Employing Genericity and Case-Based Reasoning to Effectively Reuse Code

Panagiotis K. Katalagarianos

Doctoral Dissertation

Heraklion, Crete
February 1994
Employing Genericity and Case-Based Reasoning to Effectively Reuse Code

Panagiotis K. Katalagarianos

Doctoral Dissertation

Department of Computer Science
University of Crete

February 1994
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εφαρμογή της Γενίκευσης και Περιπτωσιολογικών Συλλογισμών για την Αποτελεσματική Αναχρησιμοποίηση Κώδικα

Διατριβή που υποβλήθηκε από τον
Παναγιώτη Κ. Καταλαγκαριάνο
ως μερική απαίτηση για την απόκτησή του
Διδακτορικού Διπλώματος

Ηράκλειο, Φεβρουάριος 1994

Συγγραφέας:

Εξεταστική Επιτροπή:

Καθηγητής Γιάννης Βασιλείου, Εθνικό Μετσόβιο Πολυτεχνείο, Επόπτης (Πανεπιστήμιο Κρήτης έως Ιούνιο 1993)

Αναπληρωτής Καθηγητής Πάνος Κωνσταντόπουλος, Πανεπιστήμιο Κρήτης

Αναπληρωτής Καθηγητής Μανόλης Κατεβαίνης, Πανεπιστήμιο Κρήτης

23/3/94

Αναπληρωτής Καθηγητής Χρήστος Νικολάου, Πανεπιστήμιο Κρήτης

Αναπληρωτής Καθηγητής Χρήστος Νικολάου, Πανεπιστήμιο Κρήτης

Καθηγητής Στέλιος Ορφανούδακης, Πανεπιστήμιο Κρήτης

Αναπληρωτής Καθηγητής Τιμός Σελλής, Εθνικό Μετσόβιο Πολυτεχνείο

6 Δεκεμβρίου του 1994

Καθηγητής Μανόλης Σκορδαλάκης, Εθνικό Μετσόβιο Πολυτεχνείο

Δεκτή:

Αναπληρωτής Καθηγητής Πάνος Κωνσταντόπουλος, Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών
Employing Genericity and Case-Based Reasoning
to Effectively Reuse Code

Panagiotis K. Katalagarianos

Doctoral Dissertation

Department of Computer Science
University of Crete

Abstract

Systematic reuse of code has been proposed as a promising means to address the legendary productivity increase in software development. While object-oriented programming languages are, by nature, well suited for reusability-based development of applications, additional mechanisms to effectively reuse code are necessary.

Effective reuse of code requires a rich collection of designed-for-reuse software components and knowledge on how to locate them in a repository, adapt them if needed, and even create new ones based on information provided by other components exploiting similar characteristics.

The application of techniques and methods from artificial intelligence to software engineering is one mechanism through which reusability of software might be achieved. By abstracting and encoding the expertise of experienced software engineers into knowledge bases together with software components, system developers can gain effective access to the artifacts in the software repository as it evolves over time.

Under this perspective, a novel method is presented. The method uses genericity (a technique that allows a module to be defined with parameterized types), and employs a special form of Case-Based Reasoning (a method of solving problems based on the transfer of past experience to new problem situations) in order to make object-oriented code reusable.

Using established correspondences (links) and experiences from previous situations, the system semi-automatically finds a good match (e.g., a class) in the repository to the user’s requirements, possibly adapting it to specific needs. Adaptation is based on specifying type parameters while making the necessary method modifications. This process is also reversible (generic classes from specific ones). Finally, the repository is enriched with the new component(s) and process
knowledge, followed by an automatic appropriate reorganization.

The method presented in this thesis has been evaluated through a prototype implementation, which addresses the reuse of C++ code. The prototype system runs on SparcStations and Sun 4 series under Unix. The language used for the implementation is a Prolog-like language called MegaLog. Additionally, a usage experiment performed in order to get an indication on the usage characteristics of the prototype system.

The proposed method integrates ideas and techniques from various fields like knowledge representation, software engineering and machine learning. It addresses only technical issues for the effective reuse problem. To be practical, reuse must address not only technical but managerial, economic, legal, cultural, and technology transfer issues.

In considering the application of the method in practical situations several operational characteristics need to be resolved, like synonym handling, the provision of an undo operation, etc.

Future research could include a study on the extensibility of our method for reusing applications, not just code. Concerning the retrieval of similar past cases and their adaptation, additional types of similarity should be explored. Semantic similarity is a good candidate. Furthermore, a quality assurance study is necessary. The entire software production cycle has to be integrated in such a way as to provide a strong basis for controlled evolution and expansion of the repository with new high quality applications.

Advisor: Professor Yannis Vassiliou, National Technical University of Athens, (University of Crete till June 1993)
Εφαρμογή της Γενίκευσης και Περιπτωσιολογικών Συλλογισμών για την Αποτελεσματική Αναχρησιμοποίηση Κώδικα

Παναγιώτης Κ. Καταλαγαριανός

Διδακτορική Διατριβή

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Εκτενής Περίληψη

Οι μεγάλοι οργανισμοί ανάπτυξης λογισμικού αναζητούν τα τελευταία χρόνια νέες μεθοδολογίες και εργαλεία με στόχο την αύξηση της παραγωγικότητας του επιστημονικού τους προσωπικού, τη μείωση του υψηλού κόστους παραγωγής και την εξασφάλιση της ποιότητας και της αξιοπιστίας των παραγόμενων συστημάτων. Από τις προτεινόμενες λύσεις προς αυτή την κατεύθυνση, τις περισσότερες υποσχέσεις δίνει η συστηματική αναχρησιμοποίηση λογισμικού.

Το πρώτο πρόβλημα που εμφανίζεται στην προσπάθεια συστηματικής αναχρησιμοποίησης λογισμικού σχετίζεται με τη "φόση" των όντων που πρόκειται να αναχρησιμοποιηθούν. Υπάρχουν προγράμματα, τμήματα προγραμμάτων, καθορισμοί, απαιτήσεις και πλάνα ανάπτυξης τα οποία αλληλοσχετίζονται. Η αναχρησιμοποίηση κάθε όντος συνεπάγεται την ταυτόχρονη αναχρησιμοποίηση των όντων που σχετίζονται με αυτό. Συνεπώς, οδηγούμαστε στην αναχρησιμοποίηση περισσότερων στοιχείων από απλό κώδικα. Τα όντα λογισμικού και οι συσχέτισες τους εμπεριέχουν ένα μεγάλο ποσό εμπειρίας. Η εμπειρία αυτή είναι απαραίτητη να αναχρησιμοποιηθεί στην παραγωγή νέου λογισμικού.

Πριν αναχρησιμοποιηθεί οποιοδήποτε, πρέπει πρώτα να καθοριστεί και να δημιουργηθεί. Στη συνέχεια πρέπει να περιγραφεί και να αποθηκευτεί σε ένα είδος βιβλιοθήκης (ταμειώνας). Η φάση αυτή εμπεριέχει προβλήματα που σχετίζονται με το μετασχηματισμό της αναχρησιμοποιήσης πληροφορίας σε συνιστώσες. Ποιό είναι το κατάλληλο μέγεθος των συνιστώσων και ποιος ταξινομούνται; Ποιες είναι οι κατάλληλες τεχνικές και γλώσσες για την περιγραφή των συνιστώσων;

Ο χρήστης του ταμειώνα δεν έχει πάντα πλήρη γνώση των συνιστώσων που είναι αποθηκευμένες σε αυτόν και πιθανόν δε να γνωρίζει τι ακριβώς αναζητά. Αυτή η άγνωση επάγγειλε το πρόβλημα της "επιλογής". Το πρόβλημα αυτό γίνεται ακόμα δυσκολότερο όταν συνδυάζεται με το πρόβλημα της "προσαρμογής". Σε πολλές περιπτώσεις δεν είναι δυνατό να βρεθεί στον ταμειώνα μία συνιστώσα που να ικανοποιεί πλήρως τις ανάγκες του χρήστη. Όμως, είναι
πιθανό να υπάρχει κάποια συνιστώσα η οποία με μικρές τροποποιήσεις να μπορέσει να
ικανοποιήσει τελικά τις αρχικές ανάγκες. Το πρόβλημα έγειρε στην επιτυχή αναζήτηση αυτής
της συνιστώσας, στην κατανόηση της και στην εφαρμογή των κατάλληλων τροποποιήσεων.

Τέλος, ένα σημαντικό πρόβλημα το οποίο επηρεάζει άμεσα τα προβλήματα επιλογής και
προσαρμογής είναι το λεγόμενο πρόβλημα της "εξέλιξης". Ο κύκλος ανάπτυξης λογισμικού
πρέπει να οργανωθεί κατά τέτοιο τρόπο ώστε να βελτιώνεται η ποιότητα και η αποδοτικότητα
tου ταμειοτήρα με την παροδο του χρόνου.

Στη διατριβή αυτή παρουσιάζεται μια νέα προσέγγιση για την αναχρησμοποίηση
οντοκεντρικού κώδικα, στην οποία χρησιμοποιούνται περιφερειακοί συλλογισμοί σε
συνδυασμό με την τεχνική της γενίκευσης. Με αυτόν τον τρόπο παρέχεται στο χρήστη του
ταμειοτήρα η δυνατότητα να εντοπίσει τον κατάλληλο κώδικα με έναν ημιαυτόματο τρόπο, καθώς και να τροποποιήσει τον κώδικα αυτό σύμφωνα με τις ανάγκες του. Παράλληλα,
pαρέχεται η δυνατότητα εξέλιξης του ταμειοτήρα με εισαγωγή νέων συνιστώσων και με
εφαρμογή των κατάλληλων αναδιοργάνωσεων.

Η τεχνική της γενίκευσης μιας επιτίθεται να ορίσουμε ένα τμήμα οντοκεντρικού κώδικα με
παραμετροποιημένους τύπους. Ως παραμετροποιημένος θεωρείται ένας τύπος που έχει οριστεί με
βάση έναν άλλο ακαθοριστό τύπο. Διαφορετικές εκδοχές του παραμετροποιημένου κώδικα
dημιουργούνται για συγκεκριμένους τύπους της παραμέτρου.

Μια γενική μορφή παραμετροποίησης μπορεί εύκολα να ολοκληρωθεί σε οντοκεντρικές
gλώσσες οι οποίες δεν παρέχουν από κατασκευής γενίκευσεις. Όμως οι γενίκευσεις από τη φύση
τους δεν είναι αρκετά ευέλικτες. Αυτό συμβαίνει γιατί υπάρχουν δύο επίπεδα τυπικά κώδικα: i) γενίκευμα τμήματα τα οποία είναι παραμετροποιημένα, άρα και ανοιχτά σε διαφορετικές
eκδοχές, χωρίς όμως να μπορούν να χρησιμοποιηθούν άμεσα και ii) ειδικευμένα τμήματα τα
οποία μπορούν να χρησιμοποιηθούν άμεσα αλλά δεν επιδείχνονται βελτιώσεις.

Επιπλέον, σε ένα ταμειοτήρα με γενίκευμενες συνιστώσεις δεν εξαλείφονται τα προβλήματα
αναχρησμοποίησης λογισμικού που αναφέρθηκαν παραπάνω. Παρόλο που ο χώρος
αναζήτησης μειώνεται (αποθηκεύεται μόνο μία γενικευμένη συνιστώσα αντί για αρκετές
ειδικευμένες), αυτή η μείωση δεν είναι δραστική. Επίσης, η προσαρμογή των συνιστώσων και η
αναδιοργάνωση του ταμειοτήρα ανήκουν αποκλειστικά στη δικαιοδοσία του χρήστη.

Για την αναχρησμοποίηση οντοκεντρικού κώδικα με γενικέυσεις, θα μπορούσαμε να
βασιστούμε σε εμπειρική γνώση που έχει συγκεντρωθεί από παρόμοιες αναπτύξεις λογισμικού
του παρελθόντος. Κατά συνέπεια μπορούν να χρησιμοποιηθούν μέθοδοι Τεχνητής Νοημοσύνης
με σκοπό της παροχής βοήθειας από το σύστημα ώστε i) να κωδικοποιηθεί αυτή η γνώση και ii)
να αναχρησμοποιηθεί.
Οι περιπτωσιολογικοί συλλογισμοί είναι μία μέθοδος επίλυσης προβλημάτων η οποία
βασίζεται στη μεταφορά παλαιότερης εμπειρίας για την επίλυση νέων προβλημάτων. Η μέθοδος
αυτή εμφανίζει αρκετά πλεονεκτήματα. Πρώτο, επιτυγχάνεται βελτίωση της απόδοσης του
συστήματος καθώς παρέχει: i) ελαχιστοποίηση του χρόνου που απαιτείται για συλλογισμού, ii)
δυνατότητα αποφυγής λαθών που έχουν στο παρελθόν και iii) άμεση εστίαση στα πιο
σημαντικά σημεία του προβλήματος. Δεύτερον, η μάθηση είναι αρκετά εύκολη καθώς δεν
απαιτείται ένα αυτονόμο μοντέλο ή βαθιά κατανόηση του πεδίου εφαρμογής. Τρίτον, η χρήση
γενικευμένων περιπτώσεων μπορεί να λειτουργήσει και σα μηχανισμός παροχής επεξεργασιών.

Ένα πρόβλημα που εμφανίζεται όταν προσπαθούμε να χρησιμοποιήσουμε
περιπτωσιολογικούς συλλογισμούς σε ένα περιβάλλον ανάπτυξης λογισμικού με
αναχρησιμοποίηση, είναι η κωδικοποίηση των συνιστώσων λογισμικού καθώς και της εμπειρίας
ανάπτυξης συστημάτων κατά τέτοιο τρόπο ώστε να δημιουργούνται περιπτώσεις.

Σε αυτή τη διατριβή χρησιμοποιούμε τον όρο “συνιστώσα λογισμικού” για να
αναφερθούμε σε τμήματα ενός συστήματος λογισμικού από όλες τις φάσεις του κύκλου
ανάπτυξης του. Ένα σημαντικό θέμα που μας απασχόλησε, ήταν ο προσδιορισμός των εννοιών
και των σχέσεων του πεδίου προγραμματισμού οι οποίες μπορούν να χρησιμοποιηθούν για να
μοντελοποιήσουμε την προγραμματιστική γνώση. Αυτές οι εννοιες και οι σχέσεις μετατράπηκαν
σε περιγραφές συνιστώσων και σε διασυνδέσεις με χρήση ενός σφαίρετικου μηχανισμού και με
την εφαρμογή των κατάλληλων εξειδικεύσεων.

Για να μοντελοποιήσουμε αυτές τις συνιστώσες, επεκτείναμε και τροποποιήσαμε το
πλαίσιο των Γλωσσών Διασυνδέσεως Συνιστώσων (ΓΔΣ). Επιπλέον, χρησιμοποιήσαμε τη γλώσσα
Telos (μια γλώσσα αναπαράστασης γνώσης που εμφανίζει όλα τα χαρακτηριστικά των ΓΔΣ) με
σκοπό την περιγραφή του μοντέλου μας με ένα πιο αυτονόμο και τυπικό τρόπο.

Η άλλη οργάνωσή έγινε κατά τέτοιο τρόπο ώστε να ικανοποιηθούν οι ακόλουθες
απαιτήσεις: Πρώτον, οι συνιστώσες λογισμικού περιγράφονται και διασυνδέονται με ένα τρόπο
που επιτρέπει τη δημιουργία περιπτώσεων. Αυτές οι περιπτώσεις μπορούν να χρησιμοποιηθούν
από το σύστημα περιπτωσιολογικών συλλογισμών που έχουμε αναπτύξει. Δεύτερον, οι
περιπτώσεις κατηγορημοποιούνται με ένα αποδοτικό τρόπο, χρησιμοποιούντας ένα σύνολο δεικτών
οι οποίοι εξάγονται εύκολα από τις περιγραφές των συνιστώσων. Τρίτον, η άλλη οργάνωση
παρέχει μια ισχυρή βάση για ελεγχόμενη εξέλιξη και επέκταση του τομέα.

Κάθε συνιστώσα παρέχει ένα σύνολο πόρων και συνδέεται με άλλες συνιστώσες μέσω ενός
συνόλου διασυνδέσεων. Οι βασικές συνιστώσες που χρησιμοποιούνται στην προτεινόμενη
οργάνωση είναι οι εξής:
• Συνιστώσες Υλοποίησης. Χρησιμοποιούνται για την περιγραφή μιας οντοκεντρικής κλάσης και παρέχουν σαν πόρους i) το όνομα του αρχείου όπου είναι αποθηκευμένη η κλάση, ii) τη γλώσσα προγραμματισμού που έχει χρησιμοποιηθεί, iii) τα επιμέρους τμήματα από τα οποία αποτελείται η κλάση, και iv) τις μεθόδους που παρέχει η κλάση.

• Συνιστώσες Σχεδίασης. Χρησιμοποιούνται για την περιγραφή της σχεδίασης μιας οντοκεντρικής κλάσης και παρέχουν σαν πόρους i) τους αφηρημένους τύπους δεδομένων που υλοποιούνται από την αντίστοιχη κλάση και ii) τις λειτουργίες που επιτελούνται.

• Συνιστώσες Καθορισμού. Χρησιμοποιούνται για να καθορίσουν τα λειτουργικά χαρακτηριστικά της αντίστοιχης κλάσης και παρέχουν σαν πόρους ένα σύνολο λειτουργικών καθορισμών οι οποίοι υλοποιούνται σε εραρχίες υπό μορφή δέντρων. Κάθε δεύτερο αντιστοιχεί σε μια συγκεκριμένη λειτουργία. Οι εξειδικεύσεις στο δέντρο αυτό γίνονται μέσω ενός συνόλου ιδιοτήτων των οποίων ορίζονται από το χρήστη.

Μπορούμε τώρα να δούμε τον τρόπο με τον οποίο γίνεται η διασύνδεση των συνιστώσων. Μία συνιστώσα υλοποίησης συνδέεται με μία συνιστώσα σχεδίασης που αντιστοιχεί σε αυτή την υλοποίηση, η οποία με τη σειρά της συνδέεται με την αντίστοιχη συνιστώσα καθορισμού. Οι διασυνδέσεις αυτές δεν είναι μονοσήμαντες. Ο λόγος είναι ότι κατά τη σχεδίαση μιας οντοκεντρικής κλάσης μπορούμε να πάρουμε διαφορετικές αποφάσεις. Όμως μπορούμε να πάρουμε διαφορετικές αποφάσεις κατά την υλοποίηση κάποιας συγκεκριμένης σχεδίασης.

Οι διαφορετικές αυτές αποφάσεις λαμβάνονται για να ικανοποιήσουμε ένα σύνολο μηλειτουργικών προδιαγραφών, οι οποίες σχετίζονται με την απόδοση του παραγόμενου προγράμματος. Καθώς αυτές οι αποφάσεις είναι απαραίτητες να αναχρησιμοποιηθούν κατά τη φάση της επιλογής, πρέπει να αποθηκευούν στον ταμειακό. Αυτό επιτυγχάνεται με τη χρήση ενός συνόλου συνιστώσων εξάρτησης. Κάθε συνιστώσα εξάρτησης διασυνδέεται μία συνιστώσα με την αντίστοιχη της στο υψηλότερο επίπεδο. Οι πόροι που παρέχονται από τις συνιστώσες εξάρτησης είναι το σύνολο των μηλειτουργικών καθορισμών που ικανοποιεί η αντίστοιχη εξάρτηση.

Όπως αναφέραμε πριν, η όλη οργάνωση έγινε κατά τέτοιο τρόπο ώστε να δημιουργούνται περιπτώσεις. Μία περίπτωση αποτελείται από μία συνιστώσα υλοποίησης, την αντίστοιχη συνιστώσα σχεδίασης και την αντίστοιχη συνιστώσα καθορισμού. Αυτές διασυνδέονται μέσω δύο συνιστώσων εξάρτησης.

Κάθε περίπτωση αποτελείται από 4 υπο-περιπτώσεις η ανάκληση των οποίων καθορίζει και την ανάκληση της ολικής περίπτωσης. Τα βήματα που ακολουθούνται κατά την ανάκληση περιπτώσεων είναι τα εξής:
• Βάσει των λειτουργικών καθορισμών που επιθυμεί ο χρήστης, εντόπισε το κατάλληλο σύνολο από πόρους λειτουργικών καθορισμών οι οποίοι αποτελούν ένα μονοπάτι από ένα φύλο προς τη ρίζα του αντίστοιχου δέντρου, κάνοντας αν χρειάζεται την απαραίτητη προσαρμογή (υπο-περίπτωση 1).

• Χρησιμοποιώντας το συνόλο που ανακτήθηκε κατά το προηγούμενο βήμα, εντόπισε την αντίστοιχη συνιστώσα καθορισμού και στη συνέχεια εντόπισε την αντίστοιχη συνιστώσα σχεδίασης λαμβάνοντας αν χρειάζεται την κατάλληλη σχεδιαστική απόφαση (υπο-περίπτωση 2).

• Χρησιμοποιώντας τη συνιστώσα σχεδίασης που ανακτήθηκε κατά το προηγούμενο βήμα, εντόπισε την αντίστοιχη συνιστώσα υλοποίησης λαμβάνοντας αν χρειάζεται την κατάλληλη απόφαση υλοποίησης (υπο-περίπτωση 3).

• Στην περίπτωση που η συνιστώσα υλοποίησης είναι παραμετροποιημένη, αντικατέστησε τον παραμετροποιημένο τύπο με το συγκεκριμένο τύπο που επιθυμεί ο χρήστης (υπο-περίπτωση 4).

Κατά την προσαρμογή των περιπτώσεων μετατρέπεται (αν είναι δυνατό) μια παλαιά περίπτωση σε μια νέα, ανάλογα με τις ανάγκες του χρήστη. Αυτή η μετατροπή γίνεται αντικαθιστώντας τους τύπους των ιδιοτήτων του λειτουργικού καθορισμού που ταιριάζει περισσότερο στη νέα κατάσταση (υπο-περίπτωση 1), με κάποιους νέους τύπους βάσει των καθορισμών που επιθυμεί ο χρήστης. Η μετατροπή των περιπτώσεων μπορεί να γίνει στις εξής περιπτώσεις: i) αν μπορεί να αποδειχθεί ένας κανόνας αντικατάστασης για τους συγκεκριμένους τύπους (καθώς υπάρχει μια σχέση ομοιότητας ανάμεσα τους) και ii) αν ο χρήστης επιβεβαιώνει ότι μπορεί να γίνει αυτή η αντικατάσταση (σε αυτή την περίπτωση δημιουργείται ένας νέος κανόνας αντικατάστασης ο οποίος αποθηκεύεται στον ταμειακό). Στη συνέχεια το σύστημα εντοπίζει τον αντίστοιχο κώδικα (λαμβάνοντας αν χρειάζεται τις κατάλληλες αποφάσεις) και προτείνει στο χρήστη να αντικαταστήσει τους παλαιούς τύπους με τους νέους. Επιπλέον υποδεικνύει άλλες αλλαγές που χρειάζεται να γίνονται σε ένα τμήμα κώδικα μετά από μία παρόμοια αντικατάσταση τύπων που συνέβη στο παρελθόν.

Μια ειδική μορφή προσαρμογής περιπτώσεων συμβαίνει όταν η παλαιά υπο-περίπτωση (τύπου 1) αποτελείται από ένα σύνολο παραμετροποιημένων ιδιοτήτων ενώ οι αντίστοιχες ιδιότητες της νέας υπο-περίπτωσης αποτελούνται από συγκεκριμένους τύπους. Σε αυτή την περίπτωση το σύστημα ελέγχει αν οι συγκεκριμένοι τύποι της νέας υπο-περίπτωσης μπορούν να αντικαταστήσουν τους παραμετροποιημένους τύπους. Αυτό γίνεται αποδεικνύοντας ένα σύνολο κανόνων παραμετροποίησης.
Ο τρόπος με τον οποίο εξελίσσεται ο ταμειοτήρας μας απασχολήσει ιδιαίτερα. Ο κύκλος ανάπτυξης λογισμικού έχει οργανωθεί κατά τέτοιο τρόπο ώστε να βελτιώνεται η ποιότητα και η αποδοτικότητα του ταμειοτήρα με την πάροδο του χρόνου. Η οργάνωση των περιπτώσεων που χρησιμοποιήσαμε παρέχει μια ισχυρή βάση για ελεγχόμενη εξέλιξη και επέκταση του ταμειοτήρα με γενικευμένες περιπτώσεις υψηλής ποιότητας.

Η μέθοδος εξέλιξης βασίζεται σε μάθηση και εφαρμογή γενικευμένων. Με τον όρο μάθηση εννοούμε προσαρμοστικές αλλαγές στις αποθηκευμένες περιπτώσεις. Οι αλλαγές αυτές γίνονται κατά τέτοιο τρόπο ώστε να επιτυγχάνονται τα ίδια αποτελέσματα πιο αποδοτικά και χωρίς να επαναλαμβάνονται λάθη του παρελθόντος. Στη συνέχεια ακολουθούν οι διαφορετικοί τρόποι εξέλιξης του ταμειοτήρα:

- Δημιουργία γενικευμένων συνιστώσων. Μια νέα γενικευμένη συνιστώσα δημιουργείται μετά από την προσαρμογή μιας περίπτωσης με αντικατάσταση ενός (ή περισσότερων) τύπων, στην περίπτωση που ο χρήστης συμφωνεί με την πρόταση του συστήματος. Το αποτέλεσμα είναι η αντικατάσταση της περίπτωσης που ανακτήθηκε από τον ταμειοτήρα με μια νέα η οποία διαφέρει στο γεγονός ότι ο τύπος που αντικαταστάθηκε έχει παραμετροποιηθεί. Επιπλέον, δημιουργείται ένας νέος κανόνας παραμετροποίησης (για το συγκεκριμένο παραμετροποιημένο τύπο) ο οποίος αποθηκεύεται στον ταμειοτήρα.

- Δημιουργία συγκεκριμένων συνιστώσων. Στην περίπτωση που ο χρήστης δε συμφωνεί με τη δημιουργία μιας νέας γενικευμένης συνιστώσας, τότε αποθηκεύεται στον ταμειοτήρα μια νέα περίπτωση που διαφέρει από την παλαιά στο γεγονός ότι ο παλαιός τύπος έχει αντικατασταθεί με το νέο που επιθυμεί ο χρήστης.

- Εισαγωγή νέου κώδικα. Στην περίπτωση που δεν μπορεί να βρεθεί στον ταμειοτήρα μια περίπτωση που να ικανοποιεί το χρήστη, τότε αυτός μπορεί να δημιουργήσει από μόνος του τον κατάλληλο κώδικα και να τον αποθηκεύσει στον ταμειοτήρα δίνοντας τις κατάλληλες περιγραφές.

- Δημιουργία μιας νέας απόφασης. Μια νέα απόφαση δημιουργείται στην περίπτωση που ο χρήστης δεν ικανοποιείται από το τμήμα κώδικα που ανακτήθηκε από τον ταμειοτήρα, καθώς παραβιάζονται κάποιες μη-λειτουργικές προδιαγραφές. Σε αυτή την περίπτωση ο χρήστης μπορεί να αποθηκεύσει στον ταμειοτήρα ένα νέο τμήμα κώδικα που ικανοποιεί τις συγκεκριμένες προδιαγραφές. Επιπλέον, είναι απαραίτητο να περιγράψει αυτές τις προδιαγραφές (χρησιμοποιώντας μη-λειτουργικούς καθορισμούς) για να συσχετιστούν με τη νέα συνιστώσα εξάρτησης που δημιουργείται.

- Βελτίωση Καθορισμών. Καθώς δεν υπάρχει εγγύηση ότι ο χρήστης θα παρέχει πάντα σωστούς καθορισμούς, είναι πιθανό να ανακτώνται ακατάλληλες περιπτώσεις. Για αυτό το
λόγο υπάρχει η δυνατότητα βελτίωσης των αποθηκευμένων καθορισμών με τροποποίηση των ιδιοτήτων τους.

- Βελτίωση κανόνων παραμετροποίησης. Ο κανόνας παραμετροποίησης βελτιώνεται με την εισαγωγή εξαιρέσεων για τις μη-νόμιμες τιμές του παραμετροποιημένου τύπου.

- Μάθηση Ομοιοτήτων. Όπως αναφέρθηκε παραπάνω, το σύστημα μαθαίνει για ομοιότητες τύπων κατά τη φάση προσαρμογής περιπτώσεων.

Η υλοποίηση του ταμειωτήρα όπου αποθηκεύονται οι περιπτώσεις και η απαιτούμενη εμπειρία ανάπτυξης συστημάτων συνεπάγεται την ύπαρξη ενός Συστήματος Διαχείρισης Βάσεων Δεδομένων (ΣΔΒΔ). Το συνολικό κόστος του συστήματος αποτελείται από το κόστος του ΣΔΒΔ και το κόστος της προσπάθειας των χρηστών για να δουλέψουν με το σύστημα.

Όσο προσ το κόστος του ΣΔΒΔ, οι ερωτήσεις στη βάση δεδομένων περιορίζονται σε ένα σύνολο τυποποιημένων ερωτήσεων που γίνονται κατά τις φάσεις ανάκλησης, προσαρμογής και εξέλιξης. Κατά συνέπεια, βελτιστοποιήσαμε αυτές τις ερωτήσεις με μια αυτόματο τρόπο περιορίζοντας το χρήστη στην παροχή κάποιων τιμών, σύμφωνα με τις προτροπές του συστήματος.

Στη συνέχεια υπολογίσαμε την πολυπλοκότητα των αλγοριθμίων που χρησιμοποιούνται στη μέθοδο που έχουμε αναπτύξει, ως προς τον αριθμό των απαιτούμενων προσπελάσεων στο δίσκο του συστήματος. Όπως προκύπτει από την ανάλυση της συνεχείας, η πολυπλοκότητα των αλγοριθμίων είναι στη χειρότερη περίπτωση γραμμική. Αυτό μας εξασφαλίζει την ποιότητα απόδοσης του συστήματος ακόμα και στην περίπτωση που ο αριθμός των αποθηκευμένων περιπτώσεων γίνεται πολύ μεγάλος.

Η μέθοδος που παρουσιάζεται σε αυτή τη διατριβή έχει αξιολογηθεί μέσω μιας πρότυπης υλοποίησης για την αναχρησιμοποίηση κώδικα γραμμών στη γλώσσα C++. Οι λόγοι που μας οδήγησαν στην επιλογή της γλώσσας C++ είναι: i) η δημιουργικότητα της γλώσσας, ii) η αποδοτικότητα των προγραμμάτων που γράφονται με τη γλώσσα αυτή και iii) το γεγονός ότι η C++ δεν παρέχει την τεχνική της γενίκευσης.

Το πρότυπο σύστημα τρέχει σε μηχανήματα τύπου SparStations και Sun 4 κάτω από Unix. Η γλώσσα προγραμματισμού που χρησιμοποιήθηκε είναι μια γλώσσα με τα χαρακτηριστικά της γλώσσας Prolog, η οποία ονομάζεται MegaLog. Η MegaLog ολοκληρώνει μια Βάση Γνώσης μαζί με μία γλώσσα λογικού προγραμματισμού με σκοπό τη μόνιμη αποθήκευση γνώσεων κατά ένα τρόπο που επιτρέπει την αποδοτική ανάκληση και επεξεργασία της από προγράμματα λογικής.

Θέλοντας να έχουμε μία ένδειξη για την αποτελεσματικότητα της προτεινόμενης μεθόδου, εκτελέσαμε ένα εργαστηριακό πείραμα με μεγάλο αριθμό συμμετοχών. Το πείραμα βασίστηκε σε ένα ταμειωτήρα όπου είχαν αποθηκευτεί 52 κλάσεις της γλώσσας C++ και συμμετείχαν 79
προπτυχιακοί φοιτητές του τμήματος Επιστήμης Υπολογιστών. Τα αποτελέσματα αυτού του πειράματος είναι ιδιαίτερα ενθαρρυντικά παρόλο που τα μέλη της πειραματικής ομάδας δεν είχαν συστηματική γνώση ή εμπειρία χρήσης του συστήματος.

Συμπερασματικά, η μέθοδος που παρουσιάζεται σε αυτή τη διατριβή ολοκληρώνει ιδέες και τεχνικές από διαφορετικά πεδία όπως Αναπαράσταση Γνώσης, Μηχανική Λογισμικού και Μηχανική Μάθησης, με στόχο την αποτελεσματική αναχρησιμοποίηση οντοκεντρικού κώδικα. Σε πρακτικό επίπεδο η μέθοδος έχει αξιολογηθεί μέσω μιας πρότυπης υλοποίησης για την αναχρησιμοποίηση κώδικα γραμμένου στη γλώσσα C++.

Οι επεκτάσεις αυτές της δουλειάς σε θεωρητικό επίπεδο περιλαμβάνουν μια μελέτη για την επεκτασιμότητα της μεθόδου για την αναχρησιμοποίηση εφαρμογών. Επίσης, σχετικά με την ανάκληση παρόμοιων περιπτώσεων του παρελθόντος, είναι απαραίτητο να μελετηθούν διαφορετικές μορφές ομοιότητας (π.χ. σημασιολογική ομοιότητα) πέρα από την ομοιότητα τύπων. Τέλος, είναι απαραίτητο να γίνει μια μελέτη για την εξασφάλιση της ποιότητας των συνιστώσων που εισάγονται στον ταμειατήρα.

Σε πρακτικό επίπεδο είναι απαραίτητο να γίνει μια νέα υλοποίηση του συστήματος με μια αποδοτική γλώσσα προγραμματισμού. Στη συνέχεια το σύστημα πρέπει να εγκατασταθεί σε ένα μεγάλο οργανισμό ανάπτυξης λογισμικού όπου μέσω της χρήσης του από ένα μεγάλο αριθμό χρηστών θα καταστεί δυνατή η αξιολόγηση τόσο του συστήματος, όσο και της προτεινόμενης μεθόδου. Τέλος, θα ήταν χρήσιμη η εφαρμογή τεχνικών παράλληλης επεξεργασίας κατά την ανάκληση των περιπτώσεων ώστε να πετύχουμε ακόμα καλύτερες επανόρθωσεις.

Εκπόνηση: Καθηγητής Γιάννης Βασιλείου, Εθνικό Μετσόβιο Πολυτεχνείο, (Πανεπιστήμιο Κρήτης έως Ιούνιο 1993)
Acknowledgements

This dissertation could not have been completed without the help of many people. First, I feel grateful to my thesis advisor Professor Yannis Vassiliou for his guidance, his tireless support, the patience and the confidence he showed to me and to my work.

I wish to thank the members of my dissertation committee, Professors Panos Constantopoulos, Manolis Katevenis, Christos Nikolaou, Stelios Orphanoudakis, Manolis Skordalakis, and Timos Sellis for valuable contribution to this work. They got into deep questions and with their comments helped me improve this dissertation in both content and presentation.

I also thank Professors John Mylopoulos and Matthias Jarke for their early comments, and for the useful discussions we had.

I cannot omit in this acknowledgements my colleagues Giorgos Georgiakakis, Maria Karavassili, Popi Halkia, and Costas Dadouris. The cooperation with them was a great and pleasant experience.

Finally, the completion of this work owes to the support and encouragement I received from my parents and my fiancée Maria.

This research was financially supported by the Institute of Computer Science, Foundation for Research and Technology, Hellas (FORTH) and the Commission of European Communities through ESPRIT projects DAIDA and ITHACA.
To my parents Kostas and Niki, and to Maria
## Contents

### 1. Introduction and Related Work ................................................................. 1

1. Introduction ................................................................................................. 5
   1.1 The Context ......................................................................................... 5
   1.2 Research Issues .................................................................................. 6
   1.3 Proposed Solution ............................................................................. 7
   1.4 Thesis Overview ................................................................................. 9

2. Software Reuse ............................................................................................ 11
   2.1 Introduction ....................................................................................... 11
   2.2 Software Crisis and Software Reuse .................................................. 11
   2.3 History of Software Reuse .................................................................. 14
   2.4 Reuse Framework .............................................................................. 19
   2.5 Summary ............................................................................................ 26

3. Related Research - Limitations ................................................................. 27
   3.1 Introduction ....................................................................................... 27
   3.2 Repository Organization Methodologies ............................................ 28
      3.2.1 Library Cataloguing .................................................................. 28
      3.2.2 Hierarchical Classification Schemes ........................................ 29
      3.2.3 Faceted Classification Schemes ............................................... 30
      3.2.4 Hypertext Form ....................................................................... 30
      3.2.5 Conceptual / Semantic Organization ....................................... 31
      3.2.6 Hybrid Organization ............................................................... 32
   3.3 Retrieval Methodologies ..................................................................... 34
      3.3.1 Formal Specifications .................................................................. 34
      3.3.2 Databases and Database (Query) Retrieval .............................. 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3 Knowledge Bases and Reasoning / Deductive Tools</td>
<td>35</td>
</tr>
<tr>
<td>3.3.4 Hypertext Retrieval</td>
<td>37</td>
</tr>
<tr>
<td>3.3.5 Information Retrieval</td>
<td>37</td>
</tr>
<tr>
<td>3.3.6 Hybrid Approaches</td>
<td>37</td>
</tr>
<tr>
<td>3.4 Related Research Limitations</td>
<td>38</td>
</tr>
<tr>
<td>3.5 AI-based Approaches for Software Reuse</td>
<td>40</td>
</tr>
<tr>
<td>3.6 Rule-Based Reasoning vs Case-Based Reasoning</td>
<td>42</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>46</td>
</tr>
</tbody>
</table>

**II. Organizational Framework ................................................................. 47**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Organizational Framework</td>
<td>51</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>51</td>
</tr>
<tr>
<td>4.2 Repository Organization</td>
<td>52</td>
</tr>
<tr>
<td>4.3 CILs</td>
<td>54</td>
</tr>
<tr>
<td>4.4 Telos</td>
<td>55</td>
</tr>
<tr>
<td>4.5 Descriptions / Components</td>
<td>57</td>
</tr>
<tr>
<td>4.5.1 Implementation Components</td>
<td>58</td>
</tr>
<tr>
<td>4.5.2 Design Components</td>
<td>61</td>
</tr>
<tr>
<td>4.5.3 Specification Components</td>
<td>64</td>
</tr>
<tr>
<td>4.5.4 Dependency Components</td>
<td>69</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>75</td>
</tr>
</tbody>
</table>

**III. The Method ......................................................................................... 77**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. Case Problems - Solutions</td>
<td>81</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>81</td>
</tr>
<tr>
<td>5.2 Case Problems</td>
<td>81</td>
</tr>
<tr>
<td>5.2.1 Case representation</td>
<td>82</td>
</tr>
<tr>
<td>5.2.2 Case indexing</td>
<td>82</td>
</tr>
<tr>
<td>5.2.3 Case storage and retrieval</td>
<td>83</td>
</tr>
<tr>
<td>5.2.4 Case adaptation</td>
<td>84</td>
</tr>
<tr>
<td>5.2.5 Learning and generalization</td>
<td>85</td>
</tr>
<tr>
<td>5.3 Case Representation</td>
<td>87</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.5.4 Optional Match</td>
<td>142</td>
</tr>
<tr>
<td>8.6 Evolution Algorithms</td>
<td>143</td>
</tr>
<tr>
<td>8.6.1 Specific Component Creation</td>
<td>143</td>
</tr>
<tr>
<td>8.6.2 Generic Component Creation</td>
<td>143</td>
</tr>
<tr>
<td>8.6.3 New Code Insertion</td>
<td>144</td>
</tr>
<tr>
<td>8.6.4 Decision Creation</td>
<td>145</td>
</tr>
<tr>
<td>8.6.5 Specifications Refinement</td>
<td>145</td>
</tr>
<tr>
<td>8.6.6 Parameterization Rule Refinement</td>
<td>146</td>
</tr>
<tr>
<td>8.6.7 Learning about Similarities</td>
<td>146</td>
</tr>
<tr>
<td>8.7 Summary</td>
<td>147</td>
</tr>
<tr>
<td><strong>IV. Examples and Illustrations</strong></td>
<td>149</td>
</tr>
<tr>
<td>9. Examples and Illustrations</td>
<td>153</td>
</tr>
<tr>
<td>9.1 Introduction</td>
<td>153</td>
</tr>
<tr>
<td>9.2 The C++ Case</td>
<td>153</td>
</tr>
<tr>
<td>9.3 System Overview</td>
<td>154</td>
</tr>
<tr>
<td>9.4 A Sample Session</td>
<td>162</td>
</tr>
<tr>
<td>9.5 Usage Experiment</td>
<td>167</td>
</tr>
<tr>
<td>9.6 Summary</td>
<td>170</td>
</tr>
<tr>
<td><strong>V. Conclusions and Extensions</strong></td>
<td>173</td>
</tr>
<tr>
<td>10. Conclusions - Future Work</td>
<td>177</td>
</tr>
<tr>
<td>Appendix A - Usage Scenario</td>
<td>181</td>
</tr>
<tr>
<td>Appendix B - Questionnaires and Results</td>
<td>183</td>
</tr>
<tr>
<td>References</td>
<td>187</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Software Reuse Process ............................................................... 20
4.1 An IG with three nodes ................................................................ 55
4.2 A functional specification resources’ hierarchy ............................... 67
4.3 Interconnections among software and dependency components .......... 73
5.1 An example of a matching case ..................................................... 90
9.1 General Architecture of the Prototype System .............................. 155
9.2 Implementation decisions for ordered sets ..................................... 169
Part I

Introduction and Related Work
Part Introduction

This part introduces the problems on software reuse that we will be dealing with in this thesis, our approach to these problems, and the main objectives and motivations guiding the work. Furthermore, the reuse framework and the related research are presented. More specifically, the introduction is structured as follows:

Chapter 1 The problem statement is presented, followed by the chosen solution approach, the main achievements, and the structure of the thesis.

Chapter 2 The software reuse process is analyzed, and the basic research problems encountered in this process are identified.

Chapter 3 A taxonomy of the current efforts on software reuse is presented, and the limitations of these efforts are addressed.
Chapter 1

Introduction

1.1 The Context

Developing large software systems in a highly productive, high quality, and cost-effective manner has been and will be a major challenge in the next years for all large high technology, engineering and manufacturing firms [Scacc84].

More than ever before, organizations are looking for new methodologies and tools to increase the productivity of their software development staff, to improve the cost-effectiveness of the software development process, and to ensure the quality and reliability of systems produced. Scientific progress in support of this has been generally slow.

Effective reuse of knowledge, processes, and products from previous software developments, can potentially increase productivity and quality in software projects by an order of magnitude [Bigger89]. In fact, software production using reusable components will probably be crucial to the software industry’s evolution to higher levels of maturity [Galdi91].

The concept of software reuse has been part of the programming heritage since the origins of the stored-program computer EDSAC at the University of Cambridge in 1949 where the first subroutine library was proposed. Until recently, little has been done to extend program reusability beyond this rather simple level. McIlroy [McIlr69] in 1968 envisioned software component factories, but apparently only the Japanese listened [Walt91]. Nowadays, interest in applying this
simple, but effective concept of not reinventing the wheel has been reborn, and the role of reusable software has been recognized and studied in a large number of cases [Freem83, Horow84, Bigger87].

To be practical, reuse must address not only technical but managerial, economic, legal, cultural, and technology transfer issues [Frakes90a]. However, current technology still presents several limitations in handling some very important technical issues. In this thesis, we are dealing with such technical issues which are listed below.

1.2 Research Issues

The first problem we encounter in reusing software arises from the nature of the objects to be reused. The concept is simple - use the same object more than once. But with software it is difficult to define what an object is apart from its context [Freem83]. We have programs, parts of programs, specifications, requirements, architectures, test cases and plans, all related to each other. The reuse of each software object implies the concurrent reuse of the objects associated with it, and informal information traveling with the objects. Thus, we are lead to reuse more than code. Software objects and their relationships incorporate a large amount of experience from past development. We need to reuse this experience in the production of new software. In turn, such experience makes it possible to reuse software objects [Basil88].

A second major problem in code reuse is the lack of a set of reusable components, despite the large amount of software that already exists in the disks or tapes of many software producers. Reuse efficiency and cost effectiveness require a large catalog of available reusable objects.

Before anything can be reused, it has to be identified and created. After that, it has to be described and stored in some form of library (repository). This stage includes problems related to the transformation of reusable information into components. What is the proper size and form of reusable components and how should they be classified? What are proper techniques and languages for describing certain aspects of components? How to guarantee their quality?

Another important decision that needs to be made, is the determination of the repository type. Is a relational database enough or is there a need for more powerful modeling?
1.2 Research Issues

The user of the reuse repository seldom has complete knowledge about all the components stored, and probably not even an exact understanding of what components he is looking for. This introduces the selection problem. It is the problem of deciding what existing software may be useful in meeting the design (or requirement) specifications.

The selection task becomes even harder when coupled with the adaptation problem. In many cases, it is not possible to find in the repository a component that fits exactly the developer's needs. However, there may exist a component which with minor modifications will finally meet the initial specifications. The problem is finding such a component, understanding it, and making the appropriate modifications.

Finally, the object’s repository evolution problem is a major concern, influencing directly the selection and adaptation problems. The entire software production cycle should be integrated and organized in such a way as to improve the quality of the repository as well as its maintainability.

1.3 Proposed Solution

This thesis presents a novel approach, which employs a special form of Case-Based Reasoning in conjunction with the specificity-genericity hierarchy to semi-automatically locate the appropriate code in a software repository, possibly adapting it to particular requirements, while dealing with the evolution of the repository by adding (if needed) new components and making the proper repository re-organization.

Genericity is the technique that allows a module to be defined with parameterized types. Instances of the module are then produced by supplying different types as actual parameters. Type parameterization is the ability to define a type in terms of another, unspecified type. Versions of the parameterized type may then be created for several particular parameter types.

A general form of parameterized types can be cleanly integrated into object-oriented languages that do not provide a built-in form of genericity. However, genericity by itself, is not flexible enough because it can not capture fine grain of commonality between groups of implementations of the same general data abstraction. This is because there are only two levels of modules: a) generic modules, which are parameterized and thus open to variation, but not directly usable, and b) fully instantiated modules (specific modules) which are directly usable but not open to
refinement. Moreover, a repository with generic components will still suffer to a large degree from the reuse problems mentioned before. Although the search space for selection is reduced (one generic instead of several specific implementation components is stored in the repository), this reduction is not considerable. Furthermore, all additional knowledge for adapting modules and reorganizing the evolving repository lies exclusively with the user (developer). In summary, the system’s support for reusability is quite limited.

In order to effectively reuse object-oriented code using genericity, software developers could depend on experiential knowledge gathered and stored in the repository while developing similar software components. AI methods can be incorporated in order to achieve better support from the system to a) encode and abstract this knowledge, and b) reuse it.

Case-Based Reasoning (CBR) [Barlet91] is a method of solving problems based on the transfer of past experience to new problem situations. CBR as a learning paradigm has several advantages: Firstly, there are several performance enhancements, as it provides: shortcuts in reasoning, the capability of avoiding past errors, and the capability of focusing in the most important parts of a problem first. Secondly, learning can be eased since CBR does not require a causal model or a deep understanding of the domain. Thirdly, individual or generalized cases can also serve as explanations; these explanations are trivial to generate, and probably more satisfactory than the chains generated by expert systems.

A problem for Case-Based Reasoning in a reuse-based software development is to encode software components and application developer’s experience, in a fashion that it can be reasoned about it. The term software component here enables reference to building blocks of software systems at any stage of the software life cycle. In order to model these components, the general framework of Component Interconnection Languages (CILs) has been extended and modified. In addition, the Telos language [Mylop91a] - a language that satisfies all the requirements so as to be considered a CIL - is adopted, for presenting this model in a more formal and markable form.

The software components, and the application developer’s experience are organized in a way that enables the formation of matching cases. After these matching cases are represented and indexed, they are organized in the repository into an efficient structure for retrieval. The goal of case retrieval is to return the most similar past case that is relevant to the input situation. Case adaptation takes a retrieved case that meets most of the needs of current situations and turns it
into one that meets all the situation needs.

The repository evolution is a major issue, directly related to its quality. The entire software production cycle is integrated in such a way as to improve the quality of the repository, as well as its maintainability. The organization of cases used, provides a strong basis for controlled evolution and expansion of the repository with quality generic components (including parameterized code) through the application of the evolution method. This method is based on failure-driven learning and explanation-based generalizations. Learning denotes changes in the cases stored in the repository that are adaptive in the sense that enable the system to do the same task drawn from the same population more efficiently, and without repeating previously made errors.

The method presented in this thesis has been evaluated through a prototype implementation, which addresses the reuse of C++ code. The prototype system runs on SparcStations and Sun 4 series under Unix. The language used for the implementation is a Prolog-like language called MegaLog [Horsf90], which provides a powerful programming environment for building the next generation Database and Knowledge Base Management Systems.

Additionally, a usage experiment performed in order to get an indication on the feasibility of the proposed approach and the usage characteristics of the prototype system. The results of this experiment are very encouraging, although that the participants had no previous systematic knowledge or usage experience of the system.

1.4 Thesis Overview

There are five main parts of the thesis: introduction and related work, organizational framework, the method, examples and illustrations, and conclusions and extensions. They all contain a number of chapters and are briefly described below:

Part 1 — Introduction and Related Work introduces the thesis and presents the related research. More specifically, the part is structured as follows:

Chapter 1 presents the problem statement, followed by the chosen solution approach, the main achievements, and the structure of the thesis.
Chapter 2 analyzes the software reuse process, and identifies the basic research problems encountered in it. A historical account of the early efforts on software reuse is also presented, and the limitations of these efforts are addressed.

Chapter 3 presents a taxonomy of the current efforts on software reuse and addresses their limitations.

Part II — Organizational Framework presents the organizational framework used in our approach. More specifically:

Chapter 4 presents the crucial factors that enter the picture while tackling the repository organization problem, followed by the chosen solution approach.

Part III — The Method presents the proposed method. More specifically:

Chapter 5 addresses the major research issues on Case-Based Reasoning. In addition the solutions given for the issues of case representation and indexing are presented.

Chapter 6 describes analytically the algorithms used for case retrieval and adaptation.

Chapter 7 presents a method based on failure-driven learning and explanation-based generalizations that is used for a controlled evolution and expansion of the repository.

Chapter 8 analyzes the complexity of the algorithms used for case retrieval, adaptation and repository evolution.

Part IV — Examples and Illustrations presents a prototype implementation, which addresses the reuse of C++ code using the method described in part III. In particular:

Chapter 9 presents an overview of the prototype system, and a sample session that illustrates the features and the "feel" of the prototype implementation. Furthermore, the usage experiment is presented, and the processed results are analyzed.

Part V — Conclusions and Extensions concludes the thesis. More specifically:

Chapter 10 sums up the major achievements of this thesis, and presents some guidelines for further work.
Chapter 2

Software Reuse

2.1 Introduction

The existing gap between demand and our ability to produce high-quality software cost-effectively (often referred to as the "software crisis"), calls for an improved software development technology.

It is generally recognized that a reuse-oriented development technology can significantly contribute to higher quality and productivity. This chapter presents the concept of software reuse. A historical account of early efforts on software reuse is presented, and the limitations of these efforts are addressed. In the sequel, the software reuse process is analyzed, and the basic problems occurring during this process are identified.

2.2 Software Crisis and Software Reuse

As early as 1973, Barry Boehm projected that software costs would, by 1985, reach or exceed 90 percent of the total cost of data processing (the combined hardware and software costs) [Boehm81, Boehm82]. It was clear that hardware costs were rapidly falling and that personnel costs in the labor-intensive software development arena were rising. The projection has turned out to be all true [Genue91].

The rapid pace at which hardware innovations are announced, particularly in the area of microprocessor technology, now well exceeds the capabilities of our software development
technology. An entire generation of processor hardware technology has arrived and been suspended without any software to support it reaching the marketplace. Further, the ability to develop software to meet new needs is rapidly losing ground to software maintenance [Rine91].

Organizations have already recognized how the software crisis affects them [Bound93]. Randall Jensen of Hughes Aircraft, for instance, has identified six key software problems resulting from systems-development approaches as commonly applied [Jense86]:

- products exceeding cost estimates,
- late delivery,
- inadequate performance,
- impossible maintenance,
- prohibitive modification costs, and
- unreliability.

Effective reuse of knowledge, processes, and products from previous software developments, can increase productivity and quality in software projects by an order of magnitude [Bigger89]. In fact, software production using reusable components will probably be crucial to the software industry’s evolution to higher levels of maturity [Galdi91].

Reusing software increases the productivity, due to the following reasons:

- Software reuse amplifies programming capabilities [Bigger84]. The software developer has fewer symbols to write when large portions of code or design are copied without major modifications.

- Software reuse reduces the amount of documentation and testing required.

- The system becomes easier to maintain and modify as the software developers are more familiar with the reusable building blocks from which it is constructed, and can more easily understand the complete system design [Lewis90].
2.2 Software Crisis and Software Reuse

On the other hand, reusing software improves the quality in software projects, due to the following characteristics of the reused software:

- It is on principle well designed (i.e., designed for reuse).
- It is well documented - according to an established standard.
- It is well tested [Myers92]. The more software is reused, the greater the probability an error will not be found.
- Its function is well understood and likely to be used appropriately.

Some of the major organizations (and especially some of the Japanese) have been running a software reuse program for more than a decade [Diaz93]. The following summary of the benefits shows what they have achieved [Cusum91, Kruze90]:

- Hitachi reduced the number of late projects from 72% to 7% in 4 years, and now averages 12% late. Also, it reduced the number of defects per machine in the field from 100 to 13 in 6 years.
- Toshiba improved productivity (in terms of delivered lines of code) by a factor of nearly 3 in 9 years (including 48% reused code). At the same time, the number of defects per 1000 lines of code dropped from 7-20 to 2-3.
- NEC improved productivity by 26% to 91%, and reduced defects by one third.
- Fujitsu reduced the defect rate in operating systems software by a factor of more than 10, and increased productivity by two thirds.
- Raytheon Missile Systems Division increased productivity by up to 50%.
- NobelTech has more than doubled its productivity, and expects future productivity improvements to be of a much higher level.

Success stories from Japan suggest that you can reuse anything as long as you organize properly, which would make reuse purely a management issue. However, common sense also suggests that you can waste good managers fighting technology that is bad for reuse [Walt91].
2.3 History of Software Reuse

Within the software community the concept of software reuse is considered to be a part of the programming heritage since the origins of the stored program computer EDSAC at the University of Cambridge in 1949. Mauric Wilkes [Offic73] first recognized the need of avoiding the redundant effort in writing scientific subroutines and recommended a library of routines to be kept for general use.

Later on, in the NATO Software Engineering Conference of 1968 (which in general is considered the birthplace of the software engineering field), the term "software crisis" was introduced, while from the beginning "software reuse" has been touted as a means of overcoming it. In an invited paper at this conference, McIlroy [McIlr69] proposed using modular software units, and reuse has been behind many software developments. However, the method has never acquired real momentum in industrial environments and software projects, despite its informal presence there.

Historically speaking, the subroutine was the first reusable software artifact, with run-time parameters serving as the mechanism for variation. Subroutines were building blocks that could be "glued" together (via calls) to form programs.

In contrast to parameterization at run time, assembler language programmers tended to scorn context-switching overhead in favor of in-line macro-expansion. The macro became a very useful parameterization tool whereby source code patterns could be identified and instantiated with different parameter values. In fact one can look a macro as a framework "glue" into one can "plug in" parameters to generate an executable program segment. Furthermore, using extrapolation, one could argue that an application generator is a very large macro.

Using the techniques mentioned above, most programmers do reuse a lot of code. The characteristic is that the code they reuse, is their own code, maintained in their own libraries. What often makes senior programmers good is that they have accumulated so many code fragments, that they can write apparently long and complex programs in less time. This happens because they are mainly assembling sub-programs and they do not have to write the whole program from scratch. However, when they have to move to a new operating system and a new language they often have problems for a while. Furthermore, within their own libraries they know pretty well if their library contains a particular kind of routine or not. In other’s libraries they can not be sure if they have not found yet what they were looking for, or if it is not actually there.
As programming languages evolve, and object-oriented programming becomes popular, the unit of abstraction increases to collections of both operations and data (e.g., modules, packages, classes), often based on sound mathematical principles (i.e., an algebra) [Wirfs90]. A concise genealogical tree of object-oriented languages and a discussion of their histories can be found in [Booch91].

An object has an encapsulated state that persists over time. It is characterized by a set of services that it provides, and a set of services that it requires from other objects. In order to request a service, a client must identify the object to perform it. A service is an abstraction of:

- a method in object-oriented programming,

- an operation on a private type,

- a task.

Each service has a signature which determines the number and the type of the service arguments. The set of signatures of the provided services constitutes the client interface, or class protocol, of the object. Similarly, the required interface of an object comprises the signatures of its required services. Communication between objects is only possible by service request. Services are similar to Smalltalk method calls or Ada rendezvous.

The growth of usage of object-oriented languages derives from the perceived benefits that the new technology offers the software developer. These include:

- modular architectures which are easier to design and maintain,

- prototyping life cycle for the validation of the user requirements, and

- reusable code.

The basis of these benefits is founded on the following technical characteristics:

- Data encapsulation which hides data storage behind an interface and binds it together with methods (operations) for accessing and manipulating the storage. Such units are called objects.
• Inheritance which allows for the incremental definition of objects behavior.

• Polymorphism which allows distinct objects to respond to the same operation.

Inheritance is a relationship between classes which allows a new class to be defined as a specialization of another, its parent. The inheritance relation may be shown as a tree of class interfaces. A restrictive but important use of inheritance occurs when it is used to provide subtyping. Subtyping satisfies substitution, i.e. class $B$ is a subtype of class $A$ if instances of $B$ can be substituted for instances of $A$ wherever they are used. A less restrictive form of inheritance allows the redefining of methods. At the implementation level this provides a powerful code reuse mechanism.

Although it may be considered obvious, it is necessary to stress that object-oriented technology is new and immature. Despite the publicity, there are relatively few products on the market that have used it. All the experience goes to show that there is a new technology risk and project managers should act prudently [Colem91, Jacob93]. However, this does not mean that object-oriented programming is a wrong way of programming. Experience has shown that the more complex, or unfamiliar (with respect to an established practise), a new "idea" or product is, the more time is needed to get it accepted in the computer community. For instance, the UNIX operating system, and the relational data model have both taken more than 15 years to become widely accepted and used.

Object-oriented programming is, by nature, closer to reuse than any traditional way of programming. This approach supports flexibility in defining and composing reusable components. New classes inherit the facilities of specified existing classes, and a class may augment or replace any subset of the inherited data structures and operations.

Moreover, some object-oriented languages offer more towards software reuse, as they provide a built-in form of genericity. Genericity is the technique that allows a module to be defined with parameterized types. Type parameterization is the ability to define a type in terms of another, unspecified type. Versions of the parameterized type may then be created for several particular parameter types. Instances of the module are then produced, by supplying different types as actual parameters. This is a definite aid to reusability because just one generic module is defined, instead of a group of modules that differ only in the types of objects they manipulate [Meyer87].
A language supporting type parameterization allows specification of general container types such as `list`, where the specific type of the elements is left as a parameter. Thus, a parameterized class specifies an unbounded set of related types; for example, `list` of `int`, `list` of `char`, etc. Typical languages that provide genericity are Ada, LPG [Bert83] and CLU [Liskov81].

A general form of parameterized types can also be integrated into object-oriented languages that do not provide a built-in form of genericity. Stroustrup [Strou88] proposed such a general form for the C++ language. At the same time, Free Software Foundation created a library as part of GNU CC with parameterized classes [Lea91]. Within this framework a generic class can be defined having one parameter of type `<T>`. For example the generic class array (independently of the type of the elements kept in it) may be defined as:

```
class array {
    protected:
        <T>* element;
        int size;
    public
        int search(<T>);
    ...
```

**Example 1**

Note that this code can not be compiled since `<T>` is an unknown type for the C++ compiler. In order to use the specific class of array of integers, a preprocessing stage is needed which will substitute the `<T>` of the previous example with the type `int`.

Genericity may be distinguished between unconstrained genericity, whereby no specific requirement is imposed on generic parameters, and constrained genericity, whereby a certain structure is required.

In its simplest form unconstraint genericity may be seen as a technique to bypass the unnecessary requirements imposed by static type checking. Consider the example of a simple procedure for exchanging the values of two variables. Using the C++ language enhanced with type parameterization, this procedure is written as:
void swap(<T>* x, <T>* y) {
    <T> t = *x;
    *x = *y;
    *y = t;
}

A declaration as the above does not actually introduce a procedure but rather a procedure pattern; actually procedures will be obtained by instantiating the pattern with actually type parameters without any further restriction.

On the other hand, in the case of the constraint genericity the instantiation of the parameterized types does not always succeed. Assume that, we want to define a generic sub-program finding the minimum of two values. Using the same technique as above we may write the following function:

    <T> minimum(<T>* x, <T>* y) {
        if (*x <= *y) return(*x);
        else return(*y)
    }

However, such a function declaration is not always meaningful: it should only be instantiated for types <T> in which a comparison operator "<=" is defined.

Using this substitution technique, it is possible to maintain the efficiency of object-oriented code (as we make the substitution before compiling), while retaining the benefits of genericity. However, this genericity mechanism by itself, is not flexible enough because it cannot capture fine grain of commonality between groups of implementations of the same general data abstraction. This is because there are only two levels of modules: a) generic modules, which are parameterized and thus open to variation but not directly usable, and b) fully instantiated modules (specific modules) which are directly usable but not open to refinement.

Our conjecture is that in order to effectively reuse object-oriented code using genericity, software developers should depend on experiential knowledge gathered and stored in the repository while developing similar software components. Specialized methods can then be incorporated in order to achieve better support from the system for the provision of this knowledge. Next, the reuse framework is presented, and the current research efforts on software reuse are addressed.
2.4 Reuse Framework

Reuse is the process of using previously acquired concepts or objects in a new situation. Reusability is a measure of the ease with which one can use those previous concepts or objects in the new situation. Ideally, reuse is a matching process between old and new situations and, when matching succeeds, duplication of the old object’s action [Freem83].

From the software engineering point of view, software reuse pertains solely to the process of constructing software systems. Therefore, the use of some existing source code routine in the construction of a program is an example of software reuse, while repeated invocation of the same routine with different input values during the execution of the program is not considered an interesting form of reuse.

In a typical non-reuse based software development environment, requirements for certain deliverables, e.g., design, code or documents, are submitted to a development process which after a while, without regard for previous similar work, delivers a result.

In a reuse oriented software production, a useful and suitable piece of information developed in some previous activity is recognized and reused in the development process. It is as simple as that. The reuse leads to all the advantages mentioned in section 2.2. But even though reuse simplifies the development process in many ways, it creates new tasks and problems.

Formally, reuse-oriented software development assumes that, given the project-specific requirements \( \mathcal{R} \) for an object \( x \), we consider reusing an already existing object \( x_k \) instead of creating \( x \) from the beginning. Reuse involves identifying a set of reuse candidates \( x_1, \ldots, x_n \) from an experience base, evaluating their potential for satisfying \( \mathcal{R} \), selecting the best suited candidate \( x_k \) and, if required, modifying the selected candidate \( x_k \) into \( x \) [Basil91].

Figure 2.1 provides a graphical description of a reuse-based software development process [Huff91]. In principle, any kind of information involved in the software development process could be reused. But there are many trade-offs; it is not always true that everything that is reusable should also be reused.

Before anything can be reused, it has to be identified and created. After that, it has to be described and stored in some form of library (repository). This stage includes problems related to the transformation of reusable information into components. What is the proper size and form of
reusable components and how should they be classified? What are proper techniques and languages for describing certain aspects of components? How to guarantee their quality?

Another important decision that needs to be made, is the determination of the repository type. Is a relational database enough or is there a need for more powerful modeling?

The user of the reuse repository seldom has complete knowledge about all the components stored, and probably not even an exact understanding of what components he is looking for. This introduces the selection problem. It is the problem of deciding what existing software may be useful in meeting the design (or requirements) specifications. We assume that the repository of reusable components is very large, containing useful objects created by different people, and that it will be constantly evolving. It is not possible, even for an experienced developer, to fulfill the selection task with the help of just documentation mechanisms, index keywords, and/or a "browsing" facility.

The selection task becomes even harder when coupled with the adaptation problem. In many cases, it is not possible to find in the repository a component that fits exactly the developer's needs. However, there may exist a component which with minor modifications will finally meet the initial specifications. The problem is finding such a component, understanding it, and making the appropriate modifications.
2.4 Reuse Framework

Understanding the structure and interrelationships between various entities in a software component is essential to its adaptation and integration in a new application. It is also important in deciding whether a retrieved component is actually suitable in a particular new application. Since retrieval yields components that are potentially (but not necessarily) relevant, the developer must judge carefully whether they meet current needs.

Finally, the object’s repository evolution problem is a major concern, influencing directly the selection and adaptation problems. The entire software production cycle should be integrated and organized in such a way as to improve the quality of the repository as well as its maintainability.

There is great diversity in the software engineering technologies that involve some form of software reuse. However, there is a commonality among the techniques used. For example, software component libraries, application generators, and generic software tabletes all involve selecting, and adapting software artifacts. According to Krueger’s classification [Krueg92] eight different models of software reuse can be distinguished:

1. High Level Languages.

They provide the medium for most of today’s software development. Each language provides its own operational semantics, which typically serves as an abstraction level above the assembly language architecture of the machine. High level languages are typically not regarded as examples of software reuse. However, viewed from the perspective of software development prior to their existence, the goals and achievements for high level languages have strong parallels to the current day aspirations of software reuse researchers.

Selection: In this model, selection is relatively simple since the number of different reusable artifacts is small enough for a single programmer to master in a matter of days or weeks. For novices in a particular language, the selection of language constructs is aided by the use of language reference manuals and possible tutorial examples.

Adaptation: Since the application developer does not actually reuse code when programming in a high level language, the adaptation phase is missing.

Evolution: Similarly, since no code is reused, there is not a repository to be evolved.


The goal of design and code scavenging is to reduce the cognitive effort and the number of time required to design, implement and debug a system, by copying as much of the development as
possible from analogous systems that have already been designed, implemented and debugged.

Selection: When a software developer recognizes that part of a new application is similar to one previously written, a search for existing code may lead to code fragments that can be scavenged.

Adaptation: A programmer adapts a scavenged code fragment by manually editing it. The programmer must thoroughly understand the lowest level details of the code, in order to adapt it correctly to its new context.

Evolution: Previously written code fragments are stored in the application developer’s file system, which can be considered as a very simple form of repository. Although the size of this repository increases over time (the application developer creates new fragments of code), its quality is not improved, since the selection and adaptation tasks become harder.


The goal of the off-the-shelf source code components is similar to that of informal code scavenging and furthermore a more thorough and systematic approach is attempted for creating and using components, such as a library or catalog of components. Examples of collections of reusable functions include statistics libraries such as SPSS and numerical analysis libraries as IMSL [Levine89].

Selection: Selection is easier in domain specific libraries because components can be classified, organized and retrieved using well defined properties of the domain. In a general-purpose component library, it is dependent on the quality of abstraction, classification and retrieval schemes. Describing components behavior with a specification language can help software developers to select among a small number of alternatives, but this approach is not suitable for a large library.

Adaptation: Similarly with code scavenging, one can adapt reusable components by directly editing the source code. However, this approach has the same drawbacks with code scavenging.

Evolution: On the same way as in code scavenging, the selection and adaptation tasks become harder as the size of the component’s catalog increases.

4. Program Schemas.

Reusable program schemas formally extend the notion of reusable software components. Techniques such as parameterization and classification are integral parts of program schema models rather than ad-hoc extension to software modules. Program schemas therefore offer a more systematic means of constructing software with reusable artifacts. Although the goals of program schema technology are similar to those of reusable components, the emphasis in program schemas is in reusing abstract algorithms and data structures, rather than reusing source code. A
representative schema technology has been used in the PARIS system [Katz89].

Selection: Classification, search schemes and multiple abstraction exposition forms, analogous to The TTL Data Book, offer promise as practical selection techniques for reusable program schemas. The issue of scale will make large schema libraries more difficult to use than the hardware analog. Automated assistance such as hypertext databases and theorem provers can help for schema selection.

Adaptation: The schema is conceptually one large, generalized artifact, which is specialized either by substituting language constructs, code fragments, specifications, or nested schemas into parameterized parts of a schema, or by choosing from a predefined enumeration of options. Different implementations may be taken by applying different specializations to the same schema. However, it is not possible to further adapt a specialized schema.

Evolution: There is not a systematic way for the construction of new program schemas. Furthermore, their developers are being forced to make a large number of decisions about their future usage during their creation.

5. Application Generators.

Application generators operate much like programming language compilers: input specifications are automatically translated into executable programs. They differ from traditional compilers in the sense that, the input specifications are typically very high-level special-purpose abstractions in a very narrow application domain.

The goal of software development with application generators is to eliminate the costly software design and implementation phases of conventional system construction. When compared to program schemas, where the reuse of algorithms and data structures is stressed, application generators go one step further by reusing high level system design and organization, and by automating the selection of algorithms and data structures. Reports on application generators can be found in [Cleav88, Levy86, Horow89].

Selection: Application generator libraries have not received much attention in the literature. The parallel between software schemas and application generators suggests, however, that library techniques could be used to select among a collection of application generators.

Adaptation: Since high-level abstractions from an application domain are mapped into executable software systems, no further adaptation is possible.

Evolution: As no code is actually reused, there is not a repository to be evolved.
6. Very High Level Languages (VHLLs).
Software development with VHLLs is strongly analogous to development using High Level Languages (HLLs) and their compilers. Both VHLLs and HLLs provide a notation and semantics for expressing a general form of computation. Also, they both provide compilers which take programs written in their language as input, make tests for correctness, and create executable object programs if a certain threshold of correctness is achieved. HLL specifications may be an order of magnitude more succinct than the corresponding HLL implementation, and likewise VHLL specifications may be an order of magnitude more succinct than corresponding HLL implementations.

VHLLs resemble application generators in that executable systems are created directly from high level specifications. VHLLs differ from application generators in that application generators are often restricted to specifying computation in a specific domain, while VHLLs use specifications which express a more general notion of computation. Examples of VHLLs are the SETL [Dubin87], PAISLey [Zave86], and MODEL [Cheng84].

Selection: Selection can take place at two levels: (1) selecting a VHLL that is most appropriate for a particular application and (2) selecting the language constructs that best represent the application.

Adaptation: Similarly with applications generators, VHLL create executable systems thus no further adaptation is possible.

Evolution: As no code is actually reused, no evolution is possible.

7. Transformational Systems.
The transformational approach to software development separates software development into two phases: a) specifying the behavior of a software system in a high level specification language, typically an executable VHLL and b) applying transformations to the high level specification such that execution efficiency is enhanced, while the specified computation performed by the system remains fixed. Examples of transformational systems are PADDLE [Balzer87], Glitter [Fickas85], and CASCENT [Garlan92].

Selection: Reusable transformations can be stored in a library. Selection of transformations from the library can be enhanced by rule-based analysis that identifies sequences of transformations that satisfy high-level transformation goals.

Adaptation: Transformations are typically applied without modifications. Therefore adaptation is typically not an issue for transformation reuse.
Evolution: On the same way as before, no evolution is possible.

This model deals with the formal reuse of software artifacts at the system design level. Design structures represent a significant design and implementation effort that can be reused as a whole. One approach to reusable design structures is analogous to program schemas. From this point of view, reusable design structures are essentially very large program schemas, or design schemas. Design schemas are reused by filling in undefined slots with code fragments, program schemas, or nested design schemas.

Design structures are also analogous to application generators in that large-scale system designs are reused. Application generators, however, are typically stand-alone systems with implicit architectures, whereas design structures can often be explicitly specialized and integrated with other structures to create many different composite designs.

Examples of reusable design structures include database subsystems that are specialized and reused in different applications; rule based and blackboard architectures for expert systems; and adaptable user interface architectures [Garlan90, Lane90, Shaw91].

Selection: The analogy between program schemas and design structures suggests that library techniques for doing classification, search and exposition could be used to select among a collection of reusable design structures.

Adaptation: Software developers can specialize design schemas to produce implementations that match the performance requirements of an application. Also further adaptation at the implementation level is possible, but this demands a deep understanding of the implementation components behavior.

Evolution: There is not a systematic way for the construction of new design structures. Their developers are being forced to make too much decisions about their future usage during their creation.

Frequently, strong relationships exist among the various types of software components, such as among requirements and designs, design objects and algorithms, and data types and their implementations. This makes it feasible to automatically select some reusable components based on the selection of others. By assembling the various types of reusable components into a special reuse library that encodes their relationships, support can be provided for:
• organizing and managing software and information about software development,
• finding information concerning previous software developments that may be relevant to new projects, and
• aiding the gradual evolution of software and software components.

This provides a "wide spectrum" approach to the problem of software reusability and offers a considerable gain in the software development process [Bigger87].

2.5 Summary

Reuse is the use of previously acquired concepts or objects in a new situation. From the software engineering point of view, software reuse pertains solely to the process of constructing software systems.

Effective reuse of knowledge, processes, and products from previous software developments, can increase productivity and quality in software projects.

Object-oriented programming is, by nature, closer to reuse than any traditional way of programming. Inheritance which allows for the incremental definition of object behavior, and genericity which allows for a module to be defined with parameterized types, provide a definite aid towards software reuse. However, in order to achieve actual software reuse, problems like selection, adaptation and repository evolution need to be solved.

Chapter 3 presents a taxonomy of the current efforts on software reuse, and addresses their limitations.
Chapter 3

Related Research - Limitations

3.1 Introduction

Reuse work intensified significantly in the late 1980's. Several advances were made in library systems, classification techniques, the creation and distribution of reusable components, reuse support environments, and corporate reuse programs. In spite of this, a consistent complaint voiced at workshops and conferences, was that reuse was not delivering on its promise to significantly increase productivity and quality.

In 1988, Vic Basili broadened the definition of reuse to include the "use of everything associated with a software project, including knowledge" [Basil88]. This new perspective has opened doors to research and has contributed to the recognition that the reuse problem is ubiquitous.

This chapter presents a taxonomy of the different repository organization and retrieval methodologies, and addresses their limitations. In addition, a new reasoning methodology called Case-Based Reasoning, that appears to be very promising for modeling and reusing programming knowledge, is compared to the traditional rule-based reasoning. Recent efforts on software reuse that employ special forms of Case-Based Reasoning are also presented.
3.2 Repository Organization Methodologies

Reuse in a software development environment implies the existence of some medium to store and subsequently locate the components for reuse (commonly called repository). The different methodologies used for organizing a repository include: a) library cataloguing, b) hierarchical classification schemes, c) faceted classification schemes, d) hypertext form, e) conceptual/semantic organization, and f) hybrid organization. Next, follows a brief analysis of these methodologies.

3.2.1 Library Cataloguing

Repository technology has its technical roots in library support inside software development environments. Procedural libraries of mathematical functions, graphics routines and several other domains have been available for many years. Many of these libraries have been carefully studied, profiled and optimized [Korson92].

Libraries often come together with a language compiler. For example most FORTRAN compilers have a particularly strong set of mathematical library routines. Other procedural libraries are sold as independent commercial products. The most well known of these products include IMSL [Levine89], and NAG [Hopking88].

As programming languages evolve, and object-oriented programming becomes popular, object-oriented libraries (libraries of reusable classes) are available, including:

The Classix Library [Classix90]. It is a general purpose library, divided into four major areas: Smalltalk collection classes, mathematical classes, general data structures, and primitive classes. The Classix Library includes a macro facility, which simulates a generic mechanism so that container classes can be instantiated to hold the appropriate objects.

The Eiffel Library [Meyer90]. It is a part of the Eiffel language distribution. The library is divided into eight sections: kernel library, data structures library, iterators, lexical, parsing, winpack, graphics and a support library. The support library includes classes such as a browser class, a debugger class and a memory management class. It also uses features of the Eiffel language, such as genericity and multiple inheritance.
The *NIH Class Library* [Gorlen90]. It is a public domain library developed by Gorlen. Formerly called the OOPS Library, this work is written in C++ and has been ported to a wide range of machines. The Library uses a single class as a root class from which all other classes must be derived.

A major challenge in developing large reusable object-oriented libraries is in finding concise semantic abstractions for components. Although the source code for component implementation serves as the abstraction realization level, it is unreasonable to present a huge library of source code components to a user and expect the source code to serve as the only exposition for the component behavior. What is required, is an abstraction specification level that describes the behavior of the components in the library in a more succinct way.

Furthermore, being able to use a collection of reusable components without having to do any adaptation, requires that the creators of the library have the foresight and means to produce a set of components that suit all possible user needs. Currently this is possible for domain specific components but not for general purpose components.

### 3.2.2 Hierarchical Classification Schemes

This organizational methodology is similar to that of traditional database systems. It usually follows a hierarchical or graph-oriented data model. Arbitrary nodes may be introduced, which are connected by arbitrary relationships. Both nodes and relationships may have attributes and also have a type, or a category. Examples of environments that employ such an organizational methodology are the Arcs environment [Schef89], PCTE/PACT [PCTE86, PACT88], CAIS [CAIS88], Genesis [Bator89], and Avoca [OMall90].

The main advantage of a hierarchical classification scheme is its simplicity. However, it is inadequate for modeling general software artifacts, including experience involved in the software development process. The result of such an inadequacy, is the inability of the target system to provide assistance to the application developer during the adaptation phase. Furthermore, as the size of the repository increases over time, the selection and the adaptation tasks become harder.
3.2.3 Faceted Classification Schemes

This organizational methodology was first proposed by Prieto-Diaz [Diaz89]. A facet is a viewpoint towards software components. Viewpoints may include the functions the components perform, the objects they manipulate, the system types to which they belong, and so on. The value for a facet of a component is called the facet descriptor. The characteristics of a component may be described using its facet descriptor, and a component may be understood through its facet descriptor.

The user specifies a desired function with a facet using the keywords of a fixed vocabulary. For retrieval the library is searched through and every component specification is offered which matches the target facet.

An important component that is usually used in faceted classification scheme organizations is the conceptual distance graph. Conceptual distances between items of each facet are used to evaluate their similarity, which is used in turn to evaluate the similarity between required software specifications and available components. Typical examples that use a facet classification scheme are the GTE Data Service's AMP (Asset Management Program) [Diaz91], and CART [Hsian93].

Although this approach allows an explicit similarity measure, there are also some disadvantages. First, the vocabulary for specifications is fixed and also the grain size to describe properties. Consequently, it is a hard process to construct such a vocabulary, because all possible facet values should be predicted. Secondly, the keywords which are available to the user are only the bottom nodes in the related conceptual graphs. Therefore, with the exception of the "*" keyword, which states irrelevance of this feature, no abstract descriptions of features are possible. This relates to all or nothing paradigm. Therefore, all similarity information must be directly encoded in the conceptual distance graphs.

3.2.4 Hypertext Form

A software hypertext system is a system that integrates a hypertext mechanism with software engineering tools. Software hypertext systems provide several features which allow users to view information related to a software system in an integrated manner within and across projects [Rada92]. Users of the system enter software process information into predefined (but re-definable) nodes that are internally treated as files.
The capabilities of a software hypertext system enable software engineers to document their software process in ways that support: a) analysis of the consistency and completeness of formalized document nodes, b) intra and inter-document traceability, c) formatting and display, d) indexed or query-driven browsing, e) documentation standards, f) multi-version documents with/without sharable annotations, and g) on-line software inspections and walkthroughs [Harri92, Harri93].

In general software hypertext systems are practical but passive, providing storage facilities for all the phases of the software life cycle. Examples of software hypertext systems include commercial systems like Guide [Brown87], and research projects like DIF [Garg90], Sodos [Conki87], and Neptune [Bigel88].

3.2.5 Conceptual / Semantic Organization

Semantic data models came about in the mid-seventies in response to the perceived need for better modeling tools to "capture more of the semantics of an application" [Godd79].

The research community working on data modeling further broadened its horizons in the early eighties, by noting similarities in goals with programming language research focusing on abstract specification of programs and knowledge representation ideas going beyond semantic networks. Conceptual modeling was introduced as a term reflecting this broader perspective. Since the early eighties conceptual modeling has found applications beyond capturing the meaning of a database, including modeling organizational environments, modeling software development processes, or just plain modeling some part of the world for purpose of human communication and understanding [Mylop91a]. Some representative research projects that employ a conceptual/semantic organization are:

IDEA. IDEA is a knowledge-based environment for supporting the reuse of design components and other early software process concepts [Lubar86]. Design knowledge is captured in the form of abstract reusable design schemas and refinement rules that can be applied to solve classes of related design problems. Domain-oriented data object definitions are organized into a type lattice based on the definition of data types as sets of constraints. Furthermore, the use of constraints and data type names provides a reusable domain vocabulary with which the users can describe the appropriate data objects.
PRACTITIONER. The research project PRACTITIONER (ESPRIT Project P1094 [PRACT85]) started in 1986 with the aim of improving the methods for describing reusable software designs as well as the techniques for classifying, storing and retrieving them. The entire approach of the project was based on the notion of the design concept, a term that has been defined as follows: A concept is an abstract task, described by its purpose (and/or goal), the related objects, related tasks and/or the functional principles of the underlying mechanism (which will be typically, but not necessarily, of an algorithmic nature). Such concepts have been extracted from existing designs (or programs) by experienced designers and described by means of a questionnaire, a structured combination of various existing descriptive techniques [Bold89].

ESF-ROSE. The ESF-ROSE project [Moine90, Moine91] is a sub-project of the Eureka Software Factory Project [Rockwe92]. The aim of ESF-ROSE is: 1) to analyze and define the concept of reuse of software elements, 2) to develop an environment supporting reuse of software elements (the ESF-ROSE system), and 3) to open new research and to go beyond the state of the art with more theoretical work. The ESF-ROSE system was designed as a generic and extensible system. That means that the ESF-ROSE system is able to support reuse of any kind of software information produced within a software factory (code or specifications, but also memos and contracts). The genericity of the ROSE system is achieved mainly by means of a generic conceptual model (the generic Component and System Model) which can be instantiated with the kind of software elements to be reused and the way to reuse them. The genericity of the system allows to have early prototype for a reuse environment, and to incorporate the results of the theoretical thread when there are available.

Other systems that use such an organization methodology are, the "Software Knowledge Base" [Meyer85], KADS [Wielin91], and 3C Model [Frakes90b].

3.2.6 Hybrid Organization

This category includes multi-dimension organizations as workable combinations of the organizational methodologies described above. Projects that could be classified in this category are:

ISHYS. It is a software hypertext system which is knowledgeable about its environment and can use such knowledge to assist in the software process [Garg89]. This knowledge is partly embedded in the design of ISHYS and partly defined during its use. ISHYS' intelligent behavior springs
from its ability to (1) automatically derive relationships between hypertext nodes, (2) automatically determine attributes of hypertext nodes, and (3) coordinate and schedule agent tasks in the software life cycle.

ISHYS has been designed as a distributed knowledge-based system with four major components: 1) a knowledge manipulation system with access to different pockets of knowledge for each agent about which ISHYS knows; 2) a software hypertext system (DIF) that maintains tangible products of the software life cycle; 3) software engineering tools used to engineer and manage information contained in the software hypertext; and 4) an interaction control system that interacts with users and controls user access to the knowledge maintenance system, the software hypertext system, and software engineering tools.

ITHACA. The purpose of the ITHACA Application Development Environment [Ader90] is to support application development through reuse of development information regarding both available executable software and development information, such as requirements, scripts, design documents, design decisions and motivations. Achieving reusability of not just software but of previous experience is an essential activity within this approach. This, in turn, implies the need for a different kind of software life-cycle in which the long-term development and evolution of reusable software proceeds in tandem with the short-term development of specific applications.

To enhance the effectiveness of this Application Development Environment, an object oriented approach to application development is considered in ITHACA, and a centralized repository of development information and reusable components is the core of the environment. This repository, called Software Information Base (SIB) [Const93], provides the underlying mechanisms for storing and representing descriptions. Descriptions encapsulate properties of software components and knowledge concerning application domains for use by other tools. A prototype of SIB has been built using the Telos knowledge representation language.

The information in SIB is organized into Generic Application Frames (GAFs), which describe how specific applications can be constructed from the available components. In order to ensure that it will be possible to map the application requirements to a GAF, it is essential that requirements collection and specification start by using the SIB as a basis for specifying the application.

Next, follows a brief presentation of the different methodologies used for retrieving reusable software artifacts from a repository.
3.3 Retrieval Methodologies

A taxonomy of the alternative retrieval methodologies should include: a) formal specifications, b) database query retrieval, c) knowledge-based reasoning/deductive tools, d) hypertext retrieval, e) information retrieval, and f) hybrid approaches. A brief analysis of these methodologies can be found in the sequel.

3.3.1 Formal Specifications

Formal methods and transformations have been proposed to support software reuse in several ways. They can provide "links" between program code and abstract specifications, which can be manipulated in various ways. These links provide a high degree of confidence that the components in the repository meet their specifications, and mean that new programs constructed from these components will be correspondingly reliable. Inverse engineering (the application of formal methods to extract specifications from code) can enable the reuse of existing code by extracting reusable components and provide their specifications. Transformations can also be used in the forward direction to semi-automatically develop new code from existing or modified specifications [Ward91].

A representative research project that uses formal specifications, is the ESF-ROSE project described above. The retrieval process of the ESF ROSE system can be sketched as follows:

1) A specification of an Ada program to be coded is written either as a hierarchy of Ada package or as a formal algebraic specification.

2) This target specification is compared with the specifications of the reusable components stored in the reuse library. The comparison is performed modulo a renaming: it is not required the reusable component to export exactly the names defined in the specification, it is rather required the possibility to build a stub which only exports the data types and the sub-programs required by the specification.

3) The user chooses a component among the ones that conform to the target specification.

4) The user is asked by the so-called adoption tool to choose a good renaming among the many possible renamings that can exist between the target specification and the component
3.3 Retrieval Methodologies

To be reused.

The first instance of the ESF-ROSE system is based on results from the theoretical thread. This instance follows the *as it is reuse* paradigm, i.e. the reused code must not be modified by the user. But, a reusable component seldom fits exactly the new needs. This problem is partially solved by the Ada, as its instances support non-exact matching between the target specification and the reused component. However, in order to make this, a deep understanding of the component functionality is demanded. Furthermore, there is not a systematic way to have a controlled evolution of the repository, in order to increase its quality without side-effects to the efficiency of the selection process.

Similar retrieval methodologies are also used in Z [Morran89], VDM [Jones86], and PARIS [Katz89]. The problems addressed in systems employing the formal specifications retrieval methodology are related with the complexity of formal specifications at network system level. Furthermore, as the repository size increases over time, the selection and the adaptation tasks become even harder.

### 3.3.2 Databases and Database (Query) Retrieval

This retrieval methodology is applied to repositories that employ a hierarchical classification scheme organization. Components are located using queries similar with that used in the traditional database systems. Examples of environments that employ such a retrieval methodology are the Arcs environment [Schef89], PCTE/PACT [PCTE86, PACT88] and CAIS [CAIS88].

The most serious limitation addressed in these systems, is that there is not any assistance to the application developer to locate similar components in the repository that with a proper adaptation may fit into his needs. Furthermore, as the repository size increases over time, the selection and the adaptation tasks become even harder.

### 3.3.3 Knowledge Bases and Reasoning / Deductive Tools

Knowledge-based reasoning is applied to repositories that are organized as knowledge bases. Queries are evaluated through reasoning againsts the knowledge base state, i.e., the set of the stored or deliverable (through deductive rules) facts. There are several research projects that
employ such a retrieval methodology, including:

**IDEA.** It includes a schema selection strategy based on the notion that it is frequently easier to describe a function’s data objects than to describe the actual function itself. The type lattice provides a means for identifying user-specified data objects. A modified type-checking algorithm is used to identify suitable schemas that satisfy the constraints of user specifications. The selected schema is then instantiated with the properly inferred types and incorporated into the design. Constraints in the schema instance are used by IDEA to select and apply the appropriate specialization and refinement rules.

IDEA selects schemas and applies specialization and refinement rules as users provide specifications. Thus, IDEA supports a refinement paradigm of software development in which specification and design occur in parallel. This permits the design process to be used as a means of checking the completeness and consistency of specifications. In fact, IDEA may guide the user through the process by requesting additional information, as it becomes necessary, for the refinement of the design.

**PRACTITIONER.** The questionnaires of this project mentioned above, are stored and administered by a set of software tools, the PRESS (Practitioner REuse Support System) [Elzer89] with capabilities: 1) fast analysis of the raw material (e.g. design documentation, functional specifications or program texts) for relevant terms; 2) embedding of the terms found into fields of terms (related, broader or narrower terms, synonyms, etc.) and support for the user with respect to the interpretation of search terms that are precisely those already contained in the tool’s base of material; and 3) search for and presentation of concepts found in the tool’s base of material with special emphasis on the integration of the PRESS toolset into a standard software development environment.

Similar retrieval techniques are also used in LaSSIE [Devan91] and KBEmacs [Wat86]. The problems addressed in systems employing traditional reasoning techniques, are related with the complexity of the reasoning process, as analytically explained in section 3.6. In addition, given a problem that is outside system’s original scope the system often can’t render any assistance.
3.3.4 Hypertext Retrieval

This retrieval methodology is employed in repositories that use a hypertext mechanism such as DIF, and Neptune [Bigel88]. In general software hypertext systems are practical but passive providing storage facilities for all the phases of the software life cycle. Therefore, retrieval and reuse of the information related with the development of an application, may require navigation in a large number of paths, and moreover it may never be found. This retrieval strategy becomes even harder if coupled with the adaptation problem.

3.3.5 Information Retrieval

This retrieval methodology is used in repository organizations based on the faceted classification approach. The user specifies a desired function with a tuple using the keywords of the fixed vocabulary. For retrieval the repository is searched through and every component specification is offered which matches the target tuple. In further steps, the user can restrict the area until the desired function is determined.

Furthermore, conceptual distances between items of each facet are used to evaluate their similarity, which is used in turn to evaluate the similarity between required software specifications and available components. Although reported to be very effective in retrieving software components for reuse, the faceted approach is labor intensive. Construction of a conceptual graph has not been formalized yet. Conceptual distances are assigned based on experience, intuition, and common sense.

3.3.6 Hybrid Approaches

A retrieval methodology is characterized as hybrid if there are more than one ways to locate a component in the repository. As an example consider the ISHYS software hypertext system mentioned above. Since its repository is organized in a way that combines both a hypertext organization and a knowledge-based one, a component may be located either using a hypertext retrieval methodology, or knowledge-based reasoning.

Also, the ITHACA retrieval methodology follows the hybrid approach. Different components related to different phases of the software development process, may be located using a number
of alternative ways provided by a special purpose tool called the *Selection Tool* [Const93].

### 3.4 Related Research Limitations

As already discussed previously, reuse work intensified significantly during the last five years. Several advances were made in library systems, classification techniques, the creation and distribution of reusable components, reuse support environments, and corporate reuse programs. In spite of these efforts, reuse has not delivered yet on its promise to significantly increase productivity and quality, due to certain limitations of the methodologies presented in the previous sections. Next, we address these limitations with respect of the issues of selection, adaptation, and repository evolution.

1. **Selection.** Concerning the selection problem, most of the efforts presented in the previous section offer quite good retrieval strategies for cases that the target reusable artifacts exist in the repository (i.e., for instances following the "as it is reuse" paradigm). Certain limitations that should be addresses, are related with the complexity of the retrieval process. More specifically, hypertext retrieval used in Guide [Brown87], DIF [Garg90], Sodos [Conk187], and Neptune [Bigel88], may require navigation in a large number of hypertext paths, and moreover the target hypertext node may never be found. Consequently, hypertext retrieval by itself can not be considered a good candidate for a successful reuse environment. On the other hand, a hypertext mechanism could be very useful for a better understanding of a limited set of software artifacts that have been located using a more powerful retrieval methodology.

   The complexity of the formal specifications at network system level, is the more important limitation addressed in systems like ESF-ROSE [Moine91], Z [Morr089], VDM [Jones86], and PARIS [Katz89]. As a consequence, the performance of these systems becomes inappropriate as their repository evolves over time and new formal specifications of reusable components are stored in it.

2. **Adaptation.** Although most of the retrieval methodologies presented in the previous section offer quite good retrieval strategies, their ability to locate similar artifacts in the repository that with proper adaptation may finally fit the current needs, is very limited.
In fact, in systems such as Arcs [Schef89], PCTE/PACT [PCTE86, PACT88] and CAIS [CAIS88], where database retrieval is used, and in systems such as ESF-ROSE [Moine91], Z [Morrano91], VDM [Jones86], and PARIS [Katz89] where formal specifications are employed, it is not even possible to locate similar components that with proper adaptation may finally fit the current needs. Also, as already discussed previously, hypertext retrieval is a very hard process, and there is no guarantee that the target component (or a similar one) will be finally located. However, being able to use a collection of reusable components without having to do any adaptation, requires that the creators of these repositories have the foresight and means to produce a set of components that suit all possible user needs. Currently this is possible only for domain specific components but not for general purpose components.

On the other hand, in systems such as the GTE Data Service's AMP [Diaz91], and CART [Hsian93] where a faceted classification scheme and information retrieval are employed, a similar component may be possibly located. Although traditional information retrieval mechanisms do not allow partial matches, this is accomplished by using the similarity information encoded in their conceptual distance graphs. However, these approaches are still labor intensive. Construction of a conceptual graph has not been formalized yet. Conceptual distances are assigned based on experience, intuition, and common sense. Furthermore, being able to locate a similar component, does not fully solve the adaptation problem. A deep understanding of this component as well as additional programming knowledge are necessary, in order the appropriate modifications to be applied. However, the faceted classification mechanism is not strong enough for representing such types of knowledge. Therefore, the systems' support during the adaptation phase is very limited.

Finally, in systems like IDeA [Lubar86], PRACTITIONER [PRACT85], ESF-ROSE [Moine91], KADS [Wielen91], and 3C Model [Frakes90b], the systems' support during the adaptation phase is also limited. Although, a conceptual/semantic organization provides a strong basis for expressing different types of knowledge, the adaptation problem has not been thoroughly studied.

3. Repository Evolution. The repository evolution problem has been a minor concern for the majority of the research efforts presented in the previous sections. Specifically, object-oriented libraries as Classix [Classix90], Eiffel [Meyer90], and NIH [Gorlen90] do not provide mechanisms for their expansion. An application developer may create new classes, but these classes can not become directly available to the rest of the application developers.
On the other hand, in systems such as Arcs [Scheff89], PCTE/PACT [PCTE86, PACT88], CAIS [CAIS88], Genesis [Bator88], and Avoca [OMall90], that use hierarchical or graph-oriented data models, an evolution is accomplished with the insertion of new records that describe new software artifacts. Similarly, hypertext systems may be evolved with the insertion of new hypertext nodes that describe new software artifacts. However, as already discussed, the selection task becomes harder as the size of their database increases.

In systems, like the GTE Data Service’s AMP [Diaz91], and CART [Hsian93] where a faceted classification scheme is employed, the evolution of the repository is accomplished with the insertion of new facets. Additionally, whenever a new facet is appended to the repository, the conceptual distance graph has to be reorganized in order to include new similarity information among this new facet, and the old ones. However, as already discussed, the construction of a conceptual graph has not been formalized yet. Conceptual distances are assigned based on experience, intuition, and common sense. Thus, the application developer has to manually assign the new conceptual distances among a new graph node and the old ones. This is a very hard task, and becomes even harder as the size of the conceptual distance graph increases.

Finally, in systems such as IDeA [Lubar86], PRACTITIONER [PRACT85], ESF-ROSE [Moine91], KADS [Wielen91], and 3C Model [Frakes90b], where a conceptual/semantic organization is employed, new software artifacts may be appended, using standard classification techniques. The most serious limitation addressed in systems employing traditional reasoning techniques, is related with the complexity of the reasoning and learning process, as analytically explained in section 3.6.

### 3.5 AI-based Approaches for Software Reuse

Effective reuse of software components requires a good collection of components and knowledge about how to locate, to adapt them, and to create new ones. Indeed, experienced software engineers generally benefit from this kind of expertise [Adel85]. Other studies indicate large differences in the abilities of different software developers [Sackm68, Curt81], and a number of techniques have been proposed to take the best advantage of the more experienced software engineers [Brook78].
The research efforts presented in the previous sections, have taken almost no advantage of this kind of expertise. The result is that although most of them offer quite good selection strategies, their ability to locate similar code in the repository that with the proper adaptation may finally fit the current needs, is quite limited. Furthermore, the selection problem itself becomes harder, as the repository evolves over time.

The application of techniques and methods from artificial intelligence to software engineering is one mechanism through which reusability of software might be achieved. By abstracting and encoding the expertise of experienced software engineers into knowledge bases together with software components, system developers can gain effective access to the artifacts in the software repository as it evolves over time.

Next, we address some research efforts that try to take the best advantage of this kind of expertise. These efforts employ a hybrid organization (one of the combined organizational methodologies is that of conceptual/semantic organization), while employing rule-based reasoning during the retrieval and adaptation phases.

**FMS.** Flexible manufacturing systems (FMS) combine many desirable features: modular architecture, integration of heterogeneous methods and tools, configuration and reconfiguration capabilities, and wide automation under human control. FMS come to age in the software industry through the definition of integrated software engineering environments, based on those concepts of flexibility and continuous improvement [Cusum89].

Basili has proposed an organizational framework that separates the project specific activities, from the reuse packaging activities, with process models for supporting each of the activities [Basil89]. The framework defines two separate organizations. One organization is project-oriented. Its goal is to deliver the systems required by the customer. It is called the project organization. The other organization, called experience factory, has the role of monitoring and analyzing project developments, developing and packaging experience for reuse in the form of knowledge, processes, tools and products, and supplying it to the project organization upon request.

**AIRS.** The AIRS system [Ostert92] is essentially a hybridization between the faceted index and semantic network approaches. The domain information inherent in the facets is used to reduce the rigidity and the laborious creation of a semantic structure. A hierarchical frame is used to
maintain information about which of the objects in the reuse libraries have which features, how these objects are grouped, and how the features are related. In addition the features of the frame system are used to facilitate the integration of new components into the AIRIS system, allowing a programmer to bootstrap its knowledge structures from a basic set of existing components. One important aspect of AIRIS, is the ability to reason heuristically about the desired components and components in the existing knowledge base.

Rule-Based reasoning has been attractive as a method for building intelligent reasoning systems. But, when modeling software development experience using rules, a heavy price need to be paid. Next, follows a comparison between rule-based reasoning and an alternative reasoning methodology called Case-Based Reasoning.

3.6 Rule-Based Reasoning vs Case-Based Reasoning

In the last few years research into the general area of case-base reasoning has begun to flourish [Kolod88, Schank89, CBR91]. Case-based reasoning has been proposed as a more psychologically plausible model of the reasoning of an expert, than the more fashionable rule-based reasoning systems that are the basis of the expert systems that began to be commercially available in the 1980's.

A rule-based expert system solves problems by taking an input specification (or developing one through a question-and-answer dialog with the user) and then "chaining" together the appropriate set of rules from the rule base to arrive at a solution. Given the same exact problem situation, the system will go through exactly the same amount of work to come up with a solution. In other words, rule based expert systems do not inherently learn. In addition, given a problem that is outside the rule-based system's original scope (for example, a new problem type within the same domain), the system often can not render any assistance.

Another problem with purely rule-based systems is that knowledge of the system ends up being scattered into hundreds of individual pieces. This seems fairly improbable as a way to model of the very large diverse bodies of knowledge that people have, partly because the normal effect of such an organization is to make a system run slower the more facts it knows. It seems much more plausible that knowledge is organized so that the related items are "near" each other.
Finally, rule based systems are very time-consuming to build and maintain because rule extraction from experts is labor-intensive, and rules are inherently dependent on other rules, making the addition of new knowledge to the system a complex debugging task [Barlet91].

Case-based systems operate in a very different way. Given an input specification, a case-based system will search its case memory for an existing case that matches the input specification. If such a case exists in the case memory, the case-based system will find it and it will go directly to its solution, making it possible to provide solutions to potentially complex problems quickly. If on the other hand, such a case does not exist, the case-based system will retrieve a case that is similar to the input situation but not entirely appropriate to provide a complete solution.

The case-based system must then find and modify small portions of the retrieved case that do not meet the input specifications. This is called case adaptation. The result of case adaptation is a completed solution, but it also generates a new case that can be automatically added to the system’s case memory for future use.

Case-Based Reasoning (CBR) has been attractive as a method for building intelligent reasoning systems because it appears relatively simple and natural. CBR as a learning paradigm has several advantages [Kolod90]: Firstly, there are several performance enhancements as it provides: shortcuts in reasoning, the capability of avoiding past errors, and the capability of focusing in the most important parts of a problem first. Secondly, learning can be eased since CBR does not require a causal model or a deep understanding of the domain, though either of these provides better performance. Thirdly, individual or generalized cases can also serve as explanations; these explanations are trivial to generate, and probably more satisfactory than the chains generated by expert systems.

CBR systems solve complex problems like planning, scheduling, and design, by finding a similar successful past plan, schedule, or design, and modifying it to meet all the current problem’s needs. This assumes that the system has a decent set of cases from which to reason. If we can’t derive any value from past plans or designs, CBR is not a good idea. Therefore, the decision case-based or rule-based reasoning is domain-specific. CBR approaches work well in domains that are poorly understood, because the system does not need to know why something worked in the past. On the other hand the fact that CBR does not require a causal model or a deep understanding of the domain, may be a disadvantage for well understood domains, if we cannot find an
appropriate case representation.

Since most retrieved cases that the system could start with will not meet the exact needs of the current situation, automatic case adaptation is a key issue in using CBR. Most CBR approaches rely on adaptation rules to do adaptation. This is one area where potential synergy could be realized by combining CBR with rule-based systems, and some researches are exploiting the best ways of combining the two approaches [Barlet91]. Next, follow some recent efforts on software reuse that employ special forms of case-based reasoning in order to solve software reuse problems:

The Techne Project.

Techne [Mylop91b] is about software reuse focused on information systems. In a nutshell, the project is intended to propose facilities for representing and managing knowledge about information systems so that this knowledge can be reused in the development of new ones. The Techne approach towards software reuse is founded on Case-Based Reasoning, where new reasoning tasks are addressed by adapting experiences in solving similar problems. The project’s research framework also adopts a number of premisses from the DAIDA project [Jarke92], concerning the stages of information system development (requirements, design, implementation), the knowledge that needs to be represented at each stage, and the tools that are required to move from one stage to the next. Finally, the project proposes to treat information systems as complex configurations of components and to exploit recent research on configuration management [Rose91]. Briefly, the project plan has as follows:

- Define a formal framework for case-based information system development; determine the implications of this framework on the structure of software knowledge bases.

- Generate examples of case-based information system development, starting with process reuse, where the whole development process is adopted from previous cases to the problem at hand, and to localize reuse involving particular design decisions or elements of existing requirement specifications, designs and implementations.

- Develop a prototype system in order to evaluate the overall framework.

The Techne project is intended to propose facilities for representing and managing knowledge about information systems. The languages developed during the DAIDA project (Telos, TDL,
and DBPL [Jarke92]) provide a strong basis for expressing and representing such types of knowledge. However, these languages have been designed specifically to describe components involved in any phase of an information system development, and not general purpose components. Therefore, we do not expect that the solutions that will be given in the Techne project will be possible to be extended in order to accomplish reuse of general purpose software.

**KAPTUR and LEARN**

The work of Bailin and Moore [Bail90, Bail91] is distinguished by the following characteristics: 1) It is a systematic approach to reuse of knowledge, not just of products; and 2) it implements learning as an explicitly supported function in a software engineering environment. Two prototype environments KAPTUR and LEARN support software reuse as a knowledge oriented activity. KAPTUR assists in selecting reusable artifacts, by organizing them in terms of their distinctive features. LEARN is an experimental environment that applies machine learning techniques to expand its understanding of an application domain over time.

KAPTUR automatically classifies new components as they are entered. It uses advanced pattern matching techniques to compare components to each other and to established domain norms and classification terms. In addition to storing the components and their classification vocabulary, KAPTUR also requests that users describe their reasons for building a component a particular way, and it requests that users describe how their components are to be reused and how these components have been reused in the past.

LEARN stands for Learning Enhanced Automation of Reuse eNgineering. It is an experimental work, not intended for application in the near term, but important for the evolution of software engineering environments. It combines two different machine learning approaches: Case-Based Reasoning (CBR) and Explanation Based Learning (EBL). In LEARN cases are software engineering problems (for example, a set of requirements), and the reusable knowledge consists of solutions to such problems (e.g., a design meeting the requirements).

EBL is an approach to learning by abstracting key characteristics from an example provided by the user. In LEARN the user is the software engineer, and the examples represent recommended ways of solving a problem. In every activity that LEARN performs, there is an opportunity to learn how to do it better. LEARN thus plays the role of an apprentice who learns by doing, under the supervision of an expert. Eventually, the apprentice should acquire sufficient knowledge to
perform non-trivial tasks independently.

LEARN has been used only for learning a new design repairing rule. It was able to generalize a single instance of the design repair rule and then successfully apply it to different input problems. Following this, the KAPTUR and LEARN team indent to explore learning of other types of knowledge within the case-based solving paradigm: how to match requirements to solutions, how to adapt previous solutions to new problems, and how to evaluate the quality of the solution.

Although the successful learning of a new design repairing rule, provides one data-point to illustrate the learning capabilities of the LEARN system, the exploration of learning other types of knowledge within the explanation-based learning paradigm, constitutes a long-term program of high risk research.

3.7 Summary

Effective reuse of software requires a rich collection of software components and knowledge on how to locate, to adapt them, and even create new ones. The majority of the research efforts presented in this chapter, has taken almost no advantage of this kind of knowledge. The result is that although most of them offer quite good retrieval strategies, their ability to locate similar code in the repository that with the proper adaptation may finally fit the new needs, is quite limited. Furthermore, the selection problem itself becomes harder, as the repository evolves over time.

The application of AI techniques to software engineering is one mechanism through which reusability of software might be achieved. By abstracting and encoding the expertise of experienced software engineers into knowledge bases together with software components, system developers can gain effective access to the artifacts in the software repository as it evolves over time. A relative new reasoning methodology called case-based reasoning provides a strong basis for modeling and reusing knowledge related to the software development process.

Next chapter presents the organizational framework used in our approach which in combination with a special form of case-based reasoning provides considerable assistance to semi-automatically locate the appropriate object-oriented code in a software repository, possibly adapting it to particular requirements, while dealing with the evolution of the repository by adding new components and making the proper re-organization.
Part II

Organizational Framework
Part Introduction

This part presents the organizational framework used in our approach which in combination with a special form of Case-Based reasoning provides considerable assistance to semi-automatically locate the appropriate code, possibly adapting it to particular requirements, while dealing with the evolution of the repository.

Chapter 4 The crucial factors that enter the picture while tackling the repository organization problem are presented, followed by the chosen solution approach.
Chapter 4

Organizational Framework

4.1 Introduction

Software environments typically use an environment database (repository) to provide support for all activities concerning the software development process. One requirement on such environments is that diverse but interrelated data from different phases of the software development process (specifications, system design, and implementations) have to be unified within a single data model. In our approach, we have used the term component to refer to any building blocks of software systems at any stage of the software life cycle.

An important aspect in our work has been to identify the concepts and relations in the programming domain which can be used to capture programming knowledge. These identified concepts and relations have then been explicated in component descriptions and component interconnections by introducing abstractions and generalizations.

In order to model these components and interconnections, the general framework of Component Interconnection Languages (CILs) has been extended and modified. In addition, Telos language - a language that satisfies all the requirements so as to be considered a CIL - is adopted, for presenting this model in a more formal and markable form.

This chapter, presents the solutions given for the most important repository organizational issues, followed by a description of the CIL framework and the most important features of the Telos language. Next follows, an analytical description of all the component types involved in our framework, using both the extended CIL formalism and the Telos language.
4.2 Repository Organization

As already discussed, the organization of the repository is essential to establish a successful reuse environment. There are several crucial factors that enter the picture while tackling the repository organization problem. These factors are:

- Which artifacts may be reused?
- How these artifacts are transformed into reusable components?
- What is the proper size and form of reusable components?
- How should they be classified?
- What are proper techniques and languages for describing certain aspects of components?

Regarding which artifacts are candidates for reuse, it appears that the software community is reaching a consensus: not only code, but also higher-level concepts like designs, domain knowledge, development experience and programming knowledge. Apart from the fact that higher-level software artifacts may be reused directly, it is equally important that they are used indirectly as the providers of the connecting glue between lower-level software artifacts through organizational mechanisms.

The issues of representation and presentation of the reusable artifacts do not adhere to a simplistic solution and certainly need to be separated. A main objective in the work reported, has been to use a single representation model for hosting all these drastically different artifacts in the repository as a workable and reasonable basis. Effectively, the repository stores and manages descriptions of the artifacts (in the sequel they are called components) - all expressed in an appropriate data description language. So, we concentrate on the description of these components.

An obvious starting point is the observation that the description of a component should be very much simpler than the component itself. If components can only do very trivial and simple tasks it will not be worth reusing them. A related observation is that, provided an adequate description can be found, the potential cost-savings are maximized when components are as complicated as possible. By making the components complicated we reap the greatest possible reward for our efforts in using the components in the first place. Taking these points together, we organized our
repository in a way that complicated components appear simple to the re-user.

To accomplish that, we used an abstraction mechanism. Computer scientists often use abstractions to manage the intellectual software [Shaw84]. An abstraction for a software artifact is a succinct description that suppresses the details that are unimportant to a software developer and emphasizes the information that is important.

The different organizational methodologies presented in the previous chapter, incorporate more or less this idea of abstraction explicitly or implicitly. However, none of them was adequate for the application of our method. The main reason for this, is that the applicability of Case-Based Reasoning is strongly related with the way that the software components are organized.

An important aspect in our work has been to identify the concepts and relations in the programming domain which can be used to capture programming knowledge. These identified concepts and relations have then been explicated in component descriptions and component interconnections by introducing abstractions and generalizations, under the following assumptions:

- The software components are described and interconnected in an economic, domain independent way that enables the formation of matching cases. These matching cases may be directly processed by our case-based system.

- The matching cases may be categorized effectively by indices that are easily extracted from the component descriptions.

- The whole organization provides a strong basis for controlled evolution and expansion of the repository with components of high quality through the application of an evolution method.

A consequence of describing and interconnecting the software components under these assumptions is that we are now constraining our repository to be methodology-specific, rather than general-purpose. This is a trade-off that our experience to date shows unavoidable.

The techniques and the language used for describing and interconnecting software components are described in the sequel.
4.3 CILs

Several languages from various fields like software engineering, databases and knowledge representation, provide various means to specify individual features of components and component interconnections. Informally, two components are interconnected whenever there exists at least one resource belonging to one, and is being accessible or derived by the other. All of these languages belong to a special class called Component Interconnection Languages (CILs).

The concept of a MIL (Module Interconnection Language) which is a subset of CILs was introduced by de Remer and Kron [DeRemer76], to permit the articulation of how system configurations could be constructed from their constituents (modules). The primary new idea of a MIL was to formally describe the interdependencies between the components of a system. This description was then to be used to control how the system evolved. A MIL typically deals with segments of program code at the implementation level.

CILs are not restricted only at the implementation level, but instead they use the term component to refer to a segment of software specification independently of the level in which it belongs. Using a CIL to define components and their interconnections, makes it possible to cover all of the aspects which are considered to be important to be supported when following a specific life cycle approach and employing specific languages and techniques.

The main feature of a CIL is that it is not just a language for programming, but fundamentally a language for packaging. When properly applied, this principle can provide much functionality to the application developer, without imposing inconvenient constraints or overhead. From the alternative ways of defining CILs (e.g., using attributed grammars), we choose to follow the Motschnig-Pitric and Mittermeir definition [Motsc89]. A CIL is defined via graphs, as they are considered to be more appealing for displaying the structure of a system.

An interconnected graph (IG) is defined as a tuple:

\[ IG = (C, R, C_r, I_c) \]

where

- \( C \) is a finite set of nodes labeled by the unique names of the components.
- \( R \) is a finite set of resources (nameable entities expressed in some software specification) \( R = \{ r_1, r_2, ..., r_n \}, O \leq n < \infty \).
$C_r$ is a mapping $C \rightarrow P(R)$ (the power set of $R$), so that each component is associated with a set of resources (being a subset of $R$).

$I_c$ is a finite set of ordered pairs of nodes $(c_i, c_j)$, the edges in the graph, denoting some interconnection relationship (e.g. subclass) between two components. The edges are labeled by the name of the relation $I_c$.

---

![Diagram](image)

*Figure 4.1: An IG with three nodes*

---

Figure 4.1 depicts an example of an IG having three components (nodes) named $X$, $Y$ and $Z$. The set $C$ of this IG is $\{X, Y, Z\}$ while $R = R_X \cup R_Y \cup R_Z$. Each component is associated a set of resources, $R_1$, $R_2$ and $R_3$ respectively. Two arcs labeled by the relation $ic$ are defined in the graph, namely $(X, Z)$ and $(Y, Z)$.

This general framework has been extended and modified to formally base our results. In addition to the IG model, Telos language [Mylop90] - a language that satisfies all the requirements so as to be considered a CIL - is adopted, for presenting this model in a more formal and markable form.

### 4.4 Telos

Telos is an E-R based language, designed specifically for information system development applications. It adopts a representational framework which includes structuring mechanisms analogous to those offered by semantic networks and semantic data models. In addition, Telos offers an assertional sub-language which can be used to express both deductive rules and constraints with respect to a given knowledge base.

A Telos knowledge base [Mylop90] consists of *propositions*, grouped into classes and related to other propositions through attributes, which are also propositions. Thus *individuals* (entities,
objects, concepts, nodes) and attributes (relationships, links) are both first class citizens in the adopted representational framework making the language fully extensible. Propositions can be combined to form complex descriptions, consisting of categorized (multivalued) attributes, constraints and deductive rules. Descriptions in the knowledge base are partitioned into tokens and classes, depending on whether they represent particular entities, say the person John or the number 13, or abstract concepts, say those of Person or Number. In addition, Telos supports a number of structuring mechanisms which have been used by knowledge representation languages as well as semantic data models allowing the designer of a knowledge base to introduce gradually and in an orderly fashion the detail that needs to be represented. These mechanisms are: classification (inverse instantiation), aggregation (inverse decomposition), and generalization (inverse specialization).

A formal semantics for Telos knowledge bases that accounts for the several kinds of entities, and is intuitive enough to capture the temporal dimensions of Telos and its object-centered representational framework, can be found in [Plex90]. The semantics is based on a possible-worlds model that includes a linear and non-dense temporal structure.

Using Telos, the general metaclasses of components and resources of the IG model can be defined as:

```
IndividualClass Component in M1_Class
end Component

IndividualClass Resource in M1_Class
end Resource
```

The latter metaclass is further specialized to the different types of resources that exist in the system. For instance, an implementation resource is defined as:

```
IndividualClass IMPL_Resource in M1_Class isA Resource
end IMPL_Resource
```

Finally the general metaclass of interconnections among the components is defined as:
IndividualClass Interconnection in M1_Class with
necessary, single
from_component: Component;
to_component: Component
end Interconnection

The generic attributes associated with Interconnection are listed within the declarations using the syntax `<label> : <type>`. Thus, `from_component: Component` is one such generic attribute which allows instances of Interconnection to have attributes with values which are instances of the metaclass Component. As the interconnections in our model are directed, the `from_component: Component` attribute is instantiated with the component that the interconnection starts from, while the `to_component: Component` attribute is instantiated with the component that the interconnection ends to.

Attribute categories in Telos language group generic attributes and impose constraints on their instances. In the previous declaration the necessary attribute category is a constraint to be enforced at all times for Interconnection instances, while single constraints its instances not to have multi-valued attributes.

Next section, presents an analytical description of all the component types involved in our framework, using both the extended CIL formalism and the Telos language.

### 4.5 Descriptions / Components

Since the software environment at hand covers all the stages of the software life cycle, a first partition of the set of the nodes of the IG should include:

- $C_{IMPL}$ (Implementation Components),
- $C_{DES}$ (Design Components), and
- $C_{SPEC}$ (Specification Components).

It is important to note that, in order a description to be complete it has to include all of the above component types. This happens because the system has to be able to describe and store the
different design and implementation decisions that are made during the software development process. The description of such decisions implies the existence of these component types, as analytically explained in sub-section 4.5.4.

The following sub-sections, present a detailed description of all the software components involved in our framework, including their resources and part of their interconnections.

4.5.1 Implementation Components

Object-Oriented languages have data abstraction and encapsulation constructs called packages, modules, or classes that enable one to define and enforce the boundaries separating the components of a software system. In this thesis, we refer to these abstraction constructs as classes, or implementation components.

Classes can be viewed as a kind of (usually complex) data type definition. Like data type definitions, classes consist of what we call an abstract interface, that is, the exported types and signatures of the operations (or methods); the underlying representations for the data objects; and the implementations of the operations. We refer to an abstract interface and its associated data representation collectively as a class interface.

Every class provides some resources to other classes in a system, and in turn may require some resources from other classes. These resources are specified by the class interface. Consequently, the resources distributed among the implementation components in our model are:

- $R_{PARTS}$, which denote the data representation of a component (parts of a component). They have the general form:

  \[
  \langle part\_interface, part\_type, part\rangle
  \]

  where, $part\_interface$ specifies the way that the part may be accessed, or manipulated. For example in C++ a class part may be considered as private, denoting that this part can only be manipulated by its implementer.

- $R_{METHODS}$, which denote the methods provided by an implementation component. They have the general form:

  \[
  \langle method\_interface, method\_type, method, Variables\rangle
  \]

  where, $method\_interface$ specifies the way that the method is used, and $Variables$ is the set
of the method variables. This set is defined having elements of the general form

<variable_type, variable>

In addition to these resources, we are interested in the name of the file that the component is stored (R\_FILE), and the programming language it is written in (R\_LANG).

Using the Telos syntax, the general metaclass of implementation components which is a subclass of the metaclass Component is defined as:

```plaintext
IndividualClass IMPL_Component in M1_Class
isA Component with
    necessary, single
    language: Programming_Language
    necessary
    filename: String
    attribute
        class_parts: ClassPart;
        methods: Method
end IMPL_Component
```

Note that while the attribute language is restricted to be single-valued, this does not hold for the attributes filename, class_parts and methods. The attribute types ClassPart and Method are defined as:

```plaintext
IndividualClass ClassPart in M1_Class isA IMPL_Resource with
    necessary, single
    part_type: Type
end ClassPart

IndividualClass Method in M1_Class isA IMPL_Resource with
    necessary, single
    method_type: Type
    attribute
        variable: Type
end Method
```
These resources have to be further specialized in order to specify their interfaces. For instance, a protected class part is defined as:

```
IndividualClass ProtectedPart in M1_Class isA ClassPart
end ProtectedPart
```

Up to this point, the features of implementation components and their resources are modeled using Telos metaclasses. These metaclasses can be instantiated to Telos classes corresponding to the different implementation components that are stored in the repository. For instance, the instantiations corresponding to the implementation component `array` described in Example 1 are:

```
IndividualClass array in S_Class, IMPL_Component with
  file
    : "array.cc"
  language
    : "C++"
  class_parts
    : array_element;
    : array_size
  methods
    : array_search
end array

IndividualClass array_element in S_Class, ProtectedPart with
  part_type
    : *int
end array_element

IndividualClass array_size in S_Class, ProtectedPart with
  part_type
    : int
end array_size
```
IndividualClass array_search in S_Class, PublicMethod with
   method_type
       : int
   variable
       item: int
end array_search

An important structuring mechanism provided by object-oriented languages is the
generalization/specialization (G/S) hierarchy. It is used to define a new class (called subclass or
child) as an incremental modification to an existing class (called the superclass or the parent). By
default, the subclass inherits the representation and the operations defined by the superclass. In
addition, the subclass definition can (1) extend the representation by adding more instance vari-
able declarations, (2) add new operations, (3) replace operations, (4) hide operations defined by
the parent so that they do not appear in the child abstract interface, and (5) refine or delete
instance variable declarations [Snyder91].

Referring to our IG model the interconnection $I_e$ of generalization, oriented from a class $c_i$ to its
superclass $c_j$, can be denoted by $I_{gen}(c_i, c_j)$. Using this mechanism, we may directly express
G/S hierarchies of implementation components, corresponding to source code G/S hierarchies.
Also, the isA mechanism provided by Telos, can be directly used to define such hierarchies for
the implementation components that are stored in our repository.

4.5.2 Design Components

Object-oriented design is viewed as a software (de)composition technique. It may be defined as a
technique which, unlike the classical (functional) design, bases the modular decomposition of a
software system on the classes of objects the system manipulates. Instead of building modules
around operations and distributing data structures between the resulting routines, object-oriented
design does the reverse. It uses the most important data structures as the basis for modularization
and attaches each operation to the data structure to which it applies most closely.

As explained in [Meyer87], object-oriented design tends to blur the distinction between design
and implementation. Design and implementation are essentially the same activity: constructing
software to satisfy a certain specification. The only difference is the level of abstraction - during
design certain details may be left unspecified, but in an implementation everything should be
expressed in full. Software development is made much smoother when you use an environment
than encompasses the traditional arena of design and implementation.

The resources distributed among the design components are partitioned into $R_{ADT}$ and $R_{OPER}$.
The first one denotes the abstract data types involved in a design, and the second one the opera-
tions a system must support. Using the Telos syntax a design component is defined as follows:

```
IndividualClass DES_Component in M1_Class is A Component with
  attribute
    adt: AbstractDataType;
    operations: Operation
end DES_Component
```

```
IndividualClass AbstractDataType in M1_Class is A Type, DES_Resource
end AbstractDataType
```

```
IndividualClass Operation in M1_Class is A DES_Resource
end Operation
```

Concerning the running example, the abstract data type `table` (as the array is an implementation
of a table) may be identified with a `search` operation. These correspond to the following Telos
instantiations:

```
IndividualClass array_des in S_Class, DES_Component with
  adt
    : table
  operations
    : table_search
end array_des
```

```
IndividualClass table in S_Class, AbstractDataType
end table
```
As an application consists of several implementation components we need to define how these components are interrelated. Since this is a design issue, we interconnect the design components corresponding to them.

There is only one interconnection type that can be distinguished in this level, namely \( I_{public,used} \), denoting the set of ordered pairs of design components \((c_i, c_j)\), where \( c_i, c_j \) belong to \( C_{DES} \), such that the corresponding to \( c_i \) implementation component holds a resource \( r \) declared as public and this resource is used by the corresponding to \( c_j \) implementation component. Using the Telos syntax this interconnection type is defined as follows:

\[
\text{IndividualClass PublicUsed in M1_CLASS}
\text{isA DES_Interconnection with}
\text{necessary}
\text{resource: DES_Resource}
\text{end PublicUsed}
\]

In this definition, \( DES\_Interconnection \) is a special type of interconnection which relates two design components. It is defined in Telos as:

\[
\text{IndividualClass DES\_Interconnection in M1\_CLASS}
\text{isA Interconnection with}
\text{necessary, single}
\text{from\_component: DES\_Component;}
\text{to\_component: DES\_Component}
\text{end DES\_Interconnection}
\]

Finally in the \( PublicUsed \) definition, the \( resource: DES\_Resource \) corresponds to the abstract data type or operation which is declared as public in the design component specified by the attribute \( from\_component \), and is used by the design component specified by the attribute \( to\_component \).
4.5.3 Specification Components

The ability to structure a specification is vital in any software engineering environment. A specification model in our framework, includes both functional and non-functional specifications. Functional specifications provide a description of the functions carried out by the corresponding implementation component, while non-functional specifications impose global constraints on the operation, performance, and efficiency of any proposed solution to the functional specifications model [Chun90].

To illustrate the distinction between functional and non-functional specifications consider the case of the hash table. Since a hash table is a special type of the abstract data type table, the functions (operations) that should be supported by the corresponding implementation component are the insertion and the search of elements of a specific type. A description of these operations constitutes the functional specification part for the corresponding implementation component. However, an important characteristic of the hash tables is that they are organized in a way that guarantees that the search operation will be very efficient. This has nothing to do with the specification of the operation search which externally has the same effects as the search operation in any table, but rather with the internal organization of the table and the way the operation is implemented. A description of such a characteristic, which impose a constraint on the performance of the operation, constitutes the non-functional specification part for the corresponding implementation component.

When modeling a specification component we are interested for both specification types (functional and non-functional), which are considered to be resources. Therefore, the resources distributed among the specification components are partitioned into \( R_{\text{FUNCT}} \) and \( R_{\text{NON_FUNCT}} \) corresponding to functional and non-functional specifications respectively. They both have the form:

\[
<\text{name}, \text{ATTRIBUTES}, \text{description}>
\]

where \( \text{ATTRIBUTES} \) is defined as the set of attributes of the form:

\[
<\text{label}, \text{type}>,
\]

and description is a textual description of the specification. In our example, the description of the operation \textit{search} carried out by the implementation component array is expressed with the following functional specification resource:

\[
(\text{Search}, \{(\text{where, array}), (\text{what, } <T>)\}, \"Search for an element in an array\")
\]
Using the Telos syntax the specification component and the specification resources (functional and non-functional) are defined as follows:

```plaintext
IndividualClass SPEC_Component in M1_Class
  isA Component with
    necessary
      functional: FunctionalSpecification
    attribute
      nonfunctional: NonFunctionalSpecification
  end SPEC_Component

IndividualClass Specification in M1_Class
  isA SPEC_Resource with
    necessary, single
      description : String;
      name: String
    attribute
      spec_attribute: Type
  end Specification

IndividualClass FunctionalSpecification in M1_Class
  isA Specification
  end FunctionalSpecification

IndividualClass NonFunctionalSpecification in M1_Class
  isA Specification
  end NonFunctionalSpecification
```

Concerning Example 1, the following instantiations need to be made:

```plaintext
IndividualClass SPEC_Comp1 in S_Class, SPEC_Component with
  functional
    : Spec1
end SPEC_Comp1
```
IndividualClass Spec1 in S_Class, FunctionalSpecification, LeafResource

isA Spec4 with

description

: "Search for an element in an array"

name

: "Search"

spec_attribute

where: array;

what: "<T>"

end Spec1

It has to be noted that according to the specification component definition, the provision of a non-functional specification resource is not always necessary. Later on, when describing the dependency components, we are going to present cases where non-functional specification resources are necessary.

Also, in the definition of the functional specification resource Spec1, you may notice that it is defined to be an instance of the meta-class LeafResource. In order to explain this instantiation, let us first describe the way that the specification resources are organized.

The specification resources (functional and non-functional) are organized along a set of generalization/specialization (isA) hierarchies. Each member of the set, consists of a hierarchy having as elements specification resources of the same name (e.g. Search). The most general resource of the hierarchy provides this name, which in the sequel is inherited to the rest of the elements of the hierarchy, and a short textual description. The specializations of the hierarchy are made with respect of the spec_attribute attributes. Specifically, an attribute of type spec_attribute may be either inherited to the more special resources of the hierarchy, or may be redefined to a special type of it.

Figure 4.2, depicts an example of a hierarchy for the functional specification resources with name Search. Since both lists and arrays may be considered as special cases of the abstract data type table, the functional specification resource Spec4 corresponding to the specification of the search operation in tables may be specialized to resources Spec1 and Spec5, corresponding to specifications for search operations in arrays and lists respectively. On the same way the
functional specification resource Spec 5 is further specialized to resources corresponding to specifications for search operations in special types of the abstract data type list.

\[ \text{Spec3} \]
- name: Search
- description: Search for elements

\[ \text{Spec4} \]
- description: Search for elements in tables
  - Spec_attributes: where: table
  - isA

\[ \text{Spec1} \]
- description: Search for elements in arrays
  - Spec_attributes: where: array
  - what: \(<T>\)
  - isA

\[ \text{Spec5} \]
- description: Search for elements in lists
  - Spec_attributes: where: list
  - what: \(<T>\)
  - isA

\[ \text{Spec6} \]
- description: Search for elements in linked lists
  - Spec_attributes: where: linked_list
  - isA

\[ \text{Spec} \]
- isA

\[ \text{...} \]

Figure 4.2: A functional specification resources' hierarchy

It is well known that three different types of elements may be distinguished in a generalization/specialization hierarchy. The most general element, elements which are not further specialized, and the intermediate ones. This distinction, is necessary in our model, in order to achieve better performance during the selection process. Therefore, two special types of resources are defined:

- **RootResources.** A RootResource is the most general resource in an isA hierarchy of specification resources. Formally, it is defined as:

\[
(x \text{ instanceOf RootResource}) \iff
(x \text{ instanceOf Specification}) \text{ AND does not exist } y \text{ such that}
(y \text{ instanceOf Specification}) \text{ AND } (x \text{ isA } y)
\]
• **LeafResources.** A LeafResource, is a resource in a hierarchy of specification resources that is not further specialized. Formally, it is defined as:

\[(x \text{ instanceof LeafResource}) \iff
(x \text{ instanceof Specification}) \land
\text{does not exist } y \text{ such that}
(y \text{ instanceof Specification}) \land
(y \text{ isA } x)\]

It has to be noted that specification resources that are not defined to be instances of any of the above special types of resources, are considered to be intermediate ones. Using the Telos syntax, we define Root and Leaf resources as:

```
IndividualClass RootResource in M1_Class isA Specification
end RootResource
```

```
IndividualClass LeafResource in M1_Class isA Specification
end RootResource
```

All these different types of components are interconnected. In order to do that, the concept of "dependency" is introduced. Each component depends on a component of the previous stage of the software life cycle. An implementation component for a specific application depends on a design component corresponding to the design of this implementation. Similarly, a design component depends on a specification component corresponding to the specifications that are satisfied by this design.

These dependencies are not necessarily one-to-one. This happens because we may make different design decisions in order to satisfy the same specification in different cases. For example, in order to satisfy the specification "Data organization for fast retrieval", either a hashing scheme or a B-tree organization may be chosen. Also, one design may satisfy two different specifications. For example, consider the design describing the search for an element in a table. As we may consider both array and file tables, both requirements for searching in an array or in a file are satisfied by this design.

On the other hand, different implementation decisions may be made in order to satisfy the same design. For example, consider a design component that describes a stack. It consists of the
abstract data type *stack*, with operations *pop*, *push* and *top*. However, it is well known that there are several methods for implementing a stack, using arrays, or special cases of linked lists.

It is desirable to model these design and implementation decisions in order to reuse them in similar cases in the future. In order to do that, we have to include in each dependency link the decision it represents. However, since the set of interconnection relationships $I_e$ consists of ordered pairs of components, it is not possible to encode additional information in them. Therefore, we have to introduce a new component type, called dependency component ($C_{\text{Dependency}}$), which will provide this information. An analytical description of the dependency components and their interconnections can be found in the sequel.

### 4.5.4 Dependency Components

Dependency components interrelate software components of different type, by defining dependency relationships among them. Furthermore, they are used to model the different decisions that are made during the software development process. Two different types of dependency components are distinguished: a) design dependency components ($C_{\text{DES\_Dependency}}$), and b) implementation dependency ones ($C_{\text{IMPL\_Dependency}}$).

A design dependency component defines a dependency relationship between a design component and the corresponding specification one. Additionally, it provides information about the design decision made in the past that caused the creation of a new design. In order to express these relationships in the IG model, the following interconnection relationships are introduced:

$I_{\text{cdes,dep}}$

denoting the set of ordered pairs of components $(c_i, c_j)$ such as $c_i$ belongs to $C_{\text{DES}}$, $c_j$ belongs to $C_{\text{DES\_Dependency}}$ and $c_i$ corresponds to the design component created by the design decision expressed by the component $c_j$.

$I_{\text{cdep,spec}}$

denoting the set of ordered pairs of components $(c_i, c_j)$ such as $c_i$ belongs to $C_{\text{DES\_Dependency}}$, $c_j$ belongs to $C_{\text{SPEC}}$ and $c_i$ expresses a design decision made for the satisfaction of the specification $c_j$. 
The resources distributed among the design dependency components provide information about the design decisions they represent. Actually, we are not interested in the design decisions themselves (the design component provides them) but rather in the reasons that caused them. An application developer makes different design decisions in order to satisfy different non-functional requirements. In the model presented, the non-functional requirements are expressed using non-functional specification resources. Therefore, non-functional specification resources constitute the set of resources distributed among the design dependency components. In addition, another resource that is necessary to be distributed among these components is the textual description of the decision they represent.

It has to be noted that, it is not necessary for a design dependency component to represent always a design decision. It may just relate a design component with the corresponding specification one, acting as a dependency link. A design decision is necessary only in cases that there are more than one design components describing different designs that satisfy the same specification. In such a case, the non-functional specifications justify the selection of a specific design component from the set of designs that satisfy the functional specification at hand.

Using the Telos syntax a design dependency component is defined as:

```
IndividualClass DES_Dependency in M1_Class
isA Interconnection, Component with
    necessary, single
        from_component: DES_Component;
        to_component: SPEC_Component
    attribute
        description: String;
        non_funct: NonFunctionalSpecification
end DES_Dependency
```

Note that the `DES_Dependency` metaclass is defined as a special case of both interconnection (as its instances interconnect two components), and component (as it provides resources). As an example of a design dependency instantiation, consider the specification component `Spec 10` which models a search operation in a table. Two different designs (`DES 10` and `DES 20`) satisfy this specification. The first one uses the abstract data type `array`, while the second one the abstract data type `hash table`. A reason for the selection of a hash table instead of an array, could
be the satisfaction of the non-functional requirement which demands very good performance for
the search operation. A specification of this non-functional requirement is expressed with the
resource Spec 22. The instantiations corresponding to this scenario are:

IndividualClass DES_Dep12 in S_Class, DES_Dependency with

    from_component
      : DES10
    to_component
      : Spec10
description
      : "Use a standard table"
end DES_Dep12

IndividualClass DES_Dep13 in S_Class, DES_Dependency with

    from_component
      : DES20
    to_component
      : Spec10
description
      : "Use a hash table"
non_funct
      : Spec22
end DES_Dep13

IndividualClass Spec22 in S_Class, NonFunctionalSpecification with
description
      : "Efficient search operation"
name
      : "Efficiency"
spec_attribute
      during : Search
during : Search
end Spec22
In analogy, an implementation dependency component defines a dependency relationship between an implementation component and the corresponding design one. Additionally, it provides information about the different implementation decisions that are made during the creation of an implementation component. In order to express these relationships in the IG model, the following interconnection relationships are introduced:

$I_{impl, dep}$

denoting the set of ordered pairs of components \((c_i, c_j)\) such as \(c_i\) belongs to \(C_{IMPL}\), \(c_j\) belongs to \(C_{IMPL\_Dependency}\) and \(c_i\) corresponds to the implementation component created by the implementation decision expressed by the component \(c_j\).

$I_{dep, des}$

denoting the set of ordered pairs of components \((c_i, c_j)\) such as \(c_i\) belongs to \(C_{IMPL\_Dependency}\), \(c_j\) belongs to \(C_{DES}\) and \(c_i\) expresses an implementation decision made for the satisfaction of the design \(c_j\).

The resources distributed among the implementation dependency components are non-functional specifications and textual descriptions similarly with that of the design dependency components. In addition, functional specification resources may also be distributed. This happens because in the design level abstract data types are used, while in the implementation components level these abstract data types need to be implemented. For example, consider two different specification components describing a search operation in an array and a file respectively. These specifications may be satisfied by the same design component with abstract data type resource \(table\) and operation resource \(Search\). However, in order to get the appropriate implementation component, the corresponding functional specification resources have to be also included in the implementation dependency components. Using the Telos syntax an implementation dependency component is defined as:

```
IndividualClass IMPL\_Dependency in M1\_Class
isA Interconnection, Component with
    necessary, single
        from_component: IMPL\_Component;
        to_component: DES\_Component
```
Figure 4.3 depicts the interconnections among the software and dependency components involved in two search operations for an element in a file and an array respectively.

Let us define now, how these different types of software components and dependencies are organized in order to form applications. In order to do that, we first introduce the concept of a context component. A context component is a set of components of a specific type treated in a higher level of abstraction as a component itself. For instance, an implementation context component is defined as:
$c$ belongs to $C_{IMPL\_CONTEXT}$ implies $c$ is subset of $C_{IMPL}$.

The specification context component $C_{SPEC\_CONTEXT}$ and the design context component $C_{DES\_CONTEXT}$ are defined analogously. Using this context mechanism it is possible to express that a specific application consists of several implementation components, which satisfy several design components, that satisfy in turn several specifications.

Each context component depends on a context component of the previous stage of the software life cycle. An implementation context component for a specific application depends on a design context component corresponding to the design of this implementation. Similarly, a design context component depends on a specification context component corresponding to the specifications which are satisfied by this design. Finally, a specification context component depends on a specific application component. The application component ($C_{APPL}$) is introduced to include useful information for a specific application such as the contract of the application, the application developers, deadlines, versioning information, documentation, etc.

The dependency relationships listed above are expressed in the IG model using the following interconnection relationships:

$I_{impl\_des}$

denoting the set of ordered pairs of context components $(c_i, c_j)$ such as $c_i$ belongs to $C_{IMPL\_CONTEXT}$, $c_j$ belongs to $C_{DES\_CONTEXT}$ and $c_i$ corresponds to the implementation of an application with design $c_j$.

$I_{des\_spec}$

denoting the set of ordered pairs of context components $(c_i, c_j)$ such as $c_i$ belongs to $C_{DES\_CONTEXT}$, $c_j$ belongs to $C_{SPEC\_CONTEXT}$ and $c_i$ corresponds to the design of an application with specifications $c_j$.

$I_{spec\_appl}$

denoting the set of ordered pairs of components $(c_i, c_j)$ such as $c_i$ belongs to $C_{SPEC\_CONTEXT}$, $c_j$ belongs to $C_{APPL}$ and $c_i$ corresponds to the specifications of the application $c_j$. 
These interconnections are of one-to-one type since different versions of the same application are represented with different application components. These application components correspond to different specification context components, although they may share common specifications.

### 4.6 Summary

The organization of a repository is essential to establish a successful reuse environment. A main objective in our work was to use a single representation model for placing the different software artifacts (components) in the repository as a workable and reasonable basis.

The repository used in our approach has been organized in a way that complicated components appear simple to the re-user. This has been accomplished by the identification of the concepts and relations in the programming domain that can be used to capture the programming knowledge. These identified concepts and relations have been explicated in component descriptions and component interconnections by introducing abstractions and generalizations.

The component descriptions have been modeled using the extended CIL framework and the Telos Knowledge Representation language.

Several additional requirements are satisfied by the organization proposed in this chapter. First of all, the software components are described in a way that enables the formation of matching cases that are easily processed by a case-based system. Secondly, these matching cases may be categorized effectively by indices that are easily extracted from the component descriptions. Thirdly, the whole organization provides a strong basis for controlled evolution and expansion of the repository with quality components through the application of an evolution method.

Part III, presents the way that the organization of the components presented in this chapter, is used in combination with Case-Based Reasoning and genericity, to semi-automatically locate the appropriate code in the software repository, possibly adapting it to particular requirements, while dealing with the evolution of the repository by adding new components and making the proper re-organization.
Part III

The Method
Part Introduction

This part presents the proposed method which employs a special form of Case-Based Reasoning in conjunction with the specificity-genericity hierarchy, to semi-automatically locate the appropriate code in the software repository, possibly adapting it to particular requirements, while dealing with the maintainability of the repository. More specifically, the part is structured as follows:

Chapter 5 The major research issues on Case-Based Reasoning are addressed. In addition the solutions given for the issues of case representation and indexing are presented.

Chapter 6 The algorithms used for case retrieval and adaptation are analytically described.

Chapter 7 A method based on failure-driven learning and explanation-based generalizations for controlled evolution and expansion of the repository, is presented.

Chapter 8 The complexity of the algorithms used for case retrieval, adaptation and repository evolution is analyzed.
Chapter 5

Case Problems - Solutions

5.1 Introduction

Case-Based Reasoning (CBR) has been attractive as a method for building intelligent reasoning systems because it appears relatively simple and natural. However, there are still problems that need to be worked out for case-based reasoning to be feasible. Of primary concern there are five problems [Barlet91]: a) case representation, b) case indexing, c) case storage and retrieval, d) case adaptation, and e) learning and generalization.

This chapter addresses the different approaches used by the CBR community to overcome these problems. In addition, the formation of matching cases using the reusable software component descriptions described in chapter 4, is presented. These matching cases are represented and indexed, in a way that enables effective case retrieval and adaptation.

5.2 Case Problems

Case-Based Reasoning is a rapidly emerging AI technology that can use past experiences (cases) to solve current problems. The idea of reasoning from relevant past cases is appealing because it corresponds to the process an expert uses to solve "new" problems quickly and accurately. Most CBR research falls into a few broad categories: case representation, indexing, storage and retrieval, adaptation, and learning and generalization.
5.2.1 Case representation

What is a case? How can the case be described to the computer? In its simplest form, a case is a list of features that lead to a particular outcome. In its most complex form a case is a connected set of sub-cases that form the problem-solving task’s structure.

Determining the appropriate case features is the main knowledge-engineering task in CBR systems. It involves defining the terminology of the domain and gathering representative examples (such as cases) of problem solving by the expert.

Another important problem that must be considered, is the structure of cases themselves. Some case-based systems store cases in their entirety in one place in memory, e.g., Cognitive System’s Cased Based Reasoning Shell [Riesb88], Waltz and Stanfill’s MBR [Stanf86], CASEY [Koton88], and HYPO [Rissi88]. The advantage of this is that one case might provide an almost complete solution to a new one. The disadvantage is that it makes it hard to use pieces of old cases to solve pieces of new cases. An alternative, that has been used in our approach, is to break cases into pieces, and to store the pieces individually along with a set of pointers that can be used to reconstruct the whole, as in, e.g., Carbonell’s derivational analogy [Carbo83], and Kolodner’s JULIA [Kolod88]. This makes it easier to access parts of old cases to solve parts of new ones, allowing complex problems to be solved by combining partial solutions of several other problems. It also makes it easier to assess the applicability of part of a previous problem to a new situation.

5.2.2 Case indexing

A CBR system derives its power from its ability to retrieve relevant cases quickly and accurately from its memory. Figuring out when a case should be selected for retrieval in similar future situations, is the goal of case-indexing process. Building a structure or process that will return the most appropriate case, is the goal of the case memory and retrieval process. Case indexing processes usually fall into one of three kinds: nearest neighbor, inductive, and knowledge-based, or a combination of the three [Barlet91].

Nearest-neighbor approaches let the user retrieve cases based on a weighted sum of features in the input case, that match cases in the memory [Barlet88]. This approach is a good one to use if
the retrieval goal is not well defined, or if few cases are available. The biggest problem with using this approach exclusively, is that it can be impossible to converge on a set of global feature weights that will accurately retrieve cases in all situations.

Inductive approaches to indexing are a significant improvement over nearest-neighbor approaches in situations where the retrieval goal or case-outcome is well-defined and there are enough examples of each type of goal [Barlet91]. They have two advantages. First, they can automatically, objectively, and rigorously analyze the cases to determine the best features for distinguishing them. Second, the cases can be organized for retrieval into a hierarchical structure. To perform induction, however, the system needs a reasonable quantity of cases to generate accurate discriminating features and it can take a lot of upfront time to perform the inductive analysis, which is inductive indexing’s primary drawback.

Finally, knowledge-based indexing approaches try to apply existing knowledge to each case in the library to determine which features are important for retrieving each case [Owens88]. In our work, knowledge-based indexing is the preferred approach because a) it was not possible to find a global set of discriminating features for the matching cases, b) there is no guarantee that the system is always provided with a reasonable quantity of cases to generate accurate discriminating features, and c) the necessary explanatory knowledge for knowledge-based reasoning (programming domain knowledge) is available and representable by component descriptions.

### 5.2.3 Case storage and retrieval

Once cases are represented and indexed, they can be organized into an efficient structure for retrieval. Most case-memory structures fall into a range between purely associative retrieval, where any or all of the features of a case are indexed independently of the other features, and purely hierarchical retrieval, where case features are highly organized into a general-to-specific concept structure [Barlet91].

Discrimination networks [Feige63] are one approach to memory organization, but retrieval of a case depends on a conjunction of features being present. This leads to fragility in domains where features can be missing, but the basic approach can be extended to support redundant indexing [Kolod83, Lebow83]. In addition, one can store abstract summary descriptions at internal nodes (in our approach software component descriptions) in the network, giving generalization between
individual cases. This provides the memory with a way of knowing that several cases with different details can be treated the same for some retrieval probes. Only the most prototypical of these cases need to be retrieved under these circumstances. Thus, the memory component of case-based reasoning can provide one way of doing incremental conceptual clustering, and in fact, some researchers in conceptual clustering have drawn on the case-based approach [Fisher88].

The goal of case retrieval is to retrieve "good" cases that can support the reasoning that comes in the next steps. Good cases are those that have the potential to make relevant predictions about the new case. Retrieval is done by using features of the new case that were relevant in solving past cases, as indexes in the case-memory.

The indexing methods, memory organizations, and retrieval algorithms available can recall a set of partially-matching cases for a case-based reasoner to use, but often provide too many cases. Thus, an additional problem to be considered is the choice of best-matching case.

Choosing the best case requires being able to match two cases together to generate a correspondence between their parts. Researchers concerned with the process of analogy have devoted considerable attention to this issue, with most approaches involving some form of heuristic search through the space of partial matches. In this framework, the main issue becomes finding ways to constrain and direct the search of a useful match. For instance Falkenhainer's [Falke86] Structure-Mapping Engine finds mappings that preserve high-order relations between two cases in preference to ones that preserve simple features shared by the cases. Carbonell [Carbo83] addresses the ways derivations are used in this process. Other researchers have proposed different methods, since matching of cases tends to be domain-specific. Fortunately, the organization of cases used in our approach, allows us to create a relatively simple and inexpensive process for matching two cases using some simple heuristics, as analytically explained in chapter 6.

5.2.4 Case adaptation

The goal of case-retrieval is to return the most similar past case that is relevant to the input situation. Case adaptation takes a retrieved case that meets most of the needs of the current situation and turns it into one that meets all of the situation's needs. Case adaptation can involve minimal changes to the input requirements to meet a known goal embodied in a stored case. It is difficult to define a single generically applicable approach to perform case adaptation, because adaptation
tends to be problem-specific [Barlet91].

Most existing CBR systems achieve case adaptation for the specific problem domains they address, by encoding adaptation knowledge in the form of a set of adaptation rules or domain model [Colli87]. Two very different kinds of adaptation have been described in the CBR literature. First, there is structural adaptation, in which adaptation rules apply directly to the solutions stored in a case. CHEF [Hammon89], for example modified particular recipes. Secondly, there is derivational adaptation, where the rules that generated the original solution are re-run to generate the new solution.

Derivational adaptation was used in MEDIATOR [Simps85]. The idea here is to store not only the solution with a case, but the planning sequence that constructed that solution. When a case is retrieved, the reasoner checks to see if the differences between the old situation and the input case affect any of the decisions underlying the solution stored in the case. If so, those decisions are re-evaluated using the values extant in the input situation. In other words, the stored solution is adopted not by changing it directly, but by re-executing parts of the original solution process.

Derivational adaptation is not a replacement for structural adaptation. In any real CBR, both kinds of mechanisms need to be present, because derivational adaptation depends on the presence of the planning structures for stored solutions, and not all solutions have them. Under this perspective, we have used both types of adaptation. Structural adaptation becomes possible by employing type substitution rules, while derivational adaptation is necessary in decision making phases as analytically explained in chapter 6.

5.2.5 Learning and generalization

Learning and generalization are important to CBR systems. Taking advantage of existing techniques for extracting useful information from examples, lets case-based systems avoid some of the main problems of rule-based approaches in gathering problem-solving or classification knowledge and putting it to good use.

There are three major kinds of changes that can occur in the memory of a CBR system. It can add:
new instances of cases,

- new abstractions, or
- new indexes.

New instances are added during normal problem solving. For example, when CHEF creates a recipe, it creates a new instance of some recipe [Hammon89]. New abstractions are formed when a number of cases are discovered to share some common features. The common features are used to create new cases, and the unshared features are used as indexes. The processes for noticing when features are shared, deciding that it is worth creating an abstraction, and choosing indexes, vary from system to system. When IPP [Lebow83] adds a new instance to a set of instances, it automatically checks to see if any generalizations are worth forming from that set of instances.

Forming new abstractions simply on the basis of shared features is not a very good technique. In realistic domains such as IPP's terrorism domain, spurious generalizations or unreasonable abstractions will be formed, and will remain until enough further examples are found. On the other hand, in CYRUS [Kolod84] generalizations are formed in stages. When a new event is indexed to the same place in memory, i.e., under the same case, as a previous event, an abstraction is formed, with all the features the events shared in common that are not already in the existing cases above the events. The norms of this abstraction are marked as "potential" because so little evidence for them exists. As more events are added to this generic case, the actual norms can be determined. Some features that are potential will disappear, because later events do not share them, and some features that are not potential will be added, because they are shared by the most events.

One approach to avoiding some of the problems with forming norms that turn out to be accidental, is a technique known as Explanation-Based Generalization (EBG) [DeJong86]. The idea is that an abstraction is made only when a plausible reason for its existence can be inferred, based on prior causal knowledge. An abstraction can be formed from just one example, if the system can supply enough of the causality to explain the basic features of the case. Furthermore, only features relevant to the causality are kept in the abstraction, so spurious features are less of a problem. The problem with explanation-based generalization is that it can end up doing a lot of work to create an abstraction for a one-time only event.
5.2 Case Problems

Several case-based reasoners do a form of EBG called failure-driven learning. CHEF and MEDIATOR [Simps85] learn not only by saving solutions but also by forming general explanations of why some solutions do not work. In a case-based reasoner with failure-driven learning, a failure report comes back from the real world, entered either by the user or generated automatically by the CBR attempting to execute the solution. The reasoner repairs the case, stores the repair, and reorganizes the case memory so that the repair will be retrieved in similar situations in the future.

In our approach, we have used a special form of failure-driven learning. As cases accumulate, explanation-based generalization is used to define prototypical cases that embody the common features of a group of specific cases, while the rest of their features become parameterized. Furthermore, in cases that a failure takes place the case memory is reorganized in a way that the accuracy of the system is improved in the long run. A detailed description of this process can be found in chapter 7.

Next sections, describe the solutions given for the problems of case representation and indexing discussed above.

5.3 Case Representation

To a large extend the requirements on the knowledge stored along with a case depend upon the goal of the case reuse. If the goal of the reuse is to just re-instantiate the old case either partially or fully in a new problem situation, it is enough to remember the applicability condition on the overall case. However, for the flexibility of reuse, the ability to modify retrieved cases to make them applicable in the new problem situation is essential.

The autonomy and reliability that are required for this modification plays a significant role in deciding the contents of the stored case for flexible case reuse. If we want the process of modifying a case to be very autonomous and reliable, the representation of the stored case will have to include additional information than just the stored case.

If we relax either the autonomy, or the reliability requirement on the case modification, the stored case can be reused without the help of much additional information.
In our research, we focused on finding an economic, domain independent representation for stored cases, which can be easily acquired, and which can guide the various processes involved in the adaptation of existing cases to new problem situations. A case in our framework is defined as a triple $C$ of the form:

$$C = < S, \text{des}, \text{impl}>$$

where:

- $S$ is a set of functional specification resources that constitute a path from a leaf resource to a root resource along an isA hierarchy. It is defined as:

$$S = \text{SetOf } s \text { such that (}
\begin{align*}
\text{instanceOf}(s, \text{FunctionalSpecification}) \text{ AND } \\
\text{exists}\_\text{unique } s_1 \text{ of } S \text{ such that } \text{instanceOf}(s_1, \text{RootResource}) \text{ AND } \\
\text{exists}\_\text{unique } s_2 \text{ of } S \text{ such that } \text{instanceOf}(s_2, \text{LeafResource}) \text{ AND } \\
\text{forall } s_3 \text{ of } S \text{ such that } (s_3 \text{ not equal } s_1) \text{ (isa}(s_3, s_1)) \text{ AND } \\
\text{forall } s_4 \text{ of } S \text{ such that } (s_4 \text{ not equal } s_2) \text{ (isa}(s_2, s_4))
\end{align*}$$

In the above definition, the $\text{instanceOf}(x, y)$ predicate specifies that $x$ is an instance of the class $y$. The expression $\text{exists}\_\text{unique } s$ of $S$ such that $P(s)$, where $P$ is a predicate with argument $s$, is defined as:

$$\text{exists } s \text{ such that (}
\begin{align*}
s \text{ belongs to } S \text{ AND } P(s) \text{ AND } \\
\text{does not exist } s_1 \text{ such that (}
\begin{align*}
s_1 \text{ belongs to } S \text{ AND } (s_1 \text{ not equal } s) \text{ AND } P(s_1)
\end{align*}
\end{align*}$$

Finally, the $\text{isa}(s_2, s_4)$ predicate expresses the transitivity of the $\text{isA}$ relation. It is defined as:

$$\text{isa}(x, y) <\Rightarrow
\begin{align*}
\text{isa}(x, y) \text{ OR } \\
\text{exists } z \text{ such that } \text{isa}(x, z) \text{ AND } \text{isa}(z, y)
\end{align*}$$

- $\text{des}$ is a design component ($\text{instanceOf}(\text{des}, \text{DES}\_\text{Component})$), corresponding to the specification component that provides as resource, the leaf resource of the set $S$. 
5.3 Case Representation

- \textit{impl} is a specific implementation component (an implementation component without parameterized types) corresponding to the design component \textit{des}. The definition of a specific component is:

\[
\text{specific(i) } \leftrightarrow \forall l, x \text{ such that } \text{attribute}(i, \text{class_part}, l, x) \ ( \\
\text{does not exist } y \text{ such that } \text{attribute}(x, \text{part_type}, y, <T>))
\]

where \text{attribute}(c, a, l, t) defines that the class \textit{c} has an attribute of category \textit{a} with label \textit{l} and type \textit{t}.

Each case \textit{C} is broken into pieces (sub-cases). These sub-cases are stored individually along with a set of pointers that can be used to reconstruct the whole. This makes it easier to access parts of old cases to solve parts of new ones, allowing the complex reuse problems to be solved by combining partial solutions of several other problems. It also makes it easier to assess the applicability of part of a previous problem to a new situation. Four matching sub-cases can be distinguished:

i. Given the user's specifications, match an \textit{S} that best suits his needs (sub-case \textit{C}_1).

ii. Given an \textit{S} match a design component by making (if needed) the appropriate design decision.

\[
\textit{C}_2 = < \textit{S}, \text{des_dep}, \textit{des} >
\]

where \text{instanceOf (des_dep, DES\_Dependency)}.

iii. Given a design component \textit{des} match an implementation component \textit{impl}_1 by making (if needed) the appropriate implementation decision.

\[
\textit{C}_3 = < \textit{des}, \text{impl_dep}, \text{impl}_1 >
\]

where \text{instanceOf (impl\_dep, IMPL\_Dependency)}.

iv. If the implementation component \textit{impl}_1 is generic, then transform it to the corresponding specific one according to the user needs.

\[
\textit{C}_4 = < \text{impl}_1, \text{impl} >
\]

where

\[
\text{(NOT(specific(impl\_1))) AND specific(impl)) OR} \\
\text{(specific(impl\_1) AND impl\_1 equals impl)}
\]
Figure 5.1 present graphically an example of a matching case. The exact matching process is analyzed in chapter 6. Before that, the indexing used for cases is presented.

![Diagram of a matching case]

Figure 5.1: An example of a matching case

5.4 Case Indexing

It has been generally assumed that indexes - specially marked features - are an important part of every case in a case memory [Barlet88, Kolod88]. The reasons are 1) indexes are needed to assure that case retrieval is a tractable computational problem, and 2) that we would like to retrieve those cases that match a situation along its important dimensions. Furthermore, a good
index must be:

- easy to extract from low-level descriptions of situations,
- usable as a search and retrieval cue, and
- able to categorize the cases in memory along some interesting dimensions.

Under this perspective, the knowledge-based indexes for the sub-cases presented in the previous section are organized as follows:

- **Sub-case C₁.** Each set $S$ of functional specification resources that constitute a path along an isa hierarchy, is indexed in a complex form by 1) the unique attribute *name* provided by the root resource of the hierarchy and inherited to the rest of its elements, and 2) the specification attributes of the leaf resource of the hierarchy. Later on, when describing the retrieval process, it is going to be analyzed how these indexes are used to retrieve relevant cases quickly and accurately from the system's memory.

- **Sub-case C₂.** The interconnection links $l_{cdes, dep}$ and $l_{cdes, spec}$ provide a retrieval pathway for locating the corresponding design component $des$, given a specification component $spec$ that provides as resource, the leaf resource of the hierarchy $S$. In the case that more than one paths exist ($l_{cdes, dep}, l_{dep, spec}$), the non-functional specification resources provided by the dependency components $dep_i$ are used as additional indexes for matching the best sub-case.

- **Sub-case C₃.** Similarly, the interconnection links $l_{cimpl, dep}$ and $l_{cimpl, des}$ provide the retrieval pathway for locating the implementation component $impl$, given the design component $des$. In the case that more than one paths exist ($l_{cimpl, dep}, l_{dep, des}$), the functional and non-functional specification resources provided by the implementation dependency component $dep_i$ are used as additional indexes for matching the best sub-case.

- **Sub-case C₄.** No indexes are used. This happens because the desired specification implementation component (in the case that a generic component has been located in the previous sub-case) does not actually exist, but has to automatically be created by the system as analytically explained in chapter 6.
5.5 Summary

CBR has been attractive as a method for building intelligent reasoning systems because it appears relatively simple and natural. However, there are still problems to be worked out for case-based reasoning to be feasible. Such problems include, the representation of matching cases, their indexing, the retrieval and possible adaptation of them, and the learning and generalization phase.

The different approaches that have been used by the CBR community to overcome these problems, have been presented. Concerning case representation, in our approach we focused on finding an economic, domain independent representation for stored cases, which is easily acquired, and which is able to guide the various processes involved in the adaptation of existing cases to new problem situations. Each case is broken into sub-cases that are stored individually along with a set of pointers that can be used to reconstruct the whole. This makes it easier to access parts of old cases to solve parts of new ones, allowing the complex reuse problems to be solved by combining partial solutions of several other problems.

Concerning case indexing, the knowledge-based indexing approach has been selected as the necessary explanatory knowledge (programming domain knowledge) is available and representable by component descriptions. Furthermore, the indexes used are a) easily extracted from component descriptions, b) usable as a search and retrieval cue, and c) able to categorize the cases in memory along their interesting dimensions.

The organization of cases in combination with the indexing used in our approach, assures that only very small parts of the memory are accessed during the retrieval and adaptation phase. Chapter 6 presents the algorithms used for case retrieval and adaptation.
Chapter 6

Case Retrieval and Adaptation

6.1 Introduction

The goal of case retrieval is to return the most similar past case that is relevant to the input situation. After the phase of case retrieval, a case adaptation may be necessary. Case adaptation takes a retrieved case that meets most of the needs of the current situation and turns it into one that meets all the situation's needs.

Retrieving cases from a case memory is a massive search problem. Compared to information retrieval techniques it is made even harder by the fact that search is for partial matches. One must be careful to assure that the whole database does not get retrieved with each probe.

The organization of cases in combination with the indexing used in our approach, assures that only small parts of the case memory are accessed each time. The retrieval and adaptation algorithm proposed in this thesis, involves several basic operations. Upon accepting a new case (finding the appropriate source code), the algorithm proceeds as follows:

Step 1. Given the user's specifications, match a functional specification resources hierarchy $S$ that best suits his needs by making, if needed, the proper adaptation (match a sub-case of type $C_1$).

Step 2. Given the functional specification resources hierarchy $S$ located in Step 1, match a design component $des$ by making, if needed, the appropriate design decision (match a sub-case of type $C_2$).
Step 3. Given the design component des located in Step 2, match an implementation component impl by making, if needed, the appropriate implementation decision (match a sub-case of type $C_3$).

Step 4. If the implementation component impl is generic (i.e., it has parameterized types), then transform it to the corresponding specific one according to the user needs (sub-case $C_4$).

The application developer may use one of the following two retrieval methods: a) manual retrieval, and b) CBR-based retrieval.

### 6.2 Manual Retrieval

The application developer (user) has to manually locate the appropriate source code stored in the repository. Initially, the user selects one functional specification resources hierarchy $S$ that best suits his needs (Step 1). He does not have to specify fully the desired $S$, but tries to find it manually, by traversing top-down the isA hierarchies of the functional specification resources stored in the repository.

The traversal starts from the root resource of the hierarchy. According to the definition of the functional specification resources hierarchies presented in chapter 4, each root resource is identified uniquely by its name. Consequently, the system prompts the application developer for the resource name $n$ that should be provided by the target functional specification. In the sequel, the system locates the unique root resource $s_1$ having as "name" this "n". In order to do that, it evaluates the following query:

$$s_1 = x \text{ such that exists } y \text{ such that (}$$
$$\text{attribute}(x, \text{name}, y, n) \text{ AND}\)  
$$\text{instanceOf}(x, \text{RootResource}) \text{ AND}\)  
$$\text{instanceOf}(x, \text{FunctionalSpecification}))$$

Next, the system presents the set $SUB$ of the functional specification resources that are special cases of the root resource $s_1$:

$$SUB = \text{SetOf } x \text{ such that } \text{isA}(x, s_1)$$
The user is prompted for the selection of the most appropriate resource $s_i$ of the set $SUB$, and the same process is repeated until he reaches to a leaf resource $s_n$:

\[
\text{instanceOf}(s_n, \text{LeafResource})
\]

Two different alternatives can be distinguished for the functional specification resource $s_n$:

1) $s_n$ does not have parameterized specification attributes:

\[
\text{does not exist } x \text{ such that } \text{attribute}(s_n, \text{spec\_attribute}, x, <T>)
\]

This is the case of an exact match, i.e exists a case in the case memory that fits exactly the user needs.

2) $s_n$ has parameterized parameterized specification attributes:

\[
\text{exists } x \text{ such that } \text{attribute}(s_n, \text{spec\_attribute}, x, <T>)
\]

In such a case the system prompts the user for the provision of specific types for all the parameterized specification attributes. Next, the system has to verify that the parameterized specification attributes may take as values the types provided by the user. In order to do that, one (or more depending on the number of the parameterized attributes) rule associated with the functional specification resource $s_n$ need to be proven. The rule corresponding to the functional specification resource of the component $C_{spec1}$ of Figure 5.1 has the form:

\[
\text{substitute}(\text{rule12}, s_n, \text{what}, x);-
\text{isA}(x, \text{BasicType})
\]

This rule is interpreted as: The parameterized type $<T>$ of the functional specification resource $s_n$ for its specification attribute with label $\text{what}$, may take as values types $x$ that are defined to be special cases of the general type $\text{BasicType}$.

If the type(s) provided by the user are valid for the parameterized one(s) a generic match happens.

After a successful match (exact or generic) of a leaf resource $s_n$, the set $S$ of the functional specifications resources that constitute a path from a leaf resource to a root resource is constructed, by following recursively the $\text{isA}$ hierarchy links:

\[
S = \text{setOf } x \text{ such that } \text{isa}(s_n, x)
\]
This set constitutes a sub-case $c_1$ of type $C_1$ that fits exactly the user needs. After a successful match of a sub-case of type $C_1$, the system tries to match a sub-case $c_2$ of type $C_2$ (Step 2). In order to do that, the system initially creates the set $SC$ of the specification components that provide as a functional specification resource, the leaf resource $s_n$ of the hierarchy $S$. The query used for this purpose is:

$$SC = \text{SetOf } x \text{ such that } \exists y \text{ such that (}
\text{attribute}(x, \text{functional}, y, s_n) \text{ AND }
\text{instanceOf}(x, \text{SPEC_Component}))$$

If the set $SC$ consists of more than one elements, the application developer has to select the most appropriate according to his needs. Such a selection should be based on the additional functional specification resources provided by the members of the set $SC$. The query used for the retrieval of the functional specification resources provided by a member $spec$ of the set $SC$ is:

$$FS = \text{SetOf } x \text{ such that } \exists y \text{ such that (}
\text{attribute}(spec, \text{functional}, y, x))$$

This step is missing if $SC$ is a single element set. In the sequel, the system tries to locate a design component $des$, given that the specification component that has already been located is $spec$. Two different alternatives can be distinguished:

i) No design decision needs to be made. This means that the specification component $spec$ is connected with only one design dependency component. Formally:

$$\text{exists\_unique } d, y \text{ such that (}
\text{instanceOf}(d, \text{DES\_Dependency}) \text{ AND }
\text{attribute}(d, \text{to\_component}, y, spec))$$

In such a case, the corresponding design component $des$ is located with the evaluation of the following query:

$$des = x \text{ such that } \exists l_1, l_2 \text{ such that (}
\text{attribute}(d, \text{to\_component}, l_1, spec) \text{ AND }
\text{attribute}(d, \text{from\_component}, l_2, x))$$

and

$$c_2 = < S, d, des >$$
ii) A design decision is required. This means that the specification component \( \text{spec} \) is connected with more than one design dependency components. Formally:

\[
\begin{align*}
\text{exist } d_1, d_2, l_1, l_2 \text{ such that (}
&d_1 \text{ not equals to } d_2 \text{ AND} \\
&\text{instanceOf}(d_1, \text{DES\_Dependency}) \text{ AND} \\
&\text{instanceOf}(d_2, \text{DES\_Dependency}) \text{ AND} \\
&\text{attribute}(d_1, \text{to\_component}, l_1, \text{spec}) \text{ AND} \\
&\text{attribute}(d_2, \text{to\_component}, l_2, \text{spec}))
\end{align*}
\]

For the design decision to be made, the set \( \text{DES}_1 \) is initially constructed. This set consists of the design dependency components \( d_i \) that are interconnected with \( \text{spec} \):

\[
\text{DES}_1 = \text{setOf } x \text{ such that } \exists y \text{ such that (}
\begin{align*}
&\text{instanceOf}(x, \text{DES\_Dependency}) \text{ AND} \\
&\text{attribute}(x, \text{to\_component}, y, \text{spec})
\end{align*}
\]

The set \( \text{NF}_1 \) is consequently constructed. This set consists of the couples \( (x, d_i) \) where \( x \) is a non-functional specification resource associated with the design dependency component \( d_i \) of the set \( \text{DES}_1 \):

\[
\text{NF}_1 = \text{setOf } (x, d_i) \text{ such that } \exists y \text{ such that (}
\begin{align*}
&d_i \text{ belongs to } \text{DES}_1 \text{ AND} \\
&\text{attribute}(d_i, \text{non\_func}, y, x))
\end{align*}
\]

Finally, the application developer is responsible for the selection of the couple \( (n_i, d_i) \) from the set \( \text{NF}_1 \) that best meets his needs, and the corresponding design component is located with the query:

\[
\text{des} = x \text{ such that } \exists y \text{ such that ( attribute}(d_i, \text{from\_component}, y, x))
\]

The corresponding implementation component \( \text{impl} \) is located (Step 3) similarly. Again, two different alternatives can be distinguished:

i) No implementation decision needs to be made. This is interpreted as, the design component \( \text{des} \) is connected with only one implementation dependency component. Formally:
exists_unique d, y such that (
    instanceOf(d, IMPL_Dependency) AND
    attribute(d, to_component, y, des))

In such a case, the corresponding implementation component *impl* is located with the evaluation of the following query:

```
impl = x such that exist l_1, l_2, such that (  
    attribute(d, to_component, l_1, des) AND
    attribute(d, from_component, l_2, x))
```

and

```
c_3 = < des, d, impl >
```

ii)  An implementation decision is required. This means that the design component *des* is connected with more than one implementation dependency components. Formally:

```
exists d_1, d_2, l_1, l_2, such that ( 
    d_1 not equals to d_2 AND
    instanceOf(d_1, IMPL_Dependency) AND
    instanceOf(d_2, IMPL_Dependency) AND
    attribute(d_1, to_component, l_1, des) AND
    attribute(d_2, to_component, l_2, des))
```

For the implementation decision to be made, the set *DEP_2* is initially constructed. This set consists of the implementation dependency components *d_i* that are interconnected with *des*:

```
DEP_2 = setOf x such that exists y such that (  
    instanceOf(x, IMPL_Dependency) AND
    attribute(x, to_component, y, des))
```

The set *NF_2* is consequently constructed. This set consists of the couples (*x, d_i*) where *x* is a non-functional specification resource associated with the implementation dependency component *d_i* of the set *DEP_2*:
\[ NF_2 = \text{setOf} (x, d_i) \text{ such that exists } y \text{ such that } \]
\[ d_i \text{ belongs to } DEP_2 \text{ AND } \]
\[ \text{attribute}(d_i, \text{non}_\text{funct}, y, x) \]

Finally, the application developer is responsible for the selection of the couple \((nf_i, d_i)\) from the set \(NF_2\) that best meets his needs, and the corresponding implementation component is located with the query:
\[ \text{impl} = x \text{ such that exists } y \text{ such that } (\text{attribute}(d_i, \text{from}_\text{component}, y, x)) \]

and
\[ c_3 = < \text{des}, d_i, \text{impl} > \]

It has to be noted that an implementation dependency component may also be associated with a functional specification resource (the leaf resource \(s_n\) of the hierarchy \(S\)). If this happens, the following predicate has to be appended in the query used for the construction of the set \(DEP_2\):
\[ \text{attribute}(d_i, \text{funct}, z, s_n). \]

If a generic match has taken place in the first step of the algorithm, the generic implementation component located must be transformed in a specific one \((C_4)\). The detailed transformation process is described in chapter 9.

The manual retrieval method provides limited assistance to the application developer for locating the most appropriate sub-case of type \(C_1\) from the case memory. It is adequate only for sub-cases that no further adaptation is needed to fit exactly the user needs. However, this is not always the case. The most important characteristic of a CBR system is its ability to adapt similar sub-cases stored in a case memory if the desired sub-case does not actually exist. This is accomplished with the application of the CBR-based retrieval method which is described in the sequel.

### 6.3 CBR-based Retrieval

In CBR-based retrieval method, the system interacts with the application developer in a more automatic fashion, and locates, with possible adaptation, the most appropriate case meeting the user needs. Upon accepting a new sub-case of type \(C_1\) (Step 1), the system proceeds as follows:
Step 1.1. Recall relevant sub-cases from case memory. The goal of this step is to retrieve "good" sub-cases that can support the reasoning that comes in the next steps. Good sub-cases are those that have the potential to make relevant predictions about the new case.

Step 1.2. From the collection of sub-cases retrieved in Step 1.1, select the most promising sub-cases to apply the reasoning mechanism. The purpose of this step is to winnow down the set of relevant cases to a few most-on-point candidates worthy of intensive consideration, as the foundation of the solution to be generated in the next step.

Step 1.3. Construct a solution for the new sub-case. During this step a solution is constructed for the new case by matching the most promising of the sub-cases retrieved in step 1.2, and by making, if needed, the proper adaptation.

A detailed description of these steps can be found in the sub-sections following.

6.3.1 Recalling relevant sub-cases

As already discussed, the goal of this step is to retrieve "good" sub-cases of type $C_1$ that can support the reasoning that comes in the next steps. Good sub-cases are considered to be all the hierarchies of functional specification resources, whose elements have the same resource name according to the user needs.

Consequently, the application developer is prompted for the provision of the resource name $n$, that should be provided by the target functional specification resource. Since each of these hierarchies is uniquely identified by a leaf resource having as name this $n$, instead of recalling the complete hierarchies, it is sufficient to recall the set $LR$ of these leaf resources. In order to do that the system evaluates the following query:

$$LR = \text{setOf } x \text{ such that exists } y \text{ such that (}
\text{attribute}(x, \text{name}, y, n) \text{ AND}
\text{instanceOf}(x, \text{LeafResource}) \text{ AND}
\text{instanceOf}(x, \text{FunctionalSpecification}))$$
6.3.2 Selecting the most promising sub-cases

This process is based on the structure of the target functional specification resource, meeting the user needs.

Up to this point the only information available for the target functional specification resource is its resource name $n$. Through an interactive process, the target resource is refined with the provision of more details as prompted by the system. These details are related with the desired (according to the user needs) specification attributes associated with it.

During the first step of the resource refinement process, the system retrieves the set $ALT$ of the triplets $(funct, label, type)$ where $funct$ is a functional specification resource that it is member of the set $LR$, and $label$, $type$ are the labels and types of its specification attributes respectively:

$$ALT = \text{setOf}(x, y, z) \text{ such that } (x \ \text{belongsTo} \ \text{LR AND})$$

$$\text{attribute}(x, \text{spec\_attribute}, y, z))$$

Next, the system constructs the set $AL$ of all attribute labels associated with the elements of the set $LR$. These labels are extracted from the set $ALT$ with the following query:

$$AL = \text{setOf} \ y \text{ such that}$$

$$\text{exist } x, z \text{ such that } (x, y, z) \text{belongsTo} \ \text{ALT})$$

Then, the system prompts the user for the provision of a specification attribute type $t_i$ for each of the elements $l_i$ of the set $AL$. Four alternatives can be distinguished:

1) The user may select one of the elements of the set $T$, as proposed by the system. This set has members the valid (known) types $t_i$ associated with the labels $l_i$ of the set $LR$:

$$T = \text{setOf} \ x \text{ such that}$$

$$\text{exists} \ y \text{ such that } (y \ \text{belongsTo} \ \text{LR AND})$$

$$\text{attribute}(y, \text{spec\_attribute}, l_i, x))$$
2) The user may provide a type \( t_i \) that does not belong to the set \( T \), but according to his opinion fits better to the new problem situation.

3) The user may exclude a label \( l_i \) of the set \( AL \) from the target specification resource. This is possible with the provision of a \( no \) value for the requested type of the specification attribute with label \( l_i \).

4) The user may provide a \( no\_care \) value as type for the specification attribute with label \( l_i \). The provision of such a value, specifies that the user does not actually care whether such a label exists in the target functional specification resource.

It has to be noted that the system does not allow the user to provide as type the parameterized type \(<T>\), independently if this type belongs to the set \( T \).

Reaching the end of this process, the system creates a set \( D \) of couples \((l_i, t_i)\) that in combination with the name \( n \) constitute a description of the target functional specification resource. In fact this set includes much more information due to the existence of some \( no \) or \( no\_care \) values. Therefore, we partition the set \( D \) into its subsets \( D_1, D_2, \) and \( D_3 \), as follows:

\[
D = D_1 \cup D_2 \cup D_3
\]

\[
D_1 = \text{setOf } (l_j, t_j) \text{ such that } \begin{align*}
(l_j, t_j) & \text{ belongsTo } D \text{ AND } \\
t_j & \text{ not equals to } "no" \text{ AND } \\
t_j & \text{ not equals to } "no\_care"
\end{align*}
\]

\[
D_2 = \text{setOf } (l_j, t_j) \text{ such that } \begin{align*}
(l_j, t_j) & \text{ belongsTo } D \text{ AND } \\
t_j & \text{ equals to } "no"
\end{align*}
\]

\[
D_3 = \text{setOf } (l_j, t_j) \text{ such that } \begin{align*}
(l_j, t_j) & \text{ belongsTo } D \text{ AND } \\
t_j & \text{ equals to } "no\_care"
\end{align*}
\]

The specification attributes of the set \( D_1 \), in combination with the resource name \( n \), provide a description of the target functional specification resource. The set \( D_2 \) is used to exclude candidate functional specification resources from the set \( LR \). Using \( D_2 \), the system constructs the set...
**6.3 CBR-based Retrieval**

$LR_1$ of the actual functional specification resources having at least the same specification attributes with that described by $D_1$ (the rest are specification attributes that the application developer does not actually care about them):

$$LR_1 = \text{setOf x such that (}}$$

$$x \text{ belongs to } LR \text{ AND}$$

$$\text{forall } l_i, t_i \text{ such that } (x, l_i, t_i) \text{ belongs to } ALT (\text{) } (l_i, \text{"no"}) \text{ does not belong to } D_2))$$

Next, follows the matching process (Step 1.3). During this step, the target functional specification resource, described by $D_1$, is compared with the candidate functional specification resources that are members of the set $LR_1$. Five matching alternatives can be distinguished. These are described in the sub-sections following.

### 6.3.3 Exact Match

A specification resource matches exactly the user needs. This is the case where an element $s$ of the set $LR_1$ matches exactly the target functional specification resource described by $D_1$. In other words, each label-type couple of the set $D_1$ is also a label-type couple of the component $s$. Formally:

$$\text{exact\_match}(s) \leftrightarrow$$

$$\text{exists } s \text{ such that (}}$$

$$s \text{ belongs to } LR_1 \text{ AND}$$

$$\text{forall } (l_i, t_i) \text{ such that } (l_i, t_i) \text{ belongs to } D_1 (\text{) } (s, l_i, t_i) \text{ belongs to } ALT))$$

After an exact match of a leaf resource $s$, the set $S$ of the functional specifications resources that constitute a path from a leaf resource to a root resource, is constructed. In order to do that, the system follows recursively the isA hierarchy links:

$$S = \text{setOf } s_1 \text{ such that isa}(s, s_1)$$

This set constitutes a sub-case $c_1 = < S >$ of type $C_1$ that fits exactly the user needs. The matching of sub-cases $c_2$ of type $C_2$ (Step 2), and $c_3$ of type $C_3$ (Step 3) are similar with that described
in section 6.2.

6.3.4 Generic Match

There is no specification resource that matches exactly the user's request:

\[ \text{does not exist } s \text{ of } LR_1 \text{ such that } \text{exact\_match}(s) \]

However, the only difference between the target functional specification resource (described by \( D_1 \)), and a resource \( s \) of the set \( LR_1 \), is that the specification attributes of the target functional specification resource are of specific type, while some of the specification attributes of the resource \( s \) are parameterized:

\[
\text{generic\_match}(s) \iff \\
\exists s \text{ such that (}
\text{ } s \text{ belongs to } LR_1 \text{ AND } \\
\forall (l_i, t_i) \text{ such that } (l_i, t_i) \text{ belongs to } D_1 ( \\
(s, l_i, t_i) \text{ belongs to ALT OR (} \\
(s, l_i, <T>) \text{ belongs to ALT AND} \\
\text{substitute}(rule_k, s, l_i, t_i)))
\]

The substitute\((rule_k, s, l_i, t_i)\) predicate is interpreted as: there exists a rule \( rule_k \) associated with the parameterized component \( s \) that permits its parameterized specification attribute with label \( l_i \) to take as type the type \( t_i \). Later on, when describing the evolution of the repository, we are going to present the way that this rule is constructed.

After a generic match, the system constructs the sub-case \( C_1 \) and then matches the sub-cases \( C_2 \) and \( C_3 \) in exactly the same way as in the manual retrieval case.

As mentioned in sub-section 6.2, if a generic match happens, the generic implementation component located need to be transformed in the corresponding specific one (\( C_4 \)). This process is analytically explained in chapter 9.
6.3.5 Similarity Match

There is no specification resource (generic or specific one) that matches the user’s request:

\[
\text{does not exist } s \text{ of } LR_1 \text{ such that (}
\text{exact_match}(s) \text{ OR generic_match}(s))
\]

However, the only difference between the user specifications and that described by an element \( s \) of the set \( LR_1 \) is that some of their specification attribute types are not exactly the same, but they are similar. For example, an array and a linked list can be characterized as similar as they both are special cases of tables. Such similarities are user-defined and the system learns about them as explained in chapter 7. Formally a similarity match is defined as:

\[
similarity\_match(s) \iff \text{exists } s \text{ such that (}
\text{s belongs to } LR_1 \text{ AND }
\text{forall } (l_i, t_i) \text{ such that } (l_i, t_i) \text{ belongs to } D_1 (}
\text{((s, l_i, t_i) belongs to ALT OR (}
\text{((s, \ l_i, <T>) belongs to ALT AND}
\text{substitute(rule}_k, \ s, \ l_i, \ t_i)\text{)) OR (}
\text{(exists } t_j \text{ such that } ((s, l_i, t_j) \text{ belongs to ALT AND}
\text{modification(rule}_l, \ substitute, \ t_i, \ t_j, \ last\_attempt) \text{ AND}
\text{last\_attempt equals to true))))}))
\]

Note that, in the above definition partial generic matches are also permitted, as some types of the specification attributes of \( s \) may be parameterized. The predicate \( \text{modification(rule}_l, \ substitute, \ t_i, \ t_j, \ last\_attempt) \) expresses similarity between the types \( t_i \) and \( t_j \). An example of such a predicate is:

\[
\text{modification(rule}_21, \ substitute, x, y, \text{true)}:-
\text{isA(x, table) AND}
\text{isA(y, table)}
\]

This predicate is interpreted as: there exists a modification rule \( rule_21 \), that permits the substitution of the type \( x \), with the type \( y \), if they are both known to the system as special cases of the general type \( table \). The \text{last\_attempt} variable is a boolean, expressing if such a substitution was
valid the last time it was proposed by the system. The way that this variable is treated, is going to be explained later.

If such a similarity is discovered, the system proposes the modifications that are necessary to be made in one repository’s implementation component in order to create a new implementation component that fits the new case. The implementation component of the repository is located by matching the sub-cases $C_2$, $C_3$ and $C_4$ (if a parameterized type is discovered) similarly with the previous matching alternatives.

The modifications required in the implementation component which was located, are distinguished into:

1) The substitution of the type $t_i$ with the known similar type $t_j$, and

2) other code modifications that were necessary in a similar case in the past, as reported by the system. For example, when substituting the abstract data type array with that of a linked list, the method for accessing the next element of the array should be also modified. This happens because while the next element of an array is located by incrementing the current element pointer, in the case of a linked list the next pointer associated with each element of the list should be followed. An analytical example of a similarity match where such modifications are necessary, can be found in chapter 10, where the prototype implementation is presented.

After these modifications are made and the new component is tested by the user, the system suggests that a new generic component may be created by parameterizing the similar types. If the user agrees, the system automatically creates the new generic component and reorganizes the repository by removing the specific interconnections, and by creating new interconnections that relate the generic component with the appropriate components stored in the repository. If the user does not agree, then the system creates a new specific component with its appropriate interconnections. These operations are related with the evolution of the repository and they are described in much detail in chapter 7.
6.3.6 Optional Match.

There is no specification resource (generic, specific or similar one) that matches the user’s request:

$$\text{does not exist } s \text{ of } LR_1 \text{ such that (}
$$
\begin{align*}
\text{exact_match}(s) & \text{ OR } \text{generic_match}(s) \text{ OR } \text{similarity_match}(s))
\end{align*}

However, the system detects a stored resource resembling the specifications, where in some types which are not the same, no similarity has been defined, even though common characteristics exist. In such a case the user may establish a similarity definition (if according to his opinion such similarity exists) between the given types and the repository ones. Formally:

$$\text{optional_match}(s) \iff \text{exists } s \text{ such that (}
$$
\begin{align*}
s & \text{ belongs to } LR_1 \text{ AND } \\
& \text{forall } (l_i, t_i) \text{ such that } (l_i, t_i) \text{ belongs to } D_1 ( \\
& ((s, l_i, t_i) \text{ belongs to } ALT \text{ OR (} \\
& ((s, l_i, <T>) \text{ belongs to } ALT \text{ AND } \\
& \text{substitute}(rule_k, s, l_i, t_i))) \text{ OR (} \\
& (\text{exists } t_j \text{ such that } ((s, l_i, t_j) \text{ belongs to } ALT \text{ AND } \\
& \text{modification}(rule_l, \text{substitute}, t_i, t_j, \text{true})) \text{ OR (}} \\
& (\text{does not exist } \text{modification}(rule_n, \text{substitute}, t_i, t_j, \text{true}) \text{ OR } \\
& \text{modification}(rule_n, \text{substitute}, t_i, t_j, \text{false})) \text{ AND } \\
& \text{get_similarity}(t_i, t_j))))\)
\end{align*}

Note that, according to the above definition partial generic or similarity matches may also occur.

The definition part

$$\text{does not exist } \text{modification}(rule_n, \text{substitute}, t_i, t_j, \text{true})$$

verifies that no similarity has been defined between the types $t_i$ and $t_j$ in the past. Also, the $\text{get_similarity}(t_i, t_j)$ predicate prompts the user about the possible similarity between the types $t_i$ and $t_j$. This predicate succeeds if the user answers positively. In a negative answer the predicate fails (and consequently the optional match fails), the system removes the candidate functional specification component $s$ from the set $LR_1$, and the following predicate which defines that the types $t_i$ and $t_j$ are not similar is appended to the repository:

$$\text{modification}(rule_m, \text{substitute}, t_i, t_j, \text{false}) :- \text{true}$$
It is noted that a negative answer from the user does not exclude a possible similarity between the types \( t_i \) and \( t_j \) at a later stage. This is the reason for the existence of the predicate

\[
\text{modification}(\text{rule}_n, \text{substitute}, t_i, t_j, \text{false})
\]

in the optional match definition. The position of this predicate guarantees that an attempt for prompting the user for such a similarity in another case in the future, will take place if all attempts for unknown similarities fail. In such an attempt, if the user answer positively, the \( \text{last_attempt} \) variable becomes \text{true}. The case that a true \( \text{last_attempt} \) variable becomes false is described in section 7.8.

In the case that an optional match takes place, the system locates the corresponding implementation component by matching the sub-cases \( C_2 \), \( C_3 \) and \( C_4 \) (if a parameterized type exists) similarly with the previous matching alternatives.

Consequently, the user is prompted for other source code modifications (additional to substitutions of the type \( t_i \) with the type \( t_j \)), that were necessary in order for the implementation component located to fit the new case. These modifications are also stored in the repository, in order to be used in similar cases in the future. The way that the system handles this information is analytically explained in chapter 7.

Finally, the system proposes the creation of a new generic component similarly with the similarity match case.

### 6.3.7 No Match

When nothing of the above applies:

\[
\text{does not exist s of LR}_1 \text{ such that (}
\text{exact_match(s) OR generic_match(s) OR}
\text{similarity_match(s) OR optional_match(s))}
\]

the user has to create a new component by scratch, and then interactively provide the system with appropriate information, in order to store this new component in the repository. This interactive process is described in chapter 7 where the evolution of the repository is presented.

Using the context mechanism described in chapter 4 we are able to reuse not only implementation components but also generic and specific applications. A \textit{generic application} is defined as an
application, in which a number of the implementation components it consists of, are parameterized. The user has access to all the information related to an application such as specifications, design and implementation, so he can reuse them in order to create a similar application. If certain changes are needed for the new application, these must be done in the implementation components level as described before, and then the system is automatically reorganized by creating a new application component and new specification, design and implementation context components which describe this new application.

6.4 Summary

The goal of case retrieval is to return the most similar past case that is relevant to the input situation. Case adaptation takes a retrieved case that meets most of the needs of the current situation and turns it into one that meets all the situation's needs.

The organization of cases in combination with the indexing used in our approach, assures that only small parts of the case memory are accessed each time.

Two alternative retrieval methods have been proposed: a) manual retrieval, and b) CBR-based retrieval. In the first one, the application developer tries to manually find a case that best suits his needs. It is adequate only for cases where no further adaptation is needed. However, this is not always the case. The most important characteristic of a CBR system is its ability to adapt similar cases stored in a case memory, if the desired case does not actually exist. Such an adaptation is possible only if the CBR-based retrieval method is used. In this method, the system interacts with the application developer in a more automatic fashion, and locates, with possible adaptation, the most appropriate case according to the user specifications.

In our approach, the entire software production cycle is integrated in such a way as to improve the quality of the repository, as well as its maintainability. The organization of cases used, provides a strong basis for controlled evolution and expansion of the repository with quality generic components through the application of an evolution method. Chapter 7, presents the different ways that the repository may be evolved.
Chapter 7

Repository Evolution

7.1 Introduction

In our approach, the object's repository evolution is a major concern. The entire software production cycle is integrated in such a way as to improve the quality of the repository, as well as its maintainability.

The organization of cases, described in chapter 5, provides a strong basis for controlled evolution and expansion of the repository with quality components through the application of an evolution method.

The evolution method is based on learning and generalizations. Learning denotes changes in the cases stored in the repository that are adaptive in the sense that enable the system to do the same task drawn from the same population more efficiently, more effectively, and without repeating previously made errors.

As already discussed in chapter 5, a special form of failure-driven learning mechanism is used. In cases that a failure takes place (i.e., a wrong match) the case memory is reorganized in a way that the accuracy of the system is improved in the long run. Furthermore, as cases accumulate, explanation-based generalizations are used to define prototypical cases that embody the common features of a group of specific cases, while the rest of their features become parameterized.

The different ways that the repository may be evolved include:
• **Specific component creation.** A new specific component is created after a similarity or an optional match if the user does not agree with the system’s proposal for the creation of a generic component, as already discussed in chapter 6.

• **Generic component creation.** A new generic component is created (by parameterizing similar attributes) after a similarity or an optional match if the user agrees with the system’s proposal for such a creation.

• **New code insertion.** This is the case where the repository contained no matching component. The application developer may provide the system with new code created by scratch, and the appropriate descriptions. In the sequel, the repository is reorganized in order to include this new matching case.

• **Decision creation.** A new decision (design or implementation one) must be created if after a successful match the user is not satisfied with such a solution due to the violation of some non-functional specifications. The repository is reorganized in order to include such a decision. This is accomplished by the creation of a new dependency component that provides as resources the violated non-functional specifications, in order these resources to be used in a future decision making phase.

• **Specifications refinement.** There is no guarantee that the application developer always provides specifications that fully describe the contents of the repository. These incomplete specifications may cause a future match that it is not correct according to the user needs. In order to overcome this problem, a specifications refinement is necessary. This is accomplished by three alternative ways: a) insertion of a new specification attribute, b) removal of a specification attribute, and c) type adaptation.

• **Parameterization rule refinement.** A parameterization rule has to be refined if a wrong generic match has taken place due to an erroneous parameterization. The refinement process constraints the parameterized type to take as values any of the old types except the invalid ones.

• **Learning about similarities.** The system learns about type similarities during the optional match phase as already discussed in chapter 6. New similarity rules are created in order to be used in similar future cases. Additionally, the system learns about differences in the
similar types usage.

Next sections, describe analytically the algorithms related with the repository evolution phase.

### 7.2 Specific Component Creation

A new specific component must be created after a similarity or an optional match if the user does not agree with the system's proposal for the creation of a generic component. Suppose that \( c_i \) is the matching case of type \( C \) corresponding to the optional or similarity match happened:

\[
\begin{align*}
    c_i &= <S_i, des_i, impl_i > \\
\end{align*}
\]

This matching case is further analyzed to the following matching sub-cases (for simplicity no sub-case of type \( C_4 \) exists):

\[
\begin{align*}
    c_{i1} &= <S_i> \\
    c_{i2} &= <S_i, des_{dep_i}, des_i > \\
    c_{i3} &= <des_i, impl_{dep_i}, impl_i > \\
\end{align*}
\]

For simplicity reasons we assume that the specification component \( spec_i \) that has been located during the matching phase, provides a single functional specification resource \( s_i \) (the leaf resource of the hierarchy \( S_i \)). If more than one functional specification resources are provided, the algorithms presented in the sequel have to be repeated for each of these resources.

The creation of a new specific component causes the creation of a new matching case \( c_j \) of type \( C \) such that:

\[
\begin{align*}
    c_j &= <S_j, des_i, impl_j > \\
\end{align*}
\]

This matching case is analyzed to the following matching sub-cases:

\[
\begin{align*}
    c_{j1} &= <S_j> \\
    c_{j2} &= <S_j, des_{dep_j}, des_i > \\
    c_{j3} &= <des_i, impl_{dep_j}, impl_j > \\
\end{align*}
\]

The matching sub-case \( c_{j1} \) consists of a hierarchy \( S_j \) of functional specification resources. The leaf resource \( s_j \) of the hierarchy has as specification attributes, the specification attributes that belong to the set \( D_1 \). These attributes are the same with that of \( s_i \) (where \( s_i \) is the leaf resource of the hierarchy \( S_i \)) but some of their types have been replaced by some other similar types that caused the similarity or the optional match. Formally, \( s_j \) is defined as:
forall \((l_k, t_k)\) such that \((l_k, t_k)\) belongs to \(D_1\) (attribute\((s_j, \text{spec\_attribute}, l_k, t_k)\))

In addition to these specification attributes, a textual description \(txt\) provided by the user for the new functional specification resource is necessary:

\[
\text{attribute}(s_j, \text{description}, l, \text{txt})
\]

The rest of the functional specification resources that constitute the hierarchy \(S_j\) are located as follows. For simplicity of the presentation, assume that the only difference between the target resource and the one located during the similarity (or optional) match, is that two types \(t_i\) and \(t_j\) are not the same but similar (they are both defined as special cases of the type \(t_k\)). Two alternatives can be distinguished:

i) There exists one functional specification resource \(s_k\) in the hierarchy \(S_i\) having a specification attribute of type \(t_k\). In such a case \(s_j\) becomes a special case of the resource \(s_k\) and the hierarchy \(S_j\) is defined as:

\[
S_j = S_k \cup \{s_k, s_j\}
\]

where

\[
S_k = \text{setOf } S_l \text{ such that } \text{isa}(s_k, s_l)
\]

ii) No functional specification resource in the hierarchy \(S_i\) has a specification attribute of type \(t_k\). In such a case the functional specification resource \(s_j\) becomes a special case of the root resource of the hierarchy \(S_i\). Formally:

\[
S_j = \{s_j\} \cup \{s_r\}
\]

where

\[
s_r = x \text{ such that } x \text{ belongs to } S_i \text{ AND } \text{instanceOf}(x, \text{RootResource})
\]

If more than one similarities have been identified between couples \(t_{ij}\) and \(t_{ij}\) \((i=1, ..., n)\), where the types of each couple are special cases of the type \(t_{ij}\), the second alternative described above remains the same. However, for the first alternative the resource \(s_k\) is the resource \(s_{ij}\) of the hierarchy \(S_i\) having a specification attribute of type \(t_{ij}\) which is closest to the root resource of the hierarchy. Formally:
\[ S_k = \text{setOf } x \text{ such that for } l=1,\ldots,n ( \\
\quad x \text{ belongs to } S_l \text{ AND } \\
\quad \text{attribute}(x, \text{spec}_l, l_1, t_{ul})) \]

and

\[ s_k = x \text{ such that (} \\
\quad x \text{ belongs to } S_k \text{ AND } \\
\quad \text{forall } y \text{ such that } y \text{ belongs to } S_k ( \\
\quad \quad x \text{ equals to } y \text{ OR } \text{isa}(y, x)) \]

The matching sub-case \( c_{j2} \) consists of the functional specification hierarchy \( S_j \), a design dependency \( \text{des}_{dep_j} \) (no decision is needed) and the design component \( \text{des}_i \) of the matching case \( c_{i2} \). The old matching case and the new one consist of the same design component since the differences between these cases are related with type definitions which do not affect the design level. The design dependency component \( \text{des}_{dep_j} \) is defined as:

\[
\text{instanceOf}(\text{des}_{dep_j}, \text{DES\_Dependency}) \text{ AND} \\
\text{attribute}(\text{des}_{dep_j}, \text{to\_component}, l_1, \text{spec}_j) \text{ AND} \\
\text{attribute}(\text{des}_{dep_j}, \text{from\_component}, l_2, \text{des}_i)
\]

where \( \text{spec}_j \) is a new specification component that provides as resource the functional specification \( s_j \). In the case that the specification component \( \text{spec}_i \) provides additional functional specification resources, these resources are also provided by \( \text{spec}_j \). The only difference is that their specification attribute types \( t_{ul} \) (if exist) have been substituted with their similar types \( t_{ul} \).

Finally, the matching sub-case \( c_{j3} \) consists of the design component \( \text{des}_i \) of the matching subcase \( c_{j2} \), a new implementation dependency component \( \text{impl}_{dep_j} \), and a new implementation component \( \text{impl}_j \). The implementation dependency component is defined as:

\[
\text{instanceOf}(\text{impl}_{dep_j}, \text{IMPL\_Dependency}) \text{ AND} \\
\text{attribute}(\text{impl}_{dep_j}, \text{to\_component}, l_1, \text{des}_i) \text{ AND} \\
\text{attribute}(\text{impl}_{dep_j}, \text{from\_component}, l_2, \text{impl}_j) \text{ AND} \\
\text{attribute}(\text{impl}_{dep_j}, \text{funct}, l_3, s_j)
\]
Note that, the reason for the creation of this new implementation decision for the design $des_i$ is the creation of the new functional specification resource $s_j$. This is expressed by the last predicate of the above definition.

### 7.3 Generic Component Creation

As already discussed in sub-section 5.2.5 there is a diversity in the number of case examples that are required in order to form a generalized case in different case-based systems.

In our approach we have used a special form of explanation-based generalizations [DeJong86]. The idea is that a generalization is formed from just one example (a similarity or optional match between a stored case and the target one) if the user agrees with the system’s proposal for such a creation. In cases that a failure takes place due to the incomplete information during the creation of the generalized case, the reasoner repairs this case by applying "parameterization rule refinement" which is described later.

Suppose that $c_i$ is the matching case of type $C$ corresponding to the optional or similarity match happened:

$$c_i = < S_i, des_i, impl_i >$$

This matching case is analyzed to the following matching sub-cases (for simplicity no sub-case of type $C_4$ exists):

$$c_{i1} = < S_i >$$
$$c_{i2} = < S_i, des\_dep_i, des_i >$$
$$c_{i3} = < des_i, impl\_dep_i, impl_i >$$

For simplicity reasons we assume that the specification component $spec_i$ that has been located during the matching phase, provides a single functional specification resource $s_i$ (the leaf resource of the hierarchy $S_i$). If more than one functional specification resources are provided, the algorithms presented in the sequel need to be repeated for each of these resources.

The creation of a new generic component causes the substitution of the matching sub-case $c_i$ with a new matching case $c_j$ of type $C_1$ such that:

$$c_j = < S_j, des_j, impl_j >$$

This matching case is analyzed to the following matching sub-cases:
\[ c_{j1} = \langle S_j \rangle \]
\[ c_{j2} = \langle S_j, \text{des}_p, \text{des}_i \rangle \]
\[ c_{j3} = \langle \text{des}_i, \text{impl}_p, \text{impl}_j \rangle \]

The matching sub-case \( c_{j1} \) consists of a hierarchy \( S_j \) of functional specification resources. The leaf resource \( s_j \) of the hierarchy has as specification attributes the common specification attributes between the set \( D_1 \) and the functional specification resource \( s_i \). The rest of the specification attributes are parameterized. Formally:

\[
\text{forall } (l_k, t_k) \text{ such that (}
(l_k, t_k) \text{ belongs to } D_1 \text{ AND }
(s_i, l_k, t_k) \text{ belongs to } ALT (\text{attribute}(s_j, \text{spec\_attribute}, l_k, t_k)))\]
\[
\text{AND}
\]

\[
\text{forall } l_k \text{ such that (}
(l_k, t_k) \text{ belongs to } D_1 \text{ AND }
(s_i, l_k, t_m) \text{ belongs to } ALT \text{ AND }
t_m \text{ not equals } t_k (\text{attribute}(s_j, \text{spec\_attribute}, l_k, <T>))
\]

In addition to these specification attributes, a new textual description \( \text{txt} \) provided by the user for the modified functional specification resource is necessary:

\[
\text{attribute}(s_j, \text{description}, l, \text{txt})
\]

The rest of the functional specification resources that constitute the hierarchy \( S_j \) are located as follows. For simplicity reasons, assume that the only difference between the target resource and the one located during the similarity (or optional) match, is that two types \( t_i \) and \( t_j \) are not the same but similar, (they are both defined as special cases of the type \( t_k \)). Two alternatives can be distinguished:

i) there exists one functional specification resource \( s_k \) in the hierarchy \( S_i \) having a specification attribute of type \( t_k \). In such a case \( s_j \) becomes a special case of the resource \( s_k \) and the hierarchy \( S_j \) is defined as:

\[
S_j = S_k \cup \{s_k, s_j\}
\]
where

\[ S_k = \text{setOf } s_l \text{ such that } \text{isa}(s_k, s_l) \]

ii) No functional specification resource of the hierarchy \( S_i \) has a specification attribute of type \( t_k \). In such a case the functional specification resource \( s_j \) becomes a special case of the root resource of the hierarchy \( S_i \). Formally:

\[ S_j = \{ s_j \} \cup \{ s_r \} \]

where

\( s_r = x \text{ such that } x \text{ belongs to } S_i \text{ AND } \text{instanceOf}(x, \text{RootResource}) \)

If more than one similarities have been identified between couples \( t_{il} \) and \( t_{jl} \) (\( l=1,\ldots,n \)), where the types of each couple are special cases of the type \( t_{il} \), the second alternative described above remains the same. However, for the first alternative the resource \( s_k \) is the resource \( s_{kl} \) of the hierarchy \( S_i \) having a specification attribute of type \( t_{kl} \) which is closest to the root resource of the hierarchy. Formally:

\[ S_k = \text{setOf } x \text{ such that for } l=1,\ldots,n \text{ ( } x \text{ belongs to } S_i \text{ AND } \text{attribute}(x, \text{spec}\_\text{attribute}, l_{il}, t_{il}) \) \]

and

\[ s_k = x \text{ such that } ( x \text{ belongs to } S_k \text{ AND } \forall y \text{ such that } y \text{ belongs to } S_k \text{ ( } x \text{ equals to } y \text{ OR } \text{isa}(y, x) \) ) \]

During the creation of the matching case \( c_{j1} \) a set of parameterization rules associated with the resource \( s_j \) have to be appended in the repository. Each rule corresponds to one of the parameterized attributes of the functional specification resource \( s_j \), and has the form:

\[ \text{substitute}(\text{rule}_l, s_j, l_j, x):- \]

\[ \text{isa}(x, t_{il}) \]

where \( l_j \) is the label of a parameterized attribute of the resource \( s_j \):

\[ \text{attribute}(s_j, \text{spec}\_\text{attribute}, l_j, \langle T \rangle) \]

and \( t_{il} \) is the common general type identified during the similarity or the optional match for the similar types \( t_{il} \) and \( t_{jl} \) of the resources \( s_i \) and \( s_j \) respectively. Formally:
forall $l_\beta$ such that 

\[(l_\beta, t_\beta) \text{ belongs to } D_1 \text{ AND} \]
\[(s_i, l_\beta, t_\beta) \text{ belongs to } ALT \text{ AND} \]
\[t_\beta \text{ not equals } t_\beta (\]
\[\text{insert_rule((substitute(rule1, s_j, l_\beta, x):-isa(x, t_\beta)))}) \]

where the predicate insert_rule(x) appends the rule x in the repository.

The matching sub-case $c_{j2}$ consists of the functional specification hierarchy $S_j$, a design dependency (or decision) $des_{depj}$ and the design component $des_i$ of the matching case $c_{i2}$. The design dependency (or decision) $des_{depj}$ is similar to the design dependency (or decision) $des_{depi}$ and differs only in the attribute of category to_component, where the old specification component $spec_i$ has been replaced with the new component $spec_j$. This new component provides the same functional specification resources with that of $spec_i$, and differs only in that they are parameterized. Formally:

forall $x, y, z$ such that 

attribute($des_{depi}, x, y, z$) AND 

$x$ not equals to_component ( 

attribute($des_{depj}, x, y, z$) AND 

attribute($des_{depj}, to_component, l_1, spec_j$)

Finally, the matching sub-case $c_{j3}$ consists of the design component $des_i$ of the matching sub-case $c_{j2}$, a new implementation decision (or dependency) component $impl_{depj}$, and a new generic implementation component $impl_j$. The implementation decision (or dependency) component $impl_{depj}$ is similar to the implementation decision (or dependency) component $impl_{depi}$ and differs only in the attribute of category from_component, where the old implementation component $impl_i$ has been replaced with the new generic implementation component $impl_j$. Formally:

forall $x, y, z$ such that (attribute($impl_{depi}, x, y, z$) AND 

$x$ not equals from_component) ( 

attribute($impl_{depj}, x, y, z$)) AND 

attribute($impl_{depj}, from_component, l_1, impl_j$)
7.4 New Code Insertion

This is the case where the repository contained no matching component. The application developer may provide the system with the new code created by scratch. In such a case, the user is prompted for the appropriate specifications describing this new code. For simplicity reasons we assume that only one specification resource is provided by the user. If more than one functional specification resources are provided, the algorithms presented in the sequel need to be repeated for each of these resources. This phase causes the creation of a new matching case $c_j$ of type $C$ such that:

$$c_j = <S_j, des_j, impl_j>$$

This matching case is analyzed to the following matching sub-cases:

$$c_{j1} = <S_j>$$

$$c_{j2} = <S_j, des_{dep_j}, des_j>$$

$$c_{j3} = <des_j, impl_{dep_j}, impl_j>$$

The matching sub-case $c_{j1}$ consists of a hierarchy $S_j$ of functional specification resources. The leaf resource $s_j$ of the hierarchy has as specification attributes, the specification attributes described by the set $D_1$:

$$\forall (l_k, t_k) \text{ such that } (l_k, t_k) \text{ belongs to } D_1 \left( \text{attribute}(s_j, \text{spec\_attribute}, l_k, t_k) \right)$$

It has to be noted that the new functional specification resource $s_j$ may comprise some additional (user-defined) specification attributes that their labels are not included in the set $AL$ (and consequently in the set $D_1$), but according to the user’s opinion they are necessary. Also, the user may create a new functional specification resources hierarchy for a new specification name (corresponding to a new operation) that it is unknown for the system. Finally, a textual description for the new functional specification resource is necessary:

$$\text{attribute}(s_j, \text{description}, l, \text{txt})$$

The rest of the functional specification resources of the hierarchy are user defined. In order to do that, the user follows the most appropriate path (or create a new path if necessary) in the different functional specification resources hierarchies until he reaches to a resource $s_k$ that the resource $s_j$ can be specialized to it. The new hierarchy $S_j$ is defined as:

$$S_j = S_k \cup \{s_k, s_j\}$$
where

\[ S_k = \text{setOf } s_i \text{ where } \text{isa}(s_k, s_i) \]

The design component \( des_j \) and the implementation component \( impl_j \) of the matching cases \( c_{j2} \) and \( c_{j3} \), are the design and implementation descriptions of the new code respectively.

Finally, a design dependency \( des\_dep_j \) and an implementation one \( impl\_dep_j \) are created. These dependencies interconnect the new design component \( des_j \) with the new specification component \( spec_j \) (the component that provides as resources all the new functional specifications provided by the user), and the new implementation component \( impl_j \) with the new design component \( des_j \) respectively. They are defined as:

\[
\begin{align*}
\text{instanceOf}(\text{des}\_\text{dep}_j, \text{DES}\_\text{Dependency}) \text{ AND} \\
\text{attribute}(\text{des}\_\text{dep}_j, \text{from}\_\text{component}, l_1, des_j) \text{ AND} \\
\text{attribute}(\text{des}\_\text{dep}_j, \text{to}\_\text{component}, l_2, spec_j) \text{ AND} \\
\text{instanceOf}(\text{impl}\_\text{dep}_j, \text{IMPL}\_\text{Dependency}) \text{ AND} \\
\text{attribute}(\text{impl}\_\text{dep}_j, \text{from}\_\text{component}, l_1, impl_j) \text{ AND} \\
\text{attribute}(\text{impl}\_\text{dep}_j, \text{to}\_\text{component}, l_2, des_j)
\end{align*}
\]

### 7.5 Decision Creation

A new decision must be created after a successful match according to the user’s functional specifications (i.e., the implementation component located satisfies fully the functional specifications provided by the user), if the user is not satisfied with such a solution due to the violation of some non-functional specifications.

Suppose that \( c_i \) is the matching case of type \( C \) corresponding to the successful match happened:

\[ c_i = \langle S_i, des_i, impl_i \rangle \]

This matching case is analyzed to the following matching sub-cases (for simplicity no sub-case of type \( C_4 \) exists):

\[ c_{i1} = \langle S_i \rangle \]

\[ c_{i2} = \langle S_i, des\_dep_i, des_i \rangle \]

\[ c_{i3} = \langle des_i, impl\_dep_i, impl_i \rangle \]
The creation of a new decision causes the creation of a new matching case \( c_j \) of type \( C \). Two different alternatives can be distinguished for this matching case, according to the decision type:

i) **Creation of a new implementation decision.** The user decides that the reason for the violation of the non-functional specifications is not the design of the component located, but the way it is implemented. The creation of a new implementation decision implies the availability of new source code which satisfies the violated non-functional specifications. If such code is available, a new implementation dependency component is created. This component interconnects the implementation component corresponding to this new code with the design component located during the matching phase (it remains the same as both the old and new implementations satisfy the same design). In addition, the new dependency component includes the non-functional specification resources describing the violated non-functional specifications, in order these resources to be used in a future decision making phase. Formally, a new matching case \( c_j \) is created. It is defined as:

\[
c_j = < S_i, des_i, impl_j >
\]

which is further analyzed to:

\[
c_{j1} = < S_i >
\]

\[
c_{j2} = < S_i, des_{dep_i}, des_i >
\]

\[
c_{j3} = < des_i, impl_{dep_j}, impl_j >
\]

Note that, the matching sub-cases \( c_{j1} \) and \( c_{j2} \) remain the same with the sub-cases \( c_{i1} \) and \( c_{i2} \) respectively, since both the old matching case \( c_i \) and the new matching case \( c_j \) share the same design and consequently the same specifications. The matching sub-case \( c_{i3} \) has been replaced with that of \( c_{j3} \). This matching sub-case consists of the same design component \( des_i \) with the matching sub-case \( c_{i3} \). However, the creation of the new implementation decision causes the creation of a new implementation dependency component \( (impl_{dep_j}) \), and the insertion in the repository of a new implementation component \( (impl_j) \).

The implementation dependency component provides as **from_component** resource (defining the component that the interconnection starts from) the new implementation component \( impl_j \), and as **to_component** resource (defining the component that the interconnection ends to) the design component \( des_i \) located during the matching phase. Additionally, it provides a functional specification resource pointing to the leaf resource \( s_j \) of the hierarchy \( S_i \), and one or more (for simplicity assume one) non-functional specification resources
describing non-functional specifications that are satisfied by the new implementation component $impl_j$. Formally:

$$\text{instanceOf}(impl\_dep_j, IMPL\_Dependency) \text{ AND}$$
$$\text{attribute}(impl\_dep_j, from\_component, l_1, impl_j) \text{ AND}$$
$$\text{attribute}(impl\_dep_j, to\_component, l_2, des_i) \text{ AND}$$
$$\text{attribute}(impl\_dep_j, funct, l_3, s_i) \text{ AND}$$
$$\text{attribute}(impl\_dep_j, non\_funct, l_4, s_{nf}) \text{ AND}$$
$$\text{attribute}(impl\_dep_j, description, l_5, txt)$$

where $txt$ is a textual description of the implementation decision that is provided by the user.

ii) **Creation of a new design decision.** The user decides that the reasons for the violation of the non-functional specifications are related with the design of the component located. The creation of a new design decision implies the availability of new source code that satisfies the violated non-functional specifications. If such code is available, a new design dependency component is created, which interconnects the specification component located during the matching phase (it remains the same as both the old and new designs satisfy the same functional specifications) with the new design component corresponding to the design description of the new source code. In addition, the new dependency component includes the non-functional specification resources describing the violated non-functional specifications, in order these resources to be used in a future decision making phase. Finally, the new design component is interconnected with the implementation component describing the new source code, through a new implementation dependency component.

Formally, a new case $c_j$ is created. It is defined as:

$$c_j = <S_i, des_j, impl_j>$$

which is further analyzed to:

$$c_{j1} = <S_i>$$
$$c_{j2} = <S_i, des\_dep_j, des_j>$$
$$c_{j3} = <des_j, impl\_dep_j, impl_j>$$

Note that, the matching sub-case $c_{j1}$ remains the same with the matching sub-case $c_{i1}$, since both the old matching case $c_i$, and the new matching case $c_j$ share the same specifications.
The new matching case $c_{j2}$ consists of the functional specification hierarchy $S_j$, the new design dependency component $des_{dep_j}$, and the new design component $des_j$. This new design component corresponds to the design description of the new source code.

The new design dependency component provides as from_component resource (defining the component that the interconnection starts from) the new design component $des_j$, and as to_component resource (defining the component that the interconnection ends to) the specification component $spec_i$ located during the matching phase. Additionally, it provides one or more (for simplicity assume one) non-functional specification resources that describe the non-functional specifications that are satisfied by the new design component, and a textual description $txt$ of the decision provided by the user. Formally:

\[
\text{instanceOf}(des_{dep_j}, \text{DES}\_\text{Dependency}) \land \\
\text{attribute}(des_{dep_j}, \text{from}\_\text{component}, l_1, des_j) \land \\
\text{attribute}(des_{dep_j}, \text{to}\_\text{component}, l_2, spec_i) \land \\
\text{attribute}(des_{dep_j}, \text{non}\_\text{func}, l_3, s_{nf}) \land \\
\text{attribute}(des_{dep_j}, \text{description}, l_4, txt)
\]

The new matching sub-case $c_{j3}$ consists of the new design component $des_j$, an implementation dependency component $impl_{dep_j}$ and the new implementation component $impl_j$ satisfying the design $des_j$. The implementation dependency component $impl_{dep_j}$ provides as from_component resource the new implementation component $impl_j$, and as to_component resource the new design component $des_j$. Formally:

\[
\text{instanceOf}(impl_{dep_j}, \text{IMPL}\_\text{Dependency}) \land \\
\text{attribute}(impl_{dep_j}, \text{from}\_\text{component}, l_1, impl_j) \land \\
\text{attribute}(impl_{dep_j}, \text{to}\_\text{component}, l_2, des_j)
\]

### 7.6 Specifications Refinement

The evolution algorithms described above are related with new information that is appended to the repository. However, there are cases that the already stored information need to be refined in order to avoid wrong matching of cases that may happen due to incomplete or erroneous information.
One way used to refine the repository, is what we call *specifications refinement*. There is no guarantee that the application developer always provides specifications that fully describe the contents of the repository. These incomplete specifications may cause a future match that it is not correct according to the user needs. Consider for example, the implementation component $impl_k$ corresponding to a linked list of strings, where each time a new string is appended, it is placed at the end of the list. Also consider that when the application developer described the specifications corresponding to this component, he provided as specifications attributes only the couples $(where, list)$ and $(what, string)$ and he did not care about the position that a new element is appended. If in a future selection the application developer wishes to locate the implementation component corresponding to a linked list of sorted strings, the system will provide as solution the implementation component $impl_k$ which is not appropriate for the problem at hand.

In order to overcome this problem, the specifications corresponding to the implementation component $impl_k$ need to be refined. In the running example, this may be accomplished with the provision of the couple $(position, last)$ as a new specification attribute.

Suppose that, the matching case $c_i$ needs a specifications refinement:

$$c_i = <S_i, des_i, impl_i>$$

After the refinement, we have a new mapping case $c_j$:

$$c_j = <S_j, des_i, impl_i>$$

where $S_j$ consists of the same elements with that of $S_i$ except the leaf resource which is refined:

$$S_j = \{s_j\} \cup S_k$$

where

$$S_k = \text{set of } x \text{ where (}
\begin{align*}
&x \text{ belongs to } S_i \text{ AND} \\
&\text{NOT}(\text{instanceOf}(x, \text{LeafResource}))
\end{align*}$$

Next follow the different alternatives that may be used for the refinement of the leaf resource $s_i$ of the hierarchy $S_i$:

i) Insertion of a new specification attribute $(l_k, t_k)$. 
ii) Removal of a specification attribute \((l_k, t_k)\). It has to be noted that the system does not permit the removal of a specification attribute \((l_k, t_k)\) that it is inherited:

\[
\text{exist } s_k, t_m \text{ such that } (s_k \text{ belongs to } S_j \text{ AND attribute}(s_k, \text{spec_attribute}, l_k, t_m))
\]

iii) Type adaptation. This is the case where the type \(t_k\) of a specification attribute is replaced with a new type \(t_m\). It has to be noted that if the attribute with label \(l_k\) is inherited, the new type \(t_m\) must be a special case of the attribute types (corresponding to specification attributes with label \(l_k\)) of the more general resources of the hierarchy.

iv) Any workable combination of the above.

### 7.7 Parameterization Rule Refinement

A parameterization rule \(rule_i\) (associated with a leaf resource \(s_i\)) must be refined if a wrong generic match has taken place due to an erroneous parameterization. For instance, consider the case of the generic component array with a sorting operation, having the type of its elements parameterized. Consider also that this generic component has been created after a similarity (or optional) match between the types \(int\) and \(float\). In such a case the parameterization rule associated with the corresponding functional specification resource is:

\[
\text{substitute}(rule_i, s_i, \text{what}, x):- \\
\text{isA}(x, \text{BasicType})
\]

This rule defines that the parameterized type can take as values types that are special cases of a basic type (\(int\), \(char\), \(float\), etc.) (see section 7.3).

If in a future selection the application developer wishes to find a component that sorts characters, a generic match is going to happen (as \(char\) is a special case of the type \(BasicType\)). However, this is not a correct match because the comparison operators (">", "<") used for number comparison are not valid for character comparison.

In order to avoid such an invalid parameterization, the rule \(rule_i\) needs to be refined. This refinement constraints the parameterized type to take values any basic type but character, and the parameterization rule becomes:
7.7 Parameterization Rule Refinement

substitute(rule₁, sᵢ, what, x):-
    isa(x, BasicType) AND
    NOT (x equals to char)

7.8 Learning about Similarities

The system learns whether two types $t_i$ and $t_j$ are similar during the optional match phase. The application developer is prompted to verify such a similarity and then (in the case that such a similarity is valid) he is asked to provide the general type $t_k$ that both $t_i$ and $t_j$ are special types of it. It has to be noted that the system proposes some of the already known common general types $t_l$ for the types $t_i$ and $t_j$:

$$t_l = \text{x such that (}
    \text{isa}(t_l, \text{x}) \text{ AND}
    \text{isa}(t_j, \text{x})\text{)}$$

The application developer may select one of these types or provide a new one. If a new type $t_k$ is provided, the repository is initially refined with the new specialization information:

$$\text{isa}(t_i, t_k) \text{ AND}
\text{isa}(t_j, t_k)$$

Next, follows the creation of the new similarity rule $\text{rule}_m$. It has the form:

$$\text{modification}(\text{rule}_m, \text{substitute, x, y, true}):-
\text{isa}(x, t_k) \text{ AND}
\text{isa}(y, t_k)$$

Furthermore, the system prompts the application developer for additional modifications he made to the source code located, in order to be stored and presented in a future similarity match between the types $t_i$ and $t_j$. These differences are described in textual form and they are stored as:

$$\text{difference}(\text{rule, method, before, after})$$

where, $\text{rule}$ is the identifier of the new similarity rule ($\text{rule}_m$ in our case) and $\text{before}$ and $\text{after}$ are two strings describing the implementation of the method $\text{method}$ before and after the modifications respectively.
The last argument of a similarity rule is set to true during its creation. This true value indicates success for the last time such a similarity validation was attempted. If in a future selection this rule causes a similarity match that it is not correct according to the user’s opinion, the system will set this argument to false. Also, this argument is set to false, in the case that the user does not verify the similarity when first asked during the optional match phase.

This false value does not exclude a possible similarity between these two types in the future. However, the system will first try to locate similarities that their rules have a true value in their last argument (see similarity match condition). If such a similarity can not be identified, the system will prompt the user to verify type similarities using rules with false values in their last argument. In a positive answer of the application developer, the last argument of such a rule becomes true.

### 7.9 Summary

The organization of cases used in our approach, provides a strong basis for controlled evolution and expansion of the repository with quality components through the application of an evolution method.

The evolution method is based on a special form of failure-driven learning. In cases that a failure takes place (i.e., a wrong match) the case memory is reorganized in a way that the accuracy of the system is improved in the long run. Furthermore, as cases accumulate, explanation-based generalizations are used to define prototypical cases that embody the common features of a group of specific cases, while the rest of their features become parameterized.

The different ways that the repository may be evolved are: a) specific component creation, b) generic component creation, c) new code insertion, d) decision creation, e) specifications refinement, f) parameterization rule refinement, and g) learning about similarities.

The complexity of the algorithms used for case retrieval, adaptation and repository evolution is analyzed in chapter 8.
Chapter 8

Complexity Issues

8.1 Introduction

The implementation of the repository where the matching cases and some additional software development experience are stored, implies the existence of a DBMS. Here, we refer to an extended relational DBMS where in addition to the information (facts) that fit naturally into traditional record-oriented models, it often requires special interpretation or analysis.

The overall cost of the system is composed of the DBMS cost, and the cost of the user efforts to work with the system. This chapter presents the optimized definition of the queries used in our retrieval and adaptation algorithms, and analyzes their costs. Furthermore, the costs of the repository evolution algorithms (maintenance costs) are also computed.

8.2 Computing System Cost

Every DBMS embodies a layered architecture which is in charge on external storage devices to the objects visible at the data model interface. At the bottom, the database is a very large string stored on disk which needs to be interpreted by the DBMS code. Proceeding bottom up, each layer derives objects containing more structures and allowing more powerful operations. Finally, the uppermost interface supports the objects operations, and integrity constraints of the data model.
Here, we refer to an extended relational DBMS where in addition to the information (facts) that fit naturally into traditional record-oriented models, it often requires special interpretation or analysis. Thus, our DBMS must provide facilities for storing rules and some special deductive operations (e.g. recursive definition of the isA relationship).

The overall cost of the system is composed of the DBMS cost and the cost of the user efforts to work with the system. The interface in the two areas consists of the functional capabilities and usability of the query language [Vassil84], mainly in the response time of the system. The total cost to be minimized is the sum of the following [Jarke84]:

Secondary Access Cost: The cost of (or time for) loading data pages from secondary storage into main memory. This is influenced by the number of data to be retrieved (mainly by the size of intermediate results), the clustering of data on physical pages, the size of the available buffer space, and the speed of the devices used.

Storage Cost: The cost of occupying secondary storage and memory buffers over time. Storage costs are relevant only if storage becomes a system bottleneck and if it can be varied from query to query.

Computation Cost: The cost for (or time of) using the central processing unit (CPU).

In centralized systems, the costs are dominated by the time for secondary storage accesses although the CPU costs may be quite high for complex queries [Gotli75].

For the optimization of single queries, storage costs are usually assumed to be of secondary importance. They are considered only for the simultaneous optimization of multiple queries. There, remain the cost of secondary storage accesses (usually measured by the number of comparisons performed).

Exact optimization of query evaluation is in general computationally intractable and is hampered further by the lack of precise statistical information about the database. However, in our approach the queries are limited to a set of standard queries performed during the case retrieval or repository evolution phases. Thus, they have been optimized manually by programming the associated procedures and restricting the user's input to the provision of some values when prompted by the system.
8.2 Computing System Cost

Queries can be represented in a number of forms. In the context of query optimization, an appropriate query representation must fulfill the following requirements. It should be powerful enough to express a large class of queries, and it should provide a well-defined basis for query transformation.

The relation algebra as defined by Codd [Codd71] is a collection of operators on relations. These operators fall into two classes, that is, traditional set operators such as Cartesian products, union, intersection and difference, and special relation algebra operators such as restriction, projection, join and division. These special operators are defined in the sequel.

The restriction operator applied to a relation $rel$ constructs a horizontal subset according to a quantifier-free predicate containing only monadic terms or intrarelational dyadic terms comparison (comparison between two attributes of the same element):

$$\text{Rest}(rel, \text{pred})$$

The projection operator serves to construct a vertical subset of the relation $rel$ by selecting a set $A$ of specified attributes and eliminating duplicate tuples within these attributes:

$$\text{Proj}(rel, A)$$

The join operation permits two relations $rel_1$ and $rel_2$ to be combined into a single relation whose attributes are the union of the attributes of $rel_1$ and $rel_2$:

$$\text{Join}(rel_1, A \text{ op } B, rel_2)$$

The comparison operators usually allowed in joins are $=, \neq, <, >, \geq$, and $\leq$.

Finally the division operator provides an algebraic counterpart to the universal quantifier.

The database queries involved in the algorithms presented in the previous chapters may be solved with many possible strategies. Before we present the most efficient of them, using the relational algebra notation, let us first define the relations that should be included in the schema of the database that the matching cases are stored. These relations are:

- The `instanceOf` relation:

  $$\text{instanceOf}(+, \text{name, class})$$
the generalization/specialization relation:

\[ \text{isA}(+ \text{ class}, \text{superclass}), \text{and} \]

the attribute relation:

\[ \text{attribute}(+ \text{ class}, \text{attribute\_category}, \text{label}, \text{type}) \]

Key attributes are written in italics; a given combination of key attribute values identifies a relation element uniquely. The character '+' before a relation field, denotes that an index is used for this field.

The following sections present the optimized definition of the queries used in our algorithms for case retrieval and adaptation and the computations of their costs. Furthermore, section 8.6 presents the computation of the costs of repository evolution algorithms (maintenance costs). It has to be noted that the detailed data used for the computations below are not always available, but have to be estimated.

### 8.3 Recalling relevant sub-cases

The first database query that is evaluated during the CBR-based retrieval phase, is used for the construction of the set \( LR \). This set consists of the leaf resources having as "name" the value "n" provided by the user (sub-section 6.3.1). The equivalent optimized representation of this query in relational algebra is:

\[
\text{Proj( Join( Rest(attribute, type=n AND attribute\_category="name") } \\
\text{class=name, } \\
(\text{Proj(Rest(instanceOf, class="LeafResource"), name) } \cap \\
\text{Proj(Rest(instanceOf, class="FunctionalSpecification"),name}))}, \\
\text{name})
\]

Let \( N_1, N_2 \) be the number of elements of the relations attribute, and instanceOf respectively. A sketch of the algorithm that evaluates the above query follows:

FOR \( i:=1 \) TO \( N_1 \) DO

read the \( i_{th} \) element of the relation attribute
8.3 Recalling relevant sub-cases

IF (i.type=n AND i.attribute_category="name")
  X:=i.class
  if (scan(X,"LeafResource") AND scan(X,"FunctionalSpecification"))
    append X to the target relation

where scan(X,A) scans the instanceOf relation for the existence of the tuple instanceOf(X,A).
Since the instanceOf relation is indexed on the field name, instead of scanning sequentially, the
matching elements are retrieved directly. Thus, the scan(X,A) function requires in the worst case
C₁ database accesses, where C₁ is the maximum number of meta-classes that a stored class is
their instance (6 in our approach). Consequently, the number of disk accesses that are required
for the above algorithm are:

\[ N₁ + C₂ * (C₁ + C₁) = N₁ + C \]

where C₂ is the number of elements of the relation attribute, that have in their field type the value
"n". This number is not considerable compared with the number N₁. Thus, we may easily say that
the complexity of the algorithm for the construction of the set LR is O(N₁) (measured in disk
accesses).

The complexity of this algorithm, may be further improved, if we sacrifice part of our storage
resources. This may be possible, with the introduction of a new relation:

leaf_resource(+ name, class)

which has as fields the name provided by the user (name), and the leaf resource that such a name
appears (class). In such a case, the relational algebra query for the construction of the set LR
becomes:

\[ \text{Proj( Rest( leaf_resource, name=n ), class) } \]

The index on the field name, guarantees that the disk accesses required for the evaluation of this
query, is equal to the number of the elements of the set LR (|LR|).

8.4 Selecting the most promising sub-cases

The first algorithm of this phase is responsible for the construction of the set ALT. This set con-
sist of the triplets (x, y, z) such that x is a functional specification resource of the set LR and y, z
are the associated label-type couples. A sketch of the algorithm follows:
ALT=\{
FOREACH i IN LR
    ALT:=ALT \cup Proj( Rest (attribute, class= i AND
    attribute_category="spec_attribute"),
    class, label, type)
\}

Since the attribute relation is indexed on the field class, the number of disk accesses required for each of the database queries of the above loop, can not be more than the maximum number of attributes \( C_2 \) that a functional specification resource may have. From our experience, this number is always a single digit. Thus, the total number of disk accesses that are required for the evaluation of the above algorithm can not be more than:

\[ |LR|*C_2 \quad (O(|LR|)) \]

It has to be noted that the number of leaf resources that are members of the set \( LR \) is much less than the number of the implementation components that provide the operation "n". This happens because there usually exist more than one design components for each of the functional specifications corresponding to different design decisions, and in turn, there usually exist more than one implementation components for each of the design ones corresponding to different implementation decisions.

Since the set ALT is going to be used during the matching phase, the membership operation should be as effective as possible. Thus, we assume that this set is implemented via chained hash tables [Lea91]. The first element of each triplet is considered to be the hash key. Therefore, we do not expect to locate a search element in one step. In the worst case the number of the required steps will be equal to the maximum number of attributes that a functional specification resource may have (from our experience this number is of one figure).

The result of each of the above database queries is a set that it is disjoined with the already constructed set ALT (corresponds to a different functional specification resource). Thus, the union operation is equivalent with the insertion of each element of the query result to the set. Since the number of these elements is limited, the complexity of the algorithm that constructs the set ALT is not considerably affected.
The construction of the set $AL$ of all attributes labels associated with the elements of the set $LR$ may be made simultaneously with that of $ALT$, since the desired labels are included in the result of each database query. For the same reasons as in the case of the set $ALT$, the total complexity of the algorithm is not considerably affected.

In the sequel, the system prompts the user for the provision of a specification attribute type for each of the elements of the set $AL$. By the end of this process the set $D$ has been created, which constitutes a description of the target functional specification resource.

Next, the set $D$ is partitioned into the sets $D_1$, $D_2$, and $D_3$, with respect of the "no" or "no-care" values provided by the user. This partitioning does not require any database accesses. The algorithm that performs the partitioning, just scans the set $D$ and depending on the values of the types $t_i$ places the couples $(l_i, t_i)$ into the appropriate subset of $D$. Therefore, the complexity of this algorithm is $O(|D|)$ (measured in comparisons).

Finally, the last algorithm of this phase constructs the set $LR_1$ of the functional specification resources having at least the same specification attributes with that of $D_1$ (the rest are specification attributes that the application developer does not actually care about them).

This algorithms works as follows: It scans the set $LR$, and for each of its elements, locates the triplets of the set $ALT$ that have this element as a key. In the sequel, it checks whether the label of the triplet does not belong to the set $D_2$.

The number of steps required to locate the triplets of the set $ALT$ given their key can not be more than $C_1$ (the number of attributes that a functional specification resource may have). The number of the elements of the set $D_2$ is less than the number of the elements of the set $AL$ and compared to the number of the elements of the set $LR$ is not considerable. Under these assumptions the complexity of this algorithm, measured in comparisons, is $O(|LR|)$.

Next, follows the matching process. The analysis of the complexity of the matching algorithms can be found in the sequel.
8.5 Matching Algorithms

8.5.1 Exact Match

The first of the matching algorithms examines whether an element $s$ of the set $LR_1$ matches exactly the target functional specification resource. In order to do that, the algorithm scans the set $LR_1$ and for all the label-type couples $(l_i, t_i)$ of the set $D_1$ it checks whether the triplet $(s, l_i, t_i)$ is a member of the set $ALT$.

Since the first element of the triplets of the set $ALT$ is considered to be the hash key, and the number of elements of the set $D_1$ is very small ($D_1$ is a subset of the set $D$), the number of the comparisons required by this algorithm in the worst case is:

$$|LR_1|*|D_1|*C_1 = |LR_1|*C_2$$

where $C_1$ is the maximum number of steps required to locate an element of the set $ALT$ given the hash key (this number is equal to the maximum number of attributes a specification resource may have). Therefore, the complexity of this algorithm measured in comparisons is $O(|LR_1|)$.

After a successful match of a leaf resource $s$, the set $S$ of the functional specification resources that constitute a path from a leaf resource to a root resource, must be constructed. The fact that this path is unique (it is a branch of a tree) allows us to optimize this algorithm as follows:

```plaintext
path:=\{s\}; cl:=s;
repeat
    result:=Proj(Rest(isA, class=cl), super_class)
    cl:=result.super_class; path:=path \cup result;
    until result=\{\}
```

The database query of the above algorithm requires a limited number of disk accesses (equal to the maximum number of the superclasses that a class may have) since an index is used on the field `class` of the isA relation. Actually, in our case the number of superclasses of a functional specification resource can not be more than one. The loop of the algorithm is executed as many times as the distance from the leaf resource to the root resource of the hierarchy. Thus, we may easily conclude that this algorithm does not affect the overall performance of the system.
After a successful match of a sub-case of type $C_1$, the system tries to match a sub-case $c_2$ of type $C_2$. In order to do that, the system initially creates the set $SC$ of the specification components that provide as functional specification resource the leaf resource $s$ of the hierarchy $S$. The corresponding optimized query in relation algebra notation is:

\[
\text{Proj}( \text{Join( Rest(attribute, type=s AND attribute\_category="functional")}
\text{class=name,}
\text{Rest(instanceOf, class="SPEC\_Component")))},
\text{class})
\]

Let $N_1$, $N_2$ be the number of elements of the relations attribute and instanceOf respectively. A sketch of the algorithm that evaluates the above query follows:

\[
\text{FOR i:=1 TO } N_1 \text{ DO}
\]
\[
\text{read the } i_{th} \text{ element of the relation attribute}
\]
\[
\text{IF (i.type=s AND i.attribute\_category="functional")}
\]
\[
\text{if (scan(X, "SPEC\_Component")) append X to the target relation}
\]

Since the instanceOf relation is indexed on the field name, instead of scanning sequentially, the matching elements are retrieved directly. Thus, the scan(X,A) function requires in the worst case $C_1$ database accesses, where $C_1$ is the maximum number of meta-classes that a stored class is their instance (6 in our approach). Consequently, the number of disk accesses that are required for the above algorithm are:

\[
N_1 + C_2 \times C_1 = N_1 + C
\]

where $C_2$ is the number of elements of the relation attribute, that have in their field type the value "s". This number is not considerable compared with the number $N_1$. Thus, we may easily conclude that the complexity of the algorithm for the construction of the set $SC$ is $O(N_1)$ (measured in disk accesses).

If the set $SC$ consists of more than one elements, the application developer has to select the most appropriate according to his needs. Such a selection should be based on the additional functional specification resources provided by the members of the set $SC$. The query used for the retrieval of the functional specification resources provided by a member spec of the set $SC$ is expressed in relational algebra as:
Proj( Rest( attribute, class= spec AND attribute_category= "functional" ), type )

Since the attribute relation is indexed on the field class, the number of disk accesses required for each of the above database queries can not be more than the maximum number of attributes (C_1) that a specification component may have. From our experience this number is usually a single digit.

In the sequel, the system tries to locate a design component des, given that the specification component that has already been located is spec. Two different alternatives can be distinguished: i) no design decision has to be made, and ii) a design decision is necessary.

In this analysis, we are interested in the second alternative, which is considered to be more complex (the first one can be considered as a special case of the second).

The first algorithm involved in the design decision making phase is responsible for the construction of the set DES_1. This set consists of the design dependency components d_i interconnected with spec. The corresponding optimized query in relational algebra notation is:

\[
\text{Proj( Join( Rest(attribute, type=spec AND attribute_category= "to\_component")}
\begin{align*}
\text{class=}&\text{name,} \\
\text{Rest(instanceOf, class=}&\text{"DES\_Dependency")}, \\
\text{class)}
\end{align*}
\]

Let N_1, N_2 be the number of elements of the relations attribute, and instanceOf respectively. A sketch of the algorithm that evaluates the above query follows:

FOR i:=1 TO N_1 DO

read the i_th element of the relation attribute

IF (i.type=spec AND i.attribute_category= "to\_component")

if (scan(X, "DES\_Dependency")) append X to the target relation

Since the instanceOf relation is indexed on the field name, instead of scanning sequentially, the matching elements are retrieved directly. Thus, the scan(X,A) function requires in the worst case C_1 database accesses, where C_1 is the maximum number of meta-classes that a stored class is their instance (6 in our approach). Consequently, the number of disk accesses that are required
for the above algorithm are:

\[ N_1 + C_2 \times C_1 = N_1 + C \]

where \( C_2 \) is the number of elements of the relation attribute, that have in their field *type* the value "spec". This number is not considerable compared with the number \( N_1 \). Thus, we may easily say that the complexity of the algorithm for the construction of the set \( DES_1 \) is \( O(N_1) \) (measured in disk accesses).

In the sequel, the set \( NF_1 \) is constructed. This set consists of the couples \((x, d_i)\) where \( x \) is a non-functional specification resource associated with the design dependency component \( d_i \) of the set \( DES_1 \). The algorithm that constructs this set just scans the set \( DES_1 \) and for each element \( d_i \) of this set executes the following database query:

\[ \text{Proj}( \text{Rest} (\text{attribute, class}=d_i \text{ AND attribute_category}="\text{non}_\text{funct}"), \text{class, type}) \]

Since the attribute relation is indexed on the field class, the number of disk accesses required for each of the above database queries can not be more than the maximum number of attributes \( (C_1) \) that a design dependency component may have. From our experience, this number is always of one figure. Thus, the total number of disk accesses that are required for the evaluation of the above algorithm can not be more than:

\[ |DES_1| \times C_1 \quad (O(|DES_1|)) \]

Finally, the application developer selects the best \((nf_i, d_i)\) from the set \( NF_1 \) according to his needs, and the corresponding design component is located with the query:

\[ \text{Proj}( \text{Rest} (\text{attribute, class}=d_i \text{ AND attribute_category}="\text{from}\_\text{component}"), \text{type}) \]

For the same reasons as above, the number of disk accesses required for this query can not be more than \( C_1 \).

After a successful match of a sub-case of type \( C_2 \), the implementation decision phase follows, where the system tries to match to match a sub-case \( c_3 \) of type \( C_3 \). The algorithms used in this phase are similar with the algorithms used for the design decision phase. Thus, the complexity of the algorithm for the construction of the set \( DEP_2 \) (the set of the implementation dependency components interconnected with \( des \)) is \( O(N_1) \) (where \( N_1 \) is the number of elements of the attribute relation), and the complexity of the algorithm for the construction of the set \( NF_2 \) (the set of the couples \((x, d_i)\) where \( x \) is a non-functional specification resource associated with the
implementation dependency component \( d_i \) of the set \( \text{DEP}_2 \) is \( O(|\text{DEP}_2|) \) (measured in disk accesses).

### 8.5.2 Generic Match

A generic match happens when the only difference between the user specifications \( (D_1) \) and the specifications described by an element of the set \( LR_1 \) is that some given specification attributes are of specific type, while that of \( LR_1 \) are parameterized.

In order to match such a resource, the matching algorithm scans the set \( LR_1 \) and for all the label-type couples \((l_i, t_i)\) of the set \( D_1 \) it checks whether the triplet \((s, l_i, t_i)\) or the triplet \((s, l_i, <T>)\) is a member of the set \( ALT \). If the second alternative is true (that is the attribute \( l_i \) of the resource \( s \) is parameterized), a rule associated with the parameterized resource \( s \) has to be proven. The relation that should be included in the schema of the database for storing such rules is:

```
substitute( rule, + component, label, type, body)
```

where the field \( \text{body} \) is used for storing (in some internal form) the part of the rule which follows the "\:-\:" symbol (section 7.3).

The database query used for the retrieval of a parameterization rule is:

```
Proj( Rest(substitute, component=s, AND label=l), body)
```

The evaluation of such a query requires \( C_1 \) disk accesses which is equal to the number of parameterized attributes of the component \( s \) (from our experience this number is usually less than three). The result of the above query is then used for the construction of a new database query which is used for the validation of the parameterization rule. This query consists of a search for a specific tuple of the isA relation (e.g. isA(int, BasicType)). Such a query requires a limited number of disk accesses \( (C_2) \) due to the indexes used on the field \( \text{class} \) of the isA relation. It has to be noted that the validation of the exceptions that a parameterization rule may include, does not require any additional disk accesses, since these exceptions are expressed using inequalities (section 7.7).

The time required for the construction of the query that validates the parameterization rule, and the time required for the validation of the exceptions, are not considerable compared with the time needed for the disk accesses.
8.5 Matching Algorithms

From the above, we may easily conclude that the maximum number of the disk accesses required for a generic match (all the set $LR_1$ is scanned) is:

$$|LR_1| \times |D_1| \times (C_1 + C_2)$$

Since the number of elements of the set $D_1$ is limited and the numbers $C_1$, and $C_2$ are very small and independent of the size of the database, the complexity of the generic match algorithm is $O(|LR_1|)$.

8.5.3 Similarity Match

A similarity match happens when the only difference between the user specifications and that described by the set $LR_1$ is that some types are not exactly the same, but they are similar.

In order to match such a resource, the matching algorithm scans the set $LR_1$ and for all the label-type couples $(l_i, t_i)$ of the set $D_1$ examines which of the following is valid:

- the triplet $(s, l_i, t_i)$ is a member of the set $ALT$,
- the triplet $(s, l_i, <T>)$ is a member of the set $ALT$, and a parameterization rule can be proven, and
- the triplet $(s, l_i, t_j)$ is a member of the set $ALT$, and a modification rule can be proven.

The database accesses required by this algorithm are the same with that of the generic match algorithm for the validation of the first two of the above cases. For the validation of the third case, some extra disk accesses are necessary. These include the retrieval of the modification rule, and some additional accesses for its evaluation. The relation that should be included in the schema of the database for storing such rules is:

$$\text{modification}(\text{rule}, \text{operation}, + \text{type1}, \text{type2}, \text{last_attempt}, \text{body})$$

where the field $\text{body}$ is used for storing (in some internal form) the part of the rule which follows the ":-" symbol (section 7.8).

The database query used for the retrieval of a parameterization rule is:

$$\text{Proj( Rest(modification, type1=t_i AND type2=t_j AND last_attempt="true"), body) }$$
The evaluation of such a query requires $C_3$ disk accesses which is equal to the number of the known as similar types of the type $t_i$. The result of the above query is then used for the construction of a new database query which is used for the validation of the modification rule. This query consists of a search for two specific tuples of the isA relation. Such a query requires a limited number of disk accesses ($2^*C_4$) due to the indexes used on the field class of the isA relation.

Thus, the number of the additional disk accesses required for a similarity match is limited, and in general independent of the size of the database. So we may easily conclude that the complexity of the similarity match algorithm is $O(|LR_1|)$.

### 8.5.4 Optional Match

An optional match happens when the system detects a stored resource resembling the specifications, where in some types which are not the same no similarity has been defined, even though common characteristic exist. In such a case the user may establish a similarity definition between the given types and the repository ones.

In order to match such a resource, the matching algorithm scans the set $LR_1$ and for all the label-type couples $(l_i, t_i)$ of the set $D_1$ examines which of the following is valid:

- the triplet $(s, l_i, t_i)$ is a member of the set $ALT$,
- the triplet $(s, l_i, <T>)$ is a member of the set $ALT$, and a parameterization rule can be proven,
- the triplet $(s, l_i, t_j)$ is a member of the set $ALT$, and a modification rule can be proven, and
- the triplet $(s, l_i, t_j)$ is a member of the set $ALT$, but no modification rule exists or can be proven. However, the user verifies that the types $t_i$ and $t_j$ are similar.

The database accesses required by this algorithm are the same with that of the similarity match algorithm for the first three of the above cases. For the validation of the fourth case, an interaction is necessary between the user and the system. Concerning disk accesses, no additional accesses are required. Thus, the complexity of the optional match algorithm, measured in disk accesses, is $O(|LR_1|)$. 
8.6 Evolution Algorithms

8.6.1 Specific Component Creation

A new specific component is created after a similarity, or an optional match, if the user does not agree with the system's proposal for the creation of a new generic component.

Initially, the algorithm creates a new functional specification resource \((s_j)\) and appends it to the repository. This phase requires

\[ C_1 + C_2 \]

disk accesses (tuple insertions), where \(C_1\) is the number of meta-classes that a functional specification resource is their instance (5 in our approach), and \(C_2\) is the number of its attributes (\(|D_j|\)). Since the number of the attributes that a functional resource may have is of one figure, this phase does not affect the performance of the system.

Next, the system places this new functional specification resource in the appropriate position of the functional specification resources hierarchy (\(s_j\) becomes a special case of a resource \(s_k\)). This position is located by scanning the set \(S_i\). Since the set \(S_i\) is in the system's memory (it has been created during the matching phase), no additional disk accesses are needed, except the insertion of an isA tuple which defines that the resource \(s_j\) is a special case of the resource \(s_k\).

In the sequel, a new specification component is created. This phase requires a limited number of disk accesses (tuple insertions), since all the related information is already known to the system. Finally, the creation of the new design and implementation dependency components requires just 7 disk accesses, as it can be easily seen in section 7.2.

From the above, we may easily conclude that the total complexity of the specific component creation algorithm is not considerable.

8.6.2 Generic Component Creation

A new generic component is created after a similarity, or an optional match, if the user agrees with the system's proposal for such a creation.
This phase requires $C_1$ tuple updates, equal to the number of attributes of the functional specification resource $s_i$ that need to be parameterized, in order to update their types with the symbol "$<T>$". These attributes are already known to the system (from the matching phase), thus no additional accesses are required for them.

The re-placement of this modified functional specification resource in the appropriate position of the functional specification resources hierarchy is similar with that of the specific component creation. It only requires one additional tuple deletion, which removes the tuple corresponding to the specialization of the functional specification resource $s_i$.

During this phase, a set of parameterization rules associated with the resource $s_j$ (the modified resource $s_i$) are also appended to the repository. Each rule corresponds to one of the parameterized attributes of the functional specification resource. The construction of each rule does not require any additional disk accesses, since all the necessary information is already known to the system. Thus, the total number of disk accesses required during this phase is equal to the number of parameterization rules ($C_1$). From our experience this number is usually less than three.

Concerning the modification of the design dependency component (section 7.3), the number of disk accesses required (tuple updates) is equal to the number of attributes that it has (this number is always a single digit). The same holds for the modification of the implementation dependency component.

From the above, we may easily conclude that the total complexity of the generic component creation algorithm is not considerable.

### 8.6.3 New Code Insertion

This is the case where the application developer provides the system with the new source code descriptions. Concerning the sub-cases $c_{j1}$, the disk accesses required (tuple insertions) are equal with that of the specific component creation. For the sub-cases $c_{j2}$ and $c_{j3}$ some additional disk accesses are required which correspond to the insertion of the descriptions of the implementation component $impl_j$, and the descriptions of the corresponding design component $des_j$. 
The number of the facts corresponding to these descriptions depends on the size of the source code. From our experience this number is usually of two figure per class defined in the source code. Thus, the total time required for the insertion of new source code does not affect seriously the performance of the system.

### 8.6.4 Decision Creation

A new decision must be created after a successful match according to the user's functional specifications, if the user is not satisfied with the proposed solution due to the violation of some non-functional specifications. Two alternatives can be distinguished:

i) **Creation of a new implementation decision.** The number of the disk accesses required for the creation of a new implementation decision is equal to:

\[ C_1 + C_2 + C_3 \]

where \( C_1 \) is the number of the instanceOf and attribute relations (tuples) described in section 7.5, \( C_2 \) is the number of the relations corresponding to the description of the violated non-functional specification (usually of one figure per specification), and \( C_3 \) is the number of relations that comprise a description of the new implementation component.

ii) **Creation of a new design decision.** The number of the disk accesses required for the creation of a new implementation decision is

\[ C_1 + C_2 + C_3 + C_4 \]

where \( C_4 \) is the number of the tuples that comprise a description of the new design dependency component.

For the same reasons as above, we may easily conclude that the creation of new design and implementation decisions does not affect considerably the performance of the system.

### 8.6.5 Specifications Refinement

This algorithm is related with tuple-at-a-time operations (append, update, or delete attributes of a specific resource). Since the attribute relation is indexed on the field class, all of these operations require a limited number of disk accesses.
Concerning the case of attribute deletion, the system verifies that this attribute \( (l_k) \) is not inherited from a functional specification resource that is more general from the resource that is refined. In order to do that, the set \( S_j \) is scanned, and for all of its elements \( s \), the system verifies that the triplet \((s, l_i, x)\) is not a member of the set \( ALT \). These two sets have been constructed during the matching phase. Thus, no additional disk accesses are required for this verification.

On the other hand, during the attribute adaptation the system verifies that in the case that the attribute with label \( l_k \) is inherited, the new type \( t_m \) is a special case of the types of the more general attributes. In order to do that, the set \( S_j \) is scanned, and for all of its elements \( s \) such that \((s, l_k, x)\) belongs to the set \( ALT \), it check whether the fact 
\[
isA(t_m, x)
\]
exists in the database. Thus, the number of the disk accesses required for such a verification is in the worst case equal with the maximum number of the superclasses that a class may have. This number is very small compared with the size of the isA relation.

### 8.6.6 Parameterization Rule Refinement

This algorithm consists of the following steps. First retrieves the parameterization rule, next it updates the field body with the new exception, and finally restores it in the database. Since the substitute relation is indexed on the field component, the number of disk accesses required by this algorithm is twice the number of the parameterization rules that a component is associated with (usually less than three).

### 8.6.7 Learning about Similarities

This algorithm creates a new modification rule and appends it in the database. Also, some tuples that describe differences are appended to it (if there exist additional source code differences than type substitution). Since all the related information is already known to the system, the disk accesses required by this algorithm are limited.
8.7 Summary

The overall cost of the system is composed of the DBMS cost and the cost of the user efforts to work with the system. Concerning the DBMS cost, the database queries involved in our method are limited to a set of standard queries performed during the case retrieval and adaptation phase. Thus, they have been optimized manually by programming the associated procedures and restricting the user's input to the provision of some types when prompted by the system.

From the complexity analysis of the algorithms involved in our method, it has been conjectured that a limited number of disk accesses are required for most of them. This guarantees that all the phases of the proposed method will be very efficient.

The method presented in this thesis, has been evaluated through a prototype implementation which addresses the reuse of C++ code. Part IV presents the prototype implementation, a sample session that illustrates the features and the "feel" of the prototype system, and a usage experiment performed in order to get an indication on the usage characteristics of the prototype system.
Part IV

Examples and Illustrations
Part Introduction

This part presents a prototype implementation which addresses the reuse of C++ code using the method described in part III. In particular:

Chapter 9 An overview of the prototype system, and a sample session that illustrates the features and the "feel" of the prototype implementation are presented. Furthermore, this chapter describes a usage experiment performed in order to get an indication on the feasibility of the proposed approach and the usage characteristics of the prototype system.
Chapter 9

Examples and Illustrations

9.1 Introduction

A prototype implementation is presented which addresses the reuse of C++ code using the method described in part III. The prototype system runs on SparcStations and Sun4 series under Unix. The language used for the implementation is a Prolog-like language called MegaLog.

Initially, the C++ case is presented. Next, follows an overview of the prototype system, and a short presentation of the sub-components that are embodied in it. Also, this chapter includes a sample session that illustrates the features and the "feel" of the prototype implementation.

Finally, a usage experiment performed in order to get an indication on the feasibility of the proposed approach and the usage characteristics of the system built, is presented.

9.2 The C++ Case

C++ was developed from the C programming language and with very few exceptions retains C as a subset. The name C++ is a quite recent invention. Early versions of the language, collectively known as "C with Classes", have been in use since 1980 [Strou83]. The language was originally invented because Stroustrup wanted to write some event-driven simulations for which Simula67 would have been ideal, except for efficiency considerations.
C++ has become very popular as it adds polymorphism and inheritance to C in order to support object-oriented programming, while maintaining the speed and efficiency of C. The difference between C and C++ is primarily in the degree of emphasis on types and structures. C is expressive and permissive. C++ is even more expressive, but to gain that increase in expressiveness, the programmer must pay more attention to the types of objects.

It has been argued that, the largest single problem using C++ is the lack of an extensive standard library. A major obstacle in producing such a library is that C++ does not provide a sufficiently general facility for defining parameterized types. In our approach, we have integrated into C++ the general form of parameterized types described in chapter 2. Within this framework a generic class can be defined having one or more parameterized types namely $<T>$, $<T_1>$, $...$, $<T_N>$.

Using this technique, it is possible to maintain the efficiency of C++ code (as we make the substitution before compiling), while retaining the benefits of genericity.

The methods presented in the previous chapters may be easily applied for reusing implementation components written in any object-oriented language. The reasons for our decision to store and reuse C++ code are: 1) the popularity of the C++ language, 2) the efficiency of the C++ code, and 3) the fact that C++ does not provide a built-in form of genericity.

Our system may be easily extended for storing and reusing source code written in any object-oriented language. What is needed, is a filter that takes as input source code of the language, and produces as output the corresponding implementation component descriptions.

### 9.3 System Overview

The prototype system runs on SparcStations and Sun4 series under Unix. The language used for the implementation is a Prolog-like language called MegaLog [Horsf90]. MegaLog provides a powerful programming environment for building next generation Database and Knowledge Base Management Systems.

MegaLog integrates a knowledge base with a logic programming language to provide large scale persistent storage of knowledge in a way that it can be efficiently accessed and processed by logic programs. MegaLog has all the features of a database system (multi-user, recovery, efficient set-oriented retrieval, secondary storage management) and uses the relational data model as a basis.
Furthermore, it allows for complex data types storage of deductive rules together with the facts, while incorporating standard Prolog syntax and procedural semantics.

Figure 9.1 depicts the architecture of the prototype system. The user has access to the repository through the Case-Based Reasoner which is also responsible for the control of the different filters that are embodied in our system.

![Diagram of the Prototype System](image)

**Figure 9.1: General Architecture of the Prototype System**

The system's repository is the medium used to store and subsequently locate the matching sub-cases described in chapter 5. In order to express the different kinds of components that constitute a matching case as MegaLog facts, we used the term of proposition. Formally, propositions are triplets with components "from", "label", and "to" denoting the source, label and destination of proposition. Propositions are grouped into classes and related to other propositions through attributes, which are also propositions. These propositions may be easily stored as MegaLog deductive relations. In our prototype implementation, we used the following three types of propositions:

- The instantiation proposition defined as $(x, \text{instanceOf}, \text{class})$ which represents instantiation relationships.
• The isA proposition defined as \((x, \text{isA}, \text{superclass})\) which represents specialization relationships.

• The attribute proposition defined as \((x, \text{label}, \text{value})\) which denotes that \(x\) has an attribute labeled \text{label} with value \text{value}.

Note that, the attribute proposition is defined without the attribute category field used in the definitions of the previous chapters. This happens because a data normalization has taken place. The attribute categories of an attribute are derived by selecting the attribute classes that this label is defined as their instance.

The predicate used to define a deductive relation is \(<===>/2\), where the first argument is the relation’s name and the second the schema. The full syntax is as follows:
\[
\text{Relation\_Name} <==>[\{+\}\text{Attribute\_Name1}, \{+\}\text{Attribute\_Name2}, \ldots, \{+\}\text{Attribute\_NameN}]
\]
where:

• Each \text{Attribute\_Name} is preceded by a + when the attribute is to be included in the key.

• The attribute type and the field length do not need to be specified, as MegaLog allocates memory as required.

To define the instantiation proposition the following predicate has been used:
\[
\text{instanceOf} <==>[\ +\text{name, class}]
\]

In the same way we define the isA relation as:
\[
\text{isA} <==>[\ +\text{class, superclass}]
\]
and the attribute relation as:
\[
\text{attribute} <==>[\ +\text{class, label, type}].
\]

Clauses are added to these relations using the predicate \text{insert\_clause(Clause)}. For example:
\[
\text{insert\_clause}(\text{in(SPEC3, FunctionalSpecification)}),
\text{insert\_clause}(\text{attribute(SPEC3, name, "Insert")})
\]
defines the functional specification resource \text{SPEC3}, with name \text{Insert}.

To unify a goal with a clause in the knowledge base, the \text{exec(Goal)} predicate is used. This retrieves and executes clauses from a relation to find a clause that unifies with the goal. If we
want to find the functional specification resource with name *Insert*, the corresponding goal has the form:

```
exec(in(_x, FunctionalSpecification)),
exec(attribute(_x, name, "Insert"))
```

Using this kernel mechanism we are able to represent the different components that constitute a matching case, which include design, specifications and process artifacts. This representation encompasses two layers, the first consisting of meta-classes which model features of the components is application-independent, while the second one consisting of simple classes which represent components participating in a particular application development is application-dependent.

As mentioned before, the MegaLog environment provides the capability of storing in its database rules together with facts. Thus, all the rules associated with the generic components and their parameterizations are stored in the repository. Also, we store in it, all the information concerning similarity of types and the anticipated previous mistakes. New rules are dynamically appended to the repository as the system evolves over time.

Through a rather simple interface, the application developer (user of the system) may have direct access to the Case-Based Reasoner of the system which is responsible for:

- the representation of the matching cases,
- the indexing of matching cases,
- the case storage and retrieval,
- the adaptation of cases to fit a new case,
- the evolution of the repository, and
- the control of the different filters used by the system.

Next, follows a description of the different filters used by the system (Figure 9.1).

**1. Filter Telos2MegaLog.**

Expressing the meta-model of components using the low level concept of propositions, is a very
hard process. Thus, we employ the conceptual language Telos as a basis for these definitions. After the Telos code is parsed and semantically checked, it is possible to be converted into MegaLog facts, using the special filter Telos2MegaLog.

This filter takes as input Telos source code and produces the corresponding MegaLog facts. For instance, consider the definition of the meta-class design component:

```plaintext
IndividualClass DES_Component in M1_Class isA Component with
   attribute
       adt: AbstractDataType;
   operations: Operation
end DES_Component
```

After passing this code through the filter Telos2MegaLog, the following MegaLog facts are produced:

```plaintext
instanceOf(DES_Component, M1_Class)
isA(DES_Component, Component)
attribute(DES_Component, adt, AbstractDataType)
instanceOf(adt, attribute)
attribute(DES_Component, operations, Operation)
instanceOf(operations, attribute)
```

The filter uses the Telos parser and semantic checker, and has been implemented in the C++ language. The format for calling this filter is:

```
Telos2MegaLog filename
```

where `filename` is the name of the file with the Telos source code.

2. Filter MegaLog2Telos.

This filter performs the inverse task than the previous one. It reconstructs Telos source code using the MegaLog facts that are stored in the repository. It is useful for demonstration reasons, as the Telos code can be easier understood than the corresponding MegaLog facts.

The language used for its implementation is MegaLog, and it is called from the MegaLog environment as:

```
?-MegaLog2Telos(classname)
```
where \textit{classname} is the name of the Telos class to be reconstructed.


This filter automatically creates a specific implementation component from the corresponding generic one. As explained in sub-section 6.3.4, the filter is automatically called by the system after a generic match. The specific types that substitute the parameterized ones are derived automatically by the system, using the information collected during the matching process. Next section presents a detailed example of such a substitution.

The filter is a modified version of the GNU \textit{genclass} shell-script, and it is called as:

\begin{verbatim}
Gen2Spec [-2] type 1 [type 2]
\end{verbatim}

where the "-2" flag denotes that two parameterized types \textit{<T>} and \textit{<T1>} are going to be replaced with the specific types \textit{type 1}, and \textit{type 2} respectively. If this flag is missing, the unique parameterized type \textit{<T>} of the generic component, is replaced with the specific type \textit{type 1}.


This filter creates in a semi-automatic fashion a generic implementation component from two specific ones. As mentioned in section 7.3, a new generic component is created after a similarity (or an optional match), if the user agrees with the proposal of the system. The two specific implementation components are the one located in the repository (file \textit{impl1}), and the new one created after the proper modifications applied to the first one by the user (file \textit{impl2}).

The creation of a new generic component has as follows (for simplicity reasons assume that only one type is parameterized). Initially, the differential file comparator of unix \textit{diff} is used:

\begin{verbatim}
diff impl1 impl2 > impl.diff
\end{verbatim}

Next, the output file of the \textit{diff} operation (\textit{impl.diff}) is edited, using the \textit{vi} editor of unix. The editing session is started automatically using the "-c" option which executes the following command:

\begin{verbatim}
1.$s/type2/<T>/gc
\end{verbatim}

This command substitutes the \textit{type2} (where \textit{type2} is the type that replaced its similar type \textit{type1} in the component located in order to get the new implementation component) with the parameterized type \textit{<T>}. Also this command demands that each substitution has to be confirmed by the user. This is the reason that this filter is not fully automatic.
Finally, the system considers the new \texttt{impl.diff} file as a patch file to the initial file \texttt{impl1}, and the generic implementation component \texttt{gen_impl} is ready:

\begin{verbatim}
patch -f -s -o gen_impl impl.diff
\end{verbatim}

The file of the differences \texttt{impl.diff} is not removed as it may contain further modifications made (than type substitution), as explained in section 7.8. This file is used during the creation of a specific component from this generic one, by the filter \texttt{Gen2Spec} described above.

4. Filter \texttt{cc2MegaLog}.

This filter takes as input C++ source code, and automatically produces the MegaLog facts corresponding to the implementation component descriptions described in chapter 4. A new implementation component is created for each of the classes defined in the source code file. In addition to them, the corresponding design components are also created. In order to create this filter, the GNU g++ parser of Free Software Foundation Inc has been modified. It may be called using the command:

\begin{verbatim}
cc2MegaLog filename
\end{verbatim}

where \texttt{filename} is the name of the file with the source code. When you invoke \texttt{cc2MegaLog}, it normally does preprocessing, and parsing. It has to be noted that, the name of the file of the source code must ends in ".cc".

It should be clear that, when the filter \texttt{cc2MegaLog} is combined with the filter \texttt{MegaLog2Telos}, it is possible to have the Telos representation of the implementation components corresponding to the C++ classes defined in the file of the source code.

For the time being the system provides a rather simple interface. As soon as the system is started, the main menu of the system appears of the screen. Next follows a short description of the different choices presented by this menu:

1. Repository Operations. The selection of this menu item causes the activation of a submenu related with operations on the system’s Knowledge Base. In summary these operations are:

i. New KB. Causes the creation of a new Knowledge Base. The new Knowledge Base consists of the meta-model of the components that may be stored in the repository. It is considered blank in the sense that no matching cases are stored in it. The
application developer may append his code and evolve the repository. During the creation of a new Knowledge Base, the system prompts the application developer for a name identifying it.

ii. **Open KB.** This item may be selected in order to have access in a different Knowledge Base than the current one. The selection of a new Knowledge Base is made by the provision of its name.

iii. **Current KB.** Provides the name, schema, arity and number of clauses for each of the relations in the opened Knowledge Base.

iv. **List Kbs.** Lists the different Knowledge Bases that may be opened.

v. **Previous menu.** This item is selected in order to return to the basic menu of the system.

2. **Show a Component.** Presents the Telos description of a component stored in the repository.

3. **Selection Phase.** Activates the selection process of the system. A sample session of this process can be found in section 9.4.

4. **Code Insertion.** This menu item may be selected in order to append new source code in the repository, as described in section 7.4.

5. **Specifications Refinement.** Starts the specification refinement process. A detailed description of this process can be found in section 7.6.

6. **Exit System.** This menu item is selected in order to leave the system and return to the calling process.

Next, follows a sample session that illustrates the features and the "feel" of the prototype implementation.
9.4 A Sample Session

Since the objective is not to present all functionalities of the system but to illustrate the way the method is applied, we present only a simplified version of the interaction. The first scenario assumes a user looking for an implementation component which inserts an integer into a queue. Such a component does not exist in the repository, but there exists a generic component for queue operations.

The third choice of the main menu starts the selection process. Initially, the user has to specify if he wishes to search the specification resources paths manually, or to use the CBR-based retrieval method provided by the system. If the second alternative is selected, the system prompts him for a resource name that describes the component he searches for. After giving to the system the name "Insert", the user proceeds with clarifying information (defining types for the specification attributes associated with the name). To the system's prompt:

Insert where? [array, linked_list, queue, bag, set]:

the user can choose any of the suggested known (to the system) answers or introduce a new one. He proceeds by requiring search in a "queue". In the same way, he may answer to the prompt:

Insert what? [char, int, float]:

by int.

After all the attribute types have been defined the system starts the matching process at the specification resources level. Suppose that no specification resource exists that matches exactly the needs. After notifying the user for this, the system checks if there is a generic component that meets the user needs. If such a match is possible the system reports it in the following way:

No Specific Component that matches was found in the KB

A Generic Component that matches was found:


EXPLANATION: "Insert an element into a queue"
DESCRIPTION:
NAME: "Insert"

ATTRIBUTE: what: <T>
ATTRIBUTE: where: queue

Parameterized type <T> can take the value "int" because:

IF isa(int, BasicType) THEN
substitute(rule1, SPEC3, what, int)

isa(int, BasicType) is a fact in the Knowledge Base

In the sequel, the system follows the dependency links as explained in the previous chapter, and prompts the user for making the appropriate design and implementation decisions. The design decision is related with the way that the queue is expanded (dynamically-sized or static), while the implementation decision has to do with the way that the abstract data type queue is implemented (using linked-lists, dynamic arrays, static-arrays etc).

In the sequel, the system automatically substitutes the parameterized type <T> with the type "int" in the file located, and reports to the user the name of the new file created with the corresponding specific code.

A second scenario assumes a user looking for an implementation component which searches a file of integers. The system does not have information about searching in files, but it does know about searching for characters in arrays. Also it has been told that arrays and files are similar structures as they are special types of tables.

After giving to the system the name "Search", and attribute values "file" and "int" for the attributes "where" and "what" respectively, the system starts the matching process. Suppose that no specification resource exists that matches exactly the needs. After notifying the user for this, the system checks whether there is a generic component that meets the needs. An unsuccessful search guides the system into an attempt to identify similarities between the given types and the types of specifications resources with basic name Search. The system reports them in the following way:
A SIMILAR Specification (SPEC102) has been found:

EXPLANATION: Search for a character in an array

DESCRIPTION:
NAME: "Search"

ATTRIBUTE: what: char
ATTRIBUTE: where: array

DIFFERENCES:
what values are not the same (char, int) and
where values are not the same (array, file)

Concerning what:

In a SIMILAR case in the past, rule RULE2 provided a solution.

RULE Description:
modification(RULE2, substitute, char, int, true)

RULE Proof:
IF isA(char, BasicType) AND isA(int, BasicType)
THEN modification(RULE2, substitute, char, int, true)

isA(char, BasicType) is a fact in the Knowledge Base
isA(int, BasicType) is a fact in the Knowledge Base

ACTION:
substitute (char,int)
Concerning where:

In a SIMILAR case in the past, rule RULE5 provided a solution.

RULE Description:
modify(RULE5, substitute, array, file, true)

RULE Proof:
IF isA(array, table) AND isA(file, table)
THEN modify(RULE5, substitute, array, file, true)

isA(array, table) is a fact in the Knowledge Base
isA(file, table) is a fact in the Knowledge Base

ACTION:
substitute (array, file)

DIFFERENCES found for the following operations:
array: reset="set pointer to 0"
file: reset="function fopen"
array: next_element="pointer increment"
file: next_element="function fscanf"
array: last_element="pointer==size"
file: last_element="function feof"

DO YOU WISH TO PROCEED WITH THESE MODIFICATIONS? [Y/N]

In this case the user has to take the final decision if the substitutions proposed by the system meet his requirements. If the user does not agree, then he will have to select one of the evolution methods and provide the appropriate information. In the sequel, the system will be reorganized in order to avoid the same suggestion in a similar case in the future. However, in our case the
answer is positive as these substitutions will provide solution to the problem at hand. Next, the system follows the dependency links and presents to the user the implementation component and the corresponding file where the substitutions will be made.

In the sequel, the user is asked if he wants to create a generic component for searching tables. If the answer is negative the system is automatically reorganized and the only thing the user has to provide is a description of the new specific component created (in our case "Search for an integer in a file"). If the answer is positive then the system using the filter Spec2Gen described before, creates a new generic component. Finally, all the information regarding the searching for a character in an array is removed, the system is reorganized with the information related to the new component created, and the selection process is terminated. Next follows the actual listing of the generic component created from this run:

class <T1><T> {
  protected:
    <T1> *<T>;
    <T1> *current_element;
    int size;
  public:
    <T1><T>(int);
    void reset();
    int search_element(<T1>);
    int found(<T1>);
    void next_element();
    int last_element();
};

inline int <T1><T>::last_element() {
    return(current_element==(<T>)+size-1));
}

inline int <T1><T>::found(<T1> item) {
    return(*current_element==item);
}
inline void <T1>::next_element() {
    current_element++; }

inline void <T1>::reset() {
    current_element=<T1>; }

<T1>::<T1>(int sz) {
    <T1>=new <T1> [size=sz];
    current_element=<T1>; }

int <T1>::search_element(<T1> item) {
    reset();
    while (!found(item) & !last_element())
        next_element();
    return(found(item)); }

Note that the protected class parts and the operations reset, last_element, and next_element have to do with arrays and not files. A patch file which contains their differences with the corresponding file parts and operations is also stored in the repository, in order to be used when generating a specific component which searches files for an element.

9.5 Usage Experiment

This section presents a laboratory experiment performed, in order to get an indication on the feasibility of the proposed approach and the usage characteristics of the system built. The experiment was based on an application environment consisting of 52 GNU C++ library classes [Lea91]. The system's repository evolved in an incremental way with the provision of the appropriate descriptions, and the making of the necessary design and implementation decisions. Furthermore, several generic components were created by identifying type similarities between specific ones.

The usage experiment involved 79 Computer Science undergraduate students (3rd, 4th year). It comprised the following steps:
1. One-hour tutorial on the system for all subjects, giving a general presentation of the system's functionality while paying special attention to the CBR-based retrieval method.

2. Training of the subjects in the system usage. This step has not received the attention it deserved, because of time constraints. Training was limited to few instructions that were given to the subjects just before they started performing the experiment.

3. The usage experiment performed individually by each subject according to a prescribed usage scenario, identical for all subjects. The scenario involved the following (see Appendix A):
   - Search for a component that existed in the repository.
   - Search for a component that did not actually exist, but a similar one did. In a successful similarity match, the generation of the corresponding generic component was asked.
   - New code insertion.

   During the usage experiment, each subject was watched, taking notes on reactions and usage patterns. The author was also acting as the "reference manual" of the system for the subjects.

4. Immediately following the usage experiment, the subjects filled out one questionnaire evaluating the general features of the prototype system, and the specific functions involved in the experiment (see Appendix B).

Finally, the notes taken while watching the subjects, and their answers to the questionnaire were processed. Since the processed results do not have an important statistical significance, we present only the conclusions drawn from this experiment.

First of all, the system's assistance for the construction of the target functional specification resource (the system prompts the user for some specification attribute types) can be characterized adequate. The subjects did not have any serious difficulties during this phase.

Also, the subjects where making easily the necessary design and implementation decisions using the non-functional specification resources provided by the corresponding dependency
components. It has to be noted that the subjects had to make decisions that they were not familiar with. For instance, the abstract data type ordered set may be implemented using Sleator and Tarjan's splay trees, or linked lists. (Figure 9.2). Although the subjects had no previous knowledge about splay trees, they correctly selected this implementation decision by noticing the non-functional specification resource Spec62 which describes an efficient membership operation (see Appendix A, usage scenario 2). The latter verifies that it is important to store not only the decisions themselves, but also the non-functional specifications that they satisfy.

![Diagram of implementation decisions for ordered sets]

Figure 9.2: Implementation decisions for ordered sets

Another interesting observation was that the filter Spec2Gen provided adequate assistance for the creation of new generic components. Most of the subjects created such a component successfully.

Finally, it has to be noted that when the subjects were describing the new specifications corresponding to the implementation component array (Appendix A, usage scenario 3), they proposed some extra specification attributes than what we expected (e.g. dimension: single corresponding to the dimension of the array). It is possible that such attributes will be necessary in the future. This verifies the need of the specification refinement phase, since it is not possible to predict all the necessary specification attributes when building a new specification resource.

In summary, the results of this experiment were very encouraging having in mind that the users had no previous systematic knowledge or usage experience of the system. However, it should be
noted that a reliable evaluation of the proposed approach will be possible only by the installation of our system in a software development organization, where multiple users will evaluate the proposed method through system usage. Next, follow the results of the questionnaire filled-up by the subjects.

The system was characterized easy at use. The information layout as plain text, was found easy to read and well organized. However, the subjects would be much more pleased if the interface of the system was enhanced with graphical representations and mouse usage.

Considering the answers that were given for learning of the system, we may say that a combination of reading and 'trial and error' is the most suitable.

Also, both retrieval methods (manual retrieval, and CBR-based retrieval) were found easy in use. However, the CBR-based retrieval was found more appropriate. In addition, the role of design and implementation decisions during the retrieval phase was characterized very important.

Finally, concerning the repository evolution (generic component creation and new code insertion) the results were very encouraging, as the subjects found the evolution operations easy in use, and the assistance provided by the system very useful.

It is worth to mention that a few of the subjects had the feeling that they were giving an examination, and not that they were the ones who were examining the whole system through. However, most of them understood the importance of their contribution not only to the extend of being trustworthy at their answers, but to the degree of what was actually being done and what it could mean for the project itself. Finally, most of the subjects would like to spend more time with the system to learn it and test it more, after the first contact with it.

### 9.6 Summary

A prototype implementation has been presented. The prototype system addresses the reuse of C++ code using the method described in part III. The reasons for our decision to store and reuse C++ code were: 1) the popularity of the C++ language, 2) the efficiency of the C++ code, and 3) the fact that C++ does not provide a built-in form of genericity.
The language used for the implementation is a Prolog-like language, called MegaLog, which provides a powerful programming environment for building next generation Database and Knowledge Base Management Systems.

An overview of the prototype system has been presented, followed by a description of the special-purpose filters that are embodied in it. Furthermore, an illustration of the features and the "feel" of the prototype implementation has been accomplished through a sample session.

Finally, a usage experiment and the processed results have been presented. The experiment was based on an application environment consisting of 52 GNU C++ library classes. The experiment involved 79 Computer Science undergraduate students. It performed individually by each subject according to a prescribed usage scenario, identical for all subjects.

The results of the usage experiment are very encouraging, considering that the participants had no previous systematic knowledge or usage experience of the system. However, it should be noted that a reliable evaluation of the proposed approach will be possible only by the installation of our system in a software development organization where multiple users will evaluate the proposed method through system usage.

Chapter 10, presents the conclusions of this work, followed by some directions for further work.
Part V

Conclusions and Extensions
Part Introduction

This last part concludes the thesis and suggests some directions for further work.

Chapter 10 The major achievements of this thesis are summed up, followed by some guidelines for further work.
Chapter 10

Conclusions - Future Work

Object-oriented programming is, by nature, closer to reuse than any traditional way of programming. Inheritance which allows for the incremental definition of objects behavior, and genericity which allows for a module to be defined with parameterized types, provide a definite aid towards software reuse.

In order to systematically reuse object-oriented code, problems like selection, adaptation and repository evolution have to be addressed. The conjecture is that in order to overcome these problems, knowledge gathered during the creation of similar software in the past should be also modeled and reused. This is possible by the application of AI techniques to software engineering. A relatively new reasoning methodology called Case-Based Reasoning provides considerable assistance for modeling and reusing such knowledge.

Under this perspective, a novel method has been developed that employs a special form of Case-Based Reasoning in conjunction with the specificity-genericity hierarchy to semi-automatically locate the appropriate code in a software repository, possibly adapting it to particular requirements, while dealing with the evolution of the repository by adding new components and making the proper repository re-organization.

At the theoretical level, the work reported in this thesis can be viewed as an integration and extension of ideas and techniques from the fields of Software Engineering, Knowledge Representation and Machine Learning.
In particular, the CIL framework has been extended and modified in order to model the software components and their relationships in a way that the programming knowledge is captured, and the formation of matching cases is enabled.

The organization of cases in combination with the knowledge-based indexing used in our approach, assures that only small parts of the case memory are accessed during the reasoning process. This process assumes a semi-automatic environment where the application developer (user) interacts with the system. It consists of two phases: case retrieval, and case adaptation.

The goal of case retrieval is to return the most similar past case that is relevant to the input situation. Case adaptation takes a retrieved case that meets most of the needs of the current situation and turns it into one that meets all of the situation's needs. Two kinds of adaptation mechanisms have been used: a) structural adaptation (a type substitution takes place), and b) derivational adaptation (a design or an implementation decision is made).

Finally, a major concern in this work has been the development of a method for controlled expansion and evolution of the repository. This method is based on explanation based generalizations and failure driven learning. As cases accumulate, case generalization is used to define prototypical cases that embody the common features of a group of specific cases, while the rest of their features become parameterized. Furthermore, in cases that a failure takes place, the case memory is reorganized in a way that the accuracy of the system is improved in the long run.

At the practical level, an evaluation of the proposed method has been accomplished through a prototype implementation which addresses the reuse of C++ code. The prototype system runs on SparcStations and Sun 4 series under Unix. The language used for the implementation is a Prolog-like language called MegaLog. This language integrates a knowledge base with a logic programming language to provide large scale persistent storage of knowledge in a way that it can be efficiently accessed and processed by logic programs.

Additionally, we performed a laboratory experiment in order to get an indication on the feasibility of the proposed approach and the usage characteristics of the system built. The usage experiment results are very encouraging, considering that the members of the evaluation team had no previous systematic knowledge or usage experience of the system.
The proposed method can be extended in many ways. At the theoretical level, an important issue that should be investigated, is the extensibility of the method for reusing applications, not just code. Initially, the software components that are stored in the repository have to be organized in a certain way that will enable the formation of application components. The context mechanism described in chapter 4 could be a good starting point. In the sequel, the following have to be determined:

- The way that the application components and the additional knowledge related to the application development process should be organized in order to form matching cases,

- the indexes that should be used on these matching cases,

- the case storage, retrieval and adaptation, and

- the evolution of the repository.

Concerning the retrieval of similar past cases and their adaptation, additional similarity types should be explored. Semantic similarity is a good candidate. Since parameterized components are created using type similarities, additional to type parameterization ways have to be employed for the creation of new generic applications.

Furthermore, the configuration problem is a concern in all development environments. It is the problem of deciding how individual components retrieved from the repository can be configured in order to form a new application.

Finally a quality assurance study is necessary. The entire software production cycle should be integrated and organized in such a way as to provide a strong basis for controlled evolution and expansion of the repository with new applications of high quality.

At the practical level, additional system development is needed to turn the prototype system into a functional system. First of all, a better user interface must be provided. Such an interface should include browsing facilities, and interactive menus with descriptions of choices in various levels of detail, permitting users of various skills and experience to use the system.

Also, in considering its application in practical situations several operational characteristics like synonym handling (the system must come up with the same solution, independently if the user
provides the name "Lookup" or "Search"), provision of an undo operation, etc., need to be resolved.

Furthermore, a re-implementation of the prototype system is necessary for better performance. This new implementation should be based on a more efficient programming language than MegaLog. An important issue that should be a concern, is the application of parallel processing techniques to the candidate components during the matching phase.

Finally, an important issue that should be investigated, is the applicability of our method to large scale real world problems. This will be possible only by the installation of our system in a software development organization where multiple users will evaluate our method through system usage. Such an experiment will help us to give answers to the following questions:

- What are the accessing privileges that should be granted to the different (according to their experience) users of the system?

- Who will be responsible for the evolution of the repository?

- Is a view mechanism necessary? If so, who will be responsible for deciding whether some local changes should become global?

Concluding, the approach proposed in this thesis suggests a novel way of thinking about repository evolution and usage, which emphasizes the software developer's assumptions and justifications. This re-orientation can be of particular importance. However, in order to get the desired productivity and quality improvement, a major cultural change has to take place in all organizations. The system developers have to adjust to reusing instead of creating from scratch. The software reuse community should also work towards this direction.
APPENDIX A

Usage Scenario

1. Find the source code which corresponds to the implementation of the well known data structure queue. This queue is going to be used for storing and extracting some integers in a network application. You should have in mind that there is not an upper limit for the queue size, and the operations of insertion and extraction should be as fast as possible. In this scenario you will use two different retrieval methods:

   - Manual retrieval, and
   - CBR-based retrieval.

2. Find the source code which implements the abstract data type set with all the appropriate operations. The elements that are going to be stored in the set are real numbers. You should have in mind that among the set operations you are most interested in comparison and set membership. These must be as fast as possible.

   It has to be noted that such code does not actually exist in the repository. However, there exists a similar code which with minor modifications will meet your needs. After finding this code and applying the appropriate modifications, give a positive answer to the system’s prompt for the creation of a new generic component. Test this new generic component by searching for a set of characters.
3. The file "array.cc" contains the source code which implements the well known abstract data type array with all the corresponding operations. Such code does not exist in the repository. Using the fourth choice of the system’s main menu (Code insertion), put this code in the system’s repository and provide the appropriate descriptions.
APPENDIX B

Questionnaires and Results

SCALE
[1] not at all
[2] little
[3] enough
[4] much
[5] very much

Mean values of answers are reported
Sample size = 79

Questionnaire
OVERALL REACTIONS TO THE SOFTWARE

<table>
<thead>
<tr>
<th></th>
<th>difficult</th>
<th>easy</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>rigid</th>
<th>flexible</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>unfriendly</th>
<th>friendly</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not appropriate</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>appropriate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.7</td>
</tr>
</tbody>
</table>

**TERMINOLOGY**

[5] Terms on the screen are

<table>
<thead>
<tr>
<th>ambiguous</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>precise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.9</td>
</tr>
</tbody>
</table>

[6] Abbreviations

<table>
<thead>
<tr>
<th>confusing</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.2</td>
</tr>
</tbody>
</table>

**LEARNING**

Learning to operate the system is

[7] difficult | easy

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.1</td>
</tr>
</tbody>
</table>

[8] Exploring new features easier by

<table>
<thead>
<tr>
<th>trial and error</th>
<th>reading</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.6</td>
</tr>
</tbody>
</table>

**SYSTEM PERFORMANCE**

[9] Speed

<table>
<thead>
<tr>
<th>too slow</th>
<th>fast enough</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.4</td>
</tr>
</tbody>
</table>
CODE SEARCH


<table>
<thead>
<tr>
<th>difficult</th>
<th>easy</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td></td>
<td>4.0</td>
</tr>
</tbody>
</table>

[11]

<table>
<thead>
<tr>
<th>not appropriate</th>
<th>appropriate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td></td>
</tr>
<tr>
<td>Mean Value</td>
<td>4.1</td>
</tr>
</tbody>
</table>

[12] CBR-based Retrieval

<table>
<thead>
<tr>
<th>difficult</th>
<th>easy</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td></td>
<td>4.1</td>
</tr>
</tbody>
</table>

[13]

<table>
<thead>
<tr>
<th>not appropriate</th>
<th>appropriate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td></td>
</tr>
<tr>
<td>Mean Value</td>
<td>4.5</td>
</tr>
</tbody>
</table>

[14] Past Design and Implementation decisions are:

<table>
<thead>
<tr>
<th>confusing</th>
<th>helpful</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td></td>
<td>4.2</td>
</tr>
</tbody>
</table>

FINDING SIMILAR CODE

[15] Finding similar code is:

<table>
<thead>
<tr>
<th>difficult</th>
<th>easy</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
<td></td>
<td>4.2</td>
</tr>
</tbody>
</table>
[16] System’s assistance for this purpose:

<table>
<thead>
<tr>
<th>little</th>
<th>much</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**GENERIC COMPONENT CREATION**

[17] Generic component creation is:

<table>
<thead>
<tr>
<th>difficult</th>
<th>easy</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

[18] System’s assistance for such a creation:

<table>
<thead>
<tr>
<th>little</th>
<th>much</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

[19] Storing generic components in the repository is:

<table>
<thead>
<tr>
<th>not useful</th>
<th>useful</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**NEW CODE INSERTION**

[20] Appending new code in the repository is:

<table>
<thead>
<tr>
<th>difficult</th>
<th>easy</th>
<th>Mean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


