sélectionner et intégrer les bons COTS par rapport à l'architecture choisie [103] et à terme d'avoir une application qui ne satisfait pas les besoins initiaux, en particulier les besoins non-fonctionnels [212]. Un exemple flagrant est le fiasco du Service Ambulancier de Londres (LAS) en 1992. Leur logiciel de délégation automatique d'ambulances est tombé irrémédiablement en panne quelques jours à peine après sa mise en service, causant la mort de dizaines de personnes [69, 90, 217]. Parmi les causes de ce fiasco, on trouve notamment une mauvaise spécification des besoins de l'application [68], ainsi qu'un mauvais processus de sélection des composants nécessaires à ce logiciel [180]. Depuis, c'est devenu un cas d'école qui illustre les conséquences parfois dramatiques des erreurs commises dans la phase de sélection de composants [206].

1.2 La sélection de composants

Nous avons vu que dans une approche à base de composants, une nouvelle phase émerge : la phase de sélection. Nous avons vu que cette phase était d'une importance capitale [66, 83]. Nous allons maintenant l'examiner en détail.

Selon [114], la sélection de composants est un processus qui consiste à déterminer l'adéquation

de composants déjà développés au contexte d'un système particulier. Les processus de sélection existants ont en commun un certain nombre de phases. La sélection se rapproche de l'illustration de la figure 1.2[2]. Les étapes de cette sélection sont : i) la description du besoin de l'application selon un certain format ; ii) l'inspection des marchés et bibliothèques pour extraire les candidats potentiels (étape 1 sur la figure 1.2) ; iii) la traduction des descriptions des candidats dans le même format que celui qui a servi à décrire besoin (étape 2 sur la figure) ; iv) la comparaison des descriptions traduites des candidats avec celle du besoin dans un format commun (étape 3 sur la figure).

Face à l'immensité de l'offre disponible (des milliers de candidats répartis dans de nombreux marchés différents) ainsi qu'à la complexité des applications actuelles, il est crucial d'avoir pour la sélection de composants un processus automatisé, donc bien défini, organisé et systématique, afin d'assurer le succès de cette phase de sélection [31, 35, 288, 296]. Cependant, l'automatisation de la sélection devient problématique si elle est mal réalisée. Nous allons adresser successivement les problèmes posés par chacune des étapes qui constituent cette phase de sélection : la description du besoin, l'inspection des bibliothèques, la traduction des descriptions, l'opérateur de comparaison, et les propriétés à comparer.

1.2.1 Description du besoin

Un préalable à toute sélection de composant est de préciser ce que l'on souhaite trouver [168]. Il faut par conséquent décrire le besoin. Le problème qui se pose alors est celui du format de description. Dans les processus de sélection de composants, les besoins sont le plus souvent exprimés de manière informelle (en langage naturel, par exemple) et traduits en critères d'évaluation mesurables par des métriques. Comme le code source des COTS est généralement inaccessible, l'utilisateur a souvent recours à des critères abstraits pour décrire les propriétés externes souhaitées. Il y a cependant un risque de s'en tenir à des critères trop vagues, éloignés des besoins réels de l'application, ce qui peut entraîner des différences d'interprétation de la part des évaluateurs des candidats [152], et, à terme, coûter très cher [41].

Dans les processus de recherche de composants en bibliothèque, le besoin est exprimé sous forme de requête, décrite dans un format spécifique grâce à un "encodeur" [192]. Il s'agit alors d'identifier les candidats dont la description correspond à une requête particulière. Or, une telle phase d'identification est prévue par certains processus de sélection des composants, ce qui permet l'évaluation d'un nombre limité de candidats pertinents. Cette fois, le format de description du besoin et des candidats est plus précis. Le plus souvent, il correspond à une certaine représentation des composants (par exemple, des mots-clés [182] ou des signatures d'opération [301]).

Le problème majeur est qu'à ce jour il n'existe pas de format de documentation de composants faisant consensus. Chaque bibliothèque et marché aux composants use donc de son propre format aussi bien pour documenter les propriétés fonctionnelles (opérations fournies et requises) que non-fonctionnelles (niveau de fiabilité, sécurité, prix, origine...). Les mêmes in-

[2]Cette figure ressemble à la figure 3.1, qui illustre la problématique de la recherche de composants décrite par H. Mili *et al* dans [192]. La principale différence est qu'ici, le but n'est pas seulement d'identifier les candidats d'une bibliothèque particulière qui correspondent à une requête, mais de sélectionner le meilleur d'entre eux depuis plusieurs bibliothèques différentes selon plusieurs critères.

formations peuvent être fournis par deux bibliothèques différentes dans deux formats différents, ou qu'une bibliothèque peut fournir des informations qu'on ne retrouvera pas dans une autre bibliothèque, ou encore qu'au sein d'une même bibliothèque peuvent coexister plusieurs formats différents. À titre d'exemple, un marché comme *ComponentSource* [63] va prévoir pour chaque composant plusieurs pages Web, une par type d'information (c'est-à-dire une page pour le prix, une autre pour la compatibilité, etc...). Un site *Open Source* comme *SourceForge* [265] va pour sa part laisser aux contributeurs le soin d'apporter toutes les informations nécessaires dans des fichiers dédiés à la documentation. Les pages Web que *ComponentSource* dédie à ses composants contiennent généralement beaucoup plus d'informations que celles de *SourceForge*. Enfin, sur *ComponentSource*, la documentation des composants fournis par *Elegant Microweb* consiste en des fichiers *Javadoc*, tandis que celle fournie par */n software* consiste en un fichier ".hlp", sans parler des différentes manières de documenter des composants de type ActiveX et des composants de type JavaBeans. Cette hétérogénéité est liée suivant les cas : i) à la présence au sein d'un même marché de composants développées dans des technologies différentes qui supposent donc la mise à disposition d'informations propres à l'une ou l'autre de ces technologies ; ii) au caractère laxiste du marché qui autorise l'enregistrement de composants sans pour autant contraindre la complétude ou le format des informations fournies. C'est le cas, on l'a vu, du marché *ComponentSource* [63], dans lequel plusieurs modèles de documentation sont présents en fonction des technologies (ActiveX, JavaBeans, .NET...) et des exigences documentaires qu'ont souhaitées respecter les concepteurs de ces composants.

L'objectif étant de pouvoir comparer le besoin d'une application avec des composants documentés selon des formats potentiellement variés et très différents, il est impératif d'user pour sa description d'un format suffisamment riche et flexible. Ceux existants pour les composants commerciaux (ActiveX, JavaBeans, .NET...) dépendant fortement de technologies qui leurs sont propres, il n'est pas envisageable d'user de l'un d'entre eux pour ce faire. Il n'y a pas d'autre solution que de proposer un modèle plus abstrait pour atteindre le niveau de généricité requis. Dans cette hypothèse, il faut toutefois veiller à ce qu'il garde une certaine proximité aussi bien syntaxique que sémantique avec les formats concrets existants. En effet, que ce soit pour l'identification préliminaire des candidats ou pour l'évaluation de ceux qui sont pertinents, leur comparaison avec le besoin doit se faire au sein d'un format de description commun, et pour des raisons d'efficacité, il ne suffit donc pas de se contenter de simples mots en guise de critères d'évaluation [206]. Il y a donc là un vrai défi, celui de la définition d'un format de description du besoin qui soit suffisamment abstrait pour embarquer un maximum d'informations présentes dans tout format de composant existant, mais ne nécessitant pas pour autant l'emploi de mécanismes de traduction trop complexes. C'est le cas du format de description du besoin défini dans cette thèse, qui repose sur la notion de composant recherché. Dans la suite de cette section nous désignerons l'ensemble de toutes les descriptions que ce format peut engendrer par le terme "espace de comparaison", et nous le noterons $Ecomp$.

Problème : Il faut un format de description du besoin capable d'englober les concepts qu'on retrouve dans la plupart des formats de composants existants. Or, à ce jour il n'existe pas de format faisant consensus.

1.2.2 Inspection des marchés et bibliothèques

Il est facile de constater rien qu'en se limitant aux marchés aux composants disponibles sur internet que l'accès à leur documentation se fait selon des modes très différents. Dans certains marchés, les composants sont groupés par thème de manière aborescente et totalement propriétaire. C'est le cas de *ComponentSource*, qui classe ses composants selon leur domaine d'application : image, programmation, communication internet... D'autres marchés, au contraire, livrent les composants tels quels sans classement particulier. On peut supposer que certains marchés offriront à terme des mécanismes de recherche plus ou moins complets pour faciliter la découverte de leurs composants. Bien que les processus de recherche prévoient *a priori* qu'un même encodeur puisse à la fois formuler les requêtes dans un format particulier et "indexer" les composants [192], c'est-à-dire les traduire dans le même format que celui des requêtes (voir figure 3.1), cela ne concerne qu'une seule bibliothèque à la fois, et il y a peu d'espoir qu'à terme un protocole standard de consultation appliqué par tous n'émerge. Il est donc nécessaire, si l'on souhaite automatiser un processus capable d'attaquer plusieurs marchés, de prévoir pour chacun d'entre eux une procédure d'indexation spécifique (notée *Am* sur la figure 1.2), qui devra être capable d'extraire depuis un marché ou une bibliothèque l'intégralité des descriptions des composants candidats. Appliquée à *ComponentSource*, une telle procédure serait capable de "parser" les pages Web de chaque composant candidat afin d'en extraire les informations correspondantes (pour la page consacrée à la compatibilité, ce serait par exemple les systèmes d'exploitation compatibles, les applications possibles pour le déploiement...). Éventuellement cette procédure pourra user d'un premier filtre s'appuyant sur les mécanismes existants de recherche de composants, afin de n'extraire que les seuls candidats répondant à une certaine requête. Toute la difficulté réside dans le fait que ces procédures ne peuvent être écrites qu'à la main et au cas par cas. Il serait cependant envisageable à terme de faciliter l'écriture de ces procédures si certaines d'entre elles offraient une modélisation rigoureuse de leur protocole d'accès dans un format exploitable par des outils développés dans ce but. En l'état cette problématique est ouverte et pose des problèmes autant technologiques que méthodologiques. Nous ne l'aborderons pas dans cette thèse.

> **Problème :** Il faut être capable d'indexer différents marchés et bibliothèques pour extraire les descriptions de leurs composants. Or, il n'existe pas de consensus sur l'accès aux composants de ces bibliothèques, ni sur la méthode d'indexation.

1.2.3 Traduction des descriptions

Les comparaisons prendront place dans l'espace de comparaison (noté $Ecomp$ sur la figure 1.2). Dans les processus de recherche de composants, cela correspond à l'union entre "l'espace des requêtes", qui regroupe l'ensemble des requêtes formulées par l'encodeur dans un certain format, et "l'espace des codes", qui regroupe l'ensemble des composants traduits par l'encodeur dans le même format [192] (voir figure 3.1). Il est donc nécessaire de projeter dans cet espace commun de comparaison l'intégralité des descriptions de composants candidats obtenues après parcours des marchés et bibilothèques. Dans les processus de recherche,

la phase d'indexation effectuée par l'encodeur s'occupe à la fois de l'extraction d'informations (abordée dans le paragraphe précédent) et de la traduction dans un format commun [192], mais cela ne marche que pour une seule bibliothèque de composants issus en général du même format. Or, si on se place dans le contexte de marchés multiples et de formats multiples, les descriptions des composants sont éléments de plusieurs espaces de composants différents. On doit donc disposer pour chaque format de composant concret d'un mécanisme intermédiaire capable de traduire de façon adéquate toute description exprimée dans un format A vers une description conforme au modèle de comparaison (voir figure 1.2). Formellement, cette traduction est une fonction que nous notons TA, depuis l'espace de description associé au modèle A (noté EA) dans l'espace de comparaison $Ecomp : EA \rightarrow Ecomp$. Deux descriptions différentes de l'espace de A peuvent être projetées sur la même description de l'espace de comparaison, et à l'inverse certaines descriptions de l'espace de comparaison ne sont pas atteignables depuis l'espace de A. Ces pertes sont liées respectivement au manque de puissance du modèle de comparaison qui peut omettre certaines informations existantes dans le modèle de A, et à l'inverse à une puissance supérieure du modèle de comparaison par au rapport au modèle A.

D'un point de vue pratique, il est nécessaire de proposer autant de fonctions de traduction qu'il existe de formats de description dans les bibliothèques et marchés existants. C'est un travail important qui ne saurait prétendre à l'exhaustivité, et qui est d'autant plus délicat que les formats en question ne sont que rarement explicités et encore moins formalisés. Dans le cas de *ComponentSource* [63], mis à part les pages Web qui contiennent un certain nombre d'informations, la documentation proposée par le fournisseur est très inégale d'un composant à l'autre, certains d'entre eux n'en possédant même pas. Comme il est illusoire d'automatiser la reconstruction de ces formats en partant de la description des composants, on doit se résoudre en l'état à écrire ces traductions à la main et au cas par cas. Ce problème n'est pas non plus adressé dans cette thèse. De toute manière, une automatisation complète et exhaustive de toutes les traductions n'est envisageable qu'à condition de disposer d'un format universel respecté par tous. Mais en l'état des marchés et bibliothèques, on est encore loin du compte.

Nous considèrerons par conséquent que les traductions sont déjà faites et qu'on peut passer directement à l'étape suivante, celle où il faut identifier les candidats et les comparer à la description du besoin sur une base commune.

> **Problème :** Il faut être capable de traduire toutes les descriptions de composants issus de différentes bibliothèques vers des description conformes à un format commun. Or, il faudrait autant de fonctions de traduction qu'il existe de formats dans les bibliothèques.

1.2.4 Propriétés requises pour un opérateur de comparaison

Une fois que l'on a documenté le besoin dans un format *ad hoc*, parcouru convenablement chaque marché et bibliothèque pour obtenir l'ensemble des descriptions des composants candidats, et traduit toutes ces descriptions dans un même format, le problème consiste à proposer un opérateur de comparaison. Dans le domaine de la sélection de composants, on utilise en général des techniques de prise de décision multi-critères pour évaluer et classer les candidats.

Celui qui posséde le "score d'évaluation" le plus élevé est considéré comme le meilleur. Dans le domaine de la recherche de composants, un *matcher* se charge de comparer les descriptions candidates de l'espace des codes avec chaque description de requête issue de l'espace des requêtes [192] (voir figure 3.1). Cependant il ne s'agit pas ici de sélectionner le meilleur composant candidat selon plusieurs critères, mais de retrouver tous ceux dont les descriptions correspondent à la requête.

Intéressons nous tout d'abord aux propriétés que doit posséder un tel opérateur que nous désignerons dans la suite par C. Cet opérateur de comparaison est une fonction à deux opérandes $C(E1,E0)$: $E1$ est la description d'un composant candidat et $E0$ est la description du besoin. Cette fonction d'arité 2 a pour ensemble de définition le produit cartésien de l'espace de comparaison avec lui même (voir figure 1.2). Il faut noter que ce domaine est le produit cartésien dans son intégralité. On souhaîte pouvoir en effet comparer tout couple de description. De plus, à l'aide du résultat rendu par C on doit pouvoir classer tous les composants candidats. Mathématiquement parlant cela suppose que l'on puisse comparer tout couple de résultat, donc que l'ensemble d'arrivée de C soit totalement ordonné. Au final l'opérateur de comparaison est une fonction dont la signature est : $Ecomp \times Ecomp \rightarrow I$. I désignant un ensemble totalement ordonné, par exemple un intervalle de l'ensemble IR.

Maintenant il convient de s'intéresser à la relation d'ordre que l'on cherche à induire à l'aide de C sur $Ecomp$ en partant de l'ordre total défini sur I. Cet ordre induit est formellement défini par : $E1 \leq E2$ si et seulement si $C(E1,E0) \leq C(E2,E0)$. Il n'y pas une mais plusieurs relations d'ordre induites par C : une par besoin, ou plutôt par description de besoin $E0$. Une relation d'ordre sur $Ecomp$ associée à C pour un besoin $E0$ doit au minimum être capable de respecter la relation d'ordre classique de sous-typage issue du principe de substitution : toute description $E1$ peut être considérée comme sous-type de $E0$ selon C si et seulement si $C(E1,E0) = C(E0,E0)$. Pour les descriptions sous-types, un classement ne présente donc aucun intérêt puisqu'elles répondent toutes parfaitement au besoin exprimé. Elles ne peuvent être départagées par l'opérateur C (classés deux à deux) à moins de renforcer les exigences présentes dans la description du besoin. L'opérateur C donne pour tous les sous-types d'une description de besoin $E0$ la même valeur égale à $C(E0,E0)$. On peut décider sans perte de généralité que cette valeur est une constante pour toute description de besoin EO (par exemple égale à 100% pour manifester une couverture totale du besoin). De plus, il est logique de considérer que cette constante est le plus grand élement de I puisqu'il ne peut y avoir mieux que $E0$ pour se substituer à $E0$.

Cette première propriété ne définit encore que très partiellement la sémantique de C. Il reste encore à définir la sémantique que nous voulons associer à $E1 \leq E0$ ($E0 \leq E1$ n'est pas possible car $C(EO,EO)$ est de par la remarque précédente le "meilleur" élément de I) pour une description $E0$ et une description candidate $E1$. Cette sémantique consiste à déterminer pour les descriptions $E1$ qui ne sont pas sous-types de $E0$ le sens de $C(E1,E0)$; plus précisémment le sens que nous donnons à la différence strictement positive $C(E0,E0) - C(E1,E0)$ manifestant "l'écart" existant entre le besoin et tout composant candidat qui n'est pas sous-type selon C. Notre objectif est de sélectionner parmi tous les composants candidats celui représenté par la description $Ebest$ qui répond le "mieux" au besoin décrit par $E0$ (on retrouve une préoccupation similaire dans le monde de la recherche de composants, plus précisément dans la mesure de distance fonctionnelle entre spécifications relationnelles [139]). Ce composant est formellement

défini comme vérifiant : pour tout $E1$ candidat de l'espace de comparaison, $E1 \leq Ebest$. Cependant, cette notion de "mieux" contient une ambiguïté qui rejaillit sur la sémantique donnée à C. On peut avoir, par exemple, un candidat qui possède la plupart des éléments importants exigés par $E0$, mais qu'on ne peut pas adapter pour obtenir les derniers éléments restants. Et à l'inverse, on peut avoir un candidat auquel il manque de nombreux éléments importants, mais qu'on peut facilement adapter afin d'ajouter rapidement ces éléments. Entre les deux choix, lequel faut-il faire et quel type d'écart C doit-il mesurer ? Il y a au moins deux façons d'aborder cet écart, soit sous l'angle de la satisfaction fonctionnelle et non-fonctionnelle, soit sous l'angle de l'effort à produire pour éventuellement intégrer un composant qui ne satisfait pas pleinement le besoin. Cette notion d'effort est cruciale dans n'importe quel logiciel car elle permet d'anticiper les problèmes futurs. C'est particulièrement vrai pour les composants COTS, dont le code source est inaccessible, et qui sont fournis par une tierce personne. Le fait de dépendre du travail d'un autre décuple les risques liés à l'intégration de ces COTS [295].

La mesure de satisfaction est de toute façon un prérequis de la mesure d'effort. Il faut d'abord identifier les candidats pertinents, puis évaluer leurs manques, avant de tenter d'estimer les coûts d'intégration qu'ils impliquent. De cette complémentarité se dégage la nécéssité de ne pas se limiter à une seule sémantique de comparaison pour C. Il est également nécessaire de ne pas se limiter à un seul niveau de granularité dans la description. En effet, certaines techniques de recherche de composants telles que la recherche par mots-clés sur la documentation [182] peuvent être utiles pour réduire le nombre de composants candidats, mais inappropriées pour une analyse détaillée des propriétés des candidats restants. À l'inverse, d'autres techniques comme le matching de signature [301] peuvent se révéler efficaces pour une analyse précise d'un petit nombre de candidats, mais trop coûteuses face à une centaine de composants qui ne sont pas tous du domaine souhaité. Enfin, le besoin d'intégrer un composant devant être décrit selon plusieurs critères, il est nécessaire de se poser la question de l'importance et de la hiérarchisation de ces critères. En effet, il se peut que l'absence d'un élément particulier dans un candidat soit pardonnable si cela ne coûte pas trop de le rajouter par ailleurs (voire de passer outre), tandis que l'absence d'un autre élément qu'on ne peut pas ajouter ultérieurement vaudra le rejet du candidat. C'est pour cette raison que C doit être doté d'un mécanisme de pondération.

En résumé, l'opérateur de comparaison C doit pouvoir calculer l'écart entre chaque description d'un candidat et la description d'un besoin selon plusieurs sémantiques de comparaison, plusieurs niveaux de description, et plusieurs pondérations. D'où la nécessité d'avoir à terme une véritable "stratégie de sélection" capable de combiner tous ces éléments de manière cohérente, afin de sélectionner progressivement les candiats les plus pertinents jusqu'à choisir le "meilleur" selon un besoin particulier.

Problème : Il faut un opérateur de comparaison capable de déterminer, parmi deux composants décrits dans un même format, lequel est le "meilleur" selon certains critères. Il faut également que cet opérateur tienne compte de la sémantique de comparaison (le "meilleur des deux", par rapport à quoi ?), et de la hiérarchisation des critères.

1.2.5 Propriétés fonctionnelles et non fonctionnelles exprimées dans les descriptions

Dans la sous-section précédente, nous nous sommes intéressés d'un point de vue théorique aux propriétés souhaitables pour l'opérateur de comparaison C. Intéressons-nous maintenant d'un point de vue pratique aux problèmes qui se posent lorsqu'on veut utiliser cet opérateur pour mesurer l'écart entre deux descriptions. Il est clair qu'il faudra être capable de déterminer si un service fonctionnel ou non fonctionnel exigé dans la description du besoin trouve un équivalent ou une solution proche dans le candidat. Examinons successivement le cas d'une propriété fonctionnelle puis ensuite celui d'une propriété non fonctionnelle.

Si l'on ne se limite qu'à un service fonctionnel, force est de constater que deux opérations peuvent avoir deux signatures différentes tout en représentant le même service. Considérons une opération qui traite du téléchargement de fichiers. Dans la section "Communication internet" de *ComponentSource* [63], plusieurs composants traitant du protocole FTP (*File Transfer Protocol*) proposent cette opération. Un exemple est le composant FTP de *ComponentSpace*, qui fournit une opération appelée *Download* et dont la signature est : $String \times String \rightarrow String$. Un autre exemple est le composant FTP de la suite logicielle *Ip*Works !*, qui fournit une opération elle aussi appelée *Download*, mais dont la signature est : $Void \rightarrow Void$. D'autres exemples sont le *Catalyst File Transfer Control* et le le composant FTP de *Xceed*, qui fournissent respectivement une opération $GetFile : String \times String \times Long \rightarrow Long$ et $ReceiveFile : String \times String \times String \rightarrow Void$.

Plus encore, les mêmes services peuvent être organisés différemment d'un composant à un autre. Un composant peut fournir une interface unique dans laquelle il y aura toutes les opérations, tandis qu'un autre peut fournir les mêmes opérations, mais réparties entre plusieurs interfaces. Pour reprendre l'exemple des composants de *ComponentSource*, le composant FTP de *ComponentSpace* fournit plusieurs opérations réparties dans plusieurs interfaces différentes. Par exemple, les opérations de connexion et de déconnexion sont situées dans l'interface *FTPClient*, tandis que les opérations de téléchargement sont situées dans l'interface *RemoteFile* et celles consacrées à la création et destruction de dossiers se trouvent dans l'interface *RemoteDirectory*. À l'opposé, le composant FTP de la suite logicielle *IP*Works !* fournit toutes ses opérations traitant de FTP dans une seule et même interface.

Par ailleurs, un service donné pourrait très bien être réalisé non par une seule opération mais par plusieurs opérations de granularité plus fine, ou au contraire par une opération plus générique via un paramétrage adéquat. Le composant FTP de la suite *IP*Works !* fournit une seule opération pour le téléchargement, tandis que la version .NET du composant FTP de *PowerTCP* en fournit plusieurs, selon qu'on télécharge un fichier ou plus.

On le voit le calcul d'un écart sur le plan fonctionnel est à lui seul un vrai défi. Une piste possible serait d'user d'outils complexes et coûteux tentant de mettre en œuvre des matchings entre spécifications comportementales [302] ou relationnelles [139, 191] ou des concepts tels que la composition et la comparaison de machines à états, de contrats ou de graphes. Une réponse moins ambitieuse serait de tenter une simple comparaison de signatures telles que le font les systèmes de type dans le monde des composants. Dans tous les cas et avec les moyens actuels, on ne peut qu'espérer trouver un écart qui ne soit pas trop loin de l'écart réel.

Le calcul d'un écart entre deux propriétés non-fonctionnelles est un problème tout aussi

complexe. Contrairement aux propriétés fonctionnelles, souvent décrites en listant les signatures des opérations dans la documentation d'un composant, les propriétés non-fonctionnelles du composant (comme la performance, la fiabilité, la sécurité...) sont difficiles à décrire et donc rarement documentées. De toute façon, il n'y a de standard communément admis pour leur description. Même si l'on trouve des standards limités à certaines catégories de propriétés non fonctionnelles (par exemple, le modèle ISO-9126 [132] pour la qualité), ces standards n'imposent pas la manière de quantifier à l'aide de métriques les propriétés référencées. Il est par conséquent difficile de comparer des propriétés qui ne sont pas exprimées et quantifiées avec les mêmes métriques. Malheureusement, l'axe non-fonctionnel d'une application ne peut pas être omis. Toute application doit offrir un certain niveau de qualité. Il est donc impensable de choisir un composant en faisant fi de ses propriétés non-fonctionnelles, pour ensuite espérer qu'il remplisse les exigences de qualité requises par l'application dans laquelle il sera intégré. C'est d'autant plus vrai pour les COTS, où certaines propriétés non-fonctionnelles comme la sûreté sont des facteurs décisifs dans la sélection [296]. Certaines études considèrent même que les problèmes engendrés par le non-respect des exigences non-fonctionnelles sont les plus coûteux et les plus difficiles à corriger [41, 67, 68, 71].

Problème : Il faut être capable de décrire les propriétés fonctionnelles et non-fonctionnelles des composants, afin de les englober dans une même comparaison. Or, un même service peut être décrit de plusieurs manières, et les propriétés non-fonctionnelles sont difficiles à décrire.

1.3 Travail présenté dans cette thèse

1.3.1 Les problèmes abordés

Dans le développement d'une application à base de composants, la phase de sélection, bien qu'étant d'une importance capitale, comporte du point de vue de son automatisation de nombreux verrous conceptuels et technologiques : i) la difficulté à trouver un format de description du besoin capable d'englober la plupart des concepts communs aux modèles de composants existants ; ii) le problème de l'inspection des marchés et bibliothèques de composants, afin de trouver les candidats pertinents et d'extraire leurs descriptions ; iii) le problème de la traduction des descriptions de composants de différentes origines vers un format commun ; iv) la difficulté à définir un opérateur de comparaison qui prenne en compte les écarts entre composants et qui donne un sens à cet écart ; v) la difficulté à décrire certaines propriétés, en particulier non-fonctionnelles. Dans cette thèse, nous aborderons les problèmes i, iv et v. En ce qui concerne les problèmes ii et iii, nous supposerons que l'inspection des bibliothèques, l'extraction des candidats et leur traduction dans un format commun, auront déjà été exécutées.

1.3.2 Organisation du mémoire

Ce chapitre introductif vient de présenter la problématique de recherche qui a été traitée dans cette thèse. Le reste du mémoire est organisé comme suit :

Partie II : Cette partie présente un état de l'art des travaux en relation avec le sujet de cette thèse. On trouve tout d'abord les travaux sur la sélection de composants. Bien que ce terme fasse référence à un domaine de recherche précis (présenté dans le premier chapitre de cette partie) qui se consacre à la description du besoin par une approche mutlicritères, le problème de la sélection englobe en réalité d'autres domaines qui concernent davantage les méthodes de recherche et de comparaison des candidats. Ces domaines sont présentés dans le second et le troisième chapitres de cette partie.

Chapitre 2 : Ce chapitre présente un aperçu des processus de sélection de composants. Les phases principales de ces processus sont la définition de critères d'évaluation, la hiérarchisation de ces critères et l'évaluation des candidats en fonction de ces critères. Certains processus innovent en proposant des phases supplémentaires, tandis que d'autres innovent dans la définition les critères d'évaluation.

Chapitre 3 : Ce chapitre se concentre sur les techniques et les processus de recherche de composants en bibliothèque. Selon la description de la requête, qui sert aussi à représenter les composants candidats, plusieurs techniques différentes ont été proposées, qui incluent la recherche par mots-clés, la classification par facettes, le matching de signature ou le matching de spécification. Certains processus proposent même la création de bibliothèques intermédiaires, dans lesquelles les descriptions de composants sont déposées et hiérarchisées pour de futures recherches.

Chapitre 4 : Ce chapitre se concentre davantage sur les techniques de comparaison fonctionnelle et non-fonctionnelle des composants. D'un point de vue fonctionnel, les techniques de substitution adaptées au monde composant sont présentées. D'un point de vue non-fonctionnel, l'accent est mis sur les contrats de qualité de service et sur les modèles de qualité adaptés aux composants.

Partie III : C'est dans cette partie qu'est présentée une approche pour la sélection de composants logiciels, qui constitue la contribution de cette thèse. Cette approche comporte : un premier chapitre sur la description et la comparaison de composants, un deuxième chapitre sur la manière de comparer, et un dernier chapitre sur la validation de cette approche.

Chapitre 5 : Ce chapitre présente des outils conceptuels pour la description et la comparaison automatique de composants. Le concept de composant recherché est introduit. Son premier aspect est de modéliser le besoin sous forme de composant virtuel, afin qu'il puisse être comparé aux composants "concrets". Son deuxième aspect est de traiter les manques et les insuffisances de ces candidats concrets par rapport au composant recherché. Le premier aspect de ce concept est traité au moyen d'un format de description qui regroupe les propriétés fonctionnelles et non-fonctionnelles des composants. Le deuxième aspect de ce concept est traité au moyen de la notion de pondération. Ce format et cette notion sont intégrés dans un mécanisme de comparaison hiérarchique et multi-niveaux. Ce mécanisme, grâce à des techniques issues de divers domaines connexes, adapte la comparaison au niveau de granularité souhaité, et permet l'automatisation de cette comparaison.

Chapitre 6 : Ce chapitre se consacre aux manières d'utiliser les outils conceptuels présentés dans le chapitre précédent afin de sélectionner le "meilleur" candidat d'une bi-

bliothèque. Nous introduisons un indice de satisfaction avec sa pondération associée. Cet indice adapte le mécanisme hiérarchique pour déterminer ce qu'un composant candidat possède en commun avec le composant recherché. Un autre mode de comparaison, qui intervient après la satisfaction, est l'effort d'adaptation des candidats. Il s'agit de savoir ce que cela va coûter d'adapter un candidat afin que celui-ci corresponde complètement au composant recherché. La satisfaction et l'effort, ainsi que d'autres méthodes, peuvent être combinés dans une stratégie de sélection cohérente. Ces stratégies peuvent être elles-mêmes incluses dans un processus de sélection itératif et systématique.

Chapitre 7 : Ce chapitre présente l'outil qui implémente l'approche proposée dans ce travail de thèse. Ensuite, deux exemples concrets sont présenté sur la sélection d'un composant à partir d'un même marché aux composants existant et sur la base d'un besoin précis. Chaque exemple exécute le processus de sélection défini dans le chapitre précédent selon l'une des stratégies présentées dans le chapitre précédent.

Partie IV : Cette dernière partie conclut ce mémoire.

Chapitre 8 : Dans ce chapitre, les principaux apports de travail de thèse sont présentés, ainsi que ses limites, et des pistes pour le prolongement de ce travail. C'est ce chapitre qui clôt le présent mémoire.

Deuxième partie

État de l’art

Dans cette partie, un état de l'art des différents travaux connexes est présenté. Le chapitre 2 aborde les processus de sélection des composants, ainsi que les techniques de prise de décision multi-critères qui sont utilisées par ces processus. Le chapitre 3 passe en revue les différentes techniques de recherche de composants en bibliothèque, qui pourraient être utilisées pour identifier les candidats à évaluer. L'évaluation détaillée de ces candidats, quant à elle, pourrait être réalisée au moyen des techniques de comparaison fonctionnelle et non-fonctionnelle présentées dans le chapitre 4.

Chapitre 2

Techniques et processus de sélection de composants

Sommaire

2.1 Introduction

Nous avons vu dans le chapitre 1 que la sélection de composants est une phase cruciale, considérée comme l'un des facteurs principaux de succès pour une entreprise [83]. La plupart des processus de sélection de composants ont en commun deux phases principales : i) la définition de critères d'évaluation depuis les besoins de l'application dans lequel le candidat sera intégré ; ii) l'évaluation et le classement des logiciels candidats sur la base de ces critères hiérarchisés [158, 83]. Les processus les plus avancés proposent en supplément des phases de hiérarchisation de ces critères [83] et d'identification des candidats potentiels [152, 158, 205].

Selon le standard IEEE-1209 [124], les critères d'évaluation sont les paramètres sur la base desquels les composants sont évalués et les décisions sont prises pour leur sélection. Il est communément admis que l'évaluation des composants doit être basée sur les besoins de

l'application [152, 23]. La description de ces se fait généralement au moyen d'entretiens avec l'utilisateur, de consultation de la documentation de l'application, d'études de marché, et de connaissance de l'existant [264]. Les critères d'évaluation sont des interprétations de ces besoins.

L'évaluation des candidats se faisant sur la base de plusieurs critères, on utilise en général dans la phase correspondante des techniques du domaine de la "prise de décision multi-critères" (Multi-Criteria Decision Making ou MCDM). Ces techniques font l'objet de la section 2.2. Ensuite, dans la section 2.3 nous présentons un état de l'art sur les principaux processus de sélection de composants. Nous commencerons par OTSO et PORE, deux des travaux les plus cités, avant de détailler les autres processus, classés selon les innovations qu'ils apportent.

2.2 Les techniques de prise de décision multi-critères

Le problème de la prise de décision multi-critères, qui consiste à trouver le meilleur candidat (ou à classer les candidats disponibles) selon plusieurs critères différents, a été introduit par Koopmans [154] ainsi que par Kuhn et Tucker [157]. Depuis, la prise de décision multi-critères s'est développée en tant que domaine de recherche appliqué à de nombreux autres domaines, comme par exemple l'intelligence artificielle [80]. Parmi les applications de ces techniques, on trouve la sélection de composants. En effet, la finalité d'un processus de sélection est de choisir le meilleur candidat pour son application. Ce choix est le résultat d'une décision, qui est prise après avoir évalué tous les candidats en fonction des mêmes critères. Cette phase doit donc être réalisée au moyen d'une technique capable d'évaluer et de classer les candidats selon plusieurs critères d'évaluation préalablement définis, en accordant éventuellement plus d'importance à certains d'entre eux.

La figure 2.1 montre un exemple de problème de type MCDM. L'objectif initial (*Goal*) est de choisir le meilleur produit parmi trois candidats A, B et C, aussi appelés *Alternatives*. Le problème est d'évaluer chaque candidat en fonction des critères choisis : *Cost*, *Usability*, *Supplier capability*, et *functionality*. Pour évaluer les candidats, il faut résoudre deux difficultés. La première concerne l'évaluation individuelle des candidats pour chaque critère. La deuxième concerne l'agrégation de ces "scores" individuels afin d'obtenir l'évaluation globale des candidats. Cette agrégation pose également le problème de l'importance des critères les uns par rapport aux autres.

Plusieurs techniques ont été proposées afin de résoudre ces problèmes. Dans cette section, nous présenterons certaines de ces techniques parmi les plus connues et les plus utilisées dans le domaine qui nous intéresse, à savoir la sélection de composants.

2.2.1 MCDA (Multi-Criteria Decision Aid)

L'aide à la décision multi-critères MCDA [261, 245, 246, 286] consiste : i) à identifier les critères qu'un composant devrait remplir ; ii) à assigner à chacun de ces critères un nombre réel qui représente son poids, c'est-à-dire son importance dans la décision ; iii) à assigner aux candidats des valeurs réelles indiquant leur score de satisfaction de ces critères (il y a un score

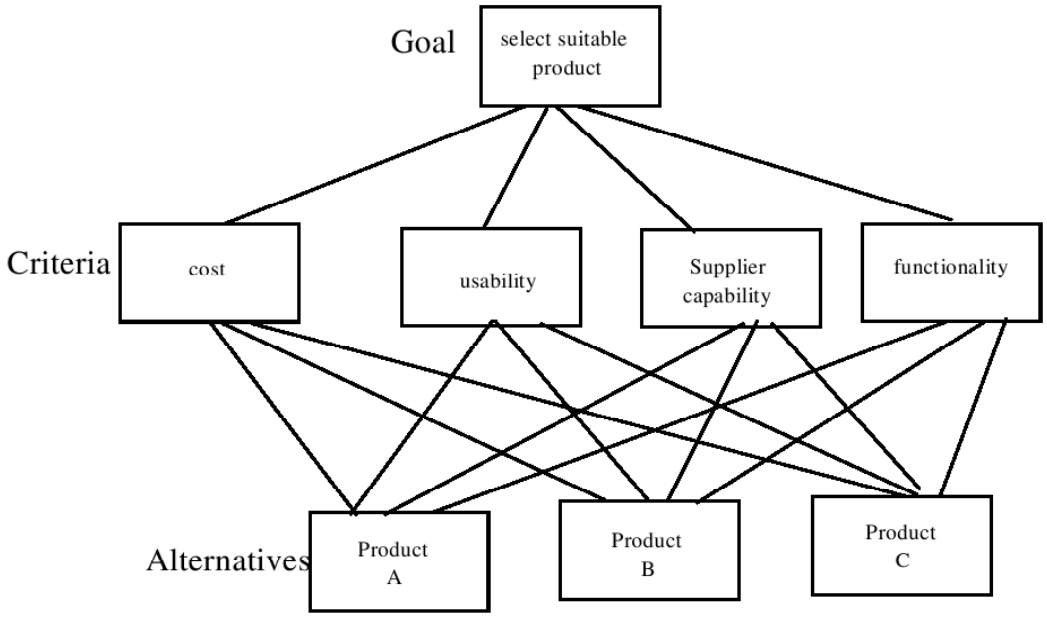

FIG. 2.1 – Exemple de problème de prise de décision multi-critères (d'après [206])

par critère pour chaque candidat, le score maximal étant égal au poids du critère) ; iv) à additionner ces scores pour chaque candidat afin d'obtenir son score total. Le meilleur candidat est celui qui a le score le plus élevé. On retrouve les phases de définition et de hiérarchisation des critères d'évaluation communes aux processus de sélection de composants. L'évaluation des candidats, quant à elle, se fait d'abord localement pour chaque critère, puis globalement en agrégeant les scores locaux.

Le tableau 2.1 montre un exemple d'application de la technique MCDA à la sélection de composants. L'utilisateur a défini 6 critères pour l'évaluation des candidats (première colonne sur le tableau). Il a ensuite estimé manuellement l'importance de chacun de ces critères en leur attribuant un poids (*Weight* sur la deuxième colonne). Puis il a évalué manuellement le score de chaque candidat pour chaque critère (colonnes restantes). Le poids représente également le score maximal qu'un candidat peut avoir pour ce critère, selon l'utilisateur. Il suffit ensuite d'additionner les scores du candidat afin d'obtenir son score total (par exemple, celui du candidat *C* est égal à : 20 + 1 + 4 + 12 + 17 + 5 = 59). Afin de garder des valeurs homogènes, la somme des poids est égale à 100, ainsi le score final de chaque candidat représente son pourcentage total de satisfaction des critères d'évaluation. On constate donc que le candidat *A* est de loin le meilleur avec un score total de 77, loin devant les candidats *B* (45) et *C* (59). C'est donc le candidat *A* qui doit être sélectionné.

L'utilisation telle quelle de MCDA peut s'avérer problématique. Tout d'abord, le score local pour chaque critère dépend étroitement du poids, qui représente le score maximal. Ensuite, toutes les attributions de poids et de scores locaux, pour chaque candidat et chaque critère, sont manuelles. Face à un nombre élevé de critères et de candidats, cette technique devient donc

Evaluation criteria	Weight	Score Comp. A	Score Comp. B	Score Comp. C
Usability	25.0	23.0	15.0	20.0
Compatibility	10.0	7.0	2.0	1.0
Cost	15.0	10.0	5.00	4.0
Functionality	20.0	15.0	4.0	12.0
Security	20.0	15.0	12.0	17.0
Technical support	10.0	7.0	5.0	5.0
Total score	100.0	**77.0**	**45.0**	**59.0**

TAB. 2.1 – MCDA : Exemple d'application.

vite fastidieuse [207].

2.2.2 WSM (Weighted Scoring Method)

La méthode intitulée WSM ou WAS (pour "Weighted Average Sum") [200, 230, 195] est la technique d'agrégation la plus utilisée dans les situations de prise de décision [207]. Le principe général tel qu'il est rappelé dans [153] est le suivant : *"des critères sont définis et à chaque critère on assigne un poids"*. WSM présente quelques points communs avec MCDA. Premièrement, à chaque critère d'évaluation on associe un poids qui représente son importance. Deuxièmement, à chaque couple critère-candidat on associe un score qui représente la capacité du candidat à remplir le critère. Troisièmement, le candidat qui possède le score total le plus élevé est considéré comme le meilleur, celui qui doit être sélectionné. Cependant, contrairement à MCDA, le poids ne représente pas le score maximal pour un critère. En effet, il influe sur le score du candidat. L'avantage est de pouvoir assigner des poids indépendamment des scores locaux. L'inconvénient est que les poids ne sont plus bornés. Pour calculer le score total d'un candidat, on multiplie pour chaque critère d'évaluation le score correspondant par le poids du critère, et on agrège les produits ainsi obtenus.

La formule WSM la plus commune est celle donnée dans [153], qui utilise l'addition comme fonction d'agrégation :

$$score_c = \Sigma_{j=1}^{n}(weight_j * score_{cj}) \tag{2.1}$$

Le poids $weight_j$ représente l'importance du j-ème critère par rapport aux n-1 autres critères d'évaluation. Le score local $score_{cj}$ évalue le degré de satisfaction du critère numéro j par le candidat c. Le score total $score_c$ représente la valeur globale d'évaluation de c.

Le tableau 2.2 donne un exemple d'application de la méthode WSM sur 3 composants candidats A, B et C. L'utilisateur a défini 6 critères pour l'évaluation de ces candidats (première colonne sur le tableau), puis a estimé l'importance de chacun de ces critères en leur attribuant un poids (deuxième colonne) avant d'évaluer le score de chaque candidat pour chaque critère (colonnes restantes). Les critères sont les mêmes que pour le tableau 2.1, mais cette fois, les valeurs pour les scores sont indépendantes des poids. Afin de préserver malgré tout l'homogénéïté des valeurs, chaque poids et chaque score sont compris entre 1 (la moins bonne valeur) et 5 (la

Evaluation criteria	Weight	Score Comp. A	Score Comp. B	Score Comp. C
Usability	2.0	3.0	3.0	3.0
Compatibility	4.0	1.0	5.0	2.0
Cost	3.0	3.0	5.0	1.0
Functionality	5.0	4.0	4.0	3.0
Security	4.0	1.0	2.0	5.0
Technical support	5.0	2.0	5.0	3.0
Total score	XXXX	**53.0**	**94.0**	**67.0**

TAB. 2.2 – WSM/WAS : Exemple d'application (d'après [207])

meilleure valeur). Par exemple, pour le critère "sécurité", l'utilisateur a considéré que le candidat *A* n'était pas du tout sûr, que le candidat *B* l'était peu, et que le candidat *C* avait le niveau maximal. Contrairement à la méthode MCDA, où l'on se contente d'additionner les scores des candidats, avec la formule WSM on multiplie ces scores avec le poids donnée plus haut et on obtient les scores totaux de chaque candidat (par exemple, le score total du candidat *C* est égal à : 2 * 3 + 4 * 2 + 3 * 1 + 5 * 3 + 4 * 5 + 5 * 3 = 67). On s'aperçoit que le candidat *B* possède le meilleur score. C'est donc lui qui va être sélectionné.

Comme il est rappelé dans [153, 207], l'inconvénient majeur de cette technique réside dans la difficulté de définir un ensemble de critères d'évaluation pertinents, puis de leur assigner les bons poids, surtout quand ils sont nombreux. De plus, l'évaluation des scores locaux et l'assignation des poids restent manuelles, ce qui, comme pour MCDA, rend cette technique fastidieuse face à un nombre élevé de critères et de candidats.

2.2.3 AHP : Analytic Hierarchy Process

AHP est une technique de prise de décision multi-critères mise au point par T. Saaty [249, 250, 251]. Elle est utilisée avec succès dans de nombreux domaines [252], dont celui du génie logiciel [91, 194]. Par rapport aux techniques précédentes, AHP aide à décrire le besoin de manière organisée.

Tout d'abord, il faut définir l'objectif principal à atteindre pour prendre une décision sur la sélection d'un produit candidat. À partir de là, on décompose cet objectif en un arbre hiérarchique de critères et de sous critères d'évaluation, dont les feuilles sont les candidats à évaluer. Plus on descend dans l'arbre, plus les critères d'évaluation se précisent. La figure 2.2.3 montre un exemple d'application de cette décomposition. Si l'objectif principal est d'acheter une voiture, les critères principaux d'évaluation d'une voiture sont sa consommation, son prix, la sécurité qu'elle offre, etc... Les critères se décomposent ensuite en sous-critères, par exemple la sécurité consiste en la présence d'ABS, d'airbags... Enfin, on relie les voitures candidates (*Volvo*, *Mercedes*...) aux critères et aux sous-critères qui vont être utilisés pour les évaluer.

Ensuite, à l'intérieur de chaque critère-nœud, on détermine l'importance de chaque sous-critère par rapport aux autres. Pour ce faire, on assigne un poids à chaque nœud de l'arbre par rapport aux autres nœuds ayant le même parent. Par exemple, supposons que l'achat d'une voi-

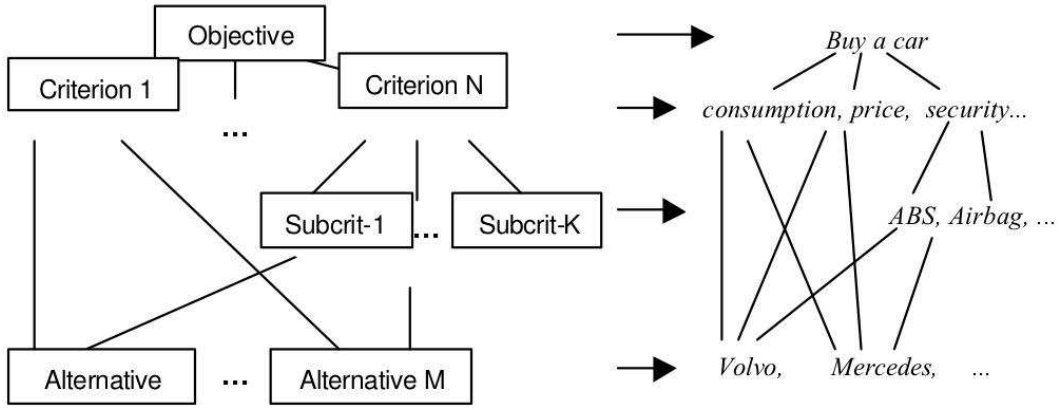

FIG. 2.2 – Analytic Hierarchy Process (d'après [175])

ture ne se décide qu'en fonction des trois critères montrés sur la figure 2.2.3 : la consommation, le prix et la sécurité. On peut pondérer ces trois critères de telle façon que la consommation soit considérée comme 2 fois plus importante que la sécurité, mais 6 fois moins importante que le prix. À partir de là, on peut utiliser une méthode comme WSM pour évaluer chaque candidat c en additionnant les scores locaux $score_{cj1}, ..., score_{cjn}$ à l'intérieur de chaque nœud j, et en propageant les sommes obtenues jusqu'à la racine de l'arbre pour obtenir le score total du candidat.

AHP est donc un moyen de définir et de hiérarchiser les différents critères d'évaluation avec précision. Cela permet d'utiliser WSM en assignant des poids cohérents. En effet, à l'intérieur de chaque nœud, les poids des nœuds fils sont bornés les uns par rapport aux autres. Cependant, comme il est rappelé dans [207], il est difficile d'employer AHP tel quel pour des systèmes logiciels complexes, qui comporteraient un nombre élevé de critères et de sous-critères, pour lesquels il y aurait une vaste bibliothèque de candidats à évaluer.

2.2.4 La fonction de satisfaction d'Alves, Franch *et al.*

Dans [8], C. Alves, X. Franch *et al* ont proposé une approche pour l'évaluation des candidats. Cette approche inclut notamment une fonction de satisfaction. Considérons un ensemble de critères d'évaluation $[q_1...q_n]$ tel que le niveau de chaque critère q_i peut être mesuré par une valeur numérique et qu'à chaque q_i est associé un besoin g_i sur son niveau. Soit M l'ensemble des valeurs que peut prendre q_i. On définit la fonction de satisfaction $Sat_{g_i} : M \rightarrow [0,1]$ qui, comme son nom l'indique, mesure par un pourcentage le niveau de satisfaction du besoin g_i par une valeur particulière de q_i.

Soient x_{target} et x_{worst}, respectivement la valeur de q_i recherchée (celle qui satisfait le mieux g_i) et la pire valeur de q_i (celle qui satisfait le moins g_i). Soit A_{q_i} la valeur de q_i pour le COTS candidat A. Formellement parlant, pour chaque critère d'évaluation q_i, on définit Sat_{g_i} comme

suit :

$$Sat_{g_i}(A_{q_i}) = \frac{A_{q_i} - x_{worst}}{x_{target} - x_{worst}} \tag{2.2}$$

Sachant que si $A_{q_i} < x_{worst}$ alors $Sat_{g_i}(A_{q_i}) = 0$ (le besoin g_i n'est pas du tout satifsait), et si $A_{q_i} > x_{target}$, alors $Sat_{g_i}(A_{q_i}) = 1$ (le besoin g_i est complètement satisfait).

Considérons maintenant que chaque besoin g_i a un poids w_{g_i} qui indique son importance par rapport aux autres besoins. On peut agréger les fonctions de satisfaction Sat_{g_i} en tenant compte du poids de chaque g_i afin d'obtenir la fonction de satisfaction totale $Sat(A)$ du COTS candidat A. Formellement parlant, $Sat(A)$ est définie comme suit :

$$Sat(A) = \Sigma_{i=1}^{n} w_{g_i} \times Sat_{g_i}(A_{q_i}) \tag{2.3}$$

L'avantage de cette formule par rapport aux techniques classiques de MCDM est que, plutôt que de s'en tenir à un score local A_{q_i}, on calcule sa fonction de satisfaction $Sat_{g_i}(A_{q_i})$. L'évaluation locale des candidats pour chaque critère est donc plus précise, et tient compte des besoins de l'application. En effet, avec les techniques précédentes, on effectue une simple évaluation des candidats sur les différents critères d'évaluation pour choisir le "meilleur", sans aucune considération pour les éventuels besoins de l'application. Avec une fonction de satisfaction, on spécifie clairement le niveau que doit satisfaire le candidat pour chaque critère, puis on choisit le candidat qui a le meilleur niveau. L'inconvénient est que l'attribution des scores locaux A_{q_i} reste manuelle pour chaque critère. Rien n'est proposé pour indiquer comment mesurer tel critère de performance ou tel critère de maintenabilité. De même que rien n'est proposer pour indiquer comment mesurer des critères de nature aussi différente sur une base commune ?).

2.3 Processus de sélection de composants

Selon [83], l'idée d'avoir un processus systématique d'évaluation et de sélection de composants a été introduite en 1982 par I. Brownstein et N. Lerner pour les paquetages logiciels [43]. Brownstein et Lerner incluaient dans leur processus des phases de spécification des besoins, d'identification des activités nécessaires à la sélection des paquetages candidats, et d'estimation des ressources nécessaires pour cette évaluation. L'idée a été reprise par d'autres travaux [276, 81, 156, 100, 11, 84], avant d'être améliorée en 1991 par G. Subramanian et M. Gershon [271]. Ceux-ci ont introduit les phases de définition des critères et d'évaluation des candidats sur la base de ces critères. F. Williams [292] a introduit des phases de hiérarchisation des critères et d'identification des candidats potentiels. D'autres travaux ont mis l'accent sur les critères de qualité [3, 134]. Finalement, l'idée a été adaptée aux composants COTS en 1995 avec l'approche OTSO de J. Kontio *et al* [151, 152, 153].

Dans cette section, nous allons présenter quelques travaux ayant trait à la sélection de composants. Nous commencerons par OTSO et PORE, car ce sont les processus de sélection de composants COTS les plus complets. Ils donnent non seulement une définition des différentes étape de la sélection, mais aussi des solutions et des techniques pour réaliser chacune de ces

étapes. OTSO fut en particulier le premier processus de sélection dédié aux COTS, tandis que PORE fut le premier processus à proposer une sélection progressive et itérative. Nous présenterons ensuite les autres travaux, classés en deux catégories. Dans la première, nous détaillerons les travaux qui contribuent à redéfinir le processus de sélection lui-même. Dans la seconde catégorie, nous présenterons ceux qui se concentrent sur une étape particulière de la sélection, particulièrement la définition des critères d'évaluation.

2.3.1 OTSO

OTSO (Off-The-Shelf-Option) est un processus mis au point par J. Kontio *et al* [151, 152, 153]. À l'origine, il s'agissait d'un mécanisme de recherche de composants en bibliothèque (un état de l'art du domaine appelé "recherche de composants" est fait dans le chapitre 3). On pouvait le considérer comme une extension et une application aux composants COTS de la méthode de recherche *Experience Factory* de V. Basili *et al* [17, 16]. C'est finalement devenu l'un des premiers processus existants de sélection de COTS, ainsi que l'une des contributions pionnières au domaine naissant à l'époque de la "sélection de composants".

Sachant que l'objectif principal est d'intégrer un ou plusieurs composants COTS dans une application, OTSO propose : i) de donner une définition claire des tâches à mener dans le processus de sélection ; ii) de donner une définition incrémentale, hiérarchique et détaillée des critères d'évaluation ; iii) d'utiliser de manière appropriée les méthodes existantes de price de décision (MCDM) pour analyser les résultats de l'évaluation. Pour chacun de ces objectifs, OTSO recommande un certain nombre de méthodes. La figure 2.3 montre les principales phases du processus OTSO (représentées par des cercles) ainsi que les paramètres et données utilisées (encadrées par deux barres).

Une première phase consiste à définir les critères d'évaluation selon quatre paramètres liés à l'application dans laquelle on va intégrer le COTS candidat. Le premier paramètre concerne les besoins de l'application (*Requirement specification*). Cela inclut en particulier les besoins fonctionnels. Le deuxième paramètre concerne l'architecture globale de l'application (*Design specification*). Les troisième et quatrième paramètres sont les contraintes d'organisation (*Organisational characteristics*) et de projet (*Project plan*) liées à son développement. Pour cette phase de définition des critères d'évaluation, OTSO propose ses propres critères, détaillés dans [153]. Dans la phase de recherche (*Search*), le but est d'identifier les candidats potentiels. Cela peut conduire à modifier la spécification des besoins et à influencer la phase de définition des critères d'évaluation. La phase de pré-sélection (*Screening*) consiste alors à filtrer les candidats les plus pertinents au moyen de critères de base et de certaines informations sur ces candidats (*COTS Sources*). On peut alors passer à la phase suivante, qui consiste à évaluer en détail chaque candidat restant au moyen de critères plus précis. La phase finale consiste à analyser les résultats de ces évaluations en fonction des critères précédemment définis, de méthodes d'estimation de coût et d'approches multi-critères telles que WSM/WAS et AHP [249, 250, 251]. Cela permet de classer les candidats et de sélectionner le meilleur d'entre eux.

Un inconvénient majeur de la méthode OTSO, pointé dans [7, 180, 206], est qu'elle ne propose rien pour spécifier les besoins. En effet, elle suppose que c'est déjà fait. De plus, ce sont surtout les besoins fonctionnels de l'application qui sont pris en compte, au détriment des besoins non-fonctionnels, qui sont pourtant très importants. Enfin, les techniques utilisées pour

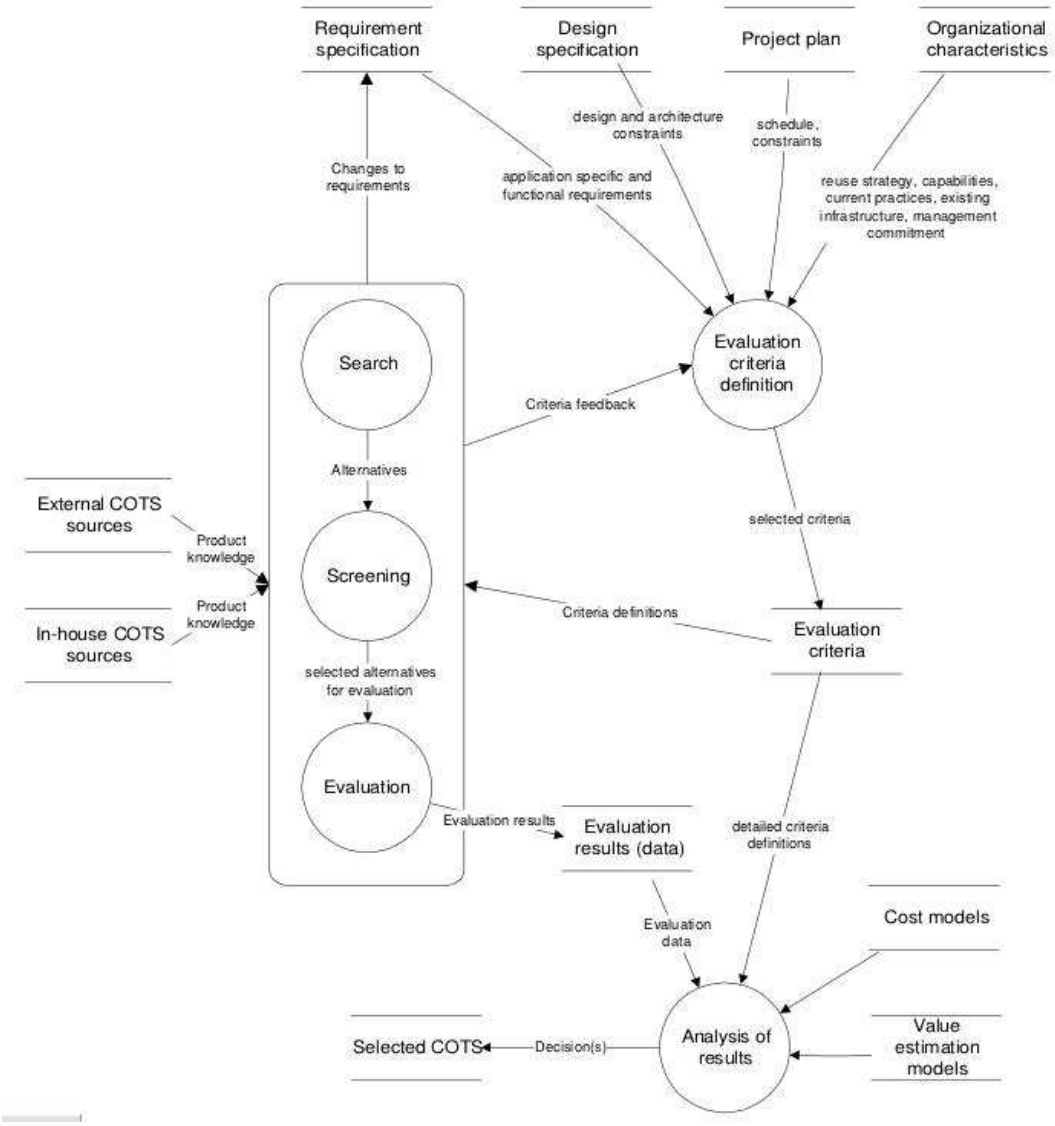

FIG. 2.3 – OTSO (d'après [152])

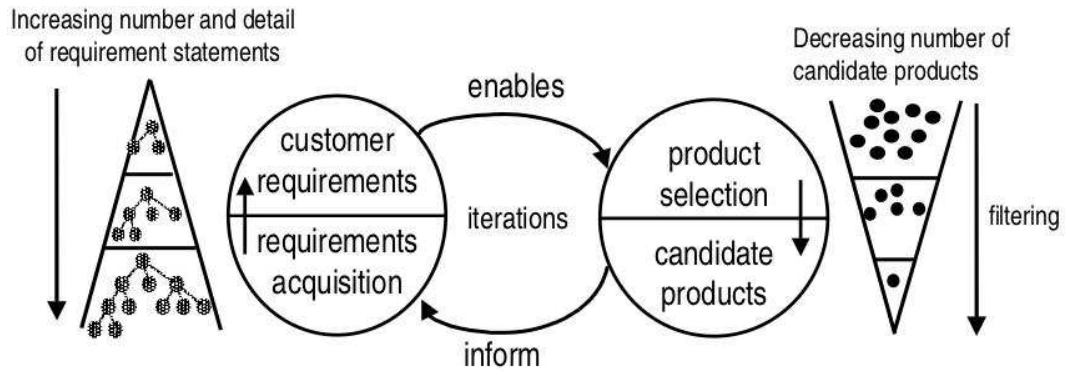

FIG. 2.4 – PORE : Principe (d'après [206])

l'évaluation des candidats se basent sur une agrégation de scores locaux obtenus manuellement.

2.3.2 PORE et ses applications

N. Maiden et C. Ncube ont constaté [181, 180, 206] que les méthodes "classiques" de spécification des besoins (*Requirements Engineering*) étaient inappropriées pour les applications à base de composants. Or, il est communément admis que de toutes les erreurs qui peuvent survenir lors du développement d'un logiciel, ce sont celles qui sont liées à la spécification des besoins de ce logiciel qui coûtent le plus cher. C'est-à-dire celles qui causent le plus de dégâts [41] et qui sont les plus difficiles à rattraper [30, 71, 67]. Le problème s'aggrave quand il s'agit de systèmes à base de COTS, puisque le succès du développement de telles applications dépend largement de la sélection des COTS qui les composent [83]. Pour que cette sélection soit un succès, il est donc capital que les besoins soient spécifiés avec précision [206].

Or, au moment où Maiden et Ncube publient leurs travaux, cet aspect de la sélection de composants est encore très peu traité. En particulier, il y a peu de travaux permettant à l'utilisateur de spécifier correctement des besoins tenant compte de l'offre qui existe sur le marché. Afin d'aider l'utilisateur à bien choisir son COTS en fonction de l'existant, Maiden et Ncube ont mis au point la méthode PORE (Procurement-Oriented Requirements Engineering) [180, 206]. Cette méthode entrelace la spécification des besoins et la sélection de candidats, comme le montre la figure 2.4. D'une part, les besoins de l'utilisateur (*customer requirements*) permettent la sélection des candidats pertinents (*product selection*) parmi ceux qui sont disponibles. D'autre part, l'examen des candidats ainsi filtrés (*candidate products*) informe l'utilisateur et l'incite à définir de nouveaux besoins ou à détailler ceux qui sont toujours valables. La précision dans la spécification des besoins s'accroît au fur et à mesure que le nombre de candidats décroît.

À la base, PORE définit trois types de besoins : i) les besoins fonctionnels principaux, ii) les besoins fonctionnels secondaires, et : iii) les besoins non-fonctionnels. Afin de satisfaire ces trois types de besoins, PORE utilise un processus itératif, décomposé en 5 processus génériques illustrés sur la figure 2.5. Le premier, *Identify Product*, consiste à repérer les candidats potentiels au moyen d'une étude de marché ou d'une expertise. Le second processus,

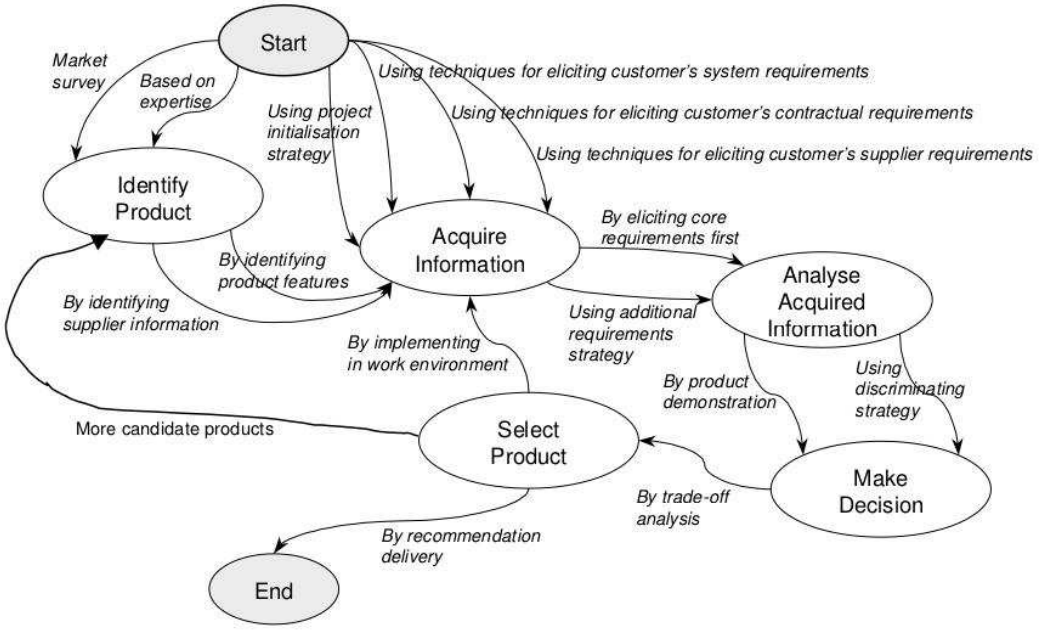

FIG. 2.5 – PORE : Les 5 processus génériques (d'après [206])

Acquire Information, consiste à obtenir des informations aussi bien sur les besoins de l'utilisateur (*customer requirements*) que sur les candidats précédemment identifiés (*product features*). Le troisième processus, *Analyse Acquired Information*, consiste à analyser l'information ainsi acquise pour produire des données exploitables dans le quatrième processus, *Make Decision*. Celui-ci consiste, à partir de ces données et de techniques de prise de décision multi-critères (MCDM), à déterminer si les candidats satisfont les besoins ou non. Ensuite, le cinquième processus, *Select Product*, consiste à rejeter les candidats qui ont été évalués comme non-satisfaisants par le processus précédent. Enfin, une fois que la sélection est faite, deux choix sont possibles. Soit on recommence l'opération en retournant aux premier et deuxième processus afin de détailler les besoins et réduire la liste des candidats. Soit on arrête la sélection (*End* sur la figure 2.5) quand on estime qu'on a trouvé ce qu'on cherchait. Chaque processus peut être répété plusieurs fois en cas d'information insuffisante.

Afin d'exécuter ces processus génériques, PORE se base sur les méthodes existantes. Par exemple, l'acquisition d'informations dans le deuxième processus se fait au moyen de cas d'utilisation et de techniques de *Knowledge Engineering* [247]. Et AHP [251] est utilisé dans le quatrième processus afin de choisir les critères de sélection des candidats. En ce qui concerne le cœur de la méthode PORE, il se compose de trois méta-modèles utilisés pour l'évaluation des candidats. Le premier sert à décrire les candidats. Le deuxième sert à décrire les besoins fonctionnels et non-fonctionnels. Enfin, le troisième sert à exprimer la satisfaction des besoins par un candidat au moyen d'un "score de satisfaction" [147]. Ce score est estimé manuellement [206].

Une extension de PORE est le processus SCARLET [178] (Selecting Components Against

Requirements). Ce processus offre des techniques élaborées pour aider l'utilisateur à éliminer progressivement les candidats qui ne satisfont pas ses besoins. SCARLET a été validé par un outil et utilisé dans le projet européen BANKSEC [179]. Ce projet est consacré aux systèmes à base de composants dans le domaine de la banque. À partir de BANKSEC et d'autres travaux [47, 99], N. Maiden a ensuite proposé l'approche REACT (REquirements ArchiteCTure) avec V. Sai et X. Franch [254]. L'originalité de REACT réside dans l'application de *i** à la sélection de composants. L'approche *i**, mise au point par E. Yu [300], utilise les agents pour modéliser les systèmes complexes et socio-techniques. De tels systèmes sont constitués d'acteurs, reliés entre eux par des "dépendances stratégiques". Une dépendance stratégique (ou *SD*) entre un acteur A_1 et un acteur A_2 représente, soit une ressource dont A_1 a besoin et que A_2 possède, soit une tâche dont A_1 a besoin et que A_2 peut exécuter, soit un objectif qui doit être atteint pour que A_1 fonctionne et que A_2 peut atteindre. On appelle "modèle *i** SD" l'ensemble des acteurs et des dépendances stratégiques qui les relient. Dans REACT, les composants sont vus comme des acteurs dont les services fournis par les uns et requis par les autres créent des relations de dépendance stratégique entre eux. Cela amène à voir les architectures des logiciels comme des modèles SD. À partir des composants candidats, on peut représenter par un modèle *i** SD les architectures candidates. Par "architectures candidates", on entend les architectures possibles de l'application une fois qu'on y a intégré les composants candidats. Cela permet de représenter les besoins à un niveau architectural et de réutiliser les techniques de BANKSEC à ce niveau. On peut donc effectuer des sélections multiples de composants [47, 99]. Une approche semblable à PORE est CRE (COTS-Based Requirements Engineering), proposée par C. Alves et J. Castro [7]. Cette approche reprend le principe d'élimination progressive des candidats au fur et à mesure que les besoins se précisent. La nouveauté réside dans l'utilisation du *NFR Framework* de L. Chung, B. Nixon et J. Mylopoulos [204, 59] . Cette approche consiste à décomposer l'objectif non-fonctionnel principal en besoins et sous-besoins non-fonctionnels hiérarchisés entre eux. En quelque sorte, il s'agit d'une méthode AHP pour les besoins non-fonctionnels.

L'avantage de la méthode PORE et de ses applications réside dans le processus itératif et progressif de sélection. Cela permet d'alterner la spécification des besoins et le filtrage des candidats. Cependant, certains inconvénients ont été pointés. Dans [158], l'auteur souligne qu'il est parfois difficile de savoir comment arrêter l'itération. Et dans [7], les auteurs trouvent peu claire la manière d'utiliser les besoins dans le processus d'évaluation, ainsi que la manière d'éliminer les candidats.

2.3.3 Innovations dans la définition du processus de sélection

Nous allons maintenant présenter les approches qui innovent dans la définition des étapes de la sélection de composants. La plupart de ces processus se contentent d'indiquer de manière informelle les différentes étapes que doit avoir le processus. D'autres, comme IusWare, essaient de définir formellement ces étapes.

2.3.3.1 CEP

Le processus CEP (Component Evaluation Process), mis au point par B. Phillips et S. Polen [227], fait partie d'un ensemble de processus regroupés dans l'approche BASIS (Base Application Software Integration System). Cette approche propose d'améliorer l'architecture des logiciels à base de composants en facilitant la sélection et l'intégration de ceux-ci [15]. CEP reprend plusieurs phases proposées dans OTSO. Parmi elles, on retrouve les phases de recherche et d'identification des candidats potentiels, de définition des critères, et d'évaluation des candidats précédemement identifiés. De plus, comme OTSO, CEP fait la différence entre la collecte des données et l'analyse de ces données. C'est-à-dire, entre l'évaluation des candidats et l'interprétation des résultats de cette évaluation. Cependant, CEP ajoute par rapport à OTSO une phase préliminaire, qui consiste à mesurer "l'effort d'évaluation". On mesure cet effort en estimant le nombre de COTS candidats qu'il faudra rechercher et analyser, et en planifiant les différentes étapes de l'évaluation ainsi que les ressources utilisées. Contrairement à OTSO, CEP ne propose aucune méthode pour réaliser ces différentes phases.

2.3.3.2 PECA

PECA est un processus d'évaluation de COTS mis au point par S. Comella-Dorda *et al.* [62] et inspiré en partie par le standard ISO-14598 [131]. Ce processus se compose de quatre activités consécutives : Planification de l'évaluation, Établissement des critères d'évaluation, Collecte des données et Analyse des données (d'où le nom de PECA). Ce processus a servi de base aux travaux de G. Lewis sur le développement de systèmes à base de COTS [167] et l'évaluation de technologies dans le contexte des services Web [169]. L'approche DESMET [147, 148] et le processus proposé par H. Lin *et al* [170] sont basés sur le même principe que PECA.

Dans la phase de planification, on choisit les personnes en charge de l'évaluation, on identifie les COTS candidats potentiels et on choisit l'approche d'évaluation qu'on va utiliser. Dans la phase d'établissement des critères d'évaluation, on identifie les besoins de l'application, avant de définir des critères hiérarchisés sur la base de ces besoins. Dans la phase de collecte des données, on inspecte les COTS candidats au moyen de ces critères d'évaluation, afin d'en retirer des informations nécessaires pour la suite. Enfin, la phase d'analyse des données consiste à interpréter et analyser ces informations au moyen de l'approche d'évaluation établie dans la première phase. Cela permet d'en tirer des recommandations sur le ou les COTS qu'il faut sélectionner. Il faut noter que dans chaque phase, on peut revenir en arrière. En effet, certaines informations et résultats imprévus (des candidats trop similaires ou éliminés en trop grand nombre, par exemple) peuvent entraîner une révision des phases de planification ou d'établissement des critères. PECA permet plusieurs retours arrières si les résultats ne sont pas satisfaisants.

L'originalité de ce processus réside dans la phase de préparation de l'évaluation. Cette phase peut permettre de simplifier celles qui suivent. Les tâches sont également bien définies et départagées. Cependant, les auteurs ne donnent aucune indication sur les techniques qu'on pourrait utiliser pour exécuter l'une ou l'autre de ces tâches. Il en est de même pour DESMET [147, 148] (qui concerne davantage "l'estimation de méthodologies") ainsi que pour le

processus proposé de Lin *et al* [170].

2.3.3.3 Six Sigma et son application aux COTS

À l'origine, Six Sigma [277] est une approche en cinq phases permettant de définir un processus de sélection de paquetages logiciels. La première phase consiste à définir le processus de sélection, en identifiant et en hiérarchisant les problèmes ainsi que les besoins de l'utilisateur. La deuxième phase consiste à mesurer le processus en détaillant et en quantifiant le problème. La troisième phase consiste à analyser ce qui ne va pas en déterminant les origines du problème et en proposant des solutions. La quatrième phase consiste à améliorer le processus en implémentant et en hiérarchisant certaines solutions. Enfin, la cinquième phase consiste à contrôler le processus ainsi amélioré en mesurant les changements et en s'assurant qu'ils apportent les effets souhaités.

A. Ceccich et M. Piattini [57] se sont basés sur les trois premières phases de l'approche Six Sigma afin de définir un processus de sélection de composants COTS en trois cycles. Le premier cycle consiste à définir les critères d'évaluation à partir des besoins des utilisateurs. Le deuxième consiste à effectuer une pré-sélection sur les COTS candidats afin de ne garder que ceux qui satisfont les critères fonctionnels d'évaluation. Enfin, le troisième consiste à mesurer les propriétés architecturales et non-fonctionnelles des candidats restants afin de sélectionner celui qui possède la meilleure qualité. À tout moment, si les résultats sont insatisfaisants (par exemple, dans le cas d'un filtrage trop strict), on peut revenir à un cycle précédent.

L'originalité de ce processus est de faire se succéder les phases d'évaluation fonctionnelle et d'évaluation non-fonctionnelle. On filtre d'abord les candidats qui répondent fonctionnellement au besoin, puis on sélectionne le meilleur sur la base de sa qualité. Cependant, les auteurs ne proposent aucune technique pour réaliser ces différentes phases.

2.3.3.4 CISD

CISD (COTS-based Integrated Systems Development) est un processus de développement d'applications à base de composants COTS mis au point par V. Tran et D.-B. Liu [282]. Comme le montre la figure 2.6, ce processus prend en entrée les COTS candidats ainsi que les besoins initiaux (*System Requirements*). Puis il produit en sortie une application complète avec des COTS candidats sélectionnés et intégrés. CISD est constitué de trois étapes principales. La première (*Product Identification* sur la figure) consiste à définir et hiérarchiser les critères d'évaluation, puis à classer les COTS candidats par groupes. La deuxième étape (*Product Evaluation*) consiste à évaluer ces COTS candidats individuellement et à l'intérieur de leur groupe. Cette évaluation se fait en fonction de leurs aspects fonctionnels, architecturaux et non-fonctionnels. Elle permet de sélectionner le meilleur groupe de candidats. Enfin, la troisième étape du processus (*Product Integration*) consiste à concevoir l'architecture globale de l'application, puis à intégrer l'ensemble des COTS candidats sélectionnés précédemment. On obtient ainsi une application complète.

La particularité de CISD est, en plus de couvrir la plupart des tâches qui composent la sélection de composants, de classer les candidats par groupes afin de les évaluer et de les sélectionner en groupe pour les intégrer dans l'application. Mais son défaut majeur est de ne

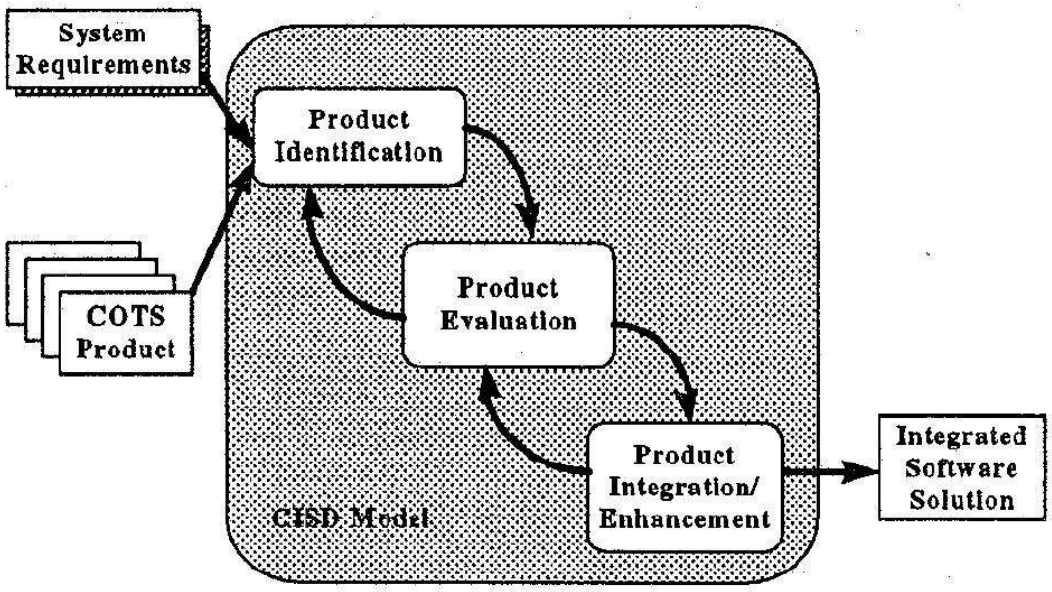

FIG. 2.6 – CISD : vue d'ensemble de l'approche (d'après [282])

rien proposer en ce qui concerne la manière d'accomplir ces tâches : ni pour l'analyse des besoins, ni pour la répartition des COTS en groupes de candidats, ni pour le choix des critères d'évaluation , ni pour le mode d'intégration des candidats retenus.

2.3.3.5 IIDA

IIDA (Infrastructure Incremental Development Approach), mise au point par G. Fox *et al* [95], est un processus de développement d'applications à base de COTS. La plupart des phases de ce processus sont validées par des prototypes. Les premières phases de ce processus consistent : i) à définir un document appelé "infrastructure recherchée", qui représente la vision globale de l'application et qui est constituée de l'ensemble des besoins et des stratégies de développement à long terme de l'entreprise ; ii) à spécifier les besoins de l'application à court terme pour chaque étape du développement. On peut en tirer les premiers critères d'évaluation pour une pré-sélection des COTS susceptibles d'être adéquats. Les phases suivantes consistent à concevoir l'architecture globale de l'application, puis à définir des critères d'évaluation plus détaillés qui serviront à sélectionner les COTS qu'on va vraiment utiliser. Enfin, les dernières phases consistent à intégrer ces composants et tester l'application.

L'originalité de cette approche repose d'une part sur la spécification des besoins à travers une "infrastructure recherchée", et d'autre part sur la validation des différentes phases du processus par des prototypes. C'est également l'un des rares processus consacré à la sélection multiple de composants. L'inconvénient de ce processus est que la manière de spécifier les besoins à travers cette infrastructure recherchée est peu claire, tout comme la manière d'évaluer les candidats [206].

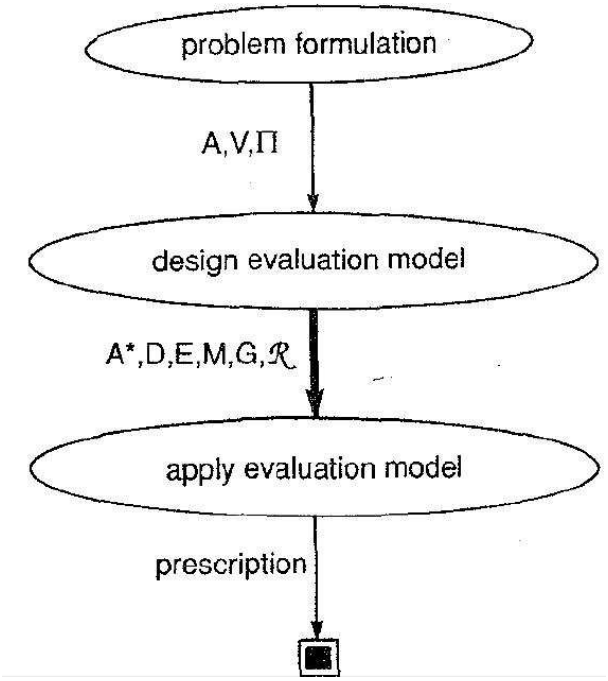

FIG. 2.7 – IusWare : Les trois phases du processus (d'après [199])

2.3.3.6 IusWare

IusWare (IUStitia SoftWARis) [199] est un processus utilisant MCDA [261, 245, 246, 286] pour formaliser la sélection de logiciels (lesquels peuvent être des composants COTS). Ce processus se décompose en trois phases, ainsi que le montre la figure 2.7.

La première phase consiste, à partir d'entrevues avec le client, à formuler le problème selon trois aspects : i) l'ensemble A des logiciels qui seront évalués ; ii) l'ensemble V des points de vue (point de vue utilisateur, producteur, vendeur...) à partir desquels ces logiciels seront évalués ; iii) la fonction Π qui indique la forme du résultat final. Par exemple, l'ensemble des logiciels peut être fragmenté en sous-ensembles de "bons" et "mauvais" produits, ou il peut être ordonné du "meilleur" au "moins bon", ou alors on peut associer à chaque produit une courte description.

La deuxième phase consiste à définir un modèle d'évaluation des logiciels (*Design evaluation model* sur la figure 2.7). Elle est détaillée sur la figure 2.8. Cela renferme la définition des critères d'évaluation, mais pas seulement. D'une part, il faut d'abord choisir l'ensemble $A^* \subseteq A$ des logiciels tels que l'évaluation de chacun d'entre eux soit indépendante de celle des autres. D'autre part, il faut définir l'ensemble D des critères d'évaluation (en général, il s'agit d'attributs qualité dérivés des points de vue de l'ensemble V). Ensuite, il faut définir l'ensemble M des mesures associées à ces critères, ainsi que l'ensemble E des valeurs associées à M (on note $e_j(a)$ la valeur du critère j mesuré sur le logiciel a). Après quoi il faut définir l'ensemble G des règles utilisées pour transformer les mesures de M en préférences. Cela permet de classer

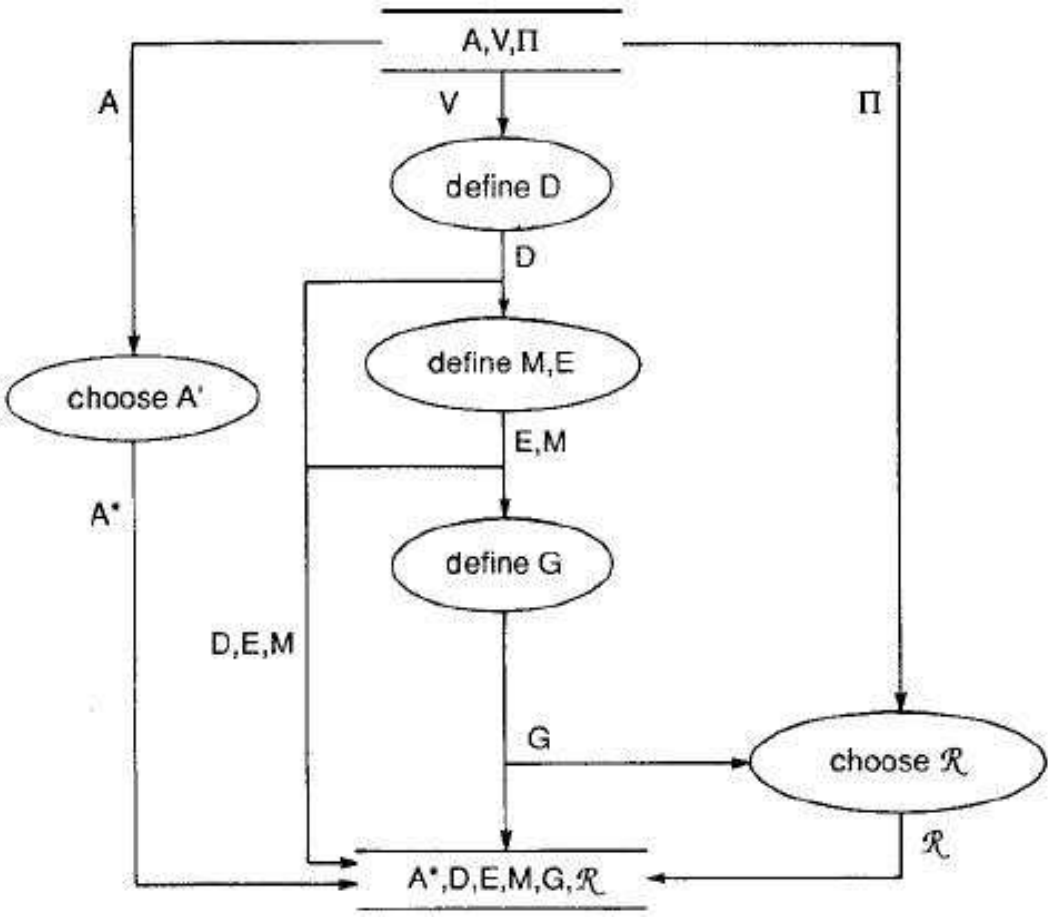

FIG. 2.8 – IusWare : définition du modèle d'évaluation (d'après [199])

les logiciels par ordre d'adéquation aux critères d'évaluation [82]. Pour deux logiciels a et b, $g_j(a)$ et $g_j(b)$ sont les rangs respectifs de a et de b obtenus en fonction de $e_j(a)$ et $e_j(b)$, et $s_j(a,b)$ est la relation d'ordre qui relie $g(a)$ et $g(b)$. s_j peut être par exemple, une relation de supériorité : $\forall x,y \in A^* : s_j(x,y) \Leftrightarrow g_j(x) \geq g_j(y)$ avec $g_j = e_j$. Enfin, il faut choisir la technique d'agrégation $\mathfrak{R}$, décrite par un algorithme capable d'agréger tous les rangs obtenus afin d'en retirer un classement global de tous les logiciels candidats. Le choix de $\mathfrak{R}$ se fait parmi plusieurs techniques possibles [144, 285], dont AHP [249, 250, 251], et en fonction de plusieurs paramètres [286, 246].

Enfin, la troisième phase consiste à appliquer le modèle d'évaluation défini précédemment (*apply evaluation model* sur la figure 2.7). Elle est détaillée sur la figure 2.9. Tout d'abord, pour chaque logiciel $a \in A^*$, on mesure l'adéquation de a aux critères d'évaluation. Cela permet d'obtenir les valeurs $e(a) \in E$ pour les attributs de D, au moyen des mesures de M. Ensuite, à l'aide de G, on obtient chaque rang $g(a)$ pour chaque logiciel a. On vérifie qu'il n'y a pas de redondance et, depuis ces rangs, on définit chaque relation d'ordre s. Enfin, après avoir vérifié

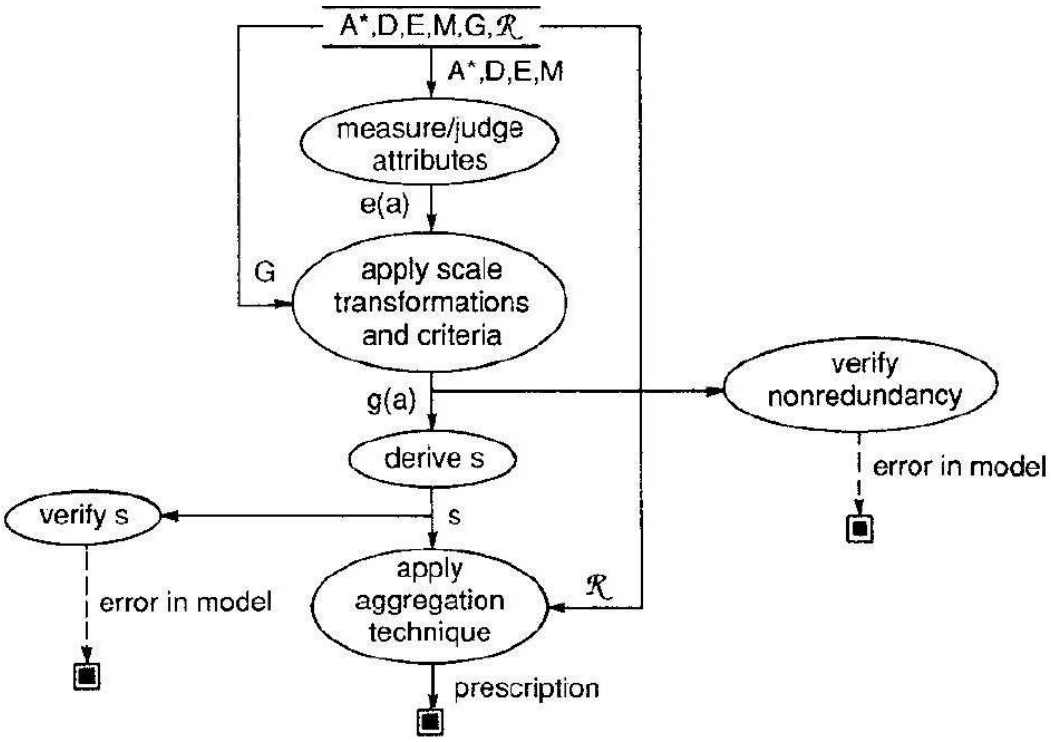

FIG. 2.9 – IusWare : Phase d'application du modèle d'évaluation (d'après [199])

qu'il n'y a pas d'erreur liée à ces relations, on applique la technique d'agrégation $\mathfrak{R}$ pour classer les logiciels et choisir le meilleur d'entre eux.

L'avantage de IusWare est de définir formellement les différentes étapes du processus d'évaluation et de sélection des logiciels (ce qui inclut les composants). Cependant, son inconvénient majeur, pointé par C. Ncube [206], réside dans le fait que rien n'est proposé pour décrire les besoins des utilisateurs. Les critères d'évaluation sont obtenus en parlant avec les différents acteurs regroupés dans "l'ensemble des points de vue" (utilisateur, producteur, vendeur...). Mais rien ne leur est donné formellement parlant pour les aider à détailler leurs besoins ou à choisir tel critère plutôt que tel autre.

2.3.4 Innovations dans la définition des besoins et des critères d'évaluation

De toutes les phases communes à la plupart des processus de sélection, la phase de définition des critères d'évaluation est l'une des plus importantes. En effet, le bon déroulement des phases suivantes dépend étroitement du bon choix des critères. Une phase de spécification des besoins peut justement aider à bien les choisir, même si les processus qui proposent une telle phase sont rares. Les innovations peuvent porter sur la manière de définir les critères d'évaluation, à l'instar de *Delta*. Elles peuvent aussi porter sur la manière de les organiser et de les hiérarchiser, comme c'est le cas pour BAREMO. Elles peuvent porter également sur la

proposition d'un ensemble de critères précis, en fonction de considérations non-fonctionnelles (SSEF, CAP et VERPRO, CBCPS) ou "socio-techniques" (STACE). Elles peuvent porter, enfin, sur la définition préliminaire des besoins et de critères d'évluation, comme c'est le cas pour RCPEP et DesCOTS.

2.3.4.1 Delta

L'approche Delta a été mise au point par A. Brown et K. Wallnau [42] afin d'aider les entreprises à évaluer une technologie en fonction de ses particularités par rapport aux technologies concurrentes. Le concept clé de cette approche est le "delta". Un "delta" décrit comment une nouvelle technologie innove par rapport aux technologies existantes sur un aspect particulier. Le principe sous-jacent est que pour évaluer une technologie à sa juste valeur, il faut connaître et comprendre ses deltas. En plus des phases classiques de définition des critères et d'évaluation des technologies candidates, l'approche Delta rajoute une phase préliminaire qui consiste à mesurer ce delta. Dans cette phase, on essaie d'anticiper les nouveaux aspects des technologies candidates et leur impact, en fonction de leur "généalogie" et de leur "habitat". Partant du principe de les nouvelles technologies sont le plus souvent des améliorations des anciennes, la généalogie d'une technologie comprend l'historique de celles dont elle est issue. Par exemple, la généalogie de la programmation orientée objet comprend les concepts dont elle a hérité (abstraction, polymorphisme, masquage d'informations...). L'habitat d'une technologie inclut son utilisation et ses liens avec d'autres technologies.

Les deux phases suivantes sont plus "classiques", puisqu'il s'agit, dans l'une, de définir des critères sur la base desquels les technologies candidates seront évaluées dans l'autre. La définition des critères se fait aussi bien en fonction des besoins des utilisateurs qu'en fonction des deltas de chaque technologie candidate évalués dans la phase précédente. Cela permet de déterminer les apports réels de chaque technologie candidate par rapport aux autres.

L'approche Delta est en fait une méthodologie d'évaluation de technologies qui peuvent être des composants COTS. Cette approche est centrée autour des apports potentiels de ces technologies, non seulement les unes par rapport aux autres, mais aussi par rapport aux technologies déjà présentes. Son originalité réside donc dans la mesure de ce "delta". Cependant, les auteurs ne proposent aucune technique pour automatiser ne serait-ce que partiellement l'une des phases de leur processus. De même, rien n'est proposé pour analyser les besoins des utilisateurs.

2.3.4.2 BAREMO

AHP [249, 250, 251] a été adapté et appliqué aux composants COTS dans de nombreux processus [61, 142, 160, 161, 198, 278, 291]. Parmi les applications les plus complètes, on trouve le processus BAREMO (pour BAlance REuse MOdel), mis au point par A. Lozano-Tello et A. Gómez-Pérez [175]. Les étapes d'AHP, décrites dans la sous-section 2.2.3, ont été modifiées afin d'être adaptées aux composants de la maniére suivante :

Dans la première étape, on spécifie précisément les objectifs du projet. La figure 2.10 montre les objectifs spécifiés par Lozano-Tello et Gómez-Pérez dans [175]. Les *Dimensions* sont les principaux critères de sélection pour un composant particulier. Elles indiquent ce qui

DIMENSIONS	FACTORS / CHARACTERISTICS	QD:Quality of documentation	M:Methodology	TH:Training help	AH:Adaptation help	DH:Development help	Mo:Modularity	Co:Complexity	S:Size	RC:Requirement covered	I:Interoperativity	TQ:Tools quality	EH:Ease of handling	AL:Acquisition license	KM:Kind of mantenance	UC:Update cost	Res:Human, Hw & Sw resources	ES:Execution speed	OA:Output accuracy	A:Applications	Ma:Maturity	Cr:Credibility
Production Time	**TT**:Training time	X	X	X			X	X	X													
	AT:Adaptation time	X	X		X		X	X	X	X	X	X										
	DT:Development time					X							X									
Cost Rating	**LP**:Licenses price													X	X	X						
	AE:Adaptation expenses				X												X					
	DE:Developm. expenses					X											X					
Product Quality	**E**:Effectiveness																X	X	X			
	R:Reliability		X												X					X	X	X
Developm. Risk	**F**:Feasibility										X									X		
	C:Capability		X		X	X																

FIG. 2.10 – BAREMO : Critères de sélection (d'après [175])

se passe si on sélectionne ce composant. Par exemple, *Production Time* indique le temps qu'il reste à développer après avoir choisi le composant. *Cost rating* représente l'investissement financier requis pour développer le projet avec ce composant. *Financial product quality* désigne la qualité du projet final si ce composant est intégré. Et *Development risk* représente la probabilité que le développement se déroule dans de bonnes conditions avec ce composant. Les facteurs (*Factors* sur la figure 2.10) sont les sous-critères qui composent les dimensions. Par exemple, le temps de production est déteminé par trois facteurs. Le premier est le temps d'apprentissage du composant choisi (*TT : Training Time* sur la figure). Le deuxième est le temps d'adaptation du composant avant son intégration (*AT : Adaptation Time*). Et le troisième facteur est le temps qu'il reste à développer le projet une fois le composant intégré (*DT : Development Time*). Enfin, les caractéristiques (*Characteristics* sur la figure) sont les éléments qui influencent les facteurs. Par exemple, la qualité de la documentation (*QD*) contribue à diminuer aussi bien le temps d'apprentissage que le temps d'adaptation. Une bonne méthodologie (*M*), quant à elle, influence ces deux temps, ainsi que la fiabilité du produit (facteur *Reliability* de la dimention *Product Quality*) et la capacité à bien mener le développement (facteur *Capability* de la dimension *Development Risk*).

Dans la deuxième étape, on construit l'arbre de décision à partir des objectifs définis dans la première étape. Le principe d'arbre de décision pour hiérarchiser les critères d'évaluation est également présent dans [253, 60], bien qu'AHP n'y soit pas utilisé. Comme le montre la figure 2.11, la racine est l'objectif principal, c'est-à-dire la sélection du composant le plus approprié pour le projet. Au premier niveau sont placées les dimensions du projet. Au deuxième ni-

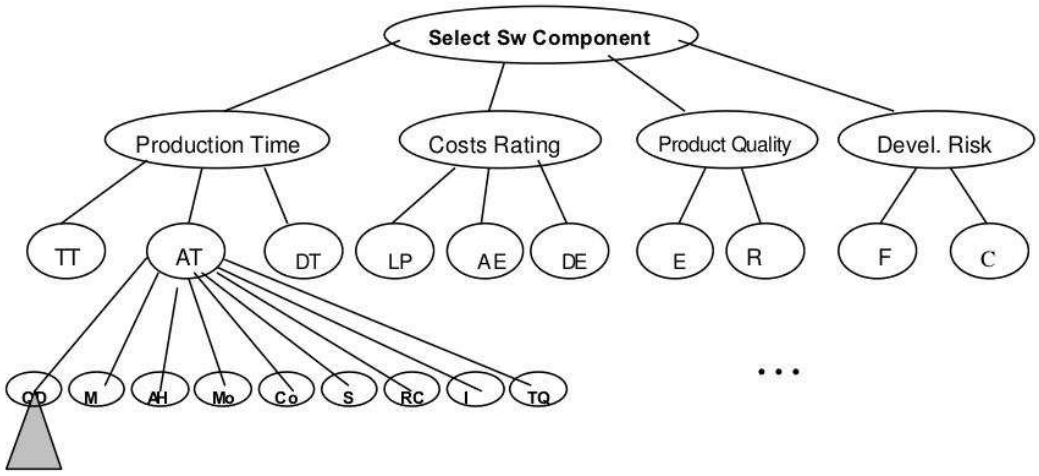

FIG. 2.11 – BAREMO : L'arbre de décision (d'après [175])

veau, on place les facteurs liés à ces dimensions. Au troisième niveau, on place pour chaque facteur les caractéristiques qui l'influencent. Par exemple, la figure 2.11 montre le développement du nœud associé au facteur *AT* (*Adaptation Time*) de la dimension *Production Time*. Les fils du nœud sont les caractéristiques qui influencent ce facteur. Au plus bas niveau, on place comme feuilles les différents candidats disponibles.

La troisième étape consiste à établir les poids W de chaque caractéristique à l'intérieur d'un même facteur, de chaque facteur à l'intérieur d'une même dimension, et de chaque dimension par rapport à l'objectif principal. Ensuite, la quatrième étape consiste pour chaque candidat à évaluer manuellement les valeurs V des caractéristiques associées (quelle est la qualité de sa documentation ? quelle est la méthodologie adaptée à son utilisation ? etc...). Enfin, dans la cinquième et dernière étape, on utilise la méthode WSM afin d'obtenir l'évaluation globale. Pour ce faire, à l'intérieur de chaque nœd et pour chaque élément i fils de ce nœud, on multiplie la valeur V_i par le poids W_i correspondant à cet élément, puis on additionne ces valeurs pondérées. La formule pour chaque nœud de n éléments est donc : $\Sigma_n W_i * V_i$. Ensuite, on propage ces valeurs au nœud supérieur jusqu'à la racine. Le candidat qui possède la valeur la plus élevée est celui qui est choisi.

L'originalité de BAREMO est d'adapter directement AHP pour les COTS, tout en proposant ses propres critères d'évaluation adaptés à ce type de composant. Cependant, comme pour AHP, l'évaluation pour chaque critère doit se faire manuellement. Ce qui rend le calcul fastidieux, compte tenu du nombre important de critères et du nombre potentiellement important de candidats.

2.3.4.3 SSEF

G. Boloix et P. Robillard ont proposé [33] l'approche SSEF (Software System Evaluation Framework). Cette approche définit pour la sélection de logiciels des critères d'évaluation

spécifiques. Ces critères sont regroupés selon trois types représentant les différents points de vue différents, et évalués au moyen d'une métrique à trois niveaux : "basique" (le moins bon), "intermédiaire" et "avancé".

Le point de vue des producteurs est représenté par les critères de type "Projet", qui évaluent l'efficacité du processus choisi, des personnes impliquées dans le projet et des outils utilisés. Le point de vue des développeurs est représenté par les critères de type "Système", qui évaluent la qualité globale du logiciel développé, sa performance, et la capacité des développeurs à maîtriser la technologie utilisée. Enfin, le point de vue des utilisateurs est représenté par les critères de type "Environnement", qui évaluent la satisfaction de leurs besoins, leur capacité à utiliser le logiciel et les bénéfices qu'ils peuvent en tirer.

Les critères d'évaluation choisis mettent donc bien en avant la partie non-fonctionnelle du point de vue de toutes les parties en présence. Cependant, si ces critères peuvent être évalués par une métrique à trois niveaux, ils ne sont pas hiérarchisés entre eux. De plus, rien n'est proposé pour les agréger [152]. On ne peut donc ni évaluer la qualité globale des candidats, ni les comparer entre eux [42].

2.3.4.4 CAP

CAP (COTS Acquisition Process) est un processus de sélection de COTS mis au point par M. Ochs *et al.* [212, 213] pour le compte de Siemens. Ce processus est constitué lui-même de trois composants. Le premier s'occupe de définir d'une part une taxonomie de critères d'évaluation et d'autre part un "plan d'évaluation". La taxonomie consiste en un ensemble ordonné de critères et de sous-critères fonctionnels et non-fonctionnels. Le plan d'évaluation prépare les différentes étapes de l'évaluation des candidats en fonction de ces critères et de l'effort requis par ces étapes. Le deuxième composant évalue et classe les candidats selon les critères et le plan définis dans le composant précédent afin de choisir le meilleur d'entre eux. Enfin, le dernier composant stocke toutes les informations relatives à chaque évaluation afin qu'elle puisse resservir pour d'autres. Des exemples d'informations stockées sont la taxonomie d'évaluation, le plan d'évaluation, les candidats évalués et les mesures collectées.

Pour le premier composant, CAP propose sa propre taxonomie, inspirée du principe de décomposition d'AHP [249, 250, 251], des critères proposés pour OTSO [153] et du modèle de qualité ISO-9126 [132]. CAP propose également son propre plan d'évaluation basé sur la méthode G/QM (Goal/Question Metric) [39, 88, 263]. Chaque critère, qu'il soit "fonctionnel" (interopérabilité, sécurité...), "non-fonctionnel" (fiabilité, portabilité...), "architectural" (compatibilité architecturale...) ou "stratégique" (coût d'intégration, risques liés au COTS...), est mesurable par une métrique dédiée. La valeur de cette métrique est rentrée manuellement par l'utilisateur pour chaque candidat. Cette taxonomie a été étendue dans l'approche VERPRO (Vendor Economics and Risk Profiler) [299] afin de tenir compte des critères économiques relatifs aux vendeurs des COTS candidats tels que leur investissement en recherche et développement, leurs atouts sur le marché et les risques financiers liés à l'achat de leur COTS.

L'originalité de CAP réside dans cette taxonomie de critères d'évaluation fonctionnels et non-fonctionnels. De plus, l'utilisation du dernier composant permet de réutiliser les critères d'évaluation précédemment définis ainsi que les candidats précédemment analysés.

2.3.4.5 CBCPS

Le processus CBCPS (Contract-Based COTS Product Selection), mis au point par F. Ye et T. Kelly [296], utilise des contrats comme critères d'évaluation afin de garantir la sûreté des COTS candidats. En effet, les auteurs reprochent aux autres processus de négliger la qualité, et en particulier la sûreté, dans la définition des critères d'évaluation. Or, selon Ye et Kelly, dans un contexte de développement à base de COTS, la sûreté est un facteur décisif de sélection. Ce facteur nécessite d'être exprimé en termes de propriétés aussi bien fonctionnelles que non-fonctionnelles.

Les étapes initiales du processus CBCPS consistent à identifier, d'une part, les candidats pertinents, et d'autre part, les risques potentiels liés à la sûreté. Cette identification se fait au moyen d'entretiens et de réunions entre utilisateurs et concepteurs. Elle permet de définir quels sont les besoins initiaux en matière de sûreté, et de faire une analyse plus détaillée des risques spécifiques encourus par l'application. Les autres étapes consistent à définir les besoins de manière plus détaillée et examiner les candidats disponibles. On peut proposer ensuite une première architecture pour l'application qu'on veut développer. Ensuite, les COTS candidats sont examinés individuellement. Cela permet d'identifier et de hiérarchiser les risques de panne ou de mauvais fonctionnement liés à leur comportement. À partir de cette série d'examens, on peut établir un contrat global appelé "contrat d'acquisition" qui comprend des critères d'évaluation détaillés en termes de fonctionnalité et de sûreté. Enfin, les COTS candidats sont sélectionnés en fonction de leur respect des termes de ce contrat d'acquisition.

L'originalité du processus CBCPS est de mettre l'accent sur la sûreté et d'utiliser les contrats pour sélectionner les composants. Cependant, l'identification des besoins et des risques liés à la sûreté est très abstraite, tout comme l'évaluation des composants une fois que le contrat est établi.

2.3.4.6 STACE et la définition de critères socio-techniques

STACE (Social-Technical Approach to COTS Evaluation) est une approche mise au point par D. Kunda [159, 158]. Son but est de proposer un mécanisme de sélection de COTS qui tiendrait compte des aspects "socio-techniques" de cette sélection [201]. Ce souci de représenter les aspects socio-techniques est présent dans certaines approches de sélection de COTS [85], telles que le *Component Comprehension Model* de S. Ghosh *et al* [12, 106] qui propose de "comprendre" les composants afin de mieux les sélectionner.

La notion de "critère socio-économique" englobe aussi bien les critères de qualité "classiques" (tels que les caractéristiques du modèle ISO-9126 [132]) que ceux liés à la technologie (modèle de composant utilisé par le candidat, complexité de l'interface...) ainsi que les critères socio-économiques. Ceux-ci incluent les aspects financiers [232] (prix du candidat, licences disponibles...), les capacités de l'utilisateur (expérience avec la technologie utilisée par le candidat, préférence pour telle ou telle marque...), les variables liées au marché (réputation du candidat ou de la technologie qu'il utilise, maturité et stabilité du candidat [149], viabilité des produits de la même marque sur le long termps [244]...), et les capacités du vendeur du COTS candidat [123] (réputation, support technique offert...).

STACE est constitué de quatre processus reliés entre eux. Le premier est un processus de

définition des besoins qui combine les études de marché, les entretiens avec les utilisateurs utilisant la méthodologie JAD (Joint Application Development) mise au point par J. Wood et J. Silver [293], et l'étude des offres existantes. Le deuxième est un processus de définition des critères d'évaluation sur une base socio-technique. Le troisième est un processus d'identification des COTS candidats qui consiste à pré-sélectionner ceux qui sont pertinents. C'est-à-dire ceux qui satisfont les besoins principaux, afin qu'ils soient soumis à une évaluation plus détaillée [281]. Les mécanismes de pré-sélection peuvent être des études de marché, ainsi que des questionnaires envoyés aux vendeurs [244, 180] ou sur les forums internet et les mailing lists [151]. Enfin, le quatrième processus consiste à évaluer les candidats identifiés sur la base des critères définis précédemment. Cette évaluation se fait au moyen d'une stratégie de type *Keystone identification* [211] appliquée à AHP [249, 250, 251]. De même qu'OTSO [152], STACE sépare la collecte des données relatives à l'évaluation et l'analyse de ces données en vue de sélectionner le meilleur candidat.

L'originalité de STACE est de prendre en compte les critères "socio-techniques", et en particulier socio-économiques, peu utilisés par les autres processus de sélection de composants. De plus, STACE a recours à de nombreuses techniques et solutions existantes pour implémenter ses processus, telles que les critères d'OTSO [152, 153], AHP [249, 250, 251] et la *Keystone Identification* [211]. Cependant, comme le rappelle [7], la définition des besoins reste très vague, et certaines méthodes manuelles préconisées pour certaines phases sont peu applicables face à de vastes bibliothèques.

2.3.4.7 RCPEP

Le processus RCPEP (Requirements-Driven COTS Product Evaluation Process), mis au point par P. Lawlis *et al.* [164], met l'accent sur les besoins de l'utilisateur ainsi que sur la collaboration entre toutes les parties. Parmi celles-ci, on trouve les utilisateurs de l'application à développer, les vendeurs des COTS candidats et les développeurs, dont on distingue en particulier les analystes des besoins et les évaluateurs des candidats. RCPEP est constitué de deux étapes principales.

La première étape consiste, à partir de besoins définis initialement, à rechercher les candidats susceptibles d'y répondre. Cette étape se fait au moyen d'études de marché et de questionnaires soumis aux vendeurs de ces candidats. À partir des réponses de ces vendeurs et des commentaires éventuels des utilisateurs, les analystes produisent un "tableau des besoins". Ceux-ci sont hiérarchisés afin de déterminer leur importance respective. On évalue ensuite la satisfaction de chacun d'entre eux par les COTS candidats, au moyen d'une méthode similaire à WSM. Cela permet de trier et de pré-sélectionner les candidats pour une évaluation future.

La seconde étape consiste à effectuer cette évaluation. À partir des besoins établis précédemment et d'une collaboration entre les analystes et les utilisateurs, on extrait des critères d'évaluation détaillés. Ensuite, afin d'organiser les différentes étapes de l'évaluation des candidats, des scénarios sont établis. Ces scénarios et ces critères sont envoyés aux évaluateurs, ainsi que le tableau des besoins obtenu dans l'étape précédente. Ces évaluateurs, en tenant compte de l'architecture existante de l'application, dressent un autre tableau. Ce tableau récapitule la satisfaction des différents besoins par les candidats restants. Il permet aux évaluateurs d'analyser les résultats obtenus, puis de produire un jugement pour déterminer le(s) meilleur(s) candi-

dat(s).

L'originalité de RCPEP réside dans le partage des tâches et la collaboration entre les différents acteurs de la sélection, vendeurs aussi bien qu'utilisateurs, évaluateurs et analystes. Cela permet de définir des besoins et des critères d'évaluation plus pertinents. Cependant, cette méthode a deux défauts. Premièrement, son succès dépend de la disponibilité de chacun, en particulier des vendeurs des COTS candidats, qui ne sont pas forcément joignables. Deuxièmement, la plupart des tâches sont basées sur une expertise "humaine", effectuée manuellement. Or, si le nombre de candidats est trop élevé (par exemple, plus d'une centaine), ces tâches deviennent vite laborieuses. Par exemple, faut-il vraiment contacter plus de 100 vendeurs pour leur demander de répondre à un questionnaire, puis attendre leur réponse ? Et même si des centaines de candidats peuvent être traités efficacement par une équipe d'évaluateurs importante en nombre et bien organisée (comme c'était le cas pour l'exemple donné dans [164], que se passe-t-il si le nombre d'évaluateurs est restreint ?

2.3.4.8 DesCOTS

DesCOTS (*D*escription, *e*valuation and *s*election of *COTS* components), proposé par G. Grau, J. P. Carvallo, X. Franch et C. Quer [113], est un processus constitué de plusieurs outils. Chacun d'eux se charge d'une étape particulière de la sélection. Tout d'abord, *QM* [53] définit un modèle de qualité composé d'attributs qualité et de métriques à partir de techniques proposées par les auteurs [98, 8] (l'emploi de métriques de qualité pour l'évaluation des candidats est également préconisé par [258]). On obtient ainsi des critères d'évaluation non-fonctionnels. Ensuite, *EV* [238] utilise ce modèle de qualité ainsi qu'une taxonomie, définie elle aussi à partir de techniques proposées par les auteurs [55], afin d'évaluer les candidats. Après quoi *SL* utilise également le modèle de qualité ainsi que la taxonomie afin d'établir et de hiérarchiser les besoins que doivent satisfaire les candidats. Par exemple, on peut imposer que les candidats possèdent certains attributs qualité avec un niveau spécifique. À partir de ces besoins et des évaluations des candidats, un choix est fait afin de sélectionner le meilleur candidat. De plus, les modèles de qualité et les taxonomies définis précédemment sont stockés par l'outil *AD* afin de pouvoir être éventuellement réutilisés par les autres outils.

2.4 En résumé

Dans ce chapitre, nous avons présenté un état de l'art sur les processus de sélection composants, ainsi que sur les techniques de prise de décision multi-critères (MCDM) qu'ils utilisent.

De tous ces processus, deux se distinguent en particulier. OTSO est un travail pionnier dans la sélection de COTS. Il a innové aussi bien dans la définition des différentes étapes de cette sélection, que dans la préconisation de critères d'évaluation particuliers, ou dans le choix des méthodes de MCDM pour l'évaluation des candidats. PORE, quant à lui, est l'un des rares travaux à traiter simultanément de la spécification des besoins et de la progressivité de la sélection. La solution proposée est de combiner ces deux aspects dans un processus itératif, où le nombre de candidats s'amenuise au fur et à mesure que les besoins se précisent. Parmi les autres processus, certains s'attachent à redéfinir les différentes étapes de la sélection. D'autres

innovent dans la définition des critères, soit en proposant de nouvelles manières de les définir, soit en proposant un ensemble de critères particuliers.

Cependant, l'inconvénient majeur de tous ces processus réside dans leur manque d'automatisation. Si le score total des candidats est obtenu par le biais d'un algorithme ou d'une formule automatisable comme WSM, les scores d'évaluation locaux $score_{cj}$ pour chaque critère d'évaluation j sont eux estimés manuellement par l'utilisateur, et cela pour chaque composant candidat c. Or, même en se limitant à une seule bibliothèque, ou à une section particulière d'une bibliothèque, on se retrouve fréquemment avec un nombre de composants candidats dépassant la centaine. A titre d'exemple, la seule section "communication internet" de *ComponentSource* ([63]) comporte plus de 120 composants. Il est donc important de chercher autant que possible à automatiser le calcul des scores locaux à la fois pour gagner en précision mais également pour gérer de manière rentable la masse souvent très importante des informations à traiter. Dans le cas contraire, un nombre élevé de candidats et de critères devient vite rédhibitoire lors d'une évaluation conduite à la main ([207]).

Lorsque l'on cherche à automatiser ces calculs on constate qu'ils ne sont pas tous de même nature. La phase de pré-selection fait le plus souvent l'usage d'un nombre restreint de critères assez généraux (comme les mots clés). Les calculs locaux sont simples, mais ils s'appliquent à un nombre très important de candidats. Les techniques de recherche de composants en bibliothèque, dont l'objectif, après avoir formulé une certaine requête, est de retrouver tous les composants qui correspondent à cette requête ([192]), offrent des outils adaptés au calcul de ce type de score. Par exemple, la recherche par mots-clés ou la classification par facettes ([234]) peuvent permettre de filtrer un grand nombre de candidats sur la base d'une requête gros-grain. Le domaine de la recherche de composants en bibliothèque fait l'objet du chapitre 3.

À l'inverse, lors de la phase d'évaluation détaillée des candidats, les critères utilisés peuvent être très nombreux, complexes et axés sur des points de détail (comparaison de signatures de méthodes, comparaison d'interfaces, comparaison de valeurs de métrique, etc.). Les calculs sont beaucoup plus complexes, car ils nécessitent d'agréger de nombreuses valeurs de nature diverse, mais ils sont appliqués qu'à un nombre restreint de candidats. On peut utiliser dans ce cas des techniques de comparaison très fines, telles que le sous-typage ([51]). Pour évaluer les composants également selon leurs propriétés non-fonctionnelles, on peut décrire celles-ci au moyen de techniques telles que les contrats de qualité de service ([75]), ou user de modèles de qualité développés spécifiquement pour les composants sur étagère ([6]). La comparaison fonctionnelle et non-fonctionnelle de composants fait l'objet du chapitre 4.

Chapitre 3

Techniques et processus de recherche de composants

Sommaire

3.1 Introduction

Dans le chapitre précédent, nous avons présenté différentes techniques et processus de sélection de composants. Face à une bibliothèque contenant un grand nombre de composants, il est nécessaire d'avoir un mécanisme de recherche automatique qui permette au moins de retrouver les candidats les plus pertinents [192].

La recherche de composants en bibliothèque (*Component Search and Retrieval*) peut être considérée comme un cas particulier de recherche d'information [34, 255, 256, 46]. On appelle bibliothèque ou dépôt de composants (*component library* ou *repository*) une collection de composants, organisée d'une telle manière qu'on peut retrouver certains de ces composants à partir d'une requête particulière. Une requête (*query*) est une expression contenant des termes. Ces termes décrivent les besoins de l'utilisateur. Un terme peut être un mot, un diagramme, une signature, etc... [146].

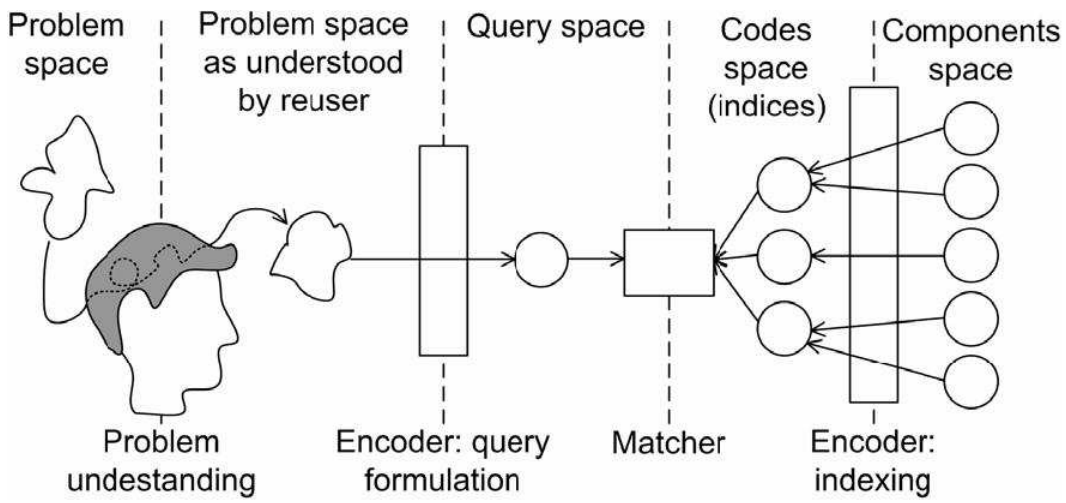

FIG. 3.1 – Principe de la recherche de composants en bibliothèque (d'après [192, 73]).

La figure 3.1 illustre le problème de la recherche de composants en bibliothèque tel qu'il a été défini dans [192]. Les problèmes auxquels l'utilisateur est confronté sont rassemblés dans un espace des problèmes (*Problem space* sur la figure). Cela peut être, par exemple, l'ensemble des besoins d'une application particulière, exprimés de manière informelle. L'utilisateur interprète chacun de ces problèmes à sa manière (*Problem understanding*) et il en résulte un espace des problèmes modifié. Cet espace peut être, par exemple, l'ensemble des services qui peuvent répondre à chacun des besoins exprimés dans l'espace précédent. Pour chaque problème identifié par l'utilisateur, il faut maintenant formuler une requête. Par exemple, l'opération qui fournira un service spécifique, identifié dans l'espace précédent. Le format dans lequel on exprime la requête peut être aussi abstrait qu'un mot ou un ensemble de mots, ou aussi complexe qu'une signature d'opération. Les requêtes ainsi formulées sont regroupées dans l'espace des requêtes (*Query space*).

Ensuite, on peut rechercher dans la bibliothèque (*Components space*) les composants qui correspondent aux requêtes formulées. Par exemple, les composants qui possèdent l'opération demandée dans la requête. Cependant, avant celà, il est nécessaire de "traduire" les composants afin d'obtenir un ensemble de descriptions appelé espace des indexes (*Codes space (indices)*). En effet, chaque description doit être de format identique à celui de la requête afin que la comparaison puisse se faire.

Pour cela, il faut trois mécanismes. Le premier est un "formulateur de requête" (*Encoder : query formulation*). Il se charge de spécifier les différentes requêtes dans un certain format. Le deuxième est un "indexeur" (*Encoder : indexing*). Il se charge d'extraire depuis les composants des descriptions dans le même format que celui de la requête. Le troisième est un "appariteur", ou *Matcher*. Une fois que la requête est formulée et que l'indexation est effectuée, il compare cette requête avec les descriptions des composants pour trouver celles qui correspondent. Supposons par exemple que le format de la requête soit une signature d'opération. L'indexeur doit donc extraire des composants les signatures de toutes les opérations que ceux-ci possèdent.

Ensuite, il faut choisir comme matcher un mécanisme de comparaison de signatures, afin de déterminer lesquelles correspondent à celle donnée en requête.

Dans ce chapitre, nous allons présenter les travaux en rapport avec la recherche de composants en bibliothèque. Certains travaux se concentrent sur la définition d'un format de requête, associé à une technique de matching. Ils font l'objet de la section 3.2. D'autres processus s'occupent également de l'indexation et de la classification des composants. Ils font l'objet de la section 3.3.

3.2 Techniques de matching

Dans cette section, nous allons présenter les travaux qui se concentrent sur le matching entre la requête et les descriptions de composants. On suppose que l'indexation est déjà faîte. Ces travaux se distinguent par le format dans lequel la requête est exprimée. Certains utilisent une représentation abstraite, comme les mots-clés. D'autres utilisent une représentation plus concrète, comme les signatures ou les spécifications des composants.

3.2.1 Matching sur une représentation abstraite

Un mécanisme de recherche de composants consiste à indexer, non pas la structure des composants directement, mais une représentation abstraite. Cela peut être par exemple, un ensemble de mots-clés, un ensemble de facettes, ou une description en langage naturel.

3.2.1.1 Recherche par mots-clés

La technique de classification externe la plus simple est la recherche par mots-clés. Y. Matsumoto [182] représente les composants par un ensemble de mots-clés, sur lesquels des recherches sont ensuite effectuées. E.-A. Karlsson [143] étend le procédé en associant un poids à chaque mot-clé. Cependant, avec ou sans poids, cette technique présente deux limites majeures. La première concerne la taille de la bibliothèque : quand elle est élevée, il est difficile de ne pas retrouver deux composants avec les mêmes mots-clés. On risque donc de retenir trop de candidats, ce qui nécessite un travail supplémentaire de classement. La deuxième limite concerne le domaine des composants. En effet, un même mot-clé peut être utilisé dans deux domaines différents avec deux significations différentes. Comment être sûr, dès lors, qu'un composant qui correspond à un mot-clé ne sera pas malgré tout totalement différent de ce qu'on attendait (autre domaine, autres fonctionnalités...) ?

3.2.1.2 Classification par facettes

Afin de rendre la recherche par mots-clés plus précise, R. Prieto-Diaz a proposé dans [233, 234] une approche à base de facettes pour classer les composants logiciels. Cette approche a été reprise depuis dans d'autres tavaux [231, 128, 303]. Une facette représente une information particulière qui permet d'identifier et de caractériser un composant. Elle est définie par son

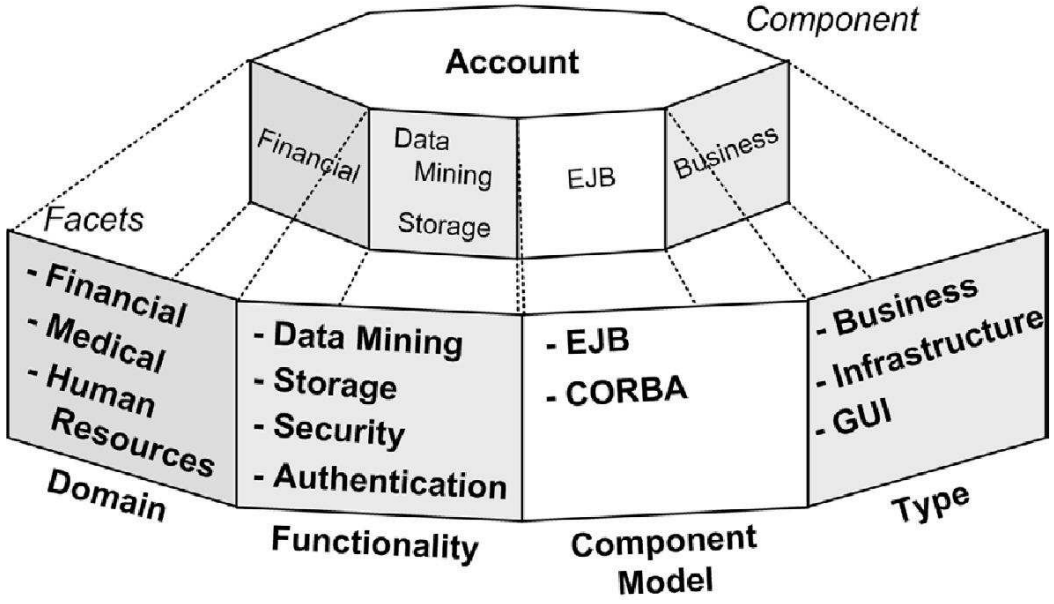

FIG. 3.2 – Classification par facettes (d'après [73]).

nom et son vocabulaire, c'est-à-dire l'ensemble des mots-clés qui permettent de la décrire. Pour décrire un composant, un ou plusieurs mots-clés doivent être choisis dans le vocabulaire de chaque facette.

La figure 3.2 montre un exemple de description de composant au moyen d'un ensemble de quatre facettes : *Domain*, *Functionality*, *Component Model* et *Component Type*. Les mots-clés *EJB* et *CORBA* du vocabulaire de la facette *Component Model* correspondent aux deux modèles de composants disponibles. De même, le domaine d'un composant peut être financier (*Financial*), médical, ou concerner les ressources humaines (*Human Resources*). La fonction (*Functionality*) d'un composant peut être : *Data Mining*, *Storage*, *Security* ou *Authentification*. Enfin, le *Type* d'un composant peut être : *Business*, *Infrastructure* ou *GUI*. Pour décrire le composant *Account* présenté sur la figure 3.2, qui représente un compte bancaire. *Financial* est le mot-clé choisi pour caractériser sa facette *Domain*, *Data Mining* et *Storage* ont été choisis pour la facette *Functionality*, *EJB* pour la facette *Component Model*, et *Business* pour la facette *Type*.

De cette manière, il est possible de décire un composant d'après ses différentes caractéristiques, pourvu qu'on choisisse bien les facettes et leur vocabulaire. Cependant, si cette technique réduit les difficultés soulevées précédemment, celles-ci restent présentes. D'une part, on peut toujours se retrouver avec plusieurs composants égaux s'il y a peu de facettes et beaucoup de candidats. D'autre part, le thésaurus choisi peut ne pas s'avérer pertinent.

3.2.1.3 Utilisation du langage naturel

La classification manuelle, par facettes, de composants est critiquée par des auteurs comme Y. Maarek, D. Berry et G. Kaiser [176]. D'une part, ils la jugent trop fastidieuse, surtout si on considère la taille croissante des bibliothèques. D'autre part, ils l'estiment trop subjective, car elle dépend du point de vue du "créateur" des facettes. Or ce point de vue est potentiellement différent de celui des utilisateurs. En d'autres termes, deux personnes différentes peuvent utiliser deux mots-clés différents pour désigner le même composant. Pour remédier à ce problème, Maarek, Berry et Kaiser ont proposé une approche basée sur l'indexation automatique des composants. Cette indexation se fait au moyen de techniques textuelles de recherche d'information [176]. Chaque composant est décrit en langage naturel, de même que les requêtes de l'utilisateur. Des techniques statistiques et linguistiques sont utilisées pour les analyser et en extraire des termes d'indexation. Le langage naturel a été utilisé dans d'autres travaux comme le système de recherche de composants LASSIE [77], ou la base de composants ROSA (*Reuse Of Software Artefact*) [109, 110, 108]. Avec une telle approche, aucun effort manuel n'est requis. Cependant, comme l'a noté Prieto-Diaz [234], les composants n'ont en général que très peu, voire pas du tout, de descripteurs en langage naturel.

3.2.2 Matching de signature

Les techniques de la sous-section précédente portaient sur une représentation abstraite des composants, c'est-à-dire leur documentation. Une autre technique consiste à utiliser une représentation concrète des composants, basée sur leur structure externe. Il s'agit du matching de signature. Ici, un composant n'est plus considéré comme un ensemble de mots-clés ou de facettes, mais comme un prestataire de services par le biais de ses opérations. Par conséquent, les requêtes se font sur les signatures de ces opérations [241].

Parmi les travaux qui se sont intéressés à cette approche, on peut citer ceux de M. Rittri [241, 242]. Celui-ci propose un algorithme qui considère les signatures d'opération comme des graphes. À partir d'une signature-requête, cet algorithme retrouve les opérations dont la signature correspond à cette requête, moyennant un isomorphisme de graphes. Le principe de cette approche a été repris dans [248, 64]. Cependant, le travail le plus avancé sur le matching de signature est celui tel qu'il a été défini par A. Zaremski et J. Wing [301].

Le matching de signature de Zaremski et Wing est basé sur la définition de type proposée par A. Field et P. Harrison [89]. Selon cette définition, un type est soit une variable de type $\in TypeVar$, soit un opérateur de type $\in TypeOp$ que l'on peut appliquer à d'autres types. Cet opérateur peut être, soit un opérateur prédéfini ($BuiltInOp$) soit un opérateur défini par l'utilisateur ($UserOp$). Partant de cette définition, l'égalité de type, notée $=_T$, est définie comme suit : deux types τ et τ' sont égaux si ce sont des variables de type lexicalement identiques, ou si ce sont des opérateurs de type qui ont les mêmes paramètres et le même résultat.

Afin de permettre la substitution, Zaremski et Wing ont défini la substitution de variables (*variable substitution*) comme suit : le type $[\tau'/\alpha]\tau$ résulte du remplacement par τ' de toutes les occurences de la variable de type α dans τ. Le remplacement peut s'effectuer à condition qu'aucune variable présente dans τ ne soit également présente dans τ'. Un exemple de substitution de

variable est : $[(int, int)/\beta](\alpha \rightarrow \beta) = \alpha \rightarrow (int, int)$. Si τ' est une variable, alors $[\tau'/\alpha]\tau$ est un renommage de variable (*variable renaming*). Dans ce cas, $\alpha, \tau' \in TypeVar$ ou $\alpha, \tau' \in UserOp$. Le renommage de variables n'est pas permis si $\alpha, \tau' \in BuiltInOp$. Zaremski et Wing ont également défini le réordonnancement (*reorder transformation*) de la manière suivante. Soit un type doté de $n-1$ paramètres. Notons-le : $\tau = (\tau_1, ..., \tau_{n-1}) \rightarrow \tau_n$. Un réordonnancement, noté T_α, définit une permutation α telle que $T_\alpha(\tau) = (\tau_{\alpha(1)}, ..., \tau_{\alpha(n-1)}) \rightarrow \tau_n$.

À partir de ces définitions, Zaremski et Wing ont défini un certain nombre de règles de matching. Ces règles s'appliquent sur deux signatures τ_l et τ_q. τ_l est le type d'une fonction appartenant à un composant dans une bibliothèque, tandis que τ_q un type entré comme requête. Les règles principales sont : le matching exact, le matching généralisé, le matching spécialisé, et le matching réordonné. τ_l matche exactement τ_q s'ils sont égaux au renommage de variables près. Par exemple, si $\tau_l = (\alpha, \alpha) \rightarrow bool$ et $\tau_q = (\beta, \beta) \rightarrow bool$, alors le premier type matche exactement le second. En effet, avec la substitution $V = [\beta/\alpha]$, on a bien l'égalité $V.\tau_l =_T \tau_q$. En ce qui concerne le matching généralisé, τ_l "généralise" τ_q si on peut retrouver τ_q à partir de τ_l *via* une séquence de substitutions de variables. Par exemple, $\alpha \rightarrow \alpha$ est une généralisation de plusieurs types, tels que $int \rightarrow int$ et $(int, \beta) \rightarrow (int, \beta)$, avec les substitutions de variables respectives $[int/\alpha]$ et $[(int, \beta)/\alpha]$. Dans le matching spécialisé, à l'inverse, c'est τ_q qui généralise τ_l. Enfin, il y a matching réordonné quand il existe une permutation des paramètres du type τ_l telle que celui-ci matche exactement avec τ_q. Par exemple, $(int, \alpha) \rightarrow int$ et $(\alpha, int) \rightarrow int$ peuvent matcher, moyennant une permutation de *int* et de α. Il est également possible de composer plusieurs matchings.

Le matching de signature de Zaremski et Wing est l'un des travaux les plus complets sur la question. Il offre un certain nombre de règles qui le rendent beaucoup plus flexible que le sous-typage. Cependant, son utilisation dans le contexte des marchés aux composants suppose qu'on connaisse l'ensemble des substitutions de variables possibles entre tous les systèmes de types utilisés par les différents modèles de composant. Sans une telle connaissance, la vérification des règles de matching nécessite un important travail manuel.

3.2.3 Matching de spécification

Outre les signatures d'opération, qui constituent la structure externe d'un composant, le matching peut porter sur des spécifications. Celles-ci informent non pas sur la syntaxe, mais sur le comportement du composant. Ces spécifications peuvent être exprimées sous forme logique ou par des contrats (invariants, pré- et post-conditions). Elles peuvent également être exprimées de manière plus formelle, sous forme d'ensembles et de relations. Nous allons présenter ici quelques-uns de ces travaux.

3.2.3.1 Spécifications comportementales

Le matching de spécification d'A. Zaremski et J. Wing [302] est le prolongement de leurs travaux sur le matching de signature, présenté dans la section précédente. D'ailleurs, le premier présuppose le succès du second. À chaque opération on associe un ensemble de pré- et post-conditions qui représentent son comportement.

Soit S=(S_{pre}, S_{post}) un couple "(ensemble de pré-conditions, ensemble de post-conditions)" représentant la spécification d'un composant dans une bibliothèque, et Q=(Q_{pre}, Q_{post}) une spécification entrée comme requête. Alors le matching de spécification est défini par la formule suivante :

$$Match(S,Q) = (Q_{pre} R_1 S_{pre}) R_2 (S_{post} R_3 Q_{post}) \tag{3.1}$$

Où R_1, R_2 et R_3 sont des connecteurs logiques tels que $\Rightarrow$ ("implique"), $\Leftrightarrow$ ("équivaut") ou $\wedge$ ("et").

À partir de là, Zaremski et Wing définissent un certain nombre de règles possibles de matching tels que le matching exact (R_1=R_3="$\Leftrightarrow$" et R_2="$\wedge$"), ou le matching *plug-in* (R_1=R_3="$\Rightarrow$" et R_2="$\wedge$") qui équivaut à la règle de sous-typage comportemental sur les pré-conditions plus faibles et les post-conditions plus fortes [172].

Bien qu'il existe d'autres travaux voisins sur le sujet, tels que ceux de J.-J. Jeng et B. Cheng [135], ceux de D. Hemer et P. Lindsay [118], ou ceux de J. Penix et P. Alexander sur la théorie de domaine [224, 222, 223], le matching de spécification de Zaremski et Wing est l'un des plus avancés en ce qui concerne les règles de matching. Cependant, le défaut principal de ce matching réside dans le fait que plusieurs règles nécessitent d'être vérifiées manuellement. Il est très délicat, par exemple, de savoir si une pré-condition d'un composant décrit dans un certain format implique une pré-condition d'un autre composant décrit dans un format différent.

3.2.3.2 Spécifications formelles

Plutôt que de représenter le comportement des composants et des requêtes par des spécifications de type pré- et post-conditions, A. Mili, R. Mittermeir *et al* ont préféré utiliser les spécifications relationnelles [37, 191, 193]. Une relation est un sous-ensemble du produit cartésien. C'est-à-dire un ensemble de couples (s,s'), avec s étant "l'argument" de la relation et s' son "image". La spécification relationnelle R d'un composant contient donc tous les couples *(entrée, sortie)* considérés comme corrects. Dit autrement, pour tout s faisant partie des arguments de R, le composant produit en sortie une image s' telle que $(s,s') \in R$.

Un exemple de spécification relationnelle tiré de [139] est celui d'un compilateur Pascal. Soient $cor(x,y)$ la propriété selon laquelle le code y compilé à partir du code source x est correct, et $opt(y)$ la propriété selon laquelle le code y est optimisé. Alors la spécification relationnelle d'un compilateur Pascal optimisé et prenant en charge tous les programmes syntaxiquement corrects est : $Comp$=$\{(s,s') \mid cor(s,s') \wedge opt(s')\}$.

À partir de là, Mili *et al* définissent un certain nombre de relations ensemblistes sur ces spécifications relationnelles, notamment le "raffinement", noté $\sqsupseteq$. Une spécification relationnelle R raffine une autre spécification relationnelle R' (noté $R \sqsupseteq R'$) si et seulement si n'importe quel couple *(entrée,sortie)* correct selon R est également correct selon R'. Dit autrement, $R \sqsupseteq R' = ((s,s') \in R) \Rightarrow ((s,s') \in R')$. On retrouve là le principe de substitution du typage classique transposé aux spécifications relationnelles. Une autre définition du raffinement de R' par R est que R doit posséder au moins les arguments et au plus les images de R'. On retrouve ici le principe du sous-typage comportemental, déjà présent dans le matching *plug-in* d'A. Zaremski et J. Wing [302]. Le raffinement peut être considéré comme un sous-typage de spécifications relationnelles.

Les autres relations ensemblistes entre spécifications relationnelles sont : l'union relationnelle (appelée *join* et notée $\sqcup$), l'intersection relationnelle (appelée *meet* et notée $\sqcap$), la différence relationnelle aussi appelée différence de raffinement (notée $\ominus$) et la différence de raffinement symétrique (notée $\otimes$). Les trois premières relations sont les equivalents respectifs de l'union, de l'intersection et de la différence ensemblistes. La différence de raffinement symétrique, quant à elle, est égale à la différence de raffinement entre l'union relationnelle et l'intersection relationnelle. Formellement parlant, $R \otimes R' = (R \sqcup R') \ominus (R \sqcap R')$.

L. Labed Jilani, A. Mili *et al* se sont servis de cette représentation de composants sous forme de spécifications relationnelles afin de mesurer efficacement la distance entre composants [138, 139]. Une distinction est faite entre la distance dite "structurelle" ou "syntaxique", et la distance dite "fonctionnelle" ou "sémantique". La distance structurelle mesure la différence entre la structure (type, signature...) des composants. La distance fonctionnelle, quant à elle, mesure la différence entre leur comportement. Par exemple, entre une opération d'addition et une opération de soustraction portant toutes deux sur des nombres réels, la distance structurelle est faible tandis que la distance fonctionnelle est importante. À l'inverse, entre une liste d'entiers et une liste de booléens gérées toutes les deux de la même manière, la distance fonctionnelle est faible tandis que la distance structurelle est importante. Pour leur mesure de distance entre composants, les auteurs ont préféré considérer la distance fonctionnelle. Les raisons invoquées sont la diversité des formats de composants existants, ainsi que la prédominance des composants COTS, qui rend difficile la mesure d'une distance structurelle.

Pour mesurer la distance fonctionnelle entre deux composants, les auteurs utilisent les relations ensemblistes régissant les spécifications relationnelles afin de définir des formules mesurant chacune un aspect particulier de cette distance. Pour prendre l'exemple qui illustre leur travail, celui de l'intégration d'un composant COTS qui requiert le minimum d'effort d'adaptation [139], ils ont sélectionné quelques mesures de distance entre une requête K et un composant candidat C. Parmi ces mesures, on trouve le déficit fonctionnel ϕ, qui indique ce qui doit être ajouté à C pour que son comportement satisfasse K, et qui est égal à : $\phi(K,C)=K \ominus (K \sqcap C)$. On trouve aussi l'excédent fonctionnel ξ, qui indique les aspects de C sans rapport avec K, et qui est égal à : $\xi(K,C)=C \ominus (K \sqcap C)$. On trouve également la distance de raffinement ρ, qui indique à la fois le déficit fonctionnel et l'excédent fonctionnel, et qui est égal à : $\rho(K,C)=\phi(K,C) \sqcup \xi(K,C)$. On considère qu'un candidat C est plus proche de la requête K qu'un autre candidat C' si et seulement si $\rho(K,C') \sqsupseteq \rho(K,C)$.

L'utilisation de telles spécifications relationnelles repose cependant sur de nombreux présupposés. En particulier, les spécifications relationnelles de chaque composant candidat doivent être connues. De même, les propriétés sur les relations, telles que $cor(x,y)$ et $opt(y)$ dans l'exemple du compilateur Pascal, doivent toutes être connues et décrites dans le même format détaillé précédemment. Cela sous-entend que les constructeurs ont tous écrit au préalable les spécifications relationnelles de leurs composants selon ce format. Ou bien qu'ils ont documenté le comportement de leurs composants de manière assez précise pour que les utilisateurs puissent faire eux-mêmes la traduction. Or, pour prendre l'exemple des marchés aux composants tels que *ComponentSource* [63], il est très rare que le comportement des composants soit documenté. Et même s'il est beaucoup plus fréquent que la structure des composants (opérations, signatures...) apparaisse dans leur documentation, il n'y a pas de consensus sur la manière de documenter.

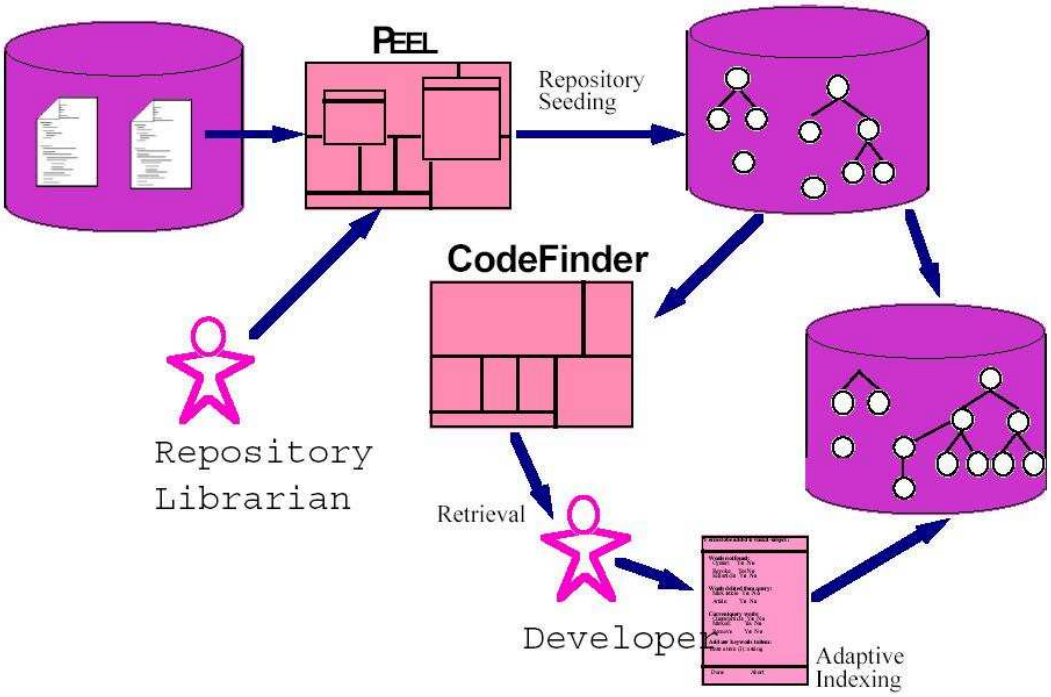

FIG. 3.3 – Approche de Henninger (d'après [120]).

3.3 Recherche avec extraction et classification de composants

Plutôt que de faire une seule recherche directement sur toute la bibliothèque de composants, d'autres techniques de recherche utilisent des bibliothèques intermédiaires. Un exemple simple est l'approche de T. Isakowitz et R. Kaufmann [128]. Dans une première étape, une requête initiale est formulée au moyen de facettes et une première liste de candidats est renvoyée. Dans une deuxième étape, les composants retenus sont inspectés au moyen d'une navigation hypertexte, afin de trouver celui qui correspond le mieux.

D'autres travaux utilisent des processus plus élaborés, qui offrent également des mécanismes d'indexation. Certains d'entre eux classent les composants au moyen de poids [119, 120, 126, 127]. D'autres mettent à jour plusieurs filtres afin d'avoir une meilleure connaissance de la bibliothèque [298]. Dans tous les cas, les "composants" sont extraits depuis divers documents provenant en général d'une bibliothèque initiale. Une fois extraits, les composants sont classés dans un format compatible avec la requête. Dans cette section, nous allons voir en détail certaines de ces approches mélangeant extraction et filtrage.

3.3.1 *PEEL* et *CodeFinder*

S. Henninger [119, 120], a observé une distance entre le "quoi" (les composants qu'on recherche) et le "comment" (la manière dont ils sont décrits). Il a également relevé certains

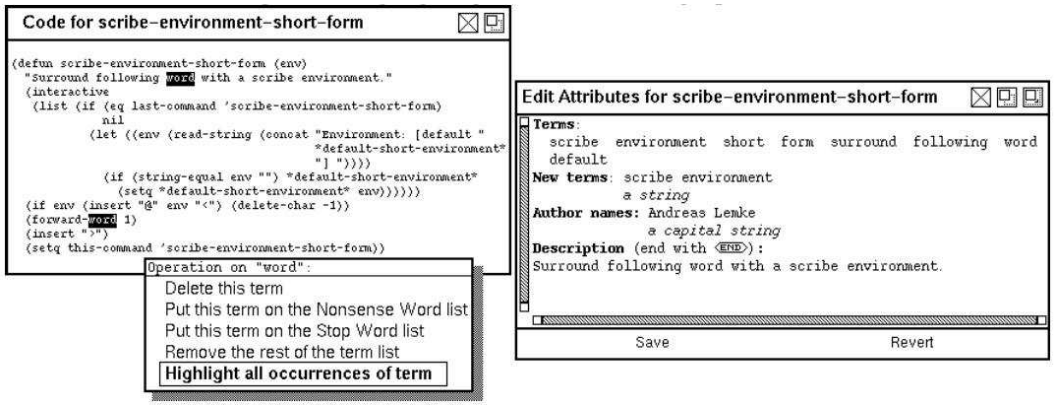

FIG. 3.4 – PEEL : Extraction de composants (d'après [120]).

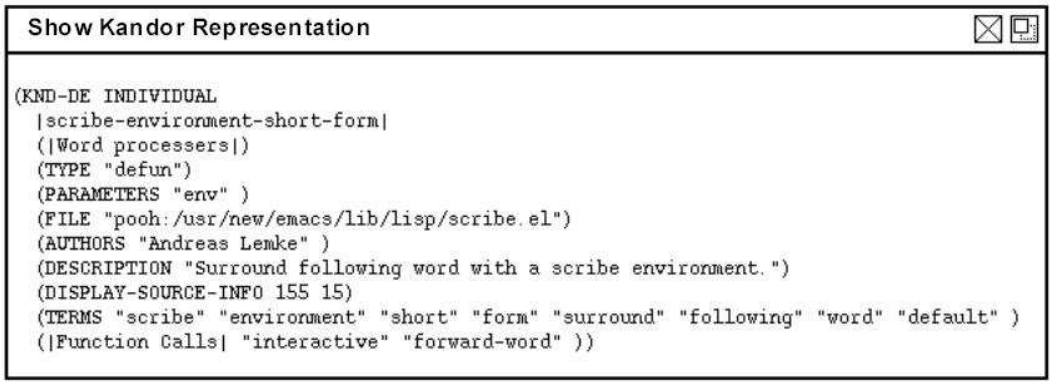

FIG. 3.5 – PEEL : Spécification d'un composant en langage Kandor (d'après [120]).

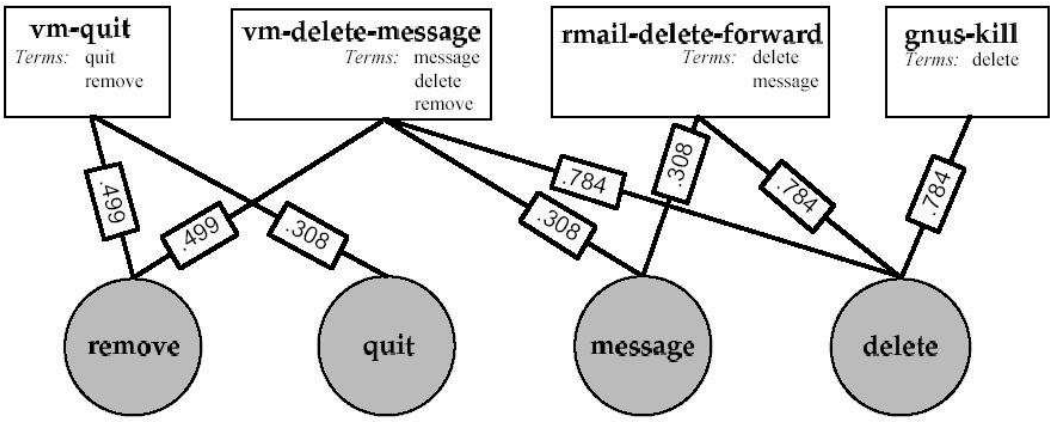

FIG. 3.6 – Un réseau de composants dans *CodeFinder* (d'après [120]).

problèmes liés aux algorithmes de recherche de composants. En effet, ces algorithmes nécessitent souvent des structures prédéfinies de classification (en catégories, sous-catégories, facettes...). Or, selon Henninger, ces structures sont généralement lourdes, coûteuses et incapables de s'adapter à un changement de contexte. Il en a déduit qu'il fallait pouvoir reformuler la requête et faire évoluer la bibliothèque dans laquelle on déposerait les composants pour une future évaluation. À partir de là, il a conçu deux outils afin de valider son approche dans le cadre des composants "boîte-blanche", dont le code source est accessible (figure 3.3).

Le premier outil, *PEEL* (*Parse and Extract Emacs Lisp*), extrait les "composants" depuis un fichier texte contenant du code source. Un utilisateur "libraire" (le *repository librarian* de la figure 3.3) peut ensuite les déposer dans une bibliothèque. Comme son nom complet l'indique, *PEEL* fonctionne avec *Emacs Lisp*, une extension de l'éditeur GNU-Emacs [267, 49]. Ici, les composants sont des fonctions Lisp. L'extraction des composants se fait en trois étapes, selon un processus similaire à celui de G. Caldiera et V. Basili [48]. Dans la première étape, les termes sont extraits automatiquement à partir du nom de la fonction et des commentaires qui suivent la définition de cette fonction. Comme le montre la partie gauche de la figure 3.4, pour chaque terme extrait, l'utilisateur a le choix entre plusieurs options. Il peut conserver ce terme, l'enlever, l'ajouter à la liste des mots qui n'ont pas de sens (*Nonsense Word list* sur la figure 3.4), ou l'ajouter à la *Stop Word List*. Cette liste, inspirée par les travaux de C. Fox sur l'analyse lexicale [94], stocke les mots à ne pas extraire tels que "and", "of" ou "the". La deuxième étape, à droite de la figure 3.4, consiste à montrer les termes ainsi extraits à l'utilisateur afin qu'il les vérifie et enlève ceux qui ne lui conviennent pas. Après quoi, la troisième étape consiste à le laisser ajouter ses propres termes et phrases. Enfin, l'extraction étant achevée, l'ensemble des termes extraits est traduit en langage Kandor [221] pour avoir la spécification d'un composant (figure 3.5).

Le deuxième outil, *CodeFinder*, a pour but de permettre au développeur (*Developer* sur la figure 3.3) de lancer une recherche "intelligente" sur la bibliothèque de composants créée précédemment. Pour ce faire, comme le montre la figure 3.6, *CodeFinder* crée un réseau entre les termes (les nœuds en forme de rond sur la figure) et les composants qui les contiennent (les nœuds en forme de rectangle). Chaque arc a un poids en fonction du terme auquel il est rattaché. Ce poids est donné en fonction de plusieurs paramètres détaillés dans [120]. Quand le développeur lance une recherche sur un mot, le nœud correspondant envoie sa valeur aux composants auxquels il est lié. Ensuite, ces composants "activent" les autres termes auxquels ils sont connectés en propageant leur valeur. Chaque nœud tient compte de la valeur et le poids de ses prédécesseurs. Pour reprendre l'exemple de la figure 3.6, une recherche sur le mot *remove* envoit une valeur de .499 sur les composants *vm-quit* et *vm-delete-message*. Ces composants "activent" ensuite les nœuds *quit*, *message* et *delete* en propageant leur valeur modifiée par le poids des arcs (.308 pour *quit* et *message*, et .784 pour *delete*), selon les formules établies dans [120]. Le composant sélectionné est celui qui récupère la valeur la plus élevée.

L'originalité de ce couple d'outils réside dans l'utilisation des poids pour classer les composants selon la requête. Cependant, la recherche se fait sur un seul niveau de granularité : les composants sont perçus exclusivement comme des ensembles de mots-clés, et sont extraits manuellement.

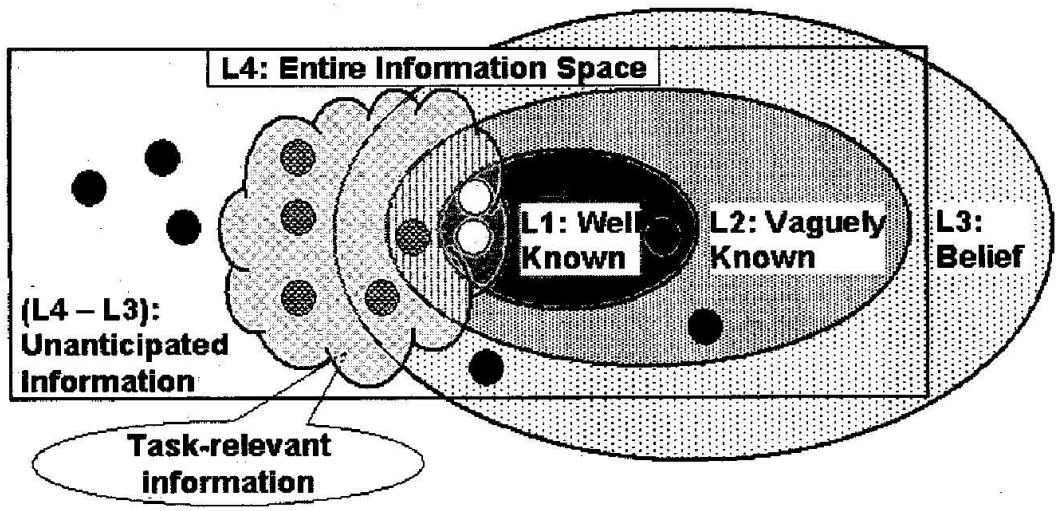

FIG. 3.7 – Problème de la connaissance de la bibliothèque (d'après [298]).

3.3.2 *CodeBroker*

Y. Ye et G. Fischer [298] ont constaté que la réutilisation logicielle n'a pas eu le succès escompté, parce que les conditions préalables à son fonctionnement n'étaient pas satisfaites. En effet, une bonne réutilisation nécessite d'une part, une bibliothèque de composants alimentée par des constructeurs, et d'autre part des développeurs qui sont à même de puiser dans cette bibliothèque pour concevoir leurs logiciels. En bref, il faut une volonté des constructeurs de composants d'investir dans la réutilisation, et une volonté des développeurs de réutiliser les composants. Or si les développeurs n'essaient pas de réutiliser ceux qui sont à leur disposition, les constructeurs vont cesser d'investir dans la réutilisation et d'alimenter les bibliothèques. Et réciproquement, si les investissements cessent, les développeurs n'auront plus rien à réutiliser. L'origine de l'impasse, selon Ye et Fischer, est que si les développeurs n'essaient pas de réutiliser les composants disponibles [96], c'est parce qu'ils n'arrivent pas à retrouver dans une bibliothèque ceux qui correspondent à leur besoin [162, 97, 133]. Et s'ils n'arrivent pas à les retrouver, c'est parce qu'ils ne connaissent pas assez la bibliothèque pour cela [92, 297].

La figure 3.7 illustre cette distance entre le manque de connaissance et la disponibilité effective des "bons" composants [93]. Sur la figure, le rectangle qui symbolise le niveau de connaissance $L4$ désigne l'ensemble des composants de la bibliothèque (*Entire Information Space*). Le niveau $L3$ désigne les composants que le développeur croit connaître (*Belief*), sachant qu'une partie existe vraiment, et que l'autre, celle qui déborde du rectangle, n'existe pas. À l'intérieur de la partie de $L3$ qui se trouve dans le rectangle, le niveau $L2$ désigne les composants que le développeur connaît au moins un peu (*Vaguely Known*). Et à l'intérieur de $L2$, le niveau $L1$ désigne les composants que le développeur connaît bien (*Well Known*). Enfin, $L4$-$L3$ représente l'ensemble des composants que le développeur ne connaît pas du tout (*Unanticipated Information*). En général l'ensemble des composants qui pourraient répondre au besoin du développeur comprend des composants de chaque niveau : $L1$, $L2$, $L3$ (la partie qui se trouve

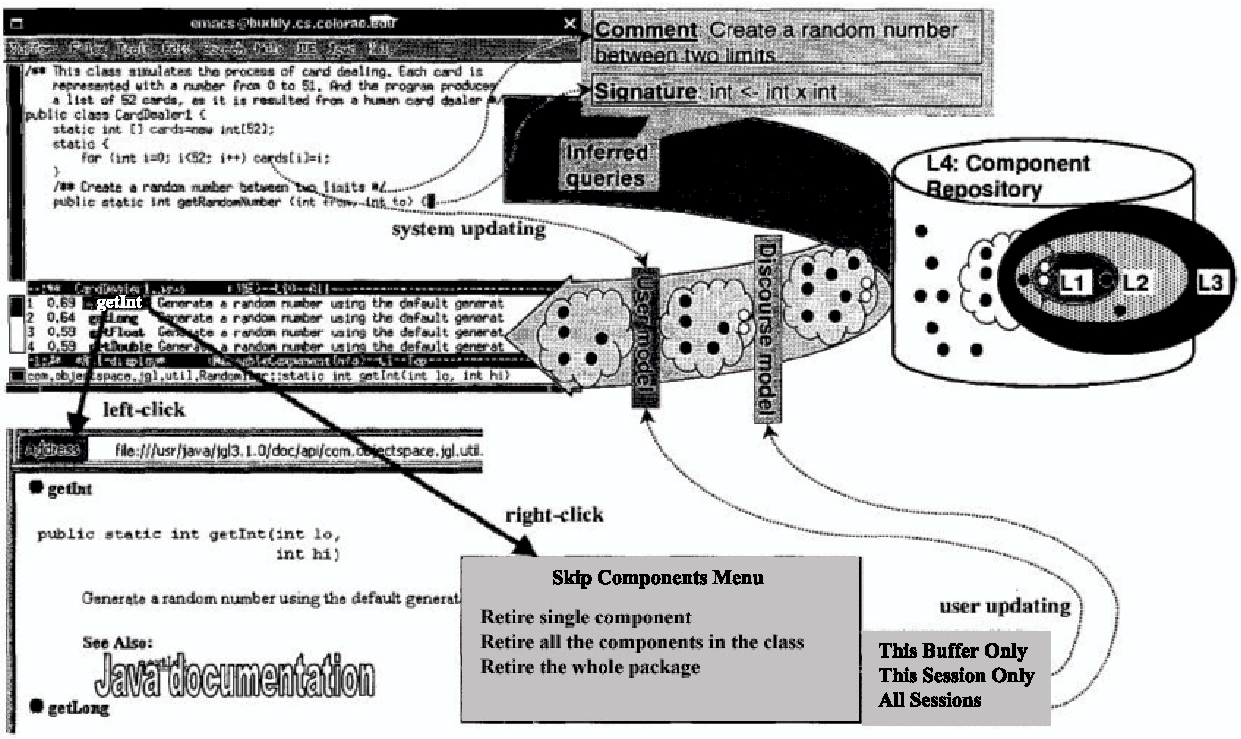

FIG. 3.8 – Architecture et fonctionnement de *CodeBroker* (d'après [298]).

dans le rectange) et *L*4-*L*3. Or, le développeur est susceptible de choisir uniquement la poignée d'entre eux qu'il connaît bien et qui se trouve dans le niveau *L*1.

Afin de réduire cette distance, Ye et Fischer ont conçu l'outil *CodeBroker* [298]. Cet outil recherche des "composants" depuis des fichiers Java chargés sous *Emacs* [267], tout en écartant ceux qui ne conviennent pas et ceux qu'on connaît déjà (figure 3.8). Ici, les composants sont des méthodes Java. Les requêtes possibles sont extraites automatiquement depuis la signature et la *Javadoc* des composants-méthodes disponibles. Lors d'une requête, les composants susceptibles de correspondre sont affichés en bas de l'écran, après avoir être passés par deux filtres : le premier (*DiscourseModel* sur la figure 3.8) enlève de la liste des candidats les composants qui ne correspondent pas, tandis que le deuxième, (*UserModel* sur la figure) enlève les composants que l'on connaît déjà, ou que l'on a déjà utilisés. Ces deux filtres peuvent être enrichis par l'utilisateur au moyen du menu `Skip Components Menu`, auquel il peut accéder par un clic droit sur le composant. Il peut ensuite retirer le composant, ou tous ceux contenus dans la classe, ou encore tous ceux contenus dans le paquetage. À partir du `Skip Components Menu`, trois commandes permettent à l'utilisateur de retirer les composants et d'enrichir les filtres (en bas à droite de la figure 3.8) : `This Buffer Only` retire le composant de la requête, `This Session Only` le retire de la requête pour en faire un composant indésirable et l'ajouter au *Discourse Model*, et `All Sessions` le retire de la requête pour l'ajouter au *User Model*. Un clic gauche,

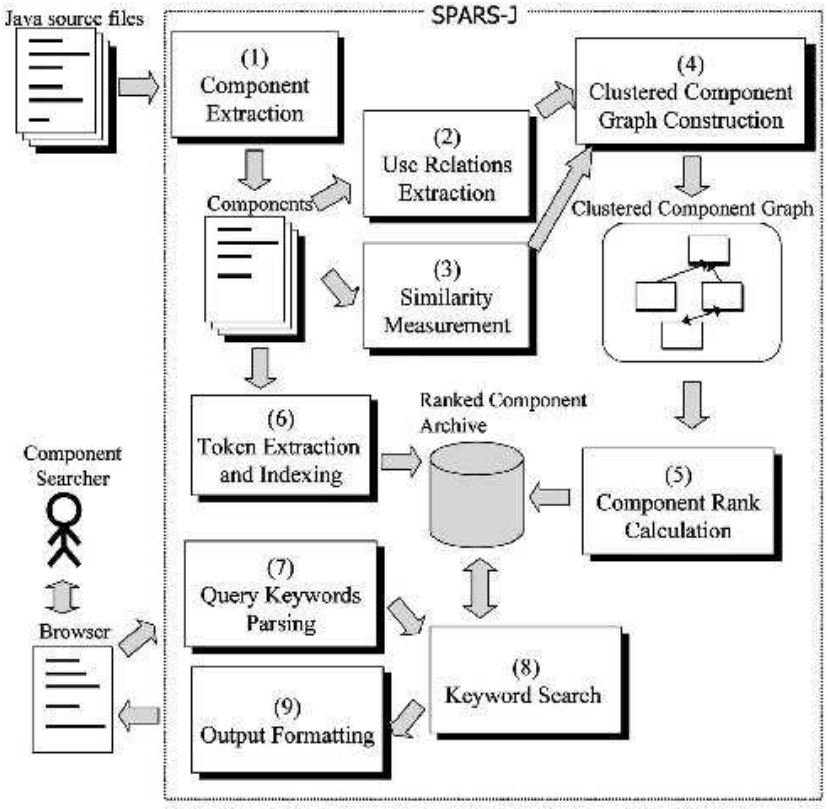

FIG. 3.9 – Architecture de *SPARS-J* (d'après [127]).

en revanche, permet à l'utilisateur de consulter la *Javadoc* du composant.

L'originalité de *CodeBroker* réside dans les deux "bibliothèques-filtres", qui permettent de se concentrer sur les composants de la bibliothèque qu'on ne connaît pas encore. De plus, l'indexation se fait aussi bien sur la documentation que sur les signatures. Cependant, le remplissage des bibliothèques-filtres est manuel, et il n'y a aucun classement des composants qui répondent à la requête.

3.3.3 *ComponentRank* et *SPARS-J*

K. Inoue *et al.* [126, 127] ont voulu mettre au point l'équivalent de *Google* [40, 111] pour les composants. Plus précisément, le but de leur approche est, suite à une recherche, de classer les candidats retenus par ordre de pertinence. Pour ce faire, Inoue *et al.* ont mis au point le modèle *ComponentRank* [126], ainsi que l'outil *SPARS-J* (*Software Product Archiving, analyzing and Retrieving System for Java*) [127] qui l'utilise, et dont l'architecture est montrée figure 3.9. *ComponentRank* est basé sur une modélisation des systèmes logiciels sous forme de graphe orienté et valué, appelé "graphe de composant" (*Component Graph*). Les nœuds du graphe sont donc des "composants" qui, dans le cas de *SPARS-J*, sont des fichiers Java collectés

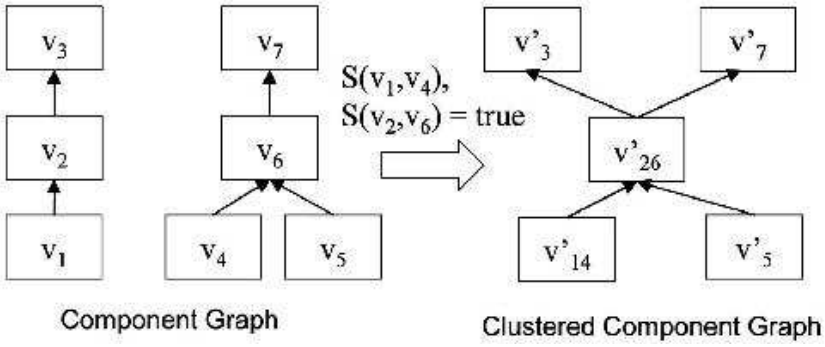

FIG. 3.10 – *ComponentRank* : Graphe de composant et sa version clusterisée (d'après [127]).

à partir de paquetages. Cette collecte est la première étape du processus de *SPARS-J*, comme le montre la figure 3.9. Un arc dirigé e_{xy} allant d'un nœud x à un nœud y représente une relation d'utilisation (*use relation*) qui signifie que le composant x utilise le composant y.

Un même graphe de composant peut modéliser un ou plusieurs systèmes logiciels, qu'ils soient reliés ou pas. La figure 3.10 à gauche montre l'exemple d'un graphe de composant constitué de deux systèmes logiciels, l'un contenant 3 composants (v_1, v_2, v_3), tandis que l'autre en contient 4 (v_4, v_5, v_6 et v_7). On remarque que v_1 utilise v_2 qui utilise v_3, tandis que v_4 et v_5 utilisent v_6 qui utilise v_7. L'extraction de ces relations d'utilisation est la deuxième étape du processus de *SPARS-J* (figure 3.9).

Après avoir extrait les relations d'utilisation, la troisième étape du processus de consiste à mesurer la similarité entre composants. Cette mesure de similarité consiste à éviter de se retrouver avec des candidats ayant le même rang. Dans *SPARS-J*, il s'agit d'une similarité entre leurs codes sources. Elle est mesurée au moyen de métriques et d'un outil *ad hoc* mis au point par les auteurs [150]. Une fois la deuxième et la troisième étapes franchies, la quatrième étape (figure 3.9) consiste à "clusteriser" le graphe de composant. Cela signifie que quand un graphe est constitué de plusieurs sous-graphes disjoints, ceux-ci fusionnent jusqu'à ce qu'il n'y ait plus de sous-graphe disjoint des autres. Par exemple, sur la figure 3.10, v_1 et v_4 sont jugés similaires, tout comme v_2 et v_6. Ils peuvent donc fusionner pour obtenir le graphe clustérisé à droite de la figure.

La cinquième étape consiste à distribuer les poids dans le graphe "clusterisé". Chaque nœud v d'un graphe de composant possède un poids $w(v)$ allant de 0 à 1. Ce poids symbolise l'influence d'un composant dans le graphe. Les composants qui ont les poids les plus élevés seront par la suite les premiers à être affichés lors d'une recherche de composants. À l'intérieur d'un même graphe, la somme des poids de tous les nœuds vaut 1. Chaque arc e_{ij} allant d'un nœud v_i vers un nœud vj a aussi un poids $w'_{ij} = d_{ij} \times w(v_i)$ (figure 3.11 à gauche), où d_{ij} est un pourcentage de distribution allant de 0 à 1. Pour tout nœud v_i, la somme de tous les d_{ij} est égale à 1. Comme le montre la figure 3.11 à gauche, le poids d'un nœud v_j est égal à la somme de tous les poids de ses arcs entrants. La figure 3.11 montre à droite un exemple de graphe de composant valué. La somme des poids des nœuds v_1, v_2 et v_3 est bien égale à 1. Pour le nœud v_1, $d_{12} + d_{13} = 0.5 + 0.5 = 1$, tandis que pour les arcs e_{12} et e_{13}, $w'(e_{12}) = w'(e_{13}) =$

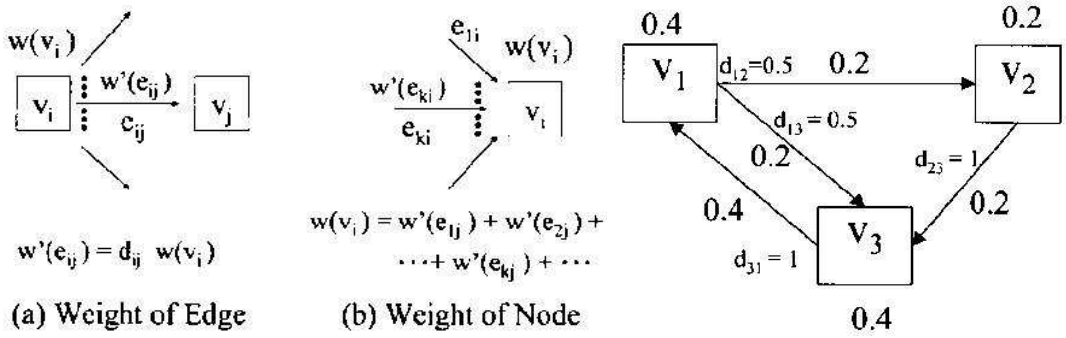

FIG. 3.11 – Valuation d'un graphe : théorie et exemple (d'après [126]).

$0.5 * w(v_1) = 0.2$. Et pour le nœud v_3 , $w(v_3) = w'(e_{13}) + w'(e_{23}) = 0.2 + 0.2 = 0.4$.

Une fois que chaque composant a reçu son poids, il est classé en fonction dans la bibliothèque (*Component Rank Archive* sur la figure 3.9). Une sixième étape du processus de *SPARS-J* consiste à extraire divers mots-clés (ou *tokens* sur la figure) depuis les composants. Ces *tokens* serviront à les indexer dans l'archive. Les trois dernières étapes du processus (en bas à gauche de la figure 3.9) concernent les requêtes de l'utilisateur. La septième étape consiste à parser les mots-clés entrés en requête. Dans la huitième étape ces mots-clés sont comparés aux *tokens* qui indexent les composants dans l'archive. Enfin, la neuvième et dernière étape du processus renvoie à l'utilisateur les résultats de la recherche, c'est-à-dire l'ensemble des composants retenus et classés selon leur rang.

L'originalité des outils d'Inoue *et al* réside dans la manière d'attribuer les poids aux différents composants. Cependant, bien que ceux-ci puissent être d'une nature quelconque, la recherche qui est faite sur eux est toujours à base de mots-clés. Il n'y a donc qu'un seul niveau de granularité qui est considéré.

3.4 En résumé

Dans ce chapitre, nous avons présenté différents mécanismes de recherche de composants dans une bibliothèque. Certains de ces mécanismes sont basés sur une représentation externe des composants, comme les mots-clés, les facettes, ou le langage naturel. D'autres, plus précis, sont basés sur la signature des opérations que possèdent les composants, ou sur la spécification de ces composants. D'autres encore vont plus loin en proposant des processus permettant l'indexation et la classification des composants candidats au moyen de poids et de bibliothèques intermédiaires. Les candidats qui correspondent à la requête sont donc filtrés progressivement. Ces mécanismes présentent tous au moins deux défauts.

Leur premier défaut est de ne considérer qu'une seule représentation du composant. À "haut niveau", on utilise les mots-clés de la documentation ou les facettes. À "bas niveau", on utilise les signatures d'opération ou les spécifications. Mais il n'y a qu'une seule représentation

par technique de recherche. Or, certains niveaux ne sont adaptés que pour certaines tailles de bibliothèques. Par exemple, il serait fastidieux d'appliquer les règles de matching de signature en présence d'un millier de composants, chacun ayant plusieurs opérations. De plus, même les techniques qui utilisent des bibliothèques-filtres ne prennent en compte qu'un seul niveau, une représentation, et n'en changent pas même pour filtrer les candidats. Or, il vaut mieux combiner les différentes représentations des composants afin de raffiner progressivement les requêtes [145, 146]. On pourrait ainsi appliquer une technique de recherche à haut niveau afin de pré-sélectionner les candidats, puis une technique de recherche à bas niveau sur les quelques composants restants. Outre le matching de signature, des techniques existent pour comparer les composants de manière détaillée. Certaines de ces techniques sont étudiées dans la première partie du chapitre 4.

Le deuxième défaut de ces mécanismes est de ne pas prendre en compte les propriétés non-fonctionnelles. En effet, les techniques de recherche de composants ne se focalisent que sur le respect d'une représentation ou d'une signature particulière, mais pas sur la qualité. D'un point de vue fonctionnel, certaines techniques comme le matching de signature se basent sur une description détaillée des composants. Il devrait être possible d'utiliser d'utiliser des descriptions aussi détaillées d'un point de vue non-fonctionnel, et même de combiner les deux points de vue. Ces techniques de description non-fonctionnelle sont étudiées dans la deuxième partie du chapitre 4.

Chapitre 4

Techniques de comparaison entre composants

Sommaire

4.1 Introduction

Dans le chapitre précédent, nous avons présenté un état de l’art des techniques de recherche de composants. Ces approches peuvent être utilisées lors de la recherche et de l’identification des candidats potentiels. Il reste le problème de l’automatisation de l’évaluation de ces candidats. Comme on veut représenter le besoin d’une application par un composant recherché, il est nécessaire de lui comparer les composants candidats. Comme expliqué dans le chapitre 1, cette comparaison doit se faire sur la base des propriétés fonctionnelles et non-fonctionnelles des composants.

Les travaux que nous allons présenter dans ce chapitre sont en rapport avec la comparaison de composants. Cette comparaison peut consister en une substitution d’un composant par un autre (cf. section 4.2), soit par un mécanisme basique de sous-typage, soit par une méthode plus avancée qui prend en compte le contexte de composition. Comme il s’agit ici d’une comparaison fonctionnelle, il faut également s’intéresser à ce qui existe comme techniques de description et de comparaison des propriétés non-fonctionnelles. C’est l’objet de la section 4.3

4.2 Modèles de substitution de composants

F. DeRemer et H. Kron, en créant en 1975 le premier langage d'interconnexion de modules [76], avaient prévu la possibilité de substituer un module par un autre, bien qu'il ne fût pas encore question de typage. Par la suite, dès 1988, P. Wegner et S. Zdonik ont défini dans [290] le principe de substitution pour les composants de la manière suivante : *"un composant sous-type (ou composant de remplacement) devrait être utilisable partout où un composant supertype (le composant courant) est attendu, sans que le client le remarque"*. Depuis, comme l'a fait remarquer P. Brada [38], peu de travaux se sont penchés sur une substitution spécifique aux composants. Après avoir discuté du sens que peut avoir une "substitution de composants", nous allons présenter quelques travaux parmi les plus avancés. Cela va du sous-typage hétérogène adapté aux composants [186] à une substitution contextuelle dépassant le sous-typage [38].

4.2.1 Pour une substitution réellement "orientée composants"

La question de la substitution dans le monde composant est un problème industriel, comme l'a souligné R. Van Ommering dans [216]. Celui-ci pose la question de savoir comment être sûr que les changements effectués sur un composant n'affecteront pas les applications concernées. La réponse qu'il préconise passe par des règles basées sur le sous-typage. Il est vrai que l'on pourrait essayer de réutiliser pour les composants les méthodes de sous-typage déjà existantes [172]. On pourrait même reprendre les concepts issus du monde objet tels que l'héritage [270]. Cependant, il faut également tenir compte des critiques qui ont été adressées à ces concepts, au point de les déclarer inadaptées pour les composants.

En effet, le principe selon lequel tout était objet [209] et que l'héritage était la base de la programmation par objets [270] a amené pendant quelque temps à amalgamer les notions de type et de classe [262]. On a donc pendant un temps assimilé le sous-typage à l'héritage (ou au "sous-classage") [209, 115]. Cependant, avec le développement du sous-typage comportemental [188, 165, 171, 172], les deux notions ont été progressivement séparées [262, 9, 177, 240, 165, 218]. L'héritage a été destiné à l'implémentation et à la réutilisation de code, tandis que le sous-typage servirait à la spécification formelle du comportement. Puis, avec le développement des premiers modèles de composant [239] et l'émergence d'un nouveau paradigme autour de ce concept [225], le paradigme objet a été de plus en plus critiqué. On a notamment reproché à l'héritage d'offrir un mauvais cadre pour la réutilisation [239, 275], et d'entraver l'encapsulation [262, 239, 274], au point de vouloir l'abandonner dans le monde composant [274, 273]. En essayant de poser les bases de ce que doit être un typage pour composants, J. C. Seco et L. Caires [257] définissent un modèle utilisant le sous-typage, mais évitant l'héritage. On a également appelé à abandonner le sous-typage [274, 44], voire le typage [225] à cause de leur rigidité.

D'un point de vue plus général, deux besoins émergent. Premièrement, le besoin pour le monde composant de dépasser le paradigme objet [259, 272, 273]. Deuxièmement, le besoin d'avoir un concept de typage ou de substitution plus flexible, vraiment orienté composants [210, 186, 38]. Cette nouvelle substitution prendrait en compte les aspects spécifiques à

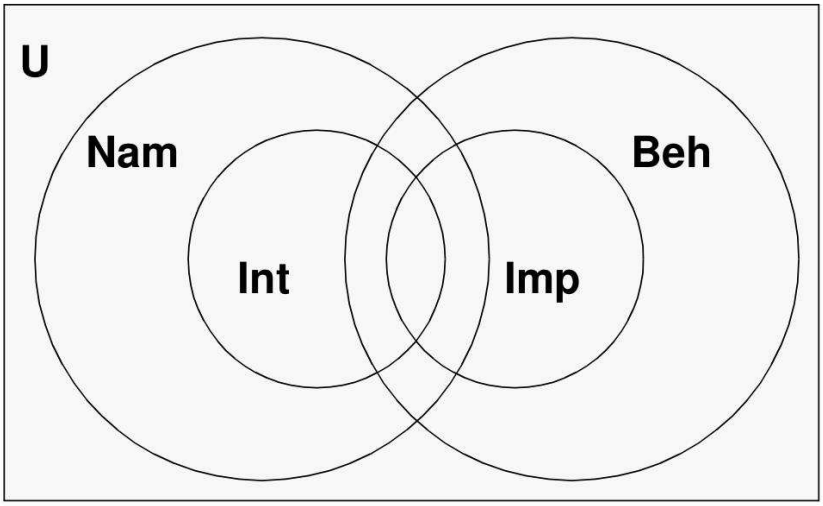

FIG. 4.1 – Catégories de sous-types (d'après [186])

ce paradigme. Par exemple, un mécanisme qui permettrait à un composant de se substituer à un autre sans forcément passer par les règles de sous-typage strict. C'est cette idée que l'on retrouve dans les deux approches que nous allons maintenant présenter.

4.2.2 Sous-typage hétérogène de Medvidovic

N. Medvidovic [186] a constaté que le typage utilisé dans les modèles de composants et les architectures logicielles était trop influencé par le typage objet. En effet, les formes existantes de sous-typage utilisées dans le monde objet n'acceptent généralement qu'une seule catégorie de sous-type [219]. Or, ces formes de sous-typage sont trop rigides pour le développement de logiciels utilisant différents composants de modèles et d'origines diverses. D'ailleurs, il existe certaines catégories de sous-types qui ne sont pas prévues par le monde objet, mais qui seraient adaptées aux composants et aux architectures logicielles. De plus, Medvidovic *et al* ont montré que dans certains cas, des composants qui ne sont pas sous-types au sens où on l'entend dans le monde objet peuvent quand même être substitués les uns aux autres avec succès dans une même architecture [185, 187]. D'où l'idée de concervoir un sous-typage adapté au monde hétérogène des composants et des architectures logicielles.

En s'inspirant du sous-typage objet [172, 219], Medvidovic a identifié quatre catégories de sous-types, qui quand on les combine peuvent engendrer plusieurs mécanismes de sous-typage. La figure 4.1 montre ces quatre catégories. *Nam* symbolise la conformité de nom, qui impose que le sous-type partage avec son supertype au moins les mêmes noms ainsi que des paramètres de même nom. Par exemple[1], si l'on considère une méthode $AddElement$:

[1]Dans [186], Medvidovic utilise pour ses exemples des composants spécifiés d'après le système de types architectural qu'il a mis au point en utilisant le langage Z [266]. Pour des raisons de simplicité, nous utiliserons des

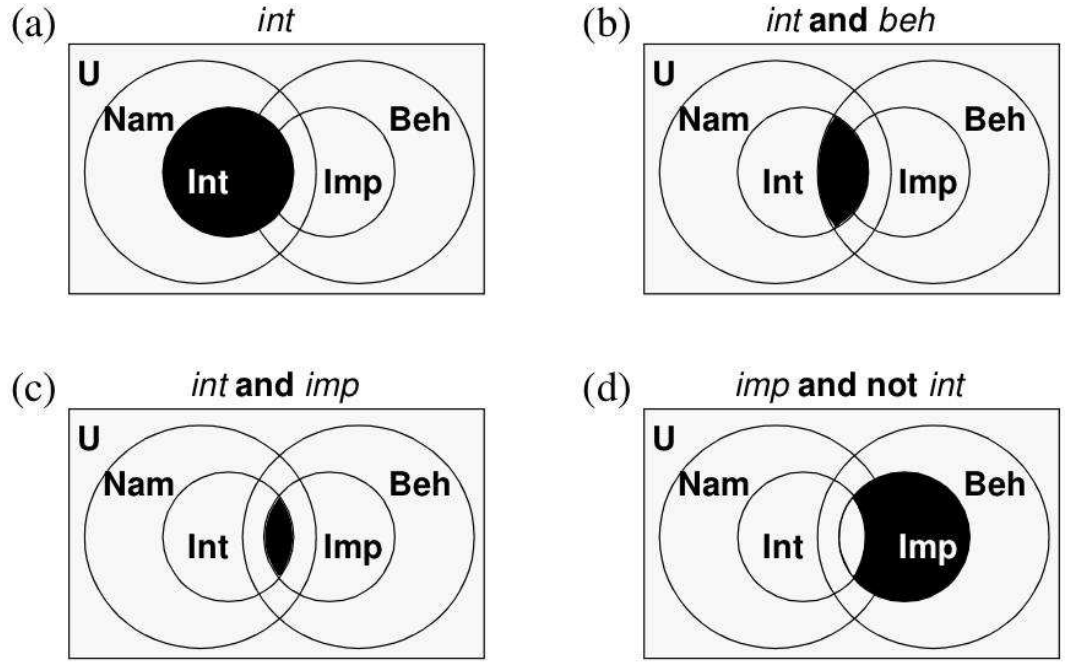

FIG. 4.2 – Exemples de sous-typages (d'après [186])

$(newElement : ELEMENT) \rightarrow INTEGER$, alors la méthode $AddElement : (newElement : ELEMENT, newElementId : STRING) \rightarrow INTEGER$ est sous-type de la précédente selon la conformité de nom. *Int* représente la conformité d'interfaces. Celle-ci impose, en plus de la conformité de nom, que le composant sous-type fournisse au moins et requière au plus les mêmes interfaces et les mêmes méthodes, sachant qu'une méthode est sous-type d'une autre en fonction de la contravariance de ses paramètres et de la covariance de son résultat [50, 51]. Par exemple, la méthode $AddElement : (newElement : ELEMENT, newElementId : STRING) \rightarrow FLOAT$ est sous-type de la méthode $AddElement : (newElement : ELEMENT, newElementId : STRING) \rightarrow INTEGER$. *Beh* concerne le sous-typage comportemental. Celui-ci impose que le sous-type possède le même comportement que le supertype. C'est-à-dire qu'il doit préserver l'invariant et proposer aussi bien des préconditions plus faibles que des postconditions plus fortes [172]. Par exemple, si une méthode $AddElement : (newElement : ELEMENT) \rightarrow INTEGER$ possède une précondition $NOT(newElement \in ListOfElements)$ et une post-condition $newElement \in ListOfElements$, une méthode *AddNewElement* qui n'a pas cette pré-condition, mais qui partage la même post-condition est sous-type de la première en terme de comportement. Seulement en terme de comportement, d'ailleurs, puisque le nom est différent. Enfin, *Imp* désigne le sous-typage d'implémentation. Celui-ci impose qu'en plus de la conformité de comportement, certaines implémentations du supertype soient également celles du sous-type.

La figure 4.2 montre quelques exemples de mécanismes de sous-typage engendrés par

exemples plus généraux. En effet, nous ne présentons pas un système de types particulier, mais un principe de sous-typage.

la combinaison de ces quatre catégories. Le mécanisme illustré par *(a)* représente la simple conformité d'interface (*Int*). Celle-ci permet, dans une application à base de composants, de changer l'un d'eux par un sous-type sans affecter le fonctionnement des composants voisins. Le mécanisme illustré par *(b)* représente la double conformité d'interface et de comportement (*Int and Beh*). Celle-ci permet de démontrer la correction de la substitution à la manière des langages de programmation comme *Eiffel* [188, 189]. On s'assure que le composant sous-type a les mêmes fonctionnalités, et qu'en plus il les utilise de la même manière que le supertype. Le mécanisme illustré par *(c)* représente la double conformité d'interface et d'implémentation appelée *strictly monotone subclassing*. Celle-ci impose que le sous-type préserve non seulement les fonctionnalités (et le comportement), mais aussi les implémentations du supertype. Ce sous-typage est aussi strict que l'héritage, la conformité de comporement en plus. Enfin, le mécanisme illustré par *(d)* représente un mécanisme de sous-typage qui offre la conformité d'implémentation sans la conformité d'interface. Cela est inhabituel dans le monde objet, mais adapté au monde composant. En effet, ce mécanisme stipule que deux types peuvent offrir une même implémentation dans un certain contexte sans être sous-types l'un de l'autre d'un point de vue fonctionnel. Dit autrement, un composant peut en remplacer un autre dans un contexte particulier sans être son sous-type. Cela signifie que dans un autre contexte, il ne pourrait pas forcément le remplacer.

L'idée de Medvidovic [186] est de proposer un mécanisme de sous-typage hétérogène qui inclurait ces quatres catégories de sous-types (*Nam*, *Int*, *Beh* et *Impl* sur la figure 4.1). On considère dans ce cas qu'un composant peut être sous-type d'un autre s'il appartient au moins à l'une des quatre catégories. On peut donc modéliser non seulement les mécanismes de sous-typage issus du monde objet, mais aussi de nouveaux mécanismes mieux adaptés au monde composant. Cependant, ce sous-typage hétérogène, bien qu'il essaie de dépasser les mécanismes existant dans le monde objet, se base encore beaucoup sur eux. De plus, il ne tient aucun compte des propriétés non-fonctionnelles, et ne propose rien pour les modéliser.

4.2.3 Substitution contextuelle de Brada

P. Brada [38] a introduit les notions de "contexte de déploiement" et de "substituabilité contextuelle". On considère qu'un composant est un ensemble de services (interfaces) fournis et requis. Le contexte de déploiement d'un composant est le sous-ensemble de ses services qui sont réellement utilisés par les autres composants de l'application. Plus précisément, le contexte de déploiement d'un composant est constitué : i) du sous-ensemble de ses services requis qui sont approvisionnés par les services fournis d'autres composants ; ii) du sous-ensemble de ses services fournis qui sont utilisés par les services requis d'autres composants. La substituabilité contextuelle de Brada consiste donc à comparer un composant candidat avec ce sous-composant qui est le contexte de déploiement, plutôt qu'avec tout le composant.

La figure 4.3 montre un exemple de substitution d'un composant C^c par un composant C^r dans un certain contexte de déploiement. Les rectangles noirs sont les services fournis des composants (*exports*, sur la figure). Les rectangles blancs sont leurs services requis (*needs*). La zone en niveaux de gris représente l'ensemble de leurs services qui sont utilisés (*ties*). Enfin, les traits (*bindings*) symbolisent chacun une connexion entre un sevice fourni d'un composant

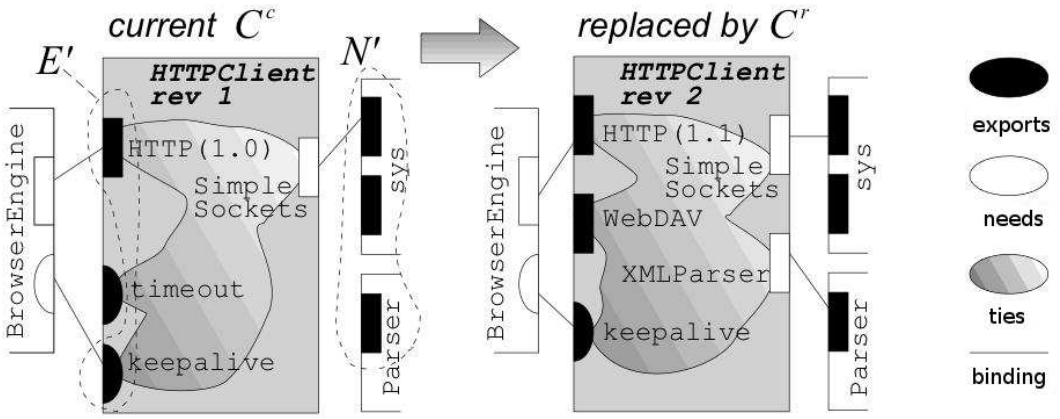

FIG. 4.3 – Substitution contextuelle de composants (d'après [38])

et un service requis d'un autre composant. Sur la figure 4.3, le composant C^c, qui est un client HTTP, fournit les services `HTTP(1.0)` et `keepalive` au composant *BrowserEngine*. C^c requiert également un service, `SimpleSocket`, qui est alimenté par le composant *sys*. Un service fourni par C^c, `timeout`, n'est pas utilisé par d'autres composants, tandis que le composant *Parser*, qui pourrait fournir des services, n'est pas utilisé par C^c. De son côté, le composant C^r, qui est un client HTTP "évolué", fournit les services *HTTP(1.1)* (version évoluée de `HTTP(1.0)`) et `keepalive`, mais pas `timeout`. De plus, il requiert le service `SimpleSocket` tout comme C^c, mais également le service `XMLParser`, que peut lui fournir le composant *Parser*. Enfin, il fournit le service supplémentaire `WebDAV` qui n'est utilisé par aucun composant.

On remarque que C^r n'est pas sous-type de C^c, car il fournit moins de services et il en requiert plus. C'est l'exact contraire du principe de contravariance-covariance du sous-typage. Cependant, le contexte de déploiement de C^c est tel que C^r fournit exactement les services qu'attend le composant *BrowserEngine* (client de C^c), et qu'il ne requiert pas plus de services que ce que *sys* et *Parser* (fournisseurs potentiels de C^c) peuvent offrir. Donc compte tenu de ce contexte particulier, C^r peut se substituer à C^c.

L'originalité du travail de Brada réside dans cette notion de contexte de composition, qui permet de remplacer un composant par un autre sans qu'il y ait de relation de sous-typage entre eux. Cependant, sa notion de contexte, basée sur l'architecture existante de l'application, concerne un composant "concret" déjà intégré, qui est remplacé par un autre composant "concret" en fonction de son déploiement dans l'architecture globale. Une telle approche ne fonctionnerait pas si on essayait de modéliser le besoin par le biais d'un composant "recheché" virtuel. De plus, comme pour le sous-typage hétérogène de Medvidovic [186] étudié précédemment, l'approche de Brada ne tient aucun compte des propriétés non-fonctionnelles.

4.3 Description et comparaison de propriétés non-fonctionnelles

On constate que les techniques de comparaison de composants présentés dans la section précédente ne s'occupent que de la partie fonctionnelle des composants. Leur partie non-fonctionnelle n'est pas prise en compte par ces techniques. La question est donc de savoir s'il existe un moyen de comparer ces propriétés non-fonctionnelles. En effet, l'un des objectifs de cette thèse est de pouvoir englober dans une comparaison commune les propriétés fonctionnelles et non-fonctionnelles du composant (cf. chapitre 1). Il est donc nécessaire que les propriétés non-fonctionnelles soient mesurables, et qu'elles le soient aussi "facilement" que les propriétés fonctionnelles.

Un préalable à la comparaison est de décrire ce qu'on veut comparer. Il existe plusieurs méthodes de description des propriétés non-fonctionnelles, et particulièrement de la qualité de service [1]. Parmi elles, on peut citer la logique temporelle [25] et les complétions [294]. Cependant, beaucoup de ces méthodes se concentrent sur une propriété comme la performance [294]. On peut également citer les langages d'expression de propriétés de qualité de service [1, 75], que nous allons présenter au début de cette section. Ces langages sont intéressants en raison de leur facilité d'utilisation et des concepts qu'ils proposent. Cependant, ces langages se concentrent sur la qualité de service, c'est-à-dire une famille de propriétés à l'intérieur de la qualité. Il reste enfin les métriques, qui constituent l'outil d'évaluation le plus simple de la qualité dans son ensemble. Nous présenterons les travaux qui leur sont consacrés dans la deuxième partie de cette section.

4.3.1 Contrats de qualité de service

A. Beugnard *et al.* [22] ont divisé les contrats pour les composants en quatre niveaux de négociabilité illustrés par la figure 4.4. Chaque niveau englobe les obligations des niveaux inférieurs. Le premier niveau, qu'on retrouve dans de nombreux langages de programmation, concerne les contrats syntaxiques. Ces contrats comportent notamment les opérations que le composant peut exécuter, et permettent de vérifier la conformité entre les interfaces fournies et requises des composants. Le deuxième niveau, qu'on retrouve dans certains langages comme Eiffel [188, 189], concerne les contrats comportementaux. Ces contrats spécifient le comportement des opérations par le biais de pré-conditions, de post-conditions et d'invariants. Le troisième niveau concerne les contrats de synchronisation. Ces contrats spécifient le comportement global des objets en terme de synchronisation entre appels de méthodes. Il s'agit de décrire les dépendances entre les services d'un composant, afin de s'assurer qu'ils seront correctement exécutés. Enfin, le quatrième niveau, celui qui nous intéresse, concerne les contrats de qualité de service. Ces contrats peuvent être spécifiés grâce à des langages dont nous allons voir certains exemples. Nous commencerons par CQML, car c'est le seul langage qui propose une méthode pour comparer et substituer des propriétés de qualité de service. Nous nous intéresserons ensuite aux langages QML et QoSCL.

4.3.1.1 CQML

CQML (Component Quality Modelling Language), mis au point par J. Aagedal [1], est un langage d'expression de propriétés de qualité de service pour les composants logiciels. Ce

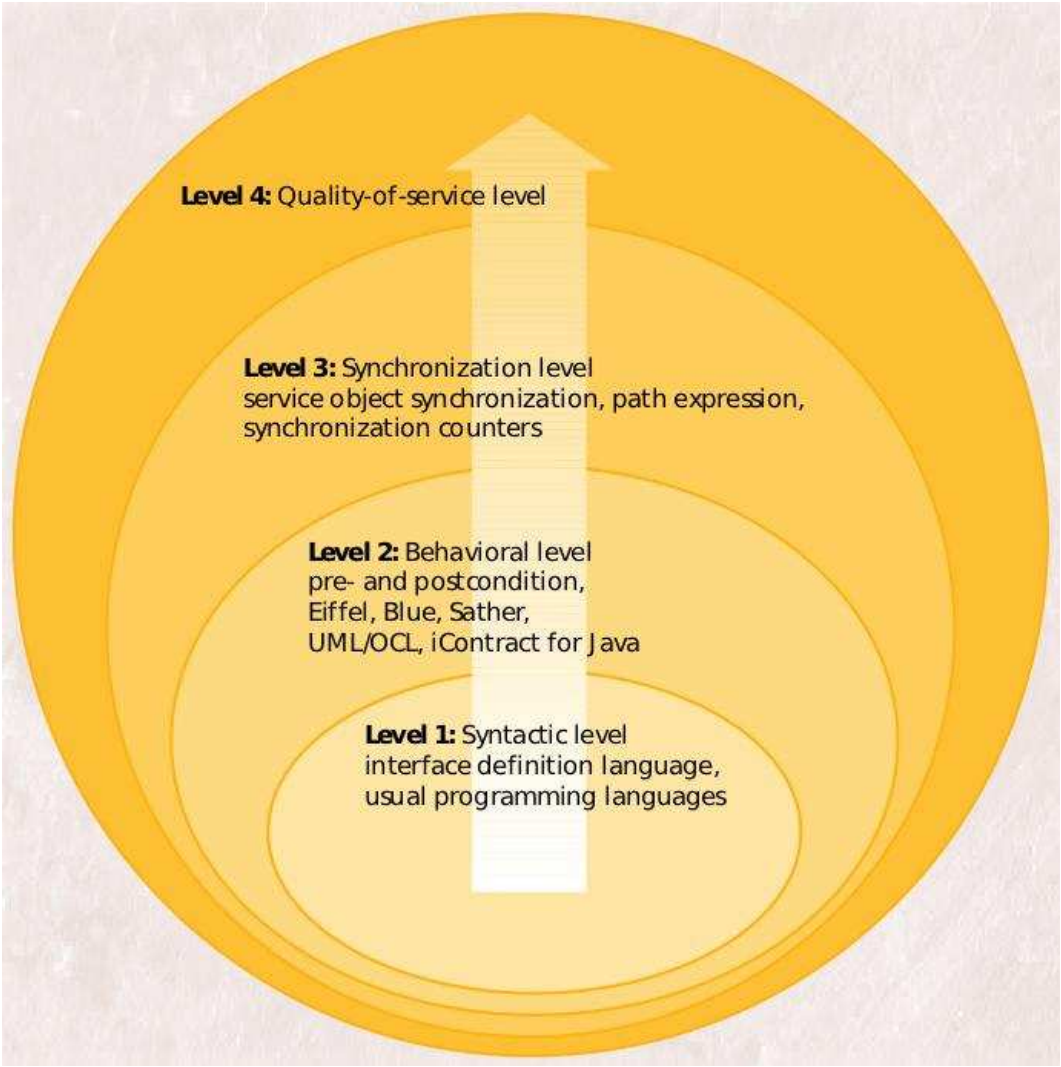

FIG. 4.4 – Niveaux de contrats pour les composants (d'après [22])

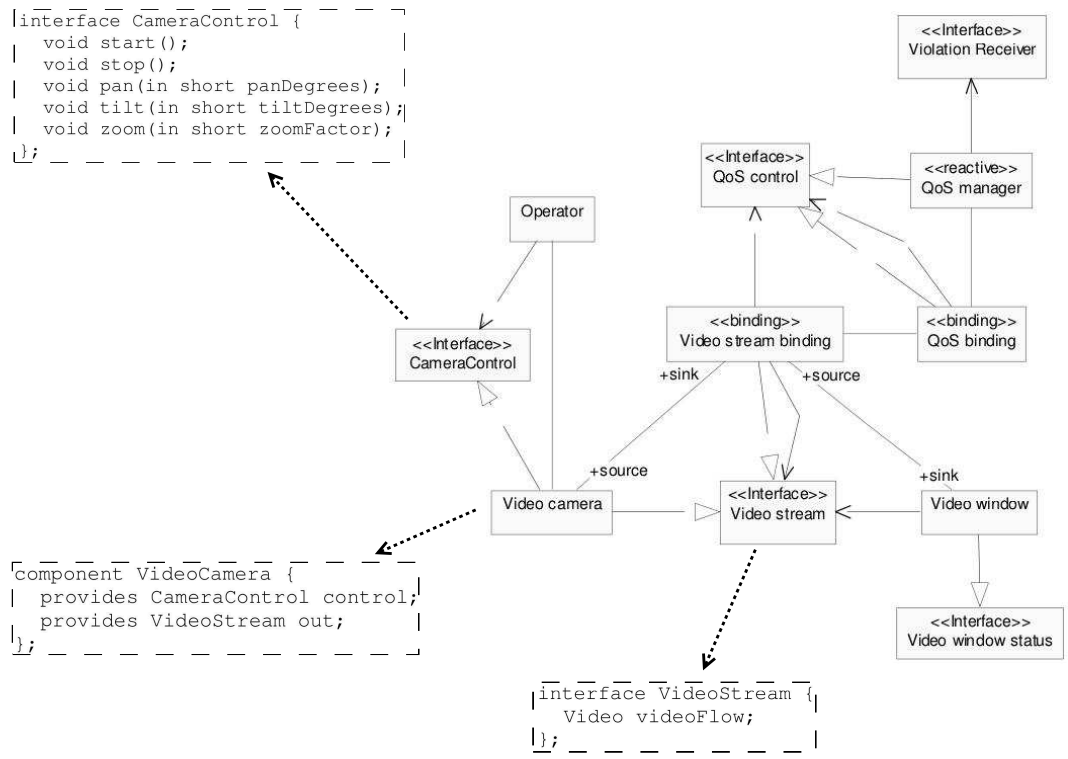

FIG. 4.5 – CQML : représentation des composants (d'après [1])

langage prend en compte aussi bien les composants primitifs que les composants composites. Sa particularité est qu'il ne fait pas que décrire les propriétés de qualité de service : il propose également des règles de substituabilité pour ces propriétés. En effet, Aagedal considère que la substituabilité est particulièrement importante dans le cas de l'initialisation ou de l'adaptation d'un service. Il considère également que tester la conformité d'un composant candidat vis-à-vis de certaines exigences revient à essayer de substituer ce candidat à un composant virtuel modélisant ces exigences.

Dans CQML, les composants possèdent des interfaces fournies et requises, lesquelles possèdent des attributs ou des opérations. Un composant peut être simple ou composite (*compound*), c'est-à-dire posséder des sous-composants. Les propriétés non-fonctionnelles d'un composant sont regroupées dans son profil de qualité de service (*profile*). Ce profil peut être lui aussi simple ou composite, selon la nature du composant auquel il est associé. Chaque profil est composé d'exigences (*statement*) qui utilisent des caractéristiques de qualité de service (*characteristics*) sur des éléments passés en paramètre. Ces éléments peuvent être, par exemple, des opérations ou des attributs d'interface. Chaque caractéristique possède donc ses propres paramètres, ainsi qu'un domaine (*domain*). Il s'agit d'une fonction qui indique comment la valeur est obtenue (*values*), et éventuellement un invariant. Chaque domaine est constitué d'une direction croissante (*increasing*) ou décroissante (*decreasing*), ainsi que d'une unité et de l'ensemble de valeurs possibles. Cela peut être, par exemple, l'ensemble des nombres réels.

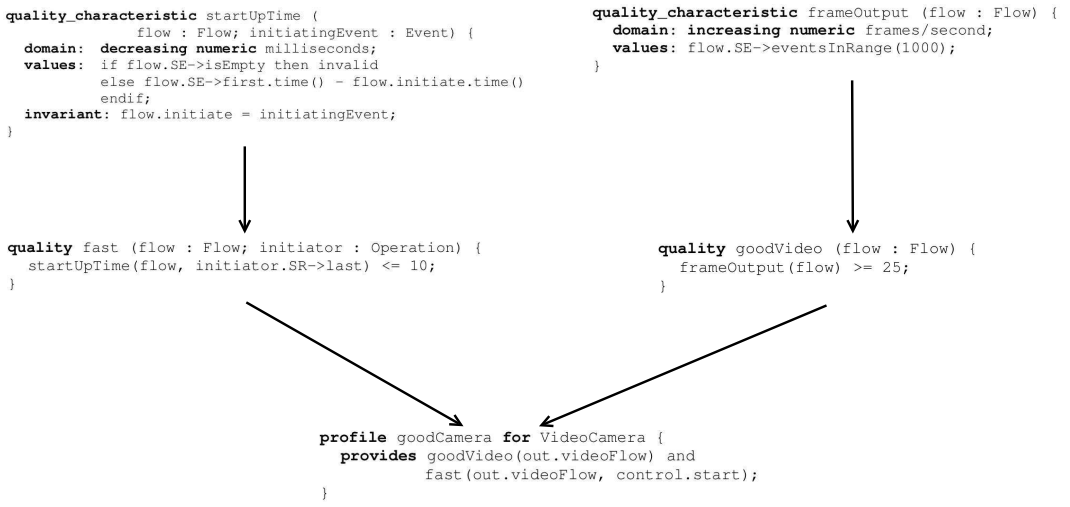

FIG. 4.6 – CQML : représentation de la qualité de service (d'après [1])

La figure 4.5 montre un exemple, adapté de [24], de représentation CQML de composants en rapport avec une caméra vidéo. En particulier, le composant simple *VideoCamera* fournit deux interfaces : *out*, de type *VideoStream*, dédiée au flux vidéo, et *control*, de type *CameraControl*, dédiée à la manipulation de la caméra. Une interface de type *VideoStream* a un attribut *VideoFlow* de type *Flow*. Une interface de type *CameraControl* possède cinq opérations pour commander à la caméra : *start*, *stop*, *pan*, *tilt* et *zoom*. Pour des raisons de simplicité, nous ne occuperons pas des autres éléments pour nous concentrer sur le composant *VideoCamera* et sa qualité.

La figure 4.6 montre un exemple d'assignation de propriétés de qualité de service à ce composant. Supposons qu'on veuille spécifier qu'une caméra vidéo émette au moins 25 frames par seconde et qu'elle démarre au plus tard 10 millisecondes après qu'on ait appelé la méthode *start*.

La première propriété nécessite de faire appel à une caractéristique de qualité de service *frameOutput*, à droite de la figure 4.6. Cette caractéristique prend en paramètre un élément de type *Flow*. Elle a pour domaine l'ensemble ordonné croissant des nombres réels (*increasing numeric*). Elle est exprimée en frames par seconde et calculée à partir de la fonction *eventInRange*. À partir de cette caractéristique, on peut modéliser l'exigence de qualité de service *goodVideo*. Selon cette exigence, la caractéristique *frameOutput* appliquée à un flux vidéo doit rendre un résultat supérieur à 25 frames par seconde.

La deuxième propriété nécessite de faire appel à une caractéristique de qualité de service *startUpTime*, à gauche de la figure 4.6. Cette caractéristique prend en paramètre un flux vidéo et un évènement quelconque. Elle a pour domaine l'ensemble ordonné décroissant des nombres réels (*decreasing numeric*). Elle est exprimée en millisecondes, et elle calcule le temps de déclenchement de l'évènement passé en paramètre, moins le temps où on a demandé son déclenchement. À partir de cette caractéristique, on peut modéliser l'exigence de qualité de service *fast*. Selon cette exigence, la caractéristique *startUpTime*, appliquée à un flux vidéo et

à une opération quelconque, doit rendre un résultat inférieur à 10 millisecondes.

On peut donc définir le profil de qualité de service *goodCamera* pour le composant *VideoCamera*. Ce profil fournit l'exigence *goodVideo* à l'attribut *VideoFlow* de l'interface *out*, ainsi que l'exigence *fast* à l'opération *start* de l'interface *control*.

Comme nous l'avons vu plus haut, la vraie particularité de CQML est qu'Aagedal a prévu un mécanisme de substituabilité pour les profils de qualité de service. Prenons deux caractéristiques de qualité de service Q_A et Q_B. On considère que Q_A peut se substituer à Q_B à condition : i) que le domaine de Q_A soit inclus dans le domaine de Q_B ; ii) que l'ensemble de valeurs que Q_B peut prendre soient incluses dans l'ensemble de valeurs que Q_A peut prendre ; iii) que les paramètres de Q_A soient conformes (en termes de type) à ceux de Q_B ; iv) que Q_A préserve les invariants de Q_B. À partir de là, on considère qu'une exigence de qualité S_A peut se substituer à une exigence de qualité S_B si toute caractéristique de qualité de service de S_B peut être substituée par une caractéristique de qualité de service de S_A. Enfin, considérons deux profils de qualité de service P_A et P_B. Si P_A et P_B sont deux profils simples, alors il faut que chaque exigence de P_B puisse être substituée par une exigence de P_A. Si P_A est simple mais que P_B est composite, il faut que chaque sous-profil de P_B puisse être substitué par P_A. Si à l'inverse P_A est composite mais que P_B est simple, il faut que P_B puisse être substitué par au moins un sous-profil de P_A. Enfin, si P_A et P_B sont tous les deux composites, il faut que chaque sous-profil de P_A puisse se substituer à un sous-profil de P_B.

L'avantage de CQML est double. D'une part, il offre un moyen de représenter les exigences de qualité de service des composants de manière simple. D'autre part, il permet de proposer des règles de substituabilité pour ces exigences. Or, il est peu fréquent que dans les travaux consacrés à la description des propriétés non-fonctionnelles, on trouve un mécanisme de comparaison de ces propriétés. De plus, le concept de composant virtuel mis en avant par Aagedal pour utiliser ces règles de substituabilité présente des similitudes avec le concept de composant recherché présenté dans cette thèse. Cependant, CQML présente quelques limites. Tout d'abord, les profils, exigences et caractéristiques de qualité de service ont tous la même importance. Rien n'est proposé pour hiérarchiser les propriétés les unes par rapport aux autres. Par exemple, dans le cas de la caméra vidéo, il se peut que l'utilisateur accorde plus d'importance à la qualité de l'image qu'au temps de réponse, mais on ne peut pas le modéliser. De plus, la susbtitution fonctionnelle est complètement séparée de la substitution non-fonctionnelle, et rien n'est proposé pour la première. Enfin, si la susbtituabilité non-fonctionnelle d'un composant est testée, elle n'est pas mesurée. Aagedal a prévu des règles pour indiquer dans quel cas deux caractéristiques, exigences ou profils peuvent être substitués l'un à l'autre. Mais il n'y a pas de mesure de distance au cas où la substituabilité n'est pas totale, contrairement à ce que nous souhaitons trouver comme propriété pour un opérateur de comparaison entre composants (cf. sous-section 1.2.4). Nous ne pouvons donc pas utiliser telle quelle cette substituabilité dans ce travail de thèse.

4.3.1.2 QML et QoSCL

QML (*Quality of service Modeling Language*), mis au point par S. Frolund et J. Koistinen [101, 102], est un langage qui permet de décrire les propriétés de qualité de service des composants sur plusieurs niveaux : interfaces, opérations... C'est l'une des approches les plus

complètes pour la spécification de la qualité de service [1].

Dans QML, les différentes caractéristiques de la qualité de service, comme la performance et la fiabilité, sont représentées par des types ou "familles" de contrat. Ces familles peuvent être instanciées par plusieurs contrats différents. Ces contrats sont ensuite affectés à divers éléments de l'interface et constituent ainsi son profil de qualité de service. Un même contrat pouvant concerner plusieurs éléments simultanément. Un type de contrat décrit tous les aspects de la caractéristique qu'il représente, chacun de ces aspects étant représenté par une dimension. Le domaine d'une dimension peut être un nombre réel ou un ensemble de valeurs quelconques, ordonné ou non. Le domaine peut être croissant (*increasing*, ce qui signifie que plus la valeur est élevée, mieux c'est) ou au contraire décroissant (*decreasing*).

Prenons l'exemple d'un type de contrat de fiabilité *Reliability*, tiré de [74]. Ce type de contrat impose un certain niveau de confiance (*confidence*) ainsi qu'un faible nombre d'échecs par an (*NOF*). En QML, ce type de contrat serait spécifié de la manière suivante :

$$
\begin{array}{l}
type\ Reliability = contract\ \{ \\
\quad confidence : increasing\ enum\ \{none, low, medium, high\} \\
\quad with\ order\ \{none < low\,;\ low < medium\,;\ medium < high\,;\} \\
\quad NOF : decreasing\ numeric\ no/year\,; \\
\};
\end{array}
$$

La dimension *confidence* indique plusieurs niveaux de confiance possibles par le biais d'un ensemble ordonné croissant (*increasing enum*). Cet ensemble va de *none* (aucune confiance) jusqu'à *high* (niveau de confiance élevé). La dimension *NOF*, par contre, renvoie une valeur de direction décroissante, car moins il y a d'échecs, plus on peut avoir confiance.

Un contrat instanciant le type *Reliability* pourrait imposer, par exemple, que le niveau de confiance soit élevé, et que le nombre d'échecs ne dépasse pas 6 par an. Ce qui donnerait ceci :

$$
\begin{array}{l}
ReliabilityBasic = Reliability\ contract\ \{ \\
\quad NOF \leq 6\,; \\
\quad confidence == high\,; \\
\}
\end{array}
$$

Un autre contrat instanciant le type *Reliability* pourrait imposer : i) que le niveau de confiance soit élevé ; ii) que la moyenne (*mean*) du nombre d'échecs ne dépasse pas 3 ; iii) que 90% des valeurs (*percentile* 90) soient inférieures à 5. Ce qui donnerait ceci :

$$
\begin{array}{l}
ReliabilityAdvanced = Reliability\ contract\ \{ \\
\quad NOF\ \{ \\
\quad\quad mean \leq 3\,; \\
\quad\quad percentile\ 90 \leq 5\,; \\
\quad \} \\
\quad confidence == high\,; \\
\}
\end{array}
$$

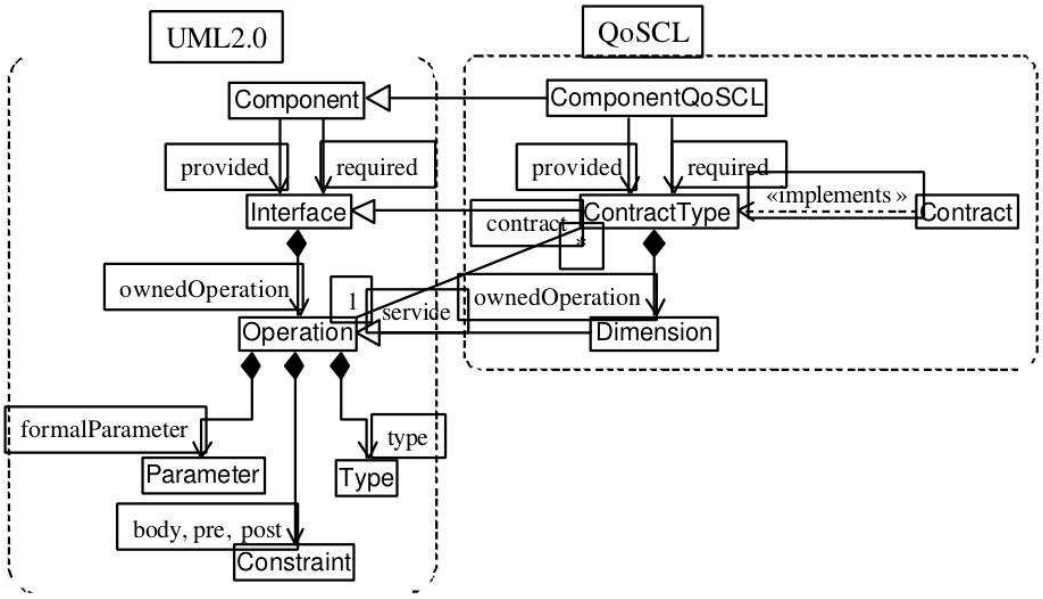

FIG. 4.7 – QoSCL comme extension d'UML 2.0 (d'après [75])

Le langage QoSCL (Quality of Service Contract Language) [74, 75] reprend les principes de QML en les améliorant. Ce langage étend également UML 2.0 [215] pour y inclure des composants spécifiés avec leurs contrats de qualité de service. La figure 4.7 illustre la manière dont QoSCL étend UML 2.0. On constate que QoSCL emprunte à QML le vocabulaire et les concepts de type de contrat, de contrat et de dimension. On constate aussi qu'un composant QoSCL est en fait un composant UML 2.0 avec des propriétés de qualité de service contractuellement spécifiées.

La principale amélioration apportée à QML réside dans la notion de fonction. Cette notion sert à exprimer des dimensions par le biais d'autres dimensions. Une fonction peut être égale à la valeur de la dimension passée en paramètre, ou correspondre à une valeur numérique, ou encore s'exprimer sous forme de combinaison d'autres fonctions.

Reprenons l'exemple du type de contrat *Reliability*. Avec QML, on peut instancier ce type pour obtenir des contrats qui imposeraient que le nombre d'échecs n'excède pas une certaine valeur. Avec QoSCL, on peut, en plus, imposer : i) que le niveau de confiance pour un certain contrat soit le même que pour un autre contrat passé en paramètre ; ii) que le nombre d'échecs pour le premier contrat soit deux fois inférieur au nombre d'échecs pour le second. Ce qui donnerait le contrat suivant :

$$
\begin{aligned}
&ReliabilityC(var : Reliability) = Reliability\ contract\ \{\\
&\quad NOF < var.NOF * 2\,;\\
&\quad confidence == var.confidence\,;\\
&\}
\end{aligned}
$$

La particularité de QML et de QoSCL est qu'ils offrent chacun un mécanisme de spécification puissant et complet. Ce mécanisme permet de représenter plusieurs caractéristiques de qualité de service différentes, et ce pour plusieurs types d'éléments différents d'un même composant. Cela dit, ces langages sont limités à la spécification de la seule qualité de service, qui est explicitement séparée de la spécification des propriétés fonctionnelles. De plus, contrairement à CQML, aucun mécanisme n'est proposé pour comparer de tels contrats entre eux.

4.3.2 Métriques et modèles de qualité pour les composants

Après avoir étudié les langages d'expression de propriétés de qualité de service, nous allons nous intéresser à un outil de description de la qualité logicielle dans son ensemble. De nombreux travaux ont été menés afin de mesurer avec précision cette qualité logicielle, et de nombreuses métriques ont été proposées [88, 174]. On peut citer par exemple la complexité de l'interface [79], l'utilisation de ressources [87], le nombre de lignes de code [140, 87], le nombre de points de fonction [5, 4], la complexité cyclomatique [183], ainsi que d'autres métriques concernant la taille ou la complexité du logiciel [116, 19, 72]. Dans le domaine de la programmation modulaire et orientée objet, on trouve notamment la suite de métriques proposée par S. Chidamber and C. Kemerer [58] (nombre de sous-classes et de méthodes par classe, profondeur de l'arbre d'héritage...), ainsi que des métriques de cohésion [203] et de couplage [121, 122] adaptées au monde objet.

Plusieurs de ces métriques ont toutefois l'inconvénient d'être difficilement adaptables aux composants, et en particulier aux COTS. En effet, elles mesurent des attributs qui concernent le code source ou d'autres aspects de la structure interne du logiciel. Des métriques ont cependant été proposées, qui concernent spécifiquement les composants COTS ou qui peuvent être adaptées à ce cadre [112]. Parmi elles, on peut citer les métriques de réutilisabilité du composant [289], de complexité de l'interface [107], et d'utilisation de services dans le cadre des architectures logicielles [284]. D'autres travaux plus généraux ont cherché à étendre les modèles de qualité existants, en proposant des caractéristiques, des sous-caractéristiques et/ou des attributs qualité supplémentaires avec les métriques associées [54, 52].

D'autres approches, enfin, ont entrepris d'adapter complètement le standard ISO-9126 [132] aux composants COTS. D'une part, elles proposent certaines caractéristiques et sous-caractéristiques supplémentaires. D'autre part, elles suppriment celles qui existaient dans ISO-9126 et qui ne sont plus adaptées pour les COTS. De plus, chaque sous-caractéristique est décomposée en plusieurs attributs qualité, auxquels sont associés des métriques (une pour chaque attribut). Ce sont les approches de ce genre qui sont les plus intéressantes, car elles permettent d'obtenir des modèles de qualité complets et spécialisés pour les COTS. Nous allons d'abord détailler le standard ISO-9126 avant de présenter quelques exemples travaux qui se proposent l'étendre et de l'appliquer aux composants COTS.

4.3.2.1 Le standard ISO-9126

Characteristics	**Subcharacteristics**
Functionality	*Suitability*
	Accuracy
	Interoperability
	Compliance
	Security
Reliability	*Maturity*
	Recoverability
	Fault Tolerance
Usability	*Learnability*
	Understandability
	Operability
Efficiency	*Time behavior*
	Resource behavior
Maintainability	*Stability*
	Analyzability
	Testability
	Changeability
Portability	*Installability*
	Conformance
	Adaptability
	Replaceability

TAB. 4.1 – ISO-9126 : caractéristiques et sous-caractéristiques de qualité (d'après [132])

ISO-9126 est constitué de caractéristiques de qualité, elles-mêmes composées de sous-caractéristiques. Elles représentent les principales propriétés de qualité qu'on peut attendre d'un logiciel. Le tableau 4.1 montre les différentes caractéristiques et sous-caractéristiques de qualité du standard, qui sont définies de la manière suivante :

- Capacité fonctionnelle (*Functionality* sur le tableau) : C'est la capacité d'un logiciel à répondre aux besoins fonctionnels. Cela inclut la capacité à se plier aux exigences (*Compliance*), la couverture du besoin (*Suitability*), le degré de précision de leur satisfaction (*Accuracy*), la capacité à répondre aux besoins de sécurité (*Security*) et la capacité à répondre aux besoins d'interopérabilité (*Interoperability*).
- Fiabilité (*Reliability*) : C'est la capacité d'un logiciel à fournir ses services dans des conditions précises et pendant une période déterminée. Cela inclut sa maturité (*Maturity*), sa capacité à se rétablir d'éventuelles pannes (*Recoverability*), et sa tolérance aux erreurs (*Fault tolerance*).
- Utilisabilité (*Usability*) : C'est la capacité d'un logiciel à être appris rapidement (*Learnability*), facilement compréhensible (*Understandability*) et opérable (*Operability*).
- Efficacité *Efficiency*) : C'est la capacité d'un logiciel à fournir ses services de manière efficace en terme de temps d'exécution (*Time behavior*) et de consommation des ressources système (*Resource behavior*).

– Maintenabilité (*Maintainability*) : C'est la capacité d'un logiciel à être maintenu à jour. Cela inclut la stabilité (*Stability*), ainsi que la facilité avec laquelle il peut être analysé (*Analyzability*), testé (*Testability*) et changé (*Changeability*).
– Portabilité (*Portability*) : C'est la capacité d'un logiciel à être transféré d'un environnement à un autre. Cela inclut la facilité avec laquelle il peut être installé (*Installability*), intégré en harmonie avec le reste de l'application (*Conformance*), adapté (*Adaptability*) et remplacé (*Replaceability*).

4.3.2.2 Modèle de qualité de Bertoa et Vallecillo

M. Bertoa and A. Vallecillo [20, 21] ont proposé un modèle de qualité complet qui modifie ISO-9126 afin de l'adapter aux composants COTS. Les changements sont illustrés sur le tableau 4.2. Les termes présentés sur le tableau ont été expliqués dans la sous-section précédente, sauf les ajouts de Bertoa et Vallecillo, qui sont signalés en gras. D'une part, les sous-caractéristiques de qualité sont divisées en deux catégories : celles qui sont observables pendant l'exécution du logiciel (*Runtime* sur le tableau), et celles qui sont observables pendant le cycle de vie du produit (*Lifecycle*). D'autre part, deux sous-caractéristiques sont rajoutées en raison de leur utilité pour les COTS. La première est la compatibilité, qui indique si la version courante du composant est compatible avec les anciennes versions. La deuxième est la complexité (qui désigne la complexité du composant quand on l'intègre). Enfin, certaines caractéristiques et sous-caractéristiques initialement présentes dans le modèle ISO-9126 sont supprimées (mention *Deleted* sur le tableau) parce qu'elles ne conviennent plus pour les COTS. C'est le cas de la tolérance aux pannes (*Fault Tolerance*) et de la stabilité. En effet, selon [20], ces deux sous-caractéristiques dépendent de l'architecture du produit. Or le modèle de qualité est au niveau composant. C'est aussi le cas de l'analysabilité, parce que selon [20], cette sous-caractéristique concerne davantage le code source, auquel on n'a pas accès dans le cas d'un composant. C'est le cas, enfin, de la portabilité. En effet, selon [20] cette caractéristique n'a pas besoin d'être évaluée, car la portabilité est une propriété intrinsèque des composants.

Une fois qu'ISO-9126 a été modifié pour correspondre aux composants COTS, Bertoa et Vallecillo le complètent en définissant des attributs qualité mesurés par des métriques. Les types de métriques utilisés sont les suivants : *Presence*, qui indique la présence ou non d'un attribut par le biais d'un booléen, *Time*, qui mesure les intervalles de temps au moyen d'un entier et d'une chaîne de caractères désignant l'unité ("secondes", "mois", etc...), *Level*, qui indique le niveau de satisfaction d'un attribut qualité au moyen d'un entier allant de 0 ("très bas") à 5 ("très élevé"), *Ratio*, utilisée pour représenter les pourcentages, et *Integer*, qui regroupe l'ensemble des entiers.

Chaque sous-caractéristique de qualité peut être décomposée en un ou plusieurs attributs, et à chaque attribut est associé une métrique. Par exemple, la sous-caractéristique *Time behavior*, mesurable à l'exécution, est décomposée en trois attributs. Le premier est le temps de réponse, mesuré par une métrique *Time*. Le deuxième est le montant de paramètres que peut traiter le composant pendant un certain laps de temps, mesuré par une métrique *Integer*. Le troisième attribut est le montant qui peut être produit en sortie par le composant pendant un certain laps de temps, mesuré par une métrique *Integer*. La sous-caractéristique *Understandability*, mesurable tout au long du cycle de vie, est décomposée en cinq attributs. Le premier d'entre eux est

Characteristics	**Subcharacteristics (Runtime)**	**Subcharacteristics (Lifecycle)**	**Deleted**
Functionality	*Accuracy* *Security*	*Suitability* *Interoperability* *Compliance* ***Compatibility***	
Reliability	*Recoverability*	*Maturity*	*Fault Tolerance*
Usability		*Learnability* *Understandability* *Operability* ***Complexity***	
Efficiency	*Time behavior* *Resource behavior*		
Maintainability		*Testability* *Changeability*	*Stability* *Analyzability*
Portability			*Installability* *Conformance* *Adaptability* *Replaceability*

TAB. 4.2 – Adaptation d'ISO-9126 aux composants COTS (d'après [20])

le niveau de qualité de la documentation. Le deuxième est le niveau de qualité de l'aide fournie. Le troisième est le niveau de qualité de la documentation utilisable par l'application dans laquelle le composant sera intégré. Le quatrième est la présence ou non d'un tutorial. Enfin, le cinquième attribut est le pourcentage de services utilisés pendant la démonstration du composant. Les trois premiers attributs sont mesurés par une métrique *Level*, le quatrième par une métrique *Presence*, et le dernier par une métrique *Ratio*. La sous-caractéristique additionnelle, *Compatibility*, est représentée par un attribut. Celui-ci indique la compatibilité ou non entre la version courante et les anciennes versions, mesurée par une métrique *Presence*. L'autre sous-caractéristique additionnelle, *Complexity*, est représentée par trois attributs. Le premier d'entre eux est le nombre d'interfaces fournies. Le deuxième est le nombre d'interfaces requises. Enfin, le troisième est le ratio de complexité, qui mesure le nombre d'opérations dans toutes les interfaces fournies divisé par le nombre d'interfaces fournies.

L'avantage de ce modèle de qualité est qu'il est complet à tous les niveaux : caractéristiques, sous-caractéristiques, attributs qualité et métriques. Il est également spécifique aux composants, et les métriques sont simples d'emploi. Par contre, les auteurs ne proposent aucun mode de

comparaison entre deux composants utilisant ce modèle de qualité. L'autre inconvénient, pointé par [112], est que l'application de ce modèle aux COTS disponibles sur le marché dépend étroitement de la documentation fournie par les fournisseurs de ces COTS. Or, il y a justement un manque de standard dans la documentation des propriétés (aussi bien fonctionnelles que non-fonctionnelles) des COTS. Automatiser la mesure de la qualité des composants à partir de ce modèle devient très difficile.

4.3.2.3 CQM

CQM (*Component Quality Model*) est une autre adaptation d'ISO-9126 aux composants, mise au point par A. Alvaro, E.S.d. Almeida et S. Meira [6]. Ce modèle de qualité est inspiré de celui de Bertoa et Vallecillo, bien que les deux modèles soient sensiblement différents.

Le tableau 4.3 illustre les changements qu'apporte CQM par rapport à ISO-9126. La plupart des sous-caractéristiques qui étaient observables à l'exécution (*Runtime*) et durant le cycle de vie (*Lifecycle*) dans le modèle de Bertoa et Vallecillo sont aux mêmes endroits dans CQM. La première différence notable est que CQM supprime beaucoup moins de sous-caractéristiques initiales d'ISO-9126 (*Deleted* sur la figure). Les seules qui disparaissent sont *Analyzability* et *Conformance*. En effet, selon [6], les composants disposent de méthodes d'auto-analyse au déploiement. Une autre sous-caractéristique, *Installability*, change de nom et devient *Deployability*. En effet, selon [6], une fois que les composants sont acquis, ils sont déployés, mais pas installés.

Enfin, par rapport au modèle de Bertoa et Vallecillo, CQM ajoute beaucoup plus de sous-caractéristiques. *Self-Contained* représente la capacité d'un composant à exécuter lui-même les services qu'il offre, sans l'aide d'autres composants. *Configurability* représente la capacité d'un composant à être configuré. Cela inclut le mode de configuration (fichier XML ou texte), le nombre de paramètres, etc... *Scalability* représente la capacité d'un composant à s'accommoder d'un nombre important de données sans changer son implémentation. Et *Reusability* représente la capacité d'un composant à être réutilisé. De plus, une nouvelle caractéristique de qualité a été ajoutée : *Marketability*, qui représente la capacité d'un composant à être commercialisé. Les sous-caractéristiques qui la composent sont le temps de développement du composant (*Development time*), son coût (*Cost*), le temps qu'il faut pour le mettre sur le marché (*Time to market*), le type de marché qui est visé (*Targeted market*) et la capacité du composant à être abordable pour ce type de marché (*Affordability*).

Comme pour le modèle de Bertoa et Vallecillo, CQM décompose ses sous-caractéristiques en attributs qualité, auxquels sont associés des métriques. Trois types de métrique sont disponibles. Le type *Presence* mesure la présence d'un attribut qualité par le biais d'un booléen. Le type *IValues* mesure la valeur d'un attribut par le biais d'un entier (l'unité étant représentée par une chaîne de caractères). Et le type *Ratio* est utilisé pour les pourcentages.

Comme pour le modèle de Bertoa et Vallecillo, chaque sous-caractéristique de qualité peut être décomposée en un ou plusieurs attributs, et à chaque attribut est associé une métrique. Par exemple, la sous-caractéristique *Time behavior*, mesurable à l'exécution, est décomposée en trois attributs, tous mesurés par une métrique *IValues*. Le premier d'entre eux est le temps de réponse. Le deuxième est le montant de paramètres que peut traiter le composant pendant un certain laps de temps. Le troisième est le montant qui peut être produit en sortie par le com-

Characteristics	**Subcharacteristics (Runtime)**	**Subcharacteristics (Lifecycle)**	**Deleted**
Functionality	*Accuracy* *Security*	*Suitability* *Interoperability* *Compliance* ***Self-Contained***	
Reliability	*Recoverability* *Fault Tolerance*	*Maturity*	
Usability	***Configurability***	*Learnability* *Understandability* *Operability*	
Efficiency	*Time behavior* *Resource behavior* ***Scalability***		
Maintainability	*Stability*	*Testability* *Changeability*	*Analyzability*
Portability	*Deployability* *(ex-Installability)*	*Adaptability* *Replaceability* ***Reusability***	*Conformance*
Marketability	***Development time*** ***Cost*** ***Time to market*** ***Targeted market*** ***Affordability***	***Development time*** ***Cost*** ***Time to market*** ***Targeted market*** ***Affordability***	

TAB. 4.3 – CQM : autre adaptation d'ISO-9126 aux composants COTS (d'après [6])

posant pendant un certain laps de temps. La sous-caractéristique *Understandability*, mesurable durant le cycle de vie, ne dispose que de deux attributs. Le premier est la disponibilité de la documentation, mesurée par une métrique *Presence*. Le second est la qualité de cette documentation, mesurée par une métrique *Ratio*. La sous-caractéristique additionnelle *Self-Contained* est représentée par un attribut. Celui-ci indique le taux de dépendabilité du composant, mesuré par une métrique *Ratio*. Une autre sous-caractéristique additionnelle, *Configurability*, est représentée par un attribut. Celui-ci indique l'effort en temps, mesuré par une métrique *IValues*, pour configurer correctement le composant. Les sous-caractéristiques qui composent la nouvelle caractéristique *Marketability* sont mesurées statiquement à partir des informations disponibles pour chaque composant : documentation, site web, etc...

L'avantage de CQM est que comme le modèle de Bertoa et Vallecillo, il est complet à tous les niveaux : caractéristiques, sous-caractéristiques, attributs qualité et métriques. Il est également spécifique aux composants, et les métriques sont simples d'emploi. Par contre, comme pour le modèle de Bertoa et Vallecillo, les auteurs ne proposent aucun mode de comparaison entre deux composants utilisant le même modèle qualité. Et surtout, l'application de leur modèle est limitée par la pauvreté de la documentation des propriétés non-fonctionnelles des COTS sur les marchés aux composants [6], ainsi que par le manque de standard dans le format de cette documentation.

4.4 En résumé

Dans ce chapitre, nous avons présenté plusieurs travaux dédiés à la description et à la comparaison des propriétés du composant, qu'elles soient fonctionnelles ou non-fonctionnelles. Leur défaut commun est de ne considérer que les propriétés fonctionnelles ou les propriétés non-fonctionnelles, mais pas les deux. Sur le plan fonctionnel, il s'agit de mécanismes proposant une substitution adaptée aux possibilités des composants logiciels. L'un de ces mécanismes combine les sous-typages déjà existants, en particulier dans le monde objet. Le second ne considère que le sous-ensemble du composant à remplacer qui est vraiment utilisé par ses voisins dans l'application. Sur le plan non-fonctionnel, plusieurs techniques de description existent. On citera notamment les langages de contrats de qualité de service et les métriques, telles que celles utilisées dans les modèles de qualité pour les COTS. Cependant, seul le langage CQML propose une méthode de comparaison de propriétés non-fonctionnelles. Cette méthode ne concerne que la qualité de service, elle met toutes les propriétés décrites sur le même plan, et elle est trop rigide pour être utilisée telle quelle. Il reste à déterminer si ces techniques de description offrent une base solide pour la comparaison non-fonctionnelle. Ce sont les métriques qui représentent la manière la plus simple de décrire et de mesurer la qualité dans son ensemble. Ce travail de thèse va donc utiliser les métriques comme base pour la comparaison, tout en empruntant aux contrats de qualité de service quelques concepts intéressants.

Dans cette partie, nous avons vu un état de l'art des domaines concernant l'approche présentée dans ce mémoire de thèse. Tout d'abord nous avons survolé l'ensemble des travaux sur la sélection de composants et la prise de décision multi-critères. Nous avons constaté à cette occasion que malgré les nombreuses innovations apportées au fil des ans par les différents processus, l'évaluation locale des candidats pour chaque critère est toujours manuelle.

Après avoir constaté qu'il serait avantageux d'automatiser le plus possible ces évaluations locales, les chapitres suivants ont été consacrées aux techniques de recherche et de comparaison dont on pourrait se servir dans ce but. Pour les évaluations gros-gain, servant à la pré-sélection des candidats pertinents, les techniques de classification externe telles que la recherche par mots-clés ou la classification par facettes sont très appropriées. Pour les évaluations de granularité fine, servant à l'évaluation détaillée, le sous-typage de signature ainsi que matching de signature peuvent servir efficacement. Cependant, ces techniques ne sont prévues que pour la partie fonctionnelle des composants. C'est pourquoi des techniques de description des propriétés non-fonctionnelles ont été présentées, afin de pouvoir définir un opérateur de comparaison permettant d'entrelacer à la fois les propriétés fonctionnelles et non-fonctionnelles des composants.

Nous allons maintenant rentrer dans le cœur de la thèse en présentant une approche multi-niveaux pour les composants. Afin de sélectionner les composants efficacement et rapidement, cette approche utilise et adapte les techniques existantes au sein d'un concept unique : celui de composant recherché.

Troisième partie

Contribution de la thèse

Dans le chapitre 1, nous avons énoncé un certain nombre de problèmes rencontrés lors de la phase de sélection de composants. Parmi eux, on trouve notamment : la description du besoin, l'opérateur de comparaison, les propriétés à comparer et comment les comparer. Afin d'adresser ces problèmes, nous proposons dans cette partie une approche multi-niveaux pour la sélection de composants.

D'une part, nous présentons dans le chapitre 5 des outils conceptuels de description et de comparaison de composants. En particulier, nous introduisons le concept de composant recherché. Ce concept sert à modéliser un besoin d'une application sous forme de composant, de manière à comparer celui-ci à un composant candidat "concret". La comparaison se fait sur la base d'un format commun de description fonctionnelle et non-fonctionnelle, et d'une pondération des éléments de ce format. Un mécanisme hiérarchique effectue la comparaison en l'adaptant au niveau de granularité souhaité. Les nombreuses techniques utilisées, issues de divers domaines, permettent l'automatisation de la comparaison sur chacun de ces niveaux.

D'autre part, nous présentons dans le chapitre 6 des méthodes d'utilisation de ces outils conceptuels dans la sélection de composants. Un indice de satisfaction utilisant le mécanisme de comparaison hiérarchique est présenté avec sa pondération associée. Cet indice permet d'évaluer la similarité des composants candidats avec le composant recherché. Un autre mode de comparaison qui prend la suite de la satisfaction est l'effort d'adaptation requis pour que le candidat corresponde complètement au composant recherché. Ces deux modes de comparaison peuvent être combinés, comme d'autres, dans une stratégie de sélection cohérente, dont nous présentons quelques exemples. Ces stratégies peuvent être à leur tour englobées dans un processus de sélection systématique.

Pour finir, le chapitre 7 présente la validation de cette approche au moyen d'un outil et de deux exemples illustratifs. Chaque exemple exécute le processus de sélection selon l'une des stratégies présentées dans le chapitre précédent.

Chapitre 5

Description et comparaison de composants

Sommaire

5.1 Introduction

Dans ce chapitre, nous allons présenter des outils conceptuels permettant de décrire les composants sur plusieurs niveaux, et d'automatiser leur comparaison. La section 5.2 introduit le concept de composant recherché. Ce concept consiste à modéliser un besoin d'une application par un composant virtuel. Répondre à ce besoin revient à remplacer ce composant par un candidat "concret". Le concept de composant recherché décrit non seulement les propriétés fonctionnelles et non-fonctionnelles attendues, mais aussi la manière dont sont traitées les écarts avec les propriétés des candidats. La section 5.3 traite du premier aspect de ce concept, en proposant un format de description regroupant les concepts communs à la plupart des modèles de composant existants. La section 5.4 traite du deuxième aspect de ce concept, en introduisant

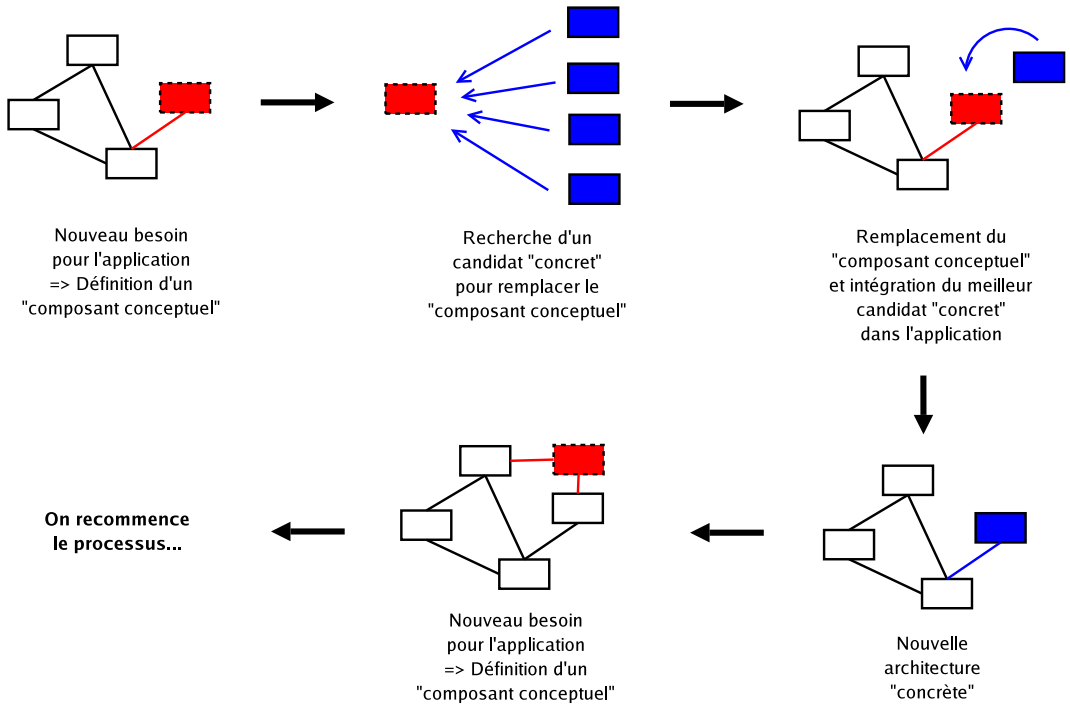

FIG. 5.1 – Construction incrémentale de l'architecture.

la pondération afin de traiter des disparités entre descriptions de composants. Enfin, dans la section 5.5, nous présentons un mécanisme de comparaison hiérarchique qui utilise ces deux aspects pour comparer un composant candidat et un composant recherché sur une base commune. Ce mécanisme permet d'adapter la comparaison au niveau de granularité souhaité. Il permet aussi d'automatiser la comparaison au moyen de plusieurs techniques de recherche et de comparaison existantes.

5.2 Notion de composant recherché

Lors de la conception d'une application, l'étape de modélisation architecturale fait émerger un certain nombre de composants. Quelques-uns de ces composants ont été placés dans le modèle d'architecture car le concepteur sait qu'une implantation concrète de ceux-ci sont à sa disposition. Les composants restants manifestent, par contre, des fonctionnalités dont il a besoin mais dont il ne sait pas si une implantation concrète existe. Ces derniers composants peuvent être qualifiés, à ce stade, de "conceptuels". Comme le montre la figure 5.1, chacun d'eux va faire l'objet d'une recherche spécifique dont le but sera d'identifier dans les marchés et bibliothèques un composant concret capable de se substituer intégralement à eux, ou tout du moins à même de constituer un ersatz à coût d'adaptation et d'intégration le plus bas possible. L'intégration d'un composant concret en place d'un composant "conceptuel" va peut-être

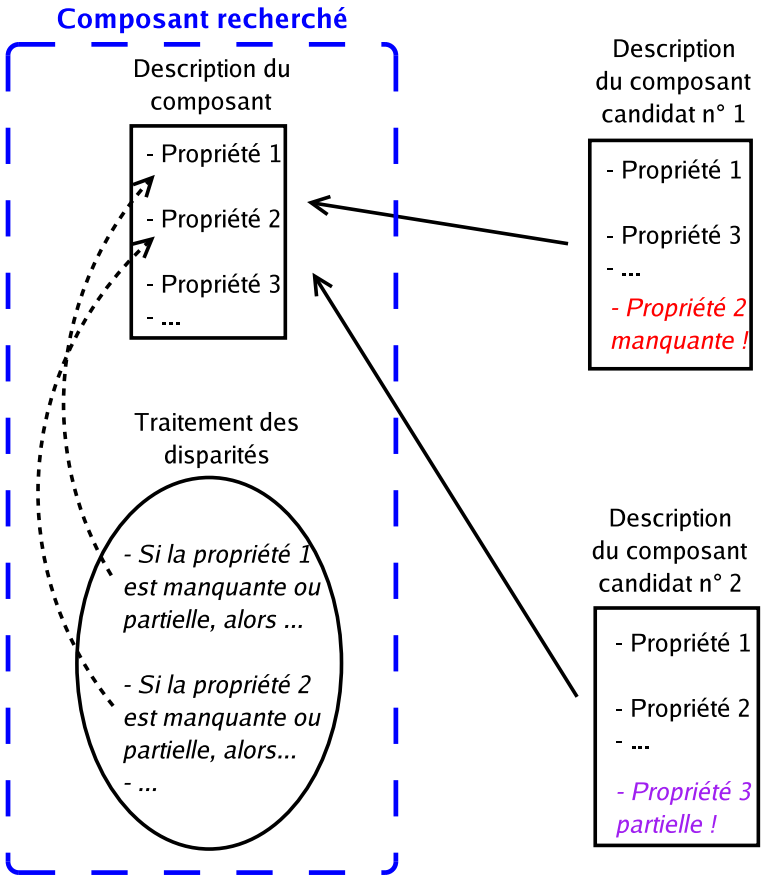

FIG. 5.2 – Structure bicéphale du concept de composant recherché.

imposer une modification du modèle d'architecture. Ces modifications peuvent concerner les composants conceptuels restants ou l'ajout de code supplémentaire pour permettre une collaboration effective avec des composants concrets. La construction d'une application à base de composants est donc incrémentale et itérative, ainsi que l'illustre la figure 5.1. Chaque composant conceptuel est partiellement déterminé par la partie courante de l'architecture qui est "concrète". À chaque fois que l'un d'entre eux est remplacé par un composant concret, celui-ci modifie éventuellement par effet de bord l'architecture de l'application. C'est en fonction de cette nouvelle architecture que sont définis les composant conceptuels restants dont on va chercher à leur tour une implantation concrète. De sorte que l'on est face à un processus de raffinement nécessitant la recherche d'un seul composant conceptuel à la fois.

Lors d'une recherche, la description du composant "conceptuel" concerné manifeste un besoin. Ce besoin est intégré dans une structure que nous appellerons "composant recherché". Suivant le nombre des composants candidats à parcourir et/ou les résultats d'une précédente recherche on peut être amené à procéder soit par filtrages successifs, soit à élargir le champ de la recherche. Cette nécessité impose de disposer pour le composant recherché d'une description

potentiellement multi-niveaux. Au plus haut niveau, on évaluera les candidats sur la base de mots-clés contenus dans la documentation du composant recherché, ou sur la base de critères plus abstraits comme la réputation du constructeur, ou la technologie utilisée. Au niveau de granularité le plus fin, on évaluera la signature des opérations contenues dans les interfaces des composants candidats. C'est pourquoi le composant recherché doit être une structure capable d'embrasser tous ces niveaux de description. Un composant recherché n'est donc pas un composant classique. Sa structure particulière est dictée par l'usage que l'on doit en faire : adapter la comparaison au nombre de candidats et au niveau de granularité exigé. Cette structure permet aussi d'utiliser dans une même comparaison des techniques issues de domaines divers.

Le simple calcul d'une différence ensembliste entre la liste des propriétés listées par le composant recherché à un certain niveau de granularité et celles offertes par un candidat concret est insuffisante pour permettre un classement pertinent. Se limiter à une telle approche c'est admettre que la présence, l'absence ou le respect partiel de deux propriétés ont les mêmes conséquences. Or l'absence ou le respect seulement partiel de certaines propriétés peut être plus ou moins pénalisant. Le cas le plus fréquent est d'ailleurs celui-là. Dans le monde des COTS en particulier, il est rare de tomber sur un composant candidat qui répond parfaitement à toutes les attentes du concepteur. Par conséquent, comme le montre la figure 5.2, le composant recherché ne doit pas seulement englober une description de composant, mais aussi une manière de traiter les disparités entre sa description et celles des candidats. Au final le concept de composant recherché doit présenter la structure bicéphale décrite sur la figure 5.2 : le premier aspect de cette structure offre une description multi-points de vue des propriétés souhaitées ; le second aspect associe à chacune des propriétés du premier niveau un moyen d'estimer la pénalité associée à son absence ou à son respect partiel vis-à-vis des autres propriétés.

5.3 Format de description pour les composants

Comme il a été expliqué dans le chapitre 1, à ce jour il n'existe pas de format de description de composants faisant consensus. Chaque bibliothèque ou marché possède sa propre manière de documenter les composants qu'elle héberge, et au sein d'une même bibliothèque peuvent cohabiter des composants décrits selon des modèles variés. Or, il est impératif que tous ces composants candidats puissent être comparés au composant recherché selon un même format de description. De plus, ce format doit être suffisemment abstrait pour englober les concepts communs à la plupart des modèles existants. C'est dans ce but qu'a été défini le format de description présenté dans cette section. Ce format est décrit au moyen d'un modèle UML (figure 5.3) dont nous allons présenter chacun des éléments.

5.3.1 Les artefacts architecturaux

Trois types d'artefacts ont été retenus(figure 5.4) : les composants, constitués de deux ensembles d'interfaces (fournies et requises, ce qui correspond respectivement aux relations *provided* et *required*), elles-mêmes constituées chacune d'un ensemble d'opérations. Cela nous rapproche de la définition des composants sur laquelle se base de nombreux modèles comme

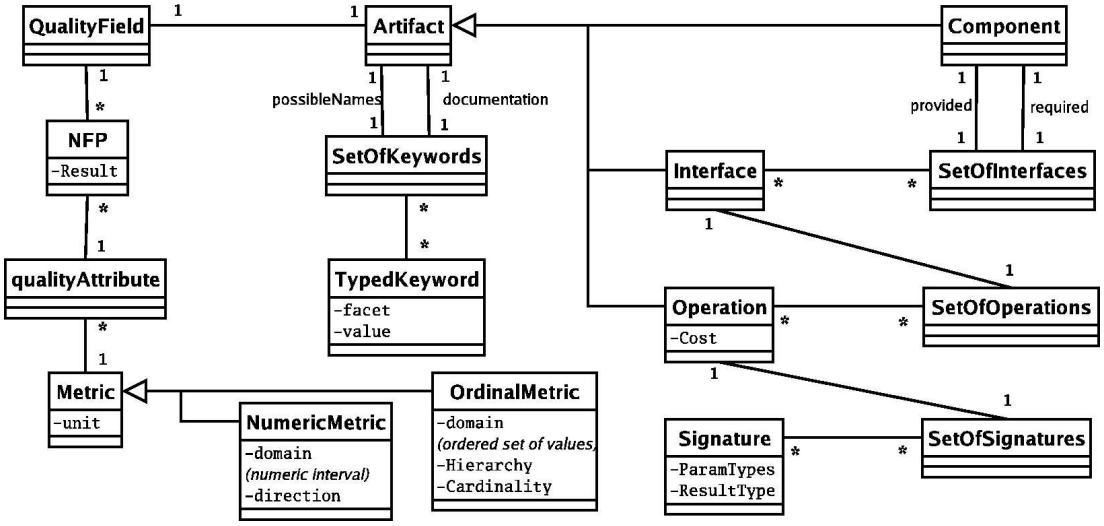

FIG. 5.3 – Modèle UML de format de description des composants.

UML 2.0 [215]. Comme ici nous ne nous intéressons qu'aux propriétés externes des composants et pas à leur structure interne, nous ne prendrons pas en compte le cas des composants "composites", c'est-à-dire contenant d'autres composants. Dans le composant recherché, l'ensemble des interfaces requises représente ce dont le composant a besoin pour fonctionner. Soit c'est déjà inclus dans l'architecture concrète de l'application, soit c'est quelque chose que l'utilisateur a prévu. En tout cas, il sait qu'il peut compter dessus.

À chaque opération est associée non pas une, mais plusieurs signatures. Il est en effet utile de prévoir plusieurs signatures afin d'augmenter la performance lors de la recherche d'un service. Une signature d'opération $S = ParamTypes \rightarrow ResultType$ détaille les types des paramètres $ParamTypes = (\tau_1, ..., \tau_n)$ ainsi que le type du résultat *ResultType*. Si l'on prend l'exemple d'une opération de création de dossier, elle pourrait être proposée avec la signature $string \rightarrow void$. C'est le cas, par exemple, pour l'opération *MakeDirectory* fournie par la version ActiveX du composant FTP de *PowerTCP*. Ce composant appartient à la section "communication internet" de *ComponentSource* [63]. Le paramètre *string* est alors le nom du dossier à créer. Mais une autre signature pourrait être $string \rightarrow boolean$; le résultat de type *boolean* indiquant si l'opération est un succès ou un échec. C'est le cas, par exemple, de l'opération *CreateDirectory* du composant *FTPWizard*, lui aussi appartenant à la section "communication internet" de *ComponentSource*. Pour des raisons de cohérence, on imposera que le même ensemble de signatures ne contienne pas deux fois la même signature. À chaque opération on associe également un attribut supplémentaire, *Cost*, qui indique ce que cela coûterait (en jours-hommes) de développer cette opération soi-même. C'est ce coût qui détermine l'importance de chaque opération à l'intérieur d'une interface.

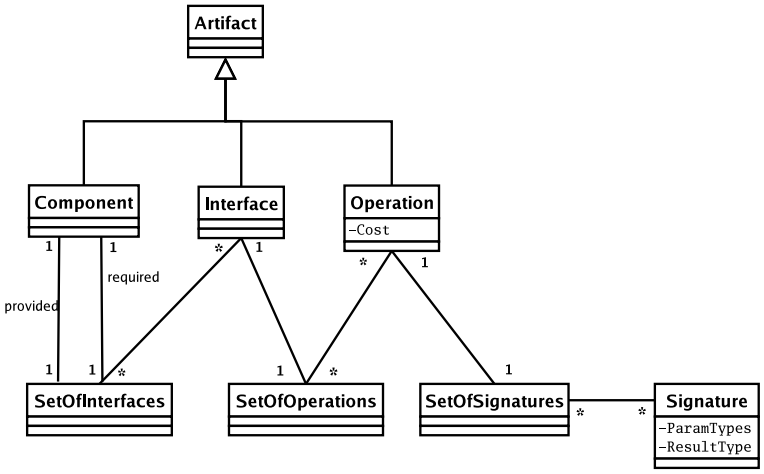

FIG. 5.4 – Artefacts retenus : composants, interfaces et opérations

5.3.2 Informations associées aux artefacts

Deux ensembles de mots-clés (*SetOfKeywords*) sont communs à tous les artefacts (figure 5.5) : un ensemble des noms possibles (relation *possibleName*) pour l'artefact et un ensemble d'informations tenant lieu de documentation (relation *documentation*). Le premier ensemble est introduit car un même artefact peut être proposé sous des vocables différents. À titre d'exemple, une opération de téléchargement peut avoir pour nom *Download* ou $GetFile$, comme c'est le cas, respectivement, pour les composants FTP de *ComponentSpace* et de *Xceed*, disponibles sur le site de *ComponentSource* [63]. Pour des raisons de cohérence, on imposera qu'un ensemble de noms possibles ne contienne pas deux fois le même nom. Le second ensemble permet d'adjoindre à tout artefact un ensemble d'informations. Chacune de ces informations est appelée un "mot-clé typé" (*TypedKeyword*). Il s'agit d'une paire de chaînes de caractères ($facet$, $value$). Le deuxième élément du couple est la valeur du mot-clé typé, positionnée dans un certain domaine d'interprétation que l'on nomme "facette". Il est ainsi possible d'associer à un artefact composant une documentation stipulant que son concepteur est la société NBOS et qu'il a été développé avec la technologie EJB. Il suffit pour cela de lui associer une documentation comportant deux facettes "Publisher" et $Technology$ elles mêmes associées respectivement aux mots clés "NBOS" et "EJB".

Un autre ensemble commun à tous les artefacts est le champ qualité (*QualityField*). Cet ensemble regroupe les propriétés non-fonctionnelles des artefacts, que nous allons maintenant étudier en détail.

5.3.3 Propriétés non-fonctionnelles associées aux artefacts

La figure 5.6 se focalise sur la partie non-fonctionnelle d'un artefact. L'idée que tout artefact architectural peut avoir des propriétés non-fonctionnelles est inspirée des langages de

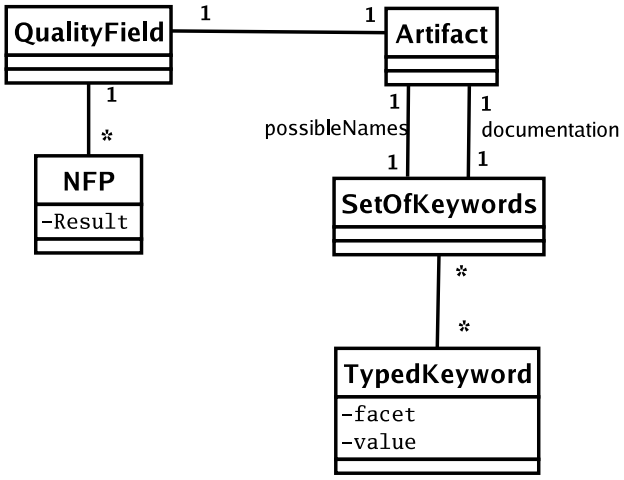

FIG. 5.5 – Éléments communs aux artefacts.

contrats de qualité de service QML [101] et QoSCL [75]. Ici, une propriété non-fonctionnelle (*NFP* sur la figure) représente une valeur (*Result*) obtenue suite à la mesure du niveau d'un attribut qualité sur un artefact. Cette mesure est faite au moyen d'une métrique associée à l'attribut qualité. Cette structure s'inspire des modèles de qualité comme CQM [6], qui étendent le standard ISO-9126 [132] aux COTS.

Les raisons qui ont conduit à choisir les métriques pour représenter et évaluer les propriétés non-fonctionnelles ont été évoquées dans la section 4.3. Il existe des standards pour les métriques, tels que IEEE 1061-1998 [125], pour lequel une même caractéristique ou sous-caractéristique de qualité peut être mesurée par plusieurs métriques, et réciproquement. Cependant, il se peut qu'un même attribut qualité soit mesuré par des métriques dont les types diffèrent d'un modèle à l'autre. Prenons l'exemple de deux modèles de qualité différents : celui de M. Bertoa et A. Vallecillo [20, 21] et CQM [6], qui ont été présentées dans la sous-section 4.3.2. Les deux modèles associent parfois les mêmes attributs qualité aux sous-caractéristiques du standard ISO-9126 [132]. Cependant, les métriques utilisées pour mesurer ces attributs sont parfois différentes d'un modèle à l'autre. Par exemple, l'attribut *Controllability* associé à la sous-caractéristique *Security* est mesuré par un pourcentage dans le modèle de Bertoa et Vallecillo, tandis qu'il est mesuré par un booléen dans CQM. Et pour mesurer l'attribut *Customizability* associé à la sous-caractéristique *Changeability*, le premier modèle utilise un entier, tandis que le second utilise un pourcentage.

L'intégration du standard IEEE 1061-1998 dans le format de description suppose donc l'existence dans la littérature de données systématiques permettant de comparer les valeurs obtenues pour un même attribut qualité avec des métriques différentes. Ce n'est malheureusement pas le cas. En conséquence, dans ce format de description, nous considèrerons qu'un attribut qualité ne peut être mesuré que par une seule métrique, même si une métrique peut mesurer plusieurs attributs.

Les attributs qualité et les métriques utilisés peuvent être tirés d'un des modèles de qualité

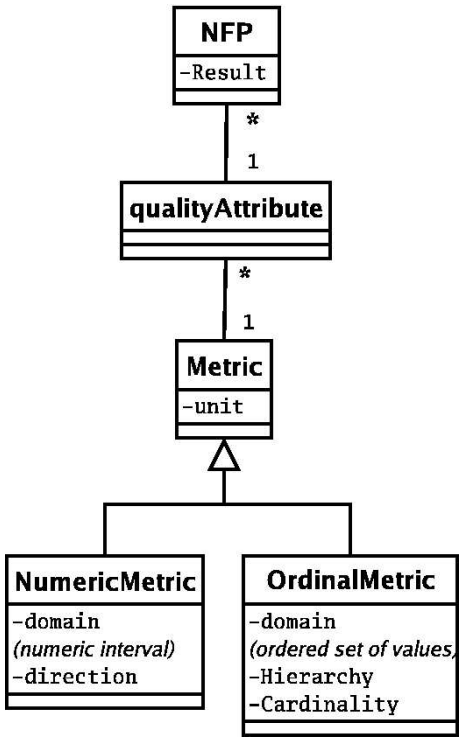

FIG. 5.6 – Partie non-fonctionnelle du format de description.

que nous venons de voir. Des exemples de tels attributs qualité sont le temps de réponse ou l'utilisation de la mémoire. Ou alors, ils peuvent être tirés d'un modèle de qualité fourni par un marché aux composants tel que *ComponentSource* [63]. Dans tous les cas, la comparaison entre deux composants doit se faire sur la base d'un modèle de qualité commun.

Une métrique peut être numérique (*NumericMetric* sur la figure 5.6) ou ordinale (*OrdinalMetric*). Cette distinction est inspirée du projet CLARIFI [279, 26], qui proposait de différencier les différents types de métriques correspondant à leur domaine de valeurs. Le domaine d'une métrique numérique est un sous-ensemble des nombres réels (entiers, pourcentage...). Le domaine d'une métrique ordinale est, pour sa part, un ensemble fini et totalement ordonné. Par exemple, $\{$*mauvais*, *moyen*, *bon*, *excellent*$\}$, ou bien le domaine booléen $\{$*faux*, *vrai*$\}$.

Une métrique numérique possède un attribut supplémentaire appelé direction. Cette direction permet d'interprèter le résultat d'une métrique. Les directions utilisables sont *increasing* ("croissante") ou *decreasing* ("décroissante"). Cette distinction est inspirée des langages de contrats de qualité de service tels que QML [101] ou QoSCL [75]. Une direction croissante signifie que la qualité est d'autant meilleure que la valeur de la métrique est élevée. Un exemple de métrique numérique de direction croissante est le nombre de messages stockés et délivrés en une seconde par un composant servant de file de messages. À l'inverse, une direction décroissante signifie que la qualité est d'autant meilleure que la valeur de la métrique est basse.

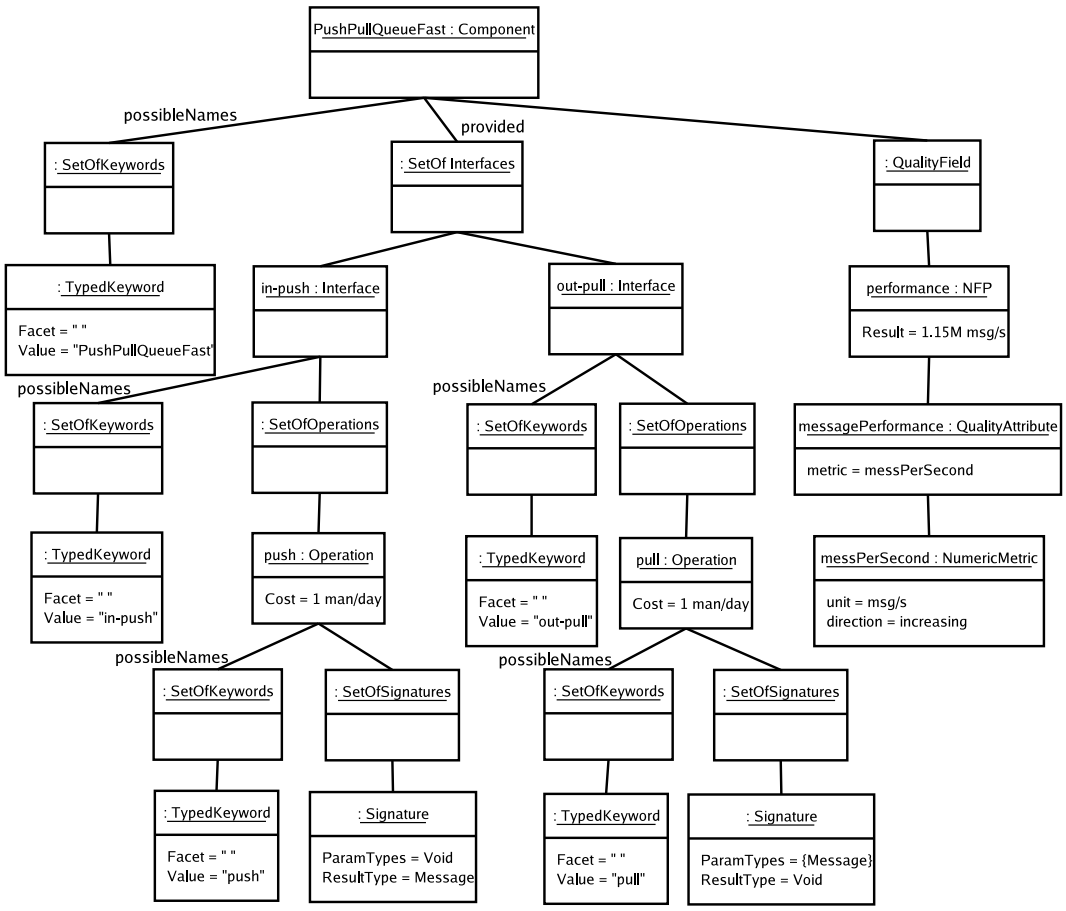

FIG. 5.7 – Exemple de description de composant selon notre format.

Un exemple de métrique numérique de direction décroissante est le temps de réponse d'une opération.

Une métrique ordinale possède deux attributs supplémentaires appelés cardinalité et hiérarchie. La hiérarchie regroupe toutes les valeurs possibles de la métrique en leur associant une clé qui symbolise leur rang. C'est ce rang qui définit la relation d'ordre totale sur le domaine de la métrique. La valeur de rang le plus faible est supérieure strictement à la valeur de rang le plus élevé. Par exemple, si une métrique ordinale M a pour domaine $\{mauvais, moyen, bon, excellent\}$, alors la hiérarchie correspondante est $Hierarchy(M)$=[(1, *excellent*), (2, *bon*), (3, *moyen*), (4, *mauvais*)], et le rang de la valeur *moyen* est égal à $rank(moyen) = 3$. La cardinalité d'une métrique ordinale est le cardinal de son domaine (dans l'exemple $Card(M)$=4).

5.3.4 Exemple de description d'un composant

La figure 5.7 montre un exemple de composant décrit selon le format de description que nous venons de présenter. Ce composant est issu de la bibliothèque offerte par le projet DREAM [166,

237]. Il est téléchargeable, comme tous les autres composants de cette bibliothèque, sur la page Web du projet [236]. Ce composant fait office de file de messages : il stocke ceux-ci et les délivre rapidement. Son nom, *PushPullQueueFast*, apparaît dans son ensemble des noms possibles comme valeur de mot-clé typé. Il en va de même pour les autres artefacts. En revanche, pour des raisons de place, nous avons préféré ne pas développer sur la figure la documentation des artefacts. Le composant est constité de deux interfaces fournies, *in-push* et *out-pull*. Ces interfaces contiennent chacune une unique opération, respectivement *push* et *pull*. La première opération sert à stocker les messages, elle prend en paramètre un objet de type *Message* et ne retourne pas de résultat. La deuxième opération sert à délivrer le premier message de la file, elle ne prend pas de paramètre et retourne un objet de type *Message*. Le coût de développement des deux opérations a été estimé à 1 jour/homme. D'un point de vue non-fonctionnel, les résultats des tests de performance affichés sur le site du projet DREAM [235] indiquent que le composant *PushPullQueueFast* peut créer 1.15 million de messages par seconde. Cela correspond à une NFP *Performance*, associée à l'attribut qualité *messagePerformance*. Cet attribut qualité est mesuré par la métrique numérique *messPerSecond*, dont l'unité est le nombre de messages par seconde et la direction est croissante. Conformément au résultat du test de performance, le résultat de la NFP est de 1.15 million. Cet exemple illustre la capacité de notre format de description à exprimer non seulement les propriétés fonctionnelles, mais aussi les propriétés non-fonctionnelles du composant.

5.4 Traitement des disparités entre les candidats et le composant recherché

Nous venons de définir le premier aspect de la structure d'un composant recherché, c'est-à-dire le format de description de ses propriétés fonctionnelles et non-fonctionnelles. Nous allons maintenant nous pencher sur son deuxième aspect, qui traite des discordances entre les propriétés d'un composant recherché et celles d'un composant candidat. En effet, il faut prévoir le cas où un candidat manque d'une propriété recherchée, ou ne la respecte que partiellement. Il faut également classer des candidats qui pour certains satisfont pleinement la propriété *A* mais pas la propriété *B*, et pour d'autres se retrouvent dans le cas inverse. Ce qui pose la question de savoir quelle propriété, de *A* ou de *B*, est la plus importante.

Dans le cas du format défini précédemment, nous avons vu qu'un composant décrit dans ce format contient des propriétés fonctionnelles et non-fonctionnelles. Supposons que l'on compare deux composants candidats à un composant recherché selon ce format. L'un des candidats possède toutes les fonctionnalités attendues, mais est de piètre qualité. L'autre n'a pas toutes les fonctionnalités demandées dans le composant recherché, mais il est de bien meilleure qualité que le premier candidat. Lequel faut-il choisir dans ce cas ? La question se pose également vis-à-vis de l'effort. Par exemple, un candidat peut possèder la plupart des opérations attendues, mais celles qui manquent peuvent totaliser malgré tout un coût de développement très élevé. À l'inverse, un autre candidat peut manquer de beaucoup d'opérations recherchées, mais ce seraient des opérations peu coûteuses à développer soi-même.

Pour résoudre ce problème, on associe au format de description une pondération permettant

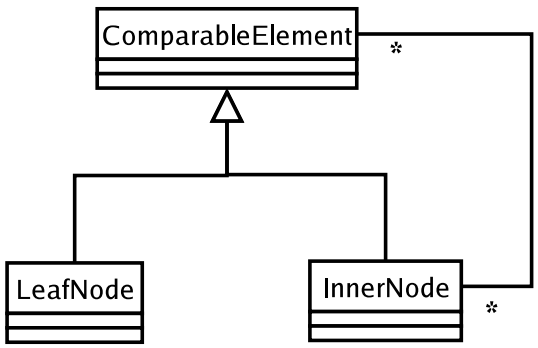

FIG. 5.8 – Structure en arbre du mécanisme de comparaison.

de décrire pleinement le composant recherché. Elle consiste à attacher un poids aux éléments du format de description afin d'exprimer l'importance de chacun par rapport aux autres. Le sens qu'on donne à ces poids ainsi que la manière de pondérer varient selon le mécanisme de pondération choisi. Un exemple de mécanisme est proposé dans la sous-section 6.2.2. Lorsque chaque élément d'un composant candidat est comparé à un élément correspondant dans le composant recherché, le poids de ce dernier élément influe sur la comparaison. Il permet d'estimer l'importance réelle de cette comparaison dans l'évaluation globale du composant candidat.

5.5 Comparaison de composants

Maintenant que nous savons décrire aussi bien le composant recherché que les composants candidats dans un format commun, nous pouvons aborder le problème de la comparaison entre deux composants décrits selon ce format. Dans cette section, nous allons introduire un mécanisme de comparaison adapté au format de description. Ce mécanisme offre un cadre commun de comparaison pour les éléments de différents niveaux appartenant à ce format. En cela, il correspond à la nature hiérarchique du composant recherché. Il permet également l'emploi de plusieurs techniques et sémantiques de comparaison différentes. Afin d'intégrer complètement le format de description dans ce mécanisme, nous détaillerons les comparaisons qui ne sont pas génériques.

5.5.1 Mécanisme de comparaison

Une lecture attentive du format de description proposé permet de distinguer une description de nature fortement hiérarchique. Dans ce format un composant est décrit au moyen d'un arbre "d'éléments comparables" (cf. figure 5.8) dont la racine est un artefact de type composant et dont les noeuds fils sont potentiellement : un ensemble d'interfaces fournies, un ensemble d'interfaces requises, un ensemble de noms possibles, une documentation, et un champ qualité. De même, un noeud de type interface peut à son tour disposer de noeuds fils qui sont :

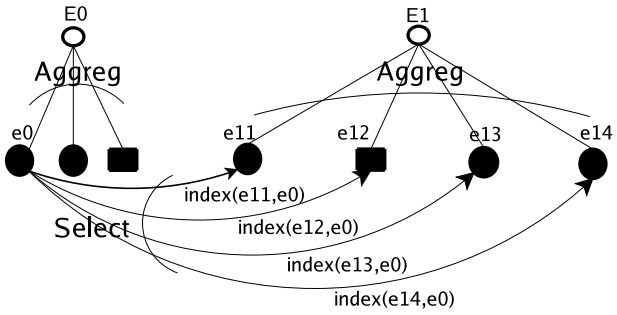

FIG. 5.9 – Mécanisme de comparaison.

un ensemble d'opérations, un ensemble de noms possibles, une documentation et un champ qualité. La fonction de comparaison doit donc de façon récursive comparer deux noeuds *a* et *b* d'arbres distincts en comparant leur noeud fils respectifs deux à deux, puis "agréger" le résultat des comparaisons des sous-noeuds pour calculer l'écart (ou la similarité selon la sémantique choisie) entre *a* et *b*. Cette démarche est la même qu'elle que soit le type des deux noeuds à comparer.

Nous allons donc décrire ce calcul de manière générique indépendamment du type des éléments concernés, qu'ils soient feuilles (*LeafNode* sur la figure 5.8) ou nœuds (*InnerNode* sur la figure). À chaque élément comparable E sont associés un type (noté $Type(E)$) et une fonction de pondération (notée $Weight(E)$). Le type permet d'assurer qu'on ne compare entre eux que les éléments de même nature : les interfaces avec les interfaces, les opérations avec les opérations... Pour deux éléments de type différent, l'indice de comparaison renverra une valeur de non-comparaison *NoComp* (par exemple, 0). La fonction de pondération renvoie pour chaque élément une valeur numérique appelée poids, qui indique son importance vis-à-vis des autres éléments de l'arbre.

On obtient le mode de calcul décrit à la figure 5.9. Pour chaque élément fils e0 du nœud recherché E0, on calcule un indice de comparaison avec chacun des éléments fils du nœud candidat E1 (respectivement, e1, e2, e3 et e4). On récupère le meilleur résultat grâce à la fonction *Select*, qui prend en paramètre un ensemble de valeurs et retourne celle qui satisfait le mieux un certain critère de comparaison (par exemple, la valeur la plus élevée de l'ensemble {index(e11,e0), index(e12,e0), index(e13,e0), index(e14, e0)}). On multiplie ce meilleur résultat par le poids de e0, puis on recommence l'opération pour tous les autres éléments fils de E0. Enfin, on agrège ces meilleurs résultats ainsi pondérés grâce à la fonction *Aggreg* (qui peut être par exemple une addition), pour obtenir l'indice total de comparaison entre E1 et E0.

Aggreg et *Select* sont définis pour chaque type d'élément noeud. En ce qui concerne les feuilles de l'arbre, chaque type de feuille possède son propre mode de comparaison local, qui consiste à calculer l'écart entre une feuille candidate et une feuille recherchée, puis à interpréter cet écart (au moyen d'une fonction f), afin d'obtenir l'indice de comparaison entre les deux feuilles. L'écart est calculé au moyen d'une fonction (*Disc*) qui renvoie une valeur entre 0, qui signifie que les feuilles sont équivalentes, et 1, qui signifie que les feuilles sont totalement

différentes. Par exemple, pour deux mots-clés, *Disc* déterminera si le mot-clé candidat équivaut ou non au mot-clé recherché. Formellement parlant, l'indice de comparaison *Index* entre un élément candidat E_1 et un élément recherché E_0 est défini comme suit :

$$Index(E_1,E_0) = \begin{cases} \text{. } NoComp \text{ si } Type(E_1) \neq Type(E_0). \\ \text{. } f(Disc(E_1,E_0)) \text{ si } E_0 \text{ est une feuille.} \\ \text{. } Weight(E_0) * Aggreg(\{Weight(e_0) * Select(\{Index(e_1,e_0) \mid e_1 \in E_1\}) \mid e_0 \in E_0\}) \\ \text{si } E_0 \text{ est la racine de l'arbre.} \\ \text{. } Aggreg(\{Weight(e_0) * Select(\{Index(e_1,e_0) \mid e_1 \in E_1\}) \mid e_0 \in E_0\}) \\ \text{si } E_0 \text{ est un autre type de nœud.} \end{cases} \tag{5.1}$$

La sélection s'effectue en calculant l'indice de comparaison entre chaque candidat de la bibliothèque et le composant recherché, puis en choisissant le candidat qui a le "meilleur" indice selon la sémantique donnée par les fonctions de sélection (*Select*), d'agrégation (*Aggreg*), de pondération (*Weight*) et de comparaison entre feuilles ($f(Disc)$, avec une fonction *Disc* par type de feuille). La question des sémantiques de comparaison et de pondération est traitée plus en détail dans le chapitre 6.

5.5.2 Intégration du format de description dans l'arbre

Comme le montre la figure 5.10, presque tous les éléments de ce format peuvent être considérés comme des noeuds de l'arbre puisqu'ils peuvent être comparés entre eux et peuvent se décomposer en d'autres éléments de l'arbre. Il y a deux types d'exceptions. D'une part, même si les attributs qualité et les métriques sont utilisés dans la comparaison entre NFPs, ils ne sont pas comparés directement entre eux. Ils ne font donc pas partie de l'arbre. D'autre part, les NFPs, les signatures et les mots-cés typés ne se décomposent pas en d'autres éléments de l'arbre. On doit donc les considérer comme trois types de feuilles et définir pour chacun d'entre eux une fonction de calcul d'écart spécifique. C'est l'objet de cette section.

5.5.2.1 Calcul d'écart entre NFPs

Soient A_0 un artefact recherché, P_0 l'une des NFPs de son champ qualité, A_1 un artefact candidat de même type que A_0, et P_1 une des NFPs de son champ qualité.

P_1 n'est comparable à P_0 que si les deux NFPs mesurent le niveau du même attribut qualité, et dans ce cas, la métrique qu'elles utilisent sera notée M. Si M est numérique, la fonction de comparaison mesurera la différence entre la valeur de P_1 et celle de P_0 en fonction de la direction de M. Si M est ordinale, la fonction de comparaison mesurera la différence entre le rang de la valeur de P_1 et celle de P_0.

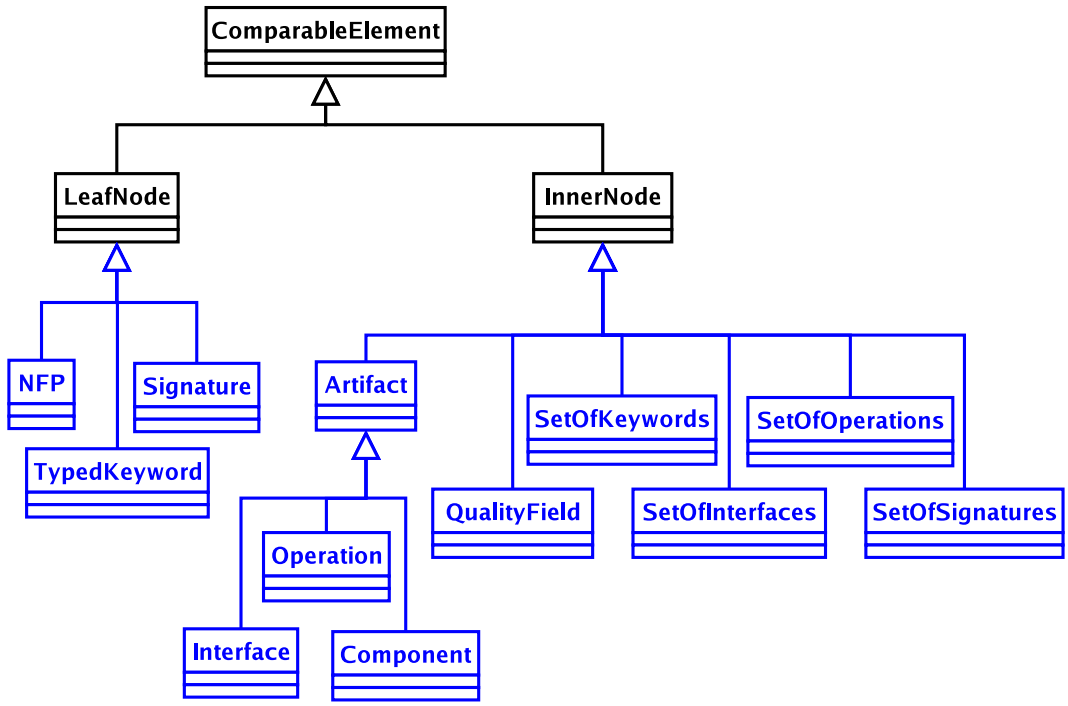

FIG. 5.10 – Éléments du format de description intégrés dans l'arbre de comparaison.

Formellement parlant, l'écart entre P_1 et P_0 est défini comme suit :

$$Disc(P_1,P_0) = \begin{cases} 1 & \text{si } P_1 \text{ et } P_0 \text{ ne mesurent pas le même attribut qualité.} \\ Disc_{inc}(P_1,P_0) & \text{si } M \text{ est numérique avec une direction croissante.} \\ Disc_{dec}(P_1,P_0) & \text{si } M \text{ est numérique avec une direction décroissante.} \\ Disc_{ord}(P_1,P_0) & \text{si } M \text{ est ordinale et si } rank(Result_{P_0}) < Card(M). \\ 0 & \text{si } M \text{ est ordinale et si } rank(Result_{P_0}) = Card(M). \end{cases} \tag{5.2}$$

Avec :

$$Disc_{inc}(P_1,P_0) = MAX(0, 1 - \frac{Result_{P_1}}{Result_{P_0}}) \tag{5.3}$$

$$Disc_{dec}(P_1,P_0) = MAX(0, 1 - \frac{Result_{P_0}}{Result_{P_1}}) \tag{5.4}$$

$$Disc_{ord}(P_1,P_0) = MAX(0, 1 - \frac{Card(M) - rank(Result_{P_1})}{Card(M) - rank(Result_{P_0})}) \tag{5.5}$$

Supposons par exemple que P_0 indique le temps d'exécution d'une opération recherchée, et que P_1 soit une NFP d'une opération candidate. Si P_1 ne mesure pas le même attribut qualité

que P_0, l'écart est total. Sinon, elles partagent une même métrique numérique et décroissante. On applique donc $Disc_{dec}$ aux deux NFPs.

Supposons maintenant que P_0 et P_1 indiquent le "niveau" d'une qualité particulière, au moyen d'une métrique ordinale de type $\{mauvais, moyen, bon, excellent\}$. Si la valeur de P_0 est fixée à *bon*, son rang est 2. Pour que l'écart entre P_1 et P_0 soit nul, il faut que la valeur de P_1 soit de rang 1 ou 2. En d'autres termes, la valeur de P_1 doit être *excellent* ou *bon*.

5.5.2.2 Calcul d'écart entre mots-clés typés

Soient K_1 un mot-clé candidat et K_0 un mot-clé recherché. Si la facette de K_0 est vide (" "), cela signifie qu'on recherche un mot-clé sans s'occuper de sa catégorie. Dans ce cas, on ne comparera que les attributs *value* de K_1 et K_0. Si par contre la facette de K_0 n'est pas vide, on lui comparera celle de K_1, et leurs attributs *value* seront également comparés. Pour des raisons de simplicité, nous considèrerons que l'écart entre mots-clés typés sera "tout ou rien". Autrement dit, si les chaînes de caractères comparées n'ont pas la même valeur, alors l'écart sera total.

Formellement parlant, l'écart entre K_1 et K_0 est défini comme suit :

$$Disc(K_1,K_0) = \begin{cases} 0 & \text{si } value_{K_1} = value_{K_0} \text{ et } (facet_{K_0} = \text{" "} \text{ ou } facet_{K_1} = facet_{K_0}). \\ 1 & \text{sinon.} \end{cases} \quad (5.6)$$

Supposons que l'utilisateur ait besoin d'un composant particulier, dont il connaît le nom. Dans ce cas, le composant recherché aura comme nom possible un mot-clé typé avec une facette vide et la valeur égale au nom recherché. Il faudra alors que le nom du composant candidat contienne exactement cette valeur. Si par contre l'utilisateur a besoin d'un composant d'une technologie particulière, par exemple EJB, la documentation du composant recherché comportera un mot-clé typé contenant la facette "Technology" et la valeur "EJB". Il faudra alors que la documentation du composant candidat contienne un mot-clé typé de même facette et de même valeur.

5.5.2.3 Calcul d'écart entre signatures d'opérations

Le calcul d'écart entre signatures d'opération utilise le sous-typage de signature [51], en particulier les règles de contravariance sur les paramètres et de covariance sur les résultats.

Soient une signature recherchée S_0=$ParamTypes_0 \rightarrow ResultType_0$ et une signature candidate S_1=$ParamTypes_1 \rightarrow ResultType_1$. L'écart entre S_1 et S_0 est défini comme suit :

$$Disc(S_1,S_0) = \begin{cases} \text{. 0 si } ParamTypes(S_0) \text{ est sous-type de } ParamTypes(S_1) \\ \quad \text{et si } ResultType(S_1) \text{ est sous-type de } ResultType(S_0). \\ \text{. 1 sinon.} \end{cases} \quad (5.7)$$

Considérons par exemple la signature recherchée $S_0 : float \rightarrow int$. Si S_1 a la même signature, l'écart est nul. Si $S_1 : float \rightarrow float$, alors S_1 est sous-type de S_0, et là encore l'écart est

nul. Si par contre $S1 : boolean \rightarrow int$, cette signature n'est pas sous-type de S_0 et l'écart est total.

D'autres techniques existent pour comparer les signatures d'opération, en particulier le matching de signature tel qu'il a été défini par A. Zaremski et J. Wing [301], qui est plus flexible que le sous-typage. Cependant, toutes les relations de matching ne sont pas automatisables car elles nécessitent une approche collaborative.

5.6 En résumé

Ce chapitre était consacré à la description et à la comparaison de composants. La notion centrale de composant recherché permet de décrire non seulement les propriétés fonctionnelles et non-fonctionnelles du composant, mais aussi la manière dont les disparités avec les candidats sont traitées. Le premier aspect est abordé au moyen d'un format de description de composants, tandis que le second est abordé au moyen d'une pondération. Ces deux outils conceptuels sont englobés dans un mécanisme hiérarchique permettant une comparaison multi-niveaux entre un composant candidat et un composant recherché. Les composants étant vus comme des arbres, leurs feuilles bénéficient de fonctions de comparaison spécifiques, tandis que les nœuds sont comparés récursivement de la même manière. Cela permet d'utiliser plusieurs techniques de recherche et de comparaison différentes selon le niveau de granularité souhaité. Cela permet également l'automatisation de l'évaluation des candidats.

Les outils conceptuels ayant été proposés, il reste maintenant à définir les moyens de les utiliser, ainsi que le but de leur utilisation. En d'autres termes, il faut exploiter leur nature hiérarchique et multi-niveaux afin de définir différents modes de comparaison et de pondération. Ceux-ci seront ensuite combinés afin de sélectionner efficacement le "meilleur" composant candidat parmi toute une bibliothèque. C'est le sujet du chapitre 6.

Chapitre 6

Stratégies et processus de sélection

Sommaire

6.1 Introduction

Dans le chapitre 5, nous avons présenté un certain nombre d'outils conceptuels offrant un cadre général pour la comparaison entre un composant candidat et un composant recherché. Il faut maintenant définir la manière de les utiliser et de les combiner afin de sélectionner le "meilleur" candidat. Il faut également donner un sens à la comparaison. En d'autres termes, il faut définir ce qui fait qu'un composant "est meilleur" qu'un autre, sachant que cette notion de "meilleur" peut varier d'un contexte à l'autre.

Dans la section 6.2, nous présentons un indice de satisfaction et la pondération par distribution qui lui est associée. Cette "satisfaction pondérée par distribution" oriente le mécanisme de comparaison afin de sanctionner la similarité entre les composants candidats et le composant recherché. Elle tient également compte du coût de développement des opérations. Dans la section 6.3, nous présentons un autre mode de comparaison basé sur l'effort d'adaptation fonctionnelle des composants candidats. Dans la section 6.4, nous présentons deux exemples de stratégies de sélection combinant ces différents modes de comparaison et de pondération. Enfin, dans la section 6.5, nous présentons un processus de sélection qui part d'un besoin, et

exploite les outils conceptuels et les manières de les utiliser afin de sélectionner le candidat qui répond le “mieux” à ce besoin.

6.2 Indice de satisfaction

Les calculs de score total dans les techniques de prise de décision multi-critères considèrent que le meilleur candidat est celui qui a le score le plus élevé. Un indice de satisfaction, noté $Index_{Sat}$, a donc été défini sur le même principe. On l'obtient en reprenant le mécanisme de comparaison présenté dans la section 5.5, et en lui associant des fonctions de sélection et d'agrégation particulières. L'indice de satisfaction permet de déterminer jusqu'à quel point un composant candidat est proche du composant recherché, c'est-à-dire combien le premier partage de propriétés fonctionnelles et non-fonctionnelles avec le second. Dans cette section nous présenterons d'abord la formule de cet indice. Nous détaillerons notamment les fonctions de sélection et d'agrégation choisies, ainsi que le traitement réservé aux services requis. Nous présenterons ensuite la pondération par distribution associée à cet indice.

6.2.1 Principe et formule

L'indice de satisfaction renvoie une valeur réelle allant de 0, qui représente la valeur de non-comparaison (le candidat n'a rien de ce qu'on recherche) à 1, qui signifie que le candidat possède toutes les propriétés attendues dans le composant recherché. En ce qui concerne les feuilles, on va rechercher la similarité entre elles plutôt que l'écart. Pour chaque type de feuille descendant d'une interface fournie, la fonction de comparaison f va donc calculer $1 - Disc$. En ce qui concerne les nœuds, on va rechercher le plus grand indice de satisfaction possible pour chacun d'entre eux, donc on va sélectionner les indices maximaux, et les additionner. Par conséquent, pour chaque type de nœud à l'exception des ensembles d'interfaces fournies, des ensembles d'interfaces requises et de leur descendance, la fonction de sélection sera MAX, qui extrait la valeur la plus élevée d'un ensemble, et la fonction d'agrégation sera la somme, Σ.

Les ensembles d'interfaces fournies n'agrègent pas leurs éléments de la même manière. En effet, considérons le cas où les opérations recherchées sont réparties entre plusieurs interfaces candidates. Au lieu de choisir entre ces interfaces candidates et se priver d'une partie de ce qu'on recherche, il vaut calculer un indice unique sur l'union ensembliste des interfaces. En ce qui concerne la fonction d'agrégation, ce sera Σ pour les ensembles d'interfaces fournies.

Les ensembles d'interfaces requises et leur descendance nécessitent quant à eux un traitement spécial. Nous avons vu dans le chapitre précédent que les services requis du composant recherché sont quelque chose qui se trouve déjà dans l'architecture concrète de l'appplication, ou que l'utilisateur a anticipé. Mais il se peut qu'un composant candidat requière plus que prévu, c'est-à-dire des services qui ne se trouvent pas dans le composant recherché. Le traitement appliqué aux services requis consiste donc, à l'inverse des services fournis, à savoir si un élément requis candidat trouve un équivalent dans le composant recherché. Entre eux, on ne mesure plus un indice de satisfaction, mais une pénalité spécifique, notée $Penalty$ et comprise entre 0 et 1. Pour chaque fils e_1 d'un nœud requis candidat E_1, on va rechercher le fils e_0 d'un nœud requis recherché E_0 pour lequel leur pénalité multipliée par le poids de e_0 est la plus

petite. On additionne ces résultats minimaux pour obtenir $Penalty(E_0, E_1)$, et on recommence de manière récursive pour toute la descendance des ensembles d'interfaces requises candidat et recherché. Pour des raisons de commodité, on imposera que tous les services requis d'un composant recherché soient regroupés dans une interface unique. Au niveau des ensembles d'interfaces requises, on va donc calculer la pénalité entre l'union ensembliste des interfaces requises du composant candidat et l'unique interface requise du composant recherché. Puis on retranchera cette pénalité à l'indice de satisfaction entre les composants. Cela signifie que plus cette pénalité entre interfaces requises sera élevée, moins le composant candidat sera "satisfaisant".

Formellement parlant, l'indice de satisfaction entre un élément candidat E_1 et un élément recherché E_0 est défini comme suit :

$$Index_{Sat}(E_1, E_0) = \begin{cases} \text{. } 0 \text{ si } Type(E_1) \neq Type(E_0). \\ \text{. } (1 - Disc(E_1, E_0)) \text{ si } E_0 \text{ est une feuille.} \\ \text{. } \Sigma(\{Weight(e_0) * (\{Index(e, e_0) \mid e = \cup\{e_1 \in E_1\}\}) \mid e_0 \in E_0\}) \\ \text{ si } E_0 \text{ est un ensemble d'interfaces fournies.} \\ \text{. } MAX(-1, \{-Penalty(e_0, e) \mid \{e_0\} = E_0 \wedge e = \cup\{e_1 \in E_1\}\}) \\ \text{ si } E_0 \text{ est un ensemble d'interfaces requises.} \\ \text{. } Weight(E_0) * \Sigma(\{Weight(e_0) * MAX(\{Index(e_1, e_0) \mid e_1 \in E_1\}) \mid e_0 \in E_0\}) \\ \text{ si } E_0 \text{ est un composant.} \\ \text{. } \Sigma(\{Weight(e_0) * MAX(\{Index(e_1, e_0) \mid e_1 \in E_1\}) \mid e_0 \in E_0\}) \\ \text{ si } E_0 \text{ est un autre type de nœud qui ne descend pas d'un ensemble d'interfaces} \\ \text{ requises.} \end{cases} \tag{6.1}$$

Avec :

$$Penalty(E_0, E_1) = \begin{cases} \text{. } 1 \text{ si } Type(E_0) \neq Type(E_1). \\ \text{. } Disc(E_0, E_1) \text{ si } E_1 \text{ est une feuille.} \\ \text{. } \Sigma(\{MIN(\{Weight(e_0) * Penalty(e_0, e_1) \mid e_0 \in E_0\}) \mid e_1 \in E_1\}) \\ \text{ si } E_0 \text{ est un nœud qui descend d'un ensemble d'interfaces requises.} \end{cases} \tag{6.2}$$

6.2.2 Pondération par distribution

Nous avons vu dans le chapitre précédent que l'indice de comparaison entre un élément candidat et un élément recherché est multiplié par le poids du second avant d'être agrégé avec les autres indices. La pondération oriente la comparaison et explique comment sont traitées les disparités entre les éléments recherchés et les éléments candidats correspondants. La question se pose de savoir comment pondérer et quel est le sens de ces poids. Par extension, la question est de savoir quel est le sens de la satisfaction.

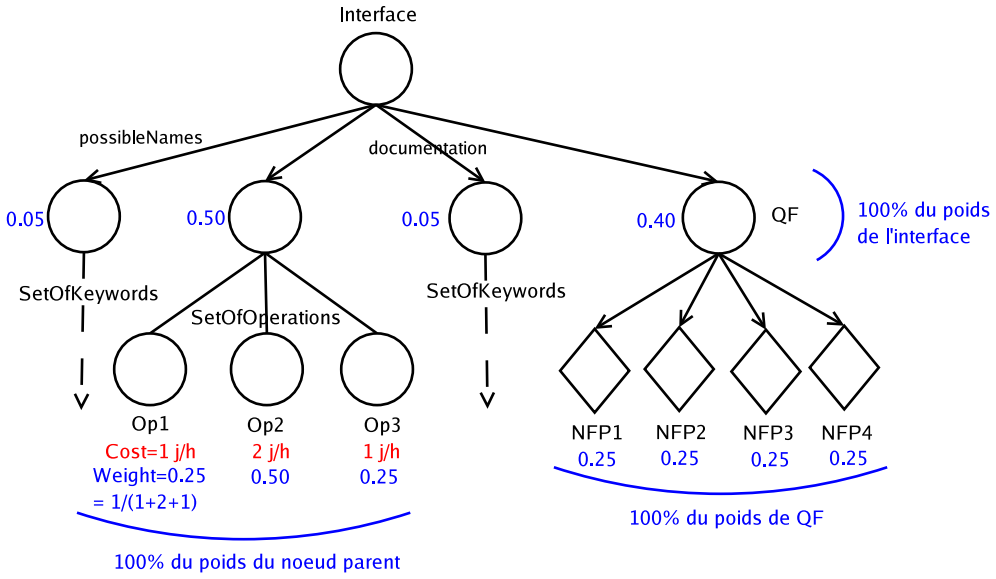

FIG. 6.1 – Pondération par distribution : exemple

En ce qui concerne les mots-clés utilisés dans les noms possibles et la documentation d'un artefact, les poids sanctionnent leur existence. En ce qui concerne les propriétés non-fonctionnelles, le composant candidat doit satisfaire non seulement leur présence, mais aussi le niveau demandé par le composant recherché pour chacune de ces propriétés. Le poids du champ qualité d'un artefact indique l'importance des propriétés non-fonctionnelles par rapport aux propriétés fonctionnelles. En revanche, en ce qui concerne les opérations et leur signature, la question principale est de savoir s'il est plus rentable de développer chaque opération recherchée soi-même, ou de la remplacer par une opération candidate. Les poids des opérations doivent donc dépendre de leur coût de développement, et tenir compte de celui des autres opérations descendant d'une même interface.

La pondération par distribution consiste à tenir compte de l'importance de chaque élément, fonctionnel et non-fonctionnel, du composant recherché. Avec cette pondération, l'utilisateur distribue la totalité du poids de chaque nœud de l'arbre entre tous ses fils directs, depuis le composant qui a un poids de 1, jusqu'aux feuilles (figure 6.1). Chaque poids est borné entre 0 (aucune importance) et 1 (importance maximale). Par exemple, un composant partagera son poids entre son ensemble des noms possibles, sa documentation, son ensemble d'interfaces fournies, son ensemble d'interfaces requises et son champ qualité, de sorte que la somme des poids de ses cinq éléments fils sera égale à 100% de son propre poids.

Il y a trois exceptions à ce partage des poids : l'ensemble des noms possibles, l'ensemble des signatures possibles, et surtout l'ensemble des opérations. D'une part, comme on cherche une seule signature ou un seul nom correct parmi tous ceux que l'on propose, chacun récupèrera la totalité du poids de leur ensemble parent. De plus, on a imposé qu'il n'y ait pas deux fois le même nom ou la même signature au sein d'un même ensemble parent. Il n'y a donc pas de

risque que l'indice de satisfaction de ces ensembles dépasse 1. D'autre part, même si toutes les opérations se partagent une partie du poids de leur ensemble parent, cette répartition est fixée par leurs coûts de développement respectifs. En d'autres termes, le poids de chaque opération est égal à son coût de développement *Cost* divisé par la somme des coûts de toutes les opérations appartenant au même ensemble parent.

Soient e_0 un élément recherché et E_0 son nœud parent. La fonction de pondération par distribution de e_0, notée $Weight_D$, est formellement définie comme suit :

$$Weight_D(e_0) = \begin{cases} \bullet\ 1 \text{ si } E_0 \text{ est un ensemble de noms possibles ou de signatures.} \\ \bullet\ \frac{Cost(e_0)}{\Sigma\{Cost(e) | e \in E_0\}} \text{ si } E_0 \text{ est un ensemble d'opérations.} \\ \bullet\ 1 - \Sigma(\{Weight_D(e) \mid (e \in E_0 \wedge e \neq e_0)\}) \text{ sinon.} \end{cases} \quad (6.3)$$

Prenons l'exemple d'une interface contenant dans sa descendance trois opérations et quatre NFPs (figure 6.1). Si l'on considère que chaque NFP est aussi importante que l'autre, on peut toutes leur attribuer un quart du poids du champ qualité, soit 0.25. Au niveau de l'interface en revanche, si l'on considère que l'ensemble des noms possibles et la documentation sont d'importance moindre par rapport au reste, et que l'ensemble d'opérations est légèrement plus important que le champ qualité, alors le poids de l'interface peut se répartir comme suit : 0.05 pour les deux premiers ensembles, 0.50 pour le troisième, et 0.40 pour le quatrième. En ce qui concerne les trois opérations, leur poids est fixé par leurs coûts de développement. Ici, il en coûterait 1 jour/homme pour développer soi-même la première et la troisième opération, tandis qu'il en faudrait 2 jours/hommes pour la deuxième. Cette dernière aura donc un poids de 2/(1+2+1) = 0.5, tandis que les deux autres auront chacune un poids de 0.25.

6.3 Estimation d'effort

Dans un cycle de développement à base de composants, la phase de sélection des composants est suivie par les phases d'adaptation et d'intégration. Ces phases requièrent un certain effort, qui dépend directement des composants sélectionnés. On peut en déduire que la phase de sélection doit s'effectuer en vue de l'effort futur d'adaptation et d'intégration. La comparaison entre les composants candidats et le composant recherché ne doit pas seulement permettre d'évaluer le taux de satisfaction pour chaque candidat. Elle doit également permettre d'estimer l'effort requis pour amener ce taux de satisfaction à sa valeur maximale. En fin de compte, l'effort est l'ultime critère de sélection d'un composant candidat. Dans cette section, nous présentons un mode de comparaison basé sur ce principe. Ce mode de comparaison se nomme "Estimation globale d'effort", et sa pondération associée s'appelle "pondération par estimation", car ici nous cherchons à évaluer l'effort d'adaptation de chaque composant candidat disponible.

Dans la pondération par distribution associée à l'indice de satisfaction qu'on a vu dans la section précédente, c'est l'effort de développement qui donne son sens au poids des opérations. Cependant, avec une telle pondération, le poids d'une opération recherchée est calculé en fonction de son coût de développement et des opérations appartenant à la même interface unique-

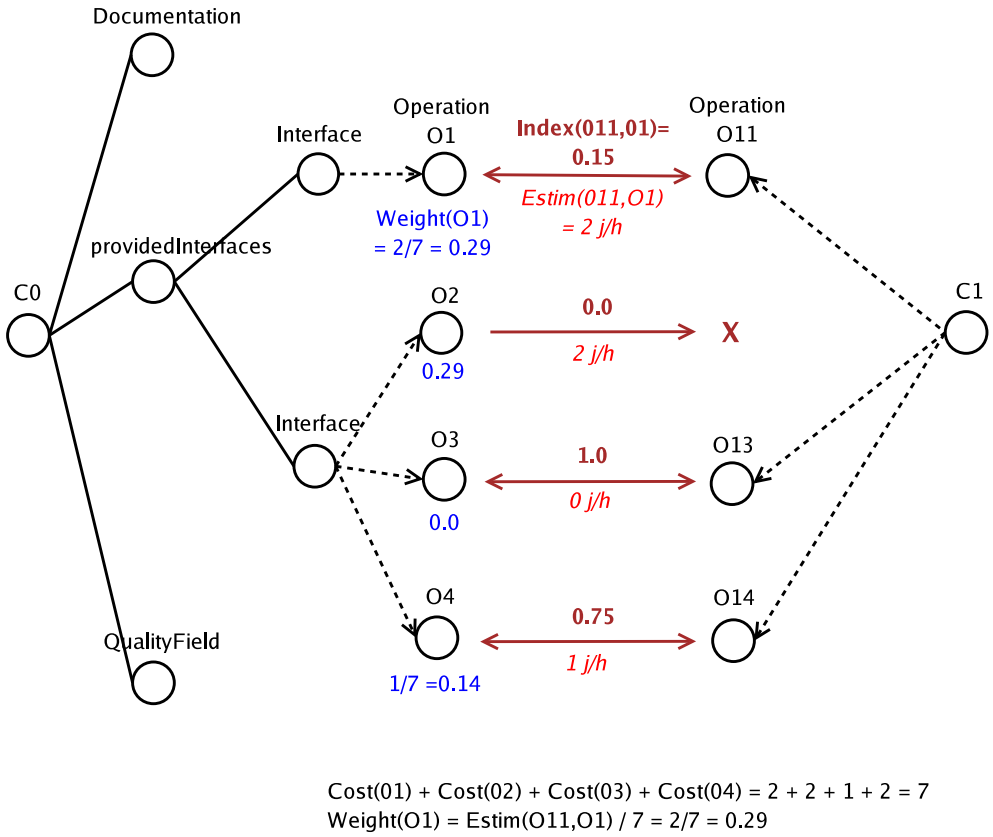

FIG. 6.2 – Estimation globale d'effort : exemple

ment. Le composant recherché est donc pondéré de la même manière quel que soit le candidat qui lui est comparé. L'estimation globale d'effort consiste, quant à elle, à estimer le coût d'adaptation des opérations candidates en fonction de leur écart avec les opérations recherchées. Cette estimation prend en fait la suite de la satisfaction pondérée par distribution, laquelle fait office de filtrage. On l'applique d'abord l'indice de satisfaction avec la pondération par distribution sur tous les candidats. On sélectionne ensuite un nombre restreint d'entre eux, par exemple les trois candidats les plus "satisfaisants" par rapport à cette pondération, et on va regarder leurs opérations. Comme le calcul de la satisfaction a déjà été effectué, on n'a donc plus à tenir compte, ni des autres éléments, ni de leur poids ni même de la répartition des opérations entre différentes interfaces. On ne va donc prendre en compte que les opérations contenues dans le composant recherché, toutes mises au même niveau (figure 6.2).

Pour chaque composant candidat C_1, l'utilisateur va comparer son arbre de description avec celui du composant recherché C_0, et considérer chaque opération recherchée o_0 par rapport à l'opération candidate correspondante o_1. Par "opération candidate correspondante", on entend l'opération candidate qui satisfait le mieux o_0. Si l'indice de satisfaction entre o_1 et o_0 est égal à 1, o_1 n'a pas à être adaptée, et le poids associé à o_0 est nul. Sinon, o_1 nécessite une

certaine adaptation. On va demander à l'utilisateur d'estimer lui-même en jour/homme le coût réel d'adaptation de o_1 afin qu'elle corresponde parfaitement à o_0. Ce coût d'adaptation est noté $Estim(o_1, o_0)$, et ne peut pas dépasser le coût de développement de o_0. Pour obtenir le poids de o_0, on va diviser $Estim(o_1, o_0)$ par la somme des coûts de développement de toutes les opérations recherchées.

Il reste à s'occuper des opérations requises du composant candidat qui n'ont pas d'équivalent dans le composant recherché. Puisqu'elles n'ont pas été prévues, il va falloir les développer soi-même. Pour chacune de ces opérations o_R, l'utilisateur va donc estimer son coût de développement, noté $Dev(o_R)$. Pour obtenir le "poids de pénalité" de o_R, on va diviser $Dev(o_R)$ par la somme des coûts de développement de toutes les opérations recherchées.

L'estimation globale d'effort du composant candidat par rapport au composant recherché s'obtient en additionnant les poids de toutes les opérations recherchées avec les poids de pénalité de toutes les opérations requises candidates non prévues dans le composant recherché. Il se peut cette fois qu'à cause des opérations requises, l'estimation globale d'effort pour un candidat soit supérieure à 1. Cela voudrait dire que l'effort d'adaptation de ce candidat est supérieure au maximum prévu.

Soient o_0 une opération du composant recherché C_0, et o_1 l'opération candidate correspondante appartenant au composant C_1. La fonction de pondération par estimation, notée $Weight_{Est}$, est formellement définie comme suit :

$$Weight_{Est}(o_0) = \begin{cases} \cdot \frac{Estim(o_1,o_0)}{\Sigma(\{Cost(op_0)|op_0 \in C_0\})} \text{ si } Index_{Sat}(o_1,o_0) < 1. \\ \cdot \ 0 \text{ sinon.} \end{cases} \tag{6.4}$$

Soit o_R une opération requise candidate, et C_0 le composant recherché. La fonction attribuant le poids de pénalité de o_R, notée $Weight_P$, est formellement définie comme suit :

$$Weight_P(o_R) = \begin{cases} \cdot \frac{Dev(o_R)}{\Sigma(\{Cost(op_0)|op_0 \in C_0\})} \text{ si } \forall op_0 \in C_0, Penalty(op_0, o_R) = 1. \\ \cdot \ 0 \text{ sinon.} \end{cases} \tag{6.5}$$

Enfin, l'estimation globale d'effort entre C_1 est C_0, noté $Effort$, est formellement défini comme suit :

$$Effort(C_1, C_0) = \Sigma(\{Weight_{Est}(o_0) \mid o_0 \in C_0\}) + \Sigma(\{Weight_P(o_R) \mid o_R \text{ est une opération requise par } C_1.\}) \tag{6.6}$$

Prenons l'exemple d'un composant recherché C0 comprenant une documentation, un champ qualité et deux interfaces fournies (figure 6.2). Ces deux interfaces totalisent quatre opérations O1, O2, O3 et O4. La documentation et le champ qualité n'étant pas comparés, on n'en tient pas compte. Les interfaces n'ayant pas d'importance en elles-mêmes, on n'en tient pas compte non plus. Seules comptent les opérations, mises au même niveau. On sait que la somme des coûts de développement des opérations recherchées est égal à 7. Un composant candidat C1 fournissant 3 opérations est comparé à C0. Considérons l'opération recherchée O1 et l'opération candidate correspondante O11. L'indice de satisfaction entre elles n'est que de 0.15. L'utilisateur estime que c'est trop peu, et que le coût d'adaptation de O11 sera égal à 2 jours/hommes. Le poids de

O1 est donc de : 2/7 = 0.29. En ce qui concerne l'opération 02, elle n'a pas d'opération candidate qui lui convient, et il faut la développer soi-même. Comme son coût de développement est de 2 jours/hommes, son poids est aussi de 2/7. L'opération candidate O13 a un indice maximal avec O3, il n'y a donc pas à l'adapter et le poids de O3 est nul. Enfin, l'opération candidate O14 a un indice de satisfaction de 0.75 avec O4. L'utilisateur estime qu'il y aura peu d'adaptation à faire, et estime ce coût à 1 jour/homme alors que le coût total de développement de O4 était de 2 jours/hommes. Le poids de 04 est donc 1/7 = 0.14. On en déduit l'estimation globale d'effort entre C1 et C0 : 2/7 + 2/7 + 1/7 = 0.71.

6.4 Stratégie de sélection

En regardant attentivement la manière dont l'estimation globale d'effort est introduite, on distingue les différentes étapes d'une stratégie de sélection du "meilleur" candidat. L'utilisation de l'effort suppose en effet que l'indice de satisfaction a été préalablement calculé, et qu'il a servi à filtrer les candidats pour qu'il n'en reste qu'un nombre restreint. Cette étape préliminaire est nécessaire en raison des estimations manuelles demandées à l'utilisateur pour chaque candidat.

D'une manière plus générale, nous avons souligné dans le chapitre précédent la structure bicéphale présentée par le concept de composant recherché. D'une part, elle offre une description multi-niveaux, qui adapte la comparaison au nombre de candidats et au niveau de granularité exigé. D'autre part, elle pondère la comparaison et permet ainsi l'interprétation des disparités entre cette description et celle des composants candidats. Compte tenu des milliers de composants existants éparpillés dans différentes bibliothèques, plutôt que de se contenter d'une seule comparaison qui pourrait être soit trop imprécise, soit trop coûteuse, il est nécessaire de pouvoir réduire progressivement le nombre de candidats tout en augmentant la précision de la description du composant recherché. Pour ce faire, le concept de stratégie de sélection qui est proposé consiste à appliquer successivement plusieurs modes de comparaison et de pondération différents, afin de filtrer progressivement les candidats. Dans cette section nous présentons deux exemples de stratégie. L'une utilise la satisfaction et l'effort, tandis que l'autre utilise la satisfaction avec un autre mode de pondération, par le biais d'un outil de visualisation proposé dans [163].

6.4.1 Une stratégie combinant satisfaction et effort

La figure 6.3 montre une stratégie de sélection à plusieurs niveaux de granularité, qui utilise successivement la satisfaction pondérée par distribution, puis l'estimation globale d'effort :

– La première étape consiste à effectuer une pré-sélection sur toute la bibliothèque de candidats, en utilisant l'indice de satisfaction sur la documentation uniquement. Tous les autres éléments du composant reçoivent un poids égal à zéro. Cela permet de filtrer les candidats les plus pertinents, car seuls ceux contenant exactement la documentation attendue (indice de satisfaction égal à 1) pourront passer à l'étape suivante. Cette pré-

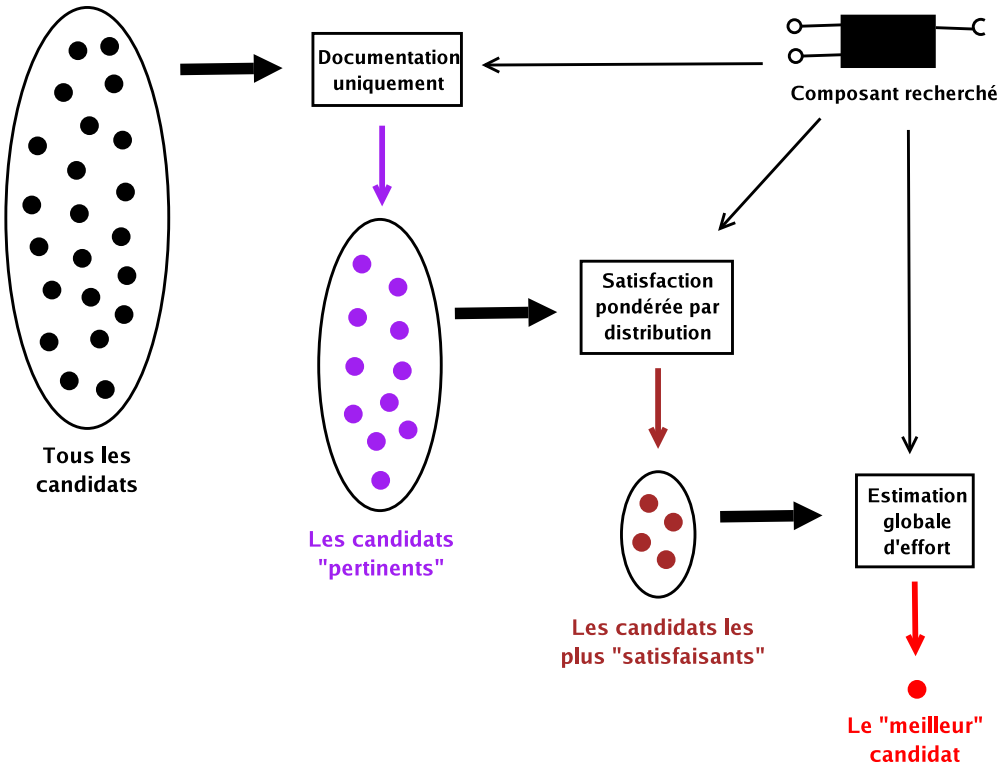

FIG. 6.3 – Distribution, puis estimation.

sélection est nécessaire pour éviter des comparaisons inutiles face à des bibliothèques contenant des centaines de candidats.

– La deuxième étape consiste à associer l'indice de satisfaction à la pondération par distribution. Cela permet de classer et de sélectionner les quelques candidats qui "satisfont le mieux" le composant recherché, sur la base de leur qualité et des fonctionnalités qu'ils partagent avec lui. Comme l'étape suivante sollicite de nombreuses estimations manuelles sur les opérations, le nombre de candidats qui pourront franchir cette étape devra être suffisemment restreint. Par exemple, si le composant recherché contient en tout une dizaine d'opérations, il vaut mieux se limiter à deux ou trois candidats.
– Enfin, la dernière étape consiste à utiliser l'estimation globale d'effort sur chacun des candidats restants afin qu'ils satisfassent complètement le composant recherché. Celui qui demande le moins d'effort est considéré comme le "meilleur" candidat, et c'est lui qui est sélectionné.

Chacune de ces étapes peut être réitérée si les résultats ne sont pas satisfaisants. Par exemple, dans la deuxième étape, il se peut que les indices de satisfaction ne conviennent pas, et cela parce que les poids accordés aux éléments du composant recherché sont mal répartis. Par exemple, on peut avoir accordé trop d'importance à une propriété non-fonctionnelle et pas assez à une autre. On peut donc recommencer les calculs avec le même mécanisme de pondération,

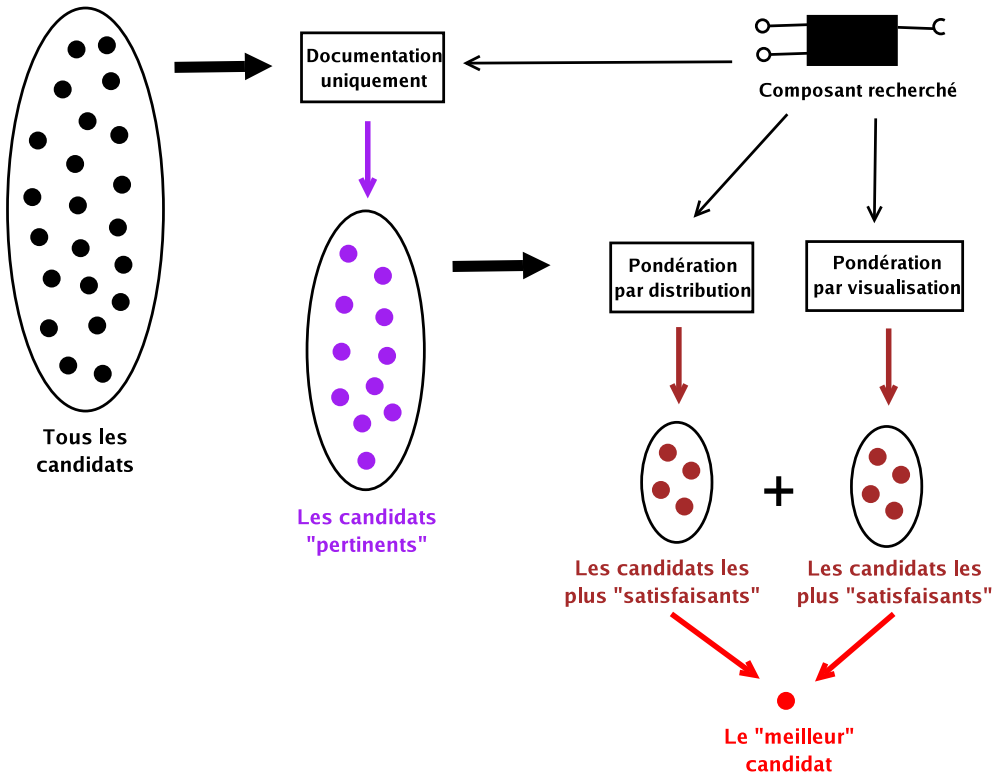

FIG. 6.4 – Distribution et visualisation.

mais en appliquant d'autres poids.

6.4.2 Une stratégie combinant distribution et visualisation

La stratégie de sélection présentée ici est plus adaptée quand on veut un composant spécifique, sans l'adapter. Par exemple, un composant d'une marque particulière avec des signatures particulières. Cette stratégie utilise exclusivement l'indice de satisfaction, et combine la pondération par distribution avec une autre pondération. Cette pondération plus "naturelle" est obtenue grâce à l'outil de visualisation de G. Langelier *et al* [163]. Cet outil permet de représenter chaque candidat par une boîte 3-D, dont la forme est obtenue par agrégation de divers attributs graphiques. Les principaux attributs qui peuvent être affichés pour chaque boîte sont : la taille, la rotation et la couleur. Chaque attribut correspond à un critère de mesure particulier. On peut, par exemple, associer la taille à l'indice de satisfaction pour l'ensemble des interfaces fournies, une couleur particulière à chaque NFP, la rotation à la documentation, etc... Plus l'indice de satisfaction pour un élément particulier du candidat est élevé, plus l'attribut graphique correspondant influera sur la forme de la boîte représentant ce candidat. L'utilisateur n'a plus à privilégier un élément par rapport à un autre en usant de poids numériques, car la pondération

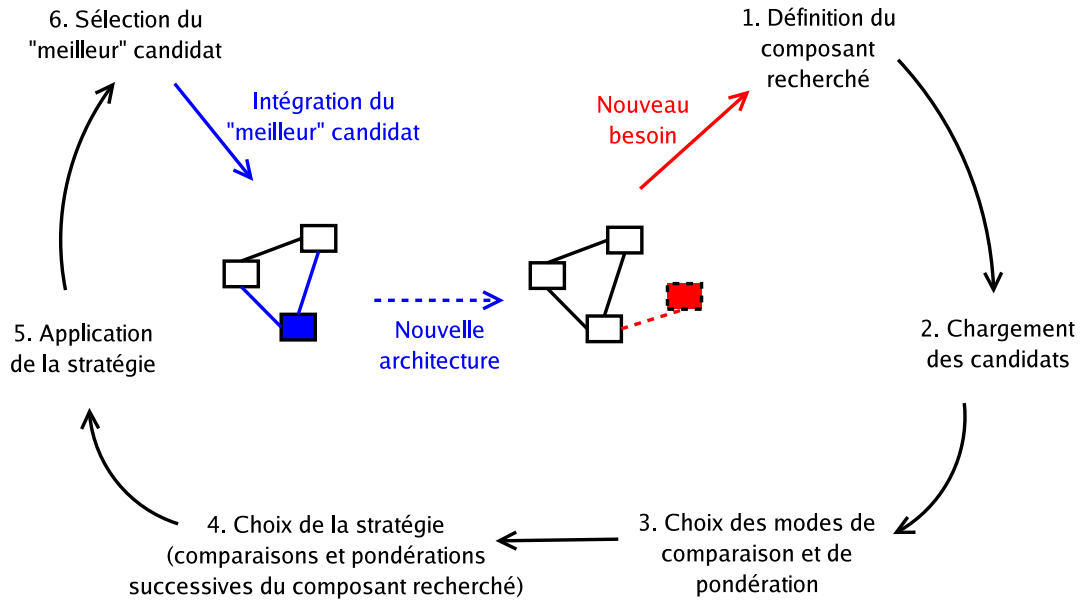

FIG. 6.5 – Processus de sélection de composants.

est visuelle, et les candidats sont évalués en un coup d'oeil. Un exemple d'utilisation de cet outil de visualisation est donné dans la sous-section 7.3.2.

La figure 6.4 montre les différentes étapes de cette stratégie. La première étape consiste, comme pour la précédente stratégie et pour les mêmes raisons, à effectuer une pré-sélection sur toute la bibliothèque de candidats en n'utilisant que la documentation. Les candidats les plus pertinents peuvent ainsi passer à l'étape suivante. Celle-ci consiste à associer l'indice de satisfaction à la pondération par distribution, puis à confirmer le résultat grâce à l'outil de visualisation de Langelier *et al* [163]. Là encore, chacune de ces étapes peut être réitérée si les résultats ne sont pas satisfaisants. Par exemple, on peut recommencer la deuxième étape en distribuant de nouveaux poids, ou en associant de nouveaux attributs graphiques aux différents éléments pour visualiser les résultats.

6.5 Processus de sélection

Une lecture attentive des stratégies précédentes permet de les associer à un même processus de sélection de composants. Leur application présuppose en effet : i) qu'un composant recherché a été défini selon le format de description proposé ; ii) que des composants candidats à son remplacement ont été identifiés ; iii) que l'indice de satisfaction a été choisi comme mécanisme de comparaison, ainsi que l'estimation globale d'effort pour la première stratégie ; iv) que certaines pondérations ont été choisies pour la comparaison. Ce n'est que quand ces étapes préliminaires sont franchies qu'on peut définir et appliquer une stratégie afin

de sélectionner le “meilleur” candidat selon un contexte donné. Nous venons de décrire dans les grandes lignes toutes les étapes d'un processus systématique de sélection, représentées sur la figure 6.5, que nous allons détailler dans cette section :

1. **Définition du composant recherché :** Dans cette étape, le concepteur de l'application utilise le format de description pour spécifier les propriétés fonctionnelles et non-fonctionnelles du composant qu'il recherche. Nous avons vu dans le chapitre précédent que la définition d'un composant recherché dépend de l'architecture “concrète” de l'application, et qu'avec ce concept la construction d'une application à base de composants est incrémentale et itérative. Il en est de même pour le processus de sélection proposé, comme le montre la figure 6.5 : on ne peut définir un nouveau composant recherché que quand on a remplacé le précédent par un candidat “concret”.
2. **Chargement de la bibliothèque de composants candidats :** Dans cette étape, on charge une bibliothèque de candidats parmi celles qui sont fournies. On suppose dans cette thèse que les descriptions de ces candidats selon le format de description choisi sont déjà faîtes.
3. **Choix des modes de comparaison et de pondération :** Une fois que les propriétés du composant recherché sont décrites, et que les candidats sont identifiés, on peut évaluer les fonctions qui seront nécessaires pour les comparer. Cela inclut non seulement les mécanismes de comparaison utilisés (satisfaction, effort...), mais aussi les pondérations associées.
4. **Choix de la stratégie de sélection :** Dans cette étape, on choisit la manière dont se succèderont les différentes comparaisons et pondérations. Cette stratégie, qui exploite la description multi-niveaux du composant recherché, permet de partir d'une vaste bibliothèque de candidats pour arriver progressivement au “meilleur” d'entre eux.
5. **Application de la stratégie :** Cette étape consiste à exécuter la stratégie définie à l'étape précédente. Si elle ne donne rien, on peut revenir en arrière.
6. **Sélection du meilleur candidat :** Une fois que la stratégie choisie a été appliquée et que les composants candidats ont été filtrés et classés, on peut sélectionner celui qui a été déclaré le “meilleur” afin de l'intégrer à l'application. Une fois que c'est fait, l'architecture “concrète” est modifiée et fait apparaître un nouveau besoin, ce qui signifie qu'on peut définir un autre composant recherché et recommencer le processus.

6.6 En résumé

Ce chapitre était consacré à la sélection du “meilleur candidat”. En utilisant les outils conceptuels présentés dans le chapitre précédent, nous avons pu définir différents modes de comparaison et leurs pondérations associées. Le premier d'entre eux concerne la satisfaction des propriétés attendues par le composant recherché. La pondération qui lui est associée consiste à répartir les poids des nœuds entre leurs fils, à l'exception des opérations, dont l'importance dépend de leur coût de développement. Le second se concentre sur l'estimation globale de l'effort d'adaptation des candidats afin que la satisfaction fonctionnelle soit totale. La pondération qui lui est associée consiste à demander à l'utilisateur d'estimer le coût d'adaptation de chaque opération candidate.

Ces mécanismes peuvent être utilisés et combinés dans des stratégies de sélection appropriées. L'une d'entre elles utilise successivement plusieurs pondérations sur l'indice de satisfaction, dont une visualisation des résultats au moyen d'un outil dédié [163]. L'autre stratégie combine la satisfaction et l'estimation d'effort afin de filtrer les candidats les plus prometteurs et de choisir celui qui demande le moins d'adaptation. Enfin, ces outils conceptuels et ces stratégies sont englobés dans un processus systématique de sélection de composants. À partir d'un besoin spécifique d'une application, ce processus consiste à définir le composant recherché correspondant, puis à choisir la stratégie de sélection la mieux appropriée pour sélectionner le "meilleur" candidat. Une fois que celui-ci est intégré, l'architecture de l'application est modifiée. Cette modification fait apparaître un nouveau besoin, et le processus peut se répéter, conformément au caractère incrémental de la notion de composant recherché.

L'approche proposée dans ce mémoire de thèse ayant été présentée, il reste à la valider par un outil et des expérimentations. C'est le sujet du chapitre 7.

Chapitre 7

Validation de l'approche

Sommaire

7.1 Introduction

Les chapitres 5 et 6 ont servi à présenter l'approche choisie pour aborder le problème de la sélection de composants. Ce chapitre servira à présenter un outil qui l'implémente (cf. section 7.2) et deux exemples qui l'illustrent (cf. section 7.3). L'outil implémente le format de description et le mécanisme de comparaison présentés dans le chapitre 5, ainsi que l'indice de satisfaction et sa pondération associée, présentés dans la section 6.2. Il propose également une bibliothèque de descriptions de candidats issus du marché aux composants *ComponentSource* [63], qui a servi de base pour les deux exemples. Ces exemples suivent les différentes étapes du processus de sélection présenté dans la section 6.5, selon l'une des stratégies présentées dans la section 6.4. Le premier exemple utilise la satisfaction et la visualisation pour choisir son candidat. Le second exemple utilise la satisfaction pour filtrer les candidats, puis sélectionne au moyen de l'estimation d'effort celui qui requiert le moins d'adaptation.

7.2 Substitute : un outil pour la comparaison de composants

Afin de valider le travail présenté dans cette thèse, un logiciel appelé *Substitute* a été développé. Il s'agit d'un outil de comparaison de tous les composants d'une bibliothèque

avec un composant recherché [70], selon l'approche proposée dans cette thèse. L'outil a été développé en JAVA avec les JDK 5.0, puis 6.0, sous l'environnement Eclipse.

Substitute prend en entrée des fichiers XML décrivant les composants ainsi que le modèle de qualité utilisé par les NFPs de ces composants. XML a été choisi en raison de son organisation arborescente, qui correspond à la structure hiérarchique du composant recherché. Une DTD est utilisée pour décrire les fichiers XML des composants candidats et recherchés dans un format commun. Deux fichiers XML servent à lister respectivement les métriques et les attributs qualité. Cette séparation d'avec les descriptions XML des composants correspond à l'arbre de comparaison proposé. En effet, les métriques et les attributs qualité sont utilisés par les NFPs des composants, mais ils ne sont pas directement comparés entre eux. Ils ne font donc pas partie de l'arbre, et doivent par conséquent être décrits à part.

Après avoir chargé le modèle de qualité, le composant recherché pondéré d'une certaine manière, et la bibliothèque de composants candidats, les indices de satisfaction sont calculés automatiquement pour tous les candidats. *Substitute* étant consacré à la comparaison automatique entre composants, seul l'indice de satisfaction a été implémenté, car c'est le seul qui soit complètement automatisable. L'outil peut afficher aussi bien le résultat des calculs locaux pour les éléments fils (opérations, NFPs...), que le résultat de l'indice global, et ce pour chaque candidat. Si on veut voir les résultats des calculs pour toute la bibliothèque, on peut les sauvegarder dans un tableau Excel.

L'organisation du logiciel *Substitute* est inspirée du patron de conception *MVC* (*Model-View-Controller*), afin de laisser une autonomie aux différentes tâches (chargement, comparaison, affichage...), et rendre le logiciel facilement extensible. *Substitute* est constitué des paquetages suivants, dont l'organisation et l'interaction sont illustrées par la figure 7.1 :

Model : Ce paquetage est consacré au mécanisme de description et de comparaison des composants. Il se constitue de trois sous-paquetages :

- **Elements :** Contient toutes les classes décrivant les éléments du format de description et de l'arbre de comparaison proposés dans cette thèse.
- **Functions :** Contient les classes relatives aux fonction de sélection, d'agrégation et de pondération proposées dans cette thèse.
- **Indexes :** Contient les classes relatives au mécanisme hiérarchique de comparaison et à l'indice de satisfaction.

Xmlparser : Ce paquetage, inspiré du patron de conception *Abstract Factory* ou Fabrique Abstraite, contient les classes qui inspectent des fichiers XML. Ces fichiers représentent les composants ainsi que les ensembles de métriques et d'attributs qualité utilisés. Le parser XML extrait de ces fichiers les informations relatives aux classes du sous-paquetage *Elements*, puis crée les objets correspondants. Des dizaines de candidats représentant des composants concrets de *ComponentSource* [63] sont disponibles, et cette bibliothèque est facilement extensible.

View : Ce paquetage est consacré à tout ce qui concerne l'interface graphique et l'affichage des différents éléments. Il comporte 7 sous-paquetages :

- **Selector :** Contient toutes les classes consacrées à la création de fenêtres popup de chargement de fichiers XML. À chaque fois que le programme demandera de charger

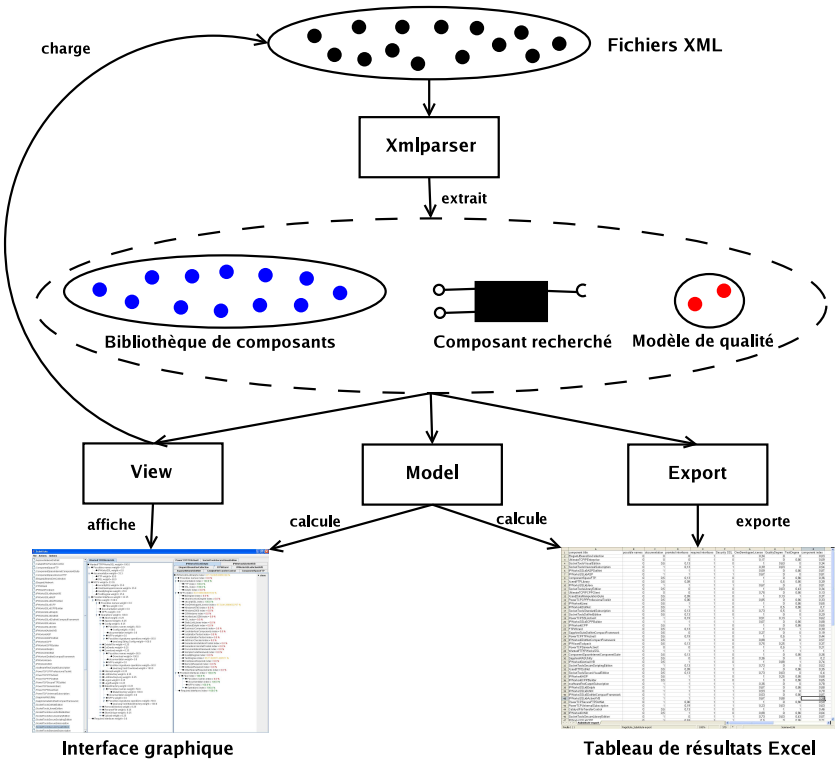

FIG. 7.1 – Substitute : Fonctionnement général et organisation des paquetages

un fichier ou d'en choisir un dans un répertoire, une fenêtre *selector* s'affichera pour que l'utilisateur puisse choisir.

Propertiesloader : Contient toutes les classes de chargement et d'affichage d'attributs qualité et de métriques.

Componentloader : Consacré au chargement du composant recherché.

Libraryloader : Consacré au chargement de la bibliothèque de composants candidats.

Rendering : Consacré à l'affichage graphique des éléments des composants candidats et du composant recherché.

Comparison : Consacré à l'affichage du résultat de la comparaison entre eux.

Progress : Consacré à l'affichage des barres de progression.

Export : Ce paquetage contient les classes consacrées à l'exportation de données. À partir du composant recherché et de la bibliothèque de candidats choisis, ces classes vont enregistrer les résultats de chaque comparaison sous forme de tableau, sauvegardé dans un fichier lisible par *Excel*.

La figure 7.2 montre une capture d'écran de l'interface graphique de *Substitute*. Sur le menu central, on retrouve la description complète du composant recherché qui a été sélectionné, avec

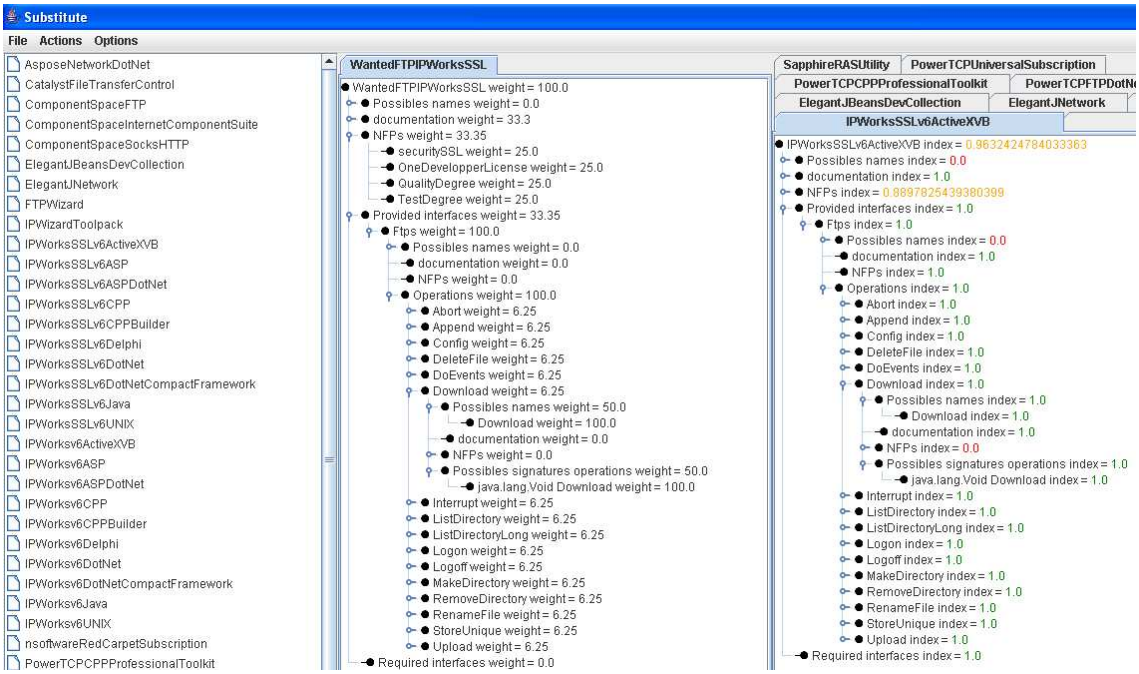

FIG. 7.2 – Substitute : Interface graphique

pour chaque élément le poids qui lui a été attribué. Sur le menu de gauche, on retrouve la bibliothèque de composants sur laquelle on travaille. Et sur le menu de droite, on peut observer le résultat du calcul d'indice de satisfaction entre certains candidats issus de la bibliothèque et le composant recherché. Pour chacun de ces candidats, un onglet affiche le résultat de la comparaison élément par élément. Il est possible de masquer les indices locaux, ou au contraire de dérouler toute l'arborescence jusqu'aux feuilles. Les indices retournant 0, c'est-à-dire la valeur de non-comparaison, sont affichés en rouge. Les indices retournant 1, c'est-à-dire la meilleure valeur possible, sont affichés en vert. Entre les deux, les indices sont affichés en orange.

7.3 Études de cas sur *ComponentSource*

Cette section présente le résultat de deux expérimentations entreprises sur un véritable marché aux composants. Ces expérimentations ont servi à valider aussi bien l'approche proposée que les outils qui la supportent. Pour les deux expérimentations nous nous sommes placés dans le contexte suivant : un concepteur recherche pour son application un composant dédié au FTP (*File Transfer Protocol*) parmi tous les candidats disponibles dans la section "communication internet" du marché aux composants *ComponentSource*. Cette section contenait 131 composants au moment où les tests ont été effectués. Depuis cette base documentaire, une

description XML a été produite pour chaque composant en suivant le format présenté dans la section 5.3. C'est sur cette documentation traduite que le processus de sélection proposé dans la section 6.5 a été testé, et que les présents résultats ont été obtenus. Cette traduction dans le format de description, qui a été réalisée à la main pour ces exemples, peut être elle-même automatisée en utilisant des techniques de transformation de modèles. C'est le sujet d'un travail en cours.

Dans cette section, nous commencerons par détailler le modèle de qualité issu de *ComponentSource*, que nous utiliserons pour les deux exemples. Ensuite, nous détaillerons les deux expérimentations utilisant les stratégies présentées dans la section 6.4. La première expérimentation combine la satisfaction pondérée par distribution et la visualisation au moyen de l'outil de G. Langelier *et al* [163]. La seconde expérimentation combine la satisfaction et l'effort.

7.3.1 Modèle de qualité utilisé

Pour chaque composant disponible, *ComponentSource* fournit une courte description, une présentation du constructeur, un descriptif des différentes licences disponibles, une version d'évaluation à télécharger, éventuellement une documentation fournie par le constructeur, et une section "compatibilités" qui regroupe un ensemble d'informations utiles sur le composant.

Parmi ces informations, on retrouve :

- La liste des systèmes d'exploitation compatibles : Windows 2000, Windows XP, Mac OS X...
- La liste des conteneurs compatibles : Visual Studio 5, Visual Basic, Borland Delphi 6...
- Le type du composant : contrôle ActiveX, classe .NET, JavaBeans, DLL...
- La présence ou non de propriétés de qualité recensées par *ComponentSource* : l'inclusion d'une signature numérique, l'annotation "scriptage sécurisé", l'annotation "initialisation sécurisée", la compatibilité avec certains protocoles...
- L'ensemble des tests effectués et validés par *ComponentSource*, parmi lesquels : l'installation, la désinstallation, la vérification antivirus, l'installation et la désinstallation de la version d'évaluation, l'examen de la documentation, l'examen d'un échantillon de code, et un test avec .NET RCW pour savoir si le composant est *".NET ready"*.

Ces deux derniers ensembles représentent donc respectivement un "degré de test" et un "degré de qualité" qui indiquent le nombre de propriétés vérifiées et attestées par *ComponentSource*. C'est-à-dire, respectivement, jusqu'à quel point le composant a été testé, et quel niveau de qualité il fournit. D'un point de vue non-fonctionnel, cette section "compatibilités" peut être considérée comme le modèle de qualité standard de *ComponentSource* constitué d'un ensemble d'attributs communs à tous ses composants. On peut rajouter à cela la gestion par certains de ces composants du protocole SSL (dans ce cas, le nom et la documentation du composant contiennent les mots "SSL" ou "Secure").

Pour la suite de cette section, et pour des raisons de simplicité, nous nous limiterons à quatre attributs qualité : le degré de qualité *QualityDegree*, le degré de test *TestDegree*, l'attribut *SecuritySSL* qui indique si le composant gère ou non le protocole SSL, et l'attribut *Licence*, qui indique le prix d'une licence pour un développeur. *QualityDegree* et *TestDegree* sont mesurés respectivement par les métriques ordinales *NineDegree* et *EightDegree*, qui indiquent le niveau de qualité d'un attribut de 0 à 9 (respectivement, de 0 à 8). *SecuritySSL* est mesuré par

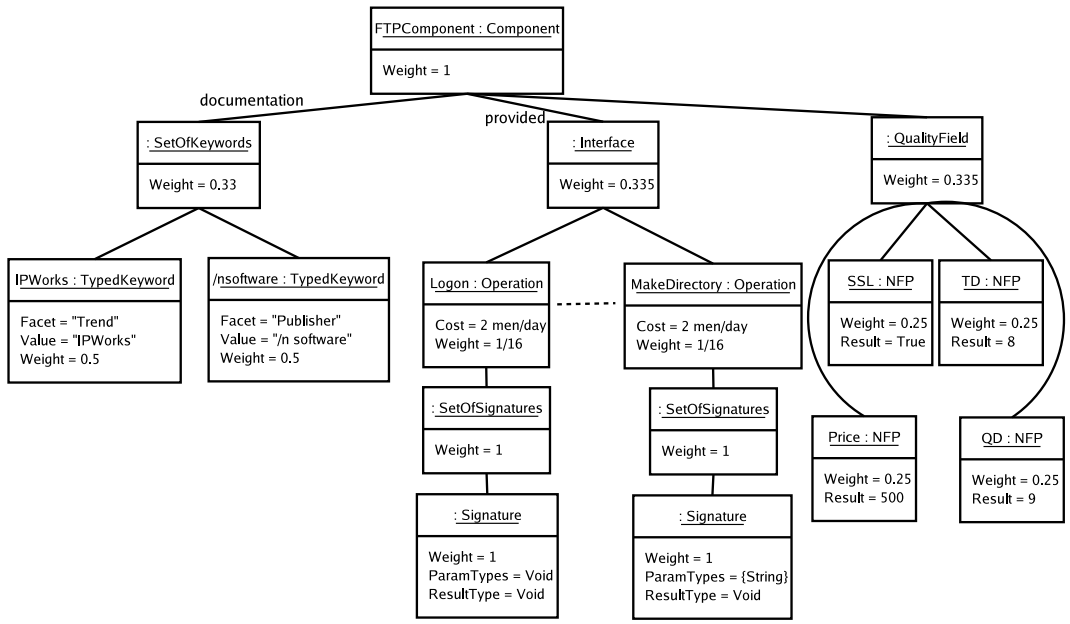

FIG. 7.3 – Composant recherché du premier exemple.

une métrique ordinale *Boolean* qui indique la présence ou non d'une propriété particulière. Et *Licence* est mesuré par la métrique numérique de direction décroissante *DollarPrice*, qui indique un prix en dollars US.

7.3.2 Premier exemple

La figure 7.3 montre le composant recherché entièrement spécifié selon notre format de description. Le concepteur a besoin d'un composant compatible avec la suite logicielle *IP*Works !* vendue par */n software*. D'un point de vue fonctionnel, cela suppose que le composant recherché fournisse une interface dotée de 16 opérations de signature spécifique telles que le téléchargement, la création de dossiers, etc... Le coût de développement de chaque opération a été estimé à 2 jours/hommes. D'un point de vue non-fonctionnel, un certain nombre de qualités sont attendues. Le protocole SSL doit être autorisé, d'où une NFP *SSL* correspondant à l'attribut qualité *SecuritySSL* avec la valeur "True". Ensuite, le composant doit avoir un degré de test élevé, d'où une NFP *TD* correspondant à l'attribut *TestDegree* avec la valeur "8". Le composant doit également avoir un degré de qualité élevé, d'où une NFP *QD* correspondant à l'attribut *QualityDegree* avec la valeur "9". Enfin, le composant ne doit pas être trop cher, d'où une NFP *Price* correspondant à l'attribut *Licence* avec la valeur "500".

Version	Doc	Interface	NFP SSL	NFP Price	NFP QD	NFP TD	**Total**
ActiveX/VB	1.0	1.0	1.0	0.67	0.89	1.0	**0.963**
Delphi	1.0	1.0	1.0	0.67	0.0	0.87	**0.879**
ASP/.Net	1.0	1.0	1.0	0.59	0.0	0.87	**0.872**
.NetCompactFramework	1.0	1.0	1.0	0.53	0.0	0.87	**0.867**
Java	1.0	1.0	1.0	0.67	0.0	0.0	**0.806**

TAB. 7.1 – Indice de satisfaction sur les meilleurs candidats

7.3.2.1 Indices de satisfaction préliminaires

La première étape de notre processus de sélection est donc réalisée. La deuxième étape l'est aussi, puisque les candidats sont connus : il s'agit de ceux de la section "Communication Internet" de *ComponentSource*. Comme nous voulons un composant qui soit à la fois de marque particulière et de très bonne qualité, il ne nous est pas nécessaire d'estimer l'effort d'adaptation des candidats. Seule compte ici la satisfaction des propriétés attendues. C'est pourquoi dans la troisième étape du processus, celle où l'on choisit les modes de comparaison et des pondération, nous allons utiliser l'indice de satisfaction avec la pondération par distribution et la pondération par visualisation [163]. Dans la quatrième étape, celle qui consiste à choisir sa stratégie de sélection, nous allons prendre celle qui est présentée dans la sous-section 6.4.2 car elle combine distribution et visualisation. Pour des raisons de cohérence avec les deux approches, nous avons choisi de pondérer le composant recherché de telle manière que le poids de chaque noeud est distribué équitablement entre tous ses éléments fils. Par exemple, chacune des 4 NFPs décrites précédemment s'est vue attribuer 25% du poids du champ qualité.

Nous pouvons maintenant effectuer la cinquième étape du processus de sélection, qui est l'application de la stratégie choisie. Celle-ci commence par une pré-sélection des candidats sur la seule base de leur documentation. Comme nous voulions mesurer l'indice de satisfaction uniquement sur les candidats susceptibles de traiter de FTP, nous avons effectué cette phase de pré-sélection avec seulement le mot-clé typé de facette "Overview" ("Aperçu") et de valeur "FTP" dans la documentation. Sur les 131 candidats que contenait la section "communication internet" de *ComponentSource*, seulement 55 composants se sont révélés "pertinents". Dans la phase suivante, les indices de satisfaction pour chaque candidat ont été mesurés avec l'outil *Substitute*.

Le tableau 7.1 montre les résultats du calcul des indices de satisfaction pour les 5 meilleurs candidats. Ces candidats sont désignés par le langage pour lequel ils sont conçus (version Java, version ActiveX/VB, etc...) car ce sont tous des composants FTP de marque *IP*Works !* en version sécurisée. De ce fait, ils possèdent tous exactement les opérations demandées dans le composant recherché, et ils autorisent SSL. La véritable différence réside dans le degré de qualité *DQ* ainsi que dans le degré de test *DT*. On constate que c'est la version ActiveX/VB qui est la meilleure, avec un indice de satisfaction de 0.963.

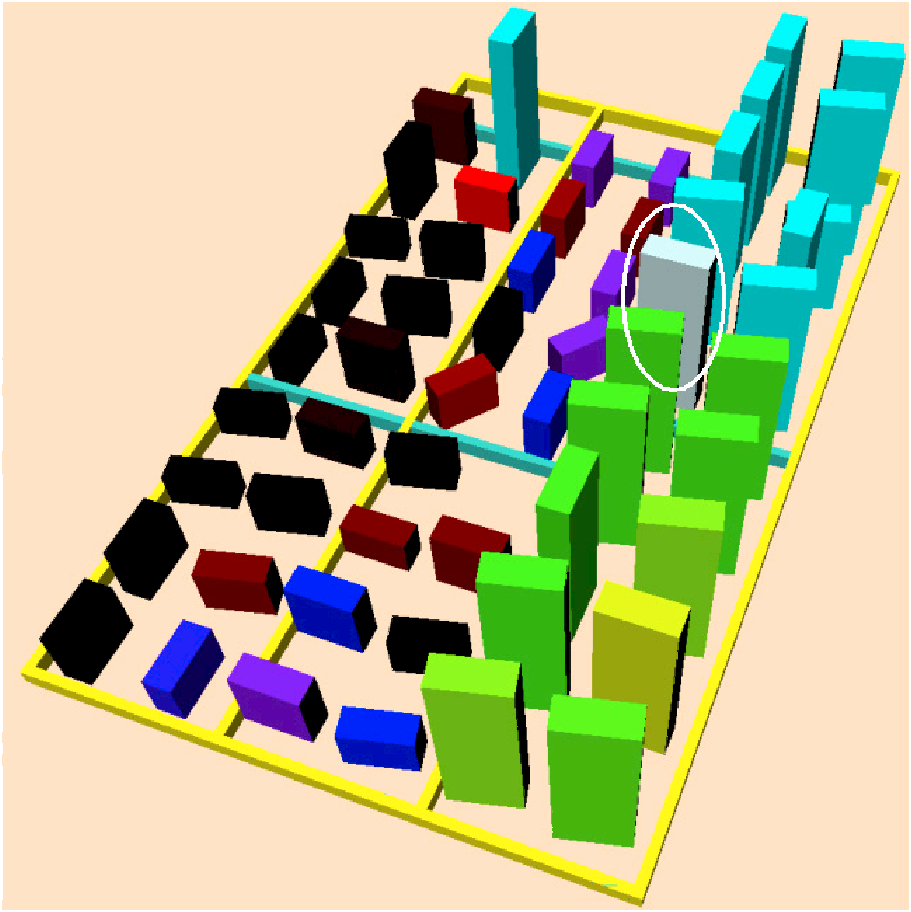

FIG. 7.4 – Visualisation des résultats.

7.3.2.2 Utilisation de la visualisation

Toutefois, ces indices globaux ont été obtenus en imposant une méthode d'agrégation et une pondération particulières. C'est là que l'outil de visualisation de G. Langelier *et al* [163] entre en jeu, afin d'obtenir une pondération plus intuitive et "naturelle". Comme nous l'avons vu dans le chapitre précédent, cet outil représente chaque composant candidat par une boîte 3-D, dont la forme est obtenue par agrégation de divers attributs graphiques : taille, rotation et couleur. Dans notre cas, 5 critères de mesure ont été retenus. On a associé à chacun d'entre eux un attribut graphique particulier : la documentation est représentée en vert, la NFP *SSL* en bleu et la NFP *DQ* en rouge, tandis que la taille représente "l'interface" (combien d'opérations candidates correspondent à une opération recherchée) et la rotation représente la NFP *DT*. Le prix étant un attribut secondaire, il n'est pas visualisé. Plus un candidat possède d'opérations en commun avec le composant recherché, plus sa taille augmentera. Et comme les couleurs se mélangent, les candidats accumulant les indices locaux élevés ont une boîte dont la couleur se rapproche davantage du blanc.

Après avoir calculé les indices de satisfaction avec *Substitute*, l'outil de visualisation a ainsi permis de montrer (figure 7.4) une vue globale des 55 composants candidats affichés et classés selon les critères de mesure initiaux. Vu le nombre de couleurs utilisées sur la figure 7.4, il est recommandé de lire la page sur laquelle elle se trouve en version couleur. Le but était d'identifier selon ces critères les boîtes les plus grandes, dont la couleur était la plus proche du blanc, et dont l'angle était le plus proche de 90 degrés. La boîte qui correspond le plus à cette description est encore une fois la version ActiveX/VB sécurisée du composant FTP d'*IP*Works !*, entouré en blanc sur la figure. D'autres boîtes de grande taille, qui possèdent aussi la plupart des opérations demandées, correspondent à de bons candidats, mais soit leur degré de qualité n'est pas suffisant, soit SSL n'est pas autorisé. Finalement, dans la dernière étape de notre processus de sélection, c'est la version ActiveX/VB sécurisée du composant FTP d'*IP*Works !* qui est choisie, car il s'agit du meilleur candidat aussi bien pour la pondération par distribution que pour la visualisation.

7.3.2.3 Bilan

Cet exemple met en évidence le principal avantage de l'automatisation de la sélection : les indices de satisfaction ont été calculés sur des dizaines de composants en quelques secondes, alors qu'une évaluation manuelle de chaque candidat pour chaque critère aurait pris plusieurs jours. De plus, l'interprétation graphique des résultats à l'aide de l'outil de visualisation offre une autre méthode d'agrégation, qui permet de repérer intuitivement et en un coup d'oeil les meilleurs candidats.

7.3.3 Deuxième exemple

Supposons maintenant que l'utilisateur n'ait plus besoin d'avoir un composant compatible avec une suite logicielle. Il s'agit maintenant de rechercher un composant de marque et de technologie quelconques, mais qui soit doté d'opérations spécifiques et d'une bonne qualité. D'un point de vue fonctionnel, cela suppose que le composant recherché (figure 7.5) fournisse une interface dotée d'un certain nombre d'opérations. Pour l'exemple, nous en retiendrons 10 : création, destruction, renommage et affichage du contenu d'un dossier ; destruction, renommage, téléchargement et mise en ligne de fichiers ; connection et déconnexion. Plusieurs noms et signatures ont été prévus pour ces opérations. Le coût de développement de chaque opération a été estimé à 2 jours/hommes. La somme des coûts de développement de toutes les opérations est donc égale à 20 jours/hommes. La documentation doit contenir un mot-clé typé de facette "Overview" et de valeur "FTP". Lors d'une recherche à haut niveau sur un nombre important de candidats disponibles, la présence ou non de ce mot-clé dans la section"Overview" d'un candidat sera un signe de sa pertinence. D'un point de vue non-fonctionnel, on attend les mêmes qualités que pour le composant recherché de l'exemple précédent. C'est-à-dire : i) un degré de qualité élevé ; ii) un degré de test élevé ; iii) la gestion du protocole SSL ; iv) un prix raisonnable autour de 500 dollars US. On en déduit, comme pour le composant recherché précédent : i) une NFP *QD* correspondant à l'attribut *QualityDegree* avec la valeur "9" ; une NFP TD correspondant à l'attribut *TestDegree* avec la valeur "8" ; iii) une NFP *Price* correspondant à l'attribut

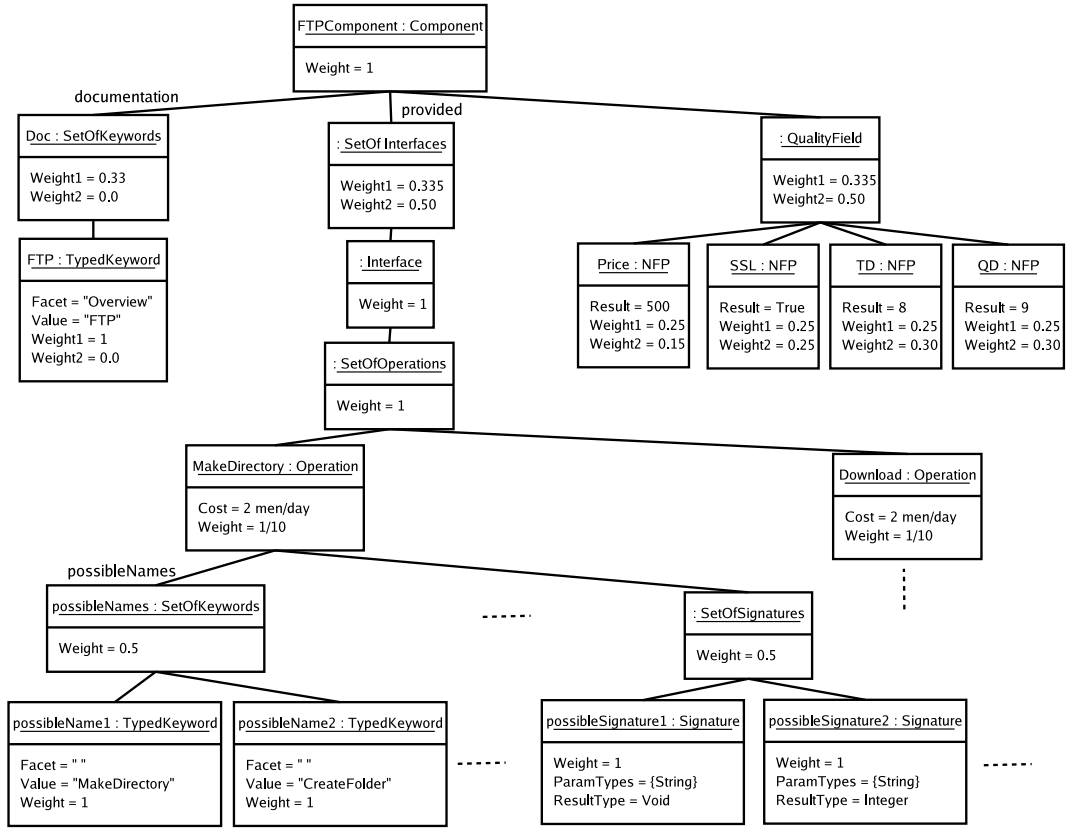

FIG. 7.5 – Composant recherché du deuxième exemple et ses pondérations.

Licence avec la valeur "500" ; iv) une NFP *SSL* correspondant à l'attribut qualité *SecuritySSL* avec la valeur *True*.

7.3.3.1 Indices de satisfaction

La première étape de notre processus de sélection (voir section 6.5) est donc réalisée, ainsi que la deuxième puisque les candidats restent les mêmes. Dans la troisième étape, celle où il faut choisir les modes de comparaison et de pondération, nous allons utiliser l'indice de satisfaction et l'estimation globale d'effort (voir section 6.3). En effet, nous ne recherchons plus une marque particulière ou une stricte conformité avec les signatures d'opération proposées, mais plutôt un certain nombre de services précis. On doit donc s'attendre à adapter le candidat qu'on sélectionnera. Passons maintenant à la quatrième étape, celle où il faut choisir sa stratégie de sélection et pondérer le composant recherché. Nous avons choisi la stratégie décrite dans la sous-section 6.4.1, car elle utilise successivement la satisfaction et l'effort.

En ce qui concerne la satisfaction pondérée par distribution, la figure 7.5 en montre deux utilisations, dont on peut se servir dans la deuxième phase de la stratégie choisie. Ces deux distributions portent sur la spécification complète du composant recherché. La première, men-

Nom	Doc	Interface	NFP *SSL*	NFP *Price*	NFP *QD*	NFP *TD*	**Total(1)**	**Total(2)**
IPW-SA	1.0	0.8	1.0	0.67	0.89	1.0	**0.895**	**0.857**
PTCP-US	1.0	0.75	1.0	0.23	0.63	1.0	**0.821**	**0.761**
IPW-A	1.0	0.8	0.0	1.0	0.89	1.0	**0.839**	**0.756**
PTCP-SA	1.0	0.75	1.0	0.51	0.13	1.0	**0.803**	**0.707**
IPW-SD	1.0	0.8	1.0	0.67	0.0	0.86	**0.811**	**0.703**
IPW-SCB	1.0	0.8	1.0	0.67	0.0	0.86	**0.811**	**0.703**

TAB. 7.2 – Indice de satisfaction et pondération par distribution : les six “meilleurs” résultats.

tionnée par *Weight1* sur la figure 7.5 accorde pour chaque nœud un poids égal pour tous ses éléments fils. La deuxième distribution, mentionnée par *Weight2*, accorde une importance différente à certains éléments, en favorisant en particulier les degrés de qualité et de test par rapport au prix. Les poids mentionnés par *Weight* s’appliquent aussi bien pour la première que pour la deuxième distribution.

Nous pouvons maintenant effectuer la cinquième étape du processus de sélection, qui est l’application de la stratégie choisie. Comme pour la stratégie de l’exemple précédent, celle-ci commence par une pré-sélection des candidats sur la seule base de leur documentation. Comme nous voulions mesurer l’indice de satisfaction uniquement sur les candidats susceptibles de traiter de FTP, nous avons de nouveau effectué cette phase de pré-sélection avec seulement le mot-clé typé de facette “Overview” (“Aperçu”) et de valeur “FTP” dans la documentation. Et comme pour l’exemple précédent, sur les 131 candidats que contenait la section “communication internet” de *ComponentSource*, seulement 55 composants se sont révélés “pertinents”. Pour effectuer la deuxième phase de la stratégie, celle où on calcule les indices de satisfactions associés à la pondération par distribution, les indices pour chaque candidat ont été mesurés avec l’outil *Substitute*.

Le tableau 7.2 montre le résultat de cette sélection sur les six candidats les plus “satisfaisants”. Les indices qui sont considérés en particulier sont ceux pour la documentation, l’interface fournie et les quatre NFPs. Parmi les candidats les plus “satisfaisants”, nous avons les versions ActiveX/VB sécurisée, Delphi sécurisée, “C++ Builder” sécurisée, et ActiveX/VB non sécurisée du composant FTP de la suite logicielle *IP*Works !*, notés respectivement *IPW-SA*, *IPW-SD*, *IPW-SCB* et *IPW-A* sur le tableau 7.2 (par “version sécurisée”, nous entendons que le composant gère le protocole SSL). Nous avons également les versions “Universal Subscription” et ActiveX sécurisée du composant FTP de *PowerTCP*, notés respectivement *PTCP-US* et *PTCP-SA* sur le tableau.

La colonne *Total(1)* donne le résultat de l’indice de satisfaction avec la première pondération par distribution. On remarque à l’occasion qu’accorder la même importance à tous les éléments n’est pas forcément une bonne manière de pondérer, car certaines NFPs comme le prix sont peut-être moins importantes que d’autres. De plus, la documentation a déjà été évaluée, il n’est donc plus nécessaire de l’inclure dans le calcul. Nous allons donc recommencer cette phase en utilisant la deuxième distribution, afin de privilégier les degrés de qualité et de test par rapport au prix. La colonne *Total(2)* du tableau 7.2 donne le résultat de l’indice de satisfaction avec

Nom	**IPW-SA**			**PTCP-US**		
Mesure	*Index*	*Estim*	$Weight_{Est}$	*Index*	*Estim*	$Weight_{Est}$
RenameFile	1.0	0.0	0.0	0.5	0.5	0.025
RenameDirectory	0.0	2.0	0.1	0.0	2.0	0.1
ListDirectory	1.0	0.0	0.0	0.5	0.5	0.025
Download	0.5	2.0	0.1	1.0	0.0	0.0
Upload	0.5	2.0	0.1	1.0	0.0	0.0
Connection	1.0	0.0	0.0	0.5	0.5	0.025
Effort total	**0.3**			**0.175**		

TAB. 7.3 – Résultats de l'estimation globale d'effort.

cette deuxième distribution. On remarque que même si les cinq candidats les plus "satisfaisants" sont les mêmes, le changement de pondération a modifié le classement des candidats et creusé les écarts entre eux. Par exemple, *PTCP-US* devient même plus "satisfaisant" que *IPW-A* qui le surclassait avec la première distribution. Il en va de même pour *PTCP-SA* par rapport à *IPW-SD* et *IPW-SCB*. En revanche, *IPW-SA*, ayant globalement la meilleure qualité, se retrouve premier loin devant les autres.

7.3.3.2 Estimation globale d'effort

La deuxième phase de la stratégie choisie ayant été réalisée, il reste la phase finale. Dans cette phase, on utilise l'estimation globale d'effort, en ne considérant que les écarts entre opérations des composants candidats et recherché. Or, il se trouve que les six candidats les plus satisfaisants appartiennent tous, soit à la suite *IP*Works !*, soit à la suite *PowerTCP*. Pour chacune de ces deux suites, les composants qui y appartiennent ont tous les mêmes opérations avec les mêmes signatures quelle que soit la version. En d'autres termes, cela revient au même d'estimer l'effort d'adaptation de la version ActiveX/VB sécurisée ou celui de la version Delphi sécurisée, puisque leurs opérations sont identiques. De plus, les deux meilleurs candidats, *IPW-SA* et *PTCP-US*, représentent chacun l'une de ces suites, et sont de meilleure qualité que les quatre autres. Ce sont donc les seuls qui vont être retenus pour la phase finale.

Les indices de satisfaction de *IPW-SA* et de *PTCP-US* sont respectivement de 0.857 et 0.761. D'un point de vue fonctionnel, le premier est légèrement meilleur que le second, avec des indices respectifs de 0.8 et 0.75. Si on regarde les dix opérations dans le détail, les deux candidats possèdent des opérations de déconnexion, de destruction de fichier, de création de dossier et de destruction de dossier qui satisfont pleinement les opérations recherchées, et qui n'ont donc pas besoin d'adaptation. Le tableau 7.3 affiche les informations relatives à chaque candidat pour les 6 autres opérations, ainsi que leur estimation globale d'effort.

On s'aperçoit que *IPW-SA* ne possède que deux opérations qui requièrent une adaptation : le téléchargement de fichier (*Download* sur le tableau) et la mise en ligne de fichiers (*Upload*). *PTCP-US*, quant à lui, possède trois opérations qui nécessitent une adaptation : le renommage de fichier (*RenameFile*), l'affichage du contenu du dossier (*ListDirectory*) et la connexion (*Connection*). Enfin, les deux candidats manquent d'une opération : le renommage

de dossiers (*RenameDirectory*). On pourrait croire que *PTCP-US*, déjà moins "satisfaisant" que *IPW-SA*, nécessite plus d'effort d'adaptation. Pourtant, à la lecture des colonnes $Estim$ du tableau 7.3, on s'aperçoit que selon l'estimation de l'utilisateur, les opérations de *PTCP-US* qui présentent un écart avec les opérations recherchées sont facilement adaptables : 0.5 jour/homme par opération, ce qui leur confère un poids minime (0.5/20 = 0.025). En revanche, l'utilisateur a estimé que le travail d'adaptation sur les opérations manquantes de *IPW-SA* serait beaucoup plus conséquent. À tel point que l'estimation d'effort pour ces opérations équivaut à leur coût total de développement, c'est-à-dire 2 jours/hommes. Quand on additionne les poids pour obtenir les estimations globales d'effort, on se rend finalement compte que *PTCP-US*, avec une estimation de 0.175, requiert moins d'effort d'adaptation que *IPW-SA*, qui écope d'une estimation de 0.3. C'est donc *PTCP-US* le "meilleur" candidat, et c'est celui qui sera sélectionné dans la dernière étape de notre processus de sélection.

7.3.3.3 Bilan

Quand l'indice de satisfaction a été mesuré avec la pondération par distribution, *IPW-SA* était le candidat le plus "satisfaisant" avec un indice de 0.857, loin devant le deuxième, *PTCP-US* (0.761). Mais comme le montre le tableau 7.3, l'utilisateur a considéré que *PTCP-US* était en fait le candidat qui demanderait le moins d'effort d'adaptation. Cet exemple montre donc que le "meilleur" candidat n'est pas nécessairement le plus "satisfaisant". Autrement dit, l'indice de satisfaction permet de rassembler les candidats les plus proches du composant recherché, mais c'est l'effort qui fait vraiment la différence quand il s'agit de choisir le meilleur d'entre eux.

7.4 En résumé

Dans ce chapitre, un outil a été présenté pour implémenter les mécanismes et le processus de sélection que nous avons proposés dans le chapitre précédent. L'outil *Substitute* implémente en particulier : i) le format de description présenté dans la section 5.3 ; ii) le mécanisme de comparaison présenté dans la section 5.5.1 ; iii) l'indice de satisfaction associé à la pondération par distribution, présenté dans la sous-section 6.2. La visualisation se fait au moyen de l'outil de G. Langelier *et al* [163].

Les deux exemples présentés, mettant en scène la sélection d'un composant FTP sur *ComponentSource* [63], ont mis en évidence le gain de temps important occasionné par l'automatisation de la comparaison. Ils ont également mis en évidence l'avantage du concept de composant recherché et de sa bonne utilisation. La comparaison multi-niveaux permet d'appliquer des filtres successifs sur la bibliothèque de candidats, afin d'éliminer un nombre important de composants qui ne sont pas pertinents. Et le traitement des disparités par la pondération permet de privilégier certains éléments plutôt que d'autres. Ce qui rend un candidat plus "satisfaisant" qu'un autre, alors que ce pourrait être l'inverse avec une distribution différente des poids.

Enfin, en ce qui concerne la combinaison de la satisfaction et de l'effort, on a pu observer que le "meilleur" candidat n'était pas forcément le plus "satisfaisant". Les deux modes de comparaison sont en fait complémentaires : le premier sert à éliminer la plupart des candidats qui ne conviennent pas, mais c'est le second qui fait vraiment la différence.

Dans cette partie, le cœur de cette thèse a été détaillé. Le concept bicéphale de composant recherché a été introduit. D'une part, ce concept permet de modéliser le besoin sous forme de composant. D'autre part, il permet de traiter les manques ou les insuffisances des candidats. Pour aborder le premier aspect de cette structure bicéphale, nous avons introduit un format de description des propriétés fonctionnelles et non-fonctionnelles des composants. Pour aborder le deuxième aspect de cette structure, nous avons introduit la notion de pondération des propriétés du format de description. Ce format et cette notion de pondération ont été intégrés dans un mécanisme de comparaison hiérarchique et multi-niveaux. Ce mécanisme automatise la comparaison au moyen de techniques diverses, qui peuvent être appliquées à différents niveaux de description.

Afin de donner un sens à la comparaison, nous avons présenté deux méthodes. La première consiste en un indice de satisfaction, qui adapte le mécanisme de comparaison pour évaluer la similarité entre un composant candidat et un composant recherché. On a associé à cet indice une pondération qui tient compte de l'importance des différents éléments les uns par rapport aux autres. Cette pondération tient également compte du coût d'adaptation des opérations. La seconde méthode consiste en une estimation globale d'effort, qui demande à l'utilisateur d'évaluer le coût d'adaptation des quelques candidats les plus "satisfaisants". Afin d'exploiter au mieux ces méthodes de comparaison, nous avons donné des exemples de stratégie de sélection qui les utilisent successivement. Et afin de choisir la stratégie la plus adaptée à un contexte donné, nous avons défini un processus systématique de sélection de composants. Ce processus part de la définition d'un composant recherché et du choix d'une bibliothèque de candidats, pour aboutir à la sélection du meilleur d'entre eux selon une certaine stratégie. Une fois que celui-ci est intégré dans l'application, l'architecture de celle-ci est modifiée, ce qui fait apparaître un nouveau besoin. On va donc définir un nouveau composant recherché et recommencer le processus.

Le format de description, le mécanisme de comparaison, l'indice de satisfaction et sa pondération associée ont été implémentés à travers l'outil *Substitute*. Cet outil est simple d'utilisation et facilement extensible. L'ensemble de l'approche a été validé sur deux exemples concrets. Chacun d'eux est basé sur le marché aux composants *ComponentSource* et exécute le processus de sélection avec l'une des stratégies de sélection proposées. Ces exemples mettent en évidence le gain de temps obtenu grâce à l'automatisation de la comparaison. Le deuxième exemple, en particulier, met en évidence le fait que le "meilleur" candidat n'est pas forcément le plus "satisfaisant", et que c'est l'effort d'adaptation des candidats qui fait la différence.

Quatrième partie

Épilogue

Chapitre 8

Conclusion et perspectives

Sommaire

La contribution principale de cette thèse est d'automatiser autant que possible les différentes phases de la sélection de composants logiciels. Cette automatisation se fait au moyen de techniques issues de domaines variés, notamment la recherche de composants en bibliothèque, le sous-typage entre composants, les contrats de qualité de service et les modèles de qualité pour les COTS. La section 8.1 récapitule les différents apports de cette thèse. La section 8.2 détaille quant à elle les prolongements possibles de ce travail. En particulier : une extension du concept de composant recherché, ainsi que l'ajout de nouveaux éléments au format de description, de nouveaux modes de comparaison et de nouvelles pondérations.

8.1 Apports de la thèse

Plusieurs outils conceptuels ont été proposés, ainsi que des méthodes pour les implanter. On retrouve notamment le concept de composant recherché, qui intègre un mécanisme de pondération. On associe à ce concept un mécanisme de comparaison fonctionnelle et non-fonctionnelle. Ces concepts ont été abordés dans [104, 105]. On retrouve également la notion de comparaison multi-niveaux, ainsi que différentes sémantiques de comparaison (satisfaction et effort), des stratégies qui les utilisent, et un processus de sélection incrémental et itératif.

8.1.1 Le concept de composant recherché

En modélisant le besoin d'une application par un composant, on peut définir des critères d'évaluation calqués sur les propriétés fonctionnelles (interfaces, opérations...) et non-fonctionnelles des composants existants. On peut évaluer les candidats directement selon ces propriétés. Cependant, l'originalité de ce concept réside dans la manière de décrire et de traiter les disparités entre les descriptions des candidats et celle du composant recherché. En utilisant un mécanisme de pondération embarqué dans le composant recherché, on peut évaluer l'importance réelle des éléments du composant recherché les uns par rapport aux autres. Et pour chaque élément du composant recherché, on peut sanctionner à sa juste valeur une éventuelle discordance de la part des candidats. Selon l'importance qu'on donne à cet élément, son absence ou son respect partiel par un candidat aura plus ou moins d'influence sur l'évaluation globale de celui-ci. Le concept de composant recherché présente donc une structure bicéphale. Le premier aspect de cette structure est la description de ce que l'on cherche. Son deuxième aspect est le traitement, par la pondération, des manquements et des insuffisances des candidats par rapport à cette description.

8.1.2 Un format de description pour les composants

Le format choisi pour décrire les composants inclut les artefacts architecturaux communs à la plupart des modèles de composant existants : les composants, les interfaces et les opérations. Notre format a également pour originalité d'englober les propriétés fonctionnelles et non-fonctionnelles dans une même description. On peut donc mélanger des techniques de comparaison qui traitent exclusivement de la partie fonctionnelle ou non-fonctionnelle, mais pas des deux. Sur le plan fonctionnel, les principaux éléments considérés sont la documentation et les signatures d'opération. La documentation sert à effectuer une recherche par mots-clés pour effectuer des recherches à haut niveau sur un grand nombre de candidats. Et le sous-typage de signature permet de vérifier la conformité fonctionnelle des candidats qui restent. Sur le plan non-fonctionnel, les éléments considérés sont les NFPs (*Non-Functional Properties*), qui mesurent le niveau d'attributs qualité au moyen de métriques. Cette représentation permet d'évaluer la qualité des composants dans son ensemble, et d'intégrer plusieurs propriétés différentes dans une même comparaison. Ce format de description est assez abstrait pour englober toutes les possibilités des modèles et formats de requêtes existant pour les composants en bibliothèque.

8.1.3 Un mécanisme de comparaison à plusieurs niveaux

Le format précédent fournit un langage permettant de décrire les composants candidats et le composant recherché. On peut ensuite comparer les descriptions obtenues dans ce format unique grâce à un mécanisme multi-niveaux. Ce mécanisme tire parti de la nature hiérarchique du format de description, en l'intégrant à une structure en arbre qui distingue les éléments "nœuds" des éléments "feuilles". En ce qui concerne les nœuds (artefacts et ensemble d'éléments divers), on va les comparer récursivement et agréger ces comparaisons jusqu'à la racine, c'est-à-dire le composant. En ce qui concerne les feuilles (NFPs, mots-clés et signatures), on va utiliser des techniques *ad hoc* afin de les comparer automatiquement. Avec une telle structure, on peut adapter la comparaison au niveau de granularité souhaité, et garantir son automatisation. Afin d'exploiter pleinement le concept de composant recherché, le mécanisme de comparaison inclut également une fonction de pondération qui, pour chaque comparaison entre nœuds, fait intervenir le poids du nœud recherché. Ce sont donc les poids qui orientent la comparaison globale entre composants, en évaluant l'importance de chaque comparaison locale entre nœuds par rapport aux autres. Un tel mécanisme de comparaison exploite parfaitement la structure bicéphale du composant recherché. Et l'aspect multi-niveaux de ce mécanisme permet de le réutiliser plusieurs fois selon le nombre de candidats et le niveau de granularité approprié. Ce qui offre un cadre favorable à une stratégie progressive de sélection.

8.1.4 Des sémantiques et des pondérations qui donnent le sens de la comparaison

Le mécanisme de comparaison précédent offre un cadre général à la comparaison entre les composants candidats et le composant recherché. Cependant, c'est le sens qu'on donne à cette comparaison qui permet de déterminer si un candidat est le "meilleur". Dans ce mémoire de thèse, deux sémantiques complémentaires ont été présentées pour la comparaison. Ces sémantiques nécessitent chacune leur propre mode de comparaison et leur propre fonction de pondération.

8.1.4.1 Indice de satisfaction et pondération par distribution

Les techniques d'évaluation utilisées dans les processus de sélection de composants visent à déterminer jusqu'à quel point un candidat remplit les critères d'évaluation initiaux. L'indice de satisfaction est une extension du mécanisme de comparaison multi-niveaux basée sur le même principe. Cet indice mesure les propriétés que le candidat partage en commun avec le composant recherché. D'un point de vue fonctionnel, il s'agit de déterminer combien les composants candidat et recherché partagent de mots-clés et d'opérations avec la signature désirée. D'un point de vue non-fonctionnel, il s'agit de déterminer combien de propriétés non-fonctionnelles recherchées sont contenues dans le composant candidat, et avec quel niveau. L'avantage principal de cet indice de satisfaction est d'automatiser l'évaluation locale des candidats pour chaque critère.

À cet indice de satisfaction, on a associé une pondération par distribution. Cette pondération consiste à partager 100% du poids de chaque nœud entre ses fils directs, depuis le composant jusqu'aux feuilles. Le poids de chaque élément est donc borné et établi en fonction du poids des autres éléments ayant le même nœud parent. Une exception notable concerne les opérations,

dont le poids dépend de leur coût de développement par rapport à celui des autres opérations de l'interface. La signification qu'on donne ainsi au poids de chaque opération est la suivante : qu'est-ce qui coûte le moins cher, entre développer cette opération soi-même, ou la remplacer par telle opération candidate ? Et que coûterait le développement de chaque opération par rapport aux autres opérations de l'interface ? L'avantage d'une telle pondération est d'offrir des poids cohérents et qui ont un sens.

De plus, un traitement à part a été réservé pour les services requis. Cela permet de tenir compte du cas où un candidat requiert plus que ce qu'on a prévu pour le composant recherché. On calcule donc une pénalité pour chaque élément candidat requis qui n'a pas d'équivalent dans le composant recherché. Ensuite, on retranche cette pénalité de l'indice de satisfaction global.

8.1.4.2 Estimation globale d'effort

Une fois qu'on a sélectionné un composant candidat parmi ceux disponibles, il faut l'adapter afin qu'il puisse être intégré sans encombre dans l'application. L'indice de satisfaction permet de distinguer les candidats les plus proches du composant recherché, mais il faut en plus estimer l'effort requis pour les adapter afin qu'ils correspondent complètement à ce composant recherché. Le "meilleur" candidat, en définitive, n'est pas forcément le plus "satisfaisant", mais celui qui requiert le moins d'effort d'adaptation pour que cette satisfaction soit maximale. L'effort est donc l'ultime critère de sélection. Partant de ce principe, une estimation globale d'effort a été définie. Ce calcul se base sur une pondération par estimation. Cette pondération consiste à ne considérer que les opérations candidates et recherchées mises sur le même niveau. Pour chaque opération recherchée qui n'est pas complètement satisfaite par une opération candidate, on demande à l'utilisateur d'estimer l'effort d'adaptation de celle-ci. Le poids de l'opération recherchée dépend donc de cette estimation par rapport au coût de développement de toutes les opérations du composant. Et l'estimation globale d'effort est égale à la somme de ces poids. Cette estimation d'effort permet de déterminer parmi les candidats les plus satisfaisants lequel est réellement le "meilleur".

8.1.5 Des stratégies de sélection qui combinent différentes comparaisons et pondérations

Afin d'utiliser au mieux le format de description, le mécanisme de comparaison, les sémantiques et les pondérations associées, nous avons mis en avant la notion de stratégie de sélection. Une comparaison unique sur toute la bibliothèque, avec un seul niveau de granularité et selon une seule sémantique, risquerait d'être trop imprécise ou trop fastidieuse. Certaines comparaisons se prêtent mieux que d'autres à un niveau de granularité particulier ou un certain nombre de candidats. Et certaines sémantiques de comparaison sont complémentaires les unes des autres. C'est le cas de l'estimation globale d'effort, qui ne peut s'appliquer efficacement que sur un petit nombre de candidats, préalablement filtrés par l'indice de satisfaction. Et il vaut mieux que l'indice de satisfaction ne s'applique que sur les candidats considérés comme les plus pertinents, afin d'éviter des calculs inutiles. Une première phase de pré-sélection sur la documentation est un bon moyen de filtrer ces candidats pertinents. Nous avons donc une première stratégie de sélection qui utilise successivement une pré-sélection sur la documenta-

tion, une mesure d'indice de satisfaction sur les candidats "pertinents", et une estimation de l'effort d'adaptation des candidats les plus "satisfaisants". Cette première stratégie permet une sélection progressive, prenant en compte plusieurs significations du "meilleur" candidat. Partant d'une vaste bibliothèque, on choisit étape après étape un candidat non seulement proche du composant recherché, mais aussi facilement adaptable pour que la correspondance soit totale. Un autre exemple de stratégie consiste à comparer les mesures d'indice de satisfaction pondérés d'une certaine manière avec la visualisation des indices locaux. Cela permet une pondération plus intuitive, qui permet de déterminer d'un coup d'œil quels sont les candidats les plus "satisfaisants".

8.1.6 Un processus de sélection qui permet de choisir une stratégie selon le besoin

La stratégie de sélection et le choix des modes de comparaison appropriés dépendent fortement du besoin de l'utilisateur. C'est pourquoi un processus de sélection de composants a été défini sur la base du concept de composant recherché. Ce processus, à partir des outils conceptuels définis précédemment, permet de choisir les moyens d'utiliser ces outils. Il faut tout d'abord définir le composant recherché et charger la bibliothèque de candidats qui vont lui être comparés. Ensuite, il faut choisir les modes de comparaison utilisés et les pondérations associés, ainsi que la stratégie qui va exploiter tous ces éléments. On peut ensuite appliquer cette stratégie pour sélectionner le "meilleur" candidat. Ce processus de sélection est calqué sur le caractère incrémental de la notion de composant recherché. En effet, chaque fois qu'un besoin a été identifié pour une application, il faut définir le composant recherché correspondant. Ce composant recherché est remplacé par le meilleur candidat "concret" sélectionné au terme de ce processus. Ce candidat est ensuite intégré à l'application, ce qui modifie l'architecture de celle-ci et fait émerger un nouveau besoin. À ce besoin correspond un nouveau composant recherché, et on peut recommencer le processus.

8.1.7 Validation de l'approche

Le principe de la comparaison fonctionnelle et non-fonctionnelle d'un composant candidat et d'un composant recherché, sur la base d'un format commun, a été abordé dans [104, 105]. Le format de description, le mécanisme de comparaison et l'indice de satisfaction ont été implémentés par l'outil *Substitute* [70]. Le processus de sélection, ainsi que les deux stratégies de sélection qui englobent toute l'approche proposée, ont été testés sur deux exemples concrets mettant en scène la sélection d'un composant FTP sur le marché aux composants *ComponentSource* [63]. Le premier exemple portait plus précisément sur le choix de la meilleure version d'un composant d'une suite logicielle particulière. On a appliqué à cet exemple la stratégie combinant calcul de satisfaction et visualisation. Grâce à cette approche intuitive, la meilleure version a pu être identifiée en un coup d'œil. Le second exemple portait sur la sélection d'un composant de marque et de modèle quelconques, qui fournirait un certain nombre de services avec une qualité élevée. On a appliqué à cet exemple la stratégie utilisant successivement la satisfaction et l'effort. L'utilisation de deux distributions des poids pour la satisfaction a permis de mettre en évidence l'influence de la pondération sur le calcul. Par exemple, avec la première

distribution, un candidat en surclassait un autre, leurs indices de satisfaction étant respectivement de 0.839 et 0.821. Avec la deuxième distribution, en revanche, c'est l'inverse qui s'est produit, le second candidat ayant un indice de satisfaction supérieur au premier (respectivement, 0.761 et 0.756). De plus, grâce à l'utilisation successive de la satisfaction et de l'effort, nous nous sommes rendus compte que le candidat le plus "satisfaisant" n'était pas automatiquement le "meilleur". La raison est qu'il est plus avantageux d'avoir un candidat un peu moins satisfaisant, mais plus adaptable. Selon les résultats de l'expérimentation, le candidat le plus satisfaisant arrivait loin en tête avec un indice de 0.857, mais son estimation d'effort était de 0.3. Le deuxième candidat le plus satisfaisant était loin derrière avec un indice de 0.761, mais c'est pourtant lui qui était le plus facilement adaptable, avec une estimation d'effort de 0.175. C'est donc lui qui a été choisi.

Ces deux expérimentations ont mis en évidence les principaux avantages de notre approche. Le premier réside dans l'automatisation de la comparaison. En effet, les calculs d'indice de satisfaction ont tous été calculés en quelques secondes, là où il aurait fallu plusieurs jours pour une évaluation locale manuelle de tous les candidats pour tous les critères. Le deuxième avantage réside dans l'aspect multi-niveaux de la comparaison, ainsi que dans le concept de composant recherché et sa bonne utilisation. La comparaison multi-niveaux peut s'adapter au nombre de candidats disponibles. Elle a permis de réduire facilement et progressivement le nombre de candidats, depuis une vaste bibliothèque de 131 candidats jusqu'au "meilleur" d'entre eux, au fur et à mesure que les besoins se précisaient.

8.2 Perspectives pour de futurs travaux

Dans cette section, nous allons présenter les prolongements possibles pour ce travail de thèse. Les extensions proposées concernent soit les outils conceptuels, soit l'ajout de nouveaux moyens d'exploiter ces outils.

8.2.1 Du composant recherché à l'architecture recherchée

Nous avons vu que la notion de composant recherché est incrémentale. C'est-à-dire que la spécification d'un tel composant se fait en fonction de l'architecture "concrète" de l'application. Lors de la sélection, on ne s'occupe donc que de la comparaison entre les candidats concrets et le composant recherché. Un tel postulat de départ interdit la sélection de composants multiples. Une autre approche consisterait à étendre la notion de composant recherché afin de définir une "architecture recherchée". Ce concept permettrait la sélection simultanée de plusieurs composants. Il permettrait également d'anticiper certains problèmes de compatibilité entre candidats concrets. L'exploitation d'un tel concept impliquerait une extension du format de description, afin de lui intégrer des éléments architecturaux comme les connecteurs. Cela impliquerait aussi quelques modifications dans le processus de sélection.

8.2.2 Extensions du format de description

D'un point de vue fonctionnel, le format de description ne considère que les aspects syntaxiques des composants, c'est-à-dire les noms et les signatures d'opérations. On peut rajouter à cela la documentation sur laquelle on peut effectuer une recherche par mots-clés. Les aspects sémantiques des composants ne sont pas pris en compte. Il est vrai que dans des marchés aux composants tels que *ComponentSource*, les composants ne disposent que très rarement de telles spécifications. Et c'est à peine si leur comportement est abordé dans la documentation. Cela dit, ne pas les prendre en compte empêche d'utiliser certaines techniques de recherche qui pourraient être utiles lors de la sélection des candidats. Notamment le matching de spécification [302] ou la mesure de distance fonctionnelle [139].

Une extension possible de notre format consisterait donc à décrire et à comparer le comportement des composants. Cela implique, d'une part, l'ajout de nouveaux éléments dans ce format, et d'autre part l'implémentation de techniques appropriées pour la comparaison de ces nouveaux éléments. Un exemple d'extension serait le suivant. À chaque opération on associerait, en plus des éléments déjà définis, un nœud d'un nouveau type : "Ensemble des sécifications possibles". Ce nœud serait consituées de feuilles d'un nouveau type : "Spécification comportementale", constituée d'un ensemble de pré- et post-conditions. Et le calcul d'écart associé à ce nouveau type de feuilles serait issu du sous-typage comportemental [172] ou du matching de spécification [302].

Parmi les autres extensions possibles, on pourrait prendre en compte les composants composites et considérer que chaque composant, en plus des éléments déjà définis, est également constitué d'un nœud d'un nouveau type : "Ensemble de sous-composants". Cela permettrait de décrire pleinement les composants dont les modèles utilisent cette notion, comme par exemple Fractal [45]. Cela permettrait aussi d'utiliser les techniques tirant parti de tels composants composites, tels que la substituabilité des profils de qualité de service proposée dans le langage CQML [1].

Enfin, à défaut d'ajouter de nouveaux types d'éléments au format de description, on pourrait trouver de nouvelles manières de comparer ceux déjà disponibles. Par exemple, en ce qui concerne les signatures d'opération, une piste possible serait d'utiliser le matching de signature de Zaremski et Wing [301] plutôt que le sous-typage de signature. Les règles de ce matching permettent en effet une certaine flexibilité, mais leur automatisation pré-suppose une connaissance étendue des différents systèmes de type et des relations possibles entre eux. Enfin, en ce qui concerne la documentation, si l'on considère que la classification par facettes n'est pas assez précise, une amélioration possible serait d'utiliser d'autres approches, telles que le langage naturel [176].

8.2.3 Nouvelles manières d'estimer l'effort

L'estimation d'effort que nous avons mise au point ne concerne que les opérations du composant candidat, toutes mises sur le même plan. Elle ne tient compte d'aucun autre facteur de coût. Une extension possible serait de considérer d'autres manières d'estimer l'effort. En particulier, on pourrait reprendre les modèles algorithmiques d'estimation d'effort qui existent déjà. Les modèles dits algorithmiques ou paramétriques prennent la forme d'une fonction qui relie

la taille ou le montant de code à écrire (noté *Size*) à l'effort ou coût requis pour l'écrire, noté $Effort$. La fonction, utilisée par de nombreux modèles, est de la forme suivante :

$$Effort = A(Size)^B \tag{8.1}$$

A est un coefficient qui représente l'ensemble des facteurs de coût "linéaires", ou multiplicateurs d'effort liés à l'environnement de développement. *B* représente l'ensemble des facteurs de coût exponentiels qui peuvent croître au fur et à mesure que le logiciel s'agrandit. *Size* peut se mesurer, par exemple, en lignes de code (SLOC, *Source Lines of Code* [140, 87]), en milliers de lignes de code (KSLOC) ou en points de fonction (FP) [5, 4]. $Effort$ peut se mesurer, par exemple, en jours-hommes (*man-day*) ou en mois-hommes (*man-month*), et en général, *A* contient une constante de conversion, par exemple 1 mois-homme par KSLOC.

L'un des modèles paramétriques les plus connus est COCOMO, dû à B. Boehm [30]. COCOMO offre différents coefficients selon le type de projet, et a connu de nombreuses applications, telles qu'Ada COCOMO [29]. Le modèle a évolué en COCOMO 2.0 [32, 27], afin de pouvoir mdéliser de nouveaux processus et paradigmes tels que les composants, le middleware et le reengineering. COCOTS (COnstructive COTS) [2] est une extension de COCOMO 2.0 destinée à modéliser les éléments du logiciel à base de composants COTS dont le code n'est pas accessible. COCOTS utilise plusieurs fonctions d'estimation d'effort qui raffinent la formule de base des modèles paramétriques, selon le contexte et le type d'effort qu'on veut calculer. Ces fonctions qui peuvent connaître de nombreuses applications telles que l'analyse des risques encourus [229, 295], ou la prédiction de l'effort dans les services Web [228, 28]. Elles apportent leurs propres facteurs de coût, qui peuvent être liés, soit au COTS à intégrer, soit à l'équipe en charge de son intégration, soit à l'application dans laquelle on va l'intégrer. Des exemples de facteurs de coût sont le support technique du fournisseur, la fiabilité et la portabilité de l'application, ou l'expérience du personnel avec l'intégration de COTS. Il serait particulièrement intéressant d'évaluer automatiquement le plus possible de ces facteurs de coût, en particulier ceux qui sont liés au COTS à intégrer. Cette évaluation automatique pourrait s'effectuer à partir d'informations extraites de la documentation ou de la structure externe du COTS.

8.2.4 Nouvelles pondérations

L'inconvénient principal de la pondération par distribution associée à l'indice de satisfaction est qu'elle ne peut pas modéliser les éléments "indispensables". Considérons l'exemple illustré par la figure 8.1. Un composant possède deux propriétés non-fonctionnelles dans son champ qualité : $NFP1$, qui indique si un test a été effectué ou non, et $NFP2$, qui indique le niveau d'une certaine propriété. $NFP1$ est importante mais pas indispensable, car le concepteur peut, dans le pire des cas, effectuer le test lui-même. En revanche, $NFP2$ est indispensable car le concepteur en a besoin et qu'il ne peut ni la rajouter ni améliorer son niveau lui-même. Par conséquent, si un candidat ne possède pas cette propriété au niveau demandé, il sera éliminé d'office quel que soit le résultat pour ses autres éléments.

Le problème qui se pose avec la pondération par distribution est le suivant : comment donner à $NFP2$ un caractère "indispensable" sans retirer de l'importance à $NFP1$? D'un côté, un poids trop élevé pour $NFP1$ par rapport à $NFP2$ (par exemple, 0.45 pour la première et 0.55

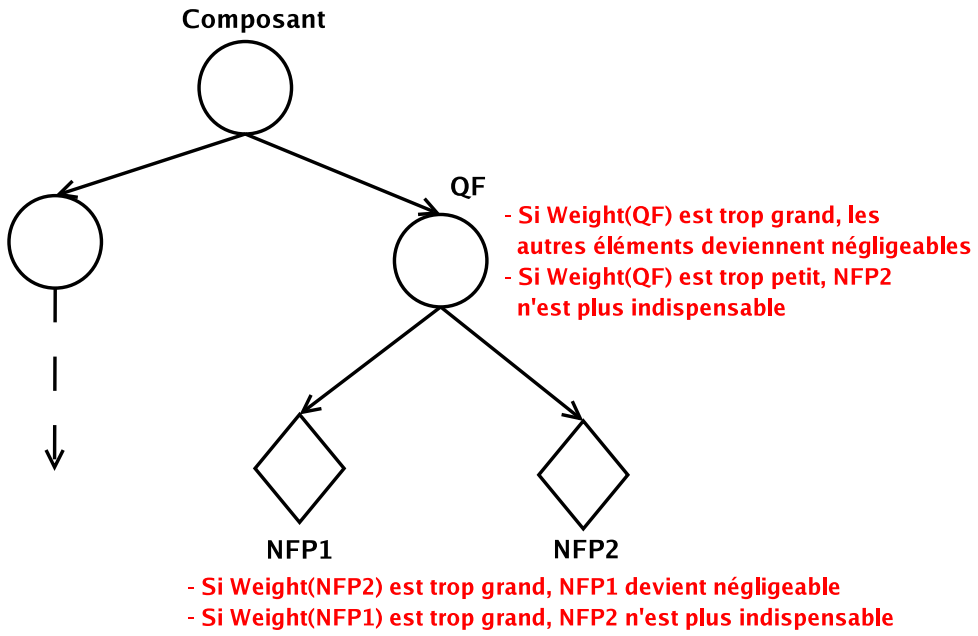

FIG. 8.1 – Limite de la pondération par distribution

pour la seconde) signifierait qu'en cas d'absence de $NFP2$, le candidat serait pénalisé, mais pas forcément rejeté. De l'autre côté, un écart trop grand entre $NFP2$ et $NFP1$ (par exemple, 0.80 pour la première et 0.20 pour la deuxième) signifierait qu'en cas d'absence de $NFP1$, le candidat ne serait pas vraiment pénalisé. Le même problème se pose au niveau supérieur, comme indiqué sur la figure 8.1 : comment donner à $NFP2$, et donc au champ qualité du composant, un caractère "indispensable" sans diminuer drastiquement l'importance des autres éléments du composant ?

Une solution serait, en plus d'attribuer un poids à chaque élément du composant recherché, de "marquer" ceux qui sont indispensables. Pour chaque élément recherché marqué, on imposerait un indice de satisfaction minimal qu'un élément candidat devrait atteindre. Si l'indice entre les deux éléments est inférieur à l'indice minimal, cela entraîne l'élimination, non seulement de l'élément, mais aussi du composant candidat qui le contient.

Une autre solution consiste à utiliser les réseaux bayésiens. Les réseaux bayésiens [65, 136] constituent un outil de modélisation qui combine la théorie des graphes et la théorie des probabilités. Cet outil a connu de nombreuses applications. En se limitant au génie logiciel, on peut citer comme exemples d'application la prédiction de l'effort [14, 13, 268, 269], des défauts de conception [86, 208] et de la maintenabilité [155]. Un réseau bayésien est un graphe orienté et sans cycle dans lequel les sommets représentent des variables aléatoires discrètes, et les arcs représentent des liens de cause à effet entre ces variables. Dans un réseau bayésien, la relation entre deux évènements A et B associés à des variables est une probabilité conditionnelle $P(A \mid B)$. Il s'agit de la probabilité que A ait lieu sachant que B a eu lieu. À chaque sommet on associe une table de probabilités qui affiche les valeurs des probabilités conditionnelles des

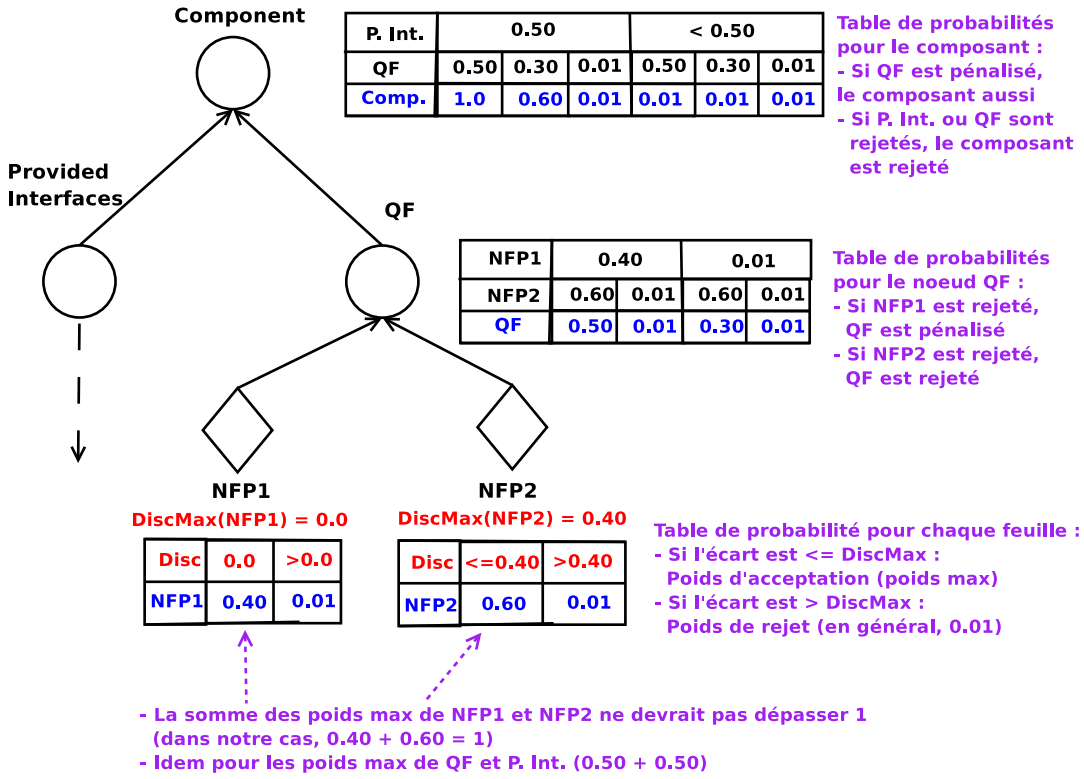

FIG. 8.2 – Pondération bayésienne et application à l'exemple précédent

évènements associés à ce sommet.

Une pondération bayésienne consiste donc à considérer l'arbre de comparaison modélisant le composant recherché comme un réseau bayésien. La figure 8.2 montre le réseau correspondant à l'exemple de la figure précédente. La différence avec l'arbre de comparaison est qu'on inverse l'orientation des arcs, car l'importance des nœuds parents est déterminée par l'importance de ses éléments fils. Les poids, quant à eux, sont assimilés à des probabilités conditionnelles. Le poids de chaque élément recherché E_0 indique la probabilité qu'il soit "accepté". Par "accepté", on sous-entend que la comparaison entre E_0 et l'élément candidat courant E_1 est prise en compte dans l'indice global entre le composant candidat et le composant recherché. Si E_0 a toutes les chances d'être "accepté", alors il se verra attribuer un poids maximum. On ajoute comme contrainte qu'à l'intérieur d'un même nœud, la somme des poids maximums de tous ses éléments fils doit être égale à 1. L'indice de satisfaction de chaque nœud ne dépassera donc pas 1.

À chaque feuille de l'arbre recherché on associe un écart maximum noté $Disc_{Max}$. Quand une feuille candidate est comparée à une feuille recherchée, on compare le calcul d'écart entre elles et l'écart maximum autorisé. Si le premier écart est inférieur ou égal au second, alors la feuille recherchée est "acceptée" dans le calcul d'indice global. On peut donc lui attribuer un poids maximum ou "poids d'acceptation". Si par contre le calcul d'écart entre feuilles est

supérieur à l'écart maximum autorisé, la feuille recherchée est "rejetée" et on lui attribue un poids de 0.01. La figure 8.2, qu'il vaut mieux lire de bas en haut pour la comprendre, montre un exemple d'application. La NFP recherchée $NFP1$, qui est booléenne, ne peut être acceptée qu'avec un écart maximum de 0. Le poids d'acceptation pour cette NFP est égal à 0.40. Pour l'autre NFP recherchée $NFP2$, l'écart maximum obtenu avec une NFP candidate est fixé à 0.40 et son poids d'acceptation est de 0.60.

À chaque nœud, on associe une table de probabilités qui détermine son poids en fonction des poids obtenus par ses fils directs. La figure 8.2 montre comment la pondération bayésienne peut modéliser le caractère indispensable d'une propriété tout en préservant l'importance des autres propriétés. En effet, la probabilité que le champ qualité soit "accepté" sachant que $NFP2$ l'est, mais que $NFP1$ ne l'est pas, n'est que de 0.30. Cela veut dire que le candidat qui ne possède pas $NFP1$ est pénalisé. En revanche, la probabilité que le champ qualité soit accepté sachant que $NFP1$ l'est, mais que $NFP2$ ne l'est pas est de 0.01. Le champ qualité est donc "rejeté d'office". On remarque que la somme des poids maximums attribués au champ qualité et à l'ensemble des interfaces fournies (*P. Int.* sur la figure) est bien égale à 1. Au niveau composant, on suppose qu'il est indispensable que l'ensemble des interfaces fournies soit complètement satisfait, de même qu'il est indispensable que le champ qualité ne soit pas rejeté. Le non-respect de l'une de ces deux conditions provoque automatiquement le rejet du composant. De même, une simple pénalisation du champ qualité (poids de 0.30 pour celui-ci) entraîne une pénalisation du composant (poids de 0.60 pour ce dernier). Par contre, si les deux éléments sont pleinement acceptés, alors on peut utiliser le poids maximum du composant, c'est-à-dire 1.0. On peut limiter les états affichés sur les tables de probabilité afin de simplifier celles-ci. Par exemple, sur la figure 8.2, c'est ce qui a été fait au niveau composant. En effet, le poids de l'ensemble des interfaces fournies a été limité à deux valeurs possibles : acceptation totale (poids maximum de 0.50) ou non (poids inférieur à 0.50, ce qui équivaut ici à un rejet).

Soient E_0 un élément recherché, E_1 un élément candidat de même type (en cas de types différents, le poids est automatiquement de 0.01), et $P(E_0 \mid \bigwedge\{Weight_B(e_0) \mid e_0 \in E_0\})$ la probabilité que E_0 soit "accepté" connaissant les poids attribués à ses sous-éléments e_0. La fonction de pondération bayésienne pour E_0, notée $Weight_B$, est formellement définie comme suit :

$$Weight_B(E_0) = \begin{cases} \text{. } 0.99 \text{ si } E_0 \text{ est une feuille et si } Disc(E_1, E_0) \leq Disc_{Max}(E_0). \\ \text{. } P(E_0 \mid \bigwedge\{Weight_B(e_0) \mid e_0 \in E_0\}) \text{ si } E_0 \text{ est un nœud.} \\ \text{. } 0.01 \text{ sinon.} \end{cases} \tag{8.2}$$

Sur la figure 8.2, par exemple, quand $Weight_B(NFP1) = 0.01$ et $Weight_B(NFP2) = 0.60$, le poids attribué à QF est : $Weight_B(QF) = P(QF \mid \bigwedge\{Weight_B(e_0) \mid e_0 \in QF\}) = P(QF \mid \{(Weight_B(NFP1) = 0.01) \wedge (Weight_B(NFP2) = 0.99)\}) = 0.30$.

La pondération bayésienne permet donc de modéliser le caractère indispensable d'un élément, tout en préservant l'importance de chacun des autres éléments de l'arbre. De plus, on peut très facilement modéliser la pondération par distribution à l'aide de la pondération bayésienne. En fait, la première pondération est un cas particulier de la deuxième, où aucun élément n'est "indispensable". Cependant, l'inconvénient majeur de la pondération bayésienne réside dans le nombre important de poids qu'il faut attribuer. Pour que cette pondération soit pleinement

efficace, il faudrait, non pas laisser le concepteur remplir toutes les tables à la main, mais attribuer automatiquement à chaque nœud des tables de probabilités obtenues suite à des constats dressés après enquête sur les marchés aux composants. On pourrait constater par exemple que telle propriété non-fonctionnelle est traditionnellement plus importante que l'autre. Ou telle opération est généralement indispensable pour un composant d'un domaine précis et répondant à un besoin spécifique.

8.2.5 Formalisation de la stratégie de sélection

Des exemples de stratégie ont été proposés dans cette thèse. L'une d'entre elles utilise successivement la satisfaction et l'effort, tandis que l'autre combine pondération par distribution et visualisation. Ce sont des stratégies parmi d'autres, dont les étapes successives utilisent : i) un niveau de granularité du composant recherché ; ii) une méthode de comparaison ; iii) une pondération associée. Ces étapes sont reliées les unes aux autres, avec certaines conditions de passage. Par exemple, seuls les candidats qui possèdent exactement la documentation recherchée peuvent passer à l'étape suivante. Et on ne teste l'effort que sur une poignée de candidats parmi les plus "satisfaisants". Tout cela permet d'obtenir une stratégie cohérente qui filtre les candidats progressivement pour arriver jusqu'au meilleur. Une extension à ce travail de thèse consiste à formaliser la mise en place de telles stratégies, par le biais d'un langage ou d'une représentation graphique.

Un travail est en cours sur la mise en œuvre d'un outil de modélisation des stratégies de sélection. Cet outil utilise les réseaux de Petri colorés. Les réseaux de Petri [226, 202] ont connu de nombreuses applications, dont la modélisation des systèmes concurrents [196] et la mesure de leurs attributs qualité [197]. Les réseaux colorés [137], sont utilisés par exemple pour estimer le risque lié à la modélisation des systèmes concurrents [10]. Leur particularité est de différencier les jetons en leur associant des couleurs. Cela peut également être des constantes ou des listes ordonnées (ou n-uplets) de paramètres. A chaque place est donc associé l'ensemble de couleurs de jetons qui peuvent y séjourner. Et à chaque transition on associe un ensemble de "gardes" permettant de filtrer les jetons.

Il y a plusieurs raisons qui ont conduit à vouloir utiliser les réseaux de Petri colorés pour modéliser les différentes stratégies de sélection possibles. D'une part, les réseaux de Petri peuvent modéliser le parallélisme. Une telle représentation permettrait donc d'effectuer plusieurs calculs simultanément, chacun selon ses propres critères. On pourrait par exemple mesurer un indice de satisfaction avec deux pondérations différentes sur le même niveau de granularité. Ou effectuer un calcul sur la partie fonctionnelle du composant recherché et, en parallèle, un calcul sur sa partie non-fonctionnelle. D'autre part, avec une telle modélisation, le concepteur peut non seulement visualiser la stratégie qu'il a choisie, mais aussi ses résultats étape par étape. Les places du réseau associées aux transitions constituent ces étapes et leur ordonnancement, tandis qu'en représentant les candidats par des jetons colorés, on peut observer le parcours de chacun d'entre eux et les résultats obtenus à chaque étape.

La représentation choisie est donc la suivante, illustrée sur la figure 8.3, qui montre une application à la stratégie utilisant successivement la satisfaction et l'effort. Chaque place du réseau symbolise un niveau de description du composant recherché. Plus précisément, chaque place représente un sous-arbre de l'arbre décrivant le composant recherché, auquel on associe

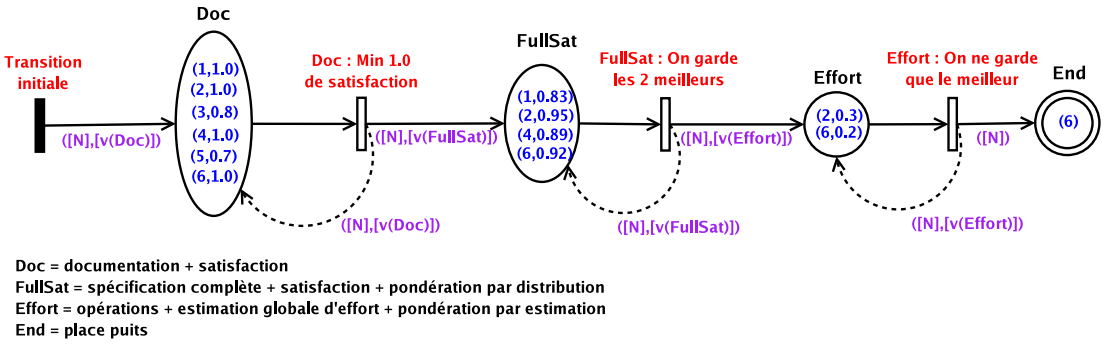

FIG. 8.3 – Modélisation de la stratégie de sélection : exemple

une méthode de comparaison et à une pondération particulières. La place *Doc* sur la figure 8.3 représente la documentation (les seuls nœuds concernés sont : le composant, sa documentation et les mots-clés typés qu'elle contient), associée à l'indice de satisfaction. La place *FullSat* représente l'arbre au complet, associé à l'indice de satisfation et la pondération par distribution. Enfin, la place *Effort* ne garde que les opérations, associées à l'estimation globale d'effort. Chaque jeton représente un candidat. La "couleur" associée à chaque jeton de chaque place P est de la forme $(N, v(N,P))$, où N est l'identifiant du jeton, et $v(P)$ est la valeur du résultat de la comparaison entre N et le composant recherché avec les paramètres de P. Chaque transition représente un filtre, qui va sélectionner certains candidats en entrée pour qu'ils se retrouvent dans les places suivantes. À chacune de ces transitions est associée une garde, qui indique la condition de passage des jetons. Sur la figure 8.3, un jeton ne peut passer de la place *Doc* à la place *FullSat* que si son indice de satisfaction dans la place *Doc* est égal à 1. Et de la place *FullSat* à la place *Effort*, on a indiqué que seuls les 2 meilleurs pouvaient passer. À chaque arc sortant d'une transition pour aller vers une place *Post* est associée la fonction $([N], v([N], Post)$. Cette fonction donne à chaque jeton filtré sa nouvelle valeur, c'est-à-dire celle du résultat de la comparaison avec le composant recherché selon les paramètres de *Post*. Les jetons qui ne peuvent pas passer resteront dans la dernière place où ils sont arrivés. Afin que le réseau soit complet, on lui associe deux "états spéciaux". La "transition initiale", représentée par un rectangle noir sur la figure 8.3, est celle par laquelle les jetons sont introduits l'un après l'autre dans le réseau. Et la place puits, représentée par un double cercle sur la figure, est celle qui termine le réseau en stockant le "meilleur" jeton-candidat.

8.2.6 Automatisation du chargement des candidats

Dans cette thèse, nous n'avons adressé ni le problème de l'extraction des descriptions de composants depuis les marchés et bibliothèques, ni le problème de la traduction de ces descriptions dans le format choisi pour le composant recherché. Nous avons considéré que cette traduction était déjà faite afin de nous concentrer sur les problèmes de la description du besoin et de la comparaison avec les descriptions traduites des candidats. Un travail est en cours

sur l'automatisation de l'extraction et de la traduction de ces descriptions, au moyen de techniques de transformation de modèles. En ce qui concerne l'extraction depuis des marchés et bibliothèques, il faut tenir compte de la manière dont chaque fournisseur documente ses propres composants. Dans le cas de *ComponentSource*, il s'agit de fichiers d'aide de formats variés (.pdf ou .hlp) contenant généralement les interfaces, avec leurs attributs, leurs méthodes et les signatures de celles-ci. Il faut également tenir compte des informations fournis sur les sites mêmes des marchés aux composants. En ce qui concerne la traduction des descriptions, il faut prendre en compte les modèles utilisées pour les composants candidats, comme EJB ou .NET.

Bibliographie

[1] Jan Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, Norway, 2001.

[2] Chris Abts. *Model Description : The COCOTS Extension of COCOMO II*. USC Center for Software Engineering, Texas A&M University, October 2002.

[3] H. Adeli and J. Wilcoski. A methodology for the evaluation of structural design software. *Computer & Structures*, 49 (5) :877–883, May 1993.

[4] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. The software maintenance project effort estimation model based on function points. In *Journal of Software Maintenance and Evolution : Research and Practice*, chapter 15, pages 71–85. John Wiley and Sons, 2003.

[5] A. Albrecht and J. Gaffney. Software function, source lines of code, and development effort prediction : a software science validation. *IEEE Transactions On Software Engineering*, SE-9 (6) :639–648, June 1983.

[6] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Meira. A software component quality model : A preliminary evaluation. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA)*, Dubrovnik, Croatia, August 2006.

[7] C. Alves and J.Castro. Cre : A systematic method for cots component selection. In *Brazilian Symposium on Software Engineering*, Rio de Janeiro, Brazil, October 2001.

[8] Carina Alves, Xavier Franch, Juan Pablo Carvallo, and Anthony Finkelstein. Using goals and quality models to support the matching analysis during cots selection. In *Proceedings of 4th International Conference on COTS-Based Software Systems (ICCBSS)*, pages 146–156, 2005.

[9] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 234–242, 1987.

[10] H. Ammar, T. Nikzadeh, and J. Dugan. A methodology for risk assessment of functional specification of software systems using colored petri nets. In *Proceedings of IEEE International Symposium on Software Metrics (METRICS)*, 1997.

[11] E. Anderson. A heuristic for software evaluation and selection. *Software Practice and Experience*, 19 (8) :707–717, August 1989.

[12] Annelise Andrews, Sudipto Ghosh, and Eun Man Choi. A model for understanding software components. In *Proceedings of International Conference on Software Maintenance (ICSM)*, 2002.

[13] J. Baik, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions On Software Engineering*, 25 (4) :513–583, April 1999.

[14] J. Baik, B. Boehm, and B. Steece. Disaggregating and calibrating the case tool variable in cocomo-ii. *IEEE Transactions On Software Engineering*, 28 (11) :1009–1022, November 2002.

[15] Keith Ballurio, Betsy Scalzo, and Lou Rose. Risk reduction in cots software selection with basis. In *Proceedings of the 1st International Conference on COTS-Based Component Systems (ICCBSS)*, pages 31–43, Orlando, Florida, USA, 2002.

[16] V. Basili, G. Caldiera, and H. Rombach. The experience factory. In *Encyclopedia of Software Engineering*, pages 470–476. John Wiley and Sons, 1994.

[17] Victor Basili. Software development : A paradigm for the future (keynote address). In *Proceedings of the 13th Annual International Computer Software and Applications Conference (COMPSAC)*, 1989.

[18] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[19] F. Bastani. An approach to measuring program complexity. In *Proceedings of the 7th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 1–8. IEEE Computer Science Press, 1983.

[20] Manuel Bertoa and Antonio Vallecillo. Quality attributes for cots components. In *Proceedings of the ECOOP Workshop on Quantitative Aspects of Object-Oriented Software Engineering (QaOOSE)*, June 2002.

[21] Manuel Bertoa and Antonio Vallecillo. Quality attributes for cots components. *I+D Computación*, 1 (2) :128–144, November 2002.

[22] Antoine Beugnard, Jean-Marc Jézéquel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32 (7) :38–45, July 1999.

[23] L. Beus-Dukic and A. Wellings. Requirements for a cots software component : A case study. *Conference on European Industrial Requirements Engineering (CEIRE)*, 3 (2) :115–120, October 1998.

[24] G. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.

[25] Gordon Blair, Lynne Blair, and Jean-Bernard Stefani. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and ISDN Systems*, 29 :473–500, 1997.

[26] Jorgen Boegh. Certifying software component attributes. *IEEE Software*, 40 (5) :74–81, May 2006.

[27] B. Boehm, C. Abts, A. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, July 2000.

[28] B. Boehm, D. Port, Y. Yang, and J. Buhta. Not all cbs types are created equally : Cots-intensive project types. In *Proceedings of 2nd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 36–50, Ottawa, Canada, February 2003.

[29] B. Boehm and W. Royce. Ada cocomo and the ada process model. In *Proceedings 5th COCOMO Users' Group Meeting*, Software Engineering Institute, Pittsburgh, PA, USA, November 1989.

[30] Barry Boehm. *Software Engineering Economics*. Prentice Hall, July 1981.

[31] Barry Boehm and Chris Abts. Software integration : Plug and pray ? *IEEE Computer*, 32 (1) :135–138, January 1999.

[32] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models fot future software life cycle processes : Cocomo 2.0. In *Annals of Software Eng. Special Volume on Software Process and Product Measurement*, volume 1, pages 45–60, Amsterdam, The Netherlands, 1995.

[33] Germinal Boloix and Pierre Robillard. A software system evaluation framework. *IEEE Computer*, 28 (12) :17–26, December 1995.

[34] A. Bookstein and D. Swanson. Probabilistic models for automatic indexing. *Journal of the American Society for Information Science*, 25 (5) :312–318, May 1974.

[35] L. Bornwsword, D. Carney, and T. Obendorf. The opportunities and complexities of applying commercial off-the-shelf components. *Crosstalk*, April 1998.

[36] L. Bornwsword, P. Obendorf, and C. Sledge. Developing new processes for cots-based systems. *IEEE Software*, 34 (4) :48–55, July-August 2000.

[37] Noureddine Boudriga, Ali Mili, and Roland Mittermeir. Semantic-based software retrieval to support rapid prototyping. *Structured Programming*, 13 (3) :109–127, 1992.

[38] Premysl Brada. *Specification-Based Component Substituability and Revision Identification*. PhD thesis, Charles University, Prague, Czesch Republic, 2003.

[39] L. Briand, C. Differding, and D. Rombach. Practical guidelines for measurement-based process improvement. *Software Process Improvement and Practice*, 2 (4) :253–280, 1996.

[40] S. Brin and L. Page. The anatomy of a large-scale hypertewtual web search engine. *Computer Networks and ISDN Systems*, 30 (1-7) :107–117, 1998.

[41] F. Brooks. No silver bullet : Essence and accidents of software engineering. *IEEE Computer*, 20 (4) :10–19, April 1987.

[42] Alan Brown and Kurt Wallnau. A framework for evaluating software technology. *IEEE Software*, 13 (5) :39–49, September 2006.

[43] I. Brownstein and N. Lerner. *Guidelines for Evaluating and Selecting Software Packages*. Elsevier Science Publishing Co., Inc., 1982.

[44] Kim Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 104–127, 1997.

[45] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal composition framework specification. final release, version 1.0. http ://fractal.objectweb.org/doc/index.html, July 2002.

[46] C. Buckley, G. Salton, and J. Allan. Automatic retrieval with locality information using smart. In *TREC-1 Proceedings*, pages 69–72, 1992.

[47] X. Burgués, C. Estay, X. Franch, J. Pastor, and C. Quer. Combined selection of cots components. In *Proceedings of the 1st International Conference on COTS-Based Component Systems (ICCBSS)*, pages 54–64, Orlando, Florida, USA, 2002.

[48] Gianluigi Caldiera and Victor Basili. Identifying and qualifying resuable software components. *IEEE Computer*, 24 (2) :61–70, 1991.

[49] D. Cameron and B. Rosenblatt. *Learning GNU Emacs*. O'Reilly and Associates, Sebastopol, CA., 1991.

[50] Luca Cardelli. A semantics of multiple inheritance. In D.B. MacQueen G. Kahn and G. Plotkin, editors, *Proceedings of International Symposium on Semantics of Data Types*, pages 51–67, Sophia-Antipolis, France, June 1984. Springer-Verlag.

[51] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76 (2) :138–164, February-March 1988.

[52] Juan Pablo Carvallo and Xavier Franch. Extending the iso/iec 9126-1 quality model with non-technical factors for cots component selection. In *Proceedings of Workshop on Software Quality (WosQ)*, pages 9–14, 2006.

[53] Juan Pablo Carvallo, Xavier Franch, Gemma Grau, and Carme Quer. Qm : A tool for building software quality models. In *Proceedings of the 11th Conference on Requirements Engineering (RE)*, Kyoto, Japan, 2004.

[54] Juan Pablo Carvallo, Xavier Franch, and Carme Quer. A process for cots software product evaluation. In *Proceedings of 2nd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 51–62, Ottawa, Canada, 2003.

[55] Juan Pablo Carvallo, Xavier Franch, Carme Quer, and Marco Torchiano. Characterization of a taxonomy for business applications and the relationships among them. In *Proceedings of 3rd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 221–231, 2004.

[56] CCITT/ITU. *ITU Recommandation E800 (08/94) : Terms and Definitions Related to Quality of Service and Network Performance Including Dependability*, 1994.

[57] Alejandra Cechich and Mario Piattini. Filtering cots components through an improvement-based procedd. In *Proceedings of 4th International Conference on COTS-Based Software Systems (ICCBSS)*, pages 112–121, 2005.

[58] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20 (6) :476–493, June 1994.

[59] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.

[60] Lawrence Chung and Kendra Cooper. A knowledge-based cots-aware requirements engineering approach. In *Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Ischia, Italy, July 2002.

[61] E. Colombo and C. Francalanci. Selecting crm packages based on architectural, functional, and cost requirements : Empirical validation of a hierarchical ranking model. *Requirements Engineering*, 9 (3) :186–203, March 2004.

[62] S. Comella-Dorda, J. Dean, E. Morris, and T. Oberndorf. A process for cots software product evaluation. In *Proceedings of 1st International Conference on COTS-Based Software Systems (ICCBSS)*, pages 46–56, Orlando, Florida, USA, 2002.

[63] ComponentSource. Website. http ://www.componentsource.com, 2005.

[64] R. Di Cosmo. Type isomorphisms in a type-assignment framework. In *Proceedings of the 19th Annual Symposium on Principles Of Programming Languages (POPL)*, pages 200–210, 1992.

[65] R. Cowell, A. Dawid, S. Lauritzen, and D. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag New York, 1999.

[66] Ivica Crnkovic, Stig Larsson, and Michel Chaudron. Component-based development process and component lifecycle. In *27th International Conference on Information Technology Interfaces (ITI)*, Cavtat, Croatia, June 2005. IEEE.

[67] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Integrating nonfunctional requirements into data model. In *Proceedings of the 4th International Symposium on Requirements Engineering (ISRE)*, Ireland, June 1999.

[68] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Nonfunctional requirements : From elicitation to modelling languages. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 699–700, Orlando, Florida, USA, May 2002.

[69] Darren Dalcher. Disaster in london : The las case study. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS)*, pages 41–52, 1999.

[70] Erwan Daubert, Thomas Boggini, and Bart George. Substitute tool v3. http ://www-valoria.univ-ubs.fr/SE/Substitute/, 2007.

[71] A. Davis. *Software Requirements : Objects Functions and States*. Prentice Hall, 1993.

[72] J. Davis and R. LeBlanc. A study of the applicability of complexity measures. *IEEE Transactions on Software Engineering*, 14 (9) :1366–1371, September 1988.

[73] Eduardo Santana de Almeida, Alexandre Akvaro, Vinicius Cardoso Garcia, Jorge Cordiro Pires Mascena, Vanilson André de Arruda Burégio, Leandro Marques do Nascimento, Daniel Lucrédio, and Silvio Lemos Meira. *C.R.U.I.S.E. : Component ReUse In Software Engineering*. C.E.S.A.R. (Recife Center for Advanced Studies and Systems) eds., April 2007.

[74] Karine Macedo de Amorim. *Modélisation d'aspects qualité de service en UML : application aux composants logiciels*. PhD thesis, Université de Rennes 1, France, 2004.

[75] Olivier Defour, Jean-Marc Jézéquel, and Noel Plouzeau. Extra-functional contract support in components. In *Proceedings of 7th International Symposium on Component-Based Software Engineering (CBSE 7)*, May 2004.

[76] Frank DeRemer and Hans Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2 (2) :80–86, June 1976.

[77] P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard. A knowledge-based software information system. *Communications of the ACM*, 34 (5) :34–39, May 1991.

[78] Edger Dijkstra. The structure of the "the" - multiprogramming system. *Communications of the ACM*, 11 (5) :341–346, May 1968.

[79] A. Dix. *Formal Methods for Interactive Systems*. Academic Press, London, UK, 1991.

[80] C. Bana e Costa. *Readings in Multiple Criteria Decision Aid*. Springer-Verlag, 1990.

[81] L. Edmonds and J. Urban. A method for evaluating front-end life-cycle tools. In *Proceedings of the IEEE International Conferene on Computers and Applications*, pages 324–331, 1984.

[82] W. Edwards and D. Von Winterfeldt. *Decision Analysis and Behavioural Research*. Cambridge University Press, 1986.

[83] Ceyda Güngör En and Hayri Baraçli. A brief literature review of enterprise software evaluation and selection methodologices : A comparison in the context of decision-making methods. In *Procedings of the 5th International Symposium on Intelligent Manufacturing Systems*, pages 874–883, May 2006.

[84] A. Eskenasi. Evaluation of software product quality by means of classifiation methods. *Journal of Systems and Software*, 10 (3) :213–216, March 1989.

[85] M. Feblowitz and S. Greenspan. Scenario-based analysis of cots acquisition impacts. *Requirements Engineering*, 3 (3-4) :182–200, March-April 1998.

[86] N. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions On Software Engineering*, 25 (5) :675–689, May 1999.

[87] N. Fenton and M. Neil. Bayesian belief nets : a causal model for predicting defect rates and resource requirements. *Software Testing and Quality Engineering*, 2 (1) :48–53, January 2000.

[88] N. Fenton and S. Pfleeger. *Software Metrics : A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[89] A. Field and P. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[90] Anthony Finkelstein and John Dowell. A comedy of errors : the london ambulance service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 2–5. IEEE Computer Society Press, 1996.

[91] G. Finnie, G. Wittig, and D. Petkov. Prioritizing software development productivity factors using the analytic hierarchy process. *Journal of Systems and Software*, 22 :129–139, September 1995.

[92] Gehrard Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4 (4) :60–72, 1987.

[93] Gehrard Fischer. User modeling in human-computer interaction. *User Modeling and User-Adapted Interaction*, 11 (1 & 2) :65–86, 2001.

[94] C. Fox. Lexical analysis and stoplists. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval : Data Structures and Algorithms*, pages 102–130. Prentice Hall, Englewood Cliffs, NJ, 1992.

[95] Greg Fox, Karen Lantner, and Steven Marcom. A software development process for cots-based information system infrastructure. In *Proceedings of the 5th IEEE International Symposium on Assessment of Software tools (SAST)*, Los Alamitos, California, USA, June 1997.

[96] W. Frakes and C. Fow. Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, 22 (4) :274–279, April 1996.

[97] W. Frakes and C. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38 (6) :75–87, June 1995.

[98] Xavier Franch and Juan Pablo Carvallo. Using quality models in software package selection. *IEEE Software*, 37 (1) :34–41, January 2003.

[99] Xavier Franch and Neil Maiden. Modelling component dependencies to inform their selection. In *Proceedings of the 2nd International Conference on COTS-Based Component Systems (ICCBSS)*, pages 81–91, Ottawa, Canada, 2003.

[100] S. Frankel. *Guidance on Software Package Selection*. NBS Special Publication, US Department of Commerce, National Bureau of Standard, Washington DC, USA, 1986.

[101] Svend Frolund and Jari Koistinen. Qml : A language for quality of service specification. Technical report, Hewlett-Packard Laboratories, Palo Alto, California, USA, 1998.

[102] Svend Frolund and Jari Koistinen. Quality-of-service specification in distributed oject systems. *Distributed Systems Engineering Journal*, 5 (4) :179–202, April 1998.

[103] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering (ICSE)*, pages 179–185, 1995.

[104] Bart George, Régis Fleurquin, and Salah Sadou. A component-oriented substitution model. In *Proceedings of 9th International Conference on Software Reuse (ICSR 9)*, June 2006.

[105] Bart George, Régis Fleurquin, and Salah Sadou. A methodological approach for selecting components in development and evolution process. *Electronic Notes on Theoretical Computer Science (ENTCS)*, 6 (2) :111–140, January 2007.

[106] Sudipto Ghosh, John Kelly, and Roopashree Shankar. Enabling the selection of cots components. In *Proceedings of 4th International Conference on COTS-Based Software Systems (ICCBSS)*, pages 122–131, 2005.

[107] N. Gill and P. Grover. Few important considerations for deriving interface complexity metric for component-based software. *ACM SIGSOFT Software Engineering Notes (SEN)*, 29 (2), February 2004.

[108] M. Girardi. *Classification and Retrieval of Software through their Description in Natural Language*. PhD thesis, University of Geneva, Switzerland, 1995.

[109] M. Girardi and B. Ibrahim. A software reuse system based on natural language specifications. In *Proceedings of International Conference on Computing and Information (ICCI'93)*, pages 507–511, Sudbury, Ontario, Canada, 1993.

[110] M. Girardi and B. Ibrahim. A similarity measure for retrieving software artifacts. In *Proceedings of the Sixth International Conference on Softare Engineering and Knowledge Engineering (SEKE 6)*, pages 478–485, Jurmala, Latvia, June 1994.

[111] Google. Website. http ://www.google.com.

[112] Miguel Goulao and Fernando Brito e Abreu. Software components evaluation : an overview. In *5th Conferencia da Associaçao Portuguesa de Sistemas de Informaçao (CAPSI)*, November 2004.

[113] Gemma Grau, Juan Carvallo, Xavier Franch, and Carme Quer. Descots : A software system for selecting cots components. In *30th EUROMICRO Conference*, pages 118–126, Rennes, France, September 2004.

[114] Gary Haines, David Carney, and John Foreman. Component-based software development / cots integration. *Software and Technology Review*, 1997.

[115] Daniel Halbert and Patrick O'Brien. Using types and inheritance in object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 20–31, 1987.

[116] M. Halstead. *Elements of Software Science*. Elsevier, New York, USA, 1977.

[117] G. Heineman and W. Councill. *Component-Based Software Engineering : Putting the Pieces Together*. Addison Wesley, 2001.

[118] David Hemer and Peter Lindsay. Specification-based retrieval strategies for module reuse. In *Proceedings of the Australian SoftWare Engineering Conference (ASWEC)*, pages 235–243, Canberra, Australia, August 2001.

[119] Scott Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 10 (5) :48–59, September 1994.

[120] Scott Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 6 (2) :111–140, April 1997.

[121] S. Henry and D. Kafura. Software metrics based on information flow. *IEEE Transactions on Software Engineering*, 7 (5) :510–518, May 1981.

[122] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

[123] M. Hokey. Selection of software : The analyic hierarchy process. *International Journal of Physical Distribution and Logistics Management*, 22 (1) :42–52, January 1992.

[124] IEEE. *IEEE Std. 1209-1992 : IEEE Recommended Practice for the Evaluation and Selection of CASE Tools*, ieee computer society press, new york edition, 1993.

[125] IEEE. *IEEE Std. 1061-1998 : IEEE Standard for a Software Quality Metrics Methodology*, ieee computer society press edition, 1998.

[126] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank : Relative significance for software component search. In *International Conference on Software Engineering (ICSE)*, 2003.

[127] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions On Software Engineering*, 31 (3) :407–423, March 2005.

[128] Tomás Isakowitz and Robert Kauffman. Supporting search for reusable software objects. *IEEE Transactions On Software Engineering*, 22 (6) :407–423, June 1996.

[129] ISO International Standards Organisation. *ISO 8402 :1986 Quality Vocabulary*, 1986.

[130] ISO International Standards Organisation. *ISO/IEC JTC1/SC21 N9309 :1995 QoS - Basic Framework*, 1995.

[131] ISO International Standards Organisation, Geneva, Switzerland. *ISO/IEC 14598-1 :1999 Information Technology - Software Product Evaluation*, 2001.

[132] ISO International Standards Organisation, Geneva, Switzerland. *ISO/IEC 9126-1 :2001 Software Engineering - Product Quality - Part I : Quality model*, 2001.

[133] S. Isoda. Experiences of a software reuse project. *Journal of Systems and Software*, 30 :171–186, 1995.

[134] J. Jeanrenaud and P. Romanazzi. Software product evaluation : A methodological approach. In *Proceedings of the 2nd International Conference on Software Quality Management*, pages 59–69, 1994.

[135] Jun-Jang Jeng and Betty Cheng. Specification matching for software reuse : A foundation. In *Proceedings of ACM Symposium on Software Reuse (SSR)*, pages 97–105, Seattle, Washington, USA, April 1995.

[136] F. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag New York, 2001.

[137] Kurt Jensen. *Coloured Petri Nets Volume I : Basic Concepts*. Springer-Verlag, 1992.

[138] Lamia Labed Jilani, Jules Desharnais, Marc Frappier, Rym Mili, and Ali Mili. Retrieving software components that minimize adaptation effort. In *Proceedings of the 12th Automated Software Engineering Conference (ASE)*, pages 255–262, November 1997.

[139] Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Defining and applying measures of distance between specifications. *IEEE Transactions on Software Engineering*, 27 (8) :673–703, August 2001.

[140] C. Jones. *Programming Productivity*. McGraw Hill, New York, USA, 1986.

[141] Philippe Jorrand. Data types and extensible languages. *ACM SIGPLAN Notices*, 6 (12) :75–83, December 1971.

[142] H. Jung and B. Choi. Optimization models for quality and cost of modular software systems. *European Journal of Operational Research*, 112 :613–619, 1999.

[143] E.-A. Karlsson. *Software Reuse - A Holistic Approach*. Wiley, 1995.

[144] R. Keeney and H. Raiffa. *Decision With Multiple Objectives*. Wiley, New York, 1976.

[145] O. Khayati, A. Front, and J.-P. Giraudin. A metamodel for a metatool to descibe and manage components. In *Proceedings of the IEEE International Conference on Information & Communication Technologies : from Theory to Applications (ICCTTA)*, Damascus, Syria, April 2004.

[146] Oualid Khayati. *Modèles formels et outils génériques pour la gestion et la recherche de composants*. PhD thesis, INPG (Institut National Polytechnique de Grenoble), France, 2005.

[147] B. Kitchenham and L. Jones. Evaluating software engineering methods and tools : Part5, the influence of human factors. *ACM SIGSOFT Software Engineering Notes*, 22 (1), January 1997.

[148] Barbara Kitchenham, Stephen Linkman, and David Law. Desmet : A methodology for evaluating software engineering methods and tools. *IEEE Computing and Control Engineering Journal*, 8 (3) :120–126, June 1997.

[149] I. Klopping and C. Bolgiano. Effective evaluation of off-the-shef microcomputer software. *Office Systems Research Journal*, 9 :46–50, 1990.

[150] K. Kobori, T. Yamamoto, M. Matsushita, and K. Inoue. Classification of java programs in spars-j. In *Proceedings of the International Workshop on Community-Driven Evolution of Knowledge Artifacts*, 2003.

[151] J. Kontio, S. Chen, K. Limperos R. Tesoriero, G. Caldiera, and M. Deutsch. A cots selection method and experiences of its use. In *Proceedings of the 20th Annual Software Engineering Workshop (NASA)*, Greenbelt, Maryland, USA, 1995.

[152] Jyrki Kontio. A case study in applying a systematic method for cots selection. In *Proceedings of International Conference on Software Engineering (ICSE)*, 1996.

[153] Jyrki Kontio, Gianluigi Caldiera, and Victor Basili. Defining factors, goals and criteria for reusable component evaluation. In *IBM Center for Advanced Studies Conference (CASCON '96)*, Toronto, Canada, November 1996.

[154] T. Koopmans. Analysis of production as an efficient combination of activities. In *Activity Analysis of Production and Allocation*, pages 33–97. John Wiley and Sons, 1951.

[155] Chikako Van Koten and Andrew Gray. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology*, 2005.

[156] J. Kuan. Nasa pc software evaluation project. In *USL/DBMS NASA/PC, R&D-18, Working Paper Series*, 1986.

[157] H. Kuhn and A. Tucker. Nonlinear programming. In *Proceedings of the 2nd Berkeley Symposium on Mathematical Statistical and Probability*, pages 481–491, 1951.

[158] Douglas Kunda. *A Social-Technical Approach to Selecting Software Supporting COTS-Based Systems*. PhD thesis, University of York, UK, 2001.

[159] Douglas Kunda and Laurence Brooks. Applying social-technical approach for cots selection. In *United Kingdom Academy for Information Systems Conference (UKAIS 1999)*, University of York, United Kingdom, April 1999.

[160] V. Lai, R. Truebloob, and B. Wong. Software selection : A case study of the application of the analytical hierearchical process to the selection of a multimedia authoring system. *Information & Management*, 36 :221–232, 1999.

[161] V. Lai, B. Wong, and W. Cheung. Group decision making in multiple criteria environment : A case using the ahp in software selection. *European Journal of Operational Research*, 137 :134–144, 2002.

[162] B. Lange and T. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of Human Factors in Computing Systems*, pages 69–73, Austin, Texas, USA, 1989.

[163] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of ACM Symposium on Automated Software Engineering (ASE)*, November 2005.

[164] P. Lawlis, K. Mark, D. Thomas, and T. Courtheyn. A formal process for evaluating cots software products. *IEEE Computer*, 34 (5) :58–63, May 2001.

[165] Gary Leavens and William Weihl. Reasoning about object-oriented programs that use subtypes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) and the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 212–223, October 1990.

[166] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. Dream : a component framework for the construction of resource-aware, configurable moms. *IEEE Distributed Systems Online*, September 2005.

[167] Grace Lewis. An approach to analysis and design for cots-based systems. In *Proceedings of 4th International Conference on COTS-Based Software Systems (ICCBSS)*, pages 236–247, 2005.

[168] Grace Lewis and Edwin Morris. From system requirements to cots evaluation criteria. In *Proceedings of 3rd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 159–168, 2004.

[169] Grace Lewis and Lutz Wrage. A process for context-based technology evaluation : Examples for the evaluation of web services technology. In *Proceedings of 5th International Conference on COTS-Based Software Systems (ICCBSS)*, 2006.

[170] Han Lin, Anh Lai, Rebecca Ullrich, Michael Kuca, Kelly McClelland, Jessca Shaffer-Grant, Sandra Pacheco, Karen Dalton, and William Watkins. Cots software selection process. In *Proceedings of 6th International Conference on COTS-Based Software Systems (ICCBSS)*, pages 114–120, 2007.

[171] Barbara Liskov and Jeanette Wing. A new definition of the subtype relation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 118–141, 1993.

[172] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16 (6) :1811–1841, November 1994.

[173] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9 (4) :50–59, April 1974.

[174] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[175] Adolfo Lozano-Tello and Asunción Gómez-Pérez. Baremo : How to choose the appropriate software component using the analytic hierarchy process. In *Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Ischia, Italy, July 2002.

[176] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17 (8) :800–813, August 1991.

[177] Ole Lehman Madsen and Birger Moller-Pedersen. What object-oriented programming may be - and what it does not have to be. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 1–20, 1988.

[178] N. Maiden, V. Croce, H. Kim, G. Sajeva, and S. Topuzidou. Scarlet : Integrated process and tool support for selecting software components. In LNCS, editor, *Component-Based Software Quality*, pages 85–98. Springer-Verlag, 2003.

[179] N. Maiden, H. Kim, and C. Ncube. Rethinking process guidance for selecting software components. In *Proceedings of the 1st International Conference on COTS-Based Component Systems (ICCBSS)*, pages 151–164, Orlando, Florida, USA, 2002.

[180] Neil Maiden and Cornelius Ncube. Acquiring cots software selection requirements. *IEEE Transactions on Software Engineering*, 24 (3) :46–56, March 1998.

[181] Neil Maiden, Cornelius Ncube, and Andrew Moore. Lessons learned during requirements acquisition for cots systems. *Communications of the ACM*, 40 (12) :21–25, December 1997.

[182] Y. Matsumoto. A software factory : An overall approach to software production. In P. Freeman, editor, *Software Reusability*, pages 155–178. IEEE Computer Society, March 1987.

[183] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2 (4) :308–320, April 1976.

[184] M. McIlroy. Mass produced software components. In *NATO Software Engineering Conference Report*, pages 79–85, Garmish, Germany, October 1968.

[185] N. Medvidovic, P. Oreizy, and R. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of Symposium on Software Reusability (SSR) and Proceedings of International Conference on Software Engineering (ICSE)*, Boston, Massachussets, USA, May 1997.

[186] Nenad Medvidovic. *Architecture-Based Specification-Time Software Evolution*. PhD thesis, University of California, Irvine, USA, 1999.

[187] Nenad Medvidovic and Richard Taylor. Exploiting arcchitectural style to develop a family of applications. In *IEE Proceedings Software Engineering*, October-December 1997.

[188] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1988.

[189] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25 (10), October 1992.

[190] Jean-Christophe Mielnik, Bernard Lang, Stéphane Laurière, Jean-Georges Schlosser, and Vincent Bouthors. ecots platform : An inter-industrial initiative for cots-related information sharing. In *Proceedings of 2nd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 157–167, Ottawa, Canada, 2003.

[191] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components : A refinement based approach. In *Proceedings of International Conference on Software Engineering (ICSE)*, 1994.

[192] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software : Issues and research directions. *IEEE Transactions On Software Engineering*, 21 (6) :528–562, June 1995.

[193] Rym Mili, Ali Mili, and Roland T. Mittermeir. Storing and retrieving software components : A refinement based system. *IEEE Transactions On Software Engineering*, 23 (7) :445–460, July 1997.

[194] H. Min. Selection of software : The analytic hierarchy process. *International Journal of Physical Distribution and Logistics Management*, 22 (1) :42–52, 1992.

[195] T. Miyoshi and M. Azuma. An empirical study of evaluating software development environment quality. *IEEE Transactions on Software Engineering*, SE-19 :425–435, 1993.

[196] Sandro Morasca. Measuring attributes of concurrent software specifications in petri nets. In *Proceedings of IEEE International Symposium on Software Metrics (METRICS)*, 1999.

[197] Sandro Morasca and Lionel Briand. Towards a theoretical framework for measuring software attributes. In *Proceedings of IEEE International Symposium on Software Metrics (METRICS)*, 1997.

[198] D. Morera. Cots evaluation using desmet methodology & analytic hierarchy process. In *PROFESS*, pages 485–493, 2002.

[199] M. Morisio and A. Tsoukiàs. Iusware : A methodology for the evaluation and selection of software products. *IEEE Proceedings Software Engineering*, 144 (3) :162–174, June 1997.

[200] V. Mosley. How to assess tools efficiently and quantitatively. *IEEE Software*, 8 (5) :29–32, May 1992.

[201] E. Mumford. *Effective Systems Design and Requirements Analysis*. Manchester Business School, Manchester, UK, 1995.

[202] Tadao Murata. Petri nets : properties, analysis and applications. In *Proceedings of the IEEE*, volume 77(4), pages 541–580, April 1989.

[203] G. Myers. *Composite/Structured Design*. Van Nostrand Reinhold, 1978.

[204] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using non-functional requirements : A process-oriented approach. *IEEE Transactions on Software Engineering*, 18 (6) :483–497, June 1992.

[205] Fredy Navarrete, Pere Botella, and Xavier Franch. Reconciling agility and discipline in cots selection process. In *Proceedings of 6th International Conference on COTS-Based Software Systems (ICCBSS)*, 2007.

[206] Cornelius Ncube. *A Requirements Engineering Method for COTS-Based Software Development*. PhD thesis, City University, London, UK, 2000.

[207] Cornelius Ncube and John Dean. The limitations of current decision-making techniques in the procurement of cots software component. In *Proceedings of the 1st International Conference on COTS-Based Component Systems (ICCBSS)*, pages 176–187, Orlando, Florida, USA, 2002.

[208] M. Neil, N. Fenton, and L. Nielsen. Building large-scale bayesian networks. *The Knowledge Engineering Review*, 15 (3) :257–284, March 2000.

[209] Oscar Nierstrasz. What is the "object" in object-oriented programming ? In *Proceedings of the CERN School of Computing*, pages 43–53, Renesse, The Netherlands, August-September 1986.

[210] Oscar Nierstrasz. Requirements for a composition language. In *Proceedings of the ECOOP Workshop on Models and Languages for Coordination of Parallelism and Distribution*, pages 147–161, 1994.

[211] Patricia Oberndorf. Facilitating component-based software engineering : Cots and open systems. In *Proceedings of the 5th IEEE International Symposium on Assessment of Software tools (SAST)*, pages 143–148, Los Alamitos, California, USA, June 1997.

[212] M. Ochs, D. Pfahl, G. Chrobok-Diening, and B. Nothelfer-Kolb. A cots acquisition process : Definition and application experience. In *Proceedings of the 11th European Software Control and Metrics Conference (ESCOM)*, pages 335–343, 2000.

[213] M. Ochs, D. Pfahl, G. Chrobok-Diening, and B. Nothelfer-Kolb. A method for efficient measurement-based cots assessment and selection - method description and evaluation results. In *Proceedings of the IEEE Symposium on Software Metrics (METRICS)*, 2001.

[214] OMG. Corba components v 3.0. http ://www.omg.org/technology/documents/, 2003.

[215] OMG. Uml 2.0 superstructure final adopted specification, document ptc/03-08-02. http ://www.omg.org/docs/ptc/03-08-02.pdf, August 2003.

[216] Rob Van Ommering. Software reuse in product populations. *IEEE Transactions on Software Engineering*, 31 (7) :537–550, July 2005.

[217] D. Page, P. Williams, and D. Boyd. Report of the inquiry into the london ambulance service. Technical report, Communications Directorate of South West Thames Regional Health Authority, London, UK, February 1993.

[218] Jens Palsberg and Michael Schwartzbach. Type susbtitution for object-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) and the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 151–160, October 1990.

[219] Jens Palsberg and Michael Schwartzbach. Three discussions on object-oriented typing. *ACM SIGPLAN OOPS Messenger*, 3 (2), 1992.

[220] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 (12) :1053–1058, December 1972.

[221] Peter Patel-Schneider. A four-value semantics for frame-based description languages. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI)*, pages 344–368, Philadelphia, USA, August 1986.

[222] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of 9th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 535–542. Knowledge Systems Institute, 1997.

[223] John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering : An International Journal*, 6 (2) :139–170, April 1999.

[224] John Penix, Phillip Baraona, and Perry Alexander. Classification and retrieval of reusable components using semantic features. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 131–138, 1995.

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

[226] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[227] Barbara Phillips and Susan Polen. Add decision analysis to your cots selection process. *CrossTalk*, pages 21–25, April 2002.

[228] D. Port and Z. Chen. Assessing cots assessment : How much is enough. In *Proceedings of 3rd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 183–198, Los Angeles, California, USA, February 2004.

[229] D. Port and Y. Yang. Empirical analysis of cots activity effort sequences. In *Proceedings of 3rd International Conference on COTS-Based Software Systems (ICCBSS)*, Los Angeles, California, USA, February 2004.

[230] R. Poston and M. Sexton. Evaluating and selecting testing tools. *IEEE Software*, 8 (5) :33–42, May 1992.

[231] J. Poulin and K. Werkman. Modeling stryctyred abstracts and the world wide web for retrieval of reusable components. In *Proceedings of Symposium on Software Reuse*, Seattle, Washington USA, April 1995.

[232] A. Powell, A. Vickers, W. Lam, and E. Williams. Evaluating tools to support component-based software engineering. In *Proceedings of the 5th IEEE International Symposium on Assessment of Software tools (SAST)*, pages 80–89, Los Alamitos, California, USA, June 1997.

[233] R. Príeto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4 (1) :6–7, January 1987.

[234] Ruben Príeto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34 (5) :88–97, May 1991.

[235] DREAM Project. Dream library performance test results. http ://dream.objectweb.org/junit/dreamlib-perf.html, 2005.

[236] DREAM Project. Dream : a component-based communication framework. http ://-dream.objectweb.org/index.html, 2006.

[237] Vivien Quéma. *Vers l'exologiciel : une approche de la construction d'infrastructures logicielles radicalement reconfigurables*. PhD thesis, Institut National Polytechnique de Grenoble, France, 2005.

[238] Carme Quer, Xavier Franch, and Xabier Lopez-Pelegrín. Descots-ev : A tool for the evaluation of cots components. In *Proceedings of the 13th Conference on Requirements Engineering (RE)*, 2005.

[239] Rajendra Raj and Henry Levy. A compositional model for software reuse. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 3–24, 1989.

[240] S. Crespi Reghizzi, G. Calli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) and the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 148–166, October 1990.

[241] Mikael Rittri. Using types as search keys in function libraries. In *Proceedings of the 4th ACM International Conference on Functional Programming Languages and Computer Architecture*, pages 174–183, Imperial College, United Kingdom, 1989.

[242] Mikael Rittri. Retrieving library identifiers via equational matching of types. Technical report, Programming Methodology Group, Department of Computer Science, Chalmers University of echnology and University of Göteborg, Göteborg, Sweden, 1992.

[243] Douglas Ross. Toward foundation for the understanding of type. *ACM SIGPLAN Notices*, 11 (3) :63–65, March 1976.

[244] J. Rowley. Selection and evaluation of software. In *Aslib Proceedings*, volume 45, pages 77–81, 1993.

[245] B. Roy. *Méthodologie multicritère d'aide à la décision*. Economica, Paris, 1985.

[246] B. Roy. *Multicriteria Methodology for Decision Systems*. Kluwer Academy, Dordrecht, 1996.

[247] G. Rugg and P. McGeorge. Laddering. *Expert Systems*, 12 (4) :339–346, April 1995.

[248] C. Runcinman and I. Toyn. Retrieving reusable software components by polymorphic type. In *Proceedings of the 4th ACM International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, United Kingdom, 1989.

[249] Thomas Saaty. A scaling method for priorities in hierarchical structures. *Journal of Mathematical Psychology*, 15 :234–281, 1977.

[250] Thomas Saaty. *The Analytic Hierarchy Process*. McGraw Hill, New York, 1980.

[251] Thomas Saaty. How to make a decision : The analytic hierarchy process. *European Journal of Operational Research*, 48 :9–26, 1990.

[252] Thomas Saaty. Analytic hierarchy. In *Encyclopedia of Science and Technology*, pages 559–563. McGraw-Hill, 1992.

[253] B. Sahay and A. Gupta. Development of software selection criteria for supply chain solutions. *Industrial Management Data Systems*, 103 (2) :97–110, February 2003.

[254] Vijai Sai, Xavier Franch, and Neil Maiden. Driving component selection through actor-oriented models and use cases. In *Proceedings of 3rd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 63–73, 2004.

[255] G. Salton, A. Fox, and H. Wu. Extended boolean information retrieval. *Communication of the ACM*, 36 (11) :1022–1036, November 1983.

[256] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill book company, New York, 1983.

[257] Joao Costa Seco and Luis Caires. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 108–128, June 2000.

[258] S. Sedigh-Ali, A. Ghafoor, and R. Paul. Software engineering metrics for cots-based systems. *IEEE Computer*, 34 (5) :44–50, 2001.

[259] Mary Shaw. Beyond objects : A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20 (1) :27–38, January 1995.

[260] Régis Simao and Arnaldo Belchior. Quality characteristics for software components : Hierarchy and quality guides. In LNCS, editor, *Component-Based Software Quality*, pages 184–206. Springer-Verlag, 2003.

[261] H. Simon. A behavioural model of rational choice. In *Models of Man*, pages 241–260. Wiley, New York, 1957.

[262] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) and the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 38–45, September 1986.

[263] R. Solingen and E. Berghout. *The G/QM Method*. McGraw Hill, New York, 1990.

[264] I. Sommerville. *Software Engineering*. Addison-Wesley, 1995.

[265] SourceForge. Website. http ://SourceForge.net, 2005.

[266] J. Spivey. *The Z Notation ; A Reference Manual*. Series in Computer Science, Prentice Hall International, 1989.

[267] Richard Stallman. Emacs, the extensibme, customizable, self-documenting display editor. *ACM SIGOA Newsletter*, 1 (1/2) :147–156, 1981.

[268] I. Stamelos, L. Angelis, P. Dimou, and E. Sakellaris. On the use of vayesian belief networks for the prediction of software productivity. *Information and Software Technology*, 45 :51–60, 2003.

[269] B. Stewart. Predicting project delivery rates using the naive-bayes classifier. *Journal of Software Maintenance and Evolution : Research and Practice*, 14 :161–179, 2002.

[270] Bjarne Stroustrup. What is "object-oriented programming" ? In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 51–70, 1987.

[271] G. Subramanian and M. Gershon. The selection of computer-aided software engineering tools : A multi-criteria decision making approach. *Decision Sciences*, 22 (5) :1109–1123, May 1991.

[272] Clemens Szyperski. Why objects are not enough. In *Proceedings of the 1st International Component Users Conference (CUC96)*, 1996.

[273] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002.

[274] Clemens Szyperski and Wolfgang Weck. Do we need inheritance ? In *Proceedings of the 1st ECOOP Workshop on Component-Oriented Programming (WCOP)*, 1996.

[275] David Taenzer, Murthy Ganti, and Sunil Podar. Problems in object-oriented software reuse. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 25–38, 1989.

[276] S. Talley. Selection and acquisition of administrative microcomputer software. *AEDS Journal*, 17 :69–82, 1983.

[277] C. Tayntor. *Six Sigma Software Development*. Auerbach Publications, 2002.

[278] A. Teltumbde. A framework for evaluating erp projects. *International Journal of Production Research*, 38 (17) :4507–4520, 2000.

[279] S. Thomason, P. Brereton, and S. Lionkman. Clarifi : An architecture for component classification and brokerage. In *Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering (CBSE 7)*, 2000.

[280] Vincent Traas and Jos Van Hillegersberg. The software component market on the internet current status and conditions for growth. *ACM SIGSOFT Software Engineering Notes*, 25 (1) :114–117, January 2000.

[281] V. Tran, D.-B. Liu, and B. Hummel. Component-based systems development : Challenges and lessons learned. In *Proceedings of the 8th IEEE International Workshop on Software Technology and Engineering Practices incorporating Computer Aided Software Engineering*, pages 452–462, Los Alamitos, California, USA, 1997.

[282] Vu Tran and Dar-Biau Liu. A procurement-centric model for engineering component based software engineering. In *Proceedings of the 5th IEEE International Symposium on Assessment of Software tools (SAST)*, Los Alamitos, California, USA, June 1997.

[283] P. Ulkuniemi and V. Seppanen. Cots component acquisition in an emerging market. *IEEE Software*, 21 (6) :76–82, November 2004.

[284] A. van der Hoek, Ebru Dincel, and Nenad Medvidovic. Using service utilization metrics to assess the structure of product line architectures. In *Proceedings of the International Symposium on Software Metrics (METRICS)*, 2003.

[285] D. Vanderpooten and P. Vincke. Description and analysis of some representative interactive multicriteria procedures. *Math. Comput. Model*, 12 :1221–1238, 1989.

[286] P. Vincke. *Multicriteria Decision Aid*. Wiley, New York, 1992.

[287] J. Voas. The challenges of using cots software in component-based development. *IEEE Computer*, 31 (6) :44–45, June 1998.

[288] J. Voas. Cots software - the economical choice ? *IEEE Software*, 15 (3) :16–19, March 1998.

[289] Hironori Washizaki, Hirokazu Yamamoto, and Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. In *Proceedings of the 9th International Symposium on Software Metrics (METRICS'03)*, September 2003.

[290] Peter Wegner and Stanley Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 55–77, 1988.

[291] C. Wei, C. Chien, and M. Wang. An ahp-based approach to erp system selection. *International Journal of Production Economics*, 96 :47–62, 2005.

[292] F. Williams. Appraisal and evaluation of software products. *Journal of Information Science*, 18 (2) :121–125, February 1992.

[293] J. Wood and J. Silver. *Joint Application Development*. John Wiley and Sons, New York, 1995.

[294] Murray Woodside, Dorin Petriu, and Khalid Siddiqui. Performance-related completions for software specifications. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2002.

[295] Ye Yang, Barry Boehm, and Betsy Clark. Assessing cots integration using cost estimation units. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[296] Fan Ye and Tim Kelly. Cots product selection for safety-critical systems. In *Proceedings of 3rd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 53–62, 2004.

[297] Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of 8th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 60–68, San Diego, California, USA, 2000.

[298] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2002.

[299] Hean Chin Yeoh and James Miller. Cots acquisition process : Incorporating business factors in cots vendor evaluation taxonomy. In *Proceedings of the 10th International Symposium on Software Metrics (METRICS)*, 2004.

[300] E. Yu. Modeling organisations for information systems requirements engineering. In *Proceedings of the 1st IEEE International Symposium on Requirements Engineering (ISRE)*, pages 34–41. IEEE Computer Society Press, 1993.

[301] Amy Zaremski and Jeannette Wing. Signature matching : a tool for using software libraries. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 4 (2) :146–170, April 1995.

[302] Amy Zaremski and Jeannette Wing. Specification matching of software components. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 6 (4) :333–369, October 1997.

[303] Z. Zhang, L. Svensson, U. Snis, C. Srensen, H. Fgerlind, T. Lindroth, M. Magnusson, and C. Stlund. Enhancing component reuse using search techniques. In *Proceedings of IRIS 23*, Laboratorium for Interaction Technology, University of Trollhttan Uddevalla, 2000.