Outils d'analyse de programmes pour la rétro-conception de bases de données

Henrard, Jean; Roland, Didier; Englebert, Vincent; Hick, Jean-Marc; Hainaut, Jean-Luc

Published in:
proceedings INFORSID'98

Publication date:
1998

Outils d'analyse de programmes pour la rétro-conception de bases de données

J. Henrard, D. Roland, V. Englebert, J-M. Hick, J-L. Hainaut

Institut d'Informatique, FUNDP
rue Grandgagnage, 21 - B-5000 Namur, Belgique

Email : db-main@info.fundp.ac.be Tél: (32)81-72-49-85 Fax : (32)81-72-49-67

Résumé

Cet article montre que la rétro-ingénierie des données (REBD) réclame des raisonnements et des techniques qui ont été développés en génie logiciel, notamment dans le domaine de la compréhension de programmes. Inversement, la compréhension du fonctionnement d'un programme existant nécessite une idée claire de la sémantique des données persistantes sur lesquelles ce programme travaille. Un des processus de la méthodologie de REBD présentée consiste à rechercher les contraintes implicites cachées dans le code procédural. Cette recherche se base, entre autres, sur la recherche de patterns (clichés), sur les graphes de dépendance, sur la fragmentation de programmes et sur la visualisation de programmes. L'article montre comment ces techniques peuvent contribuer à la rétro-ingénierie de bases de données et comment l'atelier logiciel DB-MAIN les met en oeuvre.

Mots-clés

rétro-ingénierie, fragment de programme, base de données, compréhension de programmes, graphe de dépendance, atelier logiciel

Abstract

The paper shows that database reverse engineering (DBRE) needs several reasoning, techniques and tools that have been developed in the software engineering realm, and particularly in program understanding. Conversely, program understanding requires an in-depth comprehension of the semantics of the data. One of the process of the DBRE methodology we propose is dedicated to eliciting implicit structures and constraints buried in the application programs. This elicitation is based on such techniques as pattern searching, dependency graph analysis, program slicing and program visualisation. The paper demonstrates how these techniques can contribute to DBRE, and how that have been integrated into the DB-MAIN CASE tool.

Key words

reverse engineering, program slicing, database, program understanding, dependency graph, CASE tool

1. Introduction

La rétro-ingénierie d'un composant logiciel est un processus d'analyse de la version opérationnelle de ce composant qui vise à en reconstruire les spécifications techniques et fonctionnelles. La rétro-ingénierie a comme but la redocumentation, la conversion, la maintenance ou l'évolution d'applications anciennes.

La rétro-ingénierie est un processus d'autant plus complexe que l'application est mal structurée, ancienne, non ou mal documentée. Pour les applications orientées données, c'est-à-dire les applications dont le composant central est une base de données, on admet que la complexité puisse être réduite en effectuant la rétro-ingénierie des données indépendamment (ou presque) de la partie procédurale, puis en procédant à la rétro conception de la partie procédurale sur base des connaissances acquises. Cette approche peut se justifier de la façon suivante :

- les données persistantes constituent le composant central de nombreuses applications de gestion;
- la connaissance de la structure des données persistantes facilite la compréhension de l'application complète;
- la distance sémantique entre la spécification conceptuelle et l'implémentation physique est souvent plus faible pour les données que pour le code procédural;
- la structure des données persistantes est généralement la partie la plus stable d'une application;
- les processus méthodologiques d'ingénierie des bases de données sont plus formalisés que ceux du logiciel en général.

La rétro-ingénierie des structures de données, bien qu'étant mieux maîtrisée que celle d'applications complètes, demeure une tâche complexe. La majorité des propositions de méthodes imposent des hypothèses trop restrictives pour traiter complètement des applications complexes. Elles supposent souvent que :

- la base de données a été obtenue via des règles de transformation conceptuel/logique simples, et par conséquent la traduction du schéma physique en schéma conceptuel est presque immédiate;
- le schéma n'a pas subi de restructuration d'optimisation;
- toutes les contraintes ont été traduites dans le langage de description de données;
- les noms sont significatifs.

En outre, ces méthodes sont spécifiques à un type de SGBD. Lorsqu'elles prennent en compte le code procédural, c'est le plus souvent pour en considérer les requêtes SQL ou les accès aux fichiers uniquement.

Une part importante des applications réelles violent ces hypothèses, ce qui rend leur rétro-ingénierie plus complexe. Elle ne peut être menée à bien sans l'aide de raisonnements plus élaborés, et d'outils puissants d'analyse de schémas et surtout de programmes.

Depuis peu, certains auteurs reconnaissent que le code procédural d'une application est une source d'information importante lors de la rétro-ingénierie de bases de données ([JORI92], [HAIN93], [PETI93] et [ANDE94]). Les contraintes et structures de données qui ne sont pas déclarées explicitement lors de la déclaration de la base de données s'y trouvent codées d'une manière ou d'une autre.

La communauté du génie logiciel définit la *compréhension de programmes* comme l'acquisition de connaissances à propos d'un programme existant. Cette connaissance peut servir à la correction d'un programme, la réutilisation, la documentation et à la rétro-ingénierie. Bien que beaucoup d'efforts portent sur l'automatisation de la compréhension de programmes, une part significative de ce travail doit encore se faire manuellement. Les outils d'aide vont de la simple inspection visuelle du texte à l'analyse dynamique de l'exécution du programme en passant par l'analyse statique, de plus en plus complexe, du programme [RUGA95].

Cet article explique pourquoi et comment nous avons choisi, adapté et appliqué certaines de ces techniques et outils de compréhension de programmes à la rétro-ingénierie de bases de données. L'article est organisé comme suit. La section 2 est une synthèse de la méthode générique de rétro-ingénierie de bases de données DB-MAIN. La section 3 présente en quoi les techniques de compréhension de programmes sont applicables à la rétro-ingénierie de bases de données. La section 4 décrit les principaux concepts et fonctionnalités de l'atelier logiciel DB-MAIN. La fin de l'article présente l'*utilisation* de différentes techniques de compréhension de programmes dans le cadre de la rétro-ingénierie de bases de données et leur implémentation dans l'atelier logiciel DB-MAIN, à savoir la recherche dans les textes sources (section 5), le graphe de dépendance des variables (section 6), la fragmentation de programme (section 7) et la visualisation de programmes (section 8).

2. Une méthode générique de rétro-ingénierie de bases de données

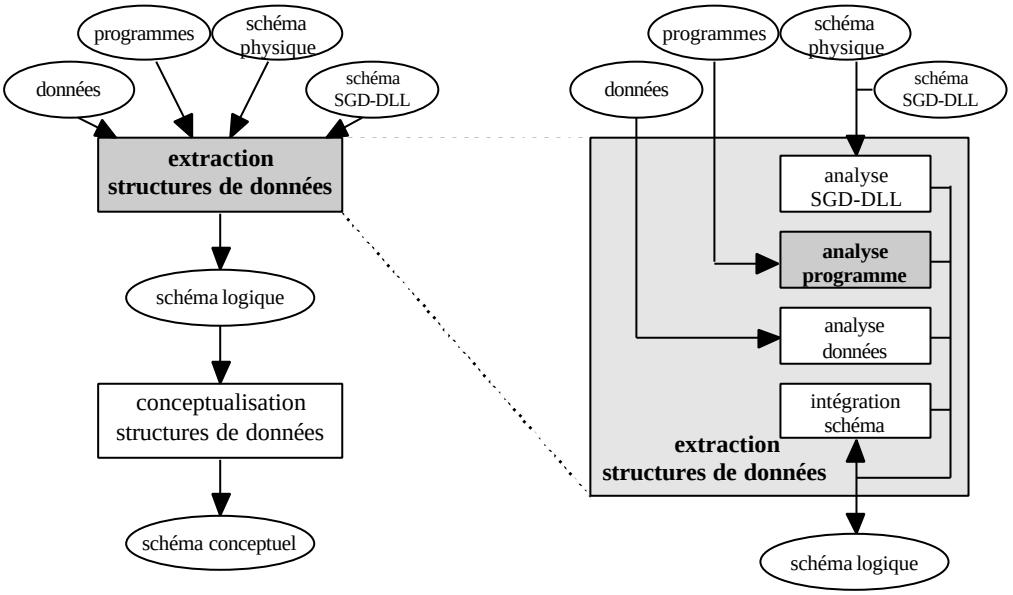


Figure 1 : *Méthode générique de rétro-ingénierie de bases de données.*

Cette méthode comporte deux processus principaux (figure 1), à savoir l'extraction des structures de données et la conceptualisation de celles-ci [HAIN93]. Ces deux parties correspondent à des schémas différents qui requièrent des raisonnements, des concepts et des outils différents. De plus, cette division est à peu de choses près l'inverse des processus de conception physique et logique habituellement admis en ingénierie des bases de données [BATI92], [NANC96].

2.1. L'extraction des structures de données

Cette phase a pour objet la reconstruction du schéma logique complet selon le modèle du SGD[1], y compris toutes les contraintes et les structures de données non explicitement déclarées.

Certains SGD (les SGBD par exemple) offrent, sous une forme ou une autre, une description du schéma global des données. Bien que ce schéma soit déjà assez complet, il devra être enrichi suite à une analyse plus approfondie des autres composants de l'application (vues, code procédural, données, écrans de saisie, ...).

Le problème est beaucoup plus complexe quand il s'agit de recouvrer le schéma logique de fichiers classiques. L'analyse de chaque programme source ne livrera qu'une partie seulement de la structure des données. Cette analyse doit aller bien au delà de la simple recherche de la structure des fichiers déclarés dans les programmes. En outre, ces vues partielles devront être fusionnées par des techniques d'intégration spécifiques.

Le problème principal est celui des structures et contraintes dites *implicites*, c'est-à-dire celles qui n'ont pas été traduites explicitement sous la forme de code déclaratif (communément dit DDL - *Data Description Language*), mais qui au contraire, ont été exprimées sous une forme procédurale par exemple, ou tout simplement ignorées.

Eliciter, c'est-à-dire rendre explicites, les structures définies dans les programmes ou perdues est une tâche complexe, pour laquelle il n'existe pas encore de méthodes déterministes. Seule une analyse méticuleuse de toutes les sources d'informations disponibles permet de retrouver les spécifications. Le plus souvent ces informations doivent être consolidées par la connaissance du domaine de l'application, ce qui rend l'automatisation complète pratiquement impossible.

Les processus principaux de l'extraction des structures de données sont les suivants :

- analyse des déclarations des structures de données dans les scripts de définition du schéma et dans les sources du programme;
- analyse du code source du programme pour y retrouver les traces des structures et contraintes implicites;
- analyse des données sur lesquelles travaille le programme pour en trouver les structures et les propriétés implicites ainsi que pour confirmer ou infirmer certaines hypothèses;
- intégration des différents schémas obtenus lors des étapes précédentes.

Dans cet article, nous illustrerons l'application de techniques d'analyse de programmes aidant à la détection de structures algorithmiques qui suggèrent des structures de données et des contraintes implicites.

2.2. La conceptualisation des structures de données

Le deuxième processus consiste en l'interprétation du schéma obtenu lors de l'extraction des structures de données pour en dériver un schéma conceptuel. Il détecte et transforme (ou élimine) les redondances et les structures non conceptuelles introduites lors de la conception de la base de données.

---

[1]Système de Gestion de Données, terme générique recouvrant ceux de SGBD et Systèmes de Gestion de Fichiers.

Finalement, nous disposons d'un schéma conceptuel qui peut encore être transformé, de manière à le rendre conforme à un standard méthodologique par exemple. Il s'agit d'un processus classique de *normalisation conceptuelle*.

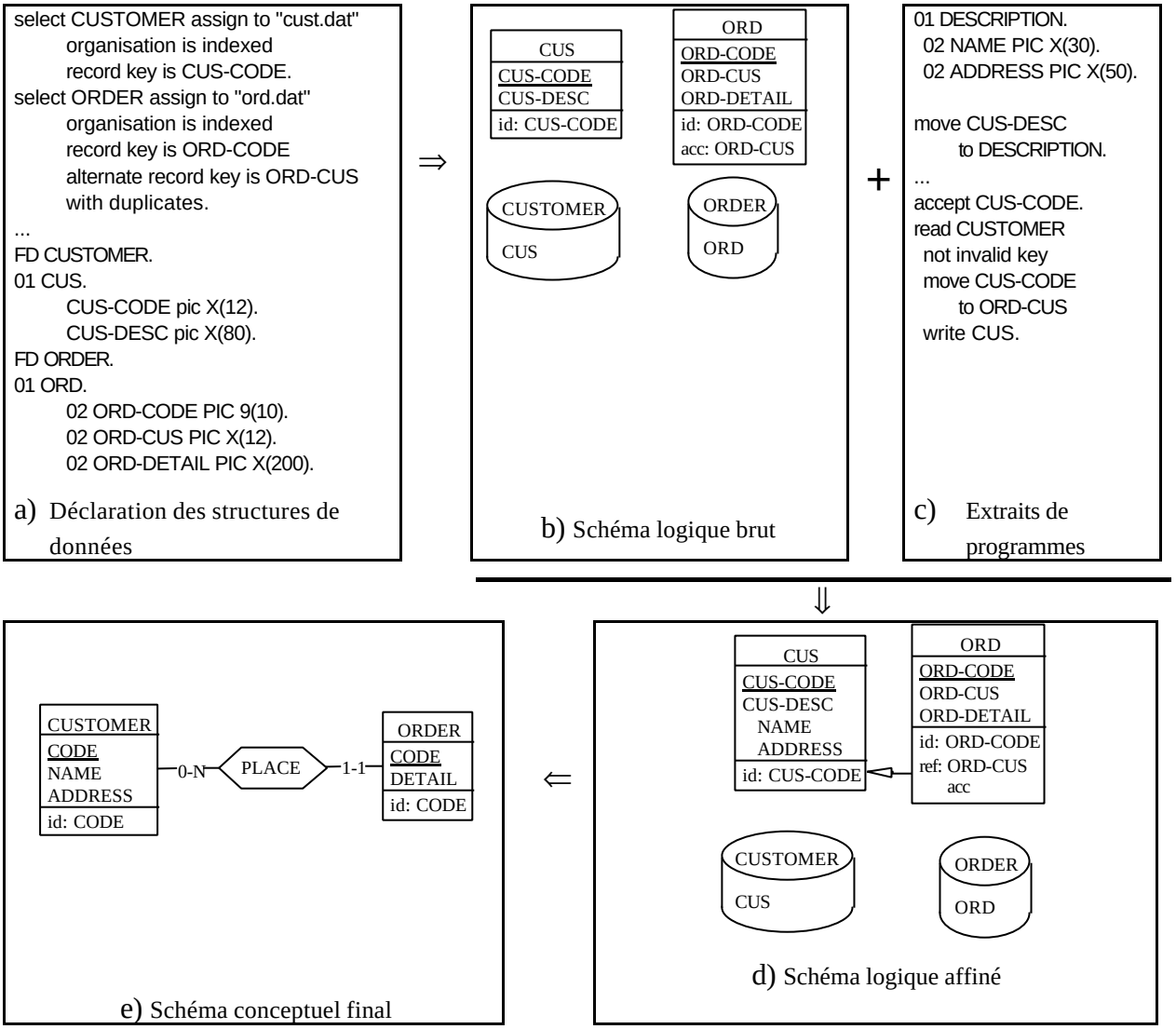


Figure 2 : Un exemple de rétro-ingénierie de base de données.

2.3. Un exemple de rétro-ingénierie

La figure 2 présente un exemple très réduit mais représentatif de l'application des techniques d'analyse de programmes en rétro-ingénierie des données. La première étape est l'analyse de la déclaration des structures de données (2.a) pour en extraire un premier schéma logique (2.b), que nous appelons *schéma logique brut*. Ce schéma ne contient que les structures de données et les contraintes qui ont été déclarées explicitement. Il faut maintenant enrichir ce schéma de manière à y inclure toutes les contraintes implicites. Pour ce faire, nous allons analyser la partie procédurale du programme (2.c). On constate que `CUS-DESC` peut être affiné comme `DESCRIPTION`, grâce à l'instruction `move` et que le code de validation avant enregistrement suggère que `ORD-CUS` est une clé étrangère vers `CUS-CODE`. Lorsque nous avons intégré ces contraintes dans le schéma logique brut, nous obtenons le schéma logique affiné (2.d). Ce schéma logique complet peut alors être conceptualisé pour produire le schéma conceptuel final (2.e).

3. La compréhension de programme en rétro-ingénierie de bases de données

Comme expliqué dans la description de la méthodologie, une des sources d'information lors de l'extraction des structures de données est le code source du programme à analyser.

La partie déclarative de ce code (DDL, déclaration des fichiers, ...) peut être très facilement analysée de façon automatique afin d'en extraire un schéma des structures de données brutes. Cependant, une part plus ou moins importante du schéma des données est déclarée implicitement dans la partie procédurale du code (entre autres). Une contrainte peut ne pas être déclarée dans la partie déclarative des données soit parce que le SGBD utilisé ne le permet pas, soit parce que, lors de l'implémentation, l'analyste a décidé de l'implémenter dans le code procédural pour des raisons d'efficacité, de facilité ou de portabilité. Il est donc important de pouvoir extraire du code procédural toutes les contraintes qui y sont déclarées implicitement. Cette tâche est fastidieuse, difficile et non déterministe, dépendant non seulement de l'expertise de l'analyste, mais aussi de la puissance des outils disponibles. Les parties du code qui implémentent ces contraintes sont généralement disséminées dans tout le programme. De plus, chaque programmeur a son (ses) propre(s) style(s) de programmation et implémente de façon souvent astucieuse, mais cohérente, les mêmes contraintes.

Pour toutes ces raisons, même si notre but n'est que l'extraction des structures de données persistantes, il nous faut également analyser la partie procédurale de l'application. Il nous faudra retrouver uniquement les règles de traitement essentielles pour comprendre la partie du code qui implémente les contraintes sur les données persistantes de l'application. Les structures et contraintes que l'on désire retrouver sont entre autres les suivantes :

- *Affinage* des enregistrements et des champs : un enregistrement ou un champ bien que déclaré atomique peut-être décomposé en une structure plus fine. Par exemple, l'utilisation d'un champ `ADRESSE` d'une longueur de 150 caractères montre qu'il est décomposé en trois champs élémentaires (`RUE`, `CODE-POSTAL` et `VILLE`).
- *Agrégation* de champs : un ensemble de champs indépendants peuvent être les composants d'un champ décomposable.
- *Clés étrangères* : un (ou plusieurs) champ peut référencer un autre enregistrement.
- *Identifiants* : un (ou plusieurs) champ peut être un identifiant d'un enregistrement ou d'un champ décomposable multivalué. Par exemple, si un champ `REVENU` multivalué contient les champs `ANNEE` et `MONTANT` et que l'on constate que les valeurs d'`ANNEE` sont uniques dans chaque enregistrement; alors, `ANNEE` est un identifiant local à `REVENU`.
- *Cardinalités exactes* d'un champ multivalué : la cardinalité d'un champ multivalué est un couple I-J qui représente le nombre minimum (I) et maximum (J) d'éléments actifs de ce champ. Un champ multivalué est le plus souvent représenté par un tableau, qui est déclaré de taille fixe. Cependant, tous les éléments du tableau ne sont pas toujours utilisés; le but ici est de retrouver le nombre minimum et maximum d'éléments du tableau réellement utilisés.

4. L'atelier logiciel DB-MAIN

La rétro-ingénierie est un processus long et complexe. Bien que la rétro-ingénierie ne soit pas totalement automatisable, l'analyste a besoin d'outils pour mener à bien ce processus. Ces outils permettent d'automatiser ce qui l'est et d'aider l'analyste à comprendre le programme et à retrouver l'information nécessaire là où il n'y a pas d'automatisation possible. Ces outils doivent être intégrés et doivent stocker leurs résultats dans un référentiel commun. Ils doivent également être facilement

extensibles et personnalisables pour répondre exactement aux besoins de l'analyste, qui varient d'un projet à l'autre et bien sûr d'un analyste à l'autre [HAIN96b].

L'environnement d'ingénierie de bases de données DB-MAIN[2] est le résultat d'un projet de R&D initié en 1993 par l'équipe de recherche en ingénierie des bases de données de l'institut d'informatique de l'université de Namur (Belgique). Cet atelier est dédié aux différents processus d'ingénierie, ce qui englobe la rétro-ingénierie. En particulier, son but est d'assister le développeur dans la conception, la rétro-ingénierie, la migration, la maintenance et l'évolution des bases de données et de leurs applications.

Comme tout atelier de génie logiciel, DB-MAIN inclut les fonctions standards d'analyse et de développement, c'est-à-dire la création, la modification, l'affichage, la gestion, la validation et la transformation des spécifications de la base de données ainsi que la génération de code, de rapports et des fonctions d'import/export. Plus de détails peuvent être trouvés dans [ENGL95] et [HAIN93].

DB-MAIN dispose d'un référentiel qui contient les définitions de tous les composants d'un projet, ce référentiel peut être étendu dynamiquement par l'utilisateur. Une interface graphique permet de manipuler les spécifications et d'exécuter les commandes. Tous les produits (schémas de bases de données, textes sources, documents, scripts, etc.) sont représentés dans le projet. Les textes sources sont affichés dans une fenêtre d'édition, c'est dans cette fenêtre que l'on utilise les outils de compréhension de programme. De plus, chaque ligne de texte peut être marquée pour un traitement ultérieur et il est possible d'y associer une description textuelle.

DB-MAIN dispose d'assistants, qui sont des outils de haut niveau dédiés à la résolution de problèmes d'ingénierie spécifiques ou répétitifs.

En plus des fonctionnalités classiques d'ingénierie de bases de données, l'atelier dispose de fonctionnalités spécifiques à la rétro-ingénierie.

Des *extracteurs* permettent d'extraire de façon automatique les structures de données d'un texte source. Ces extracteurs lisent la partie déclarative d'un texte source et créent sa représentation abstraite dans le référentiel.

Un assistant détecte les clés étrangères possibles dans un schéma. A partir d'un groupe de champs origine (ou cible) d'une clé étrangère, cet assistant cherche dans le schéma tous les groupes de champs qui peuvent être cible (ou origine) de ce groupe. La recherche est basée sur une combinaison de règles heuristiques qui tiennent compte de la fonction du groupe (identifiant), de la longueur et du type des champs et de règles de comparaison des noms des champs.

Plusieurs outils de compréhension de programmes sont aussi disponibles : des outils de recherche de *patterns* (patrons), de calcul du graphe de dépendance des variables, de fragmentation de programmes.

Pour rendre DB-MAIN extensible et le plus flexible possible, nous lui avons adjoint un langage de programmation, *Voyager* [ENGL98]. Il s'agit d'un langage procédural complet qui dispose de primitives d'accès et de manipulation (prédicatives et navigationnelles) du référentiel et de fonctions d'analyse de textes. Il permet à l'utilisateur de développer ces propres fonctions qui sont directement accessibles dans l'atelier au même titre que les outils de base.

Dans la suite de cet article, nous présenterons plus en détail les techniques et les outils nécessaires à la compréhension de programmes : la recherche de patterns textuels, le graphe de dépendance, la fragmentation de programmes et un outil de représentation des textes sources.

---

[2] Pour *Database Application Maintenance*

5. La recherche dans les textes sources

Une des techniques les plus simples de compréhension de programmes est la recherche de patterns (ou *clichés* de programmation) dans un texte source. DB-MAIN dispose d'un moteur de recherche plus puissant qu'une simple recherche de chaînes de caractères dans un texte source comme le font la plupart des éditeurs de textes. Les patterns à rechercher sont définis dans un langage de définition de patterns (PDL - Pattern Description Language) fort proche de la notation BNF.

La définition d'un pattern peut faire appel à la définition d'un pattern déjà défini. Par exemple, la définition du nom des variables COBOL ou d'une expression de comparaison est réutilisable pour définir des patterns plus complexes.

Le langage PDL dispose de variables (préfixées par @). Lorsqu'un pattern est instancié, les variables de ce patterns reçoivent une valeur qui peut être utilisée, par exemple, comme paramètre lors de l'exécution d'une procédure *Voyager* ou pour construire le graphe de dépendance (voir ci-dessous). Ces variables peuvent également recevoir une valeur, ce qui permet d'instancier partiellement un pattern avant d'effectuer la recherche.

La recherche de patterns sert principalement à rechercher une instruction ou à faire l'inventaire des instructions d'un certain type dans un programme, telles que la recherche de toutes les instructions d'écriture dans un fichier, de toutes les instructions d'assignation ou de toutes les jointures SQL.

```
var ::= /g"[a-zA-Z0-9]*[a-zA-Z][-a-zA-Z0-9]*";
var_1 ::= var;
var_2 ::= var;
move ::= "MOVE" - @var_1 - "TO" - @var_2 ;
```

Figure 3 : Définition du pattern `move`.

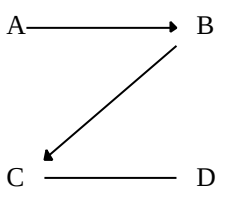
La figure 3 montre la déclaration du pattern `move`, qui va permettre de retrouver les instructions d'assignation élémentaires dans un programme COBOL. Le nom '-' est celui d'un pattern définissant tous les séparateurs COBOL, défini dans une bibliothèque secondaire de patterns. Le caractère '@' indique que le nom du pattern qui suit est associé à une variable. Les caractères entre guillemets représentent des constantes qui sont recherchées telles quelles. Par exemple, le pattern `move`, de la figure 3, peut être instancié par l'instruction `MOVE cus-addr TO address` (avec `var_1` = "cus-addr" et `var_2` = "address").

6. Le graphe de dépendance

Le graphe de dépendance (qui est une généralisation du diagramme de flux) est un graphe où chaque variable du programme est représentée par un noeud et dont les arcs (orientés ou non) représentent une relation (assignation, comparaison,...) entre deux variables. Ces relations sont choisies par l'analyste sous la forme d'une liste de patterns à deux variables (tels que `move`)

```
...
MOVE A TO B
 ...
MOVE B TO C
 ...
IF (C = D )
 ...
```

a) *Un extrait d'un programme COBOL*



b) *Le graphe de dépendance du texte 4.a*

Figure 4 : *Le graphe de dépendance.*

La figure 4.b est le graphe de dépendance de l'extrait du programme de la figure 4.a. S'il y a un chemin entre la variable A et la variable C dans le graphe de dépendance, alors il existe dans le programme une séquence d'instructions telle que la valeur de A est en relation avec la valeur de C. Quand deux variables sont en relation, cela signifie que les deux variables ont la même structure ou au moins la structure de l'une des deux est incluse dans l'autre.

Le graphe de dépendance est un moyen très efficace pour retrouver les champs qui peuvent être affinés ou agrégés. S'il existe un chemin passant par un champ et concernant une autre variable qui a une décomposition plus fine, on peut affiner le champ avec la structure de cette variable.

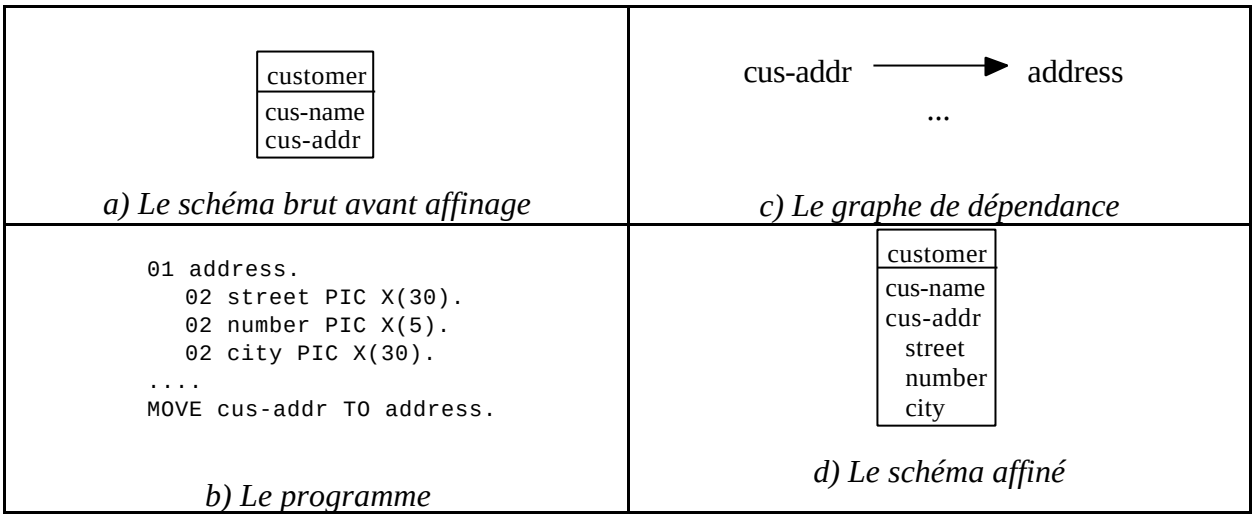


Figure 5 : Utilisation du graphe de dépendance.

La figure 5 nous montre un exemple d'affinage d'un schéma brut (5.a) grâce à l'utilisation du graphe de dépendance. Si nous calculons le graphe de dépendance (5.c) du programme (5.b), nous constatons qu'il y a un chemin (un seul arc) qui va de `cus-addr` vers `address`. Nous pouvons en conclure que les structures de `cus-addr` et de `address` sont équivalentes. Nous pouvons donc raisonnablement choisir d'affiner `cus-addr` avec la structure de `address`. Nous obtenons le schéma affiné (5.d).

Le graphe de dépendance peut aussi donner des indices de clés étrangères. S'il y a un chemin dans le graphe de dépendance entre deux champs, dont l'un est un identifiant d'un type d'enregistrements, il est probable que l'autre soit une clé étrangère.

Dans DB-MAIN, l'utilisation du graphe de dépendance se fait en deux parties : le calcul du graphe et son utilisation.

Pour calculer le graphe de dépendance d'un texte source, l'analyste donne une liste de patterns. Chacun de ces patterns doit contenir deux variables (`var_1` et `var_2`). Il définit qu'il y a une relation entre `var_1` et `var_2`, orientée de `var_1` vers `var_2` si l'analyste décide que le pattern est orienté. Les noeuds du graphe qui sera construit seront les instanciations des variables `var_1` et `var_2`.

L'utilisation du graphe de dépendance est simple, il suffit de cliquer avec le bouton des droite de la souris sur une des variables dans l'éditeur de texte source et toutes les occurrences des variables qui sont liées (directement ou indirectement) à cette variable sont coloriées dans le texte source. Nous avons décidé de ne pas représenter explicitement le graphe de dépendance sous une forme graphique,

mais de présenter les variables dans leur contexte, dans le texte source lui-même, car cela permet à l'analyste de visualiser le contexte dans lequel ces variables sont utilisées.

7. La fragmentation de programme

Dans cette section, nous décrirons une technique, appelée *fragmentation de programme* (*program slicing*). Cette technique permet d'extraire d'un programme le fragment nécessaire et suffisant (idéalement) pour comprendre et expliquer le comportement de ce programme à un point déterminé. Ce point est appelé *critère de fragmentation*, il est constitué d'un couple [instruction, variable] tel que la variable est référencée dans l'instruction. Idéalement, un *fragment de programme* est constitué de toutes les instructions qui affectent les valeurs calculées par rapport au critère de fragmentation et uniquement celles-là.

Le concept original de fragments de programmes a été introduit par Mark Weiser [WEIS84]. Celui-ci prétend qu'un fragment correspond à l'abstraction mentale que fait un programmeur quand il débogue un programme. Différentes notions de fragmentation de programmes et méthodes de calcul ont été définies depuis lors, répondant chacune à des exigences particulières.

```
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. CLIENT-COMMANDE.
3  ENVIRONMENT DIVISION.
4  INPUT-OUTPUT SECTION.
5  FILE-CONTROL.
...
17  DATA DIVISION.
18  FILE SECTION.
19  FD CLIENT.
20  01 CLI.
21    02 CLI-CODE PIC X(12).
22    02 CLI-NOM PIC X(80).
23    02 CLI-HISTORIQUE PIC X(1000).
24  FD COMMANDE.
25  01 COM.
26    02 COM-CODE PIC 9(10).
27    02 COM-CLIENT PIC X(12).
28    02 COM-DETAIL PIC X(200).
29
30  WORKING-STORAGE SECTION.
...
39  PROCEDURE DIVISION.
...
64  NOUV-COM.
65   DISPLAY "NUM COMMANDE:" WITH NO
ADVANCING.
66    ACCEPT COM-CODE.
67
68    MOVE 1 TO FIN-FICHIER.
69    PERFORM LECT-CODE-CLI UNTIL FIN-
FICHIER=0.
70
71    DISPLAY CLI-NOM.
72
73    MOVE CLI-CODE TO COM-CLIENT.
74    SET IND-DET TO 1.
75    MOVE 1 TO FIN-FICHIER.
76    PERFORM LECT-DETAIL UNTIL FIN-
FICHIER=0
        OR IND-DET = 21.
77    MOVE LISTE-DETAIL TO COM-DETAIL.
78
79    WRITE COM
80     INVALID KEY DISPLAY "ERREUR".
81
82  LECT-CODE-CLI.
83    DISPLAY "NUM DU CLIENT:"
        WITH NO ADVANCING.
84    ACCEPT CLI-CODE.
85    MOVE 0 TO FIN-FICHIER.
86    READ CLIENT INVALID KEY
87      DISPLAY "CLIENT INEXISTANT"
88      MOVE 1 TO FIN-FICHIER
89    END-READ.
90
91  LECT-DETAIL.
92  * lecture des produits commandés
```

Figure 6 : Extrait d'un programme COBOL.

Les caractéristiques des différents langages de programmation comme les procédures, les contrôles de flux arbitraires (goto), les types de données composites et les pointeurs nécessitent des solutions spécifiques. Dans le cadre de la rétro-ingénierie de bases de données, nous sommes intéressés par les programmes COBOL comportant des procédures, des types de données composées et des contrôles de flux arbitraires.

```
17  DATA DIVISION.
18  FILE SECTION.
64  NOUV-COM.
68    MOVE 1 TO FIN-FICHIER.
```

```
19  FD CLIENT.                              69    PERFORM LECT-CODE-CLI UNTIL FIN-
20  01 CLI.                                 FICHIER=0.
21    02 CLI-CODE PIC X(12).                71    DISPLAY CLI-NOM.
22    02 CLI-NOM PIC X(80).                 82  LECT-CODE-CLI.
30  WORKING-STORAGE SECTION.                84    ACCEPT CLI-CODE.
31  01 CHOIX PIC X.                         85    MOVE 0 TO FIN-FICHIER.
32  01 FIN-FICHIER PIC 9.                   86    READ CLIENT INVALID KEY
39  PROCEDURE DIVISION.                     88      MOVE 1 TO FIN-FICHIER
40  PRINCIPAL.                              89    END-READ.
...
```

*Figure 7 : Le fragment par rapport à la ligne 71 et à la variable **CLI-NOM** du programme de la figure 6.*

La figure 6 est un extrait d'un programme COBOL qui permet à l'utilisateur de spécifier une commande valide, comportant un numéro de client existant. La figure 7 est le fragment calculé par rapport à la ligne 71 et à la variable `CLI-NOM` du programme de la figure 6, c'est-à-dire l'instruction d'affichage du nom du client à qui va être attribuée la commande. On constate que les lignes concernant la gestion de la commande n'apparaissent pas.

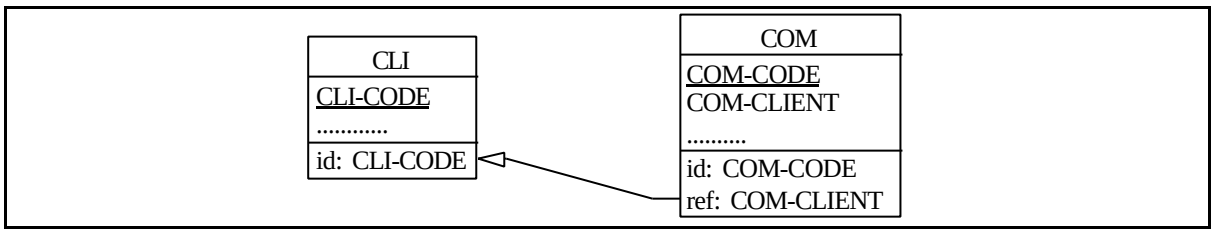


Figure 8 : Un contrainte référentielle à vérifier.

```
17  DATA DIVISION.                          39  PROCEDURE DIVISION.
18  FILE SECTION.                           ...
19  FD CLIENT.                              64  NOUV-COM.
20  01 CLI.                                 68    MOVE 1 TO FIN-FICHIER.
21    02 CLI-CODE PIC X(12).                69    PERFORM LECT-CODE-CLI UNTIL FIN-
24  FD COMMANDE.                            FICHIER=0.
25  01 COM.                                 73    MOVE CLI-CODE TO COM-CLIENT.
27    02 COM-CLIENT PIC X(12).              79    WRITE COM
30  WORKING-STORAGE SECTION.                82  LECT-CODE-CLI.
31  01 CHOIX PIC X.                         84    ACCEPT CLI-CODE.
32  01 FIN-FICHIER PIC 9.                   85    MOVE 0 TO FIN-FICHIER.
                                            86    READ CLIENT INVALID KEY
                                            88      MOVE 1 TO FIN-FICHIER
                                            89    END-READ.
```

*Figure 9 : Fragment par rapport à la ligne 79 et à **COM-CLI** du programme de la figure 6.*

La fragmentation de programme intervient lorsque l'on veut comprendre comment a été construite la valeur d'une variable à un point donné du programme. En rétro-ingénierie de base de données, on calculera souvent un fragment dont le critère de fragmentation est constitué d'une instruction d'écriture d'un enregistrement et d'une partie de cet enregistrement.

Par exemple, pour confirmer l'existence de la clé étrangère de la figure 8, on calculera le fragment dont le critère de fragmentation est constitué de l'instruction d'écriture de l'enregistrement `COM` et du champ origine de la clé étrangère (`COM-CLI`), pour vérifier que sa valeur est une valeur existante de l'identifiant de la cible (`CLI-CODE`). Ce fragment doit contenir une instruction de lecture de l'enregistrement cible (`CLI`). Si on applique le principe au programme de la figure 6, on calculera le fragment par rapport à la ligne 79 (instruction d'écriture de `COM`) et à la variable `COM-CLI` (la clé

étrangère). On obtient le fragment de la figure 9. Le fragment ainsi calculé contient une instruction de lecture de l'enregistrement `CLIENT` (ligne 86) et on conclut que la variable `COM-CLI` contient bien une valeur existante de l'identifiant de `CLIENT` (`CLI-CODE`). On peut appliquer le même raisonnement à la recherche de l'identifiant local et de la cardinalité exacte d'un tableau.

Pour réaliser un outil de fragmentation de programmes dans DB-MAIN, nous nous sommes inspirés de la technique proposée par Susan Horwirtz [HORW90] pour le calcul de fragments inter-procéduraux. Le calcul d'un fragment est défini en terme de parcours du *graphe de dépendance du système* (system dependence graph - *SDG*). Le SDG est un graphe orienté dont les nœuds correspondent aux instructions et les arcs représentent les dépendances de données, les dépendances de contrôle et les appels de procédures. Un critère de fragmentation est identifié à un nœud du graphe. Un fragment est l'ensemble des nœuds du SDG à partir desquels on peut atteindre le nœud représentant le critère de fragmentation.

Dans DB-MAIN, l'outil de fragmentation de programme permet de définir un critère de fragmentation constitué d'une instruction (noeud du SDG) et d'une ou plusieurs variables référencées dans cette instruction. Les instructions qui font partie du fragment sont coloriées dans l'éditeur de texte source. Ici encore, nous avons préféré colorier les instructions du fragment dans leur contexte, plutôt que de proposer une vue du texte source où seules apparaissent les lignes appartenant au fragment. Il existe en effet des lignes qui ne font pas partie du fragment mais qui peuvent apporter des informations utiles à la compréhension du fragment. Les commentaires ou les instructions d'affichage de messages d'erreur ne font pas partie du fragment, car ils n'influencent pas la valeur des variables du critère de fragmentation, mais ils peuvent s'avérer particulièrement utiles à la compréhension du fragment. Il est d'ailleurs possible d'isoler les lignes du fragment et de les copier dans un autre texte.

```
64  NOUV-COM.                                  77    MOVE LISTE-DETAIL TO COM-DETAIL.
65    DISPLAY "NUM COMMANDE:" WITH NO          78
ADVANCING.                                     79    WRITE COM
66    ACCEPT COM-CODE.                         80     INVALID KEY DISPLAY "ERREUR".
67                                             82  LECT-CODE-CLI.
68    MOVE 1 TO FIN-FICHIER.                   83    DISPLAY "NUM DU CLIENT:"
69    PERFORM LECT-CODE-CLI UNTIL FIN-                 WITH NO ADVANCING.
FICHIER=0.                                     84    ACCEPT CLI-CODE.
70                                             85    MOVE 0 TO FIN-FICHIER.
71    DISPLAY CLI-NOM.                         86    READ CLIENT INVALID KEY
72                                             87      DISPLAY "CLIENT INEXISTANT"
73    MOVE CLI-CODE TO COM-CLIENT.             88      MOVE 1 TO FIN-FICHIER
74    SET IND-DET TO 1.                        89    END-READ.
75    MOVE 1 TO FIN-FICHIER.
76    PERFORM LECT-DETAIL UNTIL FIN-
FICHIER=0
        OR IND-DET = 21.
```

Figure 10 : Le fragment de la figure 7, les lignes du fragment sont en noir et les autres en gris.

La figure 10 représente le fragment de la figure 7 mais où les lignes appartenant au fragment sont affichées en gras, alors que le reste du programme ne l'est pas. Certaines instructions n'appartiennent pas au fragment, comme la ligne 87 alors qu'elles apportent de l'information utile ("le client recherché n'existe pas").

DB-MAIN dispose également d'un assistant qui permet de calculer l'union ou l'intersection de plusieurs fragments, de calculer les fragments pour toutes instructions d'un type et de changer la couleur d'un fragment.

8. La visualisation de programmes

Le code source, même avec la fragmentation de programme, est souvent difficile à comprendre et à maîtriser car un programme peut comporter plusieurs milliers de lignes de code d'un seul tenant. Il peut être mal structuré et les variables n'ont peut-être pas des noms significatifs. Enfin, une application peut contenir plusieurs centaines de programmes. Pour aider l'analyste à maîtriser la structure du programme, il est utile qu'il puisse disposer de visualisations différentes du programme, comme le graphe d'appel des procédures ou des modules, le graphe représentant les entrées/sorties des différents modules, les références croisées (pour connaître dans quelles procédures les variables sont utilisées), etc.

Tous ces modes de visualisations n'ont pas pour but de directement trouver des contraintes, comme le permet le graphe de dépendance, mais plutôt de donner une vue d'ensemble de l'application à l'analyste. Cela lui permettra de visualiser et comprendre l'ordonnancement des différents modules de l'application, de retrouver ceux qui font appel à la base de données et à quelle partie de celle-ci.

9. Conclusion

Pour mener à bien la rétro-ingénierie de bases de données complexes, il est nécessaire d'analyser les programmes qui utilisent ces données. Le domaine du génie logiciel nous offre des techniques de compréhension de programmes qui s'avèrent essentielles. D'autre part, le recouvrement des spécifications d'un programme exige que l'on comprenne la sémantique de la base de données utilisée. Bien que les communautés du génie logiciel et des bases de données soient souvent culturellement éloignées, il apparaît qu'elles utilisent toutes les deux des méthodes, techniques et outils développés par l'autre communauté.

Notre objectif est le développement d'une méthodologie et d'outils supportant la rétro-ingénierie de bases de données. Nous avons très vite compris qu'il nous était impossible de mener à bien cet objectif sans des techniques et des outils de compréhension de programmes. Nous avons donc intégré ces raisonnements dans notre méthodologie et développé des outils ad hoc dans l'atelier DB-MAIN.

Nos différentes études de cas nous ont appris que chaque projet de rétro-ingénierie était différent et demandait des techniques et outils différents. Donc tout outil d'aide à la rétro-ingénierie doit être programmable, personnalisable et flexible pour être utile. C'est dans ce but que DB-MAIN est programmable et son référentiel peut être étendu dynamiquement.

DB-MAIN est développé en C++ sous MS-Windows. Il en existe une version Education (complète mais ne permettant de manipuler que des schémas de taille limitée) accessible sans frais aux institutions non commerciales (contacter `db-main@info.fundp.ac.be`)

Référence

[ANDE94] Anderson, M. Extracting an Entity Relationship Schema from a Relational Database Through Reverse Engineering, in *Proc of the 13th Int. Conf. on ER Approach*, 1994, Manchester. Springer-Verlag.

[ANDE97] Anderson, M. Reverse Engineering of Legacy Systems: From Valued-Based to Object-Based Models, *PhD thesis*, Lausanne, EPFL, 1997.

[BATI92] Batini, C., Ceri, S. and Navathe, S.B. Conceptual Database Design - An Entity-Relationship Approach, Benjamin/Cummings, 1992.

[BLAH95] Blaha, M. and Premerlani, W. Observed Idiosyncracies of Relational Database Designs, in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, IEEE computer Society Press, 1995.

[ENGL95] Englebert, V., Henrard J., Hick, J.-M., Roland, D. and Hainaut, J.-L. DB-MAIN: un Atelier d'Ingénierie de Bases de Données, in *11 Journée Bases de Données Avancées*, Nancy, 1995.

[ENGL98] Englebert, V. *Voyager 2. Version 3 Release 0*, technical report Institut d'Informatique, Namur, Belgium, 1998.

[HAIN93] Hainaut, J.-L., Chandelon, M., Tonneau, C. and Joris M. Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press.

[HAIN96] Hainaut, J.-L., Henrard, J., Roland, D., Englebert, V. and Hick J.-M. Structure Elicitation in Database Reverse Engineering, *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

[HAIN96b] Hainaut, J.-L., Roland, D., Hick J.-M., Henrard, J. and Englebert, V. Database Reverse Engineering: from Requirements to CARE Tools, *Journal of Automated Software Engineering*, **3**(1), 1996.

[HAIN97] Hainaut, J.-L., Hick, J.-M., Henrard, J., Roland, D. and Englebert, V. Knowledge Transfer in Database Reverse Engineering - A supporting Case Study, in *Proc. of the 4th IEEE Working Conf. on Reverse Engineering*, Amsterdam, October 1997, IEEE Computer Society Press.

[HENR96] Henrard, J., Hick, J.-M., Roland, D., Englebert, V. and Hainaut, J.-L. Techniques d'Analyse de Programmes pour la Rétro-Ingénierie de Base de Données, in *Actes de la conférence INFORSID'96*, Bordeau.

[HORW90] Horwitz, S., Reps, T. and Binkley, D. Interprocedural Slicing Using Dependence Graphs, ACM Trans. on Programming Languages and Systems **12**(1), Jan. 1990, 26-60.

[JOINE94] Joiner, J.K., Tsai, W.T., Chen, X.P., Subramanian, S., Sun, J. and Gandamaneni, H. Data-Centered Program Understanding, in *IEEE int. conf. on Software Maintenance*, Victoria, Canada, 1994.

[JORI92] Joris, M., Van Hoe, R., Hainaut, J.-L., Chandelon, M., Tonneau, C. and Bodart, F. et al. PHENIX: Methods and Tools for Database Reverse Engineering, in *Proc 5th Int. Conf. on Sofwazre Engieering and Applications*. Toulouse, EC2 Publish, 1992.

[NANC96] Nanci, D. and Espinasse, B. *Ingénierie des systèmes d'information Merise - Deuxième génération*, SYBEX, 1996.

[PETI94] Petit, J.-M., Kouloumdjian, J., Bouliaut, J.-F. and Toumani, F. Using Queries to Improve Database Reverse Engineering, in *Proc of the 13th Int. Conf. on ER Approach*, Manchester, 1994. Springer-Verlag.

[RUGA95] Rugaber S. Program Comprehension. Technical report, College of Computing, Georgia Institute of Technology, 1995.

[WEIS84] Weiser, M. Program Slicing, *IEEE TSE*, **10**, 352-357, 1984.