THÈSE

présentée à

L'UNIVERSITÉ d'ORLÉANS

pour obtenir le grade de

DOCTEUR

Spécialité : **INFORMATIQUE**

par

Patrick PAROT

Sujet de la thèse :

MÉCANISATION DE LA RÉUTILISATION DE COMPOSANTS LOGICIELS : approches et outils

Soutenue le 20 Octobre 1995 devant le jury composé de :

Mr.	P.	DERANSART	Président
Mme	V.	DONZEAU-GOUGE	Rapporteurs
M.	J.J.	CHABRIER	
MM.	B.	COULANGE	Examinateurs
	R.	DA SILVA BIGONHA	
	F.	LE BERRE	
	F.	ROUAIX	

À mon père,

Les sciences ont les racines amères,
mais leurs fruits sont très doux.
Aristote

Avant-propos

Ce travail commencé en France au sein de l'INRIA Rocquencourt et terminé à l'Université Fédérale du Minas Gerais (UFMG) au Brésil, aura été pour moi une source de connaissances et de découvertes permanentes riches en émotions. Ainsi, je suis heureux d'exprimer toute ma gratitude à celles et ceux qui ont contribué par leur soutien à la naissance de ce mémoire.

En premier lieu, je remercie tout particulièrement **François Rouaix** pour l'aide très précieuse et l'attention qu'il m'a généreusement fournies tout au long de ce travail.

Je tiens à témoigner toute ma reconnaissance envers **Pierre Deransart** qui a accepté la présidence de mon jury de thèse, et grâce à qui cette expérience s'est poursuivie sous les tropiques dans d'excellentes conditions.

Je remercie très vivement **Véronique Donzeau-Gouge** et **Jean-Jacques Chabrier** qui m'ont fait le grand honneur d'accepter de rapporter cette thèse et dont les précieux commentaires m'auront été d'une grande utilité.

Un grand merci à **Roberto et Mariza Bigonha** pour leur gentillesse, leur accueil chaleureux, leur compréhension et leur dévouement à mon égard. Je pense également à **José de Siqueira** que je remercie pour m'avoir soutenu dans certains moments difficiles.

Merci à **Josy Baron**, **Ghislaine Le Corre** et **Sylvie Loubressac** qui m'ont permis de résoudre, par leur gentillesse illimitée et leur disponibilité, quelques problèmes fâcheux d'organisation.

Enfin, je remercie mon professeur de portugais, **Estela Maura Silva de Castilho**, et la félicite pour la qualité de son enseignement.

Ce travail a été partiellement financé par le projet Eureka **SCI** (***S**oftware **C**omponents for the **I**ndustry*).

Résumé

Les développements de systèmes d'information complexes ont montré la nécessité de concevoir et d'écrire des programmes ou composants logiciels réutilisables. Les systèmes de gestion de bibliothèques logicielles (SGBL) sont des supports destinés à mécaniser la réutilisation de composants au travers d'une infrastructure de bibliothèque. Celle-ci doit établir formellement les mécanismes de composition et de recherche des éléments traités. Les composants sont décrits par un ensemble de facettes exprimant chacune une classe de spécifications dans une sémantique particulière. La recherche est l'opération qui consiste à trouver des composants dont la spécification est filtrée par une requête donnée. La composition, exprimable par la paramétrisation des composants, est une opération de création de nouveaux éléments. Celle-ci est définie à partir d'une relation d'ordre entre les spécifications propres à une facette déterminée. Ainsi, un langage de spécifications, une relation de filtrage et une relation d'ordre seront définis pour chaque facette. Les propriétés de correction et de complétude de la recherche mise en œuvre devront être prouvées.

Nous suggérons une approche syntaxique où les spécifications des composants sont formées à partir de termes du premier ordre. Nous présentons un modèle déclaratif permettant d'automatiser la recherche dans une bibliothèque où peuvent être définis des modules de programmes et des opérateurs du premier ordre (foncteurs) sur ces modules. En considérant les spécifications de foncteurs comme des règles de déduction, une solution à une requête peut être obtenue par l'application d'un foncteur à un résultat calculé par un moteur d'inférences. La conception précédente doit être étendue pour traiter les spécifications de foncteurs de tout ordre : la solution proposée repose sur une notion de modèle syntaxique qui exprime une sémantique des spécifications traitées.

Un système prototype a été développé et est utilisé comme support des expérimentations présentées. Une application de nos travaux en configuration automatique de systèmes par synthèse de programmes a été réalisée. Les résultats expérimentaux obtenus encouragent une étude plus approfondie dans ce domaine.

Mots clés

Réutilisation, Composants Logiciels, Modules de Programmes, Foncteurs d'Ordre Supérieur, Facettes de Spécifications, Interfaces, Configuration de Systèmes, Programmation Logique.

Abstract

Developments of complex information systems have emphasized the need to design and to write reusable programs or software components. Software libraries management systems are designed to achieve mechanization of components reuse through a library infrastructure. This later must formally establish the retrieval and composition mechanisms for the processed items. These items are described by a set of facets, each of them expresses a specification in a precise semantic. The searching process implements a component retrieval operation by specification matching. Component composition, which can be expressed by component parameterization, is an operation which generates new items. Composition will be defined from an order relation between specifications relative to a specific facet. Thus, a specification language, a matching relation and an order relation have to be defined for each processed facet. The soundness and completness of the implemented searching process have to be proved.

We suggest a syntactical approach where component specifications are built from first order terms. We present a declarative model to enhance mechanization of reuse in a software library where modules of programs and first order operators (functors) upon these modules can be defined. Considering functor specifications as deduction rules, a solution to a query may be obtained by applying a functor to a result computed by an inference engine. The previous design has to be extended to process higher order functors : the solution we propose is based upon a notion of syntactical model which expresses a semantic of the processed specifications.

A prototype system has been developed and is used as a support for the experimentations we present. We have applied the previous notions in the area of software configuration management. The obtained results encourage us to go deeper into the applications of our work in this domain.

Keywords

Software Reuse, Software Components, Modules of Programs, Higher Order Functors, Facets of Specifications, Interfaces, System Configuration, Logic Programming.

Chapitre 1

Introduction

1.1 La réutilisation en génie logiciel : un enjeu scientifique et économique

Les entreprises ont toujours été à l'affût de techniques nouvelles leur permettant d'augmenter leur productivité tout en réduisant les prix de revient et en assurant une qualité optimale des produits et services fournis : trois facteurs essentiels pour maintenir une compétitivité sur les marchés économiques. Depuis quelques années, c'est l'industrie informatique, et plus particulièrement du logiciel, qui se distingue par une évolution rapide de ces paramètres. Le logiciel, par nature reproductible quasiment instantanément, s'avère adéquat à la réutilisation systématique qui représente un potentiel scientifique et économique primordial pour atteindre de tels objectifs.

De nombreux efforts ont été faits pour perfectionner les techniques de réutilisation logicielle dont les champs d'applications ne sont d'ailleurs pas restreints à la réutilisation spécifique de code : les conceptions, les spécifications, les architectures, les documentations et toutes autres formes de connaissances relatives à un système d'information sont potentiellement réutilisables. Certaines recherches actuelles s'orientent vers la conception et la réalisation de supports sous-jacents à une infrastructure de bibliothèque permettant d'automatiser la recherche de composants logiciels validés : l'objectif est de systématiser la réutilisation considérée comme fondamentale dans le processus de développement des systèmes d'information.

On se référera à [BDRV91] [ML91] et [WESZ86] pour des discussions et analyses des impacts économiques liés à la réutilisation logicielle.

1.1.1 Le génie logiciel : un ensemble de méthodologies

La complexité des systèmes d'information est en perpétuelle croissance. La constatation des difficultés inhérentes à leur fabrication a favorisé l'émergence d'une discipline à part entière :

le génie logiciel. L'activité propre au génie logiciel se focalise sur la reconnaissance et le contrôle des différentes étapes du développement et de la maintenance des systèmes. Un ensemble de méthodologies ont été élaborées à cette fin.

Étapes du développement et de la maintenance des systèmes d'information

On distingue communément, six principales étapes dans le processus de développement et de maintenance d'un système d'information :

1. **Analyses des besoins** : expression et définition des besoins, attentes et exigences en termes de ressources et de fonctionnalités.
2. **Spécifications** : élaboration du cahier des charges où sont regroupées les définitions, descriptions et formalisations du comportement et des caractéristiques propres au système à construire. Les spécifications définissent ce que le système fera. La tendance actuelle favorise l'aspect formel des spécifications afin de s'assurer, au plus tôt, de la cohérence et de la correction de ce qui est écrit dans le cahier des charges.
3. **Conception** : développement de la structure du système. Le projet est découpé en sous-parties indépendantes. Les algorithmes et éventuelles contraintes de performances sont définis. La solution est élaborée.
4. **Implantation** : codage du système dans un langage de programmation. Les langages de haut niveau sont généralement couplés à un environnement fournissant des outils logiciels d'aide à la programmation.
5. **Validation** : exécution d'un ensemble de tests regroupés en trois parties : validation des modules qui composent le système, validation de l'intégration des modules et validation du système dans son ensemble. Des systèmes de preuves de programmes peuvent être utilisés à ce niveau.
6. **Maintenance** : tout système est sujet à des évolutions et adaptations. On distingue communément deux formes de maintenance :
   - la maintenance **corrective** qui consiste à corriger les éventuelles anomalies et dysfonctionnements du système,
   - la maintenance **adaptative** qui a pour but d'étendre et/ou de modifier certaines fonctionnalités du système.

   Cette phase, la dernière du cycle de vie d'un logiciel, est celle qui exige généralement le plus d'efforts et représente le coût le plus élevé.

Le génie logiciel est interdisciplinaire : utilisation de l'ingénierie pour établir et concevoir des projets puis organiser la coordination et le contrôle de la réalisation, des mathématiques pour analyser et certifier les algorithmes, de la gestion pour évaluer les coûts, mesurer les risques et maîtriser les temps de mise sur le marché.

Méthodologies pour le développement des logiciels

De nombreuses méthodologies, fonctions des types d'applications visés, ont été proposées. Nous en citerons deux à titre d'illustration :

PSL (*Problem Statement Language*) est un langage conçu pour exprimer des exigences fonctionnelles et de performances. Il est couplé à un processeur, appelé PSA, qui produit une base de données des spécifications du système à construire, et effectue des vérifications de la cohérence des analyses conceptuelles. Ce système, adapté à la construction de grandes applications orientées gestion, a été développé à l'université de Michigan dans le cadre du projet *ISDOS* [TH77].

SADT (*Structured Analysis and Design Technique*) est un outil graphique qui automatise une méthodologie d'analyse et de conception des systèmes d'information orientés gestion. On se référera à [RS77] pour une présentation complète de la méthode.

Le lecteur pourra également se référer à [Lon87] pour une analyse relative à l'évaluation des coûts de développement des systèmes d'information modernes.

1.1.2 Le problème de la configuration des grands systèmes

Les programmes de grande taille, sous-jacents aux systèmes d'information complexes, surpassent les capacités de quiconque à pouvoir les appréhender, les comprendre et les maintenir dans leur totalité. Ils doivent donc être découpés en entités modulaires autonomes. Un tel découpage exige beaucoup d'expérience [Par72]. En principe toute partie d'un programme qui peut être potentiellement réutilisable dans un autre contexte forme un module de programme. Les structures de données définies sous forme de types abstraits de données, définissent également des modules de programme [Gut80] [Lis87].

Le contrôle des interconnexions et des combinaisons des éléments qui forment la configuration d'un programme est un autre aspect fondamental du processus de développement des grandes applications. Un contrôle rigoureux de ces interactions favorise la compréhension de la structure globale du programme : notion de *programming in the large* [RS91] en opposition avec la notion de *programming in the small* [dRK75] qui traite des détails d'implantation des algorithmes en termes de structures de données et de structures de contrôle.

Des outils en configuration de systèmes (SCM : *Software Configuration Management systems*) fournissent une assistance automatisée pour concevoir, construire et maintenir les gros logiciels. Les techniques de SCM peuvent être vues comme une extension de la notion de *programming in the large*. Ces outils se focalisent notamment sur certains aspects présumés invariants des systèmes, et encouragent l'emploi de constructions abstraites dans les programmes.

La construction correcte d'un logiciel exige que les composants adéquats soient sélectionnés, et que des outils de dérivation leur soient appliqués : un modèle de système est à la fois *descriptif* (il documente les composants du système) et *prescriptif* (il guide le processus de dérivation de nouveaux composants obtenus par compositions).

Enfin, nous n'omettrons pas de mentionner l'importance du typage bien mis en valeur dans les langages dit fortement typés. Ceux-ci garantissent qu'un programme correctement typé s'exécutera sans erreur de type [Car89] : une forme de vérification automatique de la cohérence sémantique des programmes particulièrement appréciable pour le développement et la maintenance de gros programmes.

1.2 L'expressivité des langages de programmation majore la réutilisabilité

L'expressivité des langages de programmation de haut niveau avantage l'écriture de programmes où sont exprimés des concepts abstraits. De tels langages prennent en compte les problèmes de structuration des programmes par des constructions linguistiques telles les modules, les objets, les *packages* d'ADA [Dep83], les *clusters* de CLU [Lis81], les *forms* de ALPHARD [Sha81], etc... De plus, ces constructions modulaires supportent des formes d'abstraction basées sur le masquage d'information.

1.2.1 Abstraction : condition nécessaire de la réutilisation

Le terme *abstraction* se réfère aux constructions linguistiques (modularité, polymorphisme, fonctionnalité, héritage) permettant l'expression des algorithmes indépendamment de toute représentation des données en machine. L'abstraction des programmes est un des éléments essentiels pour, d'une part contrôler la complexité croissante des systèmes, et d'autre part écrire des composants logiciels réutilisables. Un programme est dit factorisé si les traitements sont abstraits par rapport aux représentations des structures de données sur lesquelles ils s'appliquent.

Principales formes d'abstraction :

- **La modularité** : Simula-67 [DN66], CLU [Lis81] et Modula-2 [Wir83] sont parmi les premiers langages à avoir proposé la modularité comme principe majeur de structuration des programmes. Des interfaces définissent le protocole d'accès et d'utilisation des modules. Elles sont formées par la déclaration d'un ensemble d'opérations typées primitives au module. Une interface peut être spécifiée et compilée indépendamment de son implantation. Les modules, en Modula-2, ne sont cependant pas des valeurs de première classe. La construction permettant de séparer le protocole d'un objet (*i.e.* sa représentation extérieure accessible aux autres objets) de son implantation est appelée

encapsulation. L'encapsulation d'une structure de données dans un type défini par une interface la rend réutilisable dans d'autres contextes. On accède à un module par les opérations (méthodes) spécifiées dans son interface, son implantation restant masquée. Ainsi, les modifications de la représentation interne d'un module n'ont aucune incidence sur les autres composantes du programme. Toute implantation pourra donc être réutilisée par différents programmes.

- **Le polymorphisme** : construction propre à certains langages qui permet de traiter des valeurs dont le type n'est pas unique (par exemple la constante *nil* des listes). Une fonction polymorphe pourra être appliquée avec des arguments de différents types (par exemple la fonction *length* qui calcule la longueur d'une liste pourra être appliquée sur n'importe quel type de liste). La surcharge permet de lier différentes valeurs à un même symbole ; la surcharge peut être vue comme une construction "instance" du polymorphisme dans le sens où un programme écrit en utilisant des symboles surchargés est générique. Par conséquent, un programme surchargé ou polymorphe pourra être exécuté dans différents contextes, c'est-à-dire en choisissant différentes représentations des structures de données.

- **La fonctionnalité** : construction permettant de traiter les fonctions comme les autres valeurs d'un programme. Puisque les fonctions sont des objets du langage, celles-ci peuvent être passées en arguments à d'autres fonctions appelées fonctionnelles, ou être retournées comme résultat d'une fonctionnelle. Ces dernières sont des fonctions d'ordre supérieur. Par exemple, une fonction réalisant le tri d'un ensemble d'éléments peut spécifier un paramètre fonctionnel qui exprime la relation d'ordre sur ces éléments. De telles constructions avantagent l'expression factorisée des algorithmes qui majore l'abstraction des traitements par rapport aux données sur lesquelles ils s'appliquent.

- **L'héritage** : construction caractéristique des langages de programmation orientés objets, l'héritage permet d'étendre les fonctionnalités d'un composant logiciel en héritant de tout ou partie des fonctionnalités d'un autre composant. Aucune duplication explicite de code n'est réalisée par l'utilisateur. L'extension par héritage est donc typique d'une réutilisation logicielle.

On se référera à [Kli86] pour une discussion détaillée relative aux concepts de modularité et de réutilisation tels qu'ils apparaissent dans quelques langages de programmation usuels.

1.2.2 Les paradigmes de programmation et la réutilisation

Nous présentons succinctement trois grandes familles de langages de programmation qui correspondent à trois paradigmes distincts de programmation : algorithmique, orienté objets, fonctionnel.

- **Langages algorithmiques** (C, Pascal, ...) : ce sont des langages typés exclusivement statiquement ce qui inhibe la flexibilité nécessaire aux techniques de réutilisation. Ils autorisent peu le polymorphisme des données ce qui restreint considérablement leur puissance d'expressivité.

- **Langages orientés objets** (Smalltalk [GR83], Eiffel [MNM87], ...) : ils fournissent un typage dynamique qui accroît leur flexibilité. Les langages orientés objets favorisent une approche *bottom-up* du développement des systèmes : définitions des objets (données) du programme, puis définitions des fonctions qui leur sont appliquées. De plus, ces langages définissent les notions d'encapsulation et d'héritage de données. Ils seraient donc, par essence, adéquats à intégrer les techniques de réutilisation logicielle [Ree93] [Che93].

- **Langages fonctionnels** (ML, Caml, ...) : un langage est dit fonctionnel lorsque les fonctions sont des valeurs de première classe, c'est-à-dire pouvant être traitées comme les autres données d'un programme. Certaines propositions de langages, dont Quest [CL90], offrent également la possibilité d'utiliser les types comme des données à part entière ce qui accroît leur puissance d'expression. Les langages fonctionnels favorisent une approche *top-down* du développement des systèmes : définitions des fonctions puis des données sur lesquelles elles s'appliquent. ML a introduit la notion de polymorphisme dit paramétrique [Mil78] (relative à la structure même des données) dans les langages ; les types ML peuvent contenir des variables qui seront instanciées dans différents contextes. Il devient possible d'écrire des programmes basés sur des données partiellement typées. Le polymorphisme des données avantage la réutilisation de composants logiciels.

1.2.3 Environnements de programmation

Les langages modernes sont, dans la plupart des cas, couplés à des environnements qui assistent l'activité du programmeur en mettant à sa disposition un ensemble d'outils logiciels d'aide à la programmation : éditeurs de textes, outils de mise au point des programmes, bases de données, bibliothèques de composants, etc ... Les **Systèmes de Gestion de Bibliothèques Logicielles (SGBL)** sont des supports destinés à mécaniser la réutilisation de composants logiciels au travers d'une infrastructure de bibliothèque. L'intégration de tels outils dans les environnements de programmation avantage une approche *bottom-up* du processus de développement des systèmes par réutilisation systématique de composants logiciels prédéfinis et validés.

1.3 Organisation de ce mémoire

Après avoir présenté le problème lié aux limitations des techniques actuelles de réutilisation, nous introduirons l'approche et la démarche que nous proposons pour mécaniser la recherche

de composants logiciels. Nous définirons ensuite les principaux concepts de base utilisés dans notre étude ainsi qu'un modèle abstrait d'infrastructure de bibliothèque qui établit les relations entre ces concepts (chapitre 2).

Nous développerons, dans le chapitre 3, la première partie de nos travaux relatifs à l'automatisation d'une bibliothèque où peuvent être définis des modules de programmes et des opérateurs du premier ordre (foncteurs) sur ces modules. Le modèle d'infrastructure retenu repose sur une approche entièrement déclarative dont les propriétés permettent de mettre en œuvre une technologie qui utilise un système de programmation Logique du premier ordre. En considérant les spécifications des foncteurs comme des règles de déduction, une solution à une requête peut être obtenue par l'application de foncteurs à un résultat calculé par un moteur d'inférences.

Cependant, l'approche précédente doit être étendue dans sa conception pour traiter les spécifications de foncteurs de tout ordre: la solution que nous proposons, présentée dans le chapitre 4, repose sur une notion de modèle syntaxique exprimant une sémantique des spécifications traitées.

Nous consacrerons le chapitre 5 à la présentation de notre système prototype ainsi qu'aux expérimentations pour lesquelles il a servi de support. Nous développerons plus amplement une expérience où nos techniques sont appliquées à la configuration automatique de systèmes par synthèse de programmes.

Le chapitre 6 propose un tour d'horizon des différentes approches relatives à la conception des modèles et des méthodes de réutilisation rencontrées dans la littérature.

En conclusion (chapitre 7 et 8) nous suggérerons des suites à donner à cette étude et envisagerons les divers champs d'applications qui s'ouvrent à nos techniques.