A model-based approach for extracting business rules out of legacy information systems

Valerio Cosentino

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Thèse de Doctorat

Valerio COSENTINO

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

Discipline : Informatique et applications
Laboratoire : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 18 décembre 2013

École doctorale : 503 (STIM)
Thèse n° : 2013 EMNA0138

A Model-Based Approach for Extracting Business Rules out of Legacy Information Systems

JURY

Rapporteurs :	**M. Marco BRAMBILLA**, Maître de conférences, Politecnico di Milano **M. Xavier BLANC**, Professeur, Université de Bordeaux 1
Examinateurs :	**M. Jean-Yves LAFAYE**, Professeur, Université de la Rochelle **M. Jean-Claude ROYER**, Professeur, École des Mines de Nantes
Invité :	**M. Philippe BAUQUEL**, Directeur de la recherche et du développement Rational, IBM France
Directeur de thèse :	**M. Jordi CABOT**, Maître de conférences, École des Mines de Nantes
Co-directeur de thèse :	**M. Patrick ALBERT**, Directeur de recherche, Center for Advanced Studies IBM France

Acknowledgements

I would like to thank many people (if you want to look for your name, check Fig. 1 :)). First of all, I would like to express my sincere gratitude to my advisor Jordi Cabot for the continuous support of my Ph.D study and research. I address a special thank to IBM for the financial support and in particular to the co-advisor Patrick Albert to set the basis of this Ph.D, and Philippe Bauquel and Jacques Perronnet to their help on the research topic. I thank Marco Brambilla, Xavier Blanc and Jean-Yves Lafaye to accept being part of my jury and Ruth Raventos to follow my work during these three years.

I thank the team members of AtlanMod, my family, Maria and my friends that were immensely patience and encouraged me through the good and not so good moments, helping me to arrive to this point. Finally, I thank my football team Saint-Joseph de Porterie to let me release the stress that *from time to time* the Ph.D has caused to me.

Figure 1: Acknowledgements' cloud

Contents

1

Resumé étendu

1.1 Contexte

Durant les dernières décennies, les Systèmes d'Informations ont été largement employés par les entreprises pour les aider dans leur métier. Ils ont été utilisés pour manipuler des grandes quantités de données et pour automatiser les politques commerciales établies par l'entreprise. Au fil du temps, ces politiques ont été modifiées et par conséquent la logique métier correspondante, incorporée dans le Système d'Information, a évolué pour s'adapter à ces changements.

Le monde des affaires d'aujourd'hui est très dynamique, ainsi les organisations sont obligées de modifier constamment et rapidement leurs politiques commerciales pour suivre l'évolution du marché. Les nouvelles technologies, comme les Systèmes de Gestion de Règles Métier (BRMSs), permettent de manipuler plus facilement la logique métier, puisque celle-ci est devenue plus indépendante des aspects techniques du système. D'autre part, les anciens Systèmes d'Information ne sont pas capables de répondre rapidement à ces nouvelles exigences du marché, car ils n'ont pas été conçus pour fournir une séparation claire entre la logique métier qu'ils contiennent et les spécificités techniques du système (e.g., langage de programmation utilisé, architecture, etc.).

Néanmoins, les organisations craignent d'entreprendre un processus de migration vers de nouvelles technologies en raison du risque et du coût qu'un tel processus implique. D'une part, le risque est lié à l'absence d'une vision claire du système, puisque la documentation concernant la logique métier et le code source n'est généralement pas à jour. En outre, ces organisations doivent souvent faire face à une perte de connaissance technique au fil du temps en raison du départ

des développeurs originaux. Tous ces facteurs ne garantissent pas aux entreprises d'avoir toute leur logique métier répliquée et fonctionnant dans le nouveau système. En particulier, ceci est essentiel pour des entreprises (comme des banques, des compagnies d'assurcance, etc.) se situant dans des milieux où une défaillance du système peut comporter des graves conséquences financières et juridiques.

D'autre part le coût concernant la migration des anciens systèmes est en général prohibitif, car le processus de migration correspondant doit être conçu sur mesure pour chaque système. Selon une étude du Tactical Strategy Group, le coût pour rémplacer une seule ligne de code a été estimé à environ vingt-cinq dollars [1].

Plusieurs entreprises de la Technologie de l'Information financent des recherches pour offrir un processus de migration pour les anciennes applications de leurs clients vers des infrastructures modernes. Le but de ce processus est d'extraire et de moderniser la logique métier embarquée dans ces vieux systèmes, tout en préservant le reste de l'infrastructure en place. Malheureusement , cela implique des problèmes complexes et des défis qui concernent l'identification et l'extraction de la logique métier dans le code source. Ces défis sont au cœur de cette thèse.

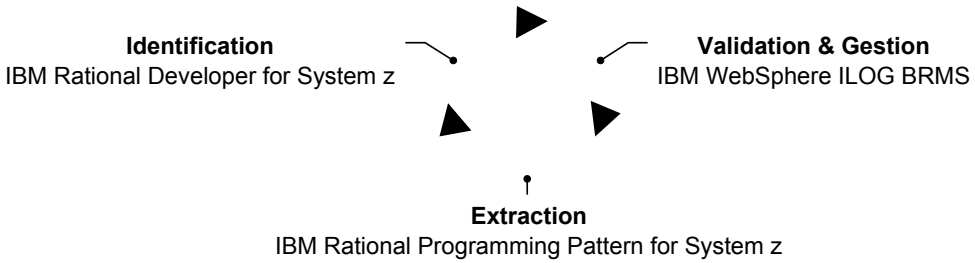

Figure 1.1: Projet de modernisation des systèmes héritées chez IBM

IBM est une des entreprises profondément intéressée par ces défis, puisque plusieurs de ses clients s'appuient encore sur d'anciens systèmes. L'extraction de la logique métier est une partie intégrante de l'initiative de modernisation des systèmes hérités au sein de IBM (fig.1.1). Elle consiste à combler les caractéristiques de trois produits différents pour identifier, maintenir et gérer les règles métier codées dans des systèmes d'explotations qui s'appuient sur des technologies anciennes et nouvelles. En particulier, IBM Rational Developer et IBM Rational Programming Pattern for System z permettent d'identifier et extraire la logique métier, tandis que IBM WebSphere ILOG BRMS est axé sur la validation et la gestion de cette logique. Enfin, ce dernier produit peut être utilisé à son tour pour piloter les redéfinitions des règles métier au sein des anciens systèmes informatiques. Par conséquent, cette thèse a été financée par IBM dans le cadre d'un accord avec l'École des Mines de Nantes. IBM a fourni son expertise, des cas d'étude et des outils.

1. http://www.csis.ul.ie/cobol/course/COBOLIntro.htm

1.2 Description du problème

La logique métier intégrée dans un système est composée de règles de gestion [1], qui sont des actions ou des procédures qui définissent ou contraintent un aspect précis du métier. Ces règles représentent les politiques commerciales de l'entreprise dans le système.

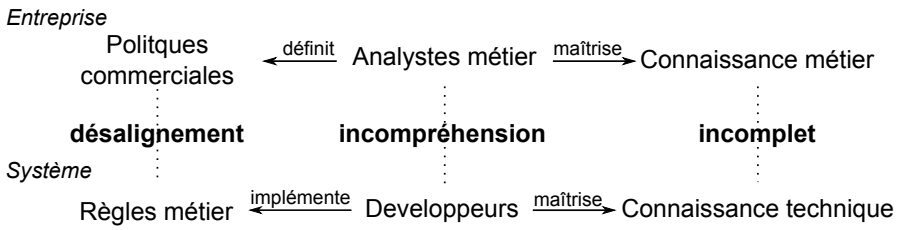

Figure 1.2: Problèmes fréquents dans les Systèmes d'Information

Dûes à diverses raisons trouver, comprendre et modifier le code source qui implémente les règles métier au cœur d'un ancient système sont des activités qui consomment beaucoup de temps et qui sont source d'erreurs à cause des différentes raisons (fig. 1.2) .

Tout d'abord, la localisation des politiques commerciales dans un contexte technique n'est pas evident, car les règles sont définies à travers d'un langage de programmation qui implemente le système et par conséquent, des connaissances techniques et métier sont nécessaires pour isoler ces règles. Malheureusement , ces connaissances sont détenues par deux différents types d'experts: les analystes et les développeurs. Par conséquent, cette séparation provoque souvent des écarts entre ce que le système implemente et ce que les politiques commerciales définissent.

Une autre source de complexité est liée au fait que les règles métier sont dispersées dans le code source. Par conséquent, lors de l'alignement des règles métier aux politiques commerciales correspondantes, les équipes de développement doivent être capable de modifier le code sans affecter d'autres règles métier. Cette tâche est particulièrement complexe pour les grands systèmes à cause de la quantité de lignes de code à analyser.

Finalement, la connaissance du système est souvent incomplète à cause du départ de développeurs originaux et d'une documentation limitée ne reflètant pas la mise en œuvre actuelle du système. Par conséquent, au fil du temps des règles métier dupliquées et des écarts avec les politiques commerciales peuvent apparaître dans le système.

1.3 État de l'art

Le processus pour l'identification de règles métier dans un système informatique est appelé Business Rule Extraction (BREX). Il est utilisé pour donner une vision spécifique du système par rapport à la logique métier qu'il contient[2]. Le BREX fait partie du domaine de la compréhension du logiciel[3], qui est le pilier de plusieurs activités du génie logiciel, tels que la maintenance, l'évolution et la rétro-ingénierie. Les relations entre ces activités et le BREX sont représentées dans la Fig.1.3.

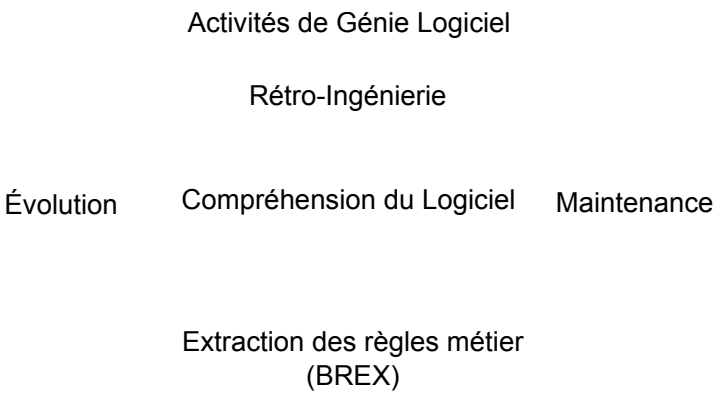

Figure 1.3: Relations entre le BREX et les activités du génie logiciel

Le processus BREX peut être appliqué à différentes parties du système. Dans cette thèse, nous nous concentrons sur l'analyse de la partie comportementale et structurelle d'un système d'explotation, où la première représente la logique applicative, tandis que la seconde concerne la gestion des données.

Chacune de ces deux parties nécessite un processus BREX différent. Le processus BREX pour la partie comportementale est axé sur des techniques de compréhension des programmes. Ces techniques sont appliquées sur le code source et leur objectif est de découvrir et de représenter les variables et les morceaux de code source liés au métier. Par conséquent et conformément à ces techniques, un processus BREX pour la partie comportementale du système est principalement composé de trois étapes ([2], [4], [5], [6]) qui visent à:

- identifier les variables métier dans le système,
- récupérer les morceaux de code relatives à ces variables,
- représenter les morceaux de code identifiés.

La première étape contient des heuristiques pour faciliter l'identification des variables métier, la seconde étape est basée sur des techniques d'analyse de programme telles que l'analyse de fluxs de données et de contrôle ainsi que des opérations de découpage du code, tandis que la dernière étape extrait la logique identifiée dans des formats compréhensibles.

D'autre part, le processus BREX pour la partie structurelle du système vise à récupérer les règles de gestion situées dans les structures de bases de données telles que tables, procédures stockées et triggers. Ce processus fait partie de la rétro-ingénierie des bases de données[7] et il est composé par deux étapes qui visent à:

- extraire et représenter les définitions des données dans une représentation de niveau supérieur,
- extraire les contraintes définies dans une base de données et les appliquées sur la représentation dérivée précédemment.

La première étape définit les règles pour passer d'un schéma de base de données au modèle conceptuel correspondant. La deuxième étape consiste à identifier les contraintes d'intégrité de la base de données et ensuite à les exprimer dans un format compatible avec le langage utilisé pour définir le modèle conceptuel.

Dans cette thèse, nous présentons des solutions pour extraire et représenter les règles de gestion pour des systèmes implementés en Java et COBOL, pour la partie comportementale, et pour des base de données relationnelles pour la partie structurelle. Nous concentrons notre analyse sur Java, COBOL et les bases de données relationnelles, car la plupart des systèmes qui dépendent de ces technologies sont encore utilisés dans les entreprises.

Les solutions proposées sont basées sur l'Ingénierie Dirigée par les Modèles (IDM [8]). Les techiques de l'IDM offrent une représentation homogène et abstraite du système en évitant tous problèmes technologiques liés au langage de programmation utilisé dans le système. En outre , les solutions de l'IDM offrent des approches génériques et modulaires et enfin, lors de la représentation d'un système sous forme de modèle, il est possible de s'appuyer sur la pléthore d'outils de l'IDM disponible sur le marché.

1.4 Ingénierie Dirigée par les Modèles

L'IDM[8] est une discipline du génie logiciel qui considère les modèles comme des citoyens étant au cœuer des processus d'ingénierie directe et de rétro-ingénierie. Cette nouvelle façon d'utiliser les modèles s'oppose aux approches précédentes, qui limitaient les modèles à un rôle passif (par exemple, de documentation) au cours des activités de génie logiciel.

L'adoption de l'IDM apporte de nombreux avantages à ces activités. En particulier, l'IDM améliore la maintenabilité et la qualité des systèmes (par exemple [9]) grâce à l'introduction des modèles et des transformations entre modèles. Les premiers répresentent un niveau d'abstraction superieur par rapport au code source, tandis que les derniers permettent d'automatiser des tâches répétitives dans les processus de développement logiciels. Dans la suite, nous décrivons les modèles et les

transformations.

1.4.1 Modèles

L'hypothèse de base de l'IDM est que les modèles sont la représentation correcte pour gérer tous les artefacts au sein d'un processus de génie logiciel; par conséquent, les modèles sont considérés comme le concept unificateur dans l'IDM. Les modèles sont définis selon une architecture à trois niveaux représentée dans la Fig.1.4. Cette architecture est composée par des modèles, des métamodèles et des métamétamodèles.

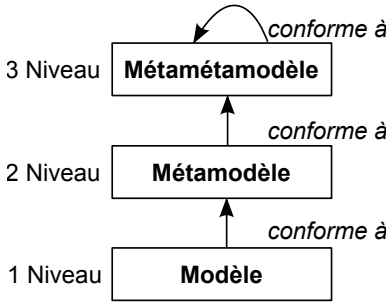

Figure 1.4: Architecture à trois niveaux dans l'IDM

Un modèle est une représentation partielle d'un système. D'autre part, la combinaison de différents modèles relatifs au même système peut être utilisée pour produire une vue globale de ce système. Chacun de ces modèles contient des éléments qui représentent des artefacts logiciels/concepts dans le monde réel. Ces concepts et leurs relations sont définis dans le métamodèle, le deuxième niveau de l'architecture de l'IDM.

Un métamodèle est lié à un modèle en fonction d'un rapport de conformité. Cette relation est équivalente à une relation programme - grammaire, de telle sorte que les programmes écrits dans un langage doivent être conformes aux règles grammaticales de ce langage ainsi que les modèles définis selon un métamodèle doivent être conformes aux concepts et relations de ce métamodèle.

Les métamodèles sont exprimés à leur tour en utilisant le métamétamodèle, le troisième niveau de l'architecture de l'IDM. Similarement à la relation modèle/ métamodèle, une relation de conformité est définie entre le métamodèle et le métamétamodèle; de telle façon qu'un métamodèle est défini en utilisant des concepts et des associations mémorisés dans le métamétamodèle. En outre, cette relation est équivalente à la relation entre la grammaire d'un langage de programmation donné et un langage pour définir des grammaires (par exemple, l'EBNF, la Forme de Backus-Naur étendue).

Cette représentation à trois niveaux est connue comme *Modelware*. Elle est similaire au *Grammarware*, qui représente l'espace technique où un langage est défini

selon sa grammaire correspondante. Par conséquent, l'EBNF est conceptuellement équivalent à un métamétamodèle, la grammaire d'un langage donné est au même niveau qu'un métamodèle, et une instance d'une grammaire est analogue à un modèle.

Enfin, les modèles, les métamodèles et les métamétamodèles peuvent être implementés selon différents standards. Par exemple , l'Object Management Group propose un métamétamodèle appelé Meta Object Facility (MOF) et différents métamodèles (UML: Unified Modeling Language [10], KDM: Knowledge Discovery Metamodel [11], etc.)

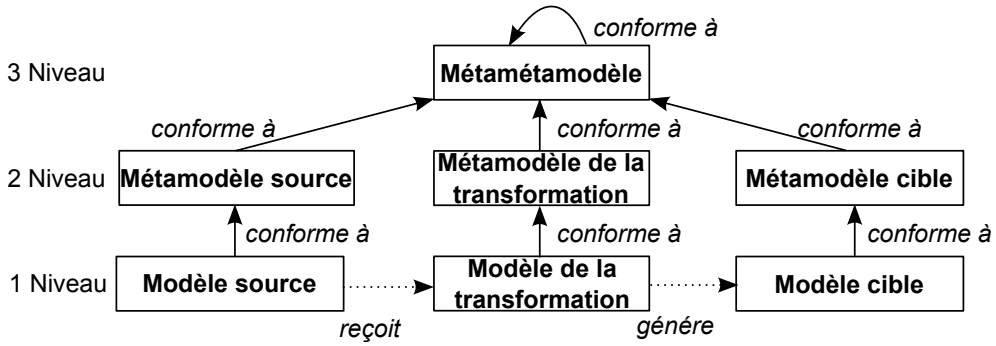

Figure 1.5: Transformation entre modèles

1.4.2 Transformations

La deuxième caractéristique de l'IDM est la manipulation de modèles, généralement mise en œuvre à travers de transformations entre modèles (Fig.1.5). Chaque transformation peut prendre en entrée un ou plusieurs modèles et générer un ou plusieurs modèles en sortie. En particulier, une transformation est capable de générer un modèle *cible* à partir d'un modèle *source* en utilisant leurs métamodèles respectifs.

Selon les métamodèles d'entrée et de sortie, deux types différents de transformations peuvent être définies. Une transformation est appelée endogène, si les métamodèles source et cible sont identiques. Au contraire, une transformation est appelée exogène si les métamodèles d'entrée et sortie sont différents.

En outre, les transformations entre modèles peuvent enregistrer des informations entre les éléments du modèle cible est les éléments correspondants du modèle source (traçabilité de [12]). En particulier, les relations entre ces deux types d'éléments sont appelées *traces*. Ces traces peuvent être utilisées pour comprendre et suivre les rélations entre les artefacts logicielles dans un processus dirigé par les modèles.

La traçabilité entre modèles peut être modélisée selon la représentation à trois niveaux de l'IDM[13]. Par conséquent, les informations de traçabilité sont mé-

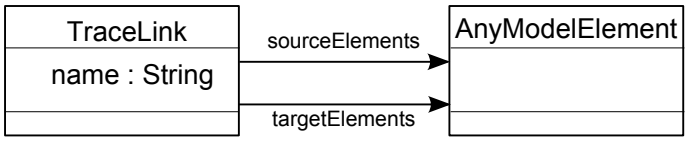

Figure 1.6: Métamodèle de traçabilité

morisées dans un modèle conforme à un métamodèle comme celui de la Fig. 1.6. Ce métamodèle est composé d'une classe *TraceLink*, qui mémorise les liens entre les éléments source et cible (*sourceElements* et *targetElements*) impliqués dans une règle de transformation. Généralement chaque *TraceLink* est identifié par un nom, qui est le nom de la règle de transformation.

Enfin, plusieurs langages de transformation sont disponibles sur le marché. Par exemple, ATL Transformation language[14], qui a été utilisé dans cette thèse et Query/View/Transformation[15] qui est le langage standard proposé par l'OMG.

1.5 Règles métier

Une règles de gestion est un concept qui est largement connu et utilisé actuellement. Plusieurs définitions en existent, puisque les praticiens et les organisations ont tendance à attribuer à ce concept différentes désignations. Ci-dessous, nous présentons certaines de ces définitions.

Selon E. Gottesdiener[16], les règles métier sont au cœur des exigences fonctionnelles, elles fournissent la connaissance derrière chaque processus d'entreprise. Pour I. Baxter et S. Hendryx [17], elles sont définies comme une obligation qui s'applique aux actions, pratiques et procédures de l'entreprise. Au contraire, le Business Rules Group [1] décrit une règle métier comme une déclaration qui définit ou contraint certains aspects de l'entreprise. Elle est destinée à contrôler ou influencer le comportement de l'entreprise. Enfin, IBM définit les règles de gestion comme tout ce qui capture et met en œuvre les politiques et pratiques commerciales d'une entreprise ([18]).

Selon les définitions précédentes, les règles de gestion sont des éléments clés à la fois pour les entreprises et leurs systèmes d'explotation. D'une part, elles sont utilisées pour décrire les politiques indépendamment des paradigmes et plates-formes techniques. D'autre part, elles conduisent le comportement du système et, lorsqu'elle sont implémentées dans un système, elle se mélangent avec les spécificités du langage de programmation que le système utilise.

Les règles de gestion sont composées de trois éléments: les termes métier, les opérateurs et les valeurs [19]. De telles constructions sont représentées dans la Fig. 1.7. Les termes métier représentent les objets qui font partie, ou sont affectés par, des règles métier. Les opérateurs permettent de comparer les propriétés ou les

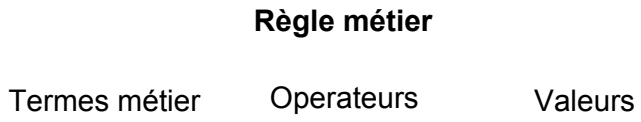

Figure 1.7: Elements d'une règle métier

caractéristiques des différents termes métier. Enfin, les valeurs peuvent être des nombres (ou expressions arithmétiques), du textes ou des termes métier prédéfinis.

Si les achats du client > 5000
alors le status du client = "Or"

Figure 1.8: Exemple d'une règle de gestion

Dans la Fig. 1.8 un exemple de règle de gestion est presentée. La règle définit qu'un client devient un client d'*or* dès que ses achats cumulés ont dépassé 5000 dollars. *Client*, *achats*, *statut* et *or* sont des termes métier, tandis que les opérateurs sont exprimés par l'opérateur logique *supérieure* et l'assignation. Enfin, les valeurs sont représentées par le term *or* et la valeur numérique 5000.

Les règles de gestion sont classées en quatre catégories[1]:

- **Définitions de termes métier.** Ces définitions précisent le vocabulaire pour exprimer des règles de gestion.
- **Faits.** Ils décrivent les relations entre les termes métier.
- **Contraintes.** Elles limitent les comportements d'organisation en définissant ce qui est autorisé et interdit dans les politiques commerciales de l'entreprise.
- **Dérivations.** Elles définissent la manière dont la connaissance de l'entreprise (termes et faits) est transformée dans d'autres connaissances.

1.5.1 Règles métier dans la partie comportementale

Cette section donne une définition des règles de gestion au niveau du code source et comment les quatre catégories décrites précédemment sont mappées dans un langage de programmation.

Les définitions de règles métier données précédemment ne peuvent pas être utilisées pour identifier des règles dans du code source, puisque les règles de gestion exprimées à haut niveau d'abstraction doivent être implementées dans un langage de programmation. Malheureusement, cette action implique que les règles soient dispersées dans le code source. Par conséquent, des relations (Fig. 1.9) doivent être définies entre les quatre catégories décrites précédemment et les structures d'un langage de programmation.

- **Définitions de termes métier.** Les définitions des termes métier sont dispersées dans le code. La plupart du temps, elle peuvent être récuperées dans des

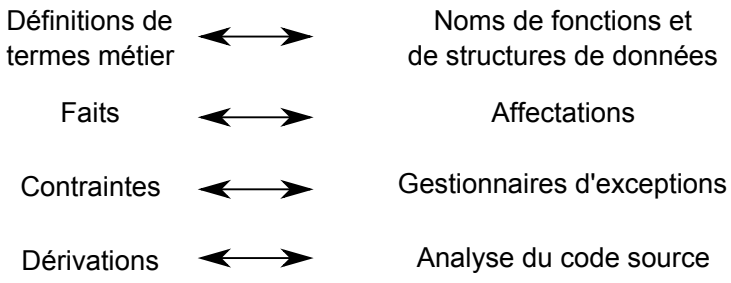

Figure 1.9: Catégories de règles métier au niveau du code source

structures de données et des noms de fonctions. Souvent, ces noms doivent être traités pour extraire les termes métier correspondants (par exemple, un nom de variable *cli* doit être transformé dans le terme métier *client*). En outre, ces termes peuvent être trouvés aussi bien dans les commentaires écrits dans le source code que dans la documentation du système.

– **Faits.** Ils représentent les relations entre les variables définies comme affectations dans le code source. Ces relations peuvent être dérivées en appliquant des techniques d'analyse de fluxs de données[20].
– **Contraintes.** Elles sont généralement mises en place en utilisant des gestionnaires d'exceptions dans le code source.
– **Dérivations.** Elles sont représentées par des déclarations liées à une variable donnée (ou groupe de variables) dans le même chemin d'exécution. Par conséquent, l'identification des règles de dérivation est une activité complexe, car elle implique l'analyse du code source. L'identification de dérivations s'appuie sur les termes métier découverts précedemment.

1.5.2 Règles métier dans la partie structurelle

Cette section fournit une définition des règles de gestion pour la partie structurelle du système et comment les termes métiers, les faits, les contraintes et les dérivations sont mappés sur les structures des bases de données .

Dans une base de données, une règle de gestion est représentée par des contraintes d'intégrité . Selon [21], ces contraintes sont divisées en trois catégories: inhérentes, implicites et explicites .

Le contraintes inhérentes concernent les relations entre les tables et leurs colonnes. Les contraintes implicites concernent les clés primaires et étrangères utilisées respectivement pour identifier un enregistrement dans une table et pour définir des relations entre les tables de la base de données. Enfin, les contraintes explicites concernent les contraintes définies sur les tables, les colonnes (par exemple , CHECK, NOT NULL, etc.) ainsi que les assertions, les procédures stockées et les triggers.

Alors que les contraintes inhérentes et implicites précisent les termes et les faits

métier, les contraintes explicites sont les règles de gestion qui définissent les politiques commerciales de l'entreprise.

Des relations (fig.1.10) sont définies entre les différentes catégories des règles métier et les contraintes d'intégrité des bases de données. Dans la suite chaque catégorie est analysée.

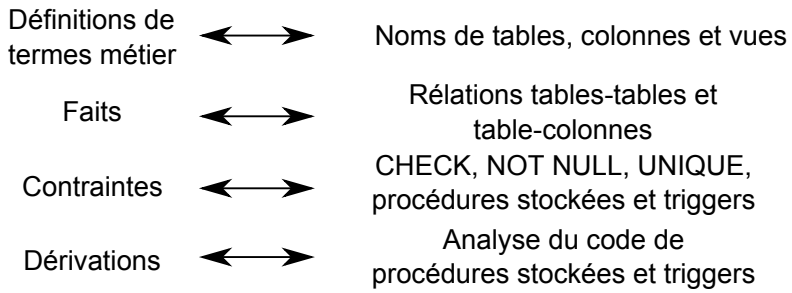

Figure 1.10: Catégories de règles de gestion dans une base de données

- **Définitions de termes métier.** Les définitions des termes métier proviennent des tables et vues contenues dans la base de données. En particulier, les termes métier sont extraits à partir du nom de tables, vues et colonnes. Puisque ces noms décrivent souvent des termes d'affaires, ils n'ont pas besoin d'être traités.
- **Faits.** Ils sont définis comme les relations entre les tables ainsi que comme relations intrinsèques entre tables et leurs colonnes.
- **Contraintes.** Elles sont dérivées à partir des contraintes explicites
- **Dérivations.** Elles peuvent être extraites en analysant le code source des procédures stockées et des triggers.

1.6 Un framework BREX dirigé par les modèles

Cette section présente un aperçu de notre processus BREX. Elle décrit les étapes qui composent ce processus et comment elles sont adaptées pour traiter avec les parties comportementales et structurelles d'un système d'explotation.

Dans cette thèse, une règle de gestion est définie pour la partie comportementale du système comme un ensemble de déclarations qui sont relatives à une variable métier dans le même chemin d'exécution. Nous considérons les variables métier, celles qui sont utilisées dans les opérations mathématiques, les commandes d'entrée/sortie, les conditions d'instructions conditionnelles et les déclarations d'initialisation. Selon notre expérience avec les cas d'études fournis par IBM et avec certains travaux antérieurs ([2] [4] , [5], [6]), ces heuristiques sont capables d'identifier des règles métier dispersées dans le code. De toute évidence, ces heuristiques ne sont pas complètes , mais le framework permet d'ajouter des heuristiques supplémentaires facilément.

Au contraire, pour la partie structurelle du système, une règle de gestion est définie comme une contrainte déclarative dans le schéma d'une base de données ([22], [23]) ou comme la condition qui provoque l'exécution d'un trigger.

Le processus BREX mis en œuvre est réalisé dans un environnement IDM. Pour ce faire, une opération spécifique est nécessaire pour passer de l'espace technologique défini par le langage de programmation employé dans le système (*Grammarware*) vers l'espace des modèles (*Modelware*). Ce dernier espace fournit une représentation du code en utilisant un modèle[24]. Les règles de gestion sont calculées en analysant ce modèle, qui contient les informations statiques du système. En particulier, pour la partie comportementale, il représente l'Arbre Syntaxique Abstrait (AST) du code source avec des liaisons entre les éléments du modèle (par exemple, les usages d'une variable sont liés à la déclaration de la variable correspondante et vice-versa, les appels d'une méthode sont liées à la déclaration de la méthode et vice-versa , etc.). Au contraire, pour la partie structurelle du système, le modèle contient les informations des structures définies dans le schéma de la base de données (tables, colonnes , vues , triggers, contraintes déclaratives).

Enfin, notre processus BREX se base sur des techniques d'analyse statique appliquées au modèle dérivé à partir du code source. Par conséquent, pour la partie comportementale du système , nous sommes en mesure d'exclure de ce modèle le code qui n'est pas accessible en effectuant une analyse du Graphe de Flux de Contrôle (GFC). Au contraire, l' identification de code inutile ou mort est hors de la portée de cette thèse.

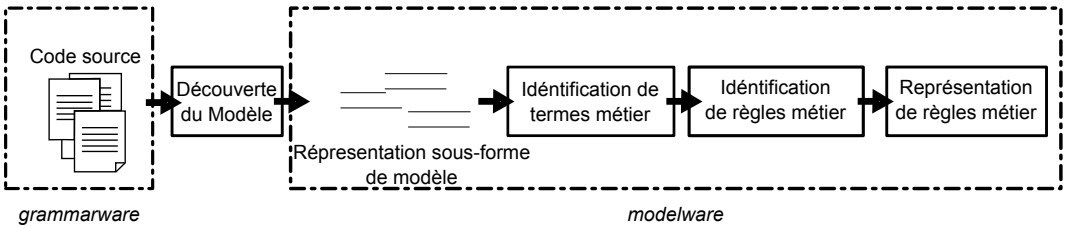

Figure 1.11: Framework pour l'extraction des règles métier

Dans la Fig.1.11, les étapes du framework qui implemente notre processus BREX sont présentées. Elles sont respectivement, la découverte du modèle, l'identification des termes métier, l'identification des règles métier et la représentation des règles métier. La première étape prend en entrée le code source exprimée dans un langage de programmation donné et génère un modèle spécifique à la plate-forme cible (Platform Specific Model, PSM). Ce modèle a une correspondance directe avec le code, de sorte que dans cette étape il n'y a pas de perte d'information et tous les éléments du code source sont traduits en éléments du modèle. Enfin, le modèle obtenu est ensuite manipulé dans les étapes suivantes qui composent le framework BREX.

L'identification des termes métier se concentre sur la découverte des variables

métier utilisées dans le système. Ces variables sont ensuite utilisées pour piloter l'étape d'identification de règles de gestion, où selon la partie du système analysée (comportementale ou structurelle), différentes heuristiques peuvent être employées. Enfin, dans la dernière étape qui concerne la représentation de règles métier, les règles découvertes sont visualisées à l'aide d'artéfacts textuels et graphiques.

La découverte du modèle, l'identification des termes métier, l'identification des règles métier et la représentation des règles métier sont décrites dans la suite.

1.6.1 Découverte du modèle

La découverte du modèle est utilisée pour présenter l'hétérogénéité d'un système informatique comme une représentation homogène et uniforme.

Cette étape est particulièrement utile lors de l'analyse des systèmes qui s'appuient sur des technologies différentes. Le but ultime de cette étape est d'obtenir un ou plusieurs modèles du système, puis de travailler directement sur ces modèles. Chaque modèle répresente un point de vue précis du système, tel que le code source, les schémas de base de données, etc.

Dans cette thèse, nous nous sommes appuyés sur différents outils de l'IDM (Fig.1.12) pour générer des modèles à partir de code source Java et COBOL, ainsi que des implémentations de bases de données relationnelles. En particulier, nous avons utilisé MoDisco [25] pour Java, IBM COBOL Application Model [2] pour COBOL, et enfin Xtext [26] pour obtenir un modèle qui était capable de répresenter des schémas de base de données et des triggers. De tels modèles ont une correspondance directe avec le code source, les schémas et les triggers.

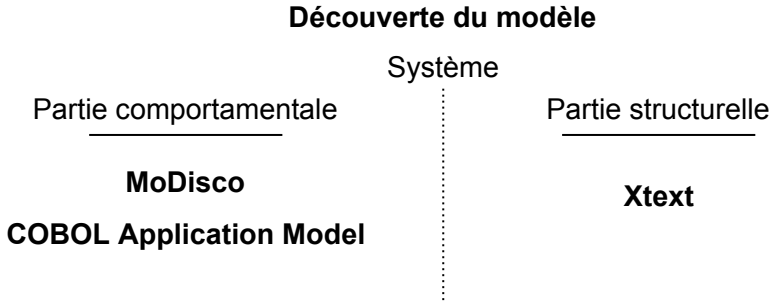

Figure 1.12: Outils utilisés dans l'étape de découverte du modèle

1.6.2 Identification de termes métier

L'identification de termes métier localise les concepts métier dans le modèle du système. Puisque les termes métier sont représentés de différentes manières en

2. http://tinyurl.com/IBMCobolApplicationModel

fonction de la partie du système analysé (fig. 1.13), nous avons développé plusieurs heuristiques capables d'identifier ces termes.

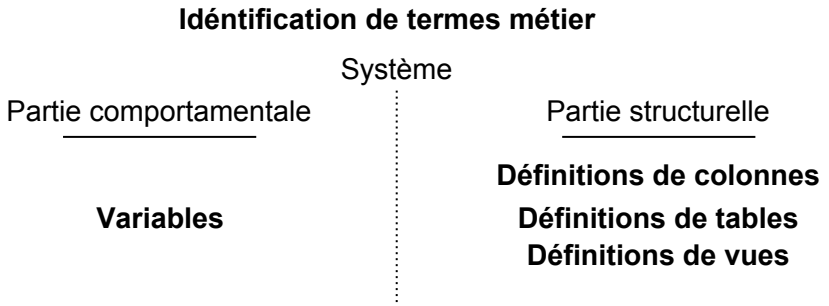

Figure 1.13: Éléments utilisés pour l'identification de termes métier

Pour ce qui concerne la partie comportementale du système, pour Java et COBOL, nous avons concentré notre analyse sur les variables utilisées dans le code afin d'identifier celles liées à des concepts métier. Les heuristiques mises en œuvre considérent comme variables métier toutes les variables qui apparaissent dans les opérations mathématiques, dans les conditions de déclarations conditionnelles, dans les déclarations d'entrée/sortie et dans les déclarations d'initialisation. Nous sommes conscients du fait que ces heuristiques ne permettent pas de localiser toutes les variables métier dans un système. Cependant, d'autres heuristiques peuvent être facilement branchées à notre framework grâce à la modularité fournies par l'IDM.

Au contraire, par rapport à la partie structurelle du système, pour les bases de données relationnelles, nous avons concentré notre analyse sur les tables, les vues et les déclarations de colonnes composant le schéma d'une base de données. Tous ces éléments sont considerés pértinents, car ils ont souvent une correspondance directe avec les concepts du métier.

Enfin, les termes métier définis dans les parties comportementales et structurelles du système sont reliés aux source code correspondant à travers de la traçabilité offerte par l'IDM.

1.6.3 Identification de règles métier

L'identification de règles métier fournit les moyens de localiser les règles de gestion relatives à un ou plusieurs termes métier. En outre, cette étape est utilisée pour spécifier une représentation interne des règles de gestion identifiées, et elle offre un support de traçabilité pour connecter les règles métier découvertes au code source correspondant.

Cette étape est composée d'opérations capables de faire face aux différents paradigmes de programmation utilisés dans la partie comportementale et structurelle des systèmes (Fig. 1.14). En particulier, l'identification des règles métier dans la partie comportementale est basée sur l'analyse de flux de contrôle et de

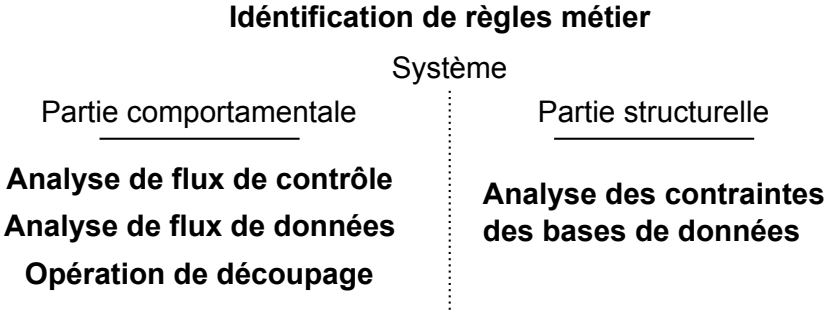

Figure 1.14: Techniques utilisées pour l'identification des règles métier

données ainsi que sur l'opération de découpage du code. Ces opérations sont appliquées au modèle qui représente le code source. Le résultat de cette étape est un modèle qui contient les parties du code qui concernent les règles de gestion identifiées.

D'autre part, l'identification de règles gestion pour la partie structurelle du système repose sur l'analyse des contraintes de base de données déclaratives et opérationnelles.

Toutes les opérations mentionnées ci-dessus sont décrites dans la partie restante de cette section.

Analyse de flux de contrôle

L'analyse de flux de contrôle est utilisée pour découvrir le graphe de flux de contrôle (GFC [27]) d'un programme en utilisant des techniques d'analyse statique du code source. Le GFC contient les informations de tous les chemins qui peuvent être traversés par un programme lors de son exécution. Par conséquent, cette analyse permet d'identifier les morceaux de source code inaccessibles.

Dans un contexte dirigé par les modéles, cette analyse est basée sur la navigation de l'Arbre Syntaxique Abstrait (AST) incorporé dans le modèle qui représente un programme donné. L'AST exprime les informations du modèle (par exemple, les éléments du modèle, les relations , etc) dans une représentation arborescente. Une telle représentation est exploitée selon la sémantique du langage de programmation (Java ou COBOL) pour définir les heuristique utilisées pour générer le GFC.

Pour cette étape, le modèle en entrée représente l'application, tandis que le résultat est le modèle d'entrée enrichi avec les informations du GFC. En particulier, conformément aux heuristique définies, une liste de successeurs est créée et fixée à chaque élément du modèle.

Analyse de flux de données

L'analyse de flux de données est utilisée pour mettre en évidence les relations entre les variables du programme. Elle permet d'identifier les règles métier codés

dans un programme. Cette étape consiste à traverser le modèle représentant le GFC afin de relier les déclarations qui affectent/utiliser les mêmes variables.

En particulier, dans cette thèse, cette étape est utilisée pour créer des relations entre chaque déclaration qui se réfère à une certaine variable avec la suivante (ou précédente) déclaration qui fait référence à la même variable. Par conséquent, chaque élément du modèle mémorise une liste de successeurs (ou prédécesseurs) qui contiennent la même variable.

Pour cette étape, le modèle en entrée est le GFC du programme, tandis que le résultat est le même modèle plus les informations concernant le flux de données.

Opération de découpage du code

L'operation de découpage (slicing) est utilisée pour récupérer les parties du code source liées à une variable donnée (ou plusieurs). Ces parties du code représentent la règle métier et son contexte. En particulier, une règle métier est composée des déclarations qui utilisent la variable métier en question, tandis que le contexte comprend les conditions qui déclenchent ces déclarations. Ces conditions se retrouvent dans les possibles chemins d'exécution du programme.

Nous avons mis en place cette étape en nous appuyant sur des techniques de découpage statique à rebours (backward static slicing) et par bloque amovible (removable block slicing), de sorte que pour une déclaration donnée contenant une variable métier, l'opération de découpage navigue à partir de cette déclaration à rebours jusq'au début du programme [3], en sélectionnant les déclarations qui sont liées à cette variable. En outre, le découpage par bloc amovible est utilisé pour ignorer les blocs de code qui n'ont pas de relation avec la variable sélectionnée.

L'analyse statique a été choisie pour deux raisons. Tout d'abord, l'étape de découverte du modèle génère un “instantané" du système sous forme de modèle, qui représente les informations statiques du système. Par conséquent, les informations dynamiques ne sont pas mémorisées dans le modèle découvert. Deuxièmement, étant donné que le processus de découverte est généralement une activité qui consomme beaucoup de temps, le modèle qui représente le code source est généré une fois seulement. Donc, à notre avis, dans un contexte dirigé par les modèles, des modèles dynamiques du système d'information ne sont pas appropriés au processus d'extraction des règles métier.

D'autre part, les techniques de découpage statique à rebours et par bloque amovible ont été choisies, car elles bénéficient de la représentation arborescente des modèles. En particulier, en utilisant cette représentation, nous sommes en mesure d'ignorer les éléments du modèle qui ne contiennent pas d'informations métier en utilisant les

3. Notez que le début d'un programme dépend du langage de programmation utilisé, par exemple, en Java, il peut être la méthode *main* ou une action associée à un bouton, tandis que en COBOL, il pourrait être la première instruction dans la division des procédures.

relations hiérarchiques entre eux.

Analyse des contraintes des bases de données

L'analyse des contraintes des bases de données se concentre sur l'identification et l'extraction des contraintes déclaratives et operationnelles dans les définitions des tables et des triggers. Dans cette thèse, nous représentons ces contraintes en OCL. Elles s'appuient sur un modèle conceptuel UML dérivé du schéma de la base de données.

Le processus qui analyse les contraintes déclaratives prend en entrée le modèle qui représente le schéma de la base de données et produits en sortie des règles métier exprimées en OCL. Ces règles sont sémantiquement équivalentes qu'aux *clés primaires* ainsi que aux contraintes *UNIQUE* , *NOT NULL* et *CHECK*.

D'autre part, l'analyse des contraintes opérationnelles extrait des règles à partir des triggers. Ces derniers sont analysés par rapport aux gestionnaires d'exceptions qu'ils contiennent [4]. Les déclarations qui gestionnent des exceptions sont généralement utilisées pour coder les violations des politiques commerciales de l'entreprise. Par conséquent, les conditions qui déclenchent ces exceptions sont censées être liées aux métier et donc elles sont traduites en règles métier.

Le processus qui analyse les contraintes operationnelles prend en entrée des modèles qui correspondent aux triggers et produit en sortie les contraintes OCL correspondantes. Puisque les triggers contiennent souvent des operations SQL, nous avons defini des traductions entre des éléments du language SQL (projections, joins, etc.) et les éléments qui composent les contraintes OCL.

1.6.4 Représentation des règles métier

La représentation des règles métier (Fig. 1.15) est la dernière étape de notre framework. Son objectif est de générer des artefacts compréhensibles qui décrivent les règles de gestion identifiées. Cette étape est composée de deux opérations: l'extraction du vocabulaire et la visualisation des règles métier. D'une part, le vocabulaire vise à fournir des verbalisations pour les éléments qui composent les règles métier (variables, tables de base de données, etc.) et qui ont été localisés dans l'étape d'identification des termes métier. D'autre part, l'étape de visualisation est utilisée pour faciliter la compréhension des règles métier trouvées en fournissant des sorties textuelles et graphiques.

Enfin, au cours de chacune de ces étapes, les informations de traçabilité sont propagées aux modèles de sortie du framework, de telle sorte que ceux-ci sont reliés aux modèles qui répresentent le code source.

4. Notez que cette approche peut être appliquée égalment aux procédures stockées

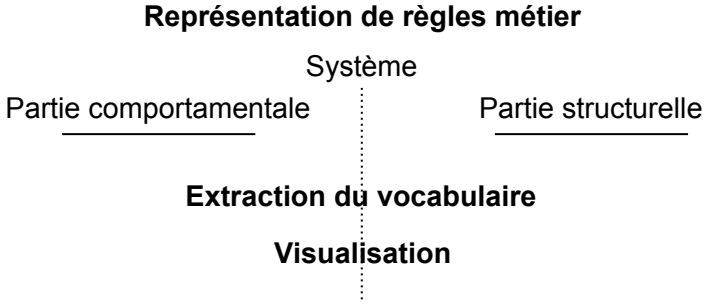

Figure 1.15: Sous-étapes de la représentation des règles de gestion

Extraction du vocabulaire

L'extraction du vocabulaire est une opération facultative qui vise à fournir des verbalisations/descriptions des éléments qui composent les règles métier.

Dans la partie comportamentale du système, cette opération est utilisée pour générer un vocabulaire composé par des paires <*nom de variable ou function, description*> . Ce vocabulaire peut éventuellement être modifié par l'utilisateur.

Le vocabulaire pour les programmes Java est généré automatiquement. Il contient une liste de noms de classes, variables et méthodes et leurs descriptions correspondantes. Ces descriptions sont générées en suivant la façon de codage propre de Java (<*getAttribute( ) , " get Attribute"* >). D'autre part, le vocabulaire pour les programmes COBOL se focalise seulement sur les variables métier. Etant donné que les identificateurs de ces variables sont généralement plus courts par rapport à ceux codés en Java, une intervention humaine est nécessaire pour compléter les descriptions correspondantes.

Par rapport à la partie structurelle du système, l'étape d'extraction du vocabulaire consiste à générer un modèle qui contient soit les termes métier soit les relations correspondantes qui existent parmi ceux-ci dans la base de données. En particulier, l'entrée de cette étape est le modèle qui correspond au schéma de la base de données, tandis que la sortie est un modèle conceptuel défini à l'aide d' UML.

Visualisation des règles métier

L'étape de visualisation fournit des représentations des règles métier identifiées. Ces représentations peuvent être soit du texte, pour analyser les règles une à une, soit des artefacts graphique, pour analyser les relations entre les règles obtenues. Les sorties de cette étape ont comme but ultime celui d'externaliser le format interne des règles trouvées, qui a été défini dans l'étape d'identification des règles métier.

Dans la partie comportementale du système, l'étape de visualisation peut utiliser les informations contenues dans le vocabulaire, s'il a été défini. La représentation sous forme textuelle est dérivée par des transformations de modèle-à-texte,

où le modèle représente le format interne des règles et le texte est une représentation basée sur le vocabulaire ou sur le code source. Au contraire, la représentation graphique est basée sur Portolan [28], un outil de l'IDM, qui fournit une solution de cartographie dans un contexte dirigé par les modèles. Cet outil vise à combler le vide qui existe entre les données et leur visualisation graphique.

Au contraire, dans la partie structurelle du système, l'étape de visualisation consiste à générer des représentations textuelles sous forme de contraintes OCL, car les règles identifiées sont exprimées à travers des modèles OCL. Enfin, la représentation graphique s'appuie sur UML, qui fournit un ensemble de notation graphique pour créer des modèles visuels.

2

Introduction

2.1 Context

In the past decades, information systems have been largely employed by organizations to assist their business. They have been used to manipulate large amounts of data and automate business decisions according to a set of organization policies. Over time, these policies have been modified and as consequence the business logic representing such policies and embedded in the systems evolved to fit the changes.

Today's business world is very dynamic, thus organizations have to tune constantly and quickly their policies to follow the market changes. Newer technologies, such as Business Rules Management Systems (BRMSs), make possible to manipulate the business logic in an easy way, since such a logic is made independent from the technical/implementation aspects of the system. On the other hand, old (i.e., legacy) information systems lack on agility to respond to these new requirements due to the technologies they rely on, that are not designed to work in such a dynamic context. In particular, legacy systems do not provide a clear separation between the business logic and the specificities of the programming language employed.

Nevertheless, organizations are afraid to undertake a migration process to newer technologies due to the risk and cost such process entails. On the one hand, the risk is related to the lack of a clear insight of the system, since the documentation concerning both business policies and source code is generally poor and outdated. Furthermore, such organizations have often to face a loss of technical/application knowledge over time due to the departure of original developers. All these factors do not guarantee the organizations to have all their business logic replicated and working in the new system. In particular, this is vital in critical areas of business like

economic infrastructure, banks, insurance companies, etc. where serious financial and legal consequences can result from a system failure.

On the other hand, the cost of migrating legacy systems is generally prohibitive. According to Tactical Strategy Group, the cost of replacing a single line of legacy code has been estimated at approximately twenty-five dollars [1]. This cost is motivated by the fact that since the programs embedded in old systems are generally tailored to a specific organization, ad-hoc migration processes must be set up.

Many information technology companies are funding research to provide a migration path for their customer legacy applications into modern infrastructures. Their ultimate goal is to extract and modernize the embedded business logic, while preserving the rest of the infrastructure in place. Unfortunately, this entails complexities and challenges that concern the identification and extraction of business logic in the source code. Such challenges are the focus of this thesis.

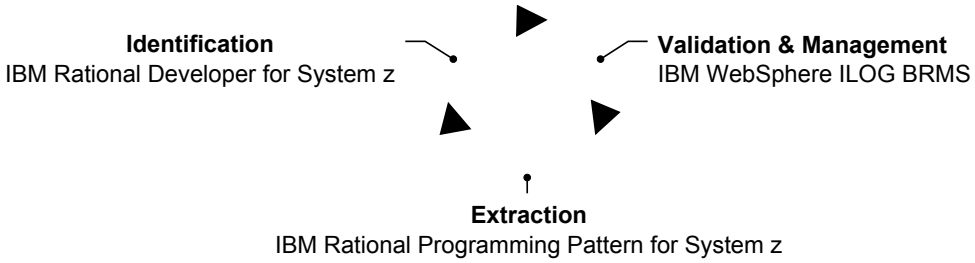

Figure 2.1: IBM system modernization project

IBM is one of the companies that is deeply interested in these challenges, since many of its customers still rely on legacy systems. Business logic extraction is an integral part of an IBM system modernization project initiative (Fig. 2.1). It consists in bridging the features of three different products to identify, maintain and manage business rules for older and newer technologies. In particular, IBM Rational Developer and IBM Rational Programming Pattern for System z are dedicated to identify and extract the business logic, while IBM WebSphere ILOG BRMS is focused on validating, operationalized and managing such logic. Finally, such validation can be used in turn to drive the redefinitions of business rules within the legacy system. As a consequence, this thesis has been funded by IBM in the context of an agreement with EMN (École des Mines de Nantes). IBM has provided expertise, use cases and tools.

2.2 Problem description

The business logic embedded in a system is composed by business rules [1], that are defined as relevant actions or procedures defining or constraining some precise

1. http://www.csis.ul.ie/cobol/course/COBOLIntro.htm

aspect of the business. Such rules represent the implementation of the organization policies.

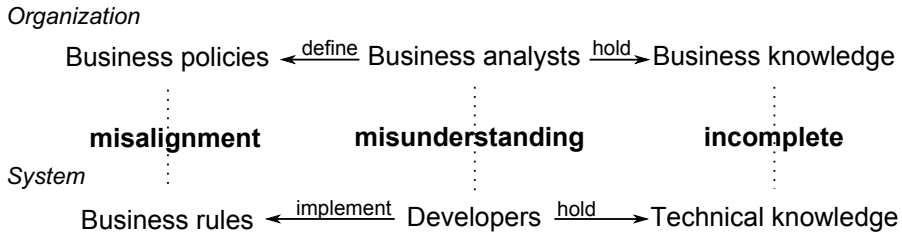

Figure 2.2: Issues in information systems

Locating, understanding and modifying the source code that implements the business rules in a legacy system is a challenging, error-prone and time-consuming task, because of different reasons (Fig. 2.2). In particular, locating business policies in a technical context is not trivial, since the rules are mixed with the constructs and aspects proper of the programming language employed. Therefore, technical skills are needed to isolate these rules.

These are not the only skills required to understand business policies in the code. In particular, two kinds of knowledge are needed, respectively the business and the technical knowledge. The former defines what the business rules enforce, while the latter specifies how such rules are implemented. Unfortunately, in organizations these kinds of knowledge are held by two different experts, business analysts/domain experts and developers. As a consequence, this separation often causes misunderstanding between what the system does and what the business policies define.

Another source of complexity is related to the position of business logic in the system. In particular, business rules at code-level are spread within the components and programs of the system. As a consequence, when aligning the rules to the corresponding business policies, development teams have to be able to identify the exact change that is required for the business without affecting other processes embedded in the system. This task is particularly complex for large systems due to the amount of lines of code to analyse. In addition, the knowledge of the system is often incomplete due to the departure of original developers and a limited documentation that does not reflect the current implementation. Therefore, duplicated rules and growing misalignment between what is defined in the business policies and what is currently implemented in the system are often introduced over time.

Misunderstood, misaligned and incomplete information slow down the adaptation/modification of the system to new requirements settled in the organization policies and threaten the consistency and coherency of the organization business.

2.3 State of the art

The process to mine business rules is called Business Rule Extraction (BREX). It is used to give a specific understanding of a system with respect to the business an organization runs [2]. As a consequence, it is part of the software comprehension domain [3]. Such domain is the pillar of several software engineering activities, such as maintenance, evolution and reverse engineering of software. The relationships between BREX and these software activities are shown in Fig. 2.3.

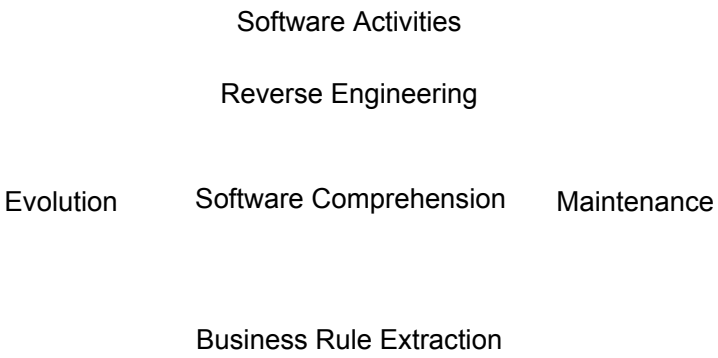

Figure 2.3: Business rule extraction process and software activities

BREX processes can be applied to different parts of the system. In this thesis, we focus on the analysis of its behavioral and structural part, where the former is meant to perform calculations and makes logical decisions to transform system's inputs into outputs, while the latter concerns the management of the storage and retrieval of data (i.e., database).

Each part of the system requires a different BREX process, according to the kinds of operation it deals with (i.e., manipulation and storage/retrieval). The BREX process for the behavioral part (see Sect. 7.1) is focused on program understanding and program slicing techniques. Such techniques are applied on the structures of the programming language implementing the system. Their goal is to discover and represent the variables and the chunks of source code relevant for the business. Therefore and according to these techniques, BREX processes for the behavioral part of the system are mainly composed by three steps ([2], [4], [**?**], [6]) that aim at:

- identifying in a system the variables that represent business terms,
- retrieving the pieces of code related to these variables,
- representing/storing the identified pieces of code.

The first step contains heuristics to ease the identification of business variables; the second step is based on program analysis techniques such as control and data flow anaylsis as well as slicing operations; while the last step extracts the logic identified to understandable formats. These output formats can be the collection of

the chunks of code obtained or higher abstraction level representations not related with the specificities of the programming language employed.

On the other hand, the BREX process for the structural part (see Sect. 7.2) aims at retrieving the business rules located in the database structures such as tables, stored procedures and triggers. It is based on the process concerning the reverse engineering of databases [7] that consists of:

- extracting and representing the embedded data definitions into a higher level representation,
- extracting the database constraints.

The first step defines rules to map database schemas to conceptual models, that are higher level representations with respect to the database structure definitions. The second step identifies the integrity constraints embedded in database and it expresses such constraints in a format that fits with the language used to define the conceptual model.

In this thesis, we present solutions to extract business rules from the behavioral part of systems implemented in Java and COBOL, and from the structural part of systems that rely on relational databases. We focus our analysis on Java, COBOL and relational databases, since most of the systems that rely on these technologies have been maintained and evolved for long time and the corresponding business rules might have become outdated over time.

The solutions proposed are based on Model Driven Engineering (MDE [8]). MDE techniques offer an abstract homogeneous representation of the system, avoiding technological issues with respect to the programming language employed for that system. In addition, MDE solutions provide generic and modular approaches adaptable to different languages. Finally, when representing a system as a model we can benefit from the plethora of MDE tools available on the market.

In the following section, the background of this thesis is introduced.

2.4 Background

We aim at extracting business logic expressed as a set of rules out of legacy systems using Model Driven Engineering (MDE) and software comprehension techniques. Therefore, business rules, MDE and software comprehension are part of the background of this thesis.

2.4.1 Software comprehension

Software comprehension is one of the core domain of software engineering. It is required to maintain, reuse, migrate, reengineer or enhances software systems [29].

It is based on finding concepts of the real world and how they are related each other within the system.

In the following, we discuss software comprehension with respect to the behavioral and structural part of systems.

System behavioral part

Any given source code embeds always a certain logic, that represents the behavior shaped by developers for that code. This logic can be expressed in different ways according to the constructs and specificities offered by the programming languages available on the market. Despite the differences that exist between these languages, common elements, that contain useful information to help understanding the code, can be identified among them. We focus on data structures, statements and functions (in a global meaning).

A data structure is a location used to keep relevant data. It represents a concept of the real world that a system handles, therefore it is the most important construct to identify business rules in the code. A data structure is supposed to be identified by a meaningful name aligned to the information stored in it. References of data structures are contained in statements as well as they can be input parameters of functions.

A statement is used to perform operations on data structures (i.e., read, write, modify) and in addition, depending on its type (e.g., conditional/loop statements), it can contain other statements. Implicitly, it defines how data structures are related together.

Finally, functions group statements logically related with respect to a given task. They are generally defined by a meaningful name/label.

Understanding how concepts of the real world are related together within a system is a cognitive process that is performed by developers and it employs much of their time. In particular, it is a time-consuming activity, since developers have to navigate the code in order to gather relevant concepts and their relationships expressed with the specificities of the programming language employed. In our case, such relevant information is composed by data structures, statements and functions related to the business logic.

Several studies have been carried out to understand how the source code is analysed in order to retrieve specific information (e.g., functionalities, processes, business logic). In [30], the authors analyse how developers seek, relate and collect relevant information during maintenance tasks on an unknown program. The authors discover three main phases that developers participating in the study used to perform. In particular, the first phase focuses on searching hints (e.g., meaningful data structures or function identifiers, etc.) in the code. In the second phase, such

hints are used as starting point to navigate the source code in order to find relevant information. Finally, the last phase collects and stores the information found in the previous steps.

The program understanding activities described in [30] can be automated using program slicing techniques. The use of such techniques to ease program understanding activities has been largely studied in the past (e.g., [31], [32]).

Program slicing [33] consists in isolating/collecting, according to a criterion, the chunks of code that affect directly or indirectly the calculation of a defined variable (i.e., or set of variables).

The program comprehension steps described in [30] can be mapped to the program slicing steps (Fig. 2.4). In particular, the first step in [30] is equivalent to select a slicing criterion for slicing the program, the second step represents the slicing operation, and finally the last step is equal to collect the sliced chunks of code. While selecting a slicing criterion is generally a manual or semi-automatic activity, since it is up to the user to choose an entry point (i.e., statement containing a given variable to analyse) to slice the program; the other two steps, respectively the slicing operation and collection step, can be automated.

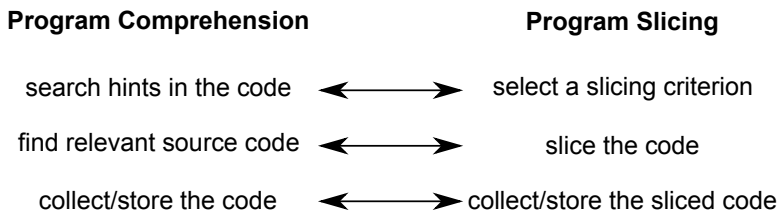

Figure 2.4: Program understanding and program slicing

Program slicing can be performed in different ways [34], depending on 1) how to traverse and 2) how to analyse the program. In the first case, a program can be traversed, according to its Control Flow Graph (CFG [35]), backwardly or forwardly for a given entry point, which is a statement containing a relevant variable identified by the slicing criterion. In the second case, the analysis of a program can be performed using different approaches that are generally based on either static or dynamic analysis.

In particular, static slicing is used to identify statements in the program that potentially contribute to the computation of a given variable or set of variables, without focusing on the possible values that a particular variable could hold at runtime. Such kind of slicing is helpful to gain a general understanding of a program with respect a specific behavior/functionality. On the other hand, dynamic slicing [36] generates slices for particular program input and it allows to understand how the input variable values change during the program execution.

A complementary approach to static and dynamic analysis is described in [37], where the authors introduce the notion of removable blocks, described as the smallest component of a program that can be removed during the slice computation without violating the syntactical correctness of that program. A block can be dropped if its removal does not interfere with the flow execution (i.e., either static or dynamic) concerning the variables related to a given slicing criterion.

System structural part

The concepts of the real world and their relations in the structural part of a system are represented by means of the database schema. Such schema defines the organization of data and a set of integrity constraints that ensure the consistency of the information stored in the database.

A database schema consists of several elements such as tables, table relations, views, data types, indexes and constraints expressed in operational (i.e., triggers and stored procedures) and declarative (i.e., table constraints and assertions) ways. In addition, tables are composed by columns, where each column describes a property (i.e., name and data type) of a given table.

Tables and columns represent real-world concepts [2] and, as consequence, they belong to the group of first class citizens in database schemas. In addition, views can be considered part of such group, since they are derived from tables. On the other hand, table relations and constraints are used to define how real-world concepts are related each other.

The identification and extraction of such database schema elements is the focus of the software comprehension in database. In particular, this is achieved by database reverse engineering processes that translate tables, columns and their data types as well as relations and constraints into an equivalent conceptual model (Fig. 6.8).

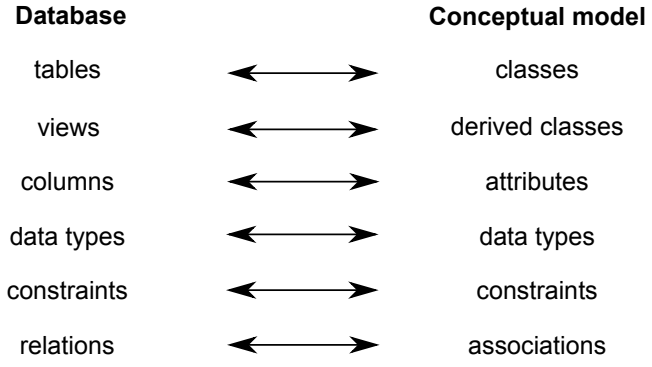

Figure 2.5: Database schema to conceptual model

2. Data types defined by the user can represent real-world concepts.

The obtained model has a higher abstraction level representation with respect to the database schema, since it hides the specificities of the database technology. It is composed by classes, class associations and constraints. In addition, each class is defined by attributes that have a name and a data type. On the one hand, tables and views are mapped as classes and derived classes in the conceptual model. Each column within a table is mapped to the attributes of the class that corresponds to that table. In addition, since a column is defined according to a given data type allowed by the database, the corresponding attribute is defined by an equivalent data type.

On the other hand, relations between tables are mapped as associations between the corresponding classes in the conceptual model, and database constraints are converted into equivalent constraints applied over the model. In particular, database constraints are expressed in different ways using Structured Query Language (SQL [38]) queries, assertions, user type definitions as well as they can embedded in stored procedures and triggers.

Database constraints can be divided in two groups: declarative and operational constraints [39]. Declarative constraints are written in SQL and applied on table and table column declarations in order to avoid duplicated or null values (i.e., primary key, not null and unique constraints), to define relations between tables (i.e., foreign keys) or to limit the value range that can be placed in a column (i.e., check constraints). On the other hand, operational constraints are defined in assertions, user type definitions, stored procedures and triggers and they are used to code more complex constraints with respect to the declarative ones.

Finally, database constraints are mapped to semantic equivalent constraints applied on the conceptual model [40]. These constraints are defined by constraint languages like Object Constraint Language (OCL [41]) or Semantics of Business Vocabulary and Business Rules (SBVR [42]) that are more suitable to express business rules at conceptual level.

2.4.2 Business rules

Business rule is a concept that is broadly known and employed nowadays. There is no a single definition, but several ones, since practitioners and organizations tend to assign to such concept specific denotations. Below, we report some of those definitions.

According to E. Gottesdiener [16], business rules are the core of functional requirements, they provide the knowledge behind any and every business structure or process. For I. Baxter and S. Hendryx [17], they are defined as an obligation that covers conduct, action, practice, or procedure, or a necessity that is intended as a definitional criterion. On the contrary, the Business Rules Group [1] describes the business rule as a statement that defines or constraints some aspect of the business.

It is intended to assert business structure or to control or influence the behavior of the business. Finally, IBM specifies the business rules as anything that captures and implements business policies and practices [18]. They are expressed using a formalized vocabulary and a series of if-then statements [19].

In addition to these definitions, it is important to highlight that within an information system, business rules can have different connotations depending on whether the perspective is data, object, procedural or expert system-oriented [16]. In other words, we can say that business rules are key elements both for companies and their systems. On the one hand, they are used to depict policies independently from designing paradigms and technical platforms. On the other hand, they drive the behavior of the system and, when implemented in a system, they are mixed with the specificities of the programming language the system employs.

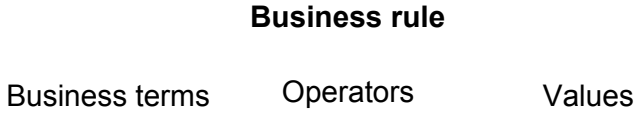

Figure 2.6: Business rule constructs

Business rules are composed by three constructs: business terms, operators and values [19]. Such constructs are shown in Fig. 2.6. Business terms represent the objects that are part of, or are affected by, business rules. Operators allow to compare the properties or characteristics of different business terms and include arithmetic, logical, etc. operators. Finally, values can be numbers (or arithmetic expressions), text values or predefined business terms.

```
If purchases of customer > 5000
then status of customer = "Gold"
```

Figure 2.7: Example of a business rule

In Fig. 2.7 an example of business rule is shown. The rule states that a customer becomes a *Gold* customer as soon as his cumulative purchases have overtaken 5000 dollars. *Customer*, *purchases*, *status* and *Gold* are business terms, while the operators are expressed by the logical operator *greater than* and the assignation. Finally, the values are represented by the term *Gold* and the numeric value 5000.

Several kinds of business rule exist, they are classified in four categories [1]:

– **Term definitions.** They specify the vocabulary for expressing business rules. In particular, a business term can be a noun, a phrase or sentence that defines concepts of the business. For instance, given a shopping system, nouns like *shopping cart*, *customer*, *products* are business terms for that system.

Finally business terms become data structures, database tables, attributes and columns when implemented in an information system.

- **Facts.** They describe relationships/associations between business terms and in particular, they allow business terms to have assigned roles in the business. For example, the sentence: "each customer has a shopping cart" defines the relation *has* between a *customer* and his *shopping cart*.
  Facts in information systems can be found either both in the behavioral and structural part. In the behavioral part they are defined at statement-level (i.e., assignments), in the definition of data structures as relations between the data structures and the embedded properties or as relations between data structures (e.g., inheritance, subtypes, etc.). On the other hand, in the structural part, they appear as associations/relations between database tables and as relations table-table columns.
- **Constraints or structural rules.** They confine the organization behaviors defining what is allowed and forbidden in the organization business. Expressions such as *must*, *cannot*, etc. identify generally constraint rules in business policies. Examples as "a customer must have a bank account assigned" or "any customer cannot be younger than 18" represent constraint rules.
  Such constraints are generally expressed on the data/class model within the structural part of a system. In particular, they are defined as integrity constructs that protect the consistency of data with respect to the business. Constraint rules can be found in the behavioral part as well. They can be implemented as exception handlers in the source code.
- **Derivations or behavioral rules.** They define how business knowledge (i.e., terms and facts) is transformed into other knowledge. Examples as *if a customer has more than fifty purchases, he is a Silver customer* or *a purchase bill is the sum of all product prices selected by the customer* represent derivation rules.
  Derivation rules are generally coded in the behavioral part of the system and they appear as statement containing mathematical operators. However, they can be found within the system's structural part implemented as stored procedures.

Business rules in the behavioral part

This section gives a definition of business rules at code-level and how the four categories, previously described, are mapped on the constructs of a generic programming language (i.e., data structures, statements, functions).

The definitions of business rule previously given cannot be used directly to identify rules at code-level, since the business rules expressed at high abstraction level

must be implemented according to the constructs a programming language offers. Unfortunately, this implementation causes the rule to be spread into the source code.

At code level, a business rule is defined as a generic function *F* that takes several variables as input and returns a variable as output [2]. *F* is represented by a set of conditional and related statements that use the variables in input *I* to calculate the value of the outcome variable *O*.

$$F(I_1, I_2, I_3, ...) = O$$

Figure 2.8: Business rule at code-level

As an example, we consider a business process of hiring a new worker within a company. This process is composed by several steps that take into account a set of characteristics (e.g., technical skills, age, experience, etc.) for each candidate. The final result of the recruitment process depends on the sum/conjunction of all these steps. As a consequence, the corresponding business rule can be seen as a function that takes as input a set of variables that represent the characteristics of a given candidate and returns a variable that contains the acceptance or rejection for that candidate.

Business rules at code-level fall under the four categories previously described, but a mapping (Fig. 2.9) must be defined between such categories and the constructs of a generic programming language (i.e., data structures, statements and functions).

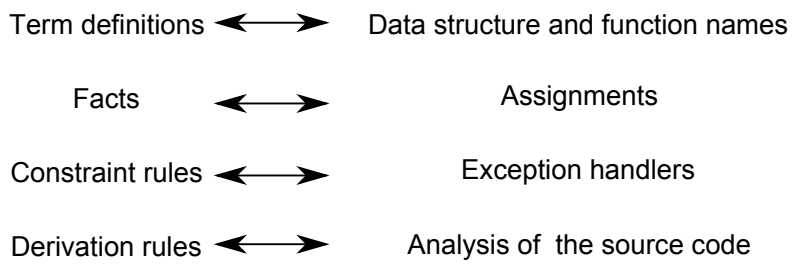

Figure 2.9: Business rule categories at code-level

– **Term definitions.** Definitions of business terms are scattered in the code, mostly in data structures and function names. Frequently, these names have to be processed to extract the corresponding business terms (e.g., a variable name *cust* must be transformed in the business term *customer*). In addition, such business terms can be found as well in comments and system's documentation; although the information contained might not be aligned with the corresponding source code.
– **Facts.** They can be derived from the analysis of the assignments in the source code. For example, for a system handling the monthly interest on cars' loans we could find a statement that relate the variables corresponding to car's price and interest's rate (i.e., *MonthlyInterest = priceCar * interestRate * (1/12)*).

Finally, facts can be highlighted applying a data-flow analysis [20]. The result of such analysis is a graph where the nodes are the statements in the program and the edges define where the variables are declared and modified. Such edges allow to determine the relations between variables.

- **Constraints or structural rules.** In the code, they are generally implemented using exception handlers related to return values of functions. According to these returning values, they change the normal flow of the code execution if a specified error condition occurs.
- **Derivations or behavioral rules.** They are represented by statements dealing with a given variable (or set of variables) in the same execution path. As a consequence, the identification of derivation rules is a complex activity, since it involves the analysis of most of the source code and the related Control Flow Graph. It relies on the recovery of the business terms (i.e., business variables) and facts (i.e., relevant statements).
  Finally, since the system's behavioral part performs calculations and makes logical decision, it is supposed to contain a large number of derivation rules. Therefore, discovering derivation rules is the most important step in a BREX process for the behavioral part of the system.

Business rules in the structural part

This section provides a definition of business rules for the structural part of the system and how business terms, facts, constraints and derivation rules are mapped on the structures of database implementations.

In a database, a business rule is represented by means of database integrity constraints. According to [21], they are divided in three categories, namely inherent, implicit and explicit. Inherent constraints concern the relations between tables and the contained columns. Implicit constraints refer to primary and foreign keys used respectively to identify a record in a table and to define relations between tables. Finally, explicit constraints involve column and table constraints (e.g., check, not null, unique, etc.) as well as create assertions, define user types (i.e., create domain statement), stored procedures and triggers.

While the inherent and implicit constraints specify business terms and facts; the explicit constraints are the rules that define aspects of the business and therefore, they are the focus of the BREX process in database.

A mapping (Fig. 2.10) is defined between the different categories of business rules and the database integrity constraints. In the following this mapping is depicted.

- **Term definitions.** Definitions of business terms are derived from the database table and view definitions as well as from the columns composing such struc-

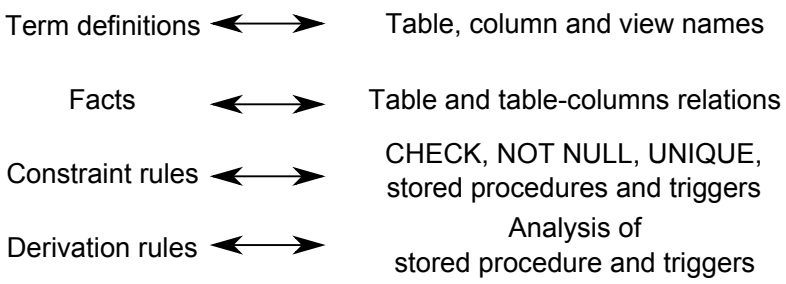

Figure 2.10: Business rule categories in databases

tures. In particular, the business terms are extracted from the name of tables, views and columns. Since those names often describe directly business terms, they do not need to be processed to extract the corresponding business term.

– **Facts.** They are defined as relations between tables as well as intrinsic relations between tables and their nested columns.
– **Constraints or structural rules.** They are derived from table and table column declarations as well as from stored procedures, triggers, user-defined type definitions and assertions.
  In particular, business rules are extracted from *CHECK*, *NOT NULL* and *UNIQUE* database constraints and from the exception handlers embedded in stored procedures and triggers. Rules are extracted as well from the values that a user-defined type can have and assertions [3].
  Finally, since databases define the type of data and values that are allowed or forbidden, the identification of constraints is the most important step in a BREX process for the structural part of the system.
– **Derivations.** Derivations rules can be extracted from the code analysis of stored procedures and triggers; although such database constructs are not often used to implement derivation rules in database due to different reasons.
  In particular, the choice of implementing derivation rules could reduce the portability of the database (e.g., the database could not be shared among systems that do not rely on the same derivation rules) or the evolution of the information system (e.g., the system would be tied to a particular database vendor dialect). In addition, the database is not suitable to execute derivation rules with respect to performance that the same logic can have if executed in the system's behavioral part.

2.4.3 Model Driven Engineering

MDE [8] is a software engineering discipline that considers models as first-class citizens for both forward and reverse engineering processes. This different way of

3. Note that the major databases do not support assertions

using models is opposed to the previous approaches, that limited models to a passive role (mostly documentation) during software engineering activities.

Adoption of MDE has the potential to bring many benefits to such activities. In particular, it improves the maintainability and quality of systems [9] thanks to its features that consist in a higher abstraction level representation with respect to the source code and the automation of repetitive activities in software development processes as much as possible. The former relies on the use of models, while the latter on model manipulations. In the following, we describe these two features.

The basic assumption of MDE is that models and not the classical programming code is the right representation level for managing all artifacts within a software engineering process; therefore, models are considered as the unifying concept in MDE. Models are defined according to a three-level architecture shown in Fig. 2.11. Such architecture is composed by model, metamodel and metametamodel.

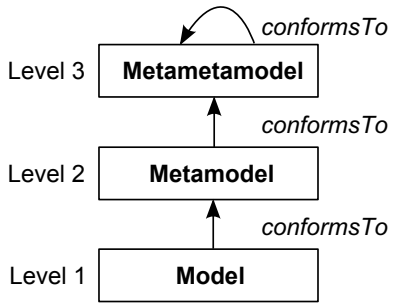

Figure 2.11: Three-level architecture in MDE

A model is a partial representation of a system that captures some of its characteristics (e.g., documentation, source code, components, etc.). The combination of different models related to the same system can be used to derive a global view of that system. Each of these models represent entities composing software artifacts/concepts in the real world. Such concepts and their associations (i.e., semantics) are defined in the second modeling level, called metamodel.

A metamodel is related to a model according to a relation of conformance. Such relation is equivalent to the program - grammar relation for a given programming language; such that, programs written in one language must conform to the syntax rules of that language as well as models defined according to a metamodel must conform to the rules embedded in that metamodel.

Metamodels are in turn expressed by means of the third modeling level called metametamodel. Similar to the model/metamodel relationship, a relation of conformance is defined between metamodels and metametamodels; such that a metamodel is defined using concepts and associations of a given metametamodel. In addition, this relation is equivalent to the relation between the grammar of a given programming language and a metasyntax/language to define grammars (e.g., EBNF: Extended Backus-Naur Form [43]).

This three-level representation, also known as modelware, is not so different from the grammarware (i.e., the technical space where a language is defined according to a grammar) in terms of basic definition and infrastructure [24]. Such equivalence is shown in Fig. 2.12. Therefore, the metasyntax is conceptually equivalent to a metametamodel, the syntax of a given language is at the same level of a metamodel, and a program (i.e., an instance of a grammar) is analogous to a model.

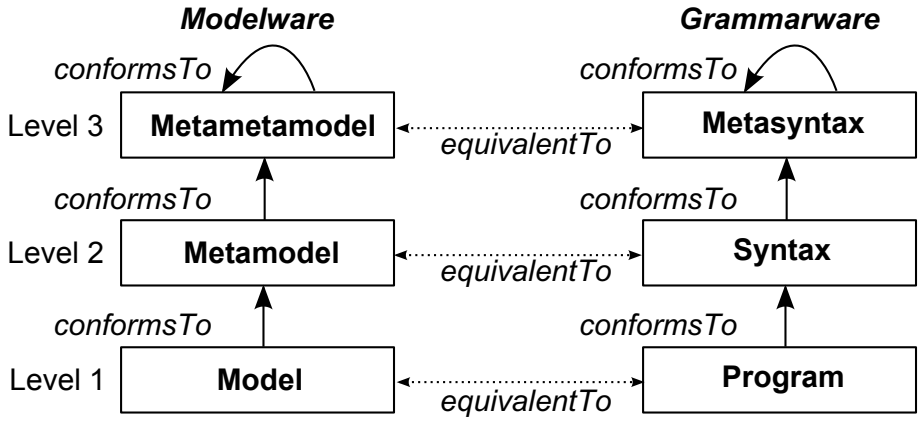

Figure 2.12: Modelware and grammarware

Finally, models, metamodels and metametamodels may be implemented according to different standards. For instance, the Object Management Group (OMG) proposes a standard metametamodel called Meta Object Facility (MOF) and different standard metamodels (UML: Unified Modeling Language [10], KDM: Knowledge Discovery Metamodel [11], etc.).

The second feature of MDE is represented by model transformations (shown in Fig.2.13) that, taking one or more models as input, generate one or more models as output according to mappings defined over the input and output metamodels. In particular, given a *source model*, the related *source metamodel* and a *target metamodel*; a model transformation is able to generate the *target model* that conforms to the *target metamodel*.

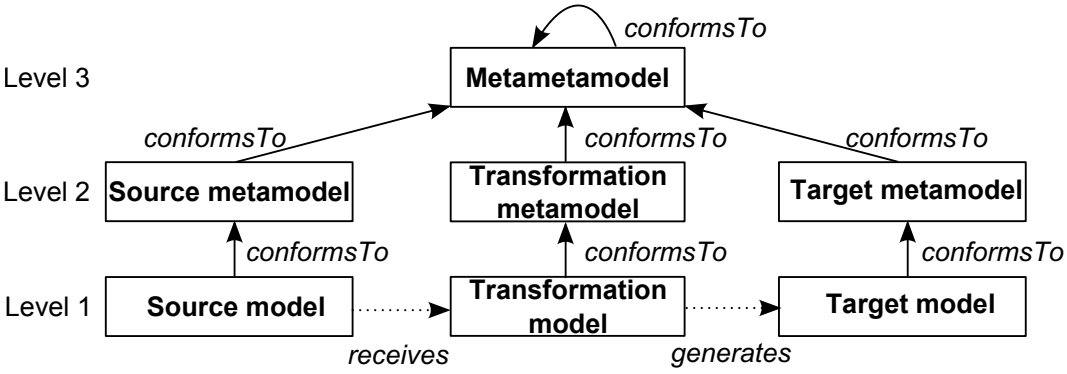

Figure 2.13: Model transformation

Depending on the input and output metamodels two different kinds of transfor-

mation exist. If the source and target metamodels are identical, the transformation is called endogenous; while if the metamodels are different the transformation is called exogenous.

Furthermore, model transformations are aligned to the three-level architecture in MDE, therefore they are represented as models. A model transformation conforms to its corresponding metamodel, that contains the means to specify mappings between model elements. In turn, such metamodel is defined according to the semantics of the metametamodel. Since a model transformation is itself a model, a model transformation can take as input one or more model transformations and producing other model transformation as output. Such kind of transformations are called Higher Order Transformation (HOT [44]).

In addition, model transformations record information about how a target model element is related to the corresponding source element that originated it (i.e., traceability [12]). The relation between source and target elements is called trace and it is useful to understand and track software artifacts within an MDE process (composed by a single transformation or a transformation chain). A trace can be represented by either a simple link to a given model element or a more complex encodings (e.g., identifiers, etc.).

Traceability can be modeled according to the MDE three-level representation [13]. Therefore, traceability information is stored within a model that conforms to a metamodel[4]. This metamodel, shown in Fig. 2.14, is composed by a class (i.e., *TraceLink*), that owns the links of the source and target elements (i.e., *sourceElements* and *targetElements*) involved in a given mapping (i.e., transformation rule). Such *Tracelink* is identified by a name, that is the name of the transformation rule originating it.

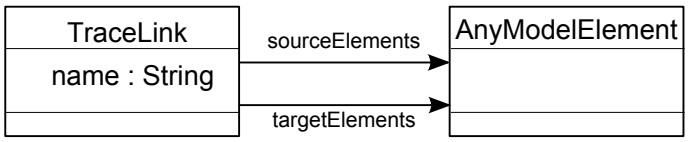

Figure 2.14: Traceability metamodel

Finally, several transformation languages are available on the market. For instance, ATL Transformation language [14] that is currently one of the most used transformation languages and Query/View/Transformation [15] that is the standard language proposed by OMG.

4. Note that the traceability information can be part of other metamodels

2.5 Objectives and contributions

We aim at providing a reverse engineering process for extracting business rules embedded within information systems. The main goal is the software comprehension of legacy systems to ease the corresponding modernization and maintenance activities. These activities are a key (economical) area in software engineering.

MDE with its methods and techniques are able to overcome misunderstandings between business analysts and developers, misalignments between business policies and rules as well as the incomplete documentation (both for business and technical knowledge) of information systems. Such new technology is mature enough to provide a complete comprehension of systems with respect to the business, ensuring in this way their correct evolution.

In the following we present the main contributions of this thesis.

2.5.1 Generic and modular framework

We have defined a conceptual framework that is applicable to different programming languages. In particular, we have identified the common steps that compose a BREX process. In addition, relying on MDE, we have designed our framework in a modular way. Therefore, each step is independent and it is related with the other steps only by its input and output models. As a consequence, the framework is suitable for different typologies of user, since the BREX process can be stopped at any moment and the user can select the level of output (technical or business-oriented) that best fits his needs.

2.5.2 Model-based approach

In this thesis, the problem of extracting business rule from information systems is studied as an instance of MDE. In particular, MDE performs an effective job with respect to the analysis of programming languages and reverse engineering processes such as BREXs.

The proposed BREX process is based on the use of MDE techniques. The higher-level abstractions used in MDE facilitate the building of effective and reusable reverse engineering solutions able to provide genericity, reusability, extensibility and evolution capabilities for legacy systems based on a clear decoupling of the represented information (as models) and the different steps of the reverse engineering process. Finally, the business rules identified from such systems are represented and manipulated using appropriate languages, to facilitate its comprehension.

2.5.3 Traceability and granularity of the extracted business rules

Previous methodologies implemented traceability with intrusive solutions (for instance by code instrumentation [45]) or they did not highlight the importance of connecting the extracted rules with the corresponding chunks of source code involved. This is crucial when providing a smooth migration path from older to newer technologies, such that the business rules can be extracted and modified within the system, that is preserved in place.

Using MDE, our BREX process takes as input a detailed representation of the code (a model) and not the source code, that is left unmodified. The mapping between this new input and the source code is guaranteed by MDE. Traceability is implemented on this higher abstraction input relating the latter with the different artifacts generated during the reverse engineering process. So in a non-intrusive way, we know exactly which input elements (i.e., from where we can retrieve the corresponding part of source code) participate to the identified rules.

In addition, in the previous works the granularity of the extracted business rules is not considered or not well emphasized. A business rule, at code level, represents a set of related statements with respect to a given variable. These statements are part of the same execution path. The granularity of an obtained rule is used to separate in the execution path the statements that contain a given business variable from those ones that define the context (i.e., a set of conditional statements) for that rule.

2.5.4 Solutions for Java, COBOL and relational databases

We have designed BREX processes for the behavioral part of systems employing Java and COBOL, as well as for the structural part specified as a relational database. Such technologies have been chosen since they have largely used for at least twenty years [5]. As a consequence, systems based on Java, COBOL or relational database are old enough to motivate a BREX analysis on them.

With respect to the system's behavioral part, we are able to extract derivation rules and business terms from Java and COBOL languages. On the other hand, the types of business rules extracted from the system's structural part are business terms and rule constraints. The former are derived from the tables and views of the database schema; while the latter are obtained from the database constraints embedded in table definitions and triggers.

5. Java has been released in 1995, COBOL 1959 and relational database in 1970

2.6 Thesis structure

This thesis is structured as follows. Chapter 3 describes our BREX conceptual framework. Chapter 4, Chapter 5 and Chapter 6 depict its implementation for COBOL-based and Java-based systems as well as for relational databases. Chapter 7 discusses the related work with regard to our proposed framework. Finally, Chapter 8 presents the conclusion and further work.

The publications related to this thesis are listed at page 139.

3

Model-based framework for business rule extraction

This Chapter presents an overview of our BREX process. It describes the steps that compose such process and how they are adapted for dealing with the behavioral and structural parts of the system.

Our BREX process aims at identifying and extracting the business rules enforced in a system. In this thesis, a business rule is defined for the behavioral part of the system as a set of statements in the same execution path related to the same (business) variable(s). In particular, a business variable is a variable that carries a business meaning. We consider business variables, the variables used within mathematical operations, input/output commands, conditions of conditional statements and initialisation statements. According to our experience on the IBM use cases and to some previous works ([2] [4], [**?**], [6]), these heuristics are able to identify business rules buried in the code. Clearly, such heuristics are not complete, but the framework allows to add additional heuristics. On the contrary, for the structural part of the system, a business rule is a declarative constraint in the database schema ([22], [23]) or the condition that causes a trigger to be executed.

The BREX process implemented is performed in a MDE environment, hence a specific operation is needed to pass from the grammarware technological space defined by the programming language implementing the system to the modelware space, that provides a model representation of the code [24] (see Sect. 2.4.3). The business rules are derived by analysing this derived model representation. Such a model contains the static information of the system. In particular, for the behavioral part, it represents the abstract syntax tree of the source code with bindings between

named model elements (e.g., usages of variables are linked to the corresponding variable declaration and vice-versa, method invocations are linked to the method declaration and vice-versa, etc.). On the contrary, for the structural part of the system, the model representation contains the structural information of the database schema analysed (tables, columns, views, triggers, declarative constraints).

Finally, our BREX process is based on static analysis techniques applied to the model derived from the source code. As a consequence, for the behavioral part of the system, we are able to exclude from this model no-reachable code by performing a control flow graph analysis. On the contrary the identification of useless or dead code is out of the scope of this thesis.

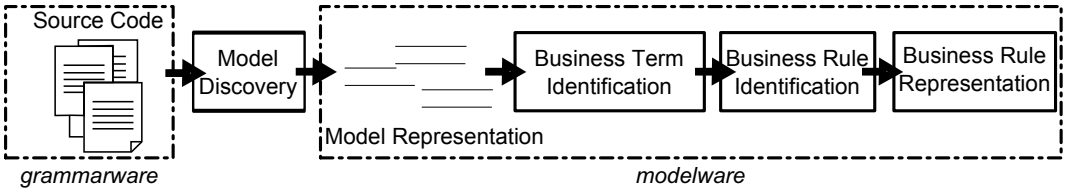

Figure 3.1: Business rule extraction framework

In Fig. 3.1, the steps that compose the framework implementing the BREX process are shown. They are respectively, Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation. Model Discovery takes as input the source code expressed in a given programming language and generates a Platform Specific Model (PSM) that has a one-to-one correspondence with the code, so that there is no information loss and all the code source elements are represented as part of the model. Finally, the obtained model is then manipulated in the next steps composing the BREX process.

Business Term Identification focuses on discovering the business terms embedded in the system that are then used to drive the Business Rule Identification step. Depending on the system's part analysed (i.e., behavioral or structural) different heuristics are defined for such steps. Finally, the Business Rule Representation step is in charge of visualizing the identified rules by means of textual and graphical artifacts.

Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation are described in the remaining part of this Chapter.

3.1 Model discovery

Model Discovery is used to present the heterogeneity embedded in a system by means of uniform and homogeneous representations (i.e., models). This is specially worthwhile when analysing systems that rely on different technologies. The

ultimate goal of the Model Discovery step is to derive one or several models from a given system, depending on the needed viewpoints, and then to work directly on these models. Each model conforms to a given metamodel expressing the chosen viewpoint such as source code, database schemas, etc.

In this thesis, we have relied on different model discovery tools (Fig. 3.2) to generate models from Java and COBOL source code as well as from relation database implementations. In particular, we have used MoDisco [25] for Java, IBM COBOL Application Model [1] for COBOL, and finally Xtext [**?**] to derive a model-based representation from database schemas and triggers.

Such models have a one-to-one correspondence with the source code. In particular, they represent the full Abstract Syntax Tree (AST [46]) of the code and contain the resolution of the bindings between identifiers (e.g., a variable definition with the corresponding usages of that variable, etc.) in the code.

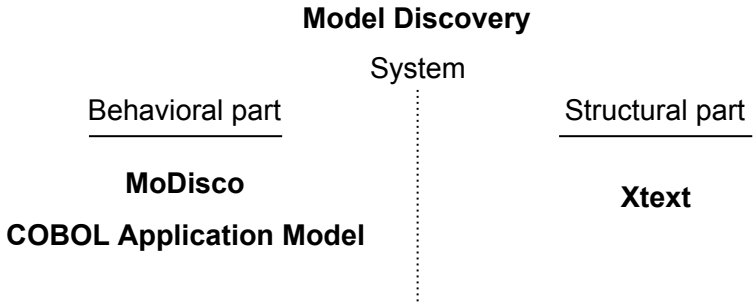

Figure 3.2: Model Discovery

3.2 Business Term Identification

Business Term Identification locates the business/domain concepts within the model of the system. Since the business terms are represented in different ways depending on the system's part analysed (Fig. 3.3), we have developed several heuristics able to identify such terms.

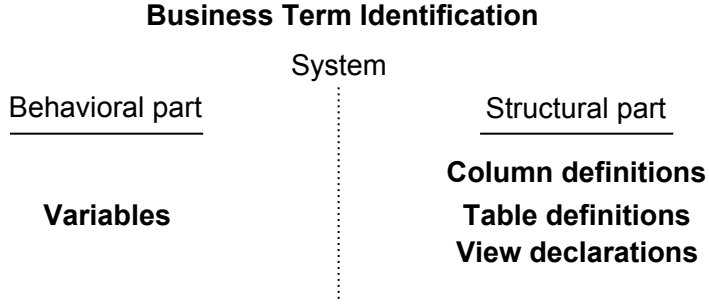

Figure 3.3: Business Term Identification

1. http://tinyurl.com/IBMCobolApplicationModel

Concerning the behavioral part of the system, for Java and COBOL source code, we have focused our analysis on the variables used in the code in order to identify those ones related to business-relevant concepts. The heuristics implemented discover variables that appear together with mathematical operations, in conditions of conditional statements, in input/output statements and in initialisation statements. We are aware of the fact that such heuristics are not complete to locate all variables hinting at business terms due to the complexity that a system can have. However, other heuristics can be easily plugged to our framework thanks to the modularity provided by MDE.

On the contrary, with respect to the structural part of the system, for relational databases, we have focused our analysis on tables, views and column declarations composing the database schema, since such structures have often a one-to-one correspondence with business terms.

Finally, the business terms identified in the behavioral and structural parts of the system are connected to the corresponding code structures using MDE traceability (see Sect. 2.4.3).

3.3 Business Rule Identification

Business Rule Identification provides the means to identify the business rules related to one or several business terms. In addition, it is used to specify an internal representation of the business rules identified, and it offers traceability support to connect the discovered business rule to the corresponding source code.

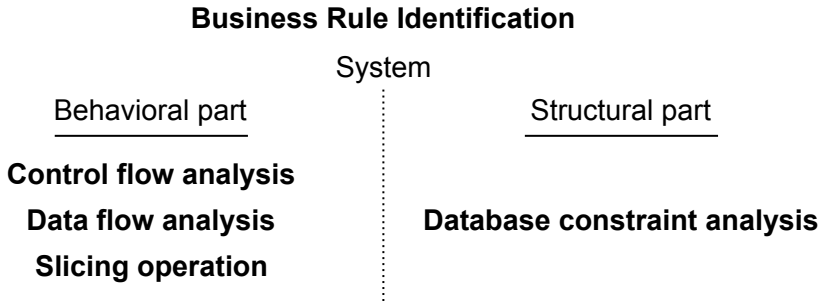

Figure 3.4: Business Rule Identification

This step is composed by specific operations able to deal with the different paradigms employed in the behavioral and structural part of systems (Fig. 3.4). In particular, Business Rule Identification for the behavioral part is based on control and data flow analysis as well as the slicing operation. Such operations are implemented on the model representing the source code. The output of this step is a model that contains the information about the part of the code composing the rules for one or more variables.

On the other hand, the Business Rule Identification for the system structural part relies on the analysis of declarative and operational database constraints.

All aforementioned operations are described in the remaining part of this section.

3.3.1 Control Flow Analyis

Control flow analysis is used to discover the Control Flow Graph (CFG [27]) of a program using static code analysis techniques. Such CFG contains the information of all paths that might be traversed through a program during its execution. As a consequence, this analysis is able to identify no reachable code in the source code.

In a MDE context, this analysis is based on navigating the AST embedded within the model that represents a given program. In particular, the AST expresses the model information (e.g., model elements, relations, etc.) in a tree-like representation. Such representation is exploited according to the semantics of the programming language (i.e., Java or COBOL) to define heuristics, that are used to generate the CFG.

The input of the control flow analysis is the model that represents the application/program, while the output is the input model enriched with the CFG information. In particular, according to the heuristics defined, for each model element, a list of successor elements is created and attached to that element.

3.3.2 Data Flow Analyis

Data flow analysis is used to highlight relations between variables and it allows to identify business facts within a program. This steps consists in traversing the model that represents the CFG in order to connect the statements that affect/use the same variables.

In particular, in this thesis data flow analysis is used to create relations between each statement that refers to a certain variable with the next (i.e., or previous) statement that refers to the same variable. Therefore, each model element that represents a statement, stores a list of successors (i.e., or/and predecessor) that deal with the same variable.

The input of this step is the CFG model of the program; while the output is the same input plus the information concerning the data flow.

3.3.3 Slicing operation

The slicing operation is used to collect the parts of the source code related to a given business variable. These parts of code represent the business rule and its corresponding context. In particular, the former is composed by statements that

use the business variable, while the latter includes the conditions that trigger such statements. These conditions are found in the possible execution paths within the code.

We have implemented the slicing operation relying on the backward static slicing as well as on the removable block slicing (Sect. 2.4.1); so that for a given statement containing a relevant variable (i.e., entry point), the slicing operation navigates from that statement backward to the beginning of the program [2], selecting the statements that are related to that variable. In addition, the removable block slicing is used to discard the blocks of code that do not have relations with the selected variable.

The static analysis has been chosen for two reasons. First, the model discovery step generates for a given software artifact (e.g., simple source code files, multi-file programs, applications, etc.) a "snapshot" (i.e., model) representing the corresponding static information; therefore dynamic information are not stored in the discovered model. Second, since the discovery process is generally a time-consuming activity, the model that represents the source code is generated only once. As a consequence, in our opinion, the generation of models that simulate the dynamic behavior of information systems is not suitable for business rule extraction analysis.

On the other hand, the backward and removable block slicings have been chosen since they both benefit of the tree-structured representation of models. In particular, using such representation, we are able to skip the model elements that do not contain relevant information using their containment/hierarchical relations.

3.3.4 Database constraint analysis

Database constraint analysis focuses on identifying and extracting the constraints in databases. Such constraints are expressed in table definitions and triggers. In this thesis, we represent such constraints by means of OCL. They rely on a UML conceptual model derived from the database schema.

The analysis of declarative constraints extracts rules from database table definitions. It takes as input the model-based representation of the database schema and returns OCL semantic equivalent constraints for *PRIMARY KEYs* as well as *UNIQUE*, *NOT NULL* and *CHECK* SQL constructs.

On the other hand, the analysis of operational constraints extracts rules from triggers. They are analysed with respect to the exception handlers they embed [3], since such constructs are generally used to code violations of business policies.

2. Note that the beginning of a program depends on the programming language employed, for instance, in Java it can be the main method or an action attached to a button; while in COBOL it could be the first statement in the procedure division.

3. Note that this approach can be applied to stored procedures

Therefore the conditions that trigger exceptions are supposed to be business relevant and they are translated into business rules. This operation takes as input the models that correspond to the database triggers and returns the corresponding OCL constraints. Since triggers often embed SQL queries executed over the database, the extraction of such OCL constraints relies on mappings between SQL to OCL (Sect. 6.7.2), that concern SQL projections, joins, functions, group by and having clauses.

3.4 Business Rule Representation

Business Rule Representation, shown in Fig. 3.5, is the last step of our framework. Its goal is to generate comprehensible artifacts that describe the identified business rules. It is composed by two operations: Vocabulary Extraction and Visualization of rules. On the one hand, the Vocabulary Extraction step is used to provide verbalizations of the business-relevant structures (e.g., variables, database tables, columns, etc.) embedded in the system and identified in the Business Term Identification step. On the other hand, the Visualization step is used to ease the comprehension of the rules by providing either textual or graphical artifacts.

Finally, during each of such steps, traceability information are propagated to the new output models (i.e., vocabulary, text, graph), such that the latter contain information that relates the output model elements to the source code.

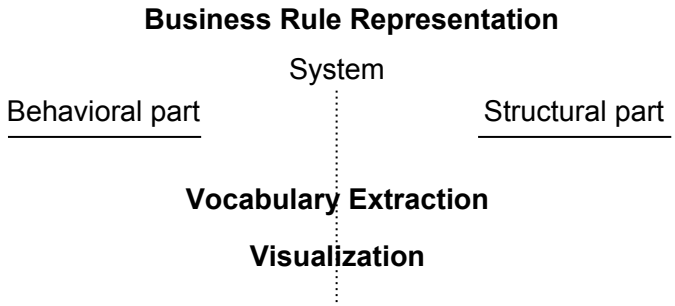

Figure 3.5: Business Rule Representation

3.4.1 Vocabulary Extraction

Vocabulary Extraction is an optional operation that aims at providing verbalizations/descriptions for the constructs of the programming language composing a business rules.

In the behavioral part of a system, such operation is used to generate a vocabulary made of pairs <*variable/function name, description*>, that can be optionally tuned by the user.

The vocabulary for Java programs is automatically generated. It contains a list of tuples composed by class, variable and method names and the corresponding de-

fault verbalizations that are based on the common way to assign names in Java (i.e., <*getAttribute(), "get Attribute"* >). On the other hand, the vocabulary concerning COBOL programs maps only variables over business terms. Since the identifiers of such variables are generally shorter with respect to the Java ones, human intervention is needed to complete the corresponding meanings.

On the other hand, in the system structural part, the Vocabulary Extraction step is based on deriving the business terms together with their corresponding relations embedded in the database implementation. In particular, the input of this step is the model that corresponds to the database schema (i.e., table declarations and views); while the output is a conceptual model defined by means of UML.

3.4.2 Visualization

Visualization provides external representations of the business rules identified. They can be represented by either text artifacts to focus on single rule behaviors or graph artifacts to analyse the relations among the obtained rules. Both text and graph representations externalize the internal format of the rules identified in the Business Rule Identification step.

In the behavioral part of the system, the Visualization step can use the information contained in the business vocabulary, if defined. The text representation is derived by model-to-text transformations, where the model represents the internal format of the rules and the text is the corresponding code-based or vocabulary-based representation. On the contrary, the graph representation is based on Portolan [28], a Model-Driven prototype tool, that provides facilities for designing and implementing cartography solutions in a MDE context, bridging the gap between a given (data) domain and the visualization world.

On the contrary, in the system structural part, the Visualization step is not merged with the business vocabulary since the semantic gap between the business terms in the conceptual model and the tables, columns and views in the database implementation is most of the time null. In addition, the text representation is aligned with the OCL format, since the identified rules are expressed by means of OCL models. Finally, the graph representation relies on UML, that provides a set of graphic notation techniques to create visual models.

4

Business rule extraction for COBOL

4.1 Motivation

COBOL [47] (COmmon Business Oriented Language) is a programming language that was popular some decades ago and nowadays it is mainly used to maintain existing systems. It is one of the most important and old legacy languages, since at the time it was created, it offered scalability, robustness, performance and mathematical accuracy. According to different surveys, COBOL is still used in large corporation companies and government organizations (i.e., banks, insurance companies, etc.) and plays a critical role in the business world. In particular, according to Microfocus [1], there are 200 billion lines of COBOL code in existence, that represents the 75 per cent of the world's actively used code. In addition, 70-75 per cent of the business and transaction systems (e.g., credit card, airline, hospital, payroll systems) around the world run on COBOL; while 90 per cent of global financial transactions are processed in such language [2].

On the other hand, only one million of COBOL programmers are estimated in the world. In addition, according to a recent Computerworld survey [3] run over 357 Information Technology professionals, 50 per cent of the respondents say the average age of their COBOL staff is 45 or older and 22 per cent say the age is 55 or older. This entails that in the next years, most of these developers will be retired.

Therefore, since on the one side there are many legacy COBOL systems that embed some critical business logic which must be preserved and maintained; and on

1. http://www.microfocus.com/aboutmicrofocus/pressroom/releases/pr20090528819202.asp
2. http://www.marblecomputer.com/cobol-Industry-Stats.html
3. http://www.computerworld.com/s/article/9227263/The_Cobol_Brain_Drain

the other side, the corresponding business knowledge (i.e., handled by developers) is being lost over time due to developer retirements; a business logic extraction process is needed to recover the business information from such COBOL systems.

In the following, we present the basic concepts for COBOL and finally we describe the Business Rule Extraction process for such systems.

4.2 COBOL basic concepts

COBOL is a procedural language that structures programs as a collection of global variables and data structures accessed and modified using procedures. A procedure is composed by sections, paragraphs, sentences and statements. A section contains paragraphs, a paragraph sentences and a sentence statements.

Special commands on paragraphs, sentences and statements can be used to alter the sequential control flow of a COBOL program. Such commands are respectively *PERFORM*, *GO TO* and *NEXT SENTENCE*. Iterations on the code are achieved using the *PERFORM* command, that transfers control to one or more statements and returns control to the next statement after the execution of such statements is completed. In addition, if the statements are contained in sequential paragraphs, *PERFORM* is extended with the word *THRU* to indicate the first and last executed paragraphs.

On the contrary, *GO TO* statements are used to transfer control from one paragraph to another; while *NEXT SENTENCE* phrases allow assigning control to the first statement of the sentence following that command.

In a COBOL program, procedures and the related sections, paragraphs, sentences and statements are nested within a logical container called *Procedure Division*. Other 3 divisions exist in a COBOL program: *Identification Division*, *Environment Division* and *Data Division*. They are used respectively to identify the program, to describe the input-output data sources used by the program and to declare the data structures of the program.

In particular, we focus on the *Data Division*, where two types of data structure can be defined: *Data Item* (i.e. or *Elementary Item*) and *Group Item*. The first are simple variables that specify different primitive types of data; on the other hand *Group Items* contain subordinate items (i.e., *Data* or *Group Items*) and they are used to represent complex data structures. In addition, both *Data* and *Group Items* are identified by a label.

The primitive types in COBOL are specified in *PICTURE* (i.e., *PIC*) clauses, that are used to describe a data in the program. They are defined according to 5 code characters (i.e., *9*, *V*, *S*, *X* and *A*), that are repeated to define the size (i.e., number of bytes) of a given data item. *9*, *V* and *S* deal with numeric representations; they

specify respectively a numeric value, the decimal point and the sign of the numeric value. On the other hand, *X* and *A* represent in turn alphanumeric and alphabetic (i.e., A-Z, space) values. These primitive types are specified in *Data Items*, that can be used to composed *Group Items*.

Finally, both *Data* and *Group Items* are defined in combination with a level number, that represent the data hierarchy. In particular, 01-49 are reserved for *Group* or *Elementary Items*; 66 for *Renames* clause, that allows regrouping *Data Items* in a *Group Item*; 77 for indipendent *Data Items*; and finally 88 for condition names, where each of them represents a value of a given conditional variable.

The concepts presented above give a small overview of COBOL and they are needed to understand the following Sections.

4.3 Running example

In order to illustrate our framework, a small COBOL program will be used as a running example. [4] The program allows a customer to buy products in a shop, if the shop is open. The shop offers several products, which are represented by an unit price and the available quantity. They can be bought if the customer has enough money and enough room in his bag to put the products in.

The data structures of the program are shown in Fig. 4.1. The shop is represented as a group item (i.e., set of variables), that define its property (i.e., open/closed variable *OP*) and the unit price and quantity for the products it sells (i.e., vegetables, meat, bread, milk, fruit). The other data structure (i.e., *MONEY*, *REST*, *BAG*, *MAX-CAP*, *NEED*) are data items that represent the customer information. In particular, *MONEY* and *REST* are respectively the money owned by the customer (i.e., the initial value is set to 50) and the money left after buying products; *BAG* and *MAX-CAP* are the maximum capacity of the bag and the number of the current products inside; and finally *NEED* defines if a product is needed or not.

```
01 SHOP.
   10 OP          PICTURE 9.
   10 QT-VEG      PICTURE 99.
   10 QT-MEAT     PICTURE 99.
   10 QT-BREAD    PICTURE 99.
   10 QT-MILK     PICTURE 99.
   10 QT-FRUIT    PICTURE 99.
   10 PR-VEG      PICTURE 9.
   10 PR-MEAT     PICTURE 9.
   10 PR-BREAD    PICTURE 9.
   10 PR-MILK     PICTURE 9.
   10 PR-FRUIT    PICTURE 9

77 MONEY     PICTURE 99, VALUE 50.
77 REST      PICTURE 99.
77 BAG       PICTURE 9.
77 MAX-CAP   PICTURE 99, VALUE 10.
77 RAND      PICTURE 9.
77 NEED      PICTURE 9.
```

Figure 4.1: Data structures of the running example

4. The source code, source model and the different output for each step of our framework concerning the running example can be found at http://docatlanmod.emn.fr/BrexCobolExample/intro.html

The structure of the program is shown in Fig. 4.2. In particular, the program starts with an initialization paragraph (i.e., *INIT*). If the shop is open, the products with their corresponding quantities and prices are initialized (i.e., *INIT-PRD*). Then, the list of products is scanned by means of five paragraphs: *BUY-VEG*, *BUY-MEAT*, *BUY-BREAD*, *BUY-MILK* and *BUY-FRUIT*. For each of these products, *ISNEEDED* checks whether that product is needed by the customer or not. If it is, the customer can buy it on condition that he has enough money and room in his bag. On the contrary, the program ends printing the information concerning the money left and the number of products bought.

Finally, if the list of the products is entirely browsed, but still enough money and room in the bag are available, a new iteration of that list can be performed.

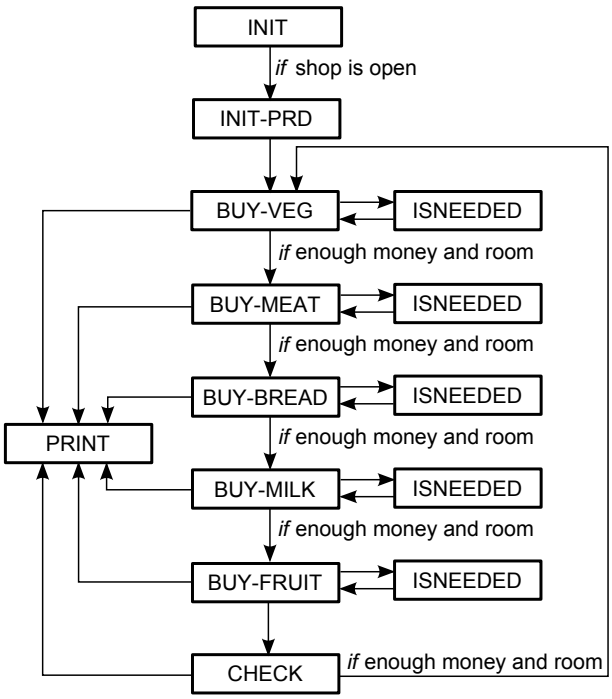

Figure 4.2: Program structure of the running example

The logic embedded in each of the paragraph representing the action of buying a given product follows the same principle. In Fig. 4.3, the logic coded in *BUY-FRUIT* and its related paragraphs are shown. The variable *NEED* is calculated in the paragraph *ISNEEDED* (for the purposes of the simulation, the variable is just assigned a random value). If the product is needed, there are still units available (*QT-FRUIT* variable), the customer has still some money and enough space in the bag, the product is added to the bag and both customer's money and product's quantity are updated. If one of the previous conditions is not true, paragraphs *CHECK* or *PRINT* are executed triggering the end of program or, depending on the remaining money and the room left in the bag, a new iteration for buying products.

```
BUY-FRUIT.
PERFORM ISNEEDED THRU ISNEEDED-FN.
a IF NEED = 1 AND QT-FRUIT > 0
    IF MONEY > PR-FRUIT AND BAG < MAX-CAP
       ADD 1 TO BAG
b      COMPUTE MONEY = MONEY - PR-FRUIT
       SUBTRACT 1 FROM QT-FRUIT
     ELSE
       GO TO PRINT
ELSE
   GO TO CHECK.
BUY-FRUIT-FN. EXIT.
```

```
CHECK.
IF MONEY <= 0 OR BAG >= MAX-CAP
   GO TO PRINT
ELSE
   GO TO BUY-VEG.
CHECK-FN. EXIT.
```

```
PRINT.
MOVE MONEY TO REST.
DISPLAY "REST:" MONEY.
DISPLAY "NB OF PRODUCTS:" BAG.
FIN.
```

```
ISNEEDED.
   COMPUTE NEED = FUNCTION RANDOM (1) * 2.
ISNEEDED-FN.EXIT.
```

Figure 4.3: BUY-FRUIT and its related paragraphs

4.3.1 Rules modeling the application

Despite the simplicity of the running example proposed, it contains different business rules. A manual inspection of the source code allows to identify many of them:

- If the shop is open, then the customer can buy products
- If a product P is needed and available and if the client has enough money and room in his bag, then the customer buys P
- If a product is bought, its quantity is subtracted by one
- If a product is bought, its price is decreased from the money of the client
- If a product is bought, it is added to the bag

In the next sections, we will show how to identify and extract those rules from the source code.

4.4 Framework description

The framework (Fig. 4.4) for extracting business rule from COBOL systems is instantiated according to the definition given in Chap. 3. Hence, it is composed by four steps: Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation.

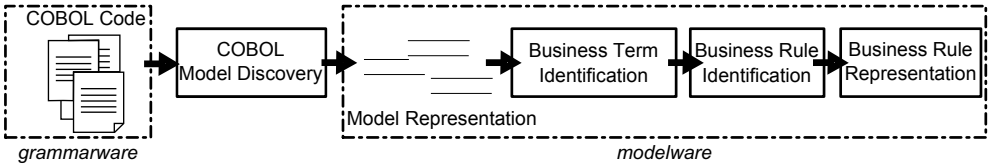

Figure 4.4: COBOL Business Rule Extraction framework

The Model Discovery generates a one-to-one model representation of the input source code. Such model is then manipulated in the next framework's steps to extract the business rules.

In particular, Business Term Identification identifies in the code the variables representing business terms/concepts. Business Rule Identification locates business rules using code slicing techniques [33] on the variables found in the previous step. Finally, the Business Rule Representation step visualizes the extracted rules.

Additionally, our framework provides rule traceability, meaning that the framework ties the source code elements to the elements composing the business rules. This helps users navigating back and forth between the rules and the input code. Traceability is implemented by explicitly linking in each phase transition the input model (or code) elements with the corresponding output model elements generated by the model transformations executed in that phase.

Next sections explain Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation steps in detail.

4.5 Model Discovery

Model Discovery generates a model representation of the source code. It relies on the facilities of Rational Developer for System z [5] that provides several development tools for creating, deploying and maintaining applications on IBM z/OS operating systems. Such applications can be implemented in COBOL, PL/I, C++, assembler and Java. In particular, within Rational Developer for System z, IBM COBOL Application Model API [6] is able to represent the COBOL source code as a model, since such API is based on a COBOL metamodel that contains a corresponding class for each element in the source code.

A simplified version of the COBOL metamodel is shown in Fig. 4.5. A *ProgramSourceFile* is composed by a list of programs. Each *Program* contains an identification, environment, data and procedure division. For the sake of comprehension, we detail only the last two ones, represented by *DataDivision* and *ProcedureDivision* classes. The former is defined by *FileSection*, *LinkageSection*, *LocalStorageSection* and *WorkingStorageSection*. In particular, the latter is composed by a list of *TopLevelVariables* (i.e., *Level01Item*, *Level77Item*), where each of them contains a *DataItem* that can be *GroupDataItem*, *TableDataItem*, etc. On the other hand, the *ProcedureDivision* is composed by sections, a *Section* by paragraphs, a *Paragraph* by sentences and finally a *Sentence* by statements (i.e., *Stmt* class). Each *Stmt* according to its type can represent different statements such as *MOVE*, *GO TO*, *PERFORM*, *EXIT*, etc.

5. http://www-03.ibm.com/software/products/us/en/developerforsystemz
6. http://tinyurl.com/IBMCobolApplicationModel

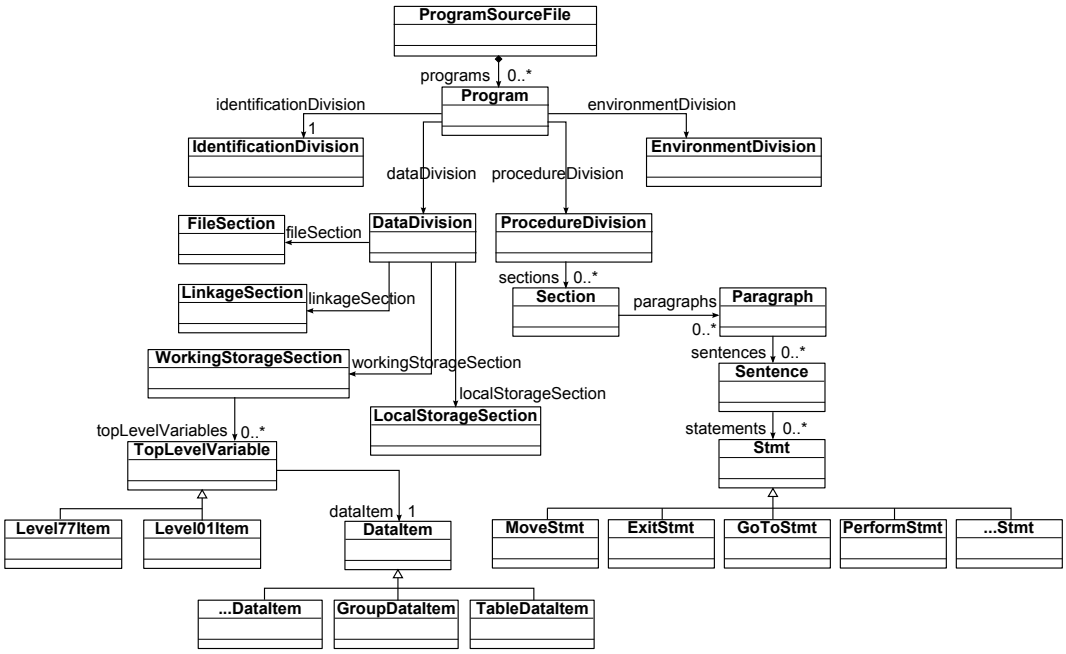

Figure 4.5: COBOL metamodel excerpt

4.6 Business Term Identification

The Business Term Identification step reduces the number of variables to analyse by filtering out those that are not business relevant. It takes as input the COBOL model and returns the "business" variables.

This step can be manual or automatic. In the first case, the user navigates the code and directly marks the variables to analyse. In the second case, an heuristic-based strategy identifies such variables based on the kind of statements in which they appear (and their role in them).

First, the statements are divided in several groups according to the COBOL command they contain. The groups are: *conditional*, *computation*, *in-out*, *end*, *move*, *goto*, *perform* and *call*. In particular, *conditional* regroups the different kinds of if-statements that exist in COBOL (i.e., if-then, if-then-goto, if-then-else, if-then-else-goto, if-next-else-goto, etc.). *Computation* contains the statements that model mathematical operations (i.e., *COMPUTE*, *ADD*, *SUBTRACT*, etc.). *In-out* group collects the statements used to prompt or get information from input sources (i.e., *DISPLAY*, *ACCEPT*). *End* includes commands used to end a program (i.e., *STOP RUN*, *GO BACK*, etc.). The remaining groups are composed by only one kind of statements, that is represented by the name of the group.

Secondly, after grouping the statements, for each of them, the variables in it are collected in four categories. *Condition variables* are variables in *if* statement conditions (e.g. *NEED* and *QT-FRUIT* at line *a* in Fig. 4.3); *index variables* are indexes of array structures; *source variables* and *target variables* are respectively the variables affecting and being affected in a statement (e.g. target variable: *MONEY*,

source variables: *MONEY* and *PR-FRUIT* at line *b* in Fig. 4.3).

Based on this classification, we have proposed two complementary heuristics, that state respectively that all target variables in *computation* statements or all the variables that appear in *in-out* statements are strong candidates to be classified as business variables.

Note that an hybrid approach is also envisagable where an automatic step returns a set of candidate variables and then the user filters some of them.

Computation	PR-FRUIT, PR-BREAD, QT-VEG, QT-MILK, BAG, QT-BREAD, PR-VEG, NEED, MONEY, PR-MEAT, QT-MEAT,PR-MILK,QT-FRUIT
In-out	MONEY, BAG
Conditional	PR-FRUIT, PR-BREAD, QT-VEG, QT-MILK, BAG, QT-BREAD, PR-VEG, NEED, MONEY, PR-MEAT, QT-MEAT, PR-MILK, QT-FRUIT, MAX-CAP, OP

Figure 4.6: Business term identification step for the running example

Figure 4.6 shows the result of the heuristics previously described concerning the running example. All target variables in *computation* statements are depicted on the upper row; while on the center and on the bottom the variables respectively in *in-out* and *conditional* statements are listed.

4.7 Business Rule Identification

Business Rule Identification (Fig. 4.7) is composed by three sub-steps. Control Flow Analysis, Rule Discovery and Data Flow Analysis.

The global inputs are the model generated from a COBOL program and one or more variables identified in the previous step [7]. The output is a Control Flow Graph enriched with the information about the statements composing the business rules for the given input variable(s).

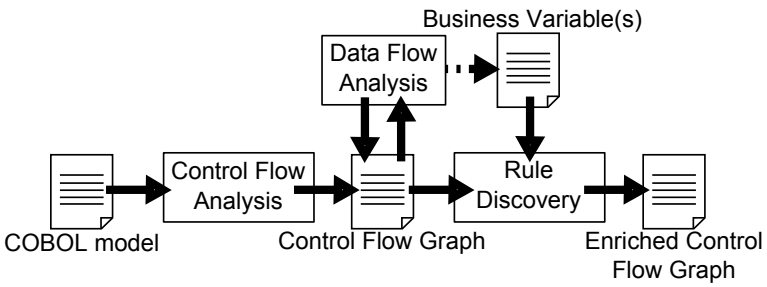

Figure 4.7: Business rule identification step

7. For the sake of comprehension, we describe the process assuming a single variable as input

4.7.1 Control Flow Analysis

This step generates a CFG model from a given COBOL model. The CFG model is derived from the original COBOL model enriched with information concerning the possible execution flows of the program (while the original model keeps only information about the syntactic order of the statements in the code).

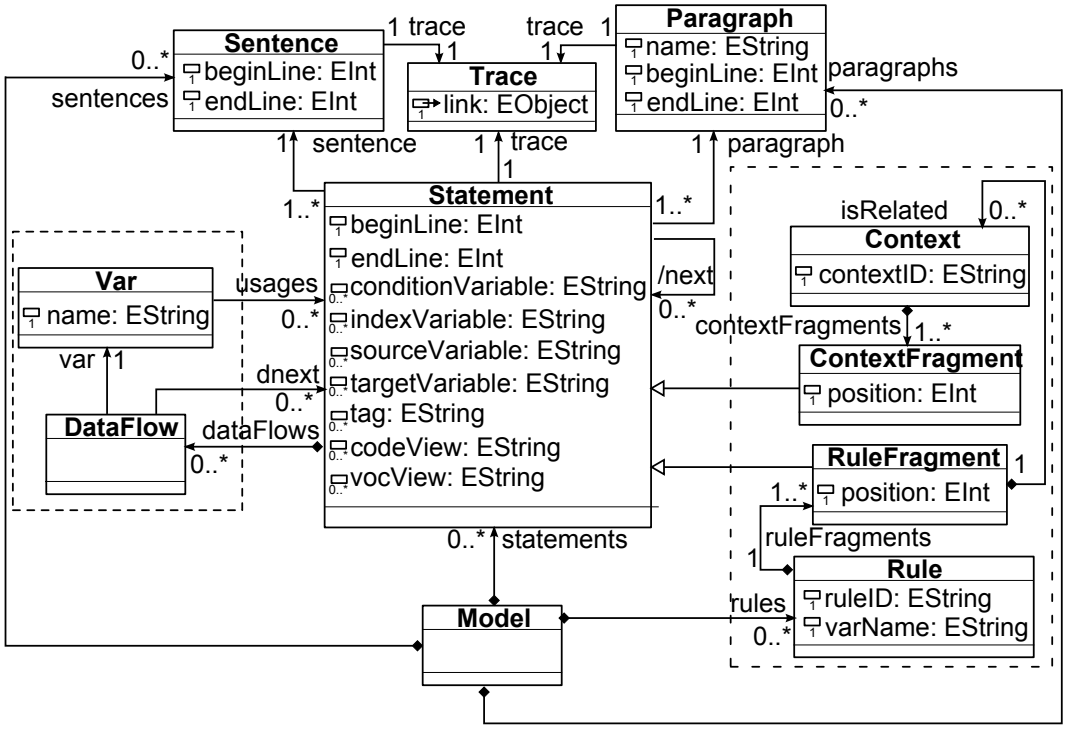

Figure 4.8: Data flow metamodel (left) control flow metamodel (center) and business rule entities (right)

The CFG model conforms to the metamodel shown on the left of Fig. 4.8. The entity *Model* stores *Paragraphs*, *Sentences* and *Statements* composing the program. They are all linked to the entity *Trace*, that is used to support the MDE traceability in the framework (i.e. the attribute *link* stores the references to the corresponding COBOL model entities).

The key element that stores the relationships between the statements is the association *next*. It indicates all possible statements to be executed next after a statement *X* (which one will be next may change on each execution depending on the run-time conditions, so this association collects all possible alternatives). From this *next* association we derive all the other next-like associations to facilitate the analysis in the following steps in the framework.

The rules to compute the *next* statements of a statement *stat* are listed in Tab. 4.1. The first six rules are related to the type of *stat*; while the following two concern its position. The last rule is applied to all statements not included in the previous rules.

The information concerning the variables referenced in statements are stored in

Current Stat	Next
Type: PERFORM-THRU	first statement of PERFORM-THRU
Type: GO-TO	first statement of the GO-TO paragraph
Type: EXIT PROGRAM, STOP RUN, GO BACK	none
Type: IF-THEN-ELSE	first statements in THEN and ELSE branches
Type: NEXT SENTENCE	first statement of the following sentence
Type: IF-THEN	first statements in THEN and after IF stat.
Position: last stat in THEN branch	first statement after IF
Position: last stat in PERFORM-THRU	statement after PERFORM-THRU
Position: - **Type:** -	following statement

Table 4.1: Rules to calculate the next statement

the attributes *conditionVariable*, *indexVariable*, *sourceVariable* and *targetVariable*; while the type of the statement is stored in the attribute *tag* (for the sake of simplicity, these concepts are directly represented as Strings instead of appearing as separate classes in Fig. 4.8). Other attributes store the position of the statement (begin and end lines) and the *Sentence* and *Paragraph* containing it for traceability purposes.

The remaining entities in the metamodel (i.e., *DataFlow*, *Var*, *Rule*, *RuleFragment*, *Context* and *ContextFragment*) are discussed in the Data Flow Analysis and Rule Discovery steps.

4.7.2 Data Flow Analysis

This step is used to find relations between (i.e., business-relevant) variables. The information collected can be used to run a further Rule Discovery step on a new set of related variables. Data Flow Analysis is an optional operation that takes as input the model that represents the CFG of the application and returns the same model enriched with the data flow information concerning the variables within that application.

The elements that stores the information related to the data flow are collected in the *DataFlow* and *Var* entities of the CFG metamodel (i.e., on the left in Fig. 4.8).

For each variable *v* within a statement *stat*, a *DataFlow* is created and it is linked to *stat* by the reference *dataflows*.

A *DataFlow* is defined respectively by a reference to the variable *v* it contains (i.e., *var*) and by a list (i.e. *dnext*) of statements. These are the first statements that

follow *stat* in the CFG (according to the rules defined in Tab. 4.1) and that contain the same variable.

Finally the entity *Var* represents the variable *v* and it contains the name of that variable and a list of statements (i.e., *usages*) that collect the statements where *v* is used.

4.7.3 Rule Discovery

Rule discovery relies on program slicing techniques to recover the business rules associated to one or more variables.

A rule represents a possible execution path in the program relevant to a business variable. It includes one or more statements modifying/accessing such variable. According to the possible execution paths, several rules may exist for the same variable. Each rule represents an independent execution path in the code, that is no fully-contained in other ones.

A rule is composed by rule fragments, that are selected statements in the code. A rule fragment can be either a statement *S* where the input variable is referenced or a conditional statement that contains in one of its branches *S*. Optionally, a rule fragment may be associated to contexts. A context contains the remaining conditions in the control flow that trigger that rule fragment. Thus, a context is composed by context fragments, that are the conditions of conditional statements.

Rule Discovery locates the business rules related to a business variable in the program. The inputs of this step are the CFG model and a variable. The output is the CFG enriched with information about the business rules related to that variable. It is divided into two steps: Rule Fragment Identification and Rule Context Identification.

Rule Fragment Identification

This process is composed by three phases. Initially, the statements containing the business variable passed as input are located in the CFG. In the second phase, the execution paths including these statements are calculated (i.e., during this calculation, only the statements identified in the first step are added to the execution paths). For each of these execution paths a *Rule* (Fig. 4.8) is created. It is defined by an identifier *RuleID* and the name of the variable passed as input (*VariableName*).

In the last phase, the conditional statements that include the statements identified in the first step are added to the corresponding paths. Finally, all these statements are stored as *RuleFragments* (Fig. 4.8) and their locations in the execution path are saved in the attribute *Position*.

In Fig. 4.9, the identification of the rule fragments concerning the variable *BAG* is shown. For the sake of comprehension, we focus only on the execution path

```
INIT.
  IF OP = 1
     DISPLAY "SHOP IS OPEN"
     PERFORM INIT-PRD THRU INIT-PRD-FN
     GO TO INIT-FN
  ELSE
     DISPLAY "SHOP IS CLOSED"
     GO TO INIT.
INIT-FN.EXIT.
BUY-VEG.
PERFORM ISNEEDED THRU ISNEED-FN.
IF NEED = 1 AND QT-VEG > 0
   IF MONEY > PR-VEG AND BAG < MAX-CAP
      ADD 1 TO BAG
   ...
   ELSE ...
```

Figure 4.9: example of Rule Fragment Identification

containing the statement in the paragraph *BUY-VEG*. The selection of the statement *ADD 1 TO BAG* is the result of the two first steps of the Rule Fragment Identification process. In the third step, the conditional statements that include this statement (i.e., *IF NEED = 1 AND QT-VEG > 0* and *IF MONEY > PR-VEG AND BAG < MAX-CAP*) are added to the rule.

Rule Context Identification

The process to identify the contexts that are related to each *RuleFragment* of a *Rule* is composed by two phases. Firstly, for each *Rule*, the corresponding *RuleFragments* are retrieved from the CFG. Later, each *RuleFragment* is used as backwards starting point to discover the ordered sets of control flow condition that might have been crossed in the program, without passing by other *RuleFragments* of the same *Rule*. Each set of if-conditions represents a *Context* (Fig. 4.8) and it is defined by an identifier *ContextID*. Any condition in a *Context* set is a *ContextFragment* (Fig. 4.8) and its location inside the context is stored in the attribute *Position*.

In Fig. 4.10, the paragraph *BUY-VEG* follows the paragraph *INIT*, which is the first in the program. The *RuleFragment*, in the box on the right, contains the *rule fragments* concerning the variable *BAG*. On the left column, the box contains the *Contex*, that is composed by one *ContextFragment* (i.e., *IF OP = 1*), since only this if-condition is crossed to reach the *RuleFragment*.

4.8 Business Rule Representation

Business Rule Representation (Fig. 4.11) is the last step of the framework. Its goal is to generate comprehensible textual and graphical representations of the discovered business rules and their orchestration (i.e. connections and precedences among the rules).

```
INIT.
 IF OP = 1
    DISPLAY "SHOP IS OPEN"
    PERFORM INIT-PRD THRU INIT-PRD-FN
    GO TO INIT-FN
  ELSE
    DISPLAY "SHOP IS CLOSED"
    GO TO INIT.
INIT-FN.EXIT.
BUY-VEG.
PERFORM ISNEEDED THRU ISNEED-FN.
IF NEED = 1 AND QT-VEG > 0
   IF MONEY > PR-VEG AND BAG < MAX-CAP
      ADD 1 TO BAG
   ...
   ELSE ...
```

Figure 4.10: example of Rule Contex Identification

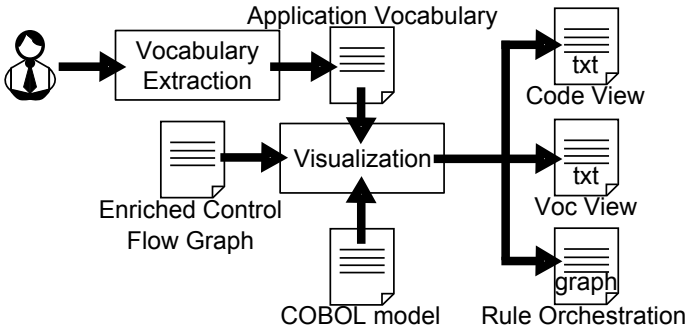

Figure 4.11: Business Rule Representation step

This step is composed by two operations: Vocabulary Extraction and Visualization.

4.8.1 Vocabulary extraction

Vocabulary extraction is an optional step simply aimed at providing the set of labels for each variable (in natural language) defined by the user. Figure 4.12 shows a vocabulary excerpt for the running example. This can be a manual operation or an assisted one.

```
<vocabulary:Model ...>
...
<entries key="OP" value="OPEN" />
<entries key="QT-MEAT" value="QUANTITY MEAT" />
<entries key="PR-MEAT" value="PRICE MEAT" />
<entries key="MAX-CAP" value="MAXIMUM CAPACITY" />
...
</vocabulary:Model>
```

Figure 4.12: Running example vocabulary

The vocabulary model conforms to the metamodel presented in Fig. 4.13. The root element of this metamodel is the entity *Model* that contains a list of *programs*.

A *Program* is defined by a *name* and a *label* containing its description. It can have zero or more *Entries*, which store the name of the variable and its description in the attributes *key* and *value*.

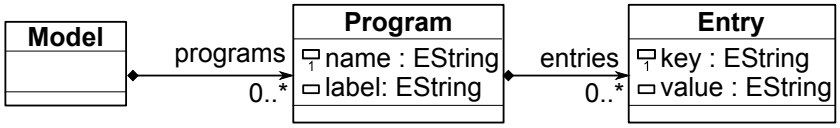

Figure 4.13: Vocabulary metamodel

4.8.2 Visualization

This step provides artifacts that ease the comprehension of the business rules and their relations. Its inputs are the COBOL model, the corresponding CFG containing the business rule information and optionally the vocabulary of the application. The outputs are text-based and graph-based representations of the gathered rules according to the information contained in the CFG model (Fig. 4.8). This step is divided into two sub-steps: Textual Visualization and Graphical Visualization.

Textual Visualization

All the *Rules* in the CFG are collected to generate the textual representations. The *RuleFragments* composing a *Rule* are retrieved and ordered (thanks to the *Position* attribute) according to their relative positions in the corresponding execution path. For each *RuleFragment*, the related entity in the COBOL model is retrieved and a code textual representation is calculated from it. If the vocabulary has been defined, also a vocabulary-based representation of the *RuleFragment* is created. In this case, the textual representation is calculated mixing the hard-coded translations of the COBOL commands with the descriptions of the variables in the vocabulary.

Finally these textual representations are stored back in the CFG (i.e. attributes *codeView* and *vocView* of the *Statement* class that is the super-classes of the corresponding *RuleFragment*) and the rules are saved in textual files. Each file will contain separately all the rules discovered for a given variable. The same process is done for the textual representations of *Contexts* and *ContextFragments*.

The example in Fig. 4.14, shows the rule *PR-MEAT/PRICE MEAT* for technical and business users, and due to space limitations only the *ContextFragments* of the first *RuleFragment*.

Graphical Visualization

Relationships between the rules are better displayed by means of a graph-based representation. Orchestration is achieved connecting together the rules that share at

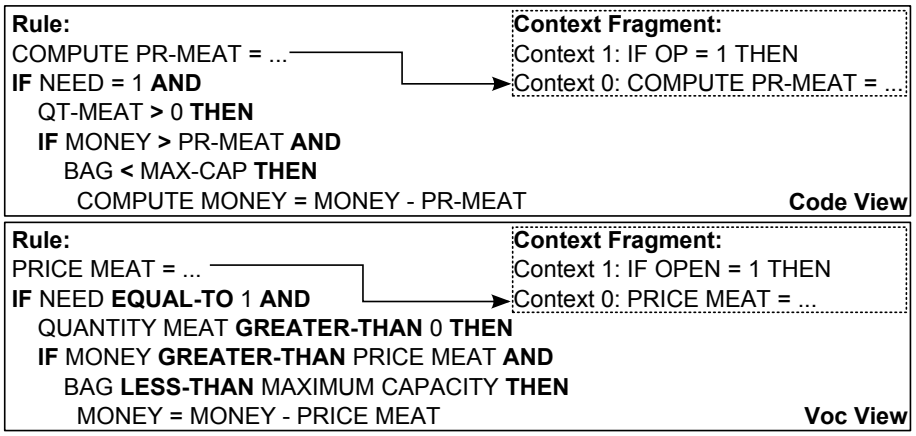

Figure 4.14: Example of textual outputs for the rule PR-MEAT/PRICE MEAT

least a *RuleFragment*/*Statement* that includes mathematical operations (i.e. ADD, ...).

In Fig. 4.15 the three rules concerning the variables *PR-MEAT* (in the box on the right), *PR-BREAD* (in the box on the left) and *MONEY* (at the center) are shown. In this example, the BREX process locates only one rule for each variable. The rules concerning the variables *PR-BREAD* and *PR-MEAT* are connected to the rule related to the variable *MONEY*, since they share with it a *RuleFragment*. The rule *MONEY* is not shown entirely due to space limitations.

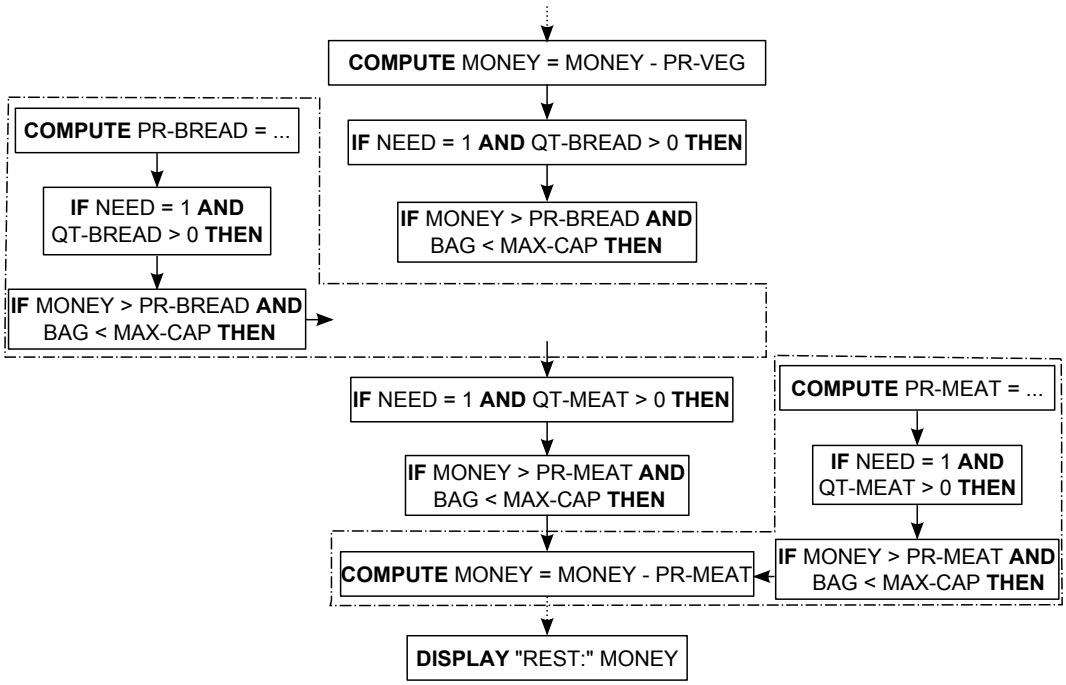

Figure 4.15: Orchestration of the rules *PR-MEAT*, *PR-BREAD* and *MONEY*

4.9 Optimization

COBOL systems are often composed by several programs. They are used to perform specific manipulations on the data that can be related or not to business rules. As a consequence, in order to quickly understand the impact of a program with respect to a given business variable, we have provided the framework with an additional graph-based representation for the identified business rules.

The proposed graph-based representation consists of a graph composed of nodes and edges that are respectively the paragraphs in a program and connections between paragraphs. Each node has a label that corresponds to the name of the paragraph and it is colored if it contains at least a *RuleFragment* or *RuleContext*.

On the other hand, the connections in the graph are derived from the analysis of the CFG (Fig. 4.8) of the program. In particular, for each statement *stat* and its container paragraph *par*, all next statements of *stat* not contained in *par* are selected and the corresponding paragraphs are collected. The pairs composed by each of such paragraphs and *par* represent edges in the graph.

In Fig. 4.16, the paragraph graph-based representation for the running example concerning the variable PR-VEG is shown. The involved paragraph are *INIT*, *INIT-PRD* and *BUY-VEG* that contain respectively the context of the rule, the rule fragment concerning the variable initialization and finally, the other rule fragments.

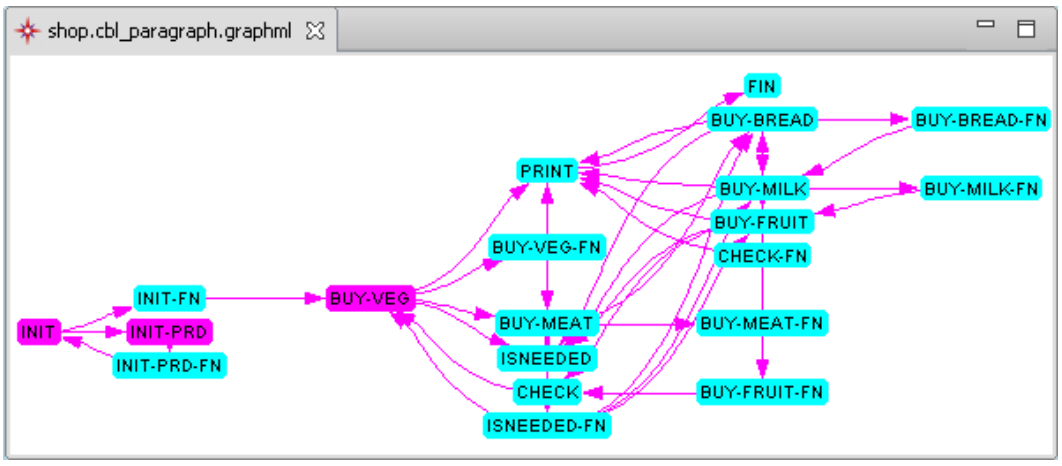

Figure 4.16: Graph-based paragraph representation for the variable PR-VEG

4.10 Evaluation

In order to evaluate the accuracy of our method we have performed two evaluations. The first evaluation has been focused on checking that the business rules returned at the end of the extraction process for the running example coincide with the ones that we discovered by manual inspection. For the running example, we

were able to generate both graphical and textual representations of all the identified rules, facilitating this way the comprehension of the application.

The second evaluation has been conducted on a use case provided by IBM. The evaluation has concerned the analysis of an IBM RPP COBOL application managing flight and pilots containing 14 programs and 130 variables in around 6500 lines of code.

We asked four internal IBM COBOL experts to analyse the business rules generated by our framework and assess whether the rules were meaningful (i.e. they were actual business rules) and understandable. They had access to the original COBOL code and were given two hours to perform the evaluation. For the sake of simplicity, instead of generating all rules for the system, we focused on the rules related to a small subset of business variables previously identified.

At the end of evaluation, they all agreed the framework was able to generate the complete set of rules for the input variables and considered the result useful to understand the COBOL code. The only concern was that the rules were not completely "clean" meaning that some of them still included technical statements (e.g read access file operations).

The IBM experiment has allowed us to improve the quality of the output artifacts generated by the framework. In particular, at the beginning of the experiment, the experts were looking for the programs that were more business relevant. Unfortunately, this search was time-consuming, due to the number of programs to check. Hence, we have added to the Business Rule Representation step an additional graph-based representation for the rules (Sect. 4.9). Such representation allows the user to have a quick understanding regarding the business-relevance of a given program, since the program is seen as a graph composed by paragraphs and their connections, where each paragraph that includes a business-relevant statement is colored.

4.11 Prototype

The framework [8] has been implemented to work with both generic and IBM-specific contexts.

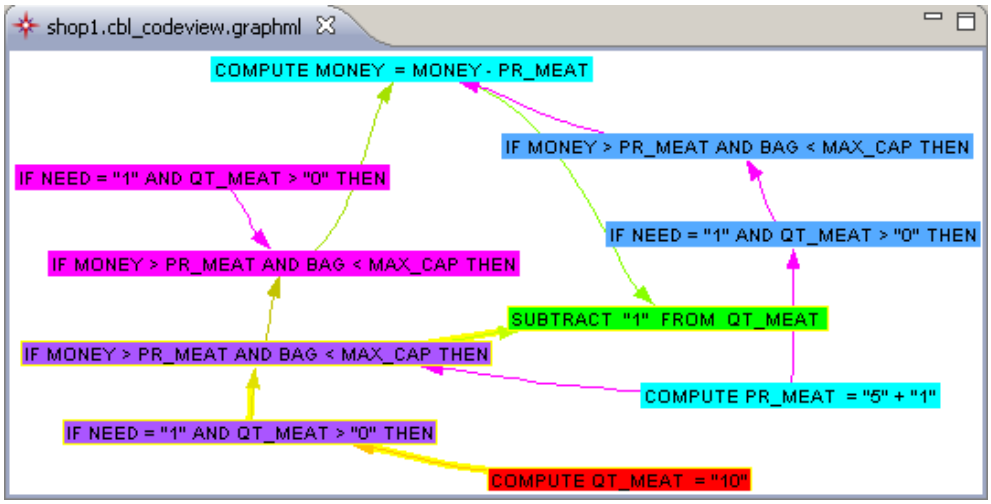

Figure 4.17: Example of graph-based representation of rules

IBM tools have been used throughout the framework. Model Discovery phase relies on COBOL Application Model of IBM Rational Developer [9] (RDZ). Business Term Identification uses the functionalities provided by IBM Rational Programming Patterns [10] (RPP). RPP is strongly designed on MDE principles, which eases the integration with the framework. RPP allows to attach to any data structure (programs, variables, ...) a label containing a short explanation and provides an interface that facilitates the navigation of all the data structures composing a COBOL system. In this way, it is possible to collect automatically those labels as part of the reverse engineering process and associate them to the corresponding entity. This vocabulary can then be used to improve the visualization of the extracted rules.

Other auxiliary tools are Portolan [11], the ATL Transformation Language (ATL) [14] and Xtext [**?**]. Portolan is a Model-Driven Cartography tool, used to represent graphically the extracted rules and to emphasize their orchestration (Fig. 4.15). In Fig. 4.17, a Portolan graph-based representation is shown. It contains the statements composing the rules PR-MEAT (i.e., price meat) and QT-MEAT (i.e., quantity meat). In addition, Portolan allows to find all possible paths from a source statement (COMPUTE QT_MEAT = Ï0ïn the figure) to a target statement (SUBTRACT ÏFŘROM QT_MEAT). On the contrary, ATL is used to implement all model

8. A demonstration of the tool is available at http://docatlanmod.emn.fr/BrexCobolExample/intro.html
9. http://publib.boulder.ibm.com/infocenter/ieduasst/rtnv1r0/index.jsp
10. http://publib.boulder.ibm.com/infocenter/rppzhelp/v8r0/index.jsp
11. http://code.google.com/a/eclipselabs.org/p/portolan/

manipulation operations required by the three steps of the method; and finally Xtext is used for the text-based representation of the rules (Fig. 4.18) . All these tools are open source source.

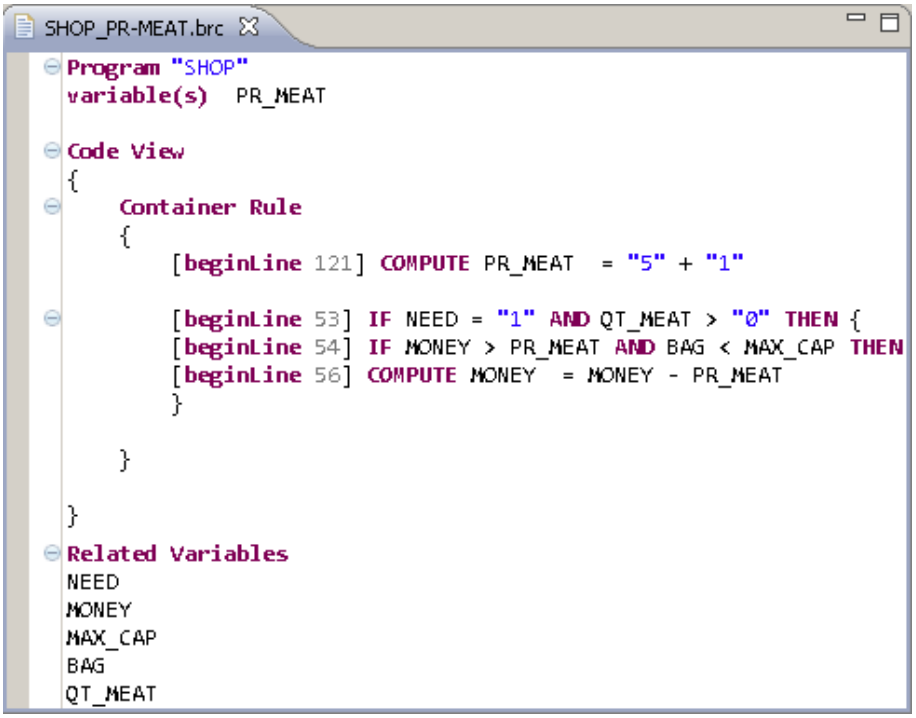

Figure 4.18: Example of text-based representation of a rule

MDE has facilitated the interoperability among the different tools employed in the framework. The framework is packaged and distributed to its users as an Eclipse plug-in.

5

Business rule extraction for Java

5.1 Motivation

Java [48] is a popular programming language used by billions of devices [1]. It was created almost two decades ago and since then many organizations have implemented their systems with such technology. Since these systems have been evolved and modified over time and the corresponding documentation is often out-of-date, most of these companies consider their systems as legacies. As a consequence, we believe that a business rule extraction process is needed to ease the comprehension of Java information systems.

In the following, we introduce the basic concepts for Java and finally we describe the business rule extraction process for Java systems.

5.2 Java basic concepts

Java is a concurrent, class-based, object-oriented language that has been designed for general purpose application development. According to [49], the basic notion of object-oriented programming is the one of *object*. An object is a programming unit that associates data with operations. In particular, these operations can be employed to use or affect such data. These operations are called *methods*, while the data they affect are the *instance variables*.

Java structures programs as a collections of files (i.e., extension .java) that may be organized in *packages*. They are identified by respectively a name and a unique

1. http://www.java.com/en/about/

namespace.

A Java file can contain *classes* and *interfaces*. A *class* is the representation of a type in the system/a concept in the real world. It is the common definition from where objects (i.e., instances of that class) are created. Finally, a class is composed by attributes (i.e., the properties/variables that a class has) and methods (i.e., the operations/functions that can be executed on that class). On the other hand, an *interface* is an abstract type, that contains method signatures.

Java allows to define hierarchical relations between classes and interfaces; in particular, a class derived from another class is called *subclass*; while the class from where the *subclass* depends is called *superclass*. In addition, a *subclass* inherits the methods (i.e., default behavior) from the corresponding *superclass*. Finally, a *class* can *implement* an *interface*, but it must define all method signatures of that *interface*.

The concepts presented above will be used in the next section to explain the business rule extraction for Java-based systems.

5.3 Running example

In order to illustrate our framework, we will use as running example a Java application that belongs to the simulation software category and that contains several business rules [2].

The application simulates the behavior of animals and humans in a meadow, where each actor, animal or human, can act and move according to its nature. Two different functionalities are implemented in this application. The former represents the business logic and in particular describes how predator-prey interactions affect population sizes. The latter is used to store statistical information about the actors participating in the simulation.

A schema of the application classes and their relationships is shown in Fig. 5.1. The application is composed by 2 packages and 16 classes. The presentation and the domain layers are clearly separated.

The presentation layer is composed by *GUI*, *Simulator*, *SimulatorView*, *AnimatedView* and *FieldView*. The class *GUI* shows the graphical interface of the application. *Simulator* simulates the predator-prey game and it stores information for statistical analysis. *SimulatorView*, *AnimatedView* and *FieldView* represent the graphical views of the application.

The remaining classes represent the application layer. The ground is represented by three classes: *Field*, *Location* and *Grass*. In particular, *Field* is a rectangular grid of field positions, where each position is modeled as a *Location* identified by Cartesian coordinates; while *Grass* models the grass on the field.

2. The source code of the running example can be found at http://docatlanmod.emn.fr/BrexJavaExample/

The participants of the simulation are represented by *Actors*. An *Actor* is an interface that contains methods to modify the actor's location and to perform the actor's behavior.

Human and *Animal* implement *Actor*, they both are abstract classes. In particular, the former is extended by the class *Hunter*, that represents hunters and stores the related current position in the field; while the latter is extended by the classes *Bird*, *Fox* and *Rabbit*. In addition, the class *Animal* stores the actual *age*, the *location* in the field and the *food level* of animals. It contains also properties that represent respectively if an animal is *alive*, its maximum age as well as its breeding age, its breeding likelihood and finally, the maximum number of births that an animal can have. All such properties are particularized for each of the subclasses that extend *Animal*.

Finally the class *Counter* provides a counter for each participant in the simulation.

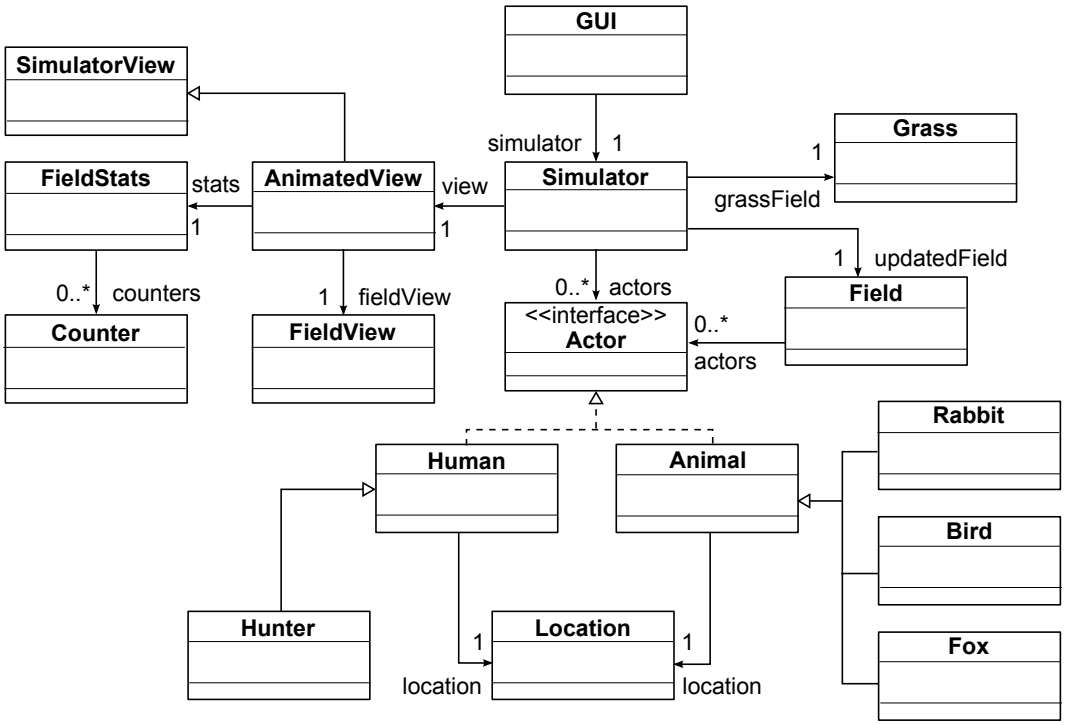

Figure 5.1: Class diagram of the running example

5.3.1 Rules modeling the application

A manual inspection of the source code of these classes reveals the existence of several business rules.

Rules modeling hunter behaviors are:

– Hunters never die

– Hunters hunt animals

Rules modeling bird and rabbit behaviors are:

- Rabbits/Birds can die by being eaten by foxes, hunted by hunters, because of starvation, old age or overcrowding
- Rabbits/Birds can breed when they reach their breeding age
- Rabbits/Birds eat grass

Rules modeling the fox behaviors are:

- Foxes can die by being eaten by hunters, because of starvation, old age or overcrowding
- Foxes can breed when they reach their breeding age
- Foxes eat rabbits or birds

In the next sections, we will show how to identify and extract those rules from the source code.

5.4 Framework description

The framework (Fig. 5.2) for extracting business rules from Java-based systems is an instantiation of the conceptual framework presented in Chap. 3. Therefore, it is composed by Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation.

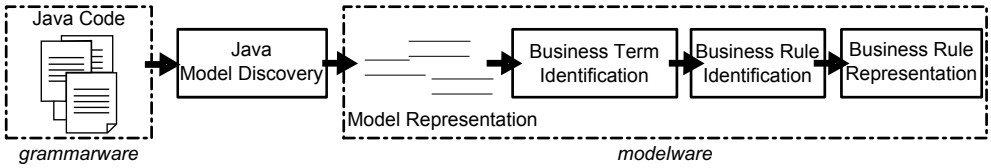

Figure 5.2: Java Business Rule Extraction framework

Model Discovery takes as input the source code of a Java application and generates a model-based representation of the code. Such representation has a one-to-one correspondence with the code, hence no information is lost when passing from the grammarware to the modelware.

The model obtained from the Model Discovery is used in the next steps of the framework. In particular, Business Term Identification identifies the variables that hint at business terms/concepts in this model. Business Rule Identification locates the business rules related to the variables discovered in the previous step using slicing techniques [33]. Finally, Business Rule Representation provides artifacts for representing the business rules identified.

The proposed framework provides traceability between the extracted business rules and the source code elements by means of MDE traceability. In particular, for each model transformation (Sect. 2.4.3) in the framework, traceability links are created between the corresponding input and output model elements.

In the following, Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation are described in detail.

5.5 Model Discovery

Model Discovery generates a model from the Java source code composing the application. It is based on the facilities of MoDisco [3], an extensible model-based framework under Eclipse [4] to support and ease software modernization processes.

In particular, we rely respectively on the MoDisco Java metamodel [5] to represent the source code, and on its Java discoverer to instantiate this metamodel based on the source code of a given Java application. Such discoverer relies on the Eclipse project Java Development Tools (JDT [6]) to navigate the AST of the Java source code. The obtained Java model contains the full AST of the Java application (e.g., each statement such as attribute definition, method invocation or loop are represented), and it stores links between model elements (e.g., bindings between a method invocation and the declaration of the corresponding method, the usage of a class and its declaration, etc.).

The MoDisco Java metamodel [7] mirrors the structure of the Java language, as defined in the Java Development Kit (JDK) 5. It is composed by 126 classes and each of them except for the root one (*Model*) represents a Java source code construct. The instances of such code constructs are the AST nodes in the AST. In addition, since many Java constructs are named (e.g., methods, types, variables, packages, etc.), the corresponding classes in the metamodel inherit from the class *NamedElement*, that stores the name of each of these constructs. Apart from this, such class is also in charge of indicating if a *NamedElement* is part of the current Java application or not (e.g., element from an external library of from the JDK). As a consequence, the external elements are tagged as proxy by using a dedicated attribute (i.e., proxy) in the *NamedElement* class.

In Fig.5.3 the model representation of the class *Thread* is shown. It is composed by two *NamedElements*, respectively one class and one method declaration. Since the class *Thread* and the method *sleep* are part of the JDK, the attribute *proxy* of such elements are initialized to *true*.

3. http://www.eclipse.org/MoDisco/
4. http://www.eclipse.org/
5. the Java metamodel can be downloaded at http://tinyurl.com/sourcejavametamodel
6. http://www.eclipse.org/jdt/overview.php
7. http://tinyurl.com/JavaModiscoMetamodel

```
<ownedElements xsi:type="java:ClassDeclaration" name="Thread" proxy="true" ...>
  <bodyDeclarations xsi:type="java:MethodDeclaration" name="sleep" proxy="true" ...>
    <parameters name="arg0" proxy="true">
      <type type="//@orphanTypes.1"/>   <----------  int      Runnable
    </parameters>                                              |
  </bodyDeclarations>                                          |
  <superInterfaces type="//@ownedElements.3/@ownedPackages.3/@ownedElements.10"/>
</ownedElements>
```

Figure 5.3: Excerpt of a JDK model element

A simplified version of the Java metamodel is shown in Fig. 5.4. The root class is *Model*, that contains a list of packages. A *Package* contains in turn *AbstractTypeDeclarations* that can be *InterfaceDeclarations* or *ClassDeclarations*. Each *AbsractTypeDeclaration* is composed by *bodyDeclarations*. A *BodyDeclaration* is a *FieldDeclaration* or a *MethodDeclaration*, in particular the former is a subclass of *AbstractVariablesContainer*, that contains *VariableDeclarationFragments* used to store the references of the variables in the code (i.e., *usagesInVariableAccess*); while the latter has a body (i.e, *Block*) that includes a list of statements (e.g., *ForStatements*, *ExpressionStatements*, *ReturnStatements*, *VariableDeclarationStatements*, etc.). Each *Statement*, depending on its type, can contain one or more expressions such as *SingleVariableAccesses*, *MethodInvocations*, *PostfixExpressions*, *InfixExpressions*, *Assignments*, *Literals*, etc. In addition, each *Expression* can contain other expressions.

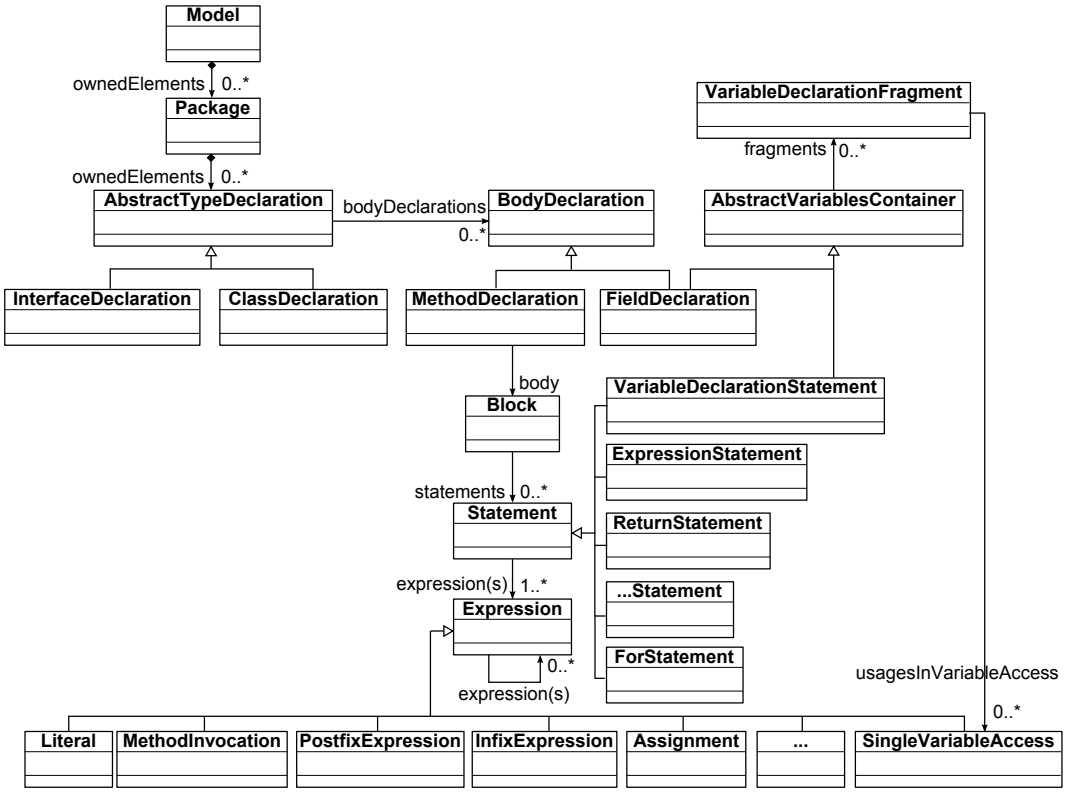

Figure 5.4: MoDisco Java metamodel excerpt

5.6 Business Term Identification

Business Term Identification is used to reduce the number of variables to analyse by filtering out those variables that do not represent business concepts. It takes as input the Java model and returns the variables hinting at business concepts.

This step has been implemented to work in a manual or automatic way. In particular, the user can select the variable to analyse, or he can rely on a set of heuristics we have defined. Such heuristics are based on the identification of variables appearing in conditions of conditional statements, input and output statements (e.g., *Scanner*, *BufferedReader*, *System.out*, etc.) as well as in expressions that employ literals (i.e., boolean, string, number) or mathematical operations. In addition, these heuristics can filter the discovered variables according to the imports declared in the container classes. In particular, this is useful for excluding variables defined in classes containing graphical imports (e.g., javax.swing.* , java.awt.*, etc.). Such variables are generally not related to the business, since they are used to define graphical objects.

As described in the Java metamodel (Fig. 5.4), different types of expression exist. Our heuristics focus on locating infix, postfix, prefix and literal expressions (Fig. 5.5) that contain mathematical operations.

infix expressions in	**example**
methodInvocation	setAge(getAge() + 1);
assignment	this.grass = grass * GRASS_GROW_PERCENTAGE;
variableDeclaration	int nextRow = row + rand.nextInt(3) - 1;
prefix expressions in	
statement	--foodLevel;
postfix expressions in	
statement	foodLevel--;
literal expressions in	
assignment	age = 0;

Figure 5.5: Business Term Identification

An infix expression can be embedded in method invocations, assignments or variable declarations (i.e., *FieldDeclarations* or *VariableDeclarationStatements* in Fig. 5.4). In particular, if a method invocation is a getter or setter, or it contains getters or setters, the variables accessed or modified are considered business relevant. On the other hand, for assignments and variable declarations, the variables that appear on the left and right hand sides of the assignment are considered relevant for the business, only if they are not used as indexes in loop conditions. Finally, in case, getter or setter methods are employed on the right hand side, the corresponding variables are retrieved from such methods.

Postfix and prefix expressions contain implicit mathematical operations applied on one variable. Since the variables within such expressions are often used as index

in loop statements and they do not hint at business terms, the heuristics implemented aim at identifying all variables appearing in postfix and prefix expressions not used as indexes. In particular, these heuristics check that a variable used in a condition of for, do-while and while statements is not used in a postfix or prefix expression within that loop statement.

Finally, all variables in assignments that contain on the right hand side a literal (e.g., boolean, string, number) are considered relevant for the business, since they represent variable initializations.

Infix	Animal.age, Animal.alive, Animal.births,Counter.count, Animal.births, Bird.grassNew, Grass.grass, Field.nextRow, Rabbit.grassNew, Field.nextCol, ...
Postfix	Animal.foodLevel
Literal	Animal.maxLitterSize, Simulator.grassNumber, Animal.breedingAge, Animal.breedingProbability, ...

Figure 5.6: Business Term Identification step for the running example

Figure 5.6 shows the result of the heuristics previously described concerning the running example. An excerpt of the variables in *infix expressions* are depicted on the upper row; while on the center and on the bottom rows, an excerpt of the variables in *postfix* and *literal expressions* are listed. Each variable in the figure appears together with its container class.

5.7 Business Rule Identification

Business Rule Identification, shown in Fig. 5.7, locates business rules in the code. It consists of two operations: Rule Discovery and Business Rule Model Extraction. This step takes as input the Java model and one or more variables [8] and it produces a model that contains the business rules related to such variable(s).

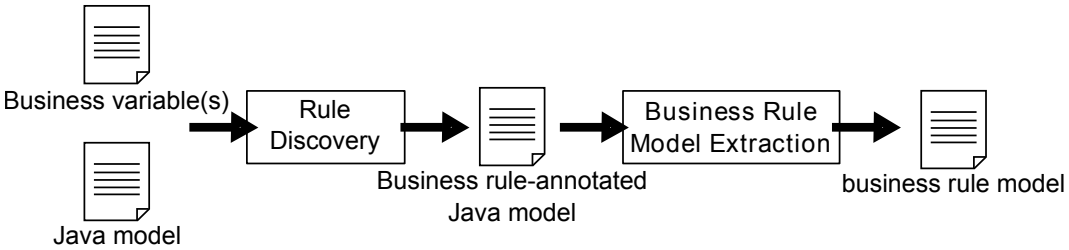

Figure 5.7: Business Rule Identification step

8. For the sake of comprehension, we describe the process for a single variable

5.7.1 Rule Discovery

Rule Discovery is used to identify and annotate the source code model elements (i.e., variable declarations, methods and statements) that compose the business rules. It is based on the static backward block slicing, according to the motivations presented in Sect. 3.3.3.

The input of this step is the Java model and a variable; whereas the output is the same Java model enriched with annotations on all statements, variable declarations and methods relevant for that variable. In particular, each annotation embeds information concerning a unique identifier as well as the granularity index and the type of relation with the variable analysed.

The identifier is composed by the name of the variable plus the rule number. In particular, this number is a value between zero and the number of references of the variable (i.e., usagesInVariableAccess in Fig.5.4) plus the number of invocations of the method that contains that variable. The name of the variable is used to identify statements used by two or more rules related to different variables. On the other hand the rule number identifies different rules related to the same variable.

On the contrary, the granularity index is the position of a method containing one of the statements relevant to the analysed variable in a possible execution path. In particular, such execution path is composed by the ordered set of methods crossed in a given program from an entry execution point (i.e., method main, etc.) to the statement that actually modifies the sliced variable. This ordered set of methods is defined as *granularity set*.

Finally, the Rule Discovery is applied on statements, variable declarations and methods. For each of them, we define different types of annotation (Fig. 5.8).

Type	**Annotations**
Statement	*rule, context*
Variable Declaration	*sliced-variable, related-variable*
Method	*related, reachable*

Figure 5.8: Business rule annotations

- Statement. It can be annotated as *rule* or *context*. In particular, all statements that allow passing from a method in the *granularity set* to another one in the same set are annotated as *rule*. On the contrary, a statement is marked as *context* if it is a conditional or loop statement that contains a *rule* statement, a *context* statement or a variable declaration statement marked as *related-variable*.
- Variable declaration. Two types of relations are defined for variable declarations: *sliced-variable* or *related-variable*. The former represents variable

declarations that contain the variable analysed. The latter represents variable declarations that contain references used inside *context* or *rule* statements.

– Method. It can be annotated as *related* or *reachable*. The former represents methods containing a *rule* statement. The latter defines methods having one of their invocations in either a *rule* or *context* statement or in another *reachable* method.

The generated annotations can be visualized by the user if desired. In particular, using MoDisco the annotated model can be transformed back into a Java application where all generated annotations will appear as comments.

The process to identify a business rule is composed by four phases, that are depicted below.

– The first phase locates the variable declaration of the selected variable. For each of its references (*SingleAccessVariables* in Fig. 5.4) the container statement is retrieved and it is annotated as *rule* with granularity zero.

– In the second phase, all methods invocations and references of other variables in the *rule* statement are respectively used to annotate the corresponding method declarations as *reachable* and the corresponding variable declarations as *related-variable*. In addition, in case the *rule* statement is contained within conditional or loop statements, such statements are marked as *context*.
This process is also applied to the method invocations, variable references, conditional and loop statements concerning the *context* statements.

– In the third phase, for each method annotated as *reachable*, all method invocations in it are collected and the corresponding method declarations are annotated as *reachable*. This step is repeated until no more *reachable* method declarations are found.

– In the fourth phase, the method containing the *rule* statement is annotated as *related* with a granularity index equal to the one of the *rule* statement; then its method invocations are retrieved. For each method invocation, the container statement is located and it is annotated as *rule* with granularity index increased by one. The phase two, three and four are repeated until the method that has triggered the execution of the program is found.

An example of slicing is shown in Fig. 5.9. It concerns the running example previously described (Sect. 5.3).

Line 9 contains the *rule* statement of the slicing variable *foodLevel* for the granularity zero. The if condition at line 7 is annotated as *context* since it contains the *rule* statement. The statement at line 6 is annotated as *related-variable* since the declared variable is used inside the *context* statement at line 7. The conditional statement at line 5 is annotated as *context* since it contains both *context* and *related-variable* statements. The statement at lines 4-1 follow the same logic already described.

The method *findFood* is marked as *related* since it contains a *rule* statement. It is

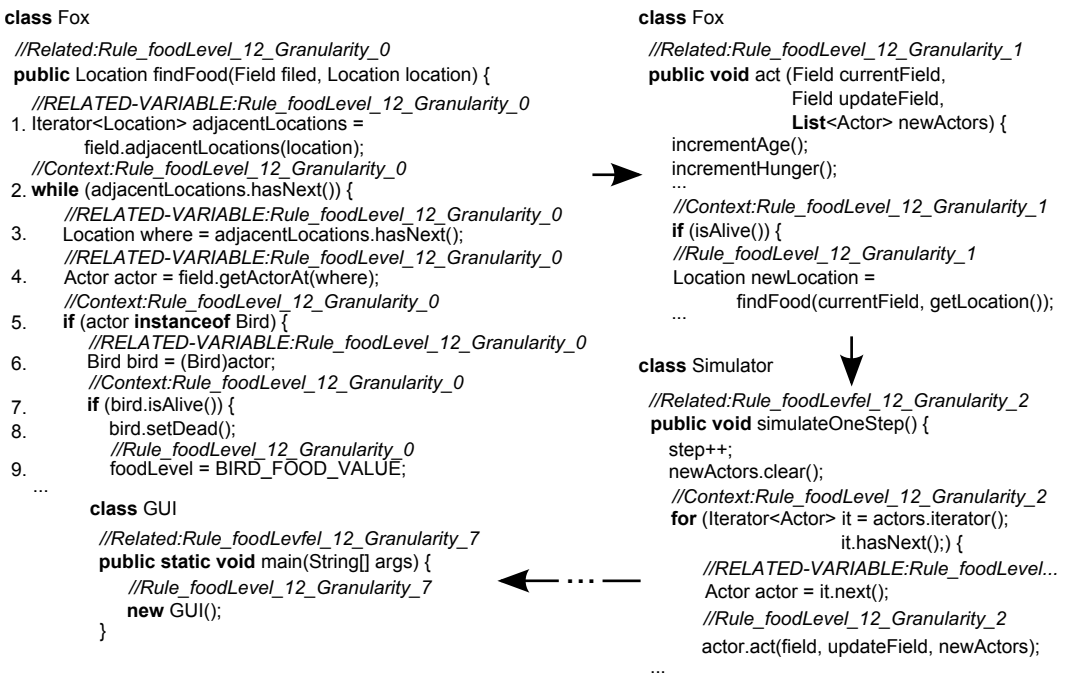

Figure 5.9: Example of Rule Discovery for the variable foodLevel

invoked by the method *act*, that has granularity index equal to 1, since it is one step far from the method that modifies the variable *foodLevel*. In turn, the method *act* is invoked in *simulateOneStep*, that contains elements with granularity index equal to 2. The same logic is applied until Rule Discovery finds the method that triggers the execution (method *main* in this case).

5.7.2 Business Rule Model Extraction

The goal of the Business Rule Model Extraction step is to provide an internal representation of the business rule identified by extracting from the Java model only those entities that have been annotated in the Rule Discovery step. The input of this transformation is the annotated Java model, while the output is a model that contains all business rules discovered. Such model represents the internal format of the rules.

The output model conforms to the metamodel shown in Fig. 5.10. The root of such metamodel is the class *Model* that collects rules. Each *Rule* represents a method that contain a *rule* statement. It is identified by the name of the sliced variable (*var*), its rule number (*number*) and the granularity index (*granularity*). In addition, a *Rule* is connected to the precedent and following rules by the references *next* and *prev*. Finally, a *Rule* has one *RuleStatement* and one *RelatedMethod*. In addition, it can have zero or more *ContextStatements*, *ReachableMethods* and *RelatedVariables*.

Finally, *RuleStatements*, *ContextStatements*, *ReachableMethods*, *RelatedVariables* and *RelatedMethods* are subclasses of *Trace*. Such class is used to keep the

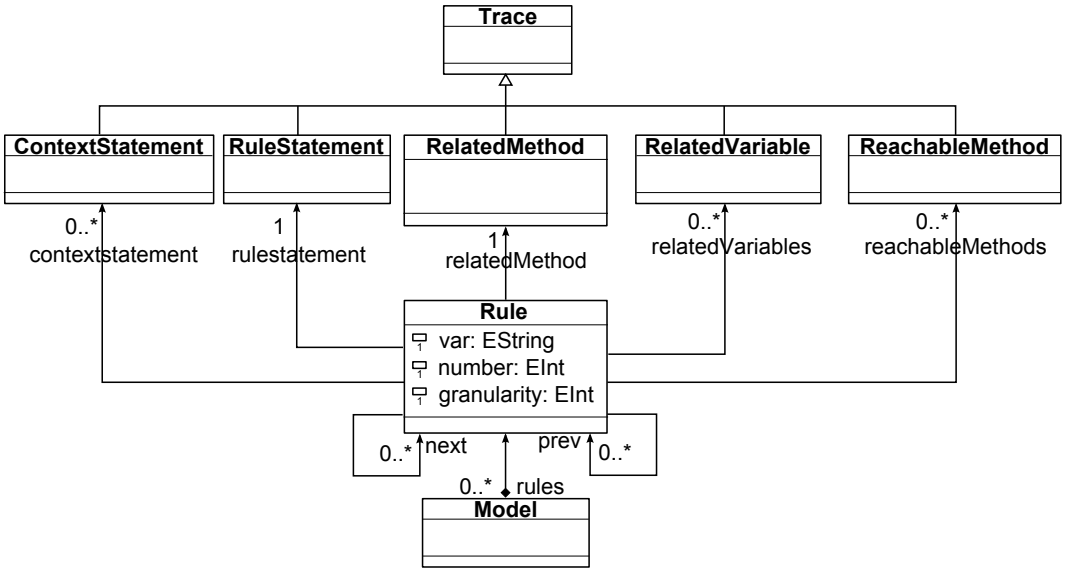

Figure 5.10: Business rule metamodel

links between the Java model elements and the internal representation of the identified rules (rule traceability).

The mappings to convert the identified business rules in the annotated Java model to the Business Rule model are defined between the classes of the two corresponding metamodels. Such mappings are shown in Fig. 5.11.

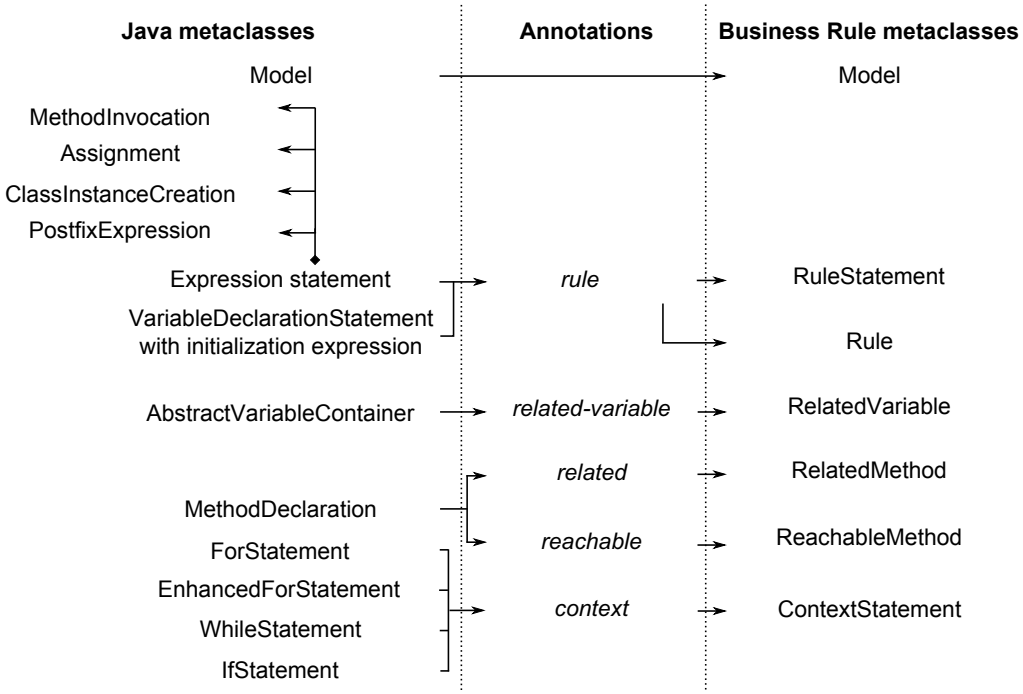

Figure 5.11: Java-2-Business Rule Model mapping rules

The class *Model* of the Java metamodel (Fig. 5.4) is mapped to the equivalent class of the Business Rule metamodel. Each statement annotated as *rule* is used to generate a *Rule* in the target model, where the name of the variable, the number

and the granularity of the rule are derived from the annotation. In addition, each *rule* statement is mapped to a *RuleStatement* in the Business Rule model. Such a statement is an instance of a sub-set of either the *VariableDeclarationStatement* or *ExpressionStatement* class in the Java metamodel. In particular, a *rule* statement can be either a *VariableDeclarationStatement* with an initialization expression or an *ExpressionStatement* that contains a *MethodInvocation*, *Assignment*, *ClassInstanceCreation* or *PostfixExpression*.

Each variable annotated as *related-variable* in the Java model is mapped to a *RelatedVariable* in the Business Rule model. Since a *related-variable* can be an attribute of a given class, a variable declared within a method or a variable passed as argument of a method; a mapping is defined between the *AbstractVariableContainer* class (that regroups the aforementioned types of variables) and the *RelatedVariable* class.

The methods in the Java model annotated as *related* or *reachable* are respectively mapped to *RelatedMethods* and *ReachableMethods* in the target model. Therefore, two mappings are defined between the class *MethodDeclaration* in the Java metamodel to the classes *RelatedMethods* and *ReachableMethods* in the Business Rule metamodel.

Finally, the statements annotated as *context* in the Java model are mapped to *ContextStatements* in the target model. Such statements are instances of all Java model classes statements that embed logic conditions like *ForStatements*, *EnhancedForStatements*, *WhileStatements*, *IfConditions*, etc.

5.8 Business Rule Representation

Business Rule Representation, shown in Fig. 5.12, provides understandable textual and graphical representations that describe the discovered business rules.

This step is composed by two sub-steps: Vocabulary Extraction and Visualization. The former is an optional step, and it provides default verbalizations for variables, methods and classes related to the business. On the other hand, Visualization provides textual and graph artifacts concerning the identified business rules.

5.8.1 Vocabulary extraction

Vocabulary Extraction is an optional operation and it is used to extract the vocabulary of the application. It can be a manual or assisted operation, in particular in the first case, the user defines verbalizations for variable, method and class declarations; while in the second case, default verbalizations are provided for all variable, method and class declarations in the source code. Such default verbalizations consist in splitting the names of classes, variables and methods according to the com-

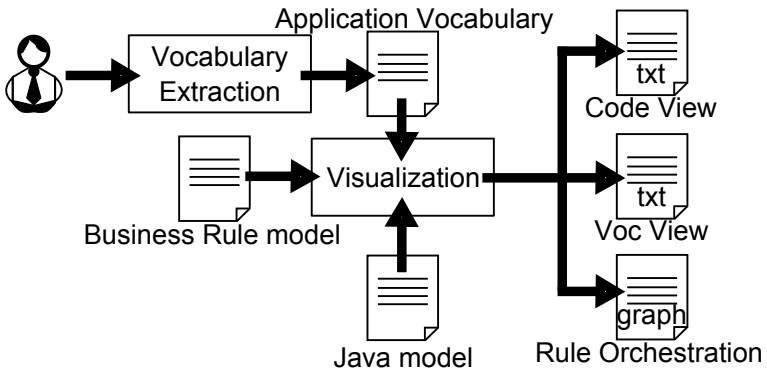

Figure 5.12: Business Rule Representation step

mon way to define them in Java. Therefore, the names of static or final method and variable are defined according to this mapping: *ABC_DEF ->ABC DEF*; while for the other cases we provide this fallback mapping: *abcDef ->abc Def*. Figure 5.13 shows an excerpt of default verbalizations for the running example.

```
<vocabulary:Model ...>
...
<entries type:Variable name="GRASS_GROW_PERCENTAGE"
                              label="grass grow percentage" class="Grass" package="Simulator"/>
<entries type:Method name="setGrass" label="set Grass" class="Grass" package="Simulator"/>
<entries type:Method name="Grass" label="create grass" class="Grass" package="Simulator"/>
...
</vocabulary:Model>
```

Figure 5.13: Running example vocabulary

The vocabulary model conforms to the metamodel presented in Fig. 5.14. The root element of this metamodel is the entity *Model*, that contains a list of *entries*. An *Entry* is defined by the attributes *class*, *package* and *label*, where the first two locate the position of the source code element and the latter contains the corresponding verbalization. An *Entry* is the super-class of *Variable*, *Method* and *Class* entities. A *Variable* is defined by its *name* and, if defined within a method, by the name of the container method. Finally, *Methods* and *Classes* are identified only by their names.

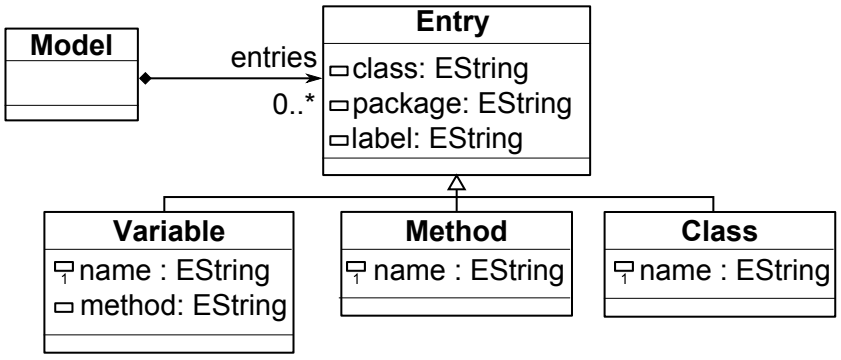

Figure 5.14: Vocabulary metamodel

5.8.2 Visualization

The visualization step provides comprehensible artifacts for the identified business rules and their relations. Its inputs are the Java model, the Business Rule model and optionally the Vocabulary model if defined. The output are text and graph artifacts that represent the external representation of the identified business rules.

Visualization is composed by two steps that respectively generate text and graph for the extracted rules. They are described in the following.

Textual Visualization

All *Rules* in the Business Rule model are collected and ordered according to the variable(s) they are related to. Initially, for each *Rule*, the *RelatedMethod* is retrieved and the contained *RuleStatement* and *ContextStatement(s)* are located in the code and ordered thanks to the traceability links kept by the class *Trace* in the Business Rule metamodel (Fig. 5.10). Optionally, the same process can be applied to *RelatedVariables* and *ReachableMethods*, if the user wants to have a more complete understanding of the rule. Finally, code-based representations are calculated for *RuleStatements*, *ContextStatements*, *RelatedVariables*, *RelatedMethods* and *ReachableMethods*. In addition, if the application vocabulary has been defined, vocabulary-based representations are calculated as well.

codeView	**vocView**
Rule var: alive, number: 1, granularity: 2	**Rule** var: alive, number: 1, granularity: 2
RelatedMethod: Bird.act **RuleStatement:** incrementAge()	**RelatedMethod:** Bird act **RuleStatement:** Animal increment Age
Rule var: alive, number: 1, granularity: 1	**Rule** var: alive, number: 1, granularity: 1
RelatedMethod: Animal.incrementAge **ContextStatement:** if (getAge() > maxAge) **RuleStatement:** setDead()	**RelatedMethod:** Animal increment Age **ContextStatement: if** Animal get Age **>** max Age of Animal **then** **RuleStatement:** Animal set Dead
RelatedVariable: int maxAge = 10 **ReachableMethod:** Animal.getAge()	**RelatedVariable:** max Age of Animal = 10 **ReachableMethod:** Animal get Age
Rule var: alive, number: 1, granularity: 0	**Rule** var: alive, number: 1, granularity: 0
Relatedmethod: Animal.setDead **RuleStatement:** alive = false	**Relatedmethod:** Animal set Dead **RuleStatement:** alive of Animal = false

Figure 5.15: Example of textual outputs for the rule alive

Code and vocabulary-based representations are obtained mixing hard-coded translations of the Java constructs with either the identifiers of variables, methods and classes that appear in the rules, or with the corresponding descriptions defined in the application vocabulary. In particular, concerning *RuleStatements*, *ContextStatements* and *RelatedVariables*, all variable, method and class identifiers in them are translated; on the other hand, with respect to *RelatedMethods* and *ReachableMethods* only their names are translated.

In Fig. 5.15 a code and vocabulary-based representations concerning a rule related to the variable *alive* are shown. In particular, it represents one of the possible causes of death for birds (old age). Each box in the figure represents a method within a possible execution path (i.e., granularity set) relevant for the variable *alive*. The granularity index is used to order such methods.

Graphical Visualization

Graphical Visualization is used to transform the Business Rule model into a graph in order to highlight relations (i.e., orchestration) between different rules. Such relations are defined on *RuleStatements* shared between different rules.

The relations between the rules concerning the different causes of death (hunted by fox or hunter as well as because of starvation, old age or overcrowding) for a bird are shown in Fig.5.16.

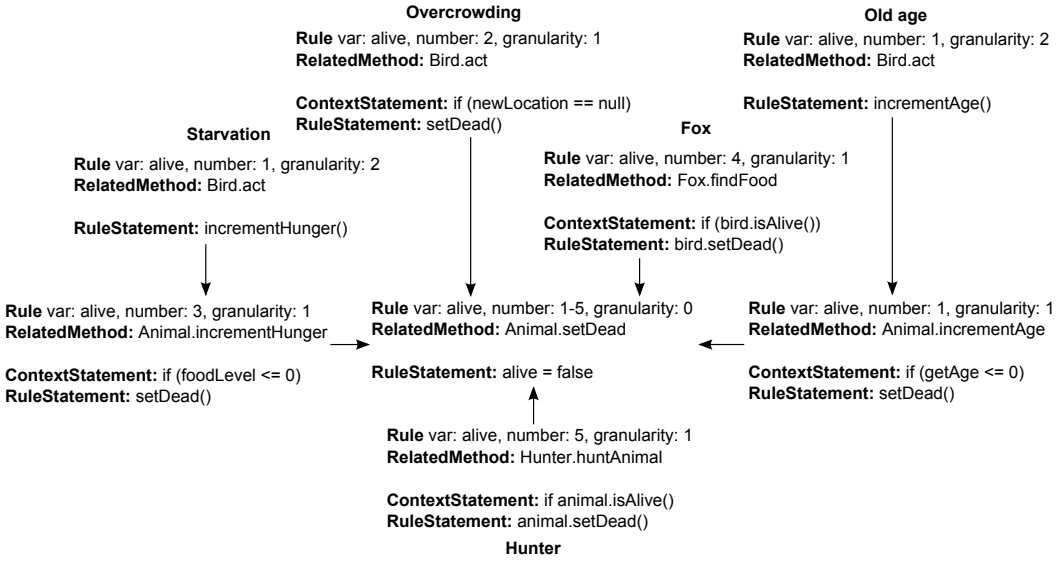

Figure 5.16: Orchestration of the rules for the variable *alive*

5.9 Optimization

The BREX process is generally a time-consuming activity, since the source code must be traversed in order to recover the embedded business rules. This is specially true when dealing with systems composed by millions of lines of code. As a consequence, in order to optimize the BREX process for large systems, the source code to analyse should be reduced.

In Java applications, the source code is wrapped in statements, that are in turn organized in packages, classes and methods. In particular, a method is a collection of statements that are grouped together to perform an operation. Such operation can be related or not to the business.

Therefore, in order to improve the efficiency of our BREX process, we have provided the framework with an additional slicing operation performed at the method-level. Such operation is executed before the Rule Discovery step (Sect. 5.7.1) and aims at reducing the size of the input model when dealing with large systems. In particular, given one or more variables to analyse, such slicing finds recursively the methods relevant for these variables. All these methods are kept in the reduced Java model and they will be annotated as *related* or *reachable* during the Rule Discovery, while the other ones are discarded. Finally, such model is passed to the Rule Discovery step.

```
helper def : getInsideInvokedStarting : Sequence(JAVA!AbstractMethodDeclaration) =
    let amds : Sequence(JAVA!AbstractMethodDeclaration) =
        thisModule.ALL_ABSTRACT_METHOD_INVOCATION
                    ->select(ami |
                            thisModule.getSelectedAbstractMethodDeclaration
                                ->union(thisModule.getInvocatingMethods(
                                        thisModule.getSelectedAbstractMethodDeclaration,
                                        thisModule.getSelectedAbstractMethodDeclaration))
                                ->exists(amd | ami.getContainingMethod = amd)
                        )->collect(ami | ami.method)->flatten()->asSet()->asSequence() in
    amds->union(thisModule.getInsideInvokedRec(amds,amds)->select(amd | amd.proxy = false))
            ->asSequence()->asSet();

helper def : getInsideInvokedRec(input : Sequence(JAVA!AbstractMethodDeclaration),
                                 output : Sequence(JAVA!AbstractMethodDeclaration))
                                 : Sequence(JAVA!AbstractMethodDeclaration) =
    let amds : Sequence(JAVA!AbstractMethodDeclaration) =
        thisModule.ALL_ABSTRACT_METHOD_INVOCATION
                    ->select(ami | input->exists(i | i = ami.getContainingMethod))
                        ->collect(ami | ami.method)->flatten()->asSet()->asSequence() in
    if amds->forAll(i | output->exists(o | o = i)) then
        output
    else
        thisModule.getInsideInvokedRec(
            amds->reject(amd | output->exists(o | o = amd)),
            output->including(amds)->flatten()->asSet()->asSequence()
        )
    endif;

helper def : getAllMethods : Sequence(JAVA!AbstractMethodDeclaration) =
    thisModule.getInsideInvokedStarting->union(thisModule.getInvocatingMethodsStarting)
                ->union(thisModule.getFieldDeclarationMethods)
                    ->flatten()->asSet()->asSequence();

helper context JAVA!AbstractMethodDeclaration def : isInAllMethods : Boolean =
    thisModule.getAllMethods->exists(amd | amd = self);

rule AbstractMethodDeclaration {
    from
        s : JAVA!AbstractMethodDeclaration (not s.isInAllMethods and s.proxy = false)
    to
        drop
}
```

Figure 5.17: Excerpt of the slicing for methods

In Fig. 5.17, an excerpt of the slicing for methods is shown. The rule drops from the Java model all *AbstractMethodDeclarations* that are not proxies [9] and not in the set calculated by the ATL helper *getAllMethods*. Such helper calculates recursively (*getInsideInvokeStarting*, *getInsideInvokeRec*) the methods related to the selected

9. A proxy method in the Java model is a method that is part of a predefined Java package

variables.

5.10 Evaluation

In order to validate our method we have performed two evaluations. The first evaluation has been focused on checking that the business rules returned at the end of the extraction process for the running example coincide with the ones that we discovered by manual inspection. For the running example, we were able to generate both graphical and textual representations of all the identified rules, facilitating this way the comprehension of the application.

The second evaluation has been based on testing our framework on a larger case study provided by IBM. Given the large model representing the case study, a special environment has been set to perform the business rule extraction process.

Our framework has been applied on IBM Rational Programming Patterns (RPP), that is an integrated environment under Eclipse for developing and maintaining COBOL Pacbase [10] applications. In addition, since RPP is designed on MDE principles, we have been able to benefit from such environment.

RPP embeds a metamodel of the system. Therefore, since a metamodel contains relevant concepts with respect to the domain; we have defined a new heuristic for the Business Term Identification step. Such heuristic extracts the attributes in the RPP metamodel entities and retrieves the corresponding variables in the RPP system. With the new heuristic, we have been able to recover 476 variables, that have been used to drive the remaining steps of the business rule extraction process.

Finally, we have been able to easily integrate this new heuristics within our extraction process thanks to the modularity of our framework.

The IBM case study has allowed us to analyse the efficiency of our framework. In particular, the Rule Discovery step (Sect. 5.7.1) has been optimized to cope with large systems. In particular, we have added an operation (Sect. 5.9) that removes from the Java model the methods not related to the business variable(s) analysed.

5.11 Prototype

A prototype [11] has been developed in collaboration with IBM, that has provided a use case to validate the framework. The framework integrates different tools of the MDE ecosystem. In addition, MDE has facilitated the interoperability among such tools.

Model Discovery relies on the Java discovery provided by MoDisco [12], that is

10. http://www-01.ibm.com/software/awdtools/vapacbase/
11. The source code of the prototype is available at http://docatlanmod.emn.fr/BrexJava/
12. http://www.eclipse.org/MoDisco/

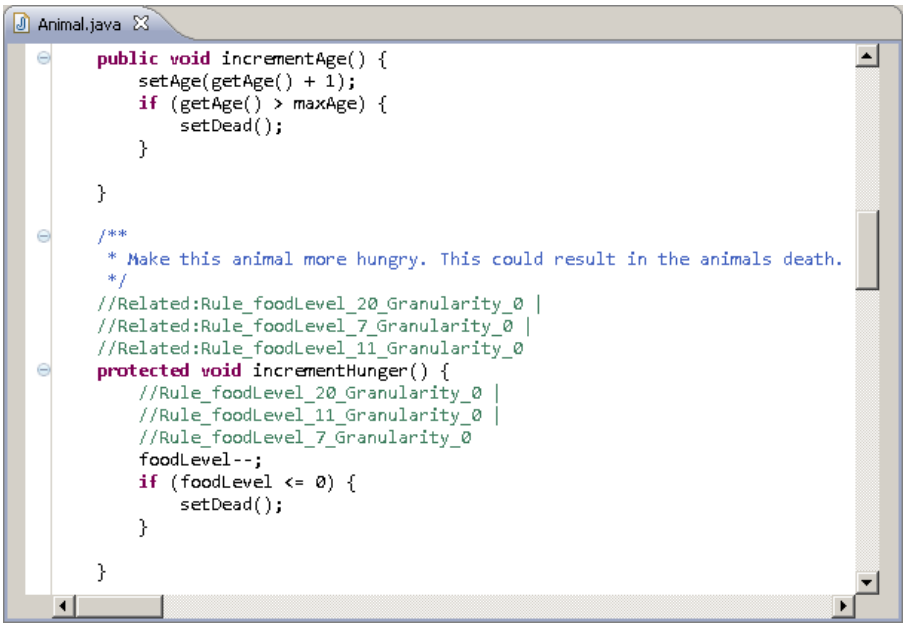

Figure 5.18: Excerpt of source code with business rule annotations

a generic and extensible framework dedicate to reverse engineering process in a model-driven context. In particular, MoDisco has been used to generate a low-level model-representation of the source code that composes the Java application. Such model has a one-to-one correspondence with the Java syntax, therefore no information is loss during the Model Discovery step. In addition, MoDisco has been used to regenerate the source code from the Java model enriched with business rule annotations (Fig. 5.18). Such annotations are represented as line comments in the obtained source code.

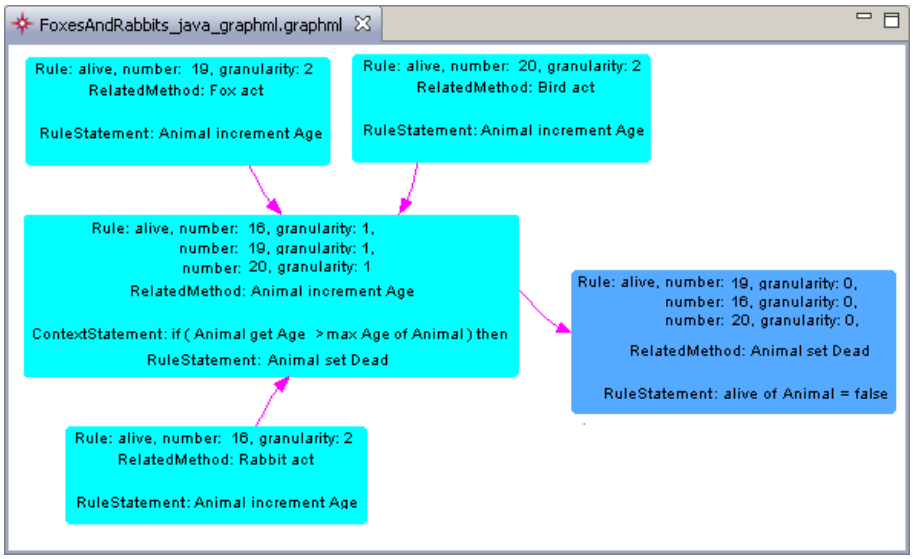

Figure 5.19: Excerpt of graph-based representation of rules

On the other hand, Business Term Identification, Business Rule Identification and Business Rule Representation rely on the ATL Transformation Language (ATL)

[14]. It has been used to implement model manipulation operations required by the three aforementioned steps.

Another tool used in our framework is Portolan [13], that applies model-driven techniques on cartography processes bridging the gap between model-driven and visualization tools. In particular, it has been used to represent graphically the extracted rules, easing in this way their understanding (Fig. 5.19).

Finally, our framework has been deployed as an Eclipse plug-in and implemented on top of the Eclipse EMF modeling framework. In particular, EMF has been used to define metamodels and access the corresponding models.

13. http://code.google.com/a/eclipselabs.org/p/portolan/

6

Business rule extraction for relational databases

6.1 Motivation

Relational databases have been playing a key role in most organizations for the past 40 years. They store relevant information according to a schema that defines their internal data structures and relations. The integrity of this information is ensured by constraints applied on the database schema. Such constraints represent part of the business logic of the organization.

Although databases are generally the most stable part of a system and they do not evolve as fast as the rest of the system does; extracting the embedded business rules can bring different benefits. For instance, they can be used to discovery possible inconsistencies with respect to the logic embedded in other system components or they can ease the migration towards different database implementations overcoming the variations that each SQL vendor language has with the standard.

Different database languages can be defined to embed business rules in database. Without loss of genericity, we particularize our solution for Oracle SQL and PL/SQL in order to derive rules from declarative and operational constraints (Sect. 3.3.4). The obtained rules are expressed in OCL and defined on the conceptual model that represents the database implementation. Such model is expressed in UML.

In the following, we introduce the basic concepts for SQL and PL/SQL as well as OCL. Finally, we describe the business rule extraction process applied to Relational Databases Management Systems (RDBMSs).

6.2 SQL, PL/SQL and OCL basic concepts

This section introduces SQL, PL/SQL and OCL. In particular, it focuses on the constructs that can be used to express business rules at database and model-level.

6.2.1 SQL

SQL (Structured Query Language [38]) is a standard programming language designed for accessing and modifying data and data structures in RDBMSs. In particular, data are manipulated using SELECT, UPDATE, DELETE and INSERT queries/statements; while the corresponding data structures are created using CREATE TABLE, CREATE VIEW and CREATE DB and modified using DROP, ALTER statements.

The integrity constraints in SQL can be defined in table and column declarations. Such declaratives constraints are used to specify the columns that identify a given table (i.e., PRIMARY KEY) or references to other column tables (i.e., FOREIGN KEY). In addition, other constraints can be defined on the value that a column can take (i.e., UNIQUE, NOT NULL, CHECK).

Finally, these constraints can be specified within CREATE TABLE statements when the table is created or inside ALTER TABLE statements when the table is modified.

6.2.2 PL/SQL

PL/SQL is an extension language for SQL that has been created for supplementing SQL with the common programming-language features (e.g., variable definitions, conditional statements, loop statements, etc.). It is used to define stored procedures.

A stored procedure in PL/SQL, depicted in Fig. 6.1, consists of three sections: an optional declaration section, an execution section and finally an optional exception handling section.

```
DECLARE
    variable and constant declarations
BEGIN
    ...
    statements
    ...
EXCEPTION
    exception handlers
END
```

Figure 6.1: PL/SQL block structure

The declaration section is identified by the keyword *DECLARE* and it is used

to define either variables or constants used in the PL/SQL code. The execution section is wrapped between the keywords *BEGIN* and *END* and contains PL/SQL statements, that may rely on SQL queries. The exception handling section deals with run-time errors thrown by the execution section. It is defined by the keyword *EXCEPTION*.

A particular procedure in PL/SQL is called trigger. In the following we describe the basic structures for triggers in PL/SQL in Oracle.

Triggers

A trigger [50] is a procedure that is stored in the database and it is implicitly fired when a given event happens. It embeds procedural code expressed in PL/SQL. In Oracle, such procedure (Fig. 6.2) is created using the clause *CREATE TRIGGER*. Optionally, the clause *OR REPLACE* can be used to drop a Trigger with the same name (i.e., if it is already been defined) and create a new Trigger.

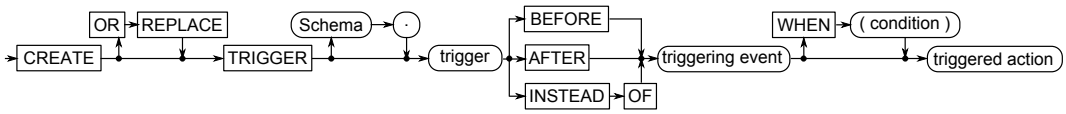

Figure 6.2: A trigger structure in Oracle

A trigger is composed by at least two parts: the triggering event and the triggered action. The triggering event is an SQL statement (i.e. INSERT, UPDATE, etc.), database or user event that causes a trigger to fire. A triggered action is a PL/SQL [51] block that contains procedural code and possibly SQL statements to be run when the triggering event occur.

An additional element that can be part of a trigger is the restriction condition (i.e. *WHEN* clause in the figure). It is used to specify a boolean expression related to the event clause and it must be true in order to fire the trigger.

Finally, triggers can be executed instead of, before or after performing the triggering event.

6.2.3 OCL

OCL is used to express constraints at model-level that cannot be defined using the visual formalisms provided by UML. In particular, it offers predefined mechanisms for retrieving the values of the attributes of an object, for navigating through a set of related objects, for iterating through collection of objects (e.g., by means of the forAll, exist and select iterators) and so forth.

As part of the language, a standard library including a predefined set of types and a list of predefined operations that can be applied on those types is also provided. The types can be primitive (i.e., Integer, Real, Boolean and String) or collection

types (i.e., Set, Bag, OrderedSet and Sequence). Some examples of operations provided for those types are depicted in Table 6.1.

Name	Description
allInstances()	returns a *Set* containing all currently existing instances of the type *self*
asSet()	returns a *Set* containing all elements of *self*
product(c2 : Collection(T2))	returns a *Set* of *Tuples* representing the Cartesian product of *self* with *c2*
forAll(it | body)	returns a *Boolean* value stating whether body evaluates to true for all elements of the source collection
exists(it | body)	returns a *Boolean* value stating whether body evaluates to true for at least one element of the source collection
collect(it | body)	returns a collection of elements which results in applying *body* to each element of the source collection
select(it | body)	returns the subset of the source collection for which *body* evaluates to true
iterate(it; var_acc = init_exp | body)	returns the value of the accumulator variable once the last iteration has been performed

Table 6.1: Predefined OCL operations

All these operations as well as the mechanisms previously described can be used in the definition of OCL constraints/invariants (Fig. 6.3) and derivation rules.

An OCL constraint is always defined by a context and a body. They respectively contain the type whereon the constraint is applied (i.e., *ContextualClass*) and the OCL expression that defines the behavior of the constraint. In addition, the body of a constraint is always a boolean expression that must be satisfied by all instances of the context type. Finally, in order to refer to the contextual instance inside an OCL expression, the reserved word *self* is used.

context ContextualClass
inv: OCL boolean expression

Figure 6.3: example of an OCL invariant

A derivation rule, shown in Fig. 6.4, constrains the value of a derived element [52]. They are defined by a context, a body and a return type. The context contains the derived type; the body is composed by a query expression defined over the classes of the model; and finally the return type is the type of the instances obtained by the query expression.

context derivedType : return type
derive: OCL query expression

Figure 6.4: example of an OCL derivation rule

6.3 Running example

In order to illustrate our framework [1], we have created a small human resources sample database (shown in Fig. 6.5), composed by three tables, a view and a trigger.

```
CREATE TABLE Employee (
    employee_id      INTEGER PRIMARY KEY,
    first_name       VARCHAR2(20),
    last_name        VARCHAR2(25) NOT NULL,
    salary           NUMBER(8,2) CHECK (salary > 0),
    department_id    INTEGER NOT NULL,
    job_id           INTEGER NOT NULL,
    CONSTRAINT emp_job_fk FOREIGN KEY (job_id)
    REFERENCES Job(job_id),

    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
    REFERENCES Department(department_id)
) ;
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT ON Employee
FOR EACH ROW
DECLARE
  Minsal          NUMBER;
  Maxsal          NUMBER;
BEGIN
SELECT min_salary, max_salary INTO Minsal, Maxsal
FROM Job
WHERE job_id = :NEW.job_id;
IF (:NEW.salary < Minsal OR :NEW.salary > Maxsal) THEN
    RAISE_APPLICATION_ERROR(-20300, 'Salary out of range');
END IF;
END;
```

```
CREATE TABLE Department (
    department_id      INTEGER PRIMARY KEY,
    department_name    VARCHAR(30) NOT NULL,
    manager_id         INTEGER UNIQUE NOT NULL,
    CONSTRAINT dept_mgr_fk FOREIGN KEY (manager_id)
    REFERENCES Employee(employee_id)
);
CREATE TABLE Job (
    job_id          INTEGER PRIMARY KEY,
    job_title       VARCHAR(35) NOT NULL,
    min_salary      NUMBER(6),
    max_salary      NUMBER(6)
);
CREATE VIEW EmployeeInDepartment AS
SELECT first_name, last_name, department_name
FROM Employee e JOIN Department d
ON e.department_id = d.department_id;
```

Figure 6.5: Human resource database sample

The database schema contains the tables of *Employees*, *Departments* and *Jobs*. An *Employee* is defined by a unique *employee ID* and has a *first name*, *last name* and *salary*. An *Employee* has one *Job* and belongs to one *Department*. A *Job* is depicted by an unique *job ID* and the *title* describing that job as well as the corresponding *maximum* and *minimum* salary for that job. Finally, a *Department* is described by an unique *department ID*, its *name* and the manager (i.e., an employee) leading it.

In addition, the view *EmployeesInDepartment* is defined on the tables *Employee* and *Department*. In particular, it returns the name of employees and the corresponding department names where they are assigned.

Finally, the trigger is fired when a new employee is inserted in the database. It checks that an employee's salary is in the range defined by the minimum and maximum salary for the job he has been hired. In case the salary is outside this range, an exception is raised by the trigger.

1. The input and output of our framework for the running example can be found at http://docatlanmod.emn.fr/BrexDBExample/

6.3.1 Rules modeling the application

Different business rules can be extracted out of this small human resources sample database. They concern respectively the integrity constraints specified within the table definitions and in the trigger. A manual inspection of such constrains reveals that:

- the last name of a employee cannot be null
- the salary of a employee must be greater than zero
- a new employee cannot have a salary greater than the maximum salary defined for a given job
- each employee must be assigned to a department
- each employee has to have a job
- a department must have a name
- a department has to have a manager
- a job must have a name

In the following, we describe how to extract the equivalent OCL invariants from the integrity constraints embedded in the database.

6.4 Framework description

The framework (Fig. 6.6) for extracting business rules from database implementations is an instantiation of the conceptual framework presented in Chap. 3. Hence, it is composed by four steps respectively Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation.

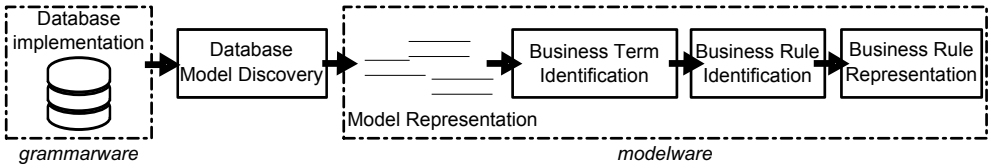

Figure 6.6: Database Business Rule Extraction framework

Model Discovery transforms the database in a model-based representation. This step is achieved by using Xtext [**?**], an open-source framework to develop programming and domain-specific languages. In particular, Xtext is used to define a parser for SQL and PL/SQL in order to generate from the database schema and triggers an equivalent model-based representation. Such representation has a one-to-one correspondence with the database implementation and as a consequence no information is lost between the grammarware and modelware.

Business Term Identification focuses on gathering business concepts and their relations from the structural elements of the database schema. Such concepts are expressed in UML. This step is composed by two operations, the first one creates

the classes and associations corresponding to the database tables and their relations; while the second one extends the obtained model by adding a set of derived classes to represent the database views.

Business Rule Identification analyses the database integrity constraints and extracts equivalent OCL invariants that complete the UML model. Sub-steps of this method cover the declarative constraints (CHECK, UNIQUE,...) and the analysis of triggers [2] since, beyond other applications, triggers can also be used to enforce complex integrity constraints.

Key elements in the Business Rule Identification step are the SQL-to-OCL and PL/SQL-to-OCL transformations. In particular, they are used to map SQL and PL/SQL constructs to OCL. Finally, since triggers often merge SQL and procedural code, SQL-to-OCL is included in PL/SQL-to-OCL.

Business Rule Representation provides text and graph representations of the identified rules. The formats of such representations are implicitly defined in the previous two steps of the framework. In particular, the Business Term Identification represents the identified concepts within a UML model, that is based on graphic notation techniques to create visual representations. On the other hand, the Business Rule Identification step presents the identified database constraints in OCL, that is a textual language.

In the following sections Model Discovery, Business Term Identification, Business Rule Identification and Business Rule Representation are described in detail.

6.5 Model Discovery

Model Discovery extracts the model that represents the schema and triggers within the database. It is based on the facilities provided by Xtext [3], a model-based framework under Eclipse [4] that supports and eases the creation of domain-specific and programming languages. In particular, for each of such languages Xtext generates automatically the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks, static analysis (i.e., validation), the interpreter and the metamodel that represents the language.

A simplified version of the metamodel that represents the grammar to parse SQL and PL/SQL source code is shown in Fig.6.7. The root element is the class *Model* that contains a list of *AbstractDeclarations*. An *AbstractDeclaration* is defined by a name and it can be a table, view or trigger declaration.

2. The SQL standard also includes a CREATE ASSERTION statement to specify complex constraints but none of the major database vendors support it
3. http://www.eclipse.org/Xtext/
4. http://www.eclipse.org/

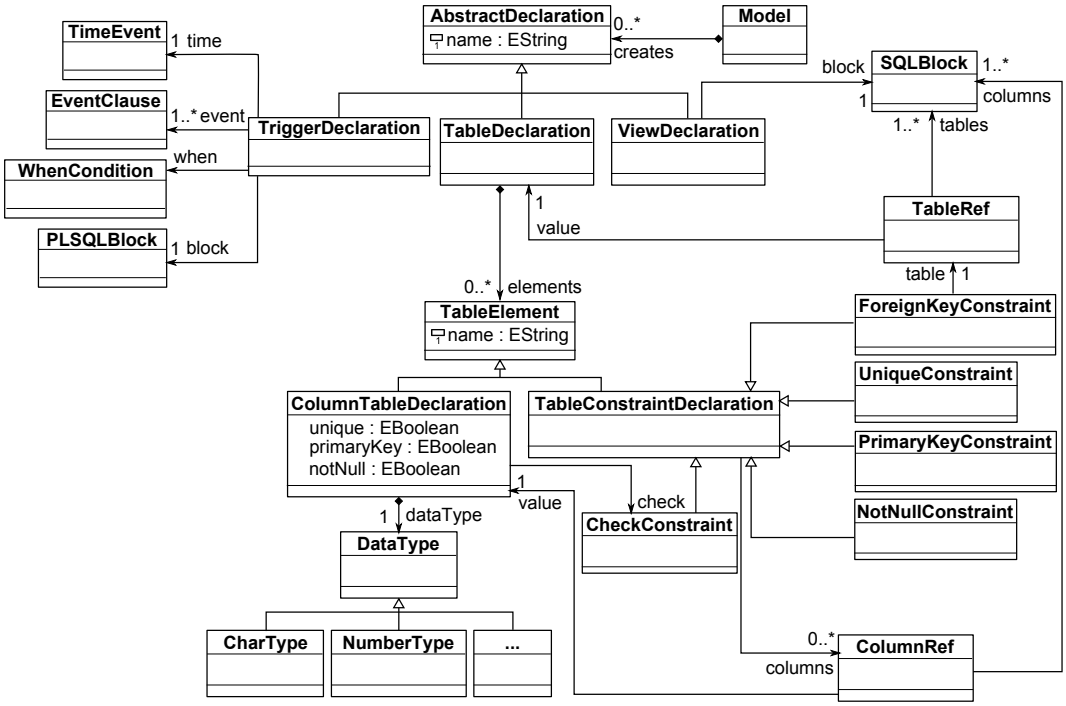

Figure 6.7: SQL and PL/SQL metamodel

A *TableDeclaration* is composed by a list of *TableElements*, that are respectively column table and constraint table declarations. A *ColumnTableDeclaration* is used to define a column and optionally a constraint such as unique, check, not null, primary key or foreign key on it. In addition, each column declaration is related to a data type allowed by the database. It can be *CharType*, *NumberType*, etc. On the contrary, *TableConstraintDeclarations* are used to define constraints that are not declared together with the column table declarations. *CheckConstraints* include expressions composed by literals, operators and references to the columns involved (*ColumnRef* class in the metamodel); *NotNullConstraints*, *UniqueConstraints* and*PrimaryKeyConstraints* contain only column references; finally *ForeignKeyConstraints* contain both references to columns and tables (*TableRef* class in the metamodel). They allow to retrieve the corresponding *ColumnTableDeclarations* and *TableDeclarations*.

A *ViewDeclaration* is composed by an *SQLBlock*, that includes SQL statements as well as *TableRefs* and *ColumnRefs* of the tables and columns contained in the statements.

Finally, a *TriggerDeclaration* is defined by a *TimeEvent*, an *EventClause*, a *PLSQLBlock* and optionally a *WhereCondition*. In particular, a *TimeEvent* defines if a trigger must be activated before, after or instead of a given database event. Such event is represented by the class *EventClause* and it can be an insert, update, delete SQL statement. On the contrary, the *WhereCondition* entity contains a boolean expression that must be true in order to execute the PL/SQL code within trigger. Such

code is contained in the *PLSQLBlock* entity.

6.6 Business Term Identification

Business Term Identification translates tables and views into an equivalent set of classes and associations in a UML class diagram.

As typically done in existing approaches, each table generates a class (or an association class) in the conceptual schema and table columns (except for foreign keys) are mapped into attributes of the corresponding class. Foreign keys are used to create associations between the classes, while the corresponding cardinalities can be calculated by performing SQL queries on the data stored in the database [53].

The type of the attributes depends on the type of the columns. Character data types (CHAR(n), VARCHAR2(n), etc.) are transformed to String types. Number and date-time data types (Integer, Float, Date, etc.) are transformed into their equivalent UML types: Integer, Real and Date. The Number(precision, scale) is transformed into an Integer data type when precision is zero and into a Real type otherwise. In addition, other Oracle data types can be represented by defining new data types in the UML model.

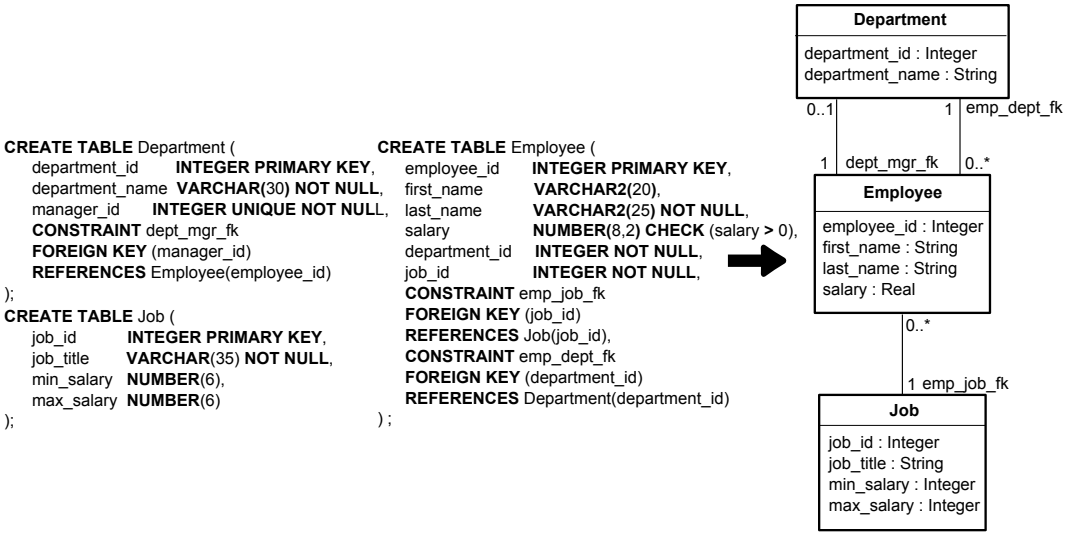

Figure 6.8: Database schema to conceptual schema

Figure 6.8 shows the translation from the database presented in Sect. 6.3 and the corresponding conceptual schema. Tables *Employee*, *Job* and *Department* are translated into the equivalent UML classes. An *Employee* in the UML model is defined by the attributes *employee ID*, *first name*, *last name* and *salary*. These attributes are derived from columns where no foreign key constraint is defined. The foreign keys are translated into UML associations; therefore an *Employee* has one *Job* (i.e., *emp_job_fk*) and belongs to one *Department* (i.e., *emp_dept_fk*). Accordingly to the previous mappings, a *Department* is described by an unique *department_id*, its

name and the manager leading it (i.e., *dept_mgr_fk*). Finally, a *Job* is depicted by an unique *job_id*, by its *title* and the corresponding *maximum* and *minimum* salaries for that job.

Besides this basic process, the Business Term Identification step also creates an additional class for each database view. In this case, the UML class is derived and has as attributes the names and corresponding types of the columns that are selected in the view definition. The derivation rule for the class is created by translating the SELECT query as described in the next section.

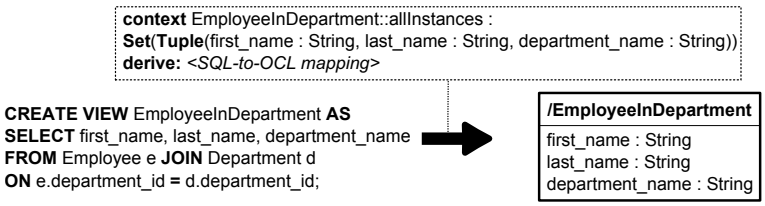

Figure 6.9: Mapping of database views

In Fig. 6.9, the mapping of a view is depicted. The view represents the projection of *Employees* (*first* and *last* names) per *Department*, that is identified by its *name*. The columns in the SELECT clause (i.e., *first_name*, *last_name* and *department_name*) become the attributes of the derived class; while their types (i.e., String in this case) are derived by mapping the built-in data type VARCHAR2 used to define those columns.

6.7 Business Rule Identification

Business Rule Identification aims at discovering the rules embedded in the database implementation. In particular, this step analyses the declarative constraints in table declarations and the operational constraints in triggers. In addition, since triggers often merge SQL statatements, SQL-to-OCL mappings are needed to derive OCL-equivalent constraints from the constraints coded within a trigger.

In the following, the extraction of declarative and operational constraints as well as the mapping between SQL and OCL are depicted.

6.7.1 Declarative constraints to OCL

This step covers the declarative integrity constraints specified within the table creation statement. The context for all these constraints is the class representing the table in the UML model.

Figure 6.10 shows the patterns used to generate the OCL constraints corresponding to the PRIMARY KEY, UNIQUE, NOT NULL and CHECK constraints [5].

5. Note that due to the high expressiveness of the OCL language, different OCL expressions can

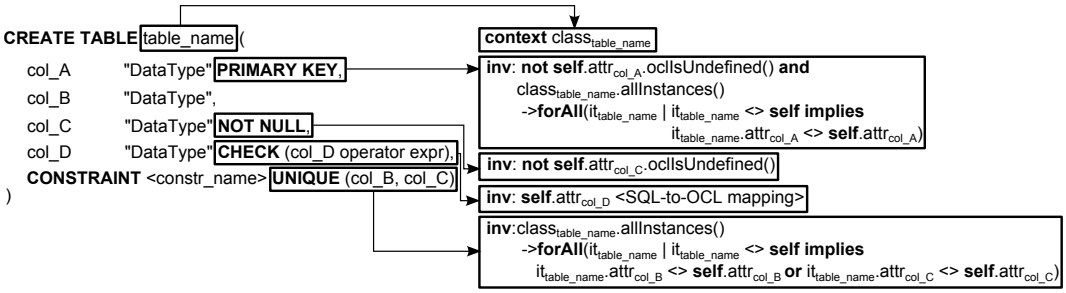

Figure 6.10: Declarative constraints mapping

The name of the table (i.e., table_name) and columns (i.e., col_A) are mapped respectively to the corresponding UML entity (i.e., $class_{table\_name}$) and attributes (i.e., $attr_{col\_A}$). In addition, the name of the table in lowercase is used as name of the iterator within the corresponding operations.

The *PRIMARY KEY* constraint specifies one or more columns that uniquely identifies each record in a table. Implicitly, these columns cannot contain null values. This constraint is translated into an OCL invariant such that no attributes, corresponding to the columns in the *PRIMARY KEY*, have undefined values and for all the pairs of different instances of the class that corresponds to the table where this check is defined, no duplicate values on the *PRIMARY KEY* attributes exist.

UNIQUE follows the same definition of the precedent constraint, but it allows null values for the columns composing it. It is translated into an OCL invariant such that for all the pairs of different instances of the class that corresponds to the table where the *UNIQUE* constraint is contained, no duplicate values exist on the attributes of the class corresponding to the columns where this constraint is applied.

The *NOT NULL* and *CHECK* constraints respectively forbid a column to have null values and limit the value range that can be stored within a column. The first constraint is mapped on an OCL invariant such that the attribute of a given class, related to the column where the *NOT NULL* constraint is used, is not undefined. The *CHECK* constraint is translated into an OCL invariant such that the conditions in this constraint are applied on the corresponding attributes of a given class in the model.

Additionally, OCL constraints are generated to enforce String attributes in the UML model respect the size constraints of the column definitions (constraints for *CHAR(n)* and *VARCHAR2(n)* mappings are depicted in Fig. 6.11).

In Fig. 6.12, the OCL invariants extracted out of the table *Job* are shown. The PRIMARY KEY on the column *job_id* is translated into an OCL constraint such that the attribute *job_id* of the class *Job* cannot be null and for all the pairs of different instances of that class, no duplicate values of *job_id* exist. The NOT NULL constraint on the column *job_title* is mapped into an OCL invariant that forbids the

represent the same semantic constraint so other equivalent alternatives are also possible

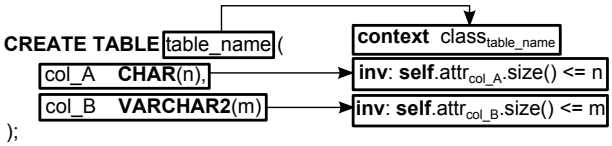

Figure 6.11: Mapping of built-in data types constraints

attribute *job_title* to have null values.

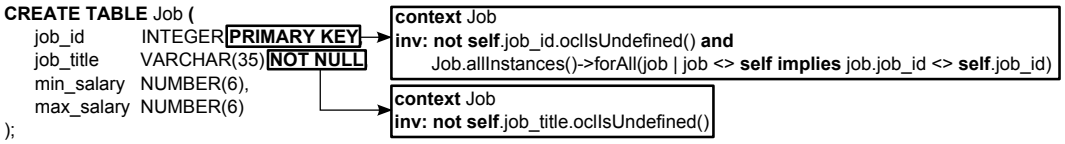

Figure 6.12: Example of declarative constraints mapping

6.7.2 SQL-to-OCL transformation

This section describes the mapping between SQL SELECT statements and the equivalent OCL expressions. In particular we describe the mappings for SQL projections, joins, functions, group by and having clauses. These mappings are needed to create the derivation rules for views as described above and to be able to extract constraints implemented as part of trigger definitions as explained in the next subsection.

In the following, we present the list of mappings.

Projection.

A SQL projection is composed by a SELECT, a FROM and, optionally, a WHERE clause. The mapping of generic projection is shown in Fig. 6.13.

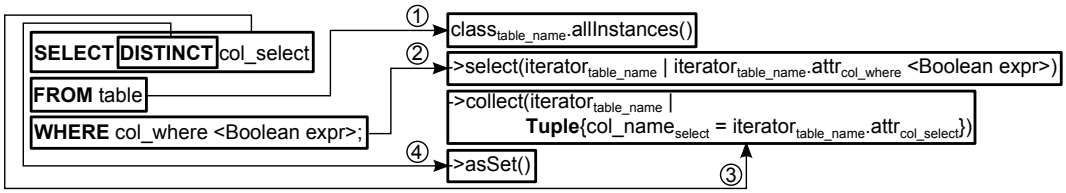

Figure 6.13: Projection mapping

1. The mapping starts by translating the FROM clause. This is done by selecting all instances (i.e., method *allInstances()*) of the class in the CS that corresponds to table in the FROM.
2. The WHERE clause, if defined, is translated into an OCL *select* iterator. The condition in the *select* iterator is created by translating the conditions in the

WHERE clause. The mapping is basically a direct mapping once the references to the column names in the WHERE are replaced by the corresponding attribute or association ends names. SQL functions are translated into their OCL counterparts (if existing, otherwise new OCL operations must be previously defined, e.g. see [54]).

3. The SELECT clause is translated into an OCL *collect* iterator that creates a collection of objects according to the structure defined in the Tuple definition. Each field in the Tuple corresponds to a column in the SELECT clause. Fields are initialized with the value of the corresponding attributes.

4. Finally, the DISTINCT clause might be used in conjunction with a SELECT statement to return only the different (i.e., distinct) values in a given table. This clause is mapped adding the operation *asSet()* after the OCL mapping of the SELECT clause.

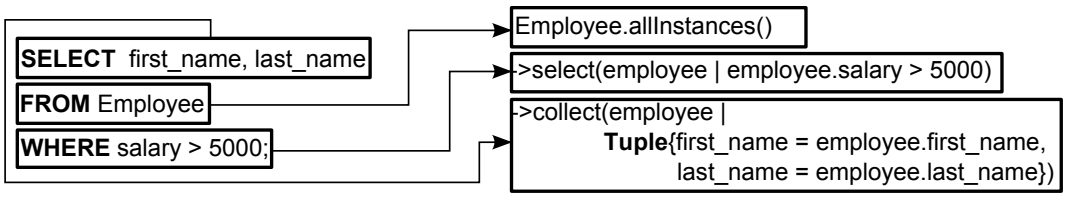

Figure 6.14: Example of a projection mapping

In Fig. 6.14, an example of projection mapping is shown. The FROM clause containing the table *Employee* is translated into an OCL operation *Employee.allInstances()* that retrieves all the instances of the UML class *Employee*. Later, a *select* operation that represents the WHERE clause is applied on those instances, such that the instances of *Employee* with a salary greater than 5000 are selected. Finally, the SELECT clause is mapped into a *collect*, such that for each remaining instance of *Employee*, its attributes *first_name* and *last_name* are collected into a tuple.

Join.

In SQL, the JOIN operation combines the values of two or more tables. Our transformation covers both inner and outer joins

– **Inner joins.**
The inner join is by far the most common case of joining tables. Giving two tables *a* and *b* and according to the join conditions, it returns the intersection of the two tables. The mapping of a generic inner join is shown in Fig. 6.15.

1. Firstly, we perform the Cartesian product of the population of all tables by retrieving all the instances of the corresponding classes in the CS and applying on them the cartesian product (named *product* in OCL).

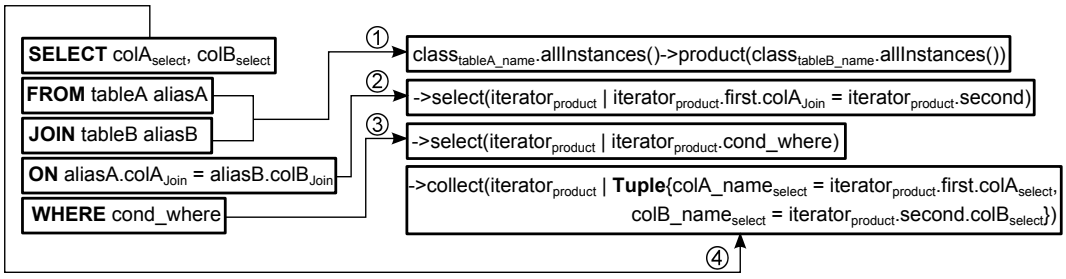

Figure 6.15: Inner Join mapping pattern

2. The join conditions are mapped to the body of a *select* operation, that iterates over the tuples of the Cartesian product (i.e., $\text{iterator}_{\text{product}}$), selecting those that satisfy the conditions. *First* and *second* are used to identify the classes in the Cartesian product (i.e., $class_{tableA\_name}$ and $class_{tableB\_name}$). Note that *colA* is compared with *second* (and not *second.colB*) since in the pattern we assume that *colA* is the name of the role of the association linking the two classes (i.e. the type of *colA* in the CS is $class_{tableB\_name}$).
3. In case the WHERE clause is defined (i.e., Implicit Join or other conditions), a new *select* operation is created following the procedure described in the previous pattern.
4. If not all columns are selected, the collect operation is used as described in the previous pattern.

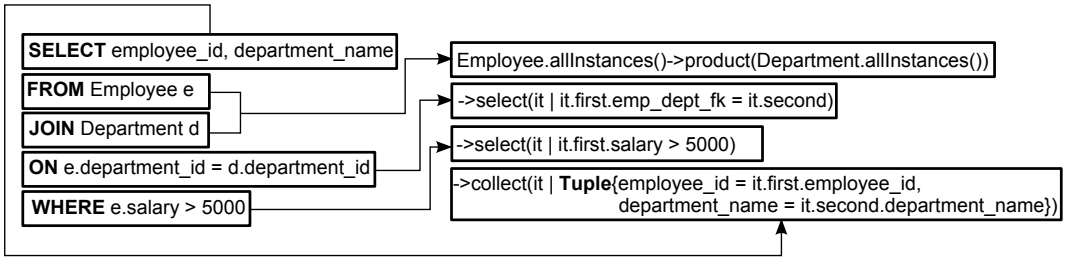

Figure 6.16: Example of explicit inner join mapping

In Fig. 6.16, an example of join mapping is shown. As first step, the Cartesian product is calculated between the instances of *Employee* and *Department*. From this set we select tuples where the value of department column of the employee (*first.emp_dept_fk*) is equals to the value of the second attribute (i.e., an instance of the class *Department*). Here, *first* refers to an employee instance since *Employee* is the first class referenced in the creation of the Cartesian product. Later, the WHERE clause is mapped into a *select* operation, that selects the employees that earn more than 5000. Finally, selected elements are projected using the *collect* operation to return a set of tuple elements with only two fields, the *employee_id* and the *department_name*.

– **Outer joins.**

The other type of join is the Outer Join that returns the same result of the Inner Join plus the rows from one/both tables that do not match any row from the other table. This Join is divided into Left, Right and Full that respectively return all the rows from the left, right or both tables, even if no matches in the right, left or in one of the tables exists. The OCL mapping of Outer Joins consists in a union operation between the OCL expression derived from the Inner Join plus the instances of the class that represents the table where the outer clause is applied. These instances are collected using the operation *allInstances()* and only those that have an undefined value on the attributes corresponding to the join columns are selected.

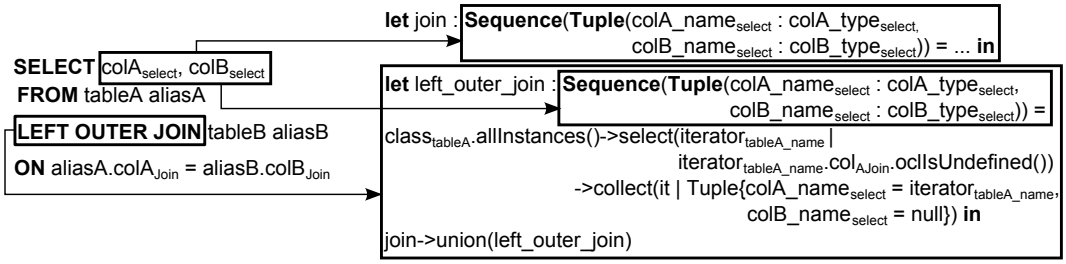

Figure 6.17: Left outer join mapping

In Fig.6.17, a generic example of Left Outer Join is presented. The result of the Inner Join is stored in the variable *join* initialized in the first *Let* expression. The Left Outer Join is defined by the variable *left_outer_join*. Its type is the same of the variable *join*; while its initialization is composed by a sub-set of instances of the class that represent the left table (i.e., table A). The instances in this sub-set have a null value on the attribute $colA_{Join}$, that represents the column whereon the join is performed. Finally, the result of the Left Outer Join is depicted by the union of the two variables *join* and *left_outer_join*.

Group By, Having and Aggregate Functions.

The GROUP BY clause is used to group the result-set of a given SELECT statement. Groups can be filtered by means of the HAVING clause. In Fig. 6.18, a generic mapping of GROUP BY and HAVING clauses is shown.

1. We first translate the SQL FROM, JOIN and WHERE clauses according to the previous mapping patterns. The result is used to initialize the *sel* variable.
2. The tuples in *sel* are then processed to create the grouped result set *group* mapping the GROUP BY clause. In short, *group* is created by iterating on the *sel* tuples and collecting together those tuples with an identical value on the attributes corresponding to the columns in the GROUP BY clause.

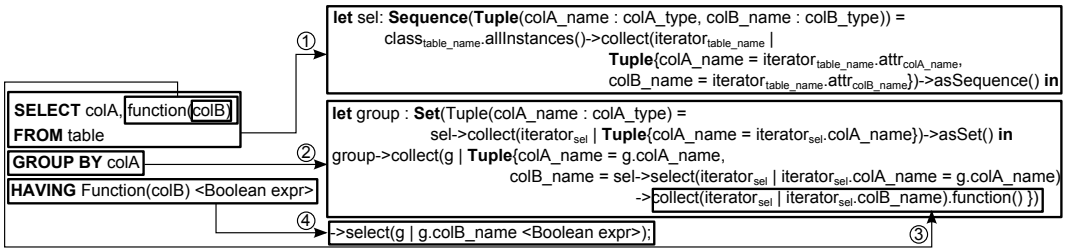

Figure 6.18: Mapping of group by and having clauses

3. If an aggregate function is applied on one of the columns in the SELECT clause, an equivalent OCL operation (see [54] for more on aggregate functions in OCL) is added at the end of the GROUP BY clause mapping.
4. Finally, if the HAVING clause has been defined, it is translated into a final *select* operation, where the body of the select expression is created by mapping the HAVING conditions.

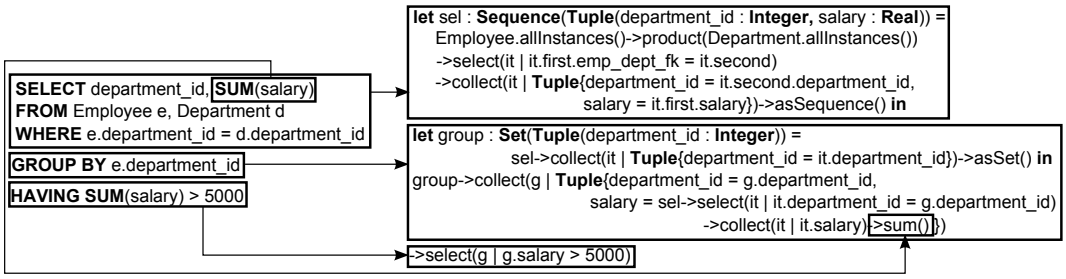

Figure 6.19: Example of a group by and having clauses mapping

In Fig. 6.19, the mapping of a GROUP BY, HAVING and SUM query is shown. To begin with, the projection of department's id and employee's salaries derived from the Cartesian product of all the *Employee* and *Department* instances is stored in the sequence *sel*. Then, for each *department_id* the sequence of *salaries* in *sel* such that the *department_id* is equal is collected. The OCL operation *sum* is applied on the result to get the sum of salaries per department. Finally, the HAVING clause is mapped into a *select* operation that selects the departments with a sum of salaries greater than 5000.

6.7.3 Triggers to OCL

A second source of integrity constraints are triggers (Sect. 6.2.2) that can be in charge of enforcing complex constraints that go beyond the expressiveness of declarative integrity constraints.

To distinguish triggers enforcing business rule from other kinds of triggers (e.g., devoted to log actions) we use the following heuristic: all triggers that embed in their triggered action section a PL/SQL statement raising an user-defined exception are classified as *constraint-enforcing-triggers*.

For each of such triggers, an OCL invariant is generated. The context of the invariant is the UML class that corresponds to the table where the trigger is defined. On the contrary, the body of the invariant is composed by the trigger restriction condition, if defined, and the output of the PL/SQL-to-OCL transformation described below.

PL/SQL-to-OCL Transformation

The PL/SQL-to-OCL transformation is used to extract OCL constraints out of PL/SQL procedures (Sect. 6.2.2), that in our case it is part of the trigger definition. The OCL constraints are extracted from the conditional statements that raise user-defined exceptions in the execution section.

PL/SQL allows defining user exceptions in two ways. One is to override an already-defined exception. In this case, the exception must be declared in the declaration section and raised explicitly in the execution section using the statement RAISE. The other way concerns the statement RAISE_APPLICATION_ERROR, that raises an user-defined exception used to communicate an application-specific error back to the user. Both kinds of exceptions are generally business relevant, since they represent a violation of the company's business and not a technology issue. Since these exceptions are raised explicitly, they are generally nested in conditional statements that check if the business constraints are violated or not.

For each exception, the conditions triggering the exception are located and mapped to an equivalent OCL expression. Note that these conditions may include variables calculated from previous SQL queries in the trigger execution section. If that case, those SQL expressions are also processed according to the SQL-to-OCL mappings described earlier.

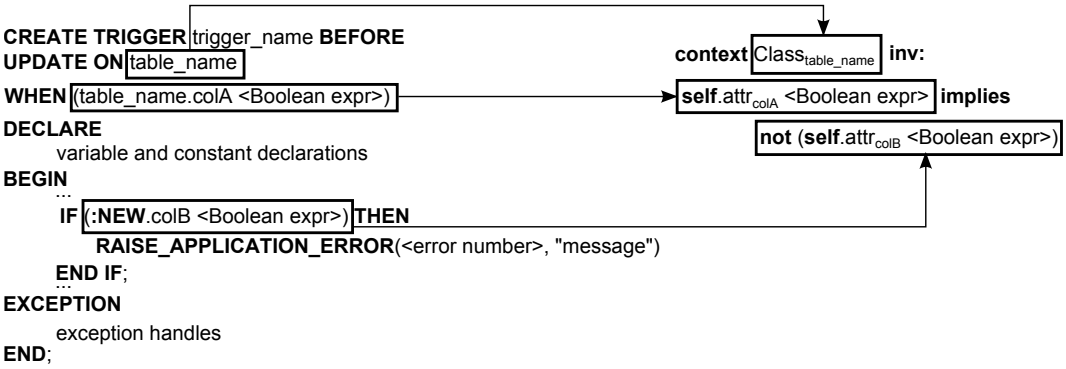

Figure 6.20: Mapping of a user exception in a trigger

In Fig. 6.20, a generic mapping from a user-defined exception in a trigger is shown. The name of the table to be updated (i.e., or inserted) is translated into the context class of the OCL invariant. The invariant's body is composed by the translations of the WHEN clause and the condition of the if-statement. Such condition

is negated since the constraint must enforce that the situation that would trigger the exception does not happen. Note that only when the tuple would satisfy the WHEN clause the second part of the expression is enforced.

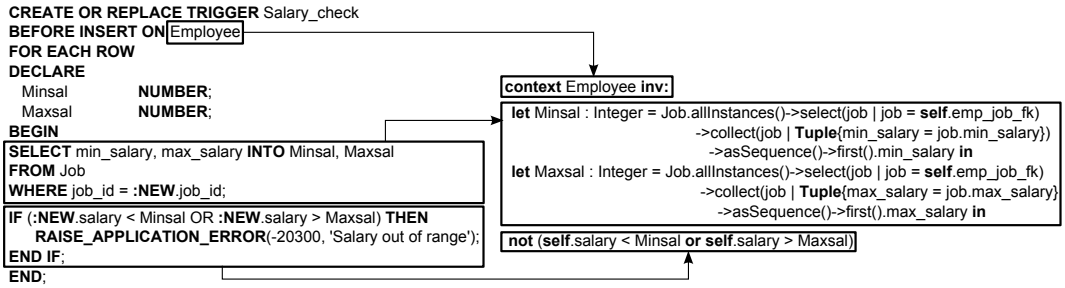

Figure 6.21: Example of a user exception mapping

In Fig. 6.21, the mapping of the *Salary_check* trigger is shown. This trigger raises an exception when a new employee has assigned a salary for a job that is not in the acceptable salary range for that job. The minimum and maximum salaries are stored in the table *Job* (Fig.6.8). They are retrieved using two variables (i.e., *Minsal* and *Maxsal*) by means of a SELECT INTO clause. The mapping of this clause extends the mapping for SQL projections seen before (Sect. 6.7.2). The only difference is that each variable is mapped to an OCL *let* expression, where the OCL variable takes the name and the type of the SQL variable defined in the DECLARE section of the trigger. Variable values are initialized with the result of the normal SQL-to-OCL mapping for the SQL query expression. Since in PL/SQL, SELECT INTO statements can only return one single row, the first elements of both projections are returned using the OCL operation *first()*. Finally, the values of the two variables are compared according to a boolean expression that maps the negation of the PL/SQL if-statement condition.

Refining OCL constraints

Strictly speaking, constraints enforced by a trigger should be linked to the event that may fire the trigger. For instance, the *Salary* constraint extracted in Fig. 6.21 should be checked only when inserting new employees in the database. In particular, such constraint should no be applied when performing other manipulations (i.e., updating, deleting) on employees. Therefore, according to the concept of creation-time constraints [55]; when extracting constraints from triggers, the stereotype mechanism provided by UML is used to annotate each constraint with information of the events that apply to it.

Only when a set of equivalent constraints applied to all the manipulation events for a given table is found; such constraints can be merged in a unique standard OCL invariant. Note that, a complete procedure to identify semantically-equivalent constraints in order to create a single one and to analyze which events can violate the

constraint in order to guess if the set of triggers is sufficient to ensure that the generated constraint should always hold is out of the scope of the proposed business rule extraction process. Such complete procedure should rely on [56] for the detection of redundant constraints and [57] for the derivation of relevant events for each constraint.

6.8 Business Rule Representation

Business Rule Representation is the last step of the framework. Its goal is to generated comprehensible textual and graphical artifacts for the identified business rules. It consists in two steps: Vocabulary Extraction and Visualization of the rules.

6.8.1 Vocabulary Extraction

The vocabulary is contained in the UML model extracted from the database implementation. Therefore, no operation must be performed to collect the business terms composing the vocabulary. Nevertheless, the UML model could be edited to improve the default class names copied from the database.

6.8.2 Visualization

Textual and graphical visualizations are employed to present the rules identified. Since UML and OCL already define a concrete syntax, this step relies on two straightforward operations to represent the UML and OCL model generated in graph-based and textual-based representations.

Textual Visualization

The rules embedded in the OCL model can be easily represented in a textual format, since OCL has both an abstract and concrete syntax. In particular, its abstract syntax is defined by using metamodeling, while its textual concrete syntax is defined by using the Extended Backus-Naur Form (EBNF) syntax.

```
context Employee inv:
not (self.last_name.oclIsUndefined());

context Employee inv:
not (self.employee_id.oclIsUndefined())
and
Employee.allInstances()->forAll(employee | employee <> self implies
                                    employee.employee_id <> self.employee_id);
context Employee inv:
self.salary > 0;
```

Figure 6.22: Excerpt of the OCL constraints extracted

In Fig. 6.22, an excerpt of the OCL invariants extracted from the database implementation is shown. Such invariants concern the class Employee (Fig. 6.8) and they state that the name of the employee cannot be null, his identifier must be unique and not null, and finally his salary must be greater than zero.

Textual-based representations of the OCL rules can be derived by using available tools on the market (e.g., Dresden OCL toolkit [6]) or implementing a model-to-text transformation (Fig.6.23) that translates the model element into text.

```
helper context OCL!BagType def : extract : String =
    'Bag';

helper context OCL!SequenceType def : extract : String =
    'Sequence(' + self.elementType.extract + ')';

helper context OCL!OrderedSetType def : extract : String =
    'OrderedSet(' + self.elementType.extract + ')';

helper context OCL!SequenceType def : extract : String =
    'Set(' + self.elementType.extract + ')';
```

Figure 6.23: Excerpt of the OCL model-to-text transformation

In Fig. 6.23, an excerpt of the OCL model-to-text transformation written in ATL is shown. The four helpers provide possible verbalization for the types *BagType*, *SequenceType*, *OrderedSetType* and *Sequence*.

Graphical Visualization

The UML model together with the OCL constraints can be used to have a global overview of how and where such constraints are applied on the model. Such representation is a UML standard class diagram, where the OCL invariants are displayed as note attachments connected by a dashed line to the corresponding class.

In Fig. 6.24, an excerpt of the UML/OCL model extracted from the database sample (Fig. 6.5) is shown. It contains the classes derived from the view and table declarations and some of the OCL invariants embedded in the declarative constraints and in the trigger.

The UML/OCL model can be visualized using different free and proprietary tools available on the market. For instance, ArgoUML [7] and Papyrus [8] are one of the most known open source UML modeling tools; while, IBM Rational Software Modeler [9] and Enterprise Architect [10] are well-known as commercial ones.

6. http://www.dresden-ocl.org/index.php/DresdenOCL:Documentation
7. http://argouml.tigris.org/
8. http://www.papyrusuml.org
9. http://www.ibm.com/developerworks/rational/tutorials/r-rsmvisual/
10. http://www.sparxsystems.com/

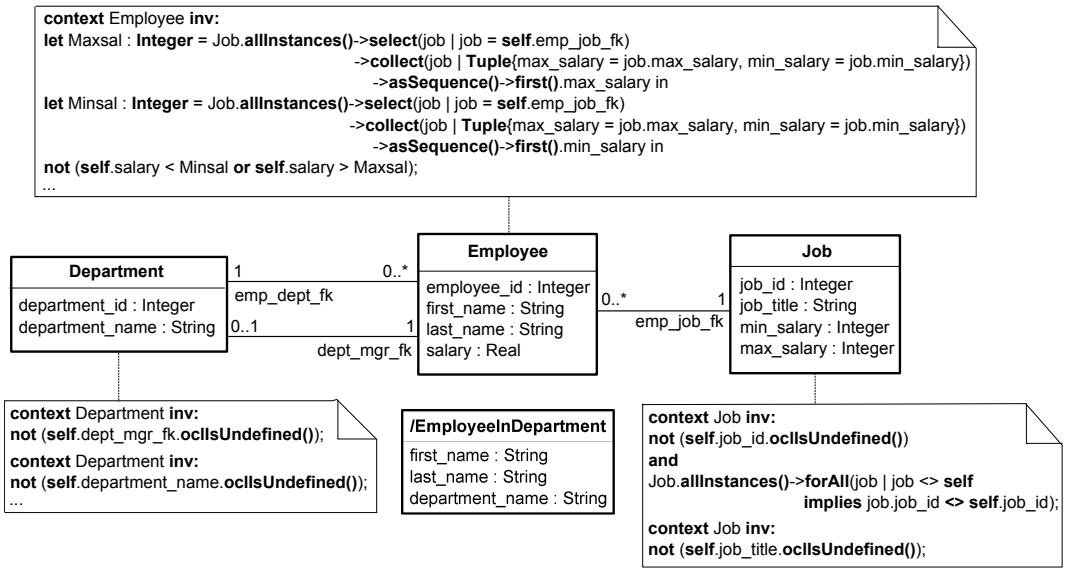

Figure 6.24: Excerpt of the UML/OCL model

6.9 Evaluation

In order to evaluate the accuracy of our method we have checked that the rules obtained using the extraction process coincide with the ones that we discovered by a manual inspection of the code. For the running example, we were able to identify and extract all the business rule embedded in the database schema.

6.10 Prototype

A prototype tool, available at [11], has been created as a proof of concept to validate the feasability of the proposed approach.

The prototype (Fig. 6.25) takes as input a database export file with the details of the database schema (tables, triggers, views,...). A first module parses this file and populates a low-level database model used to represent all the schema information of the database in the form of a model. Note that the obtained model has one-to-one correspondence with the database schema, hence no information is lost. This preliminary step is required to be able to reuse model-based techniques (which expect models as input) in the next phases of the method.

The database model is then processed as we have presented in this paper to extract the final UML/OCL conceptual schema. Finally, the OCL constraints are represented in a textual format.

The prototype embeds different model-based components available in the Eclipse platform. The parsing of the database export file is implemented with Xtext [**?**], that given the definition of the grammar of a language, it is able to generate the equiv-

11. http://docatlanmod.emn.fr/IntegrityConstraints2OCL/intro.html

alent metamodel and the parser to process text files conforming to that grammar. Given that, we have created the grammar to parse database triggers by extending an Xtext grammar defined for the Oracle PL/SQL language [12].

On the contrary, the Business Term and Business Identification steps as well as the textual representation of the extracted OCL constraints rely on ATL. In particular, the former are implemented by means of model-to-model transformations, while the latter is based on a model-to-text transformation.

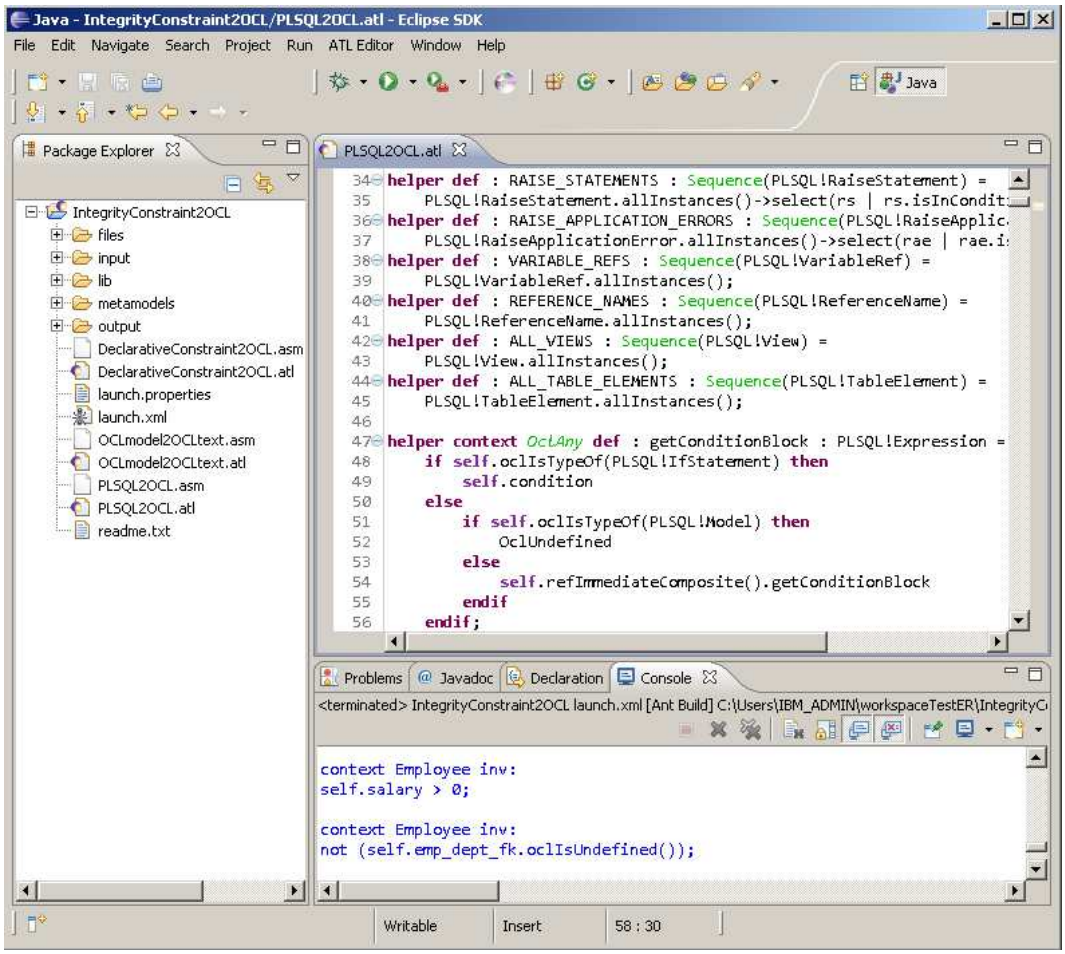

Figure 6.25: A screenshot of the prototype

12. http://sourceforge.net/p/plsql-xtext/home/Home/

7

Related work

This section presents relevant works concerning the extraction of business rules from legacy systems. These reverse engineering works are divided according to the part of the system (i.e., behavioral or structural) and programming languages whereon they are applied.

7.1 Approaches for the behavioral part

The discovery of business rules from the system's behavioral part is a research domain that has been explored extensively. We classify the existing approaches according to the type of the programming language targeted. In particular, we discuss previous works concerning approaches valid for arbitrary languages as well as, procedural and object-oriented languages.

7.1.1 Arbitrary languages

This section contains works that present generic approaches for extracting business rules. In particular, they provide definitions and common steps for identifying business rules in the source code, while no implementation details are given.

In [58], the authors describe a formal approach for business rule extraction. It is based on the mathematical assertion that programs are composed by language structures, that can be used to define extractable business rule functions. The authors present two rule extraction functions, one identifies the program structures that are part of business rules; while the other one locates the program structures that are

not part of business rules. In the following, these two rule extraction functions are depicted.

Let E be the set of all elements (variables, types, etc ...) of a language, and S be the set of all allowed language structures (statements, functions, etc ...) than can be formed from these elements. The set of all rules R that can be formed in that language can be defined as:

$$R = \{x | x \in S \wedge f(x, E, S)\} \tag{7.1}$$

where the function f is an extractable rule definition function that specifies the properties that a rule must have in terms of the language elements and structures.

According to the definition of S, a program P can be defined as a set of valid structures such that:

$$P = \{y | y \in S\} \tag{7.2}$$

Therefore, the set of all extractable rules R_e, contained in a program P can be defined as:

$$R_e = \{z | z \in P \wedge z \in R\} \tag{7.3}$$

On the other hand, all the structures in a program P that are not rules can be defined as the difference between the two sets P and R_e:

$$P - R_e = \{z | z \in P \wedge z \notin R\} \tag{7.4}$$

In [17] the authors describe a generic approach that contains the common operations to extract business rules. These operations consist in analysing the source code in order to identify the business vocabulary and rules and translating such obtained artifacts into a higher abstraction representation. In particular, two ways are proposed to extract the business vocabulary. The former relies on the analysis of clues in the code contained in data definitions, comments, variable and function names; while the latter is based on manual annotations on the code based on the analysts' understanding. On the other hand, the business rules can be extracted relying on the information flows and computational procedures embedded in the application. In particular, business rules can be derived applying static or dynamic analysis on the code (Sect. 2.4.1). In addition, the authors propose that the extracted vocabulary and rules should be connected to the corresponding source code (i.e., traceability between business rules and code). Finally, the rules identified are represented in a no code-like format relying on the business vocabulary. In particular, the authors describe an Entity-Relationship extraction process to better understand the rules with respect to the data model of the system.

In [59], a model-based approach to extract business rules and the architecture

of the corresponding tool prototype are presented. The process embedded in the tool is composed by two steps: preparation and derivation. On the one hand, the preparation step aims at extracting an initial model from different resources of the software system (i.e., configuration files, data definitions, etc.). On the other hand, the derivation step focuses respectively on discovering business terms and rules. In particular, the former are identified by performing analysis of the structural elements of the software system's GUI; while the latter are based on program slicing. Finally, the discovered set of rules is approved/analysed by the system analysts and then stored as an internal model. Such model is claimed to be further adapted to Decision Tables [60] or Semantics of Business Vocabulary and Business Rules (SBVR) [42] representations.

[58], [17] and [59] have been used to drive the definition of our model-based conceptual framework. In particular, [58] and [17] have been useful to understand which source code elements lead to business rules; in addition [17] has been considered as well with respect to the steps and features that a BREX process should provide (e.g., vocabulary extraction, traceability, etc.). Finally, [59] has been important to understand how a BREX process can be moved to the modeling environment.

7.1.2 Procedural languages

Many previous works have applied BREX approaches on legacy systems mostly written in procedural languages (COBOL above all). Each of such approaches has combined different techniques to identify the business rules in the code. In particular, they have proposed heuristics to discover variables with business-meaning, to select a slicing criteria to analyse the code as well as multiple representations for the discovered rules.

In [4] the authors propose a business rule extraction approach for COBOL systems. They define define a set of extraction criteria such that:

1. The business rules extracted should reflect the true state of the software
2. The business rules should be expressed using different representation to fit the needs of different users (e.g., programmers, manager, etc.)
3. The extraction process should be able to discover domain concepts and the business vocabulary
4. The extraction process should be interactive in order to provide the user with the specific information he needs
5. The business rules extracted should be linked to the corresponding part of the code in order to ease the maintenance of the system

The proposed approach consists of four steps: the identification of domain variables; the slicing criterion identification (e.g., backward slicing for an output vari-

able and forward slicing for an input variable); the program slicing operation concerning those variables identified in the first step and the criterion selected in the second step; and finally the presentation of the identified business rules. In particular, the domain variable identification step is composed by two heuristics. The first heuristic selects all input and output variables as domain variables, since the system can be viewed as a black box that maps inputs variables to outputs variables. On the other hand, the second heuristic considers as business variables the input and output variables of each procedure. On the contrary, three heuristics are provided for the slicing criterion identification to drive the slicing operation. In particular, the first heuristic considers the input and output statements of a program good candidates for starting a business rule extraction process. They are respectively used to perform forward and backward slicing operations. The second heuristic focuses on the part of the source code that represent dispatch centers [1] of the program. Such dispatches are good candidates to be starting points for forward slicing operations. On the contrary, the third heuristics considers end points of procedures good candidates to be starting points for backward slicing operations. Finally, the identified rules are presented in the last step of the framework by means of three different formats: code, formulæand input-output views. The code view presents business rules as code fragments; the formulae view shows the business rules as formulæ, where a formula consists in three parts: left-hand-side, right-hand-side and conditions. Finally, the input-output dependence view provides a way to trace data-flows with respect to input and output variables together.

[2] depicts a method to extract business rules out of COBOL source code. It focuses on identifying the rules in the code and presenting them with a code-like representation. The process is composed by three steps. In particular, in the first step, the user selects a variable (i.e., data item) from the list of all output variables declared in a particular program or in a set of programs. In the second step, the references to the variable previously selected are retrieved automatically. In particular, the author divides such references in three categories: usage, conditional and assignment. An usage reference is a statement where the selected variable is used to create or alter another variable. A conditional reference is a statement where the state of the variable is queried or compared. An assignment reference is a statement where the variable is created or altered and for this reason it is considered as the key to identify business rules. The last step of the process consists in capturing the conditions that trigger the assignments. A separate decision tree of the program is created and relying on the line numbers of the assignments, the relationships between the assignments and the conditions in the decision tree are established. Finally, the extracted rules are presented using a code-like representation.

[61] presents a business rule extraction method applied on an industrial use case.

1. A dispatch center delegates input data of different types to the corresponding processing units

The proposed extraction process is divided into four step. The first step consists in restructuring in an automated way the procedural code to facilitate the slicing. In particular, such code is reformatted splitting the lines with more than one instruction; indenting the nested code and the input/output operations and moving the database accesses in a separate data access section. In addition, some no-standard statements (ALTER, NEXT, etc.) are replaced with standard ones; the periods at the end of conditional statements are replaced with END-IF constructs and finally GO TO branches are replaced using PERFORM commands. The second step consists in slicing the source code according to a slicing criterion (i.e. identification of logical entry points) selected by the user. As a result, depending on such slicing criterion, several partial programs from the same original program are created. Each of such partial programs is related to a specific business rule. In the third step, all partial programs extracted from the original source are submitted to an automated documentation process. This process is in charge of generating a set of views for each partial program derived in the previous step. These views provide different kinds of information concerning the program structures involved in a particular business rule. In particular, each business rule is related to the corresponding sections and paragraphs as well as the external interfaces (e.g., input/output operations, interactions with monitors, database accesses), program logic (e.g., conditional statements, PERFORM, loops, case selections, GO TO branches), data flow and data structures. Finally, the fourth and final step aims at integrating these disjointed views into a single unified business rule documentation.

In [**?**] and [62], the authors propose a method that extracts business rules from COBOL legacy code. Such method is composed by three steps, that respectively identify business variables, discover the business rules in the code and represent them in different formats. In particular, the variable identification is based on locating the variables that appear in calculation statements. The business rule identification consists in retrieving the statements that contain these variables and their container conditions, if defined. Finally, the business rule representation for [**?**] and [62] produces different kinds of artifacts. The business rule representation in [**?**] is used to generate graph and textual outputs. The former shows how the rules are related in the program; the latter relies on clues in the code to translated technical terms to non-technical terms. In [62], the business rule representation is based on heuristics to relate the system documentation to the discovered business rules in order to provide a more abstract representation of such rules.

In [63] and [64] the authors depict a method for identifying and extracting business rules from legacy COBOL systems by means of data identification and program slicing. The extracted business rule are used to create CORBA (i.e., Common Object Request Broker Architecture [65]) components in order to reuse them across different applications. The proposed method is composed by four steps. The first

step consists in identifying relevant business variables according to a slicing criterion. This is made by classifying the code into three categories: user interface (i.e. *ACCEPT* or *DISPLAY* statements), business logic (i.e., a possible list of small pieces of code regarding to a data item in different locations) and data access (i.e., I/O statement such as *READ*, *WRITE*, *OPEN*, *CLOSE*, etc.). The second step consists in selecting an entry point for the slicing operation. This entry point is selected among the statements containing one of the business variables identified in the previous step. In the third step control flow, control dependence, and data dependence graphs are used to identify relevant program slices. Such slices are then grouped according to the business rule they contain. Finally, in the fourth step, each business rules is converted into a CORBA component.

In [66], the authors propose a framework to support the comprehension of business rules extracted from legacy systems. In particular, they focus on those business rules called constraints (see Sect. 2.4.2). The assumption in this work is that a set of constraints is extracted from a legacy system by an extraction technique and then converted from a specific formalism (e.g., the programming language they have been derived, etc.) into a generic one called Business Rule Language (BRL). Such BRL is defined to map four types of constraints:

- Constraints used to construct the domain structures, such that only particular values can be assigned to a structures.
- Constraints that restrict the maximum or minimum number of instances for a given structure.
- Constraints specifying a relationship between two or more structures (e.g., inheritance).
- Constraints defining cardinalities between two or more structure.

The proposed framework is composed by two levels, that respectively define the internal and external format of business rules. On the one hand, the internal format (i.e., representation level of the framework) expresses the rules using the BRL. On the other hand, the external format (i.e., the presentation level) offers different ways to express the semantics of business rules for different typologies of users. In addition, the separation between the representation and presentation level makes the proposed framework extensible; since when a new external format is required, a simple mapping is created between the BRL and the new external format.

In [67], the authors present a manual approach to extract business logic from source code. In particular, they focus on gathering rules that check that important business conditions have not been violated. The heuristics proposed are based on analysing the code that handle error conditions, relying on the assumption that if an error condition occurs within a program, the conditions that led to it could potentially be describing a business rule violation.

[68] focuses on recovering the business knowledge out of COBOL legacy systems. The discovery of business rules is mentioned as a possible application of the approach though the process to identify and visualize them is not discussed. The authors present a framework to extract knowledge out of legacy applications. The corresponding extraction process is composed by three steps: model discovery, extraction and interpretation. Model discovery is based on defining a Domain Specific Language (DSL) to parse COBOL programs and generating a rough model representation of such programs. The authors note that, since each COBOL vendor implements its specific dialect, a DSL must be generated for each of them. In the second step, this rough model (i.e., technology-dependent model) is translated into a technology-free KDM (Knowledge Domain Metamodel [11]) model by means of model transformations. Finally, in the interpretation step, the obtained model is further transformed into models describing the application to reformat in an object-oriented way. In particular, this step can be performed in two non-exclusive ways called essential reverse engineering and modernization. The former is an one-to-one mapping between the original code and its model, but in this case the resulting model is poor in terms of semantics. The second manner consists in a incremental software engineers intervention to introduce semantic tags in the code.

Our framework provides some additional benefits with respect to the works previously presented. In particular, [4] does not provide a representation of the identified rules for business users. In particular, no graphical visualization or vocabulary-based representations are presented. On the contrary, our framework includes both textual and graphical representations for technical and business users. [2] and [61] do not provide heuristics concerning the variable/term identification step. In addition, in both works the application vocabulary is not extracted and as a consequence the business rule presentation steps express the identified rules using only code-like representations (e.g., interfaces, program logic, source code representing the rules, etc.). Our work goes beyond, since we provide heuristics for the variable identification step and a higher abstraction level representation of the extracted rules based on the application vocabulary. [**?**] and [62] focus on the identification of single "business statements" (calculations on relevant variables) within the COBOL code. The statements modifying the same variable are not treated together to discover complete business rules. In our framework, according to the possible execution paths in the program we arrange the statements containing a given variable in order to find complete rules. [63] and [64] do not provide a business rule representation step since the identified business rules are wrapped as loosely coupled components (i.e., web services). [66] focuses on presenting the identified business rules by means of several representations in order to make them suitable to different users. On the contrary, the business term and rule identification steps are not treated. [67] limitates the rule discovery to the rules coded in exception handlers. In addition,

they propose a manual method that cannot scale when coping with large and complex systems. [68] focuses on recovering the business knowledge out of procedural legacy systems. Discovery of business rules is mentioned as a possible application of the approach though the process to identify and visualize them is not discussed. Finally, in all these previous works traceability between the rules extracted and the source code is missing or lacks on clarifying how it is implemented (i.e., [**?**] and [62]). In our framework, the rule traceability is represented by trace links embedded in the models generated during the BREX process. The correctness of such links is guarantee by MDE.

7.1.3 Object-oriented languages

Several BREX approaches have been employed on object-oriented languages. However, even if such works share the same techniques and solutions that have been used for procedural languages, the corresponding heuristics to extract business rules have be reimplemented due to the huge difference between procedural and object-oriented languages.

[6] presents a business rule extraction framework for C/C++ large legacy systems based on static program slicing. This work is an extension of [4] and it emphasizes some specific difficulties to extract business rules from large legacy systems, compared with normal sized ones. In particular, when dealing with large systems, the normal textual slice representation does not provide much guidance in the understanding due to the large number of domain and synonym variables across the system modules. Therefore, presenting and understanding rules extracted from large systems with common business rule extraction techniques can be a time-consuming and complex task. The proposed framework is composed by five steps: program slicing, variable identification, data analysis, business rule presentation and business rule validation. The first step consists in slicing the program at call-graph level. The system is represented by a set of modules and call relations between modules. A module is included in the call-graph slice if at least one statement of this module is in the program slice. In this way only a sub-set of the modules that compose the system is considered. Such sub-set should be small enough to understand the embedded rules. The second step, described as a stand-alone operation in [69], is related to the identification of domain variables. In particular, the authors propose two categories of variables, respectively pure domain and derived domain variables. The former are composed by variables that can be directly mapped to objects in the application domain; while the latter represent variables dependent on domain variables. In particular, input and output system variables are often selected as domain variables; while the derived domain variables are variables used to store partial computations on the pure domain variables. In addition, in order to reduce

the number of domain variables to analyse, a module-based variable classification is proposed. It consists in identifying for each domain variable, its synonyms across the system modules. The third step of the framework is based on data analysis to identify the business items (i.e., set of business rules) actually implemented in the slice. For each business item and relying on the information concerning the synonym variables, the corresponding business rules are then extracted and collected. The fourth step of the framework (i.e., the presentation step) collects the rules in a business rule repository according to an internal representation format and then present them to the users by means of different external formats to suit their specific needs. Finally, the last step of the framework, the validation step, consists in a manual process up to the business experts to validate the business rules extracted.

In [70], a variation of the solution proposed in [6] is presented. The framework extracts from a large C/C++ system prime business rules, that are defined as rules that hold relevant meaning with respect to the whole system or a given procedure. The framework is composed by four steps: prime program slicing, prime domain variable identification, data analysis and business rule validation. Each step of this framework follows the heuristics proposed in [6] except for the domain variable identification step, where an heuristic to locate prime domain variable is presented. In particular, for each domain variable P, the number of variables that may be modified by P (i.e., MOD) and the number of variables that may be used by P (i.e., USE) are considered. These two parameters are used to measure the influence/importance of P over all the domain variables that are in the same module of P.

In [71], a model-driven approach for extracting business rules from an enterprise resource planning (ERP [72]) system is shown. The extraction process is composed by three steps: evaluation cost, knowledge extraction and business logic abstraction. The first step consists in a preliminary phase used to determine the scope and the cost of modernizing the system. In this step the architecture, the related components and dependencies are analysed to define the strategy for extracting the business knowledge embedded in the system. The knowledge extraction step is based on representing the system information by means of KDM models. These models are representations of the documentation, database schema and source code. Finally, the business logic abstraction step aims at deriving the business logic embedded in the extracted KDM models. This is achieved by firstly building a CFG of the system, where data flow analysis is applied to locate the business processing logic; and then constructing a code-level CFG that discovers the program's structures related to the rules.

In [45] the author presents a method to extract decision tables (i.e., that can be seen as way to express business rules [73]) from Java programs in order to give to maintenance engineers a better understanding of the control flow of a given program. The proposed approach relies on the code instrumentation process, that con-

sists in inserting additional statements into a program for the purpose of gathering information about its dynamic behavior[74]. In particular, the instrumentation of Java programs is achieved by analysing the byte and source code. In the byte code analysis a call graph is built in order to extract information about polymorphic method calls that represent implicit control flow in Java programs. On the contrary, in the source code analysis, all the decision-making statements such as conditional and switch-case statements as well as looping statements (i.e., while, do-while and for statements) are instrumented to record the program path during the execution of the instrumented version of the application. Finally, when the program has been instrumented, its execution allows to extract the control flow information related to decision-making statements, which can then be used to construct the decision tables.

Our framework goes beyond the works previously described. In particular, in [6] and [70], the business rule representations do not rely on the application vocabulary and as consequence, they are difficult to be understood by business analysts. On the contrary, in [71], the business rule representation includes source code-based artifacts and more abstract representations such as graphs and vocabulary-based verbalizations. Despite this, such representations are not discussed in detail. [6], [70] and [71] do not implement rule traceability and the extracted rules are not separated from their context. [45] implements rule traceability, but it employs code instrumentation techniques. Such operation is intrusive, since the code must be modified to recover the dynamic behavior information. In our framework, rule traceability operates at model-level; as a consequence, the source code is kept in place.

7.2 Approaches for the structural part

The discovery of business rules in databases has been largely studied as part of the database reverse engineering process [7]. We separate the previous works according to the main steps of the aforementioned process.

Therefore, we discuss solutions concerning the extraction of conceptual schemas from database implementations and the mappings between database integrity constraints and business rules languages, notably OCL and SBVR.

7.2.1 Database implementations and conceptual schemas

Many reverse engineering works have been focused on extracting conceptual representations (i.e., ER/UML models) out of relation database schemas. In the following we describe the solutions proposed in [75], [7], [76], [77], [78], [79], [80], [81], [82], [83] and [84].

In [75] the author presents the common steps of a reverse engineering process for relational databases. The process is divided as follow:

1. Identification of the database schema with its tables and their corresponding relations.
2. Discovery of the domain semantics such relationships' cardinalities between tables and database integrity constraints.
3. Representation of the extracted schema and semantics in a conceptual model that ease the comprehension of the extracted information.

In [7] the authors propose a method that integrates the common steps of a reverse engineering process and program analysis techniques. The proposed approach is composed by two main-steps: data structure extraction and data structure conceptualisation. In the first step, the database structure is recovered from the physical schema, programs and the data stored in the database. This step is divided in three sub-steps: database text analysis, program analysis, data analysis and schema integration. In the text analyses, the logical schema, that contains the explicit constructs and constraints, is extracted from the data structure declarations (e.g., table declarations, etc.) defined in the database. In the program analysis, the application programs and the related procedural parts are analysed to detect implicit data structures and integrity constraints. In the data analysis, data structures and properties are discovered from the data stored in the database by means of data patterns. Finally, the schema integration merges the schemas obtained from the previous steps and returns a rough logical schema including both the implicit and explicit structures as well as the related constraints. The second step of the proposed method is the data structure conceptualisation. It consists in refining the logical schema obtained in the previous step and extracting from it an equivalent conceptual interpretation. In particular, the conceptual schema is derived by removing or transforming the non-conceptual structures and redundancies in the rough logical schema.

[76] presents a method for extracting a conceptual schema from a relational database. The method consists in analysing data manipulation statements in the code of a software system that relies on a relational database. The proposed process is composed by two main steps: the generation of a connection diagram and the derivation of a ERC+ diagram (an ER model extension) from the connection diagram. In the first step, semantically related attributes are identified by analysing the join clauses in the data manipulation statements of the application. Such attributes are then represented as relation schemes. These schemes are used to create an incomplete version of a connection diagram, that is later enriched with further information. In particular, such information concern the object types of the relation schemes identified and the corresponding relations between them (i.e. anchors). In the second step, this connection diagram is transformed into an ERC+ diagram by

means of a set of translation rules: node, link and refinement rules. Node rules analyse the object types and anchors in order to identify entity, relationship and dependent types in the ERC+ diagram. Link rules are used to identify properties and dependencies relations within the attributes composing an anchor. Finally refinement rules combine the information given by the previous rules and derive the final translation.

In [77] the authors depicts a process to extract business knowledge from an system considering the information contained in its relational database and legacy source code. The reverse engineering process is composed by two parts: schema extraction and semantic analysis that respectively extract the database schema and enrich it with information retrieved from the application code. These two parts are achieved by an eight-step process. In the first step an abstract syntax tree is generated from the application code in order to perform a semantic analysis. In particular, this analysis consists in associating the program variables to a column of a table within the database. The second step aims at extracting attribute and relation names from the legacy source. Such information can be retrieved in two different ways either by querying the database or analysing the application code. The former is a manual operation, while the latter is depicted as semi-automatic. In particular, the code analysis aims at finding possible primary key candidates using predefined rule patterns, that identify SQL queries in the code. If a primary key is not found, the reduced set of candidate keys is returned and a manual selection can be made. The third step performs a code analysis based on program slicing and pattern matching as well as data dependency and control flow graphs. These operations aim at enriching the dictionary extracted in the previous step and at identifying business rules not explicitly stored in the database. The fourth step identifies inclusion dependencies constraints (e.g., class/subclass relationships). Such constraints are derived for each pair of relations and attribute combinations (i.e., discovered in the second step) by running SQL queries and checking the number of tuples returned. The fifth step identifies the different types of relations. Such relations are classified into four categories: strong, regular, weak or specific, according to the primary key information extracted in the second step and inclusion dependencies in the fourth step. In particular, a strong entity-relation is defined by a primary key composed by attributes that are not primary keys of other entities. A weak entity-relation contains properties that are primary key attributes in other entities. Finally, regular relations represent relationship composed by entities in the database schema; while specific relations contain at least an entity that is no longer part of the schema (e.g., deleted as part of the normal schema evolution process). In the sixth and seventh step the attributes and the entities are classified. In the former the attributes of extracted entities are classified as primary, foreign, dangling keys or non-keys; while in the latter, the entities are classified as strong or weak, according to the strong and weak entity

relations discovered in the fifth step. Finally, in the eighth step the cardinalities of the relations of the extracted schema are determined according to the inclusion dependencies and the different types of relations identified.

In [78], the authors propose a process for extracting conceptual models out of a database schema. The process only relies on the information contained in the database. Initially, a tentative class is created from each table and the class attributes are derived from the columns composing the corresponding table. Then, progressively, the rough conceptual model obtained is refined according to the information discovered about primary keys and foreign key groups. In addition, the latter are used to derive both generalizations and associations among the classes in the model. Finally, transformations on the classes and associations discovered are applied in order to optimize the conceptual model. For instance, these transformations can concern changing, where possible, one-to-one associations to class attributes; or inserting intermediate classes in a generalization hierarchy to recognize common semantics, attributes and associations.

In [79] a reverse engineering method is presented to convert database schemas in UML models expressed in XMI (XML Metadata Interchange) format. The mappings are defined on tables, attributes and relations embedded in the database schema. The conversion process analyses the SQL CREATE statements and derives for each table, the corresponding UML class. In addition, a further analysis on table columns and foreign keys completes the obtained UML classes. For each table column according to its name and data type, a property in the corresponding UML class is derived. Moreover, each property that corresponds to a primary key in the database schema, is annotated with a primary key stereotype. Finally, foreign keys are mapped to properties as well and they are used to create association between the UML classes. In [80], an opposite method to [79] is presented. The authors describe how to generated relational schemas from UML models. The method relies on mappings between the XMI format and SQL.

[81] presents a method able to map a relational schema to an object-oriented schema. The method is composed by two main steps: object classes and relationships identification. In the first step, the object classes are identified from Second and Third Normal Form relations [85]. While in the second step, three kinds of relationship between classes are discovered: association, inheritance and aggregation. In particular, associations can be identified according to two cases. The former consists of primary keys composed by foreign keys. The corresponding classes are named association-classes and they are considered both associations and classes at the same time. The latter aims at creating an association between classes that are not association-classes according to the primary and foreign key information. Three kinds of inheritance relationship are discovered. The first case concerns tables that have the same primary key, such that in the relation schema one relation for the

super class and one relation for each subclass exist. The second case identifies inheritances embedded in a single class relation; while the third case refers to relations that contain repeated attributes of the superclass. Finally, the identification of aggregation relationships is based on analysing primary and foreign keys.

In [82], a method to convert data and schemas of relational databases to the corresponding equivalent in the object-oriented technology is proposed. The method is composed by three steps that include schema translation, unload and reload data from the source relational database to the target object-oriented database. The first step aims at identifying functional and inclusion dependency constraints and class objects as well as association attributes and inheritance. A functional dependency is defined on pairs of attributes X, Y; such that the value of X must be associated with precisely on value of Y. On the other hand, an inclusion dependency is defined if the set of values appearing in the attribute X are a subset of the set of values appearing in the attribute Y. The class objects are derived from relation, such that for each relation a class object is created and for each attribute in the relation, the corresponding attribute in the class is created. The foreign keys are used to determine the association attributes of the conceptual model. In particular, one-to-many and one-to-one relationships are represented by a containment hierarchy and a variable that points to the other class composing the relationship. On the other hand, many-to-many relationships are composed by two association attributes (i.e., represented by a containment hierarchy and a variable) in the class object, that respectively point to the other classes composing the relationships. Finally, the inheritance is derived from *isa* relationships. Such relationship is applied to two classes called respectively *superclass* and *subclass* and it is defined only if the primary key of the relation that corresponds to the subclass is a subset of the primary key of the relation that corresponds to the superclass. The second step of the method transforms relations' tuples into insert statements and add them to a sequential file. In a second phase, for each foreign key, its referred parent relation's tuple is translated into an update statement and unloaded into another file. Finally, for each subclass relation, its referred superclass relation's tuple is converted into an updated statement and finally loaded into a third file. The third step consists in reloading the sequential files generated into an object-oriented database. As prerequisite of this step, a conceptual model has to be derived from the relational database from which the sequential files have been calculated. The files are loaded in order as they have been generated in step two.

[83] introduces a methodology for transforming relational schemas into object-oriented ones according to ODMG-93 standard, that defines the core aspects of an object-oriented schema. In this way, the schema conversions can be made in a system-independent way by using the ODMG model as target. The proposed approach is a three-step process based on extracting the relation schema from a database application and then transforming the obtained schema into ODMG struc-

tures. Finally a refining step is performed to obtain a conceptual schema aligned with the object-oriented principles.

[84] presents a solution to express relational database views as derived classes in conceptual models. In particular, the author shows how the expressiveness of OCL can be used to represent the relational algebra embedded in database views (i.e., restricting his analysis to views without aggregates and grouping constructions). The proposed approach is divided in three steps. Firstly, a database class is defined as the root class of the conceptual model. Then, for each view, a derived class in OCL is built. In particular, the derived class is created using sets, tuples and equivalent OCL constructs able to express the Cartesian product operation in SQL. In addition, each input of the Cartesian product in OCL takes as input a base class in the conceptual model that represents a table in the database schema. In the last step, the context and the return type of the derived class are determined, such that the former is the database class defined in the first step, while the latter depends on the conversion of the column data types composing the view.

7.2.2 Database constraints and business rules

This section describes works that bridge integrity constraints embedded in the database schema (i.e., expressed in SQL) to business rules languages. In particular, we separate two kinds of approach with respect to the business rule language employed, such that we discuss mappings respectively for SQL and OCL, and SQL and SBVR.

SQL and OCL

In [40], the authors analyse the OCL language to investigate imprecise or ambiguous definitions. In addition, they provide a comparison between OCL and a language for specification of queries and integrity constraints in EER models (i.e., ERR calculus). ERR calculus and OCL share the same purpose. In particular, they are intended to specify declarative constraints in order to restrict the possible system states to desired ones. In addition, they both allow to query the current state of a system. Despite this, these languages rely on a different notation, on the one hand OCL directly allows to navigate through a class model in a path expression-like style, that improves the readability for small expressions. On the other hand, the logic-based EER calculus formulations rely on the SQL-like query notation, that results more user-friendly when dealing with large expressions.

In [86], the authors discuss the expressive power of OCL. In particular, they focus on the OCL mapping proposed for the operations of the relational calculus. These operations are respectively union, set difference, intersection, Cartesian product, quotient, projection, selection, join and natural join [87]. Union, difference, in-

tersection are primitive operations in OCL; while projections and selections can be emulated using respectively collect and select operations. On the contrary, Cartesian products, joins and natural joins are not directly expressible in OCL. Nevertheless, the Cartesian product can be mapped in OCL as an union operation between two sets of instances, that correspond to the tuples contained in two database tables. On the other hand, join and natural join operations can be represented using the OCL mapping previously proposed for Cartesian products and selections.

In [88], the authors analyse OCL as a query language for querying UML models. In particular, they propose definitions for a new data type called *tuple* and two operations named *product* and *project*. Such operations are defined respectively on collection and tuple types. The tuple type is defined in a very similar way to the OCL collection types (i.e., Set, Sequence and Bag) and the authors show how such instances can be defined as part of an OCL expression. The product operation is defined as a Cartesian product between two or more sets of instances. Finally, the project operation is defined to extract specific elements from the target tuple whereon is applied.

In [23], the authors propose a method to extract the conceptual schema from a database system, translating as well the related built-in data type and declarative constraints to their equivalent in the UML/OCL conceptual schema. The mapping between the built-in data types in the database and the predefined set of data types permitted in the UML language are provided for *Char(n)*, *Varchar2(n)*, *Integer*, *Float*, *Date* and *Number(precision, scale)*. In particular, the *Char(n)* type is transformed into a *String* type with an additional OCL constraint over all attributes of that type stating that the length of their *String* value must be exactly *n*. The *Varchar2(n)* type is transformed into a *String* type with an additional constraint verifying that the maximum length of the *String* value is *n*. *Integer*, *Float*, *Date* types are transformed into their equivalents UML *Integer*, *Real* and *Date* types. Finally, the *Number(precision, scale)* is transformed into an *Integer* data type when precision is zero and into a *Real* type otherwise. The mapping between the declarative constraints and OCL invariants are provided respectively for NOT NULL, UNIQUE and PRIMARY KEY constraints as well as for CHECK constraints. A NOT NULL constraint is represented as a minimum 1-multiplicity in the corresponding attribute or association of the class in the conceptual schema. UNIQUE and PRIMARY KEY constraints are represented as an OCL invariants. The context is the class that corresponds to the table in the database schema; while the body of the invariant is *C::allInstances()->isUnique(attr)* where *attr* is the attribute marked as UNIQUE or PRIMARY KEY in the table. In case these declarative constraints involve more than one attribute, the body of the invariant is defined as *C::allInstances()->forAll(x,y | x<>y implies (x.att$_1$<>y.att$_1$ or ... or x.att$_n$<>y.att$_n$* where *attr$_1$ ... attr$_n$* are the set of attributes restricted by the constraint. Finally, the CHECK constraints are con-

verted to OCL invariants with an equivalent expression in the body.

[21] shows how OCL, UML and SQL can be used in modeling database constraints and discuss their advantages and limitations. In particular, the authors present patterns for mapping OCL invariants to SQL code. Simple OCL invariants can be mapped to SQL table constraints, such that the name of the table to create is the context of the invariant, the current row is the contextual instance *self* in OCL and the body of the invariant is translated as a CHECK constraint in the table declaration. Nevertheless, since referring to the current row in nested queries can be a problem, this solution cannot be applied to more complex OCL expressions. On the contrary, a mapping that works for complex expressions consists in transforming the OCL invariant in a *forAll* operation, that is then mapped to an SQL predicate and nested in the EXIST predicate of a standalone constraint (e.g., assertion). This obtained constraint is satisfied if no tuple that corresponds to an object invalidating the OCL invariant is found. However, this mapping works only with single OCL expressions that return a boolean value as result type. Since an OCL expression can be composed by different subexpressions, other OCL mappings are presented for expressions concerning basic, model and collection types. The values and mostly of the operations (e.g., unary, multiplicative, additive, etc.) of the basic OCL types (i.e., *Real*, *Integer*, *String*, *Boolean*) have a direct SQL counterpart. If the return type of the OCL subexpression is *Boolean*, the aforementioned mapping is used; otherwise the subexpressions are translated into equivalent SQL predicates. The OCL model type mappings are divided with respect to classes and attributes, navigations and operations. A class is mapped to a table and an object of that class to a row of that table, so that each object identifier corresponds to a unique value of the primary key. On the contrary, attributes are mapped in a straightforward way according to this translation pattern: <*class_table*>.<*attribute*> -> *table.*<*attribute*>. Navigations are expressed by queries with nested sub-queries. The corresponding mapping depends on the kind of association and how it is represented in the database schema. Finally OCL operations can be mapped as static methods in Java called by SQL statements. The OCL collection types concern mappings for complex predicates, basic values and queries. They respectively cover all OCL collection operations that result in boolean values (e.g., *includes*, *excludes*, *forAll*, etc.), in basic values (e.g., *Real*, *Integer*, *String*) and in a collections again (e.g., *asSet*, *asBag*, *select*, *reject*, etc.).

[89] presents a method to translate and back-translate OCL expressions into SQL queries. The proposed algorithm to map OCL to SQL consists in three steps. Initially the table and the alias table names are derived from the input OCL expression, then JOINs, SELECT and WHERE clauses are calculated from the OCL operations; and finally the extracted information are merged to an SQL query template. On the other hand, the algorithm for translating SQL to OCL is composed by

two steps, such that firstly the classes are retrieved from the FROM clause and the OCL operations are derived from both SELECT and WHERE clauses. Finally, the extracted information are merged with a OCL query template.

In [90], the authors present mappings to express constraints on binary relationships within UML/OCL in SQL views. Mappings are provided for 1-1, 1-n, m-n relationships. In addition, a further mapping is provided for relationships having either the minimal multiplicity greater than one or the maximal multiplicity other than *. Finally, the obtained views are used to retrieve the records violating the constraints on the multiplicities.

SQL and SBVR

[22] presents an approach for extracting structural business rules, expressed in SBVR, from legacy databases. The business rule extraction process is performed in three steps. Firstly, the business concepts are extracted from the database; then connections (i.e., fact types) between the concepts previously discovered are identified by means of verb phrases; finally, the business rules are extracted by enriching the fact types with quantifiers, modal and logical keywords. In addition, the extracted business rules are connected to the tables, columns and comments composing the database application. The business concepts are extracted from tables and table columns, since they often represent business relevant information. Furthermore, each table column is also used to identify a fact type due to its relationship with the table that belongs to. The connections between business concepts are discovered by analysing foreign key constraints. The name of the corresponding fact types in SBVR is derived from the table/column comments the constraints reference. Finally the structural business rules are derived from SQL CHECKS and NOT NULL constraints coded in the table declarations of the database schema.

7.2.3 Stored procedures, triggers and business rules

In [91], the author describes an approach for extracting business rules from PL/SQL [51] code in order to allow a better management and a drastic reduction of traditional code. The process is composed by seven steps. In the first step the code is analysed in order to collect conditions and actions; while in the second step the information extracted in the first step are entered in decision tables. In the third step, a verification is performed for tabular correctness and completeness and in the fourth step, a decision tree is generated from the decision table. In the fifth step, the decision tree is analysed and compared with the code; while in the sixth step, business rules are generated from the decision tree and manually optimized. In addition, they are stored in a business rules repository. Finally, in the last step the source code is replaced by calling the extracted business rules from the repository.

In [92] the authors propose an approach to generate SQL views from business rules expressed as OCL invariants adapting the patterns presented in [21]. The result of a view evaluation is a set of tuples from the constrained table that represent the objects violating the invariants. In addition, the generated views can be used by triggers, that evaluate the constraints after each critical data manipulation operation. In this way, when any constraint violation is found, the trigger should rollback the current transaction and send an appropriate error message to the application. The view can be later evaluated in three ways: application driven view evaluation, assertion replacement and Event-Condition-Action trigger template. In the first case the evaluation is done by using Java and JDBC to access to the database. In the remaining cases two trigger templates are created, such that the former raises an application error if the evaluation fails; while the latter is prepared to the treatment of faulty data.

In [93], the authors provide a mapping between OCL and SQL. In particular, they focus on representing participation constraints into OCL and Trigger-based constraints. A participation constraint represents generally conditions on links between objects of a class or objects of two or more others classes. Two kinds of participation constraint are studied, those defined on classical binary associations and those defined on generalization/specialization links (inheritance links). Participation constraints on binary associations control the link of objects of the same class to objects of different classes. Several kinds of participation constraints are identified in binary associations: inclusion, simultaneity, exclusion, totality and xor. The inclusion is used when a given object A participating in a relation R1 must participate in a relation R2; the simultaneity is used when a given object A must participate in both R1 and R2 associations; the exclusion is used when the object A participates either in R1 or R2 or in none of them; the totality is used when all A-objects must participate in R1 or R2 or both at the same time. Finally, the xor is used when all A-objects must participate in only one association between R1 and R2. On the other hand, participation constraints on generalization/specialization links are divided into disjoint and complete constraints. A disjoint constraint specifies whether two objects of different specializations may be related to the same object of the generalization. A complete constraint specifies whether objects of the specifications are related to all generalization-objects.

7.3 Comparison with our framework

This section compares the previous works with our BREX framework for the behavioral and structural parts of systems.

7.3.1 Behavioral part

The discovery of business rules from legacy source code is a research domain that has been explored extensively (see Sect.7.1). Nevertheless, our framework brings some contributions to such domain. We discuss the previous works with respect to different criteria that are described below:

- Granularity. It separates the statements that compose a business rule according to the information they contain. In particular, such information can be directly related to the business variable or contained in the rest of the execution path where the rule has been discovered.
- Orchestration. It allows to understand the dependency between the extracted rules (i.e., how the rules are connected each other).
- Internal-external representation. It provides a two-level representation of the business rules. Such representation makes possible to generate different output as well as to add new external formats for the same business rules
- Vocabulary-based representation. It is used to provide a more abstract verbalization of the rules avoiding the specificities of the programming language.
- Traceability. It allows to relate the extracted rules back to the source code.

In [4], only the criteria concerning the internal-external representation and the orchestration are satisfied. In particular, the statements composing a rule are not separated with respect to the context and no traceability is kept between the extracted rules and the source code. Although an internal-external representation of the rules is implemented and several output artifacts are defined to express the rules; none of them allows to produce a verbalization based on the application vocabulary. Finally, the rule orchestration criteria is achieved by merging the obtained program slices with the data flow concerning the variables in such program slices.

In [2] and [61], the internal-external representation criteria is met. In particular, the original program is split in partial programs according to the business rules identified. Each partial program (i.e., internal representation) is used to generate five views (i.e., external representations) that highlight certain features of that partial program (e.g. data structures, program logic, etc.). In addition, the proposed frameworks does not provide heuristics concerning the variable identification step.

In [**?**] and [62], all criteria expect the granularity of the extracted rules and the vocabulary-based representation are met. The authors focus on the identification of single "business statements" (calculations on relevant variables) and the corresponding container conditions (if defined). They do not extract from the execution path that contains a rule, the set of conditions that trigger that rule. In addition, they do not provide a vocabulary-based representation of the extracted rules. Nevertheless, they bridge the identified rules to the documentation of the system. Finally, although the authors claim that rule traceability has been implemented, no explana-

tion is provided.

[63] and [64] do not meet any of the defined criteria. In addition, they do not provide a business rule representation step, since the identified rules are migrated to CORBA components.

[66] meets the criteria concerning the internal-external representation of the extracted rules and the vocabulary-based representation. In particular, the former is achieved by defining a business rule language; while the latter is based on a domain knowledge database that allows to translate low-level variable names to more comprehensible domain concepts.

In [67] no criteria is met. In addition, the proposed approach is depicted as a manual rule extraction process. This entails that such approach would hardly scale on a large legacy system.

[68] focuses on recovering the business knowledge out of legacy systems. Discovery of business rules is mentioned as a possible application of the approach though the process to identify and visualize them is not discussed. Nevertheless, the internal-external representation criteria is met. In particular, the source code is transformed into KDM models that are then used as internal representation of the system.

In [6] and [70], the criteria related to the internal-external representation as well as the orchestration of rules are met. The former relies on the work in [66], while the latter is achieved by a slicing performed on the call-graph of the application. In particular, since the obtained slices contain a set of programs that are in turn analysed for extracting business rules; it is possible to determine the position of such rules in the call graph of the application. These positions can be used to orchestrate the discovered rules.

[71] the internal-external representation and vocabulary-based criterias are met. The former is represented by a set of KDM models derived from the AST of the application. Such models are then used as internal representation of the identified business rules. The latter is achieved by discovering business terms while generating the KDM models.

[45] meets the traceability criteria. Nevertheless, the traceability is achieved relying on an intrusive approach based on the byte-code instrumentation of the application.

7.3.2 Structural part

The BREX process from the system structural part is compared with the previous works on database reverse engineering that are focused on extracting a conceptual model from the database schema and mapping the constraints expressed at database-level and their equivalent in the conceptual model.

Criteria	[4]	[2] [61]	[?] [62]	[63] [64]	[66]	[67]	[68]	[6] [70]	[71]	[45]
Granularity										
Orchestration	x		x					x		
Internal-external representation	x	x	x		x		x	x	x	
Vocabulary-based representation					x				x	
Traceability			x							x

Table 7.1: Related work comparision

The extraction of conceptual models out of database implementations has been the focus of several previous researches (see Sect. 7.2.1). Nevertheless, such works do not cover the constraints beyond primary and foreign keys also included in the schema.

On the contrary, other reverse engineering approaches center their attention on such constraints (see Sect. 7.2.2). Some of the earlier works show how business rule/constraint languages like OCL can be used to express the relational calculus ([40], [86], [88]); while others present mappings between database languages and business rule languages.

In particular, most of these previous works provide translations respectively from OCL to SQL ([21], [89], [90]) and from OCL to PL/SQL ([92], [93]) to represent business rules expressed at model-level by means of database constraints. On the other hand, only few works ([23], [22], [91]) define mappings from database languages to business rule languages. Nevertheless, none of them focuses on the analysis of business rules/constraints enforced enforced by means of triggers.

8

Conclusion and further research

8.1 Conclusion

Organizations rely on the logic embedded in their information systems for all their daily operations. This logic implements the business rules in place in the organization, which must be continuously adapted in response to market changes. Unfortunately, these business rules are not usually implemented as a single and easily identifiable component in the underlying system.

In order to recover the business rules embedded within a system, we have relied on MDE, that has shown the maturity of software modeling paradigm to be applied on the analysis of programming languages and BREX processes. In particular, we have defined a model-based reverse engineering framework that offers modularity with respect to the business rule extraction process and genericity with respect to the programming language that implements the system. In addition, the business rules obtained can be analysed at different level of detail (i.e., source code, vocabulary-based and graph-based artifacts) and they can be easily related to the corresponding source code elements (i.e., traceability).

This conceptual framework has been implemented for the behavioral and structural part of systems. In particular, COBOL and Java have been analysed for the behavioral part. The source code has been sliced to extract the relevant statements with respect to a given business variable (i.e., terms), identified by means of heuristics related to the programming language employed for the system. These statements have been stored in a model-based internal representation and then externalized by means of visualization techniques (i.e., text and graphs) to fulfill the needs of different users (i.e., business analysts, developers).

In addition, evaluations concerning the implementations of the BREX framework have been performed on two IBM use cases. The return of experience has been positive and the lessons learned have helped us to improve such implementations and point out further researches.

With respect to the structural part of the system, we have focused our analysis on relational databases. In particular, a UML conceptual model has been derived from a database implementation to identify the business terms, and the database integrity constraints coded in table definitions and triggers have been analysed to extract rules expressed as OCL invariants. Finally, a prototype has been created as a proof of concept to validate the feasibility of our BREX framework for databases.

8.2 Further research

In this section we present four possible lines of future research. The first one pretends to extend our BREX process to the part of the system that concerns the presentation of information; the second envisages the possibility to define a common/pivot business rule metamodel to represent the identified rules expressed by different programming languages (e.g., COBOL, Java, etc.), the third one consists in adding an automatic validation step for the business rules extracted in order to discover inconsistencies within the system and finally, the last future research concerns the system redocumentation according to the extracted business rules.

In the following, we discuss the four future research lines proposed.

8.2.1 Business rule extraction for the system presentation part

We have applied our framework to the behavioral and structural part of systems, discovering in this way the embedded derivation and constraint rules. We believe that to complete our BREX process in an information system, the analysis of the logic enforced as part of the system presentation part is needed.

Such part aims at integrating user interactions with the system. It consists of visual objects (i.e. screens, web pages, report, forms, etc.) used to collect user input and to display output information. In particular, we are interested in analysing the part of the source code used to enter data (e.g., forms in Java, screens in COBOL), since generally they embed checks on the data entered. Therefore, these controls can be analysed to complete the business constraints retrieved from the structural part of the system.

In addition, visual objects are related to input/output system variables, that can hint at relevant business concepts. Hence, these variables can improve the heuristics defined in the Business Term Identification step and consequently the result of our framework.

8.2.2 Definition of a pivot metamodel

Our conceptual framework has been applied to different programming languages, namely COBOL, Java and SQL-PL/SQL. Although the steps of the conceptual framework are generic, the business rules extracted from such languages are stored in models that conform to different metamodels. As a consequence, in order to automatically unify the business rules discovered in the different parts of the system, further manipulations are needed to overcome these heterogeneous model-based representations.

The use of a pivot metamodel can come in handy when analysing together the behavioral and structural part of the system or when dealing with a part of the system (i.e., behavioral, structural) that merges different programming languages. This is specially true for legacy systems that have been evolved over time, where the behavioral part contains some functionalities implemented in newer technologies as Java and other ones in older technologies as COBOL. In addition, the definition of a common metamodel would simplify the integration of an automatic validation step in the framework.

The trade-off of using such a pivot metamodel has been evaluated at the beginning of this thesis when designing the conceptual framework. In particular, on the one hand, the pivot metamodel makes easy to export, merge and automatically check the rules extracted from different programming languages since they are represented by a unified format. On the other hand, such unified representation entails the loss of the specificities proper of any programming language, hence the rules are complete, but less detailed.

8.2.3 Automatic validation of the business rules

Our conceptual framework does not include an automatic validation for the business rules extracted. Currently, the rules are validated by the business analysts/developers. Unfortunately, this activity is generally error-prone and time-consuming, since the number of discovered rules can be huge and the analyst/developer might not have a clear knowledge of the system.

Therefore, we would like to implement an automatic validation step that looks for inconsistencies and duplicated rules. We believe that this step can be implemented in two ways. On the one hand, if all extracted rules were expressed in OCL and applied on a UML model representing the whole system, the use of Constraint Satisfaction Problem techniques and the related works in MDE (e.g., [56]) could help to implement this new step. On the other hand, the extracted rules could be transformed in a suitable format to make them run in a Business Rule Management System (BRMS); since the facilities provided by such systems often include special

environment to test and validate the rules [1].

8.2.4 Rule-driven system redocumentation

The documentation related to source code of legacy systems is often poor and not aligned with the current implementations of such systems. Hence, without (or with wrong) hints from the comments, understanding the source code turns into a complex maintenance task. In order to ease such understanding, we believe that the information contained in the extracted business rules can help to redocument the source code. In particular, such information can be used to generate comments on the statements composing the rules and to derive further information about the business processes embedded in a given system.

In addition, relying on model-driven techniques, we can perform this redocumentation in a non-intrusive way, since the comments can be added to the model, that can be later used to regenerate the corresponding system/programs.

1. http://tinyurl.com/IBMDecisionManagerRuleVal

Bibliography

[1] Hay, D., Healy, K.A., Hall, J.: Defining Business Rules – What Are They Really? 11, 16, 17, 30, 37, 38

[2] Sneed, H.M., Erdös, K.: Extracting Business Rules from Source Code. In: Fourth Workshop on Program Comprehension. (1996) 240–247 12, 19, 32, 40, 49, 122, 125, 138, 140

[3] Rugaber, S.: Program comprehension. Encyclopedia of Computer Science and Technology **35**(20) (1995) 341–368 12, 32

[4] Huang, H., Tsai, W., Bhattacharya, S., Chen, X., Wang, Y., Sun, J.: Business Rule Extraction from Legacy Code. In: Computer Software and Applications Conference. (1996) 162–167 12, 19, 32, 49, 121, 125, 126, 138, 140

[5] Putrycz, E., Kark, A.W.: Recovering Business Rules from Legacy Source Code for System Modernization. In: Advances in Rule Interchange and Applications. (2007) 107–118 12, 19

[6] Wang, X., Sun, J., Yang, X., He, Z., Maddineni, S.: Business Rules Extraction from Large Legacy Systems. In: European Conference on Software Maintenance and Reengineering. (2004) 249–254 12, 19, 32, 49, 126, 127, 128, 139, 140

[7] Henrard, J., Englebert, V., Hick, J.M., Roland, D., Hainaut, J.L.: Program Understanding in Databases Reverse Engineering. In: Database and Expert Systems Applications. (1998) 70–79 13, 33, 128, 129

[8] Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering **1**(1) (2012) 1–182 13, 33, 42

[9] Mohagheghi, P., Fernandez, M.A., Martell, J.A., Fritzsche, M., Gilani, W.: MDE adoption in industry: challenges and success criteria. In: Models in Software Engineering. (2009) 54–59 13, 43

[10] Booch, G., Jacobson, I., Rumbaugh, J.: OMG unified modeling language specification. Object Management Group (2000) 1034 15, 44

[11] Pérez-Castillo, R., De Guzman, I.G.R., Piattini, M.: Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. Computer Standards & Interfaces **33**(6) (2011) 519–532 15, 44, 125

[12] Galvão I., Goknil, A.: Survey of Traceability Approaches in Model-Driven Engineering. In: Enterprise Distributed Object Computing Conference. (2007) 313–326 15, 45

[13] Jouault, F.: Loosely Coupled Traceability for ATL. In: European Conference on Model Driven Architecture workshop on traceability. (2005) 29–37 15, 45

[14] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39 16, 45, 74, 96

[15] OMG: OMG, MOF and Specification, QVT Final Adopted (2007) 16, 45

[16] Gottesdiener, E.: Capturing business rules. Software Development Magazine **7**(12) (1999) 16, 37, 38

[17] Baxter I., H.S.: A Standards-Based Approach to Extracting Business Rules. http://www.semdesigns.com/Company/Publications/ExtractingBusinessRules.pdf 16, 37, 120, 121

[18] IBM: WebSphere Integration Developer: Business rule. http://tinyurl.com/BusinessRuleDefForIBM 16, 38

[19] IBM: WebSphere ILOG Rules: What are business rules. http://tinyurl.com/BusinessRuleForIBM 16, 38

[20] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, & tools. (2007) 18, 41

[21] Demuth, B., Hußmann, H.: Using UML/OCL Constraints for Relational Database Design. In: The Unified Modeling Language. (1999) 598–613 18, 41, 135, 137, 140

[22] Chaparro, O., Aponte, J., Ortega, F., Marcus, A.: Towards the Automatic Extraction of Structural Business Rules from Legacy Databases. In: Working Conference on Reverse Engineering. (2012) 479–488 20, 49, 136, 140

[23] Cabot, J., Gómez, C., Planas, E., Rodríguez, M.E.: Reverse Engineering of OO constructs in Object-Relational Database Schemas. Jornadas de IngenierÃa del Software y Bases de Datos (2008) 20, 49, 134, 140

[24] Bézivin, J., Kurtev, I.: Model-based Technology Integration with the Technical Space Concept. In: Metainformatics Symposium. (2005) 20, 44, 49

[25] Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: IEEE/ACM international conference on Automated software engineering. (2010) 173–174 21, 51

[26] Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: ACM international conference companion on Object oriented programming systems languages and applications companion. (2010) 307–309 21

[27] Allen, F.E.: Control flow analysis. SIGPLAN Not. **5**(7) (1970) 1–19 23, 53

[28] Mahe, V., Martinez Perez, S., Doux, G., Brunelière, H., Cabot, J.: PORTOLAN: a Model-Driven Cartography Framework. Technical report (2011) 27, 56

[29] O'Brien, M.P.: Software comprehension: a review & research direction. Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report (2003) 33

[30] Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering **32**(12) (2006) 971–987 34, 35

[31] Korel, B., Rilling, J.: Program slicing in understanding of large programs. In: International Workshop on Program Comprehension. (1998) 145–152 35

[32] De Lucia, A.: Program slicing: Methods and applications. In: Source Code Analysis and Manipulation. (2001) 142–149 35

[33] Weiser, M.: Program slicing. In: International Conference on Software Engineering. (1981) 439–449 35, 62, 80

[34] Tip, F.: A Survey of Program Slicing Techniques. Journal of programming languages **3**(3) (1995) 121–189 35

[35] Allen, F.E.: Control flow analysis. In: ACM Sigplan Notices. (1970) 1–19 35

[36] Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters **29**(3) (1988) 155–163 35

[37] Korel, B., Yalamanchili, S.: Forward Computation of Dynamic Program Slices. In: ACM SIGSOFT international symposium on Software testing and analysis. (1994) 66–79 36

[38] Eisenberg, A., Melton, J., Kulkarni, K.G., Michels, J.E., Zemke, F.: SQL: 2003 has been published. SIGMOD Record **33**(1) (2004) 119–126 37, 98

[39] Widom, J., Ceri, S.: Active Database Systems: Triggers and Rules for Advanced Database Processing. (1996) 37

[40] Gogolla, M., Richters, M.: On constraints and queries in UML. In: The Unified Modeling Language. (1998) 109–121 37, 133, 140

[41] Warmer, J., Kleppe, A.: The object constraint language: getting your models ready for MDA. (2003) 37

[42] Chapin, D., Baisley, D., Hall, H.: Semantics of business vocabulary & business rules (SBVR). Hawke et al (2005) 37, 121

[43] Scowen, R.S.: Extended BNF-a generic base standard. Technical report (1998) 43

[44] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Model Driven Architecture-Foundations and Applications. (2009) 18–33 45

[45] Felix Lösch, J.L., Schmidberger, R.: Instrumentation of Java Program Code for Control Flow Analysis (2004) 47, 127, 128, 139, 140

[46] Jones, J.: Abstract syntax tree implementation idioms. Pattern Languages of Program Design (2003) 51

[47] Murach, M., Prince, A., Menendez, R.: Murach's Mainframe COBOL. (2004) 57

[48] Eckel, B., Allison, C.: Thinking in JAVA. (2003) 77

[49] Castagna, G.: Object-oriented programming. (1996) 77

[50] Oracle: Coding Triggers. http://docs.oracle.com/cd/B19306_01/appdev.102/b14251/adfns_triggers.html 99

[51] Nanda, A., Feuerstein, S.: Oracle PL/SQL for DBAs. (2009) 99, 136

[52] Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A definitive guide. In: Formal Methods for Model-Driven Engineering. (2012) 58–90 100

[53] Yeh, D., Li, Y., Chu, W.C.: Extracting entity-relationship diagram from a table-based legacy database. Journal of Systems and Software **81**(5) (2008) 764–771 105

[54] Cabot, J., Mazón, J.N., Pardillo, J., Trujillo, J.: Specifying aggregation functions in multidimensional models with OCL. In: International Conference on Conceptual Modeling. (2010) 419–432 109, 112

[55] Olivé, A.: Integrity Constraints Definition in Object-Oriented Conceptual Modeling Languages. In: International Conference on Conceptual Modeling. (2003) 349–362 114

[56] González, C.A., Buttner, F., Clarisó, R., Cabot, J.: Emftocsp: A tool for the lightweight verification of emf models. In: Software Engineering: Rigorous and Agile Approaches. (2012) 44–50 115, 143

[57] Cabot, J., Olivé, A., Teniente, E.: Representing Temporal Information in UML. In: The Unified Modeling Language. (2003) 44–59 115

[58] Ramsey, F.V., Alpigini, J.J.: A simple mathematically based framework for rule extraction from an arbitrary programming language. In: International Computer Software and Applications Conference. (2002) 763–770 119, 121

[59] Vasilecas, O., Normantas, K.: Deriving business rules from the models of existing information systems. In: International Conference on Computer Systems and Technologies. (2011) 95–100 120, 121

[60] Kohavi, R.: The power of decision tables. In: European Conference on Machine Learning. (1995) 174–189 121

[61] Sneed, H.M.: Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment. In: International Workshop on Program Comprehension. (2001) 167–175 122, 125, 138, 140

[62] Putrycz, E., Kark, A.W.: Connecting legacy code, business rules and documentation. In: Rule Representation, Interchange and Reasoning on the Web. (2008) 17–30 123, 125, 126, 138, 140

[63] Chiang, C.C.: Extracting business rules from legacy systems into reusable components. In: International Conference on Systems, Man and Cybernetics. (2006) 350–355 123, 125, 139, 140

[64] Chia-Chu Chiang, Bayrak, C.: Legacy Software Modernization. In: IEEE International Conference on Systems, Man and Cybernetics. (2006) 1304–1309 123, 125, 139, 140

[65] Pope, A.L.: The CORBA reference guide: understanding the common object request broker architecture. (1998) 123

[66] Fu, G., Shao, J., Embury, S.M., Gray, W.A., Liu, X.: A Framework for Business Rule Presentation. In: Database and Expert Systems Applications. (2001) 922– 124, 125, 139, 140

[67] Earls, A.B., Embury, S.M., Turner, N.H.: A method for the manual extraction of business rules from legacy source code. BT Technology Journal **20**(4) (2002) 127–145 124, 125, 139, 140

[68] Barbier, F., Deltombe, G., O., P., Youbi, K.: Model Driven Reverse Engineering: Increasing Legacy Technology Independence. In: Second India Workshop on Reverse Engineering. (2011) 125, 126, 139, 140

[69] Wang, X., Sun, J., Yang, X., He, Z., Maddineni, S.: Application of information-flow relations algorithm on extracting business rules from legacy code. In: Intelligent Control and Automation. (2004) 3055–3058 126

[70] Wang, C., Zhou, Y., Chen, J.: Extracting Prime Business Rules from large legacy system. In: Canadian Conference on Computer Science & Software Engineering. (2008) 19–23 127, 128, 139, 140

[71] Normantas, K., Vasilecas, O.: Business Rules Discovery from Existing Software Systems. International Journal of Scientific & Engineering Research **3** (2012) 127, 128, 139, 140

[72] Sumner, M.: Enterprise Resource Planning. (2007) 127

[73] Vanthienen, J.: Ruling the business: about business rules and decision tables. New Directions in Software Engineering (2001) 103–120 127

[74] Huang, J.: Program instrumentation and software testing. Computer **11**(4) (1978) 25–32 128

[75] Chiang, R.H.L., Barron, T.M., Storey, V.C.: Reverse Engineering of Relational Databases: Extraction of an EER Model from a Relational Database. Data & Knowledge Engineering **12**(2) (1994) 107–142 128, 129

[76] Andersson, M.: Extracting an entity relationship schema from a relational database through reverse engineering. In: International Conference on Conceptual Modeling. (1994) 403–419 128, 129

[77] Paradauskas, B., L.A.: Business Knowledge extraction from Legacy Information Systems. Journal of Information Technology and Control **35**(3) (2006) 214–221 128, 130

[78] Premerlani, W.J., Blaha, M.R.: An approach for reverse engineering of relational databases. In: Working Conference on Reverse Engineering. (1993) 151–160 128, 131

[79] Alalfi, M.H., Cordy, J.R., Dean, T.R.: SQL2XMI: Reverse engineering of UML-ER diagrams from relational database schemas. In: Working Conference on Reverse Engineering. (2008) 187–191 128, 131

[80] Chung, S., Hartford, E.: Bridging the Gap between Data Models and Implementations: XMI2SQL. In: International Conference on Internet and Web Applications and Services. (2006) 201 128, 131

[81] Ramanathan, S., Hodges, J.: Extraction of object-oriented structures from existing relational databases. ACM Sigmod Record **26**(1) (1997) 59–64 128, 131

[82] Fong, J.: Converting relational to object-oriented databases. ACM SIGMOD Record **26**(1) (1997) 53–58 128, 132

[83] Fahrner, C., Vossen, G.: Transforming relational database schemas into object-oriented schemas according to ODMG-93. Deductive and Object-Oriented Databases (1995) 429–446 128, 132

[84] Balsters, H.: Modelling database views with derived classes in the UML/OCL-framework. In: The Unified Modeling Language. (2003) 295–309 128, 133

[85] Codd, E.F.: Further normalization of the data base relational model. Data base systems (1972) 33–64 131

[86] Mandel, L., Cengarle, M.V.: On the expressive power of OCL. In: Formal Methods. (1999) 854–874 133, 140

[87] Codd, E.F.: Relational completeness of data base sublanguages. (1972) 133

[88] Akehurst, D.H., Bordbar, B.: On querying UML data models with OCL. In: The Unified Modeling Language. (2001) 91–103 134, 140

[89] Siripornpanit, N., Leckcharoen, S.: An Adaptive Algorithms Translating and Back-Translating of Object Constraint Language into Structure Query Language. In: International Conference on Information and Multimedia Technology. (2009) 149–151 135, 140

[90] Rybola, Z., Richta, K.: Transformation of Special Multiplicity Constraints - Comparison of Possible Realizations. In: Federated Conference on Computer Science and Information Systems. (2012) 1357–1364 136, 140

[91] Rabben, M.: Extracting Business Rules from PL/SQL-Code. http://www.semantec.de/en/pdf/Extracting%20Business%20Rules.pdf 136, 140

[92] Demuth, B., Hußmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: The Unified Modeling Language. (2001) 104–117 137, 140

[93] Berrabah, D., Boufarès, F.: Constraints Checking in UML Class Diagrams: SQL vs OCL. In: Database and Expert Systems Applications. (2007) 593–602 137, 140

List of Tables

List of Figures

List of Publications

[1] Cosentino, V., Cabot, J., Albert, P., Bauquel, P., Perronnet, J.: A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application. In: Rules on the Web: Research and Applications. (2012) 17–31

[2] Cosentino, V., Bauquel, P., Perronnet, J., Albert, P., Cabot, J.: Un Framework dirigé par les modèles pour l'extraction de règles métier à partir d'applications COBOL. In: Conférence en IngénieriE du Logiciel. (2013)

[3] Cosentino, V., Cabot, J., Albert, P., Bauquel, P., Perronnet, J.: Extracting Business Rules from COBOL: A Model-Based Framework. In: Working Conference on Reverse Engineering. (2013, to appear)

[4] Cosentino, V., Cabot, J., Albert, P., Bauquel, P., Perronnet, J.: Extracting Business Rules from COBOL: A Model-Based Tool. In: Working Conference on Reverse Engineering. (2013, to appear)

[5] Cosentino, V., Martínez, S.: Extracting UML/OCL Integrity Constraints and Derived Types from Relational Databases. In: Workshop on OCL and Textual Modelling. (2013, to appear)

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Thèse de Doctorat

Valerio COSENTINO

Une Approche dirigée par les Modéles pour l'Extraction de Règles Métier à partir des Systèmes d'Informations hérités

A Model-Based Approach for Extracting Business Rules out of Legacy Information Systems

Résumé
Le monde des affaires d'aujourd'hui est très dynamique, donc les organisations doivent rapidement adapter leurs politiques commerciales afin de suivre les évolutions du marché. Ces ajustements doivent être propagés à la logique métier présente dans les systèmes d'informations des organisations, qui sont souvent des applications héritées non conçues pour représenter et opérationnaliser la logique métier indépendamment des aspects techniques du langage de programmation utilisé. Par conséquent, la logique métier integrée au sein du système doit être identifiée et comprise avant d'être modifiée. Malheureusement, ces activités ralentissent la mise à jour du système vers de nouvelles exigences établies dans les politiques de l'organisation et menacent la cohérence des activités commerciales de celle-ci.
Afin de simplifier ces activités, nous offrons une approche basée sur les modèles pour extraire et représenter la logique métier, exprimée comme un ensemble de règles de gestion, à partir des parties comportementales et structurelles des systèmes d'information. Nous mettons en œuvre cette approche pour les systèmes écrits en Java et COBOL ainsi que pour les systèmes de gestion de bases de données relationnelles. L'approche proposée est basée sur l'Ingénierie Dirigée par les Modèles, qui fournit une solution générique et modulaire adaptable à différentes langages en offrant une représentation abstraite et homogène du système.

Mots clés
Ingénierie Dirigée par les Modèles, Extraction des Règles Métiers, Rétro-ingénierie.

Abstract
Today's business world is very dynamic and organizations have to quickly adjust their internal policies to follow the market changes. Such adjustments must be propagated to the business logic embedded in the organization's information systems, that are often legacy applications not designed to represent and operationalize the business logic independently from the technical aspects of the programming language employed. Consequently, the business logic buried in the system must be discovered and understood before being modified. Unfortunately, such activities slow down the modification of the system to new requirements settled in the organization policies and threaten the consistency and coherency of the organization business.
In order to simplify these activities, we provide a model-based approach to extract and represent the business logic, expressed as a set of business rules, from the behavioral and structural parts of information systems. We implement such approach for Java, COBOL and relational database management systems. The proposed approach is based on Model Driven Engineering, that provides a generic and modular solution adaptable to different languages by offering an abstract and homogeneous representation of the system.

Key Words
Model Driven Engineering, Business Rules Extraction, Reverse Engineering.