VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Kęstutis NORMANTAS

RESEARCH ON BUSINESS KNOWLEDGE EXTRACTION FROM EXISTING SOFTWARE SYSTEMS

DOCTORAL DISSERTATION

TECHNOLOGICAL SCIENCES,
INFORMATICS ENGINEERING (07T)

VGTU
LEIDYKLA
TECHNIKA
Vilnius 2013

Doctoral dissertation was prepared at Vilnius Gediminas Technical University in 2009–2013.

Scientific Supervisor

Prof Dr Olegas VASILECAS (Vilnius Gediminas Technical University, Technological Sciences, Informatics Engineering – 07T).

VGTU leidyklos TECHNIKA 2197-M mokslo literatūros knyga
http://leidykla.vgtu.lt

ISBN 978-609-457-594-5

kestutis.normantas@vgtu.lt

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Kęstutis NORMANTAS

VERSLO ŽINIŲ IŠGAVIMO IŠ EGZISTUOJANČIŲ PROGRAMŲ SISTEMŲ TYRIMAS

DAKTARO DISERTACIJA

TECHNOLOGIJOS MOKSLAI,
INFORMATIKOS INŽINERIJA (07T)

VGTU
LEIDYKLA
TECHNIKA
Vilnius 2013

Disertacija rengta 2009–2013 metais Vilniaus Gedimino technikos universitete.

Mokslinis vadovas

prof. dr. Olegas VASILECAS (Vilniaus Gedimino technikos universitetas, technologijos mokslai, informatikos inžinerija – 07T).

Abstract

The dissertation addresses the problem of software maintenance and evolution. It identifies that spending within these software lifecycle phases may account for up to 80% of software's total lifecycle cost, whereas the inability to adopt software quickly and reliably to meet ever-changing business requirements may lead to business opportunities being lost. The main reason of this phenomenon is the fact that the most of maintenance effort is devoted to understanding the software to be modified. On the other hand, related studies show that less than one-third of software source code contains business logic implemented within it, while the remaining part is intended for platform or infrastructure relevant activities. It follows that if the most of changes in software are made due to the need to adopt its functionality to changed business requirements, then facilitating software comprehension with automated business knowledge extraction methods may significantly reduce the cost of software maintenance and evolution. Therefore the main goal of this thesis is to improve business knowledge extraction process by proposing a method and supporting tool framework that would facilitate comprehension of existing software systems.

The dissertation consists of the following parts: Introduction, 4 chapters, General Conclusions, References, and 6 Annexes.

Chapter 1 presents a systematic literature review of related studies in order to summarize the state-of-the art in this research field, identify any gaps in the current research and explore possible directions for the further research. Chapter 2 formulates theoretical background for the business knowledge extraction method by introducing selected standard for the intermediate knowledge representation, defining well-formedness rules for this representation, and by revising and applying static program analysis techniques to this representation. Chapter 3 describes the proposed method for automated business knowledge extraction from existing software systems and introduces the supporting tools framework. Chapter 4 presents the case study on applying the method for knowledge extraction from the existing enterprise content management system, and evaluates study results in respect with the precision, recall, and accuracy measures. The evaluation shows that the proposed method is feasible and efficient enough to be further improved and applied in practice. The main observations are summarised and concluded within the General Conclusions chapter.

Reziumė

Darbe nagrinėjama programų sistemų palaikymo ir vystymo problema. Nustatyta, jog sąnaudos šiose programų sistemos gyvavimo ciklo fazėse siekia iki 80% visų sąnaudų, skiriamų programų sistemai kurti. Pagrindinis šio reiškinio veiksnys yra nuolatinis poreikis pritaikyti sistemų funkcionalumą prie besikeičiančių verslo reikalavimų, o tokios užduotys apima didžiąją dalį visų palaikymo veiklų. Nagrinėti tyrimai parodė, kad programų sistemose įgyvendintai verslo logikai suprasti sugaištama 40–60% pakeitimams atliki skirto laiko, kadangi atsakingi už sistemų palaikymą žmonės paprastai nėra jų projektuotai, todėl turi dėti dideles pastangas, kad išsiaiškintų sistemos veikimo principus. Be to, pakeitimai, atliekami palaikymo metu, yra retai dokumentuojami (ar net nedokumentuojami visai), o supratimas įgytas įgyvendinant pakeitimus lieka individualių programuotojų galvose. Tuo tarpu kiti tyrimai atskleidė, jog paprastai tik trečdalis programų sistemos kodo įgyvendina verslo logiką, o kita dalis yra skirta platformos ir infrastruktūros funkcijoms įgyvendinti. Iš to darytina išvada, jog išgaunant dalykinės srities žinias bei išlaikant atsekamumą tarp jų ir jas įgyvendinančio programinio kodo, galima sumažinti sistemų palaikymo ir vystymo kaštus. Todėl pagrindinis šio darbo tikslas yra patobulinti verslo žinių išgavimo ir vaizdavimo procesą, pasiūlant metodą ir palaikančias priemones, kurios palengvintų egzistuojančių programų sistemų suvokimą.

Darbas susideda iš įvado, 4 dalių, bendrųjų išvadų, literatūros sąrašo bei 6 priedų.

Pirmoje darbo dalyje pateikiama sisteminė susijusios literatūros apžvalga, kuria siekiama apibendrinti tyrimus šioje srityje, nustatyti trūkumus bei įžvelgti naujas tyrimų kryptis. Antroje darbo dalyje formuojamas siūlomo metodo teorinis pagrindas. Apžvelgiamas pasirinktas meta-modelis žinioms vaizduoti, šio meta-modelio kontekste apibrėžiami taisyklingų modelių sudarymo principai bei apibrėžiamos statinės programų sistemų analizės technikos, kurios naudojamos žinioms išgauti ir išgryninti. Trečioje darbo dalyje pristatomas metodas automatizuotam verslo žinių išgavimui iš egzistuojančių programų sistemų bei apibūdinamas palaikančių priemonių karkasas. Ketvirtoje darbo dalyje pristatomas siūlomo metodo pritaikymo žinioms išgauti iš įmonės turinio valdymo sistemos tyrimas, kuriuo siekiama patikrinti metodo įgyvendinamumą bei įvertinti jo efektyvumą. Tyrimo rezultatų įvertinimas tikslumo, patikimumo bei atkuriamumo aspektais parodo, jog pasiūlytas metodas yra pakankamai efektyvus, jog galėtų būti tobulinamas ir pritaikomas praktikoje. Darbas apibendrinamas ir padaromos bendrosios darbo išvados paskutiniajame darbo skyriuje.

Notation

Abbreviations

ADM	−	Architecture Driven Modernization;
ANSI	−	American National Standardization Institute;
AST	−	Abstract Syntax Tree;
AST	−	Abstract Syntax Tree Meta-model;
BNF	−	Backus-Naur Form;
BP	−	Business Process;
BPM	−	Business Process Modelling Notation;
BR	−	Business Rule;
CFG	−	Control Flow Graph;
CG	−	Call Multigraph;
COB	−	COmmon Business-Oriented Language;
COR	−	Common Object Request Broker Architecture;
CRM	−	Customer Relationship Management;
CSV	−	Comma-Separated Values Format;
DAG	−	Directed Acyclic Graph;
DAT	−	Data-Access Technology Format;
DCO	−	Distributed Component Object Model;
DT	−	Decision Table;
ECM	−	Enterprise Content Management;
ERP	−	Enterprise Resource Planning;
IR	−	Intermediate Representation;
IS	−	Information System;

ISO	−	International Organization for Standardization;
JDBC	−	Java interface to ODBC;
JNI	−	Java Native Interface;
KDM	−	Knowledge Discovery Meta-model;
LOC	−	Lines Of Code;
MDA	−	Model Driven Architecture;
MOF	−	Meta-Object Facility;
OCL	−	Object Constraint Language;
ODB	−	Object to DataBase Connectivity;
OMG	−	Object Management Group;
ORM	−	Object-Role Modelling;
PCG	−	Program Control Graph;
PDG	−	Program Dependency Graph;
PFG	−	Program Flow Graph;
QVT	−	Query-View-Transform;
RMI	−	Remote Method Invocation Interface;
SBV	−	Semantics of Business Vocabulary and Business Rules;
SDG	−	System Dependency Graph;
SLR	−	Systematic Literature Review;
SOA	−	Simple Object Access Protocol;
SQL	−	Structured Query Language;
SS	−	Software System;
SSA	−	Static Single Assignment;
UI	−	User Interface;
UML	−	Unified Modelling Language;
XMI	−	XML Metadata Interchange;
XML	−	eXtensible Markup Language.

Contents

Introduction

The Investigated Problem

Software maintenance and evolution are integral parts of software life cycle (IEEE 2008). According to the first Lehman's law of software evolution (Lehman 1980), once the software is in production, it undergoes continual change or becomes progressively less useful, and the change or decay process continues until it is judged more cost effective to replace the system with a recreated version.

A comprehensive research on software maintenance and evolution by Bennett and Rajlich (Bennett, Rajlich 2000) emphasizes the importance of software maintenance as it consumes a large part of the overall life-cycle costs, whereas the inability to change software quickly and reliably means that business opportunities are lost. Studies by (Canfora, Cimitile 1995) and (Seacord *et al*. 2003) have shown that the cost of software maintenance may account for up to 80% of its total life-cycle cost. The recent forecast analysis by Gartner indicates that only software maintenance consumes 38% of total cost of spending for software systems (Columbus 2013).

Numerous studies have determined that the most of software maintenance and evolution activities are caused by the need to adopt a system to ever-changing business requirements (enhancements). Although very old, but widely

cited study by Lientz and Swanson (Lientz, Swanson 1980) revealed that software enhancements to meet customer needs comprise 41.8% of the total effort spent for software maintenance. The recent study by Glass (Glass 2012) points out that enhancement is responsible for roughly 60% of software maintenance costs, indicating that this factor is increasing.

One of the main reasons of high software maintenance cost is limited understanding about its initial design and its actual implementation. As the SWEBOK (Abran *et al*. 2004) notes, 40% to 60% of the maintenance effort is devoted to understanding the software to be modified. This is due to the fact, that typically software maintainers are not its designers, so they must expend many resources to examine and learn about the system (Chikofsky, Cross 1990). Moreover, the changes are rarely well documented or even not documented at all (which is often the case) and the comprehension acquired in producing changes often remains with individual developers. Consequently existing systems are hard to modify and the modifications are hard to validate (Baxter, Mehlich 2000).

As the recent study by Ulrich and Newcomb on modernizing information systems (Ulrich, Newcomb 2010) notes, less than 30% of software source code contains business logic, while the remaining code supports infrastructure-related activities. It follows that, if the large part of software changes are due to the need to adopt its functionality to the changed business requirements, then facilitating software comprehension with automated business knowledge extraction methods may significantly reduce the cost of software maintenance and evolution. This hypothesis has been investigated by many researches during the past several decades resulting in numerous methods for business knowledge extraction from the source code of existing software systems.

However, the process of business knowledge extraction from software systems is particularly complex because even in a well-designed system business-domain specific concepts are rarely implemented within a single programming language constructs. On the contrary, they are implemented by multiple programming and/or definitional language constructs that are spread over various layers of the software architecture (e.g. database, application logic, and user interface layer), thus requiring exhaustive and rigorous knowledge discovery methods that would facilitate comprehension of the software.

Importance of the Thesis

The Architecture Driven Modernization (ADM), a recent initiative of Object Management Group (OMG 2012), aims at developing a number of standards to facilitate modernization activities by enabling software analysis and comprehension, and by allowing software transformations. The Knowledge Discovery Met-

amodel (KDM) (OMG 2011a) (adopted as ISO/IEC 19506:2012(E)) plays a key role in this set of standards, as it defines a set of architectural views (i.e. models) of many different aspects of a software system: from the inventory (physical artefacts and their distribution) and source content, to the code structure and behaviour, to the resource and data definitions, to the higher level abstractions – conceptual views. A number of organisations reported significant cost savings by applying this technology in their modernisation projects (Ulrich, Newcomb 2010). For example, the case study on modernization project at Department of Navy shows that using knowledge extraction, analysis and transformation tools and techniques, the modernization project ROI can be at least 2.47 times greater than by using the manual alternative. The case study at Italian ministry of Instruction, University and Research reports cost reduction by the factor of 27% due to employment of architecture-driven modernization approach.

Although there are numerous successful modernization case studies in practice, there is a lack of methods that rely on these standards and propose different approaches for knowledge discovery, representation, and analysis, thus aggravating invention of different tools that would support software modernization process.

The Object of Research

The object of this research is the process of business knowledge extraction from existing software systems and representation of extracted knowledge in form of business vocabulary, rules, and business use cases.

The Goal of the Thesis

This thesis aims at improving the process of business knowledge extraction from existing software systems by proposing a method and supporting tool framework.

The Tasks of the Thesis

In order to achieve the desired goal, the following tasks were established:

1. To analyse methods for business knowledge extraction from existing software systems for the purpose of identifying what kinds of business knowledge are being extracted, what techniques are being used as a basis for knowledge extraction, what types of software artefacts are being

used as input sources, and how extracted knowledge is being represented.
2. To define the formal background for the business knowledge extraction method.
3. To propose and define the method for business knowledge extraction from existing software systems and develop a supporting tool framework.
4. To evaluate the feasibility and efficiency of the proposed method by conducting a case study on the existing software system.

Research Methodology

- This thesis follows the Design Science methodology guidelines (Hevner *et al*. 2004).
- The overview of related methods follows the Systematic Literature Review guidelines (Kitchenham, Charters 2007).
- The evaluation of the method is performed by conducting a case study.

Scientific Novelty

The main contribution of this thesis is a novel method for Business Knowledge Extraction from existing Software Systems (BKESS). The method is based on the Knowledge Discovery Meta-model (KDM), recently adopted as ISO standard (ISO/IEC 19506:2012(E)).

1. The method involves a set of transformations to obtain intermediate representation of software system within the KDM.
2. The method introduces a set of extensions for the Knowledge Discovery Meta-model to allow application of program analysis and pattern matching techniques in order to support business knowledge extraction.
3. The method provides a number of algorithms for the implementation of program analysis and pattern matching techniques within the KDM.
4. The method introduces set of extensions to supplement definition of business vocabulary, rules and use cases within the KDM.
5. The method enables transformation from the intermediate representation within the KDM to representation forms relevant for business analysts and stakeholders.

The early results of this research have been applied in the high technology development program for 2007–2009 project “Business Rules Solutions for In-

formation Systems Development (VeTIS)", supported by Lithuanian State Science and Studies Foundation.

Practical Significance of Achieved Results

The method is intended to facilitate the maintenance and evolution of existing software systems and reduce the costs of these software lifecycle phases. Moreover, systematically obtained business knowledge allows revising business requirements for the information system and involving less qualified specialists in software maintenance and evolution activities.

Selection of the KDM as an infrastructure for intermediate representation and analysis of extracted knowledge ensures scalability of the method. The method supporting tool framework is designed to be integrated with the software development environment and to allow introduction of supplementary tools.

The results of case study on application of the method for knowledge extraction from enterprise content management system show the feasibility and efficiency of the proposed method and emphasize the practical significance of the results.

The Defended Statements

1. A set of KDM constraints formally defined within this work ensures producing well-formed Code and Data models of existing software systems.
2. A set of KDM extensions provided in this work enables application of program analysis techniques for extracting and representing business logic implemented within software systems.
3. The proposed method facilitates business knowledge extraction from existing software systems and ensures traceability between extracted knowledge and its implementation.

Approval of the Results

The results of this thesis were published in 3 scientific journals and 11 peer-reviewed conference proceedings and collections. The results were presented in 13 international and Lithuanian scientific conferences:

- International Conference of Information and Software Technologies ICIST 18, Kaunas, Lithuania, 2012;

- International Conference on Computer Systems and Technologies CompSysTech'12, Ruse, Bulgaria, 2012;
- International Conference on Computer Systems and Technologies CompSysTech'11, Vienna, Austria, 2011;
- International Conference on Computer Systems and Technologies CompSysTech'10, Sofia, Bulgaria, 2010;
- 6th International Conference "E-learning and the Knowledge Society", Riga, Latvia, 2010;
- Informatics: 13th Lithuanian junior scientist conference "Science – Future of Lithuania", Vilnius, 2010;
- Information technology, data analysis and modelling: 12th republic junior scientist conference "Fundamental research and innovations in juncture of sciences", Klaipeda, Lithuania, 2009;
- International Conference on Computer Systems and Technologies CompSysTech'09, Ruse, Bulgaria, 2009;
- Informatics: 12th Lithuanian junior scientist conference "Science – Future of Lithuania", Vilnius, Lithuania, 2009;
- 15th international conference on Information and Software Technologies (Information Technology), Kaunas, Lithuania, 2009 (3 presentations);
- Scientific-practical conference "Academic youth aims: economical, management and technological insights", Klaipeda, Lithuania, 2009.

Dissertation Structure

The dissertation consists of Introduction, 4 chapters, General Conclusions, and 6 Annexes. Chapter 1 presents the systematic literature review of related methods. Chapter 2 defines formal background for the business knowledge extraction method. Chapter 3 provides description of the method and supporting tool framework implementation. Chapter 4 presents the evaluation of the method in the case study. The main observations are summarised and concluded within the General Conclusions chapter.

The total scope of the dissertation is 162 papers, 52 figures, and 28 tables.

1

Systematic Literature Review of Methods for Business Knowledge Extraction from Existing Software Systems

1.1. Introduction

The recent study by Ulrich on modernizing information systems (Ulrich, Newcomb 2010) reveals that less than 30% of software source code contains business logic, while the remaining code supports infrastructure-related activities. It follows that, if the large part of software changes are due to the need to adopt its functionality to the changed business requirements, then facilitating software comprehension with automated business knowledge extraction methods may significantly reduce the cost of software maintenance and evolution. This hypothesis has been investigated by many researches during the past several decades resulting in numerous methods for business knowledge extraction from existing software systems.

This chapter presents a Systematic Literature Review (SLR) of studies in the field of business knowledge extraction from existing (legacy) software systems. Following the guidelines proposed by Kitchenham (Kitchenham, Charters

2007), the SLR was undertaken in order to: summarise the state-of-the-art in this research field, identify any gaps in current research and explore possible directions for the further research, and provide a framework to appropriately position new research activities in the field of business knowledge extraction from existing software systems. The main part of this chapter has been published in (Normantas, Vasilecas 2013); although research on representing business knowledge was presented in earlier studies (Normantas, Vasilecas, Sosunovas 2009a-g and 2010a-b).

In the following we formulate research questions used to evaluate existing literature, define research methods used to conduct SLR, design the process of SLR and present an overview of the SLR results. After reviewing the methods, we provide a discussion on the results, consider threats to validity of this review, and conclude the review with the indications for the further research.

1.2. Research Questions

In order to undertake the Systematic Literature Review we formulated the following four research questions:

- RQ1: What kinds of business knowledge are being extracted by the research?
- RQ2: What analysis techniques are used as a basis for knowledge extraction?
- RQ3: What types of software artefacts are used as input sources for knowledge extraction?
- RQ4: How the extracted knowledge is being represented?

Business knowledge in this context is considered as a certain understanding of the business domain. The commonly accepted forms for defining and for representing business knowledge are business vocabulary, business rules and business processes.

A **business vocabulary** is defined by the Semantics of Business Vocabulary and Business Rules (SBVR) (OMG 2008) as a *collection of terms and facts that are used in a business for communication*.

A **business rule** is defined by the Business Rules Group (Hay 2000) as "*a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business*". Bajec (Bajec, Krisper 2001; Bajec, Krisper 2005) emphasizes that business rules are evidently important for organization as they describe how they are doing business. Morgan (Morgan 2002) indicates that business rules should be concerned only with the conditions that must apply in a defined state, and emphasises that it is also important to know who may invoke the rule, when and

where the rule must be executed. As von Halle **(**Halle 2001**)** notices, often business rules are inaccessible or unknown, because they are buried in legacy code.

A **business process** is considered as "*a defined set of business activities that represent the steps required to achieve a business objective. It includes the flow and use of information and resources*" (OMG 2011b).

The business vocabulary, rules, and processes are gathered by business analysts into a set of documents that sustains throughout the development of a software system. However, the initially acquired knowledge changes or becomes obsolete after the delivery of the software system to the production.

There are many different formal and semi-formal analysis techniques that may be used in the field of business knowledge extraction. In this review, we are attempting to determine which of analysis techniques (such as program analysis, classification, patterns matching, model transformations, etc.) are used as a basis to formulate knowledge extraction algorithms and how they are used. The selection of certain analysis technique in the knowledge extraction process is partially determined by the software artefacts being analysed. This involves the production generated during the development process of a software system. Besides the source code, there might be included configuration, resource definitions, various types of software documentation, or data. However, this review is not concerned with studies that investigate information retrieval from documents, or business process mining from system logs. In our opinion, these research fields should be reviewed separately.

The business knowledge extraction process may involve several kinds of knowledge representation form: intermediate and output. The former is used during various analysis activities and may include data, control, and dependency graphs, lattices, dependency tables, or meta-models that formally define the knowledge about the software system. The latter is used to present and validate the knowledge with the end-users, such as business analysts and stakeholders. Although there is no standard format for representing business rules, there are many different approaches to define them using rule patterns (Sosunovas, Vasilecas 2007), ontology-based representation (Kalibatiene, Vasilecas 2008; Kalibatiene, Vasilecas 2010), Production Rule Representation format (OMG 2009), decision tables and trees (Kohavi, Sommerfield 1998), (Normantas, Vasilecas 2009b, 2009d, 2009g, 2010a), the Unified Modelling Language (UML) (Booch *et al*. 2005) and Object Constraint Language (UML/OCL) (Warmer, Kleppe 1998; Kleppe *et al*. 1999; Warmer, Kleppe 2003; Nemuraite *et al*. 2008), (Normantas, Vasilecas, Sosunovas 2009a, 2009c, 2009f), the Object-Role Modelling notation (Normantas, Vasilecas 2009c), or Semantics of Business Vocabulary and Rules (SBVR) (Chapin 2008; Skersys *et al*. 2012; Tutkute *et al*. 2013). Business processes are typically represented by process models defined with a certain kind of modelling approach, such as activity flow, data flow, workflow,

communication and interaction diagrams, or certainly gaining more popularity the Business Process Modelling Notation (BPMN) (OMG 2011b; Skersys *et al*. 2012).

We believe that formulated research questions should allow conducting an overview of the-state-of-art in the current research, to get a better understanding of the general principles of knowledge extraction from existing software systems, and to identify existing gaps and potential opportunities for the future research.

1.3. Review Methods

Data Sources and Search Strategy

In order to retrieve relevant research studies and to achieve maximal coverage, we chose to search in digital libraries and in search engines using predefined search string. Due to technical limitations, we had to limit the number of data sources. The following table summarises the data sources used to retrieve the data.

Table 1.1. A list of data sources used to search for relevant papers

Data source	***Subjects***	***Website***
ACM	Software and its engineering; Theory of computation; Information systems; Computing methodologies; Applied computing.	http://dl.acm.org
IEEExplore	Computing & Processing	http://ieeexplore.ieee.org
SpringerLink	Computer Science	http://link.springer.com/
ScienceDirect	Computer Science	http://www.sciencedirect.com
Wiley InterScience	General Computing; Computer Science; Information Science and Technology	http://onlinelibrary.wiley.com/
CiteSeerX	-	http://citeseerx.ist.psu.edu
Google Scholar	-	http://scholar.google.com

Our initial study of selected digital libraries and their particular sections has shown that they contain significant number of books, journals, peer-reviewed conferences and workshops relevant to the research field. We believe that this

list is fair enough to cover sufficiently large amount of relevant studies. Though, it could be extended in the future research by concerning other well-known digital libraries, search engines, or other sources.

We followed the general principles proposed by Brereton (Brereton *et al*. 2007) to define search string. First, we defined major terms by breaking down the research questions specified in previous section. Then, we searched for alternative spellings, abbreviations, synonyms, and related terms in the thesaurus and search engines. In addition, we checked for the keywords and titles in any relevant resource that could be obtained by combining the major terms. After identification of the major and alternative terms, we construct search string by combining the major terms with the alternatives using Boolean OR operator and by combining resulting predicates with each other using Boolean AND operator. The following table summarizes the search terms being used for searching relevant resources.

Table 1.2. A list of major and alternate terms used to construct the search strings

Major terms	***Alternative terms***
Business	Domain
Knowledge	Vocabulary OR Facts OR Terms OR Concepts OR Rules OR Process OR Logic OR Requirements OR Documentation
Extract	Discover OR Retrieve OR Derive OR Gather OR Reverse Engineer OR Re-Engineer OR Recover
Existing	Legacy
Software System	Information System OR Source Code OR Program OR Application

To validate whether the search string is constructed appropriately, we tried to search full-text of publications in several digital libraries and search engines. As we obtained that results are too large to be processed, consisting of hundreds of thousands of records, we limited the search in titles, abstracts, and meta-data (if such feature was provided by digital library/search engine). It should be noted, that not every digital library/search engine supports complex and long search queries. Therefore, we had to have separated the search string into smaller pieces without modifying the conjunction between query predicates and manually manipulate the search to obtain results.

Study Selection

Once the potentially relevant primary studies have been obtained, they need to be assessed for their actual relevance (Kitchenham, Charters 2007). For this reason we formulated inclusion and exclusion criteria. Research works were included in the review list if they met at least one of the following criteria:

- presents an approach for business knowledge extraction from existing software systems OR
- presents a case study on applying a certain business knowledge extraction approach OR
- presents an extension to a certain approach for business knowledge extraction.

Research works were excluded in the review list if they met at least one of the following criteria:

- is not available in the English language OR
- is not available as a full version paper, only an extended abstract or presentation OR
- is a duplicate of the already included paper (we have selected the most recent version) OR
- only discuss on the possibilities to extract business knowledge rather than presents an approach OR
- concerns knowledge extraction from data sources other than software artefacts (i.e. only log mining or information retrieval from documents).

In order to make decision whether to include the obtained work or not, we planned several iterations as follows. During the first iteration, the title, keywords, and abstract or summary of the obtained work were reviewed according to the formulated inclusion and exclusion criteria. During the second iteration, the full-text of the obtained work was reviewed considering inclusion criteria. The obtained work was evaluated by one participant (the first author) and another participant (the second author) validated this evaluation. In case of disagreement, the discussion on the work was held between all participants to decide the inclusion of the work in the review list.

Study Quality Assessment

Although there is no commonly agreed definition of study "quality" (Kitchenham, Charters 2007), it could be in some degree assessed by constructing check-lists of factors that need to be evaluated for each study. For this reason we defined the following questions:

1. How the proposed approach has been evaluated?
   (a) Industrial case study;
   (b) Research case study;
   (c) An example.
2. What is the automation level of the proposed approach?
   (a) Automatic;
   (b) Semi-automatic;
   (c) Manual.

3. How well the approach has been defined?
   (a) Formal enough to be reproduced;
   (b) Informal, but general steps could be reproduced;
   (c) Vaguely, impossible to reproduce.

It should be noted, that the knowledge extraction process in most cases is complex and involves multiple activities (source code parsing, transformations, analysis and refinement, information retrieval from available documentation, and etc.); therefore, it would be unreasonable to expect that all details about the implementation of the approach or its evaluation in a case study could be clearly and particularly specified, especially when there are different limitations, such as the length of paper in journals or conference proceedings. On the other hand, without assessing these criteria, it would be difficult evaluate the quality of selected works.

Data Extraction

In order to accurately record the information obtained by reviewing the works, we defined data extraction form within the references management tool (JabRef).

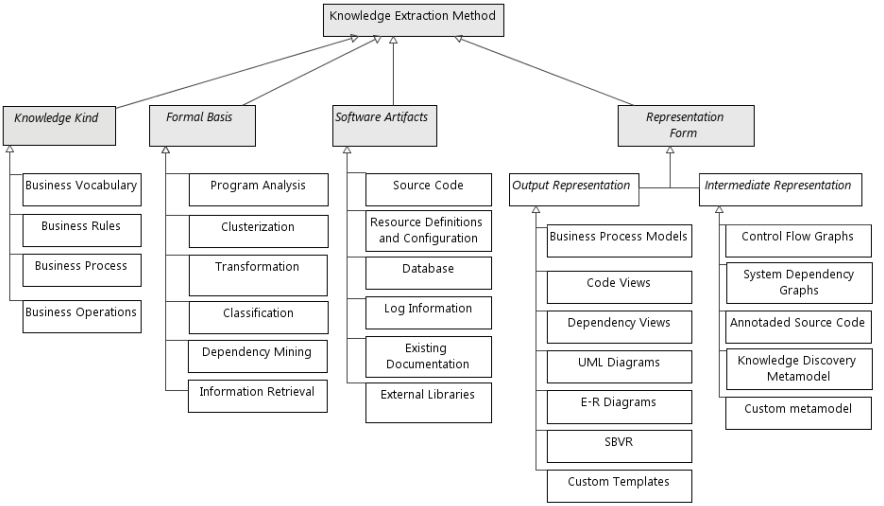

Fig. 1.1. Classification scheme for evaluating knowledge extraction methods

In addition to the default properties of a reference item, such as title, publication type, authors, annotation, etc. we introduced several custom properties to determine quality assessment, and to assign dimensions from the classification

scheme, developed according to the research questions, formulated in Section 1.2. The formulated classification scheme is presented in the figure above (Fig. 1.1).

Data Synthesis

To collate and summarize the results of the included studies we chose the descriptive (narrative) synthesis. Extracted information about the studies was tabulated in consistent with the research questions and the quality evaluation criteria. The summarising table was structured to highlight similarities and differences between selected studies. Section 1.5 presents detailed discussion and summarising table.

1.4. Conducting the Review

The Systematic Literature Review (SLR) was conducted in three phases (Fig. 1.2). At the first phase, over 3500 studies were obtained by searching digital libraries/search engines using the search string or constructing such search string with the functionality provided by the web site of digital library.

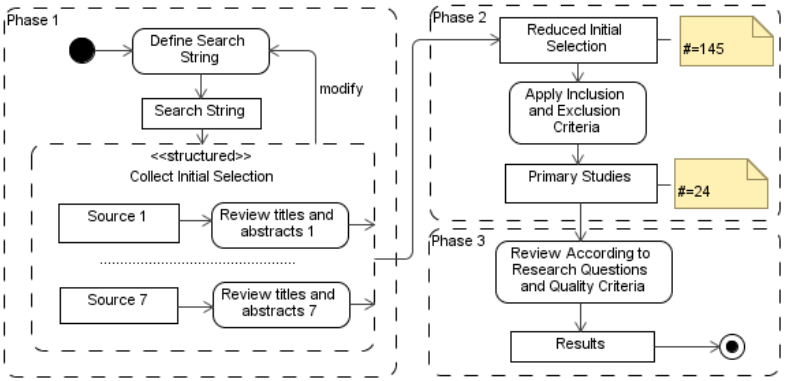

Fig. 1.2. The process of conducting the literature review

The total number of selected studies was reduced to 145 by considering their titles and abstracts for relevance. At the second phase, selected studies were reviewed considering inclusion and exclusion criteria and by removing duplicates. Although the number of studies was reduced to 24, it could be observed that certain studies were published by the same authors and concerned the same method application in different case studies or its extension. At the third phase, we reviewed in detail each of the study according to the research questions and

quality criteria and summarised the results in tabular form by grouping studies on the same method (for example, evaluated in the other case study) and treating them as a single study.

1.5. Results

The distribution of the studies selected within the systematic literature review is presented in the Table 1.3. It could be observed that although reverse engineering is relatively old research field, the steady interest in extraction of business knowledge (such concepts as business rules and processes) from software artefacts has begun only in 2001. Moreover, the overall number of selected studies as well as the considerably lower number of the selected studies in journals than in conference proceedings and collections indicates that this research field is still young and immature.

Table 1.3. Distribution of selected primary studies by year and publication type

Year	*Journal*	*Proceedings*	*Collections*	*Total*
1996	0	2	0	**2**
2001	0	1	0	**1**
2002	1	1	0	**2**
2004	0	3	0	**3**
2006	1	2	0	**3**
2007	0	2	0	**2**
2008	0	0	1	**1**
2009	0	3	0	**3**
2011	2	1	0	**3**
2012	1	1	2	**4**
Total	5	16	3	**24**

We reviewed the selected primary studies in regards with the research questions defined in Section 1.2. There were identified 10 research studies that concerns with extraction of business rules, 6 that in addition to business rules extract business vocabulary (business terms and facts), 6 that concerns with extraction of business processes, and only 2 that considers extraction of both the business rules and business processes. In the following we present a detailed overview of

the selected studies. At the end of this section we present summarizing table (Table 1.4).

1.5.1. Methods for Business Rules Extraction

1.5.1.1. Extracting Business Rules using Program Slicing

The vast majority of business rules extraction approaches are based on techniques for program analysis. The program analysis concerns static techniques for computing information about the behaviour of software systems (Nielson *et al*. 2004). Although primarily intended for optimization of compilers, the program analysis techniques have also been successfully applied in software comprehension. That resulted to the introduction of the concept of program slicing, a set of techniques to extract specific and rigorous knowledge from source code, to be suitably represented and visualized, and to provide basis for further analysis, classification and reconstruction (Rodrigues, Barbosa 2010).

Program slicing has been introduced by Weiser (Weiser 1981) as a technique to understand and debug programs. Since then, program slicing has grown as a field and a great number of researches have been performed resulting in different forms of program slicing, algorithms to compute slices, and applications to software (reverse and re-) engineering (Tip 1995; Lucia 2001). According to its initial definition, a program slice SC is an executable set of program statements that preserves the execution behaviour of the program P, with the respect to a subset of variables V of interest at a given point of program p, i.e. a slicing criterion C=<p,V>.

Although in other program slicing techniques, this definition is sometimes relaxed to allow a non-executable set of statements that directly or indirectly affects a subset of values of interest. This concept has been borrowed by many business rules extraction methods, because it enables to discover thorough and rigorous view of business logic implementation.

Chiang (Chiang, Bayrak 2006; Chiang 2006) proposes program slicing based method for business rules extraction in order to afford possibility for business rules reuse by migrating them from the legacy code to the loosely coupled components (i.e. web services). The paper considers several kinds of business rule: primitive and complex. A primitive business rule is a function that from given set of input parameters produces one parameter, and a complex rule is defined as a set of primitive function. A function represents a program slice computed using static backward slicing (Weiser 1981). Chiang and Bayrak (Chiang, Bayrak 2006) noticed that a number of slices become extremely large in the case of large software system. For this reason, program code is separated into the three categories: user interface; business logic; and data access. The starting point for slicing user interface is considered as a statement reading or

displaying the data. Data I/O statements such as read, write, rewrite, open, or close are considered as candidates for the starting point of slicing data access layer. During the analysis of business logic category, code parts that affect data variables are separated and presented for software developers for validation. Unfortunately, the method is illustrated with relatively straightforward example where it is applied for a snippet of COBOL code. From this example it is hard to evaluate the feasibility or effectiveness of the method. Moreover, the presented example does not reflect the business rule at all; rather it shows a certain kind of interface to identified data structure.

Huang (Huang 1996) proposed business rules extraction using several types of program slicing. The method proposes a number of heuristic rules for domain variables identification, slicing criteria identification, and slicing algorithm selection. Domain variables are identified considering every input and output variables of the system, arguments and return parameters of procedures. Slicing criteria involves input and output statements of the program, dispatch centres and return statements of procedures. A slicing algorithm is selected according formulated slicing criteria: for input variables and dispatch centres forward slicing is used; for output variables backward slicing is used to extract the relevant computation logic. Extracted business rules are represented either using a code-view, a formula-view (three parts formulae – left hand side for a variable, right hand side for an expression that modifies variable, and conditions under which modifications may be executed), or input-output dependence view (bidirectional data flows between input and output parameters). The proposed approach has been implemented in the prototype tool called Business Rules Extraction Environment targeting at extraction of business rules from COBOL programs. A small example of application of the tool for a particular scenario has been presented within the paper. Though the approach gives some advices on formulating heuristics for business rules extraction from legacy code, it lacks on clear definition of what kind of business rules it attempts to discover. Rather, the approach equalizes business rules with the program slices that reflect all possible paths of certain variables computation at given point of a program. One would notice that a program slice may represent a particular use case scenario and contain numerous nested rules (i.e. control statements) within it, or the slice may even be meaningless in case of computation of non-domain variable (i.e. not a business term).

The latter issue is considered in extended version of this approach presented by Wang *et al*. (Wang *et al*. 2004a; Wang *et al*. 2004b; Wang *et al*. 2009). In this approach, classification techniques are employed during domain variables identification step. The approach also emphasizes the importance of identification of synonymous variables occurring in different modules. Having extracted domain variables and their dependences, the next step, called data analysis, iden-

tifies business items that are actually implemented in the selected slice, obtained by calculating information-flow functions (Bergeretti, Carre 1985). According to the obtained information, a set of business rules is extracted and represented using multiple views in order to be validated with stakeholders. However, proposed views representing business rules require deep understanding of technological aspects of the software (i.e. to understand the code or graphs that represents dependencies between code blocks); therefore they are hardly understood by business analysts and stakeholders. An improvement of Wang *et al*. work is proposed by Gang (Gang 2009). The approach constructs a program dependence graph and after identification of data dependences, it augments the graph with edges that represent dependencies among program statements. Lastly, the backward traverse is applied to the dependence graph and a resulting dependence-cache slice is a collection of all reachable nodes by this traverse. Resulting slices are presented for validation with stakeholders as code fragments. However, code views requires deep understanding of technological aspects of the software system, therefore they are difficult to be validated with stakeholders.

The program slicing as a technique for collecting of relevant information about identified business concepts is also suggested by Paradauskas and Laurikaitis (Paradauskas, Laurikaitis 2006; Paradauskas, Laurikaitis 2011). The method uses both forward and backward slicing in regards with a subset of variables that are used in program input or output statements, such that forward slicing is used to collect all statements that are dependent on a given input statement (e.g. read user provided data), and backward slicing is used to gather statements that contribute to the variables of output statement (e.g. print data to the user). Sneed (Sneed, Erdos 1996; Sneed 2001) also suggests use program slicing for rules extraction from legacy systems. However, neither of method above does provide any further information on how to store, refine and represent the slices that correspond to business rules.

1.5.1.2. Extracting Business Rules using Pattern Matching

Apart from program slicing, Paradauskas and Laurikaitis (Paradauskas, Laurikaitis 2006; Paradauskas, Laurikaitis 2011) incorporate pattern matching for business knowledge extraction from legacy information systems. The method involves schema extraction and semantic analysis techniques to discover conceptual schema from legacy source code and relational database. In addition to explicitly defined relations between tables in the database (i.e. reference constrains), the approach extracts implicit relations by analysing the legacy source code (written in C language) and SQL queries embedded within it. A number of query patterns are proposed to identify candidate keys that would allow establishing relations between tables. Inclusion dependency mining (that is comparison of populations of attributes participating in extracted relations) is used to

classify extracted relations into the IS-A, dependent, aggregate, and other kinds of relations. Extracted schema elements are further converted from the intermediate representation defined in XML into the entity-relationship diagram notation. According to the Business Rules Approach (Hay 2000; Hay 2002), all the extracted information is valuable because it reflects business vocabulary (business terms and facts).

However, although the approach claims ability to extract business rules, it seems that it is being capable to extract only static business rules (i.e. cardinality constrains). Extraction of other kinds of business rules is neither investigated, nor discussed in the presented research. Moreover, the presented examples do not convey the feasibility of the approach in real-world scenarios, especially where the large data set are present and inclusion dependency mining is costly activity.

Chapparro (Chaparro *et al*. 2012) proposes patterns matching based method for extraction of structural business rules from legacy databases. This method a number of heuristic rules that assign certain construct of database schema (columns, table, constraint, dependencies) to a certain kind of business fact type (unary, binary, property, and category) or to the structural business rule. Then, by applying pattern matching technique, extracts corresponding constructs and stores in the relational database. The method has been evaluated in the case study of legacy system implemented using Oracle Forms technology. The results of case study showed only one third of extracted business rules being correct.

In contrast, our method combines both patterns matching and program slicing to extract more rigorous details about implementation of business rules. As in Chaparro *et al*., we define a set of patterns for extraction of different kinds of business facts from available resources of software system, including database schema, source code, and resource definitions. To refine extracted knowledge and help to identify possible candidates to business terms and facts, the method applies text comparison algorithms on indexed documentation thus retrieving a set of relevant matches for extracted terms. For the internal representation of extracted knowledge our method uses the Knowledge Discovery Meta-model (KDM). The system dependence graph is created within a set of KDM models and further is used to slice business scenarios (i.e. Use Cases) – a behavioural logic that handles various kinds of software events. By applying a set of heuristic rules, the method identifies behavioural business rules within the extracted scenarios. The method produces intermediate representation of business knowledge that facilitates creating higher-level of abstraction knowledge representation; however, transformations to more abstracted forms of knowledge representation (such as SBVR, decision tables and trees) are not considered in the current research.

Putrycz and Kark (Putrycz, Kark 2007; Putrycz, Kark 2008) uses patterns to identify code snippets that corresponds to production rules. The method considers extraction of production business rules in the form <Condition><Action> by analysing abstract syntax tree (AST) of legacy COBOL application. In order to separate code implementing business logic from setup and data transfer implementations, the approach focus on single statements that embody calculations or branching since they most often represent high level processing. The knowledge extraction process consists of the two following steps: construction of extracted knowledge base and linking the knowledge base items with existing documentation. The knowledge base is created from the production rules, identifiers, dependencies between rules, exception statements. The elements of knowledge base are linked with documents by performing key phrase analysis (in particular by employing Kea algorithm). Although the approach overcomes previously discussed approaches with the ability to refine business relevant knowledge and present it in form acceptable for a business analyst, it lacks on clarifying how the approach helped to extract business rules. The feasibility of the method has been evaluated in a case study; however, from the presented results of the case study it is hard to find out how many of business rules were identified, and how much of them were accepted as relevant after performing key-phrase extraction. It should be noted that the same issue of linking documents with extracted knowledge is being addressed by Antoniol *et al*. (Antoniol *et al*. 2000**;** Antoniol *et al*. 2002). However, these studies do not concern business rules extraction in general.

Earls *et al*. (Earls *et al*. 2002) proposes a method for manual extraction of business rules from source code. Assuming that error handling code parts reflects violations of implemented business rules, the method proposes to identify these parts and scan backwards to collect all related code parts that lead to that exception. The method consists of the following steps. First, the source code is placed into text editor and prepared for analysis by removing irrelevant code parts: comments, working storage declarations, database connectivity management, reporting heading processing, and log-displaying code. Then, each call to procedure is replaced by the procedure body. After, error-processing sections are located and classified and code parts that invoke these sections are recorded. Finally, the conditions that led to the invocation of the error-processing code are translated into business rules and stored in the rule repository for evaluation with domain experts.

The proposed method was applied in modernization project of large legacy system resulting in numerous design and business rules extracted from the source code. The design rules revealed certain decisions on the system implementation that have been taken into the consideration during migration to targeting platform. As the authors observe, not every business rule implemented in the

system was extracted; however, proposed method allowed modernization engineer to discover business knowledge in faster and more accurate way. Notwithstanding the fact that error handling code sections often reflect violation of certain rules and therefore are reasonable candidates for business rules extraction, the proposed method omits consideration of other kind of business rules, such as structural rules, derivations, or other types of behavioural rules. Moreover, manual extraction of business rules from software systems that contains multiple heterogeneous artefacts and interacts with external systems would result in extremely labour-intensive activities that would hardly produce any valuable result. Finally, simplification of source code in the proposed manner (i.e. removing irrelevant code parts) disallows the method be used in software maintenance tasks, when changes are quite common.

1.5.2. Methods for Business Processes Extraction

1.5.2.1 Extracting Business Processes by Slicing Use Cases

Hung and Zou (Hung, Zou 2007; Zou *et al*. 2004; Zou, Hung 2006) present an approach for recovering workflows from multi-tiered enterprise software systems. The approach uses static tracing to identify all possible execution paths that cover user interface, application logic, and database tiers. As an entry point to start tracing, a procedure that handles certain UI event is selected. Following the flow of control and by identifying code patterns that corresponds to the elements of workflow (e.g. fork, join, task, sequence) the code is transformed into intermediate workflow representation (custom meta-model). In order to refine workflows into more abstracted versions, the approach considers a number of heuristic rules. This resulted in a set of high level workflows are further transformed to models supported by IBM WebSphere Business Modeler (WBM).

To demonstrate the effectiveness of the approach, a case study has been performed on existing Enterprise Resource Planning (ERP) system, developed on Apache OFBiz platform. A prototype tool, capable of code preparation, analysis, and transformation has been applied in case study, resulting in high precise and recall factors, ranging from 75% to 100% (meaning that almost all of the extracted workflows are accurate). The results were manually evaluated in order to identify the number of misidentified and missed tasks according to documentation, comments and naming conventions. The evaluation has shown that the main categories of irrelevant tasks include utility functions, including logging and error handling.

Although the results are satisfactory, the approach has some drawbacks from our point of view. First, it considers only user input as main entry points for control flow tracing; though many researches emphasize the importance of analysis of different entry points to the system, such as external calls or internal

periodical events. Second, the paper considers only subset of intermediate representation (i.e. represented are only code and UI, though aiming at gathering information from other artefacts). Finally, it is neither proposed, nor discussed on how to retain traceability from extracted knowledge to its actual implementation.

1.5.2.2. Extracting Business Processes Using Pattern Matching

Perez-Castillo *et al*. (Perez-Castillo *et al*. 2011a; Perez-Castillo *et al*. 2011b; Perez-Castillo *et al*. 2011c; Perez-Castillo *et al*. 2012; Perez-Castillo *et al*. 2009) propose a method for business processes extraction from legacy information systems. The method is based on a framework called MARBLE that is aligned with Architecture-Driven Modernization (ADM) and which uses the Knowledge Discovery Meta-model (KDM) standard for representation and manipulation of the knowledge about the legacy information system.

The MARBLE framework spans the following levels of abstraction: L0 – implementation of a legacy information system; L1 – direct representation of software system artefacts using language specific meta-models (e.g. an AST of java code); L2 – various models represented within the KDM; L3 – representation of extracted business process models using Business Process Modelling Notation (BPMN). Representations within each of the level are derived by several kinds of model transformations. The most important transformation from the representation within KDM to BPMN is based on a set of patterns. The patterns varies from straightforward (e.g. sequencing, starting and termination) to more complex (e.g. branching, conditional sequencing, exceptions, and collaboration). The method formally defines corresponding KDM model structures and uses these definitions to create model-based (i.e. Query-View-Transform language, QVT) transformation rules.

The feasibility and effectiveness of the proposed method have been evaluated in the case study of a medium-size real-world legacy system implemented in Java (28 KLOC). During the case study a number of business process models have been identified by discovering interrelated business tasks that corresponds to the predefined patterns. With a help of business experts, discovered tasks were reviewed and refined. As a result, almost half of overall tasks were recognized as relevant (223 from 425) and 10 percent (41 from 425) as unidentified relevant tasks. The effectiveness evaluation has shown reasonable transformation time (i.e. linear with respect to the size of the models) showing the scalability of the proposed method.

This method shows that knowledge discovery problem may be reduced by incorporating model-based software comprehension: the business knowledge extraction is separated from obtaining the intermediate representation (i.e. “as-is” models) of the source code. The method becomes (relatively) independent

from the software implementation. Although the method is based on the KDM standard, it seems that the method does not utilize all features provided by this standard. Currently, the method considered the transformation from KDM Code model to BPMN. However, the KDM itself has different abstraction layers enabling model refinement and abstraction within the same facility. The KDM Conceptual model contains elements intended for representation of behaviour and scenario units that are very closely related to the concepts of task and business process investigated within this research. In regard with the purpose of the KDM, the problem of knowledge discovery could be further reduced to KDM Code to KDM Conceptual model transformation and KDM Conceptual to BPMN transformation, allowing reuse of extracted knowledge (i.e. business concepts and tasks) in other kinds of representation, such as business vocabulary and business rules. Unfortunately, this issue has not been considered within the research.

The model driven framework for reverse engineering process consisting of model for both low and high level processes and appropriate set of transformations is being presented by Kamran (Sartipi 2003; Sartipi, Yousefi 2010). The proposed framework is based on extraction of task scenarios that provide a common goal for the whole process. We use similar approach as we also concentrate on use-cases identification; however, our main focus is on analysis of user-interface and platform events, whereas in their approach, these models are supplied from the business domain.

Kalsing *et al*. (Kalsing *et al*. 2010) together with Nascimento *et al*. (Nascimento *et al*. 2009; Nascimento *et al*. 2012) propose method that extracts business processes by identifying business rules within the legacy code. For this reason they use pattern matching technique. Patterns are formulated according to Weiden *et al*. (Weiden *et al*. 2002) classification scheme: mathematical calculations, function/procedure calls, data persistence, user interaction, pre-processing, post-processing, and control flow. Using code transformations, the legacy code is augmented with comments denoting corresponding rule class and with invocation of logging function enabling traceability of source code execution.

Having modified and recompiled the legacy code, the approach identifies use case scenarios performed by legacy software users and executes these scenarios to log the identified rules execution. Then, by using heuristics based IncrementalMiner algorithm, it extracts dependency graphs from log information. Extracted graphs are transformed to business process models (BPM) to facilitate further modernization of legacy system. In order to evaluate the proposed approach, the research has performed a case study on the Financial Module of legacy Enterprise Resource Planning (ERP) system written in COBOL language. As a result, the case study has extracted the structure of 7 business processes together with more than 50 business rules implemented within this module,

showing that incremental process mining approach allows extraction of partial results (analysis of certain module) and thus reduces the total processing time.

However, from our point of view, this approach has several important drawbacks. First of all, this paper does not discuss on how to separate the code parts implementing business logic from the code parts intended to support infrastructure related activities. It is clear that not every mathematical calculation implements computation of business value (e.g. computing the size of user interface window), or not every control flow statement depends on evaluation of business value (e.g. condition evaluating whether certain object has been instantiated). Moreover, the paper does not consider how to identify business rules that covers nested conditional statements or even multiple procedures or modules. Finally, injection of logging instructions into the code of legacy system may not to be acceptable if it is in usage. Modifying the code of cloned legacy software, on the other hand, may not reflect the real-world scenarios because of deployment specific characteristics (platform configuration, interaction with external libraries or other software systems owned by organization), or absence of working data.

Table 1.4. Summary of Methods for Business Knowledge Extraction

Study	Knowledge Form	Analysis Techniques	Software Artefacts	Representation Form	Automation Level and Tool Support	Evaluation
Huang	Domain variables, business rules	Program slicing	Source code	Intermediate: CFG; Output:Code view, formula view, dependence view	Semi-automatic using prototype tool	Example
Sneed and Erdos	Business rules	Pattern matching	Source code	Intermediate: N/A; Output:Source code snippets	Semi-automatic using prototype tool	Example
Sneed	Business rules	Pattern matching, program slicing	Source code	Intermediate: CFG; Output:Source code snippets	Semi-automatic using prototype tool	Research case study
Earls, Embury, and Turner	Business rules	Manual backward analysis	Source code	Intermediate: Source code; Output:Source code snippets	Manual extraction using text processors	Industrial case study
Fu *et al.*	Business rules	Transformations	Source code	Intermediate: BRL; Output:N/A	N/A	Example

Table 1.4. (Continued)

Study	Knowledge Form	Analysis Techniques	Software Artefacts	Representation Form	Automation Level and Tool Support	Evaluation
Wang *et al.*	Business rules	Domain variable classification, program slicing	Source code	Intermediate: PCG; Output:Slices	Semi-automatic using prototype tool	Research case study
Zou *et al.*	Business processes	Pattern matching, control flow analysis	Web pages, source code, configuration files, database	Intermediate: Custom; Output:Business process models	Semi-automatic using prototype tool	Reseach case study
Chiang and Bayrak	Business rules	Program slicing	Source code	Intermediate: PDG; Output:Slices, source code snippets	N/A	Example
Paradauskas and Laurikaitis	Business concepts, business rules	Pattern matching, program slicing, inclusion dependency mining	Source code, database schema, data	Intermediate: PDG, Custom; Output:Entity-relationship diagrams	Semi-automatic using prototype tool	Example
Putrycz and Anatol	Business rules	Pattern matching	Source code, documentation	Intermediate: N/A; Output:Business rule templates	Semi-automatic using prototype tool	Industrial case study
Normantas and Vasilecas	Business vocabulary, business rules, use-cases (scenarios)	Patterns matching, program slicing, model-based transformations	Source code, resourse definitions, configuration, documentation	AST, PDG, KDM	KDM	Research case study
Cai, Yang, and Wang	Business processes	Interviews, static analysis, dynamic analysis	Source code	Intermediate: PDG; Output:Business process models	Semi-automatic	Research case study

Table 1.4. (Continued)

Study	Knowledge Form	Analysis Techniques	Software Artefacts	Representation Form	Automation Level and Tool Support	Evaluation
Gang	Business rules	Program slicing	Source code	Intermediate: PDG; Output:Slices	Semi-automatic using prototype tool	Example
Nascimento *et al*.	Business rules, business processes	Patterns matching, rewriting	Source code	Intermediate: N/A; Output:Business process models	Manual	N/A
Perez-Castillo *et al*.	Business processes	Patterns matching, program slicing, model-based transformations	Source code	Intermediate: AST, KDM; Output:BPMN	Semi-automatic using MARBLE 2.0 framework (tool suite)	Industrial case study
Chaparro *et al*.	Business vocabulary, business rules	Heuristics based classification, transformations	Database schema, source code	Intermediate: Custom; Output:N/A	Semi-automatic using prototype tool	Research case study

1.6. Discussion

In this review we asked "what kinds of business knowledge are being extracted by the research?". The most common business knowledge kinds that are being considered in extraction methods are business rules and business processes. Several studies show interest and provide guidance in extracting the business vocabulary as well. Although there are various business vocabulary and rule categorization schemes proposed in the business rules research field, only several methods considers using certain scheme to extract and classify the business knowledge. Nonetheless, the result of this study shows that the interest in business knowledge extraction is important and relevant topic.

The next question was "what analysis techniques are used as a basis for knowledge extraction?". This study showed that the methods for business rules extraction have several common characteristics. They are similar in the sense that most of them concern the following general procedure for knowledge extraction: gathering initial information, extracting candidate rules, refining candidates, and transforming to the output representation form. The most common

techniques for business rules extraction are program slicing, pattern matching, and transformations. These techniques are used to extract business processes as well. It is important to notice that the most of reviewed studies propose using combination of techniques in order to achieve more accurate results.

It is also should be noted, that in this review we did not include the studies that concerns usage of information retrieval or mining as primary methods, such as business processes mining from event logs or rules extraction from data or documentation. In our opinion, business knowledge extraction from software artefacts may be treated as a particular research field, therefore above mentioned topics require separate studies.

The third question asked "what software artefacts are used as input sources for knowledge extraction?". There are very few methods that consider knowledge extraction from software artefacts other than source code, for instance, resource (web pages, forms, reports) definitions or configurations. Though, within contemporary software systems, these artefacts may contain much valuable information, including business vocabulary and rules. This could be explained by the fact that most of business rules extraction methods consider legacy software written COBOL. It is important to notice, that neither of reviewed study propose a method to extract business processes from such software.

The fourth question asked "how extracted knowledge is being represented?". There are typically two types of knowledge representation: intermediate and output. Methods that use program slicing inevitably rely on the intermediate representation intended for traversing the various kinds of intermediate code representation, such as control flow graphs, call graphs, or program dependence graphs. Methods that uses patterns matching either perform it directly in code or propose custom intermediate representation (custom meta-models). Only several methods use standard based (i.e. Knowledge Discovery Meta-model) intermediate representation of the extracting knowledge.

We were surprised that very few methods consider widely accepted forms for business rules representation (such as templates, decision tables, etc.). Most of the reviewed studies suggest using either code snippets, or graph slices as the output representation, though such representation form is not acceptable solution when extracted rules must be evaluated with business analysts or stakeholders. In our opinion, this issue should be investigated in the further research.

However, this issue is irrelevant for the output representation of extracted business processes. The most of reviewed studies consider representing business processes using models defined either with Business Process Modelling Notation (BPMN), or other notation supported by business process management tools.

Considering the quality of the research, the most of reviewed papers provides informal or semi-formal definitions of knowledge extractions algorithms that we believe in certain cases could be reproduced. However, as it was already mentioned, it would be unreasonable to expect that all details about the implementation of the approach or its evaluation in a case study could be clearly and particularly specified, especially when there are different limitations, such as the length of paper in journals or conference proceedings. It is important to notice, that the most of studies propose tool support for the extraction method; however, only very few presents industrial case study on a very large software systems.

1.7. Threats to Validity

There are three main threats to validity of a Systematic Literature Research (SLR) (Kitchenham, Charters 2007): study selection bias, internal validity (design and execution of the study) and external validity (applicability of the effects observed in the study). As this review has been conducted by two researches, there is potential chance that this research has not obtained a complete coverage of the studies in the research field. This could be due to the selection of primary data sources (digital libraries/search engines) or due to absence of analysis of secondary resources. To ensure the selection of unbiased studies, we formulated research questions, defined the search string from these questions, defined inclusions/exclusion criteria, and conducted multi-phase literature review.

The review process was designed by single researcher. This could lead to misinterpretation of the main concepts during development of the research questions, defining search string, or selecting inclusion/exclusion criteria. Therefore, the design of the review process was evaluated with other researcher (scientific supervisor) and with the experts in this field in order to prevent systematic errors in design and execution of the study.

In this review we selected publications that are from academic domain, in particular from peer-reviewed scientific publications. It is likely that relevant methods are applied in industry, but not reported in scientific publications. As we had no possibilities to access such resources, these methods were not included in the review.

1.8. Conclusions of the First Chapter

In this chapter we presented the Systematic Literature Review (SLR) of the selected studies on business knowledge extraction from existing software systems. The review has been conducted following the guidelines for the SLR proposed

by Kitchenham *et al*. The review covered 24 studies that were published in peer-reviewed journals, conference proceeding and collections, available at seven digital libraries/search engines.

The results of this review allow us to draw the following conclusions:

1. There is insufficient attention paid to extraction of business vocabulary and no concern is made for widely used business rules classification schemes, therefore limiting the possibility to represent business knowledge with the most common techniques.
2. The proposed methods rarely concern knowledge extraction as a complex problem which involves more than program code analysis, thus aggravating their application in knowledge extraction from software systems developed using multiple technologies.
3. The most common techniques for knowledge extraction are program slicing, patterns matching and transformations, though certain methods use the combination of them.
4. A minority of proposed methods were evaluated in industrial case studies on large enterprise software systems, considering various software artefacts and other available knowledge sources.
5. The Knowledge Discovery Meta-model (KDM), as an intermediate representation of knowledge about existing software systems, is not studied enough and there is a lack of algorithms for creating representations within the KDM and for performing analysis on the models.

2

Knowledge Representation and Analysis

2.1. Introduction

The study of related literature showed that although there is a promising Architecture Driven Modernization (ADM) initiative formed by the Object Management Group (OMG) that has already adopted standard specification for representing knowledge about existing software systems, the Knowledge Discovery Meta-model (KDM), there is a lack of methods that rely on this standard and propose different approaches for knowledge extraction, representation, and analysis, thus aggravating invention of different tools that would support software modernization process.

By addressing this issue, in this chapter we formulate a formal background for our method which uses the KDM as an intermediate representation (IR) of knowledge about existing software systems and static program analysis for business knowledge extraction. We first of all provide a brief overview of this standard by identifying its capabilities to represent the multi-tiered software system architecture, techniques that could be used to obtain the representation, and possibilities to extend the KDM for representing software platform or domain specific knowledge. Then we define the principles for representing source code

using the KDM Code Model by introducing simple imperative programming language. In addition, we introduce a simplified version of structured query language (SQL) and define the representation principles with the KDM Data Model. Finally, we revise dataflow analysis techniques and formally define their equation systems in the context of the KDM representation as a set of the KDM extensions. Certain parts of this chapter were presented in (Normantas, Vasilecas 2011; Normantas, Sosunovas, Vasilecas 2012c).

It should be noted, that the formal definitions provided within this section are constructed using the Object Constraint Language (OCL) rather than traditional set notation used in data flow analysis. The main reasons for that are as follows: a) the main subject of this study is model-based analysis; b) KDM meta-model corresponds to the Meta-Object Facility (MOF) allowing application of OCL constraints and queries; c) definitions within meta-model may be verified and validated on models corresponding to it.

2.2. An Overview of the Knowledge Discovery Meta-model

KDM provides an intermediate representation of knowledge about existing software systems. The representation consists of several architectural viewpoints that are defined at different abstraction layers. Each viewpoint is conveyed over a set of architectural views – KDM models representing different perspectives of knowledge about the artefacts of existing software system. These models may be created automatically, semi-automatically, or manually by applying various knowledge extraction, analysis, and transformation techniques. In addition to standard tree view, every model can be visualized using appropriate type of diagram, table, or graph, to reduce the time required to comprehend particular aspect of the system.

From the meta-model perspective, the KDM is an entity-relationship representation having two fundamental elements: *KDMEntity* and *KDMRelationship* (Fig. 2.1). A KDM entity is an abstraction of some element of an existing software system that has a distinct, separate existence, a self-contained piece of data that can be referenced as a unit (OMG 2011a). In addition to the *KDMRelationship*, there are two specific relationship types: containers and groups. KDM entities may be containers for other entities by representing them with the ownership (containment) relationship between the container and entities that are directly owned by this container. KDM entities may also be groups of other KDM entities. There is a special group association (grouping) relationship between the group and the entities that are directly "grouped into" this group.

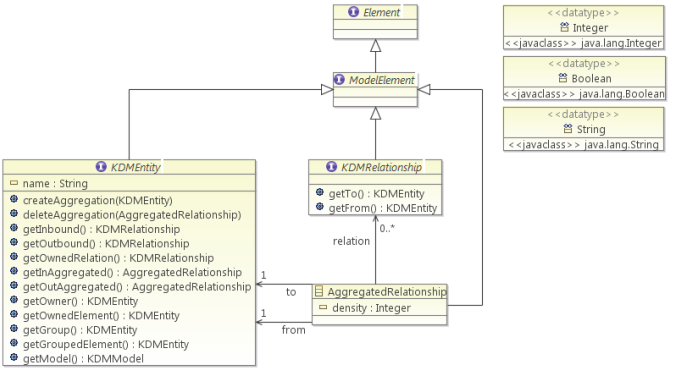

Fig. 2.1. KDM Core elements (OMG 2011a)

In order to better understand this standard, consider as an example the multi-tier software architecture presented in the left side of the Fig. 2.2. It involves a number of levels one upon the other, each consisting of heterogeneous components that serve distinct and separate tasks.

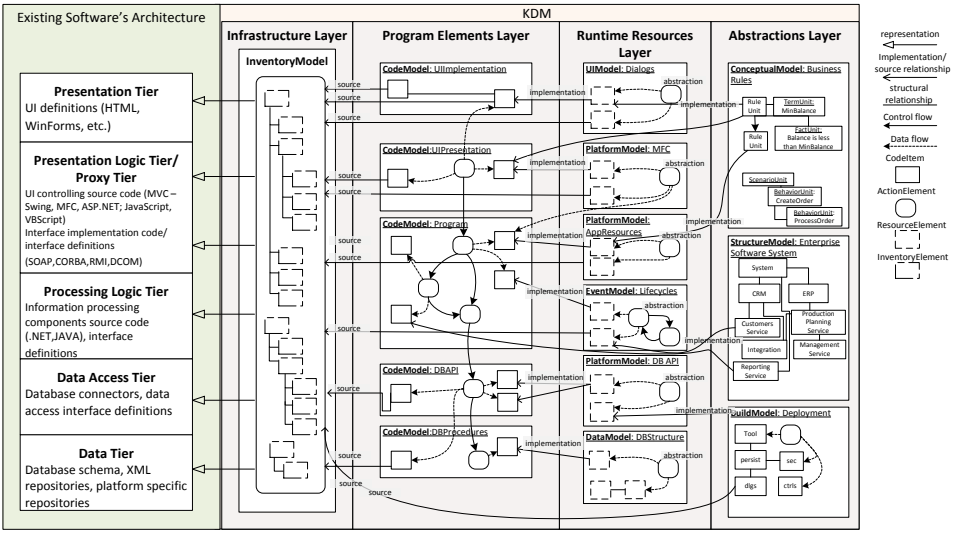

Fig. 2.2. The projection of multi-tier enterprise software architecture to the representation with KDM

The Data tier defines particular kinds of data stores: hierarchical, relational or object-oriented databases; platform specific repositories; or collections of data

files, structured in some text processing format (e.g. XML, CSV, DAT). The interfaces to database, such as database connectivity components (e.g. ODBC, JDBC) or specific file content managers, are coped within the Data Access tier. The Processing Logic tier consists of components that implement the business logic. The Presentation Logic tier consists of components that interact with UI components (i.e. implements server-client or model-view-controller pattern). In contrast to the Presentation Logic tier, the Proxy tier defines the interaction with external systems over some instances of high level communication protocols, for example SOAP, DCOM, CORBA, JNI, or RMI. Finally, the Presentation tier is responsible for producing and controlling user interface components, such as forms, web pages, or reports.

The KDM representation of the software system starts by the Inventory model that represents software artefacts and keeps traceability to the original source. A snippet of source content (e.g. code line or configuration section) is represented using source reference element that may be assigned to almost any type of KDM element. Program elements layer contains multiple Code models representing the structure (association, composition, aggregation, and inheritance relationships) and behaviour (data and control flow) of the source code.

The Data, Platform, Event, and UI models represent the structure and behaviour of the run-time resources of the system. These models are directly obtained from the resource definitions, abstracted from the Code models, or manually created by a system analyst. Traceability from abstracted element to the element it is abstracted from is kept by the property named implementation. The abstraction property of resource element adds behaviour parts that represent logic of the resource operation, including the flow of data and control.

The Conceptual, Structure, and Build models represent highest level abstractions obtained from the models of lower level of abstractions, the data processed by the system, and different kinds of software's documentation. Our findings reveal that, though documentation of software often is obsolete, the definitions of domain concept included in it typically do not change in the course of time. However, the representation of latter source of knowledge currently is not supported by the KDM; therefore, light-weight extension mechanism provided by the KDM could be employed in order to represent it.

2.2.1. Techniques for Obtaining Software System Representation using the Knowledge Discovery Meta-model

The following table summarizes our findings: techniques used to obtain different types of KDM models from the input sources and kinds of visualization that may be produced by applying transformation from the KDM to a particular modelling language (i.e. UML), tabular or textual notation.

Table 2.1. Overview of sources and techniques used to derive KDM models from various software artefacts

KDM Layer	KDM Model	Input Sources	Techniques	Model visualizations
Infrastructure Layer -defines the set of core KDM elements, the KDM framework concepts, and the elements used to represent the environment of the software system.	**Inventory Model** -represents physical artefacts of an existing software system: their types (e.g. executable, binary, configuration, etc.); their organization within software environments and/or distribution over organizational network (e.g. projects, directories); relationships between them (e.g. dependency).	Software application files repository	File systems or repositories traversal	Annotated tree views of inventory model
Program Elements Layer -defines the set of elements for representing the constructs supported by common programming languages.	**Code Model** - describes structural and behavioural aspects of the source code – the composition of code modules or data types (e.g. modules/classes, functions/operations, variables/properties, etc.) and the data and control flows (e.g. statements and expressions).	Source code API definitions Reflected or decompiled binaries and executable Database schemas Resource definitions	Parsing to AST and transformation Patterns matching	Abstract syntax tree views Class diagrams Control flow graphs System dependence graphs Activity diagrams Sequence diagrams
Run-Time Resources Layer -defines the set of elements for representing particular aspect of software resources and defines how	**Platform Model** -represents resources used by the software system, their composition and behaviour, including control flow initiated by the platform.	Inventory model Code model Resource definitions Configuration files	Parsing to AST and transformation Patterns matching Concept lattice analysis Dependency analysis Clustering	Component diagrams Class diagrams System resources dependence graphs

Table 2.1. (Continued)

KDM Layer	KDM Model	Input Sources	Techniques	Model visualizations
these resources are related with each other.	**Data Model** -represents the structure of persistent data elements of the software system (e.g. table, view, column), the information model supported by the system (e.g. object, property), the data flow involving persistent data, and the control flow initiated by the events that are triggered due to modification of the data.	Database schemas Resource definitions	Parsing to AST and transforming Patterns matching	Data structure tree views Class diagrams
	UI Model -represents the composition of UI facets and controls, their layout, the control flow initiated by them, and the data flow originated from or terminated at them.	Resource definitions Inventory model Code Model	Parsing and transformation Analysis and manual creation	Tree views of UI models Dependency graphs
	Event Model -represents the behaviour aspect of software system resources over event-driven state transitions model, which includes particular actions performed in a given state.	Resource definitions Configuration files Code model Data model Platform model UI Model	Patterns matching Transformation Analysis and manual creation	State-transition tables State machine diagrams Actor-event mapping tables
Abstractions Layer -defines the set of elements for representing software, information, or business level abstractions	**Structure Model** -represents the composition of architecture components of the software system and relationships between them, including aggregation to subsystems and grouping to architecture views or layers.	Inventory model Code model Data model Platform model UI Model	Concept lattice analysis Dependency analysis Clustering	Tree views of structure models Component diagrams Class diagrams

Table 2.1. (Continued)

KDM Layer	KDM Model	Input Sources	Techniques	Model visualizations
	Conceptual Model -represents behaviour and scenario flows, business terms, facts and rules implemented by the system.	Inventory model Code model Data model Platform model UI Model Configuration data Database data Documentation	Patterns matching Model transformation Dependence graph slicing Manual definition	Class diagrams Activity diagrams Sequence diagrams Use Case diagrams SBVR Templates BPMN Diagrams Decision trees Decision tables
	Build Model -represents the facts about the software build process: input/output; tools used to build source code; build workflows.	Inventory model	Inventory items and their property analysis Manual definition	Build graphs Build reports

Apart from the straightforward techniques that incorporate parsing the content of software artefacts and direct transformation to elements of KDM in order to produce "as-is" representation of the software, the clustering-based, patterns-based, and model slicing-based techniques may be employed to derive higher level abstractions in automatic or semiautomatic way.

Clustering-based techniques allow identifying architectural components by analysing relationships between software elements. Proximity metrics calculation techniques may be applied to elements of KDM instance (Lakhotia, Gravley 1995; Tzerpos 1998; Wiggerts 1997) to group into the cohesive components; formal concepts analysis may be employed to aggregate the groups of maximally related model elements, arranged in the neighbouring nodes of a concept lattice (Arevalo *et al*. 2005); hierarchical clustering would arrange software components in dependence graphs (Maqbool, Babri 2007).

Pattern-based techniques use approximate matching of patterns in models by evaluating the probability that particular part of model may correspond to the abstract or concrete pattern (Flores, Aguiar 2005; Sartipi 2003). For this reason similarity measures of matching parts of model must be evaluated in order to establish architectural components of the system. There are also numerous techniques for entity-relationship extraction that are useful for business vocabulary discovery (Andersson 1994; Ambler, Sadalage 2006) as it will be discussed in the next chapter.

Since the KDM supports representation of the control and data flow of the system, the system dependence graph may be obtained using data flow computations (Khedker *et al*. 2009) and sliced according to a set of domain elements using forward or backward slicing methods (Tip 1995). The former kind of methods would produce scenario flows (e.g. workflows) of the system, while the latter would allow identifying computations of particular domain elements (e.g. facts and business rules).

2.2.2. Extending the Knowledge Discovery Meta-model

Though the KDM provides a large number of meta-model elements to represent the software system from different perspectives, with the desired level of granularity and at different abstraction layers, it is not always sufficient for representing specific aspects of software system or its domain.

For this reason, the KDM introduces light-weight extension mechanism as a standard way of adding new elements to KDM. The light-weight extension mechanism provides capability to: define stereotypes that may extend either concrete or abstract meta-model elements; define tags associated with stereotypes and add values (in a form of string or reference to some modelling element) to them; group stereotypes into stereotype families; use one or more stereotypes within extended meta-model element.

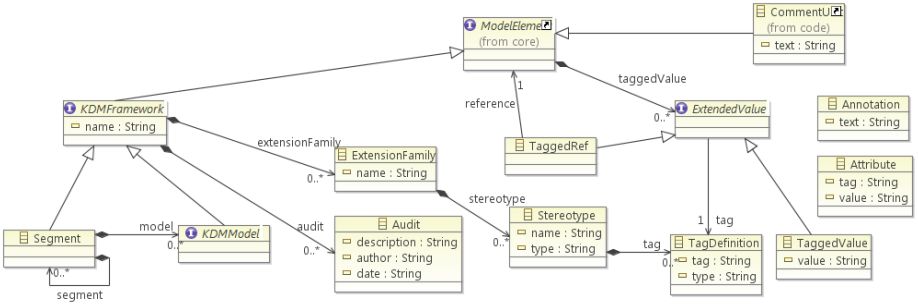

Fig. 2.3. KDM Extension mechanism (OMG 2011a)

Each KDM extension is defined over extension family, composed of stereotypes with aggregated tag definitions. When these stereotypes are applied on model elements, the tag definitions may be instantiated over tagged values. Tagged value may be either a string or a reference to another model element which type should correspond to the type defined by the tag definition. Moreover, besides the capability to define audit information, the KDM allows semanti-

cally enriching model elements with domain or platform specific attributes and annotations.

However, the as it could be observed from the Fig. 2.3, the KDM extension mechanism does not support multiplicity of tags, constraints on tags, relationships between tags or stereotypes (e.g. association or inheritance), thus limiting the expressiveness of the representation and aggravating the ability to define extension families in rigorous and unambiguous way.

2.3. Representing Source Code

In order to be able to apply dataflow analysis on the intermediate representation of existing software systems, in the following we introduce a simple imperative language that captures the most common programming language constructs, such as variable definitions, expressions and control statements. Then we identify a mapping from the language constructs to the KDM Code model elements and, by following the semantics of the KDM, we formally define well-formedness constraints of the KDM representation. We further identify and show the principles for representing the data and control flows, including the call-return representation mechanism.

2.3.1. Example Language

In this section we define a set of grammar rules of our example programming language. To define grammar, we use the syntax supported by the parser generator tool *XText* used to develop parsers for our method supporting tool framework. This syntax is similar to Extended Backus-Naur Form (EBNF) in the sense of rule definition patterns, although there are some differences in notation. For the precise specification, please refer to the *XText* grammar language documentation (XText 2013).

In our language, the source code is composed of program modules. A module consists of members which are either global variables or procedures.

```
Module: name=ID '{' member+=Member '}' ;
Member: GlobalVarDef | Procedure;
```

A procedure consists of a signature and a body. The signature of a procedure is defined by a name, a type, and a list of parameters that may be either passed by a value, or passed by a reference. If a way of parameters passing is not specified, then passing by value (ByVal) is treated by default.

```
Procedure: signature=Signature '{' body=Block '}';
```

```
Signature: type=Type name=ID '(' paramList=ParameterList? ')';
ParameterList: parameters+=Parameter (',' parameters+=Parameter)*;
Parameter: (pass=('ByVal'|'ByRef'))? type=Type name=ID;
```

The body of the procedure is defined by a block which contains multiple statements, which may be of the following types: assignment statement, conditional statement, loop statement, and invocation statement. It should be noted, that for the simplicity we include only basic constructs, common in imperative languages. In case of the need for additional constructs we can easily include more types of statements, such as unconditioned jumps (i.e. GoTo and Label), or more specific types of loop statements.

```
Block: statements+=Statement;
Statement: AssignmentStmt | ConditionalStmt | LoopStmt | InvocationStmt |
ReturnStmt | ContinueStmt | BreakStmt;
AssignmentStmt: ({Variable}|{Parameter}) '=' value=Expression ';';
ConditionalStmt: IfStmt | SwitchStmt;
IfStmt: 'if' '(' exp=ComparisonExp ')'
                ('{' thenBlock=Block? '}' | thenBlock=Block?)
                ('else' ('{' elseBlock=Block? '}' | elseBlock=Block?) )?;
SwitchStmt: 'switch' '(' condValue = PrimaryExp ')'
                        '{' cases += SwitchCase
                            default = DefaultCase? '}';
SwitchCase:     'case' exp+=LiteralExp (',' exp+=LiteralExp)* ':'
block=Block?;
DefaultCase: 'default' ':' block=Block?;
LoopStmt: 'while' '(' cond=ComparisonExp ')' '{' block=Block? '}';
InvocationStmt: inokeExp=InvocationExp ';';
ReturnStmt: 'return' (exp=Expression)? ';';
ContinueStmt: 'continue' ';';
BreakStmt: 'break' ';';
```

We define two types of variable in our language: global and local. A global variable is defined in the context of a module, while a local in the context of a procedure or a block within procedure. As the aim of introducing this language is to provide a background for developing code representation within the KDM and for formulating theoretical basis for control and data flow analysis, we omit explicit variable declaration statements by enabling declaration of a variable over the assignment statement (i.e. by defining certain type before a variable). When the type of variable is explicitly defined in the assignment statement, the block in which the value of variable is assigned will be treated as the context of the variable.

```
Variable: type=Type? name=ID;
GlobalVarDef: Variable '=' value=(ComparisonExp | ArithmeticalExp |
PrimaryExp) ';';
```

In order to provide possibility to define computation of variable values, we introduce several types of expressions: Comparison, Arithmetical, Invocation, and Primary. The Comparison and Arithmetical expressions are further specialized in unary and binary which is common in the design of programming languages. It should be noted, that for the simplicity and readability we leave the left recursion in definitions of expressions. Regarding an implementation of the grammar for this language, one should consider selection of left recursion allowing parser builder or redesigning it to equivalent right-recursive form. In the section of our method implementation, we explain principles of redesigning to such form.

```
Expression: BooleanExp | ComparisonExp | ArithmeticalExp | InvocationExp |
PrimaryExp;

BooleanExp: UnaryBoolExp | BinaryBoolExp;
UnaryBoolExp: op=Not rightOp=BooleanExp;
BinaryBoolExp: leftOp=Expression op=BoolOp rightOp=Expression;
BoolOp: 'and' | 'or';
Not:'not';

ComparisonExp: leftOp=Expression op=BinCompOp rightOp=Expression;
BinCompOp:'=='|'<>'|'!='|'<'|'>'|'>='|'<=';

ArithmeticalExp: UnaryArithExp | BinaryArithExp;
UnaryArithExp: op=UnArithOp rightOp=Expression;
BinaryArithExp: leftOp=Expression op=BinArithOp rightOp=Expression;
UnArithOp:'-';
BinArithOp:'-'|'+'|'*'|'/'|'%';

InvocationExp: ({Procedure}|{BuiltInFun}) args=ArgumentList?;
ArgumentList: argument+=PrimaryExp (',' argument+=PrimaryExp)*;

PrimaryExp: LiteralExp | {Variable} | '(' Expression ')';
LiteralExp: StringLiteral | NumericLiteral | BooleanLiteral;

StringLiteral: value=STRING;
NumericLiteral: value = (IntLiteral | RealLiteral);
IntLiteral: INT;
RealLiteral: INT '.' INT;
BooleanLiteral: value=('true' | 'false');
```

Finally, we introduce several built-in functions to allow interaction with a program, and define primitive types and main terminals for grammar rules of our language.

```
BuiltInFun: Read | Print;
Read: 'read' argumentList=ArgumentList;
Print: 'print' argumentList=ArgumentList;

Type:'int'|'string'|'real'|'bool'|'void';
```

```
terminal INT: ('0'..'9')+;
terminal ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal STRING:'"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') |
!('\\'|'"') )* '"';
terminal MULTI_LINE_COMMENT: '/*' -> '*/';
terminal SINGLE_LINE_COMMENT: '//' !('\n'|'\r')* ('\r'? '\n')?;
terminal WHITES_PACE: (' '|'\t'|'\r'|'\n')+;
```

Below we present an example of simple dummy program written in the language corresponding to the defined grammar.

```
Module Dummy{
        int a = 100; int b = 200;
        void Main (){
                read(x, z, w);
                if (x < a){
                        AddValue(x,a);
                } else if (x > b){
                        while(x < z){
                                switch(w) {
                                        case "Add":
                                                AddValue(x, 1);
                                        case "Sub":
                                                x = SubValue(x, 1);
                                        default:
                                                x = a + z;
                                }
                        }
                }
                print(x);
        }
        void AddValue(ByRef int x, ByVal int y){
                x = x + y;
        }
        int SubValue(ByVal int x, ByVal int y){
                return x - y;
        }
}
```

2.3.2. Mapping Source Code to Code Model

In order to produce well-formed representation of source code we have to define formal constraints for creating and validating KDM Code model elements. In the following table (Table 2.2) we present a mapping of the constructs from the example language with the elements of the KDM together with well-formedness constraints that apply for representing element. For each type of KDM model element, we introduce a stereotype which indicates the language construct being represented. In addition to a mapping rule, we provide references to constraints specified using OCL and presented in the Annex A. A source column of the ta-

ble points out to the section or specific annex of the KDM specification (KDM v1.3), where the semantics of target element are provided.

Table 2.2. A mapping of example language constructs with KDM Code model elements

Language construct	***KDM Element***	***Constraints***	***Source***
Module	CompilationUnit	M1-M4	KDM, 12.5
Procedure	CallableUnit	CE1-CE2	KDM, 12.6
Signature	Signature		
Parameter	ParameterUnit.kind = {byValue, byReference}		
Statements			
AssignmentStmt	ActionElement.kind=’Assign’	Input: SC1.1; Output: SC1.2; Control: SC1.3	Micro KDM, A.5
IfStmt	ActionElement.kind={’Compound’ | ’Condition’}	Input: SC2.1; Control: SC2.2;	Micro KDM, A.5
SwitchStmt	ActionElement.kind=’Switch’	Input: SC3.1; SC3.3; Control:SC3.2; SC3.4;	Micro KDM, A.5
SwitchCase	ActionElement.kind=’Guard’		
LoopStmt	ActionElement.kind={’Compound’ | ’Condition’}	Input: SC2.1; Control: SC2.2;	Micro KDM, A.5
Continue	ActionElement.kind=’Goto’ with flow to init of loop		
Break	ActionElement.kind=’Goto’ with flow to the next action		
InvocationStmt	ActionElement.kind=’Call’	Section Call-Return Mechanism	Micro KDM, A.5
ReturnStmt	ActionElement.kind=’Return’	Section Call-Return Mechanism	Micro KDM, A.5
Expressions			
BooleanExpression	ActionElement.kind={‘Not’|‘And’, ‘Or’|‘Xor’}		
ComparisonExpression	ActionElement.kind={‘Equals’ | ’NotEqual’ | ‘LessThanOrEqual’ | ‘LessThan’ | ‘GreaterThan’ | ‘GreaterThanOrEqual’}	Input: EC1.1; Output: EC1.2;	Micro KDM, A.2
ArithmeticalExpression	ActionElement.kind={'Add' | 'Multiply' | 'Substract' | 'Divide' | 'Remainder' | 'Negate' | 'Successor'}	Input: EC2.1;	Micro KDM, A.3
InvocationExpression Argument	ActionElement.kind=’Call’ Reads	Section Call-Return	

Table 2.2. (Continued)

Language construct	*KDM Element*	*Constraints*	*Source*
		Mechanism	
GlobalVarDef	StorableUnit.kind=global	DC1	KDM, 12.7
LocalVarDef	StorableUnit.kind=local	DC1	KDM, 12.7
LiteralExp	ValueElement	VC1-VC3	KDM, 12.8
Type	LanguageUnit Datatype	TC1	KDM, 12.9

Well-formedness constraints for the KDM representation of the source code are specified in regards with the *Micro*KDM semantics defined in the meta-model specification. Although one would notice that certain constraints might be merged into a single, we leave them separated for the simplicity and maintainability purposes.

2.3.3. Visualizing Intermediate Representation

In order to visualize KDM intermediate representation for the further discussion we employ UML Class diagram notation and introduce UML Profile consisting of stereotypes corresponding to KDM elements. The main principles of the profile construction are as follows.

- For the *KDMEntity* element we introduce corresponding stereotype which metaclass we define as UML Classifier. Specific types of *KDMEntity* are defined as stereotypes that inherit from the base stereotype (in correspondence to the KDM inheritances). If specific type of *KDMEntity* has certain properties, we define them as stereotype tag values.
- For the KDMRelationship element we define stereotype which metaclass is UML Association. As in the case of *KDMEntity*, specific types of *KDMRelationship* are defined as specialized stereotypes. In diagrams, we use navigable associations to give an emphasis on the direction of the relationship (in regards with from-to attributes of *KDMRelationship*).
- KDM *AggregatedRelationship* is represented using UML Composite Association element. This kind of relationship represents element containment in more general element.
- KDM *Stereotype* which is assigned to a model element is defined as Stereotype tag.

For the purpose of clarity and simplicity, we omit additional KDM elements, such as *SourceRef* or *Audit* elements. Instead of this, the snippet of source code represented within KDM element is provided as the name of an element.

2.3.4. Data Flow Representation

The data flow of the program is the vital information for any kind of static program analysis. Therefore it is very important to define the principles for the data flow representation capabilities within the KDM. For this reason, we start by considering a trivial statement $x = a + b * (c - d)$ presented in the figure below (Fig. 2.4). As it can be seen, representation of this statement according to the semantics of *Micro*KDM produces an exhaustive view of an abstract syntax tree (AST), decorated with the semantics of data and control flows.

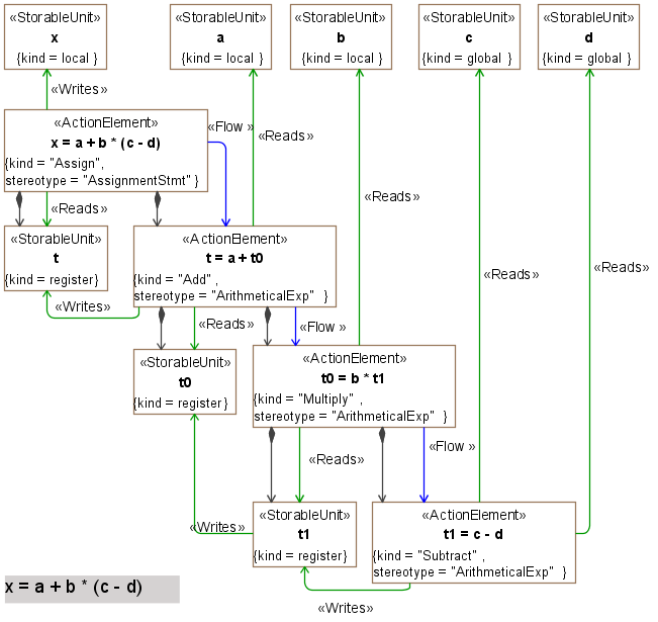

Fig. 2.4. An example of detailed representation of program statement, including composition of subexpressions, data and control flow relationships

According to the semantics of KDM representation, *Reads* relationship represents an association between an *ActionElement*, which implements a flow of data **from** a certain data element to a corresponding data element. Writes relationship represents an association between an *ActionElement*, which implements a flow of data **to** a certain data element from the corresponding data element. Data elements represent computational objects of the existing software systems that are defined by some type and are assigned with a particular value.

It should be noted that such fine-grained representation may not be efficient in certain data flow analysis techniques. Therefore, in certain cases we will de-

fine *ActionElements* that correspond to assignment statements containing complex expressions as a single *ActionElement* by accepting the following conditions:

- *ActionElements* representing complex expression are composed in the *ActionElement* representing assignment statement.
- There is no other *ControlFlow* relationship other than *Flow* between aggregated *ActionElements* that represents sub-expressions of complex expressions (e.g. t1=c-d).

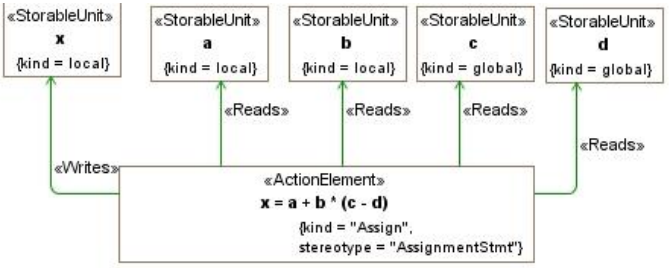

Fig. 2.5. Aggregated *ActionElement* representing the program statement

As we can see in Fig. 2.5, aggregated *ActionElement* includes all the relationships that represent accessing variables within the statement and disregards relationships that represent intermediate write to register variables.

2.3.5. Control Flow Representation

KDM provides a set of control elements capable to represent basic control flow relationships. This set includes *Flow* relationship for representing unconditioned flow of computation, *GuardedFlow*, *TrueFlow* and *FalseFlow* for representing conditioned flows, and *ExceptionFlow* and *ExitFlow* flows for representing exception mechanism. The set is sufficient enough to construct flow graph of a program written in the most common programming languages.

It is important to notice, that in correspondence with KDM semantics, conditional statement as well as loop statement (Fig. 2.6), which condition is complex expression rather than Boolean variable is represented as compound action elements. The first action element within the compound one represents evaluation of expression of condition, the second represents condition evaluation. The former is labelled with any kind of action element that represents certain programming language expression type and corresponds to *Micro*KDM semantics. The latter is labelled with the ‘Condition’ kind and strictly contains to outgoing flows: true and false.

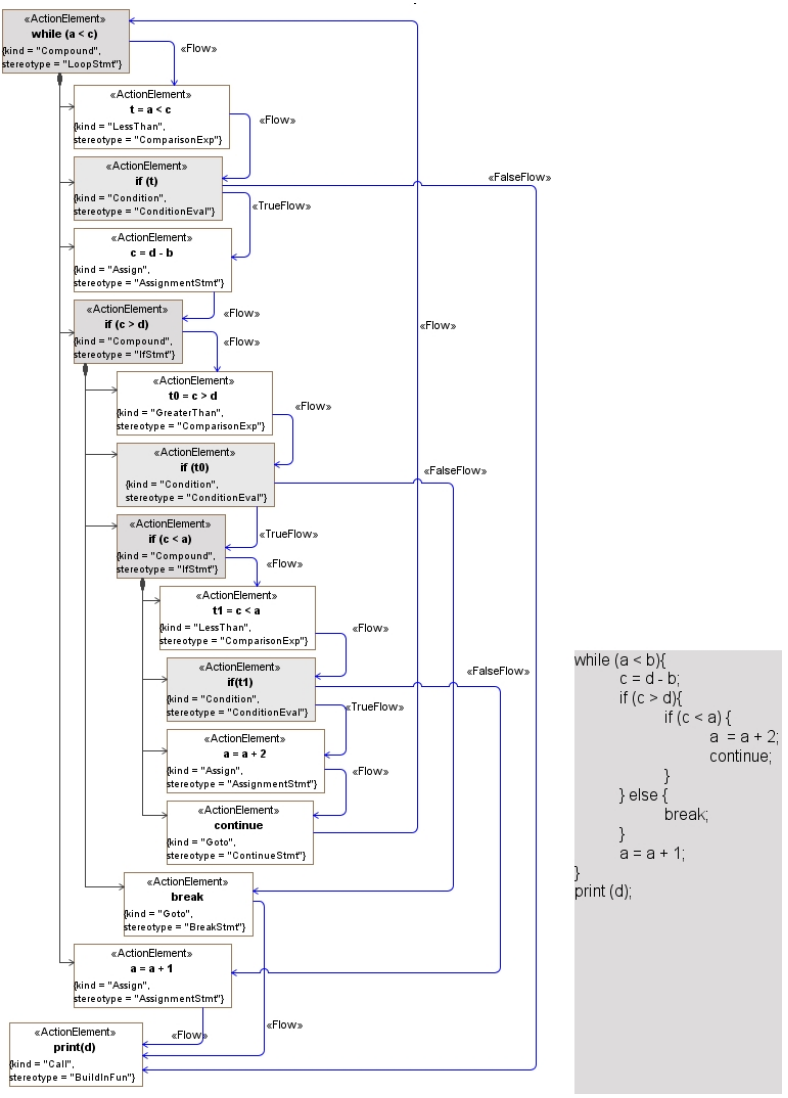

Fig. 2.6. An example representing flow graph of a part of procedure. The gray boxes within this diagram represent condition statements

2.3.6. Call-return Representation

KDM provides a set of modelling elements (Fig. 2.7) for representing call-return mechanism supported by the most common programming languages. A call to a certain procedure or a function is represented with a *Call* element. The action element from which the call relationship outgoes represents a kind of call state-

ment or invocation expression. The target of call relationship is a *CallableUnit*, which represents a global or local procedure, a static or non-static method, a virtual method, or an interface element.

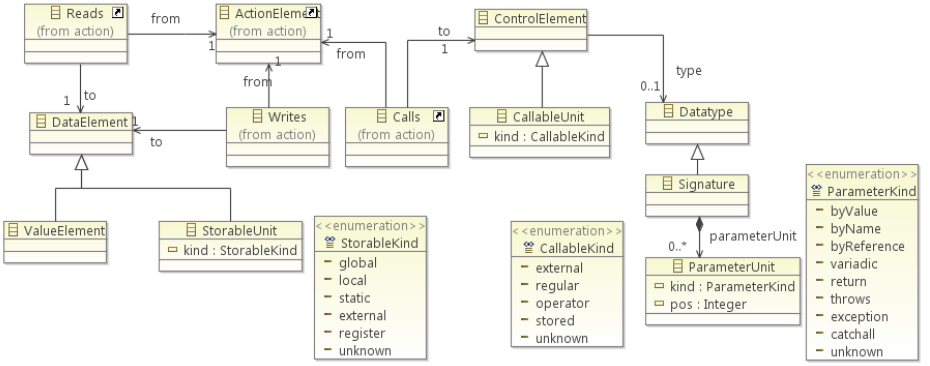

Fig. 2.7. A fragment of KDM (OMG 2011a) meta-model representing elements for constructing call-return mechanism

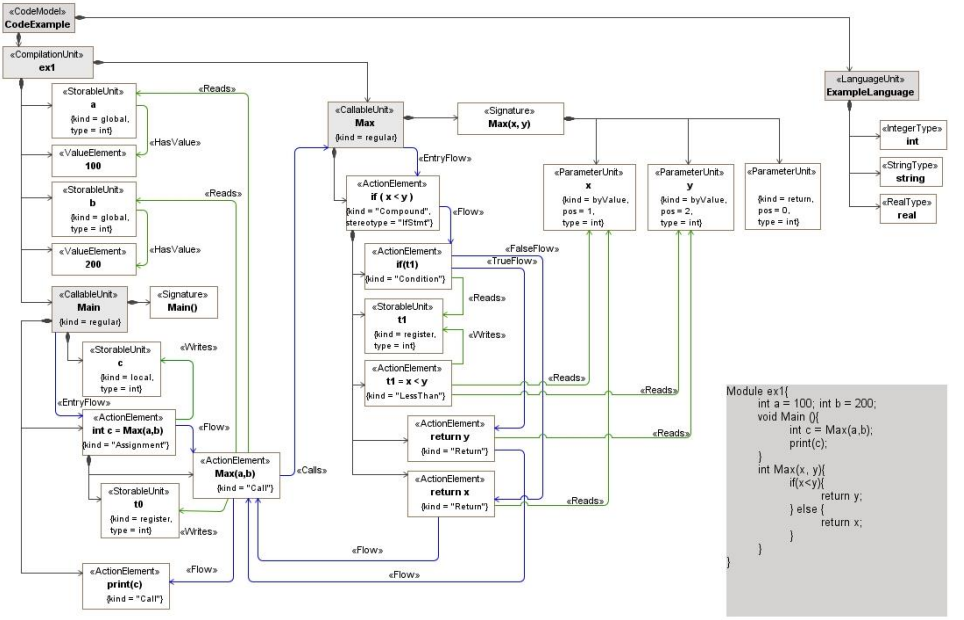

Fig. 2.8. Example of Code model representing call-return mechanism of program shown in the bottom left corner

The example given in the Fig. 2.8 presents the call-return mechanism. The *ActionElement* representing function invocation expression *Max*(*a*,*b*) reads data from storable units and passes this data to invoked function as actual parameters. The signature of *CallableUnit Max* has an ordered set of *ParameterUnit* elements that represent the formal parameters as well as return parameter of function which is depicted by the kind attribute. To represent signatures of programming languages that allow binding of actual parameters by name rather than by a position, the mechanism should be constructed with correct positions of the named parameters and with determined appropriate *ParameterUnit* elements as targets for relations to named parameters.

2.4. Representing Database Objects

In our study we refer to the ANSI/SQL-2008 grammar (ISO/IEC 2008) as the main source for defining the schema of database. For this reason we created SQL meta-model which is simplified version of the specification given in the standard (see Annex D). Below we present a fragment of the meta-model, with the main emphasis on the structure of table definition (see Fig. 2.9).

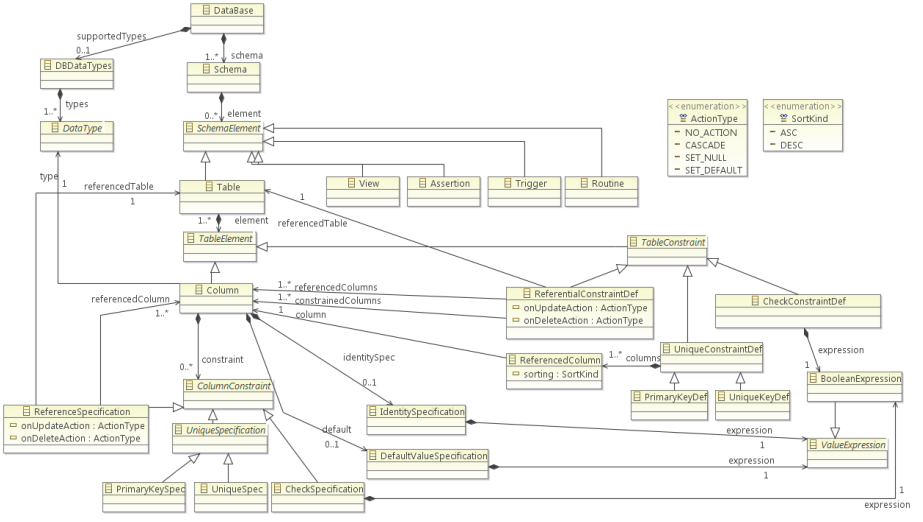

Fig. 2.9. A fragment of SQL abstract syntax meta-model

In order to represent the organization of data structure in the existing software system, the KDM supports a set of meta-model elements defined within the Data package as shown in Fig. 2.10. Some facts about the business domain in

the database are typically determined by a Data Description Language (DDL, SQL) but although may implemented using program code (as we will discuss in the next chapter).

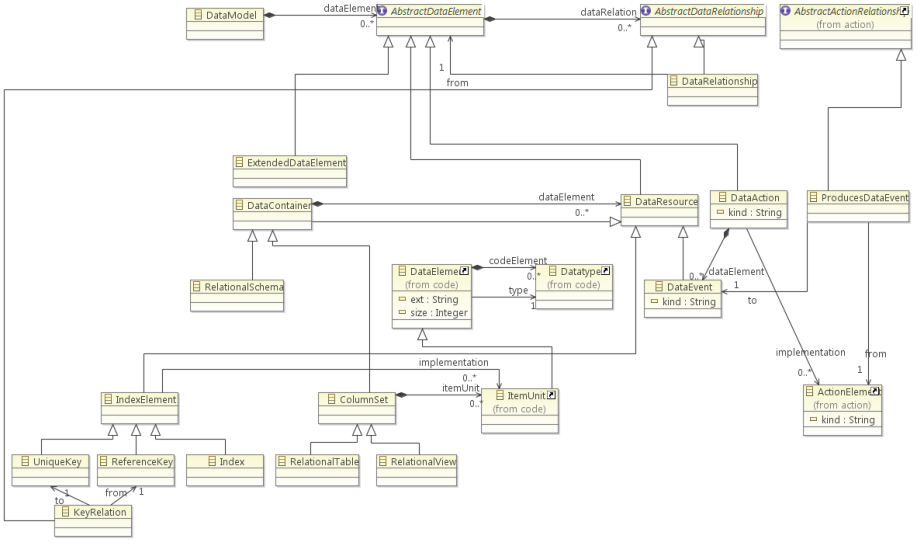

Fig. 2.10. A fragment of KDM metamodel representing the basic elements of the Data model (OMG 2011a)

It should be noted that KDM provides only the fundamental elements for representation of the relational database concepts. For example, there are no standard KDM elements to represent particular column level or table level constrains (e.g. Check, NotNull, Default, etc.), although they are standard SQL constructs. In order to represent these constructs, we use the *DataEvent* elements, which are intended to represent changes in entities or in relations among entities, such that constrains may be considered as post-conditions that must be fulfilled after the changes. The conditions (i.e. SQL Boolean expressions) are represented as *ActionElements* aggregated within the *DataEvent* element as defined in (OMG 2011a).

The following table (Table 2.3) summarizes SQL to KDM mapping rules used to obtain a representation of the database objects. Annex B presents well-formedness constraints that correspond to the KDM semantics.

Table 2.3. A mapping of SQL constructs with KDM Data model elements

SQL Construct	***KDM Element***	***Con-straints***	***Source***
DataBase	Catalog	DR	KDM 18.5.3
Schema	RelationalSchema	DR	KDM 18.5.4
Table	RelationalTable	DR	KDM 18.6.2
Column	ItemUnit	See section 2.3.	KDM 12.7.5
Index	Index	IE1,IE2,I1	KDM 18.7.1, 18.7.4
PrimaryKeySpec | PrimaryKeyDef	UniqueKey(implementation ← ItemUnit)	UK	KDM 18.7.2
ReferenceSpecification | ReferentialConstraintDef	ReferenceKey(implementation ← ItemUnit, attribute(tag ← {"OnDeleteAction" | "OnUpdateAction"}, value ← {"NO ACTION" | "CASCADE" | "SET NULL" | "SET DEFAULT"})); KeyRelation(from ← ReferenceKey, to ← UniqueKey)	RK	KDM 18.7.3
UniqueSpec | UniqueKeyDef	UniqueKey(implementation ← ItemUnit)	UK	KDM 18.7.2
NotNullSpec | NotNullDef	DataEvent(kind ← "Insert", abstraction ← ActionElement(kind ← "NotNull"))	DR	KDM 18.5.5
CheckSpecification | CheckConstraintDef	DataEvent(kind ← "Insert", abstraction ← ActionElement(kind ← "Check"))	DR	KDM 18.5.5
IdentitySpecification	DataEvent(kind ← "Insert", abstraction ←	DR	KDM 18.5.5

Table 2.3. (Continued)

SQL Construct	*KDM Element*	*Con-straints*	*Source*
	ActionElement(kind ← "Identity"))		
DefaultValueSpecification	DataEvent(kind ← "Insert", abstraction ← ActionElement(kind ← "Assign"))	DR	KDM 18.5.5
View	RelationalView	DR	KDM 18.6.3
ViewColumn	ItemUnit		KDM 12.7.5
Statement	DataAction (kind ← [name of expression kind])	DR	KDM 18.5.6
Expression	ActionElement (kind ← [name of expression kind])	See section 2.3	KDM 13.3.1
Predicate	ActionElement (kind ← [name of predicate kind])	See section 2.3	KDM 13.3.1
Value	Value	See section 2.3	KDM 12.8.1
DataType	CodeModel(codeElement ← LanguageUnit(dataType ← [coressponding SQL data type]))	See section 2.3	

As it was already mentioned, the KDM does not support specific elements for representing various kinds of SQL expressions and predicates. In order to preserve the semantics of source language, we define an extension family of stereotypes corresponding to SQL constructs within the KDM representation and apply corresponding stereotypes for action elements that represent these constructs. However, formal definition of this extension family is not in the scope of this work and we leave this issue for the further research.

The Fig. 2.11 presents an example of the KDM representation of a fragment of database schema, obtained in respect with the mapping rules defined by the mapping list provided in the Table 2.3.

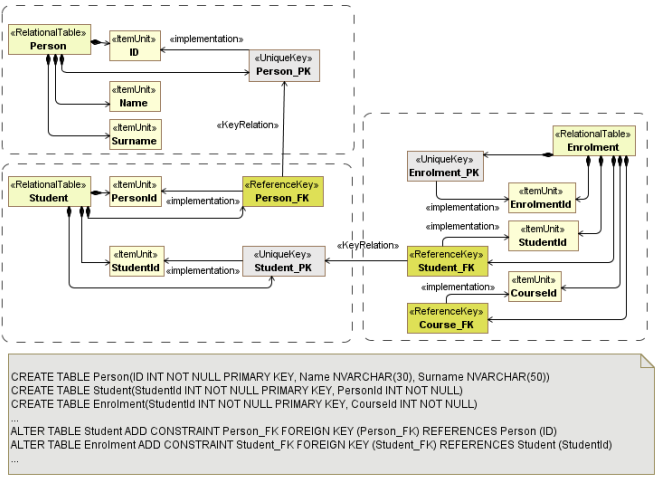

Fig. 2.11. Fragment of database schema represented using KDM Data model elements

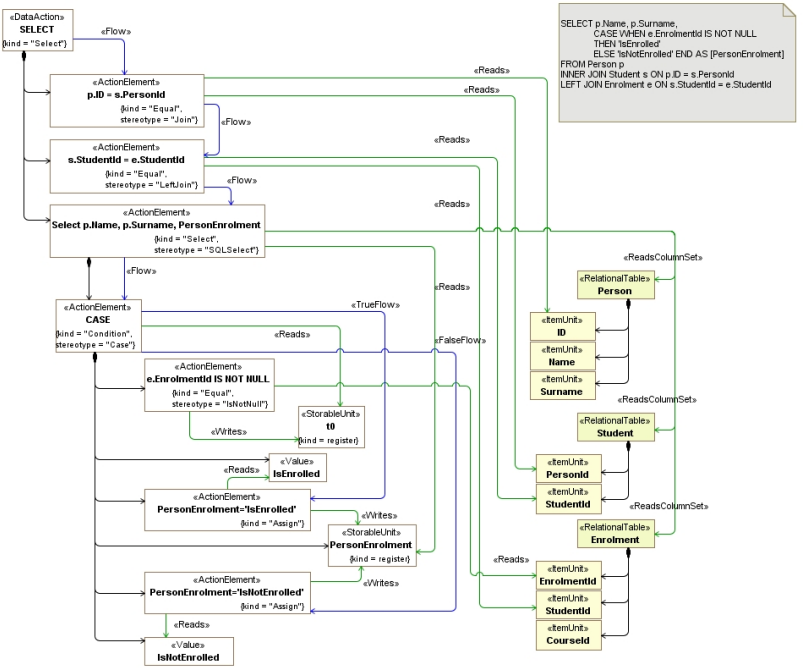

Fig. 2.12. Representing SQL query composition, control and data flow relationships

An example of SQL query representation within the KDM is presented in the Fig. 2.12. It can be seen, that this representation is constructed following the same principles as constructing the representation of code, i.e. it includes structure, control, and data flow relationship that allows performing data flow analysis on the queries, in order to discover various kinds of facts about the business domain. Contrarily to the code model, this representation has additional data flow relationships (*ReadsColumnSet*/*WritesColumnSet*) that allow defining inclusion of relational tables into the select expressions.

2.5. Data Flow Analysis within the Knowledge Discovery Meta-model

Data flow analysis views computation of data over expressions and transitions of data through assignments to variables. The analysis process involves two steps: discovering the effect of individual statements on the expression (local analysis) and relating these effects across statements in the program (global analysis) (Khedker *et al*. 2009). The effects across analysis units are related by propagating information either in a backward, or in a forward manner.

The data-flow analysis is a kind of program analysis, characterized by the following characteristics (Nielson *et al*. 2004; Khedker *et al*. 2009).

- ***Application***: data flow analysis can be used for determining semantic validity of a program, understanding the behaviour of a program, or transforming a program.
- ***Approach of program analysis***: data flow analysis uses constraint resolution systems based on data flow equations.
- ***Time***: data flow analysis is mostly static analysis, though dynamic slicing involves dynamic data flow analysis.
- ***Scope***: data flow analysis may be performed on almost all levels of scope in a program, including a block of statements (local data flow analysis), a procedure consisting multiple of blocks (intraprocedural data flow analysis), and across procedures within a program (interprocedural data flow analysis).
- ***Flow sensitivity***: data flow analysis is almost always flow sensitive because it computes program point specific information.
- ***Context sensitivity***: interprocedural data flow analysis may be either context sensitive, or context insensitive. The former is inefficient and most practical algorithms employ a limited amount of context sensitivity.

- ***Granularity***: data flow analysis may be exhaustive (i.e. information is derived from scratch) or incremental (i.e. incorporates previously derive information).
- ***Program representation***: there are many different intermediate representation forms for data flow analysis, including abstract syntax tree (AST), directed acyclic graph (DAG), control flow graph (CFG), program flow graph (PFG), call multigraphs (CG), program dependence graph (PDG), and static single assignment (SSA).

The most common representation forms for intraprocedural data flow analysis are CFG, PFG, SSA, and PDG. The interprocedural data flow analysis uses a combination of CG and CFG/PFG, or extended versions of PDG, called system dependence graph (SDG). Representation of the above given concepts within KDM will be discussed in the next section.

In the following table we present basic concepts in data flow analysis and corresponding KDM elements.

Table 2.4. Basic concepts of dataflow analysis

Data flow analysis concept	***Corresponding KDM concept***
Variable $x \in \mathbb{V}ar$	A type of *DataElement* except the *ValueElement*, i.e. *StorableUnit*, *ItemUnit*, *IndexUnit*, *MemberUnit*, or *ParameterUnit*.
Expression $e \in \mathbb{E}xp$	An *ActionElement* which kind corresponds to MicroKDM expression kinds and is extended with a special stereotype denoting the kind of expression in the programming language.
Definition $d \in \mathbb{D}ef$	The *ActionElement* which kind is "Assign".
Basic block $n \in CFG$	An *ActionElement* which kind is "Compound" or "Nop" and which is composed of one or more *ActionElements* representing consecutively executed program statements with a strictly sequential flow between them.

A basic block is a unit of data flow analysis. The effects between basic blocks are associated over entry and exit points denoted by Entry(*ActionElement*$_n$) and Exit(*ActionElement*$_n$). Data flow information associated with them are denoted as In_n and Out_n. The data flow information which is generated within the *ActionElement*$_n$ is denoted as Gen_n and information which becomes invalid is denoted as $Kill_n$.

In respect with that we can define derived properties of an action element as follows:

context ActionElement::genVars : Collection(code::DataElement)
derive: **self**.getOutbound() -> select (r | r.oclIsTypeOf(action::Reads)) and not r.oclAsType(action:: Reads).to.oclIsTypeOf(code::ValueElement))

context ActionElement::killVars : Collection(code::DataElement)
derive: **self**.getOutbound() -> select (r | r.oclIsTypeOf(action::Writes))) and not r.oclAsType(action::Writes).to.oclIsTypeOf(code::ValueElement)

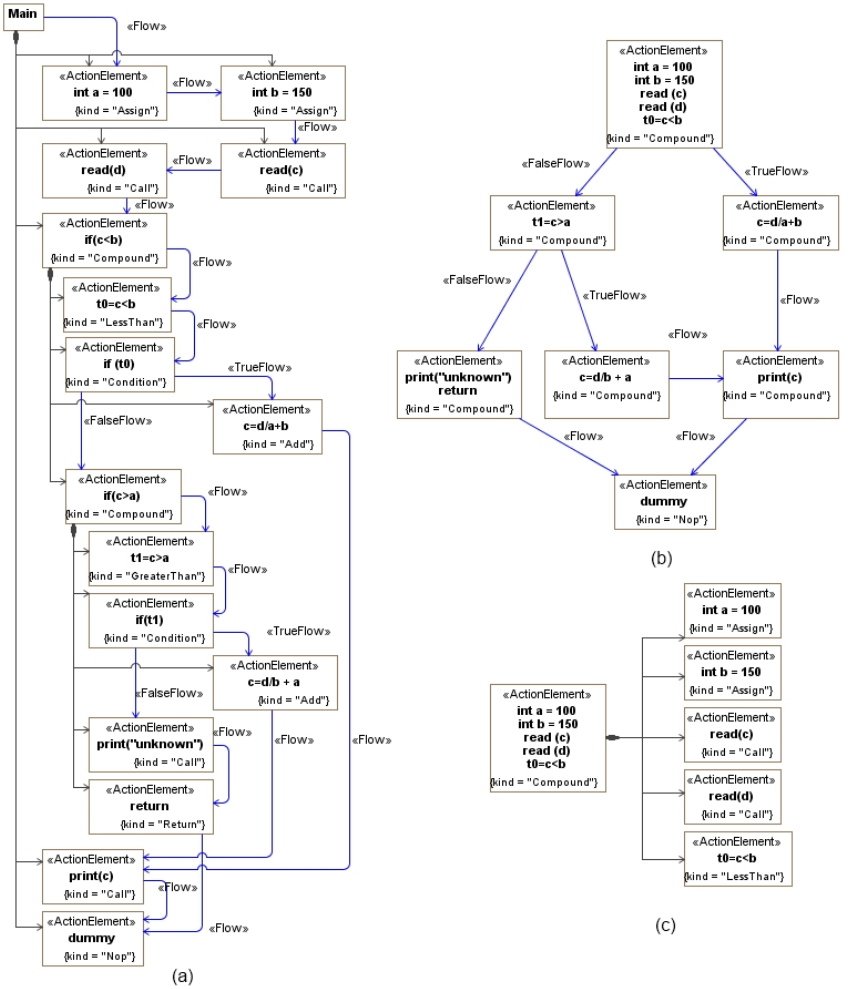

Fig. 2.13. Producing Control Flow from KDM CallableUnit: a) Flow graph within the CallableUnit that represents example procedure; b) Derived Control Flow graph; c) Aggregation of compound action element

The control flow graph (CFG) produced from the KDM Code model consists of nodes representing *ActionElements* and edges representing *ControlFlow* relationships. An edge in CFG denotes the predecessor and successor relationships between nodes n_1 and n_2, such that a relationship $n_1 \rightarrow n_2$ indicates that n_1 is predecessor of n_2 and n_2 is a successor of n_1. Predecessors and successors of a node *n* are denoted in terms of *pred*(n) and *succ*(n) and may be defined as follows:

context ActionElement::pred : Collection(action::ActionElement)
derive: **self**.getOutbound() -> select (r | r.oclIsKindOf(action::ControlFlow)).to

context ActionElement::succ : Collection(code::DataElement)
derive: **self**.getInbound() -> select (r | r.oclIsKindOf(action::ControlFlow)).from

As in the classical data flow analysis (Khedker *et al*. 2009), we assume that the CFG has two distinguished unique nodes $ActionElement_{Start}$ which has no predecessor and $ActionElement_{End}$ which has no successor (Fig. 2.13, b). If such nodes do not exist, we introduce a dummy *ActionElement* kind of "Nop" (see MicroKDM, A.5 (OMG 2011a)) without affecting the semantics of a program. We can then assume that every basic block represented with an $ActionElement_n$ is reachable from the $ActionElement_{Start}$ and that the $ActionElement_{End}$ is reachable from $ActionElement_n$.

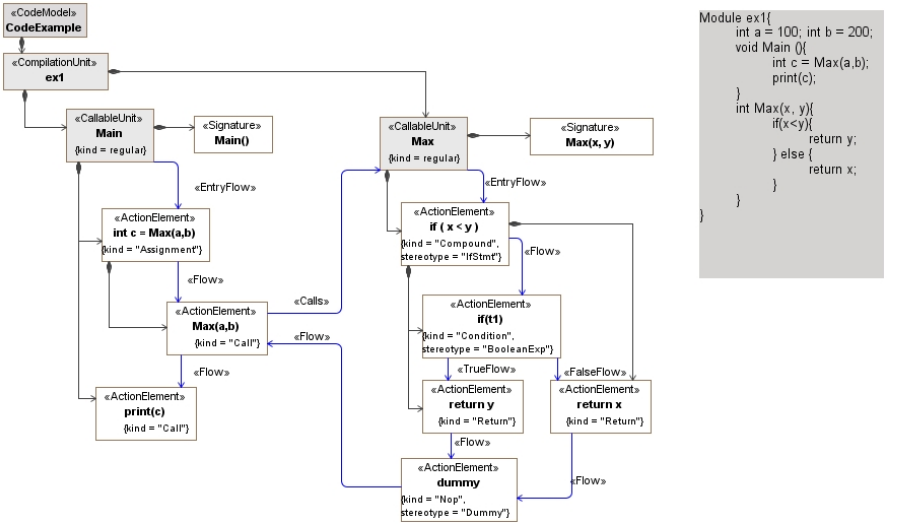

Fig. 2.14. Call-return representation in respect with modified control flow graph

It should be noted, that in order for this assumption to be hold, we have to change the call-return mechanism construction principles (as it is presented in Fig. 2.14) in certain cases. When there is more than one return statement within a procedure, a flow from every return node (*ActionElement* kind of "Return") is associated with the $ActionElement_{\text{End}}$ instead of the invocation expression or statement representing *ActionElement*. The figure below shows the modified (without affecting semantics) representation of the of call-return mechanism.

So far we have discussed on the basic concepts of data flow analysis and introduced the principles of construction data flow framework within the KDM. In the next sections we will overview several most common data flow analysis techniques and define them in the context of KDM Code model.

2.5.1. Intraprocedural Analysis

Intraprocedural analysis derives behavioural information with the body of the single procedure (or function, method, etc.). The information obtained by this analysis is further used to derive the results of interprocedural (i.e. between multiple procedures) analysis. In the following we present several most common intraprocedural techniques that were employed in the business knowledge extraction method.

2.5.1.1. Live Variable Analysis

Live variable analysis is one of the backward data flow analysis techniques that operates on variables and is defined as follows.

Definition 1:

A variable $x \in \mathbb{V}ar$ is live at a program point u if some path from this point u to the End contains a use of x which is not preceded by its definition.

The data flow equations which define live variables in the context of *ActionElement* may be defined as derived properties:

$\mathbf{Live_{In}}$
```
context ActionElement::liveInVars : Sequence (code::DataElement)
derive: (self.liveOutVars - self.killVars) -> union(self.genVars)
```
(2.1)

$\mathbf{Live_{Out}}$
```
context ActionElement::liveOutVars : Sequence (code::DataElement)
derive:
if self = self.getOwnerCallableUnit().end then
        OrderedSet{}
else
        self.succ.incomingVars
endif
```
(2.2)

Where *getOwnerCallableUnit*() is a query operation that returns the *CallableUnit* which owns the ActionElement and *end* is a final node of CFG (i.e. $ActionElement_{\text{End}}$ - *ActionElement*.kind="Nop"), Set{} is an empty set.

Variable liveness is computed by iteratively computing (depth first traversal) data flow equations for each block of CFG, until *ActionElement*.liveInVars and *ActionElement*.liveOutVars between iterations converge. The use of operation *union*() in 2.1 equation means that the liveness information at *Exit*(*ActionElement*$_n$) is a superset of the liveness information at *Entry*(*ActionElement*$_s$) where *ActionElement*$_s$ is *ActionElement*$_n$.succ. This is due to that liveness concerns "any path" that may lead to the use of variable in the block. There are four possibilities concerning the liveness of the variable (specified with OCL correspondingly):

a) liveness of *x* is unaffected by *ActionElement*

```
ActionElement.genVars->exists(g | g<>x) and
ActionElement.killVars->exists(k | k<>x)
```

b) liveness of *x* is generated by *ActionElement*

```
ActionElement.genVars->exists(g | g=x) and
ActionElement.killVars->exists(k | k<>x)
```

c) liveness of *x* is killed by *ActionElement*

```
ActionElement.genVars->exists(g | g<>x) and
ActionElement.killVars->exists(k | k=x)
```

d) liveness of *x* is unaffected by *ActionElement* in spite of *x* is being rewritten

```
ActionElement.genVars->exists(g | g=x) and
ActionElement.killVars->exists(k | k=x)
```

For a given variable *x*, live variable analysis discovers a set of liveness paths, where each path represents a sequence of ActionElements (*ActionElement*$_1$, *ActionElement*$_2$, …, *ActionElement*$_k$) – a potential execution path starting at *ActionElement*$_1$ such that:

- *ActionElement*$_k$ – contains upwards exposed use of *x*,
- *ActionElement*$_1$ is either *ActionElement*$_{START}$ or assignment to *x*,
- no other *ActionElement* in path contains an assignment to *x*.

In opposite, a variable is dead if it is being dead along all paths. If the variable is contained in *ActionElement*.liveInVars and *ActionElement*.liveOutVars it is dead instead of being live, when the following are met:

1. *ActionElement*.genVars contains a collection of variables modified within *ActionElement* (i.e. *oclIsTypeOf*(action::Write) instead of *oclIsTypeOf*(action::Reads)). *ActionElement*.killVars contains a collection of variables used anywhere regardless of what precedes or follows the uses (i.e. *oclIsTypeOf*(action::Reads) instead of *oclIsTypeOf*(action::Write)).
2. Instead of "any path" we have to identify "all paths" (i.e. instead of *union*() we will use operation *instersection*()).

3. We have to use universal set at the *Exit*(*ActionElement*$_{\text{End}}$) rather than empty set, therefore we define the query operation on *ModuleUnit* which returns both global and local variables within a module: *ModuleUnit*::getAllVariables() : Collection(code::DataElement).

The data flow equations for dead variable analysis in the context of *ActionElement* may be defined as the following derived properties:

Dead$_{\text{In}}$ (2.3)

```
context ActionElement::deadInVars : Sequence (code::DataElement)
derive: (self.deadOutVars - self.killVars)) -> intersection(self.genVars)
```

Dead$_{\text{Out}}$ (2.4)

```
context ActionElement::deadOutVars : Sequence (code::DataElement)
derive:
if self = self.getOwnerCallableUnit().end then
        self.getOwnerModuleUnit().getAllVariables()
else
        self.succ.deadInVars
endif
```

Where *getOwnerModuleUnit*() is the query operation which for the given action element returns its parent module unit (e.g. compilation unit) element.

Live and dead variable analysis is used as a basis for dead code elimination in program optimization. In the case of knowledge discovery, it is obvious that if a variable is not live along some path of execution, there is no need to include it in candidate business terms list and if a variable is dead at assignment, we can easily eliminate this assignment from the candidate facts list.

2.5.1.2. Reaching Definition Analysis

As it was already introduced, a definition of a variable is an *ActionElement* which kind is "Assign" and which has an action relationship type of *Writes* to a particular *DataElement*. For the purpose of analysis, each *ActionElement*.kind="Assign" is important and will be labelled uniquely such that different occurrences of the same assignment become different definitions. The Reaching Definition Analysis computes for each program point, which assignments may have already been made and not overwritten when program execution reaches this point along some path (Nielson *et al*. 2004).

Definition 2:

A definition $d \in \mathbb{D}ef$ of a variable $x \in \mathbb{V}ar$ reaches program point u if d_i occurs on some path from *Start* to u and is not followed by any other definition of x on this path.

The analysis operates on the information that within an *ActionElement* is derived as follows:

```
context ActionElement::genDefs : Set (action::ActionElement)
derive: self.codeElement -> select (e | e.oclIsKindOf(action::ActionElement) and
e.oclAsType(action::ActionElement).kind='Assign')
```

```
context ActionElement::killDefs : Set (action::ActionElement)
derive:
let proc:   code::CallableUnit = self.getOwnerCallableUnit(),
            allDefs: Sequence(action::ActionElement) =
            if self = proc.entryFlow.to
            then proc.getAllDefinitions()
            else self.succ.genDefs endif
in
allDefs –> iterate(d: action::ActionElement; killDefs: Sequence(action::ActionElement) =
Sequence{} |
if d.killVars -> intersection(self.genDefs.killVars) -> notEmpty()
then killDefs -> append(d)
else Sequence{} endif)
```

Where *getAllDefinitions*() is the query operation which returns a set of all definitions within a procedure (CallableUnit), $ActionElement_n$.genDefs contains a set of definitions generated within $ActionElement_n$, and $ActionElement_n$.killDefs contains definitions that are killed within this action element.

The data flow equations of reaching definition analysis in the context of KDM may be constructed as follows:

$Reach_{In}$
```
context ActionElement::reachInDefs : Sequence
(ation::ActionElement)
derive:
if self.succ -> notEmpty() then
        self.pred.reachOutDefs
else
        Sequence {}
endif
```
(2.5)

$Reach_{Out}$
```
context ActionElement::reachOutDefs : Sequence
(action::ActionElement)
derive: (self.reachInDefs - self.killDefs) -> union(self.genDefs)
```
(2.6)

Computing of these equations is similar to liveness analysis except that in this case data flow is computed in forward rather than in backward manner. It should be noted, that computing reaching definition equations requires that the *Enter*($ActionElement_{Start}$) would contain initial definitions of local variables.

For definition $ActionElement_i$ of x, reaching definition discovers a set of reaching definition paths. This path is a sequence of *ActionElement* ($ActionElement_1$, $ActionElement_2$, …, $ActionElement_k$) – a potential execution path starting at $ActionElement_1$ such that:

- $ActionElement_1$ contains the definition $ActionElement_i$.
- $ActionElement_k$ is either $ActionElement_{End}$ or contains a definition of x.

- no other *ActionElement*$_i$ in the path, where *1<i<k*, contains a definition *x*.

The reaching definitions are used in construction of direct links *ActionElements* that produce values and *ActionElements* that use them, also called *use-def* and *def-use* chains. For that purpose, we introduce additional *ActionRelationship* element with assigned stereotype, (either *use-def* or *def-use*), such that:

- *ActionRelationship* represents "use-def" chain when property **from** is associated with the action element that includes the use of variable *x*, and property **to** is associated with the action element that defines the variable *x*.
- *ActionRelationship* represents "def-use" chain when property **from** is associated with the action element that defines variable *x*, and property **to** is associated with the action element that uses the variable *x*.

There are several usages of reaching definition analysis in our study. One of them is for constructing the system dependence graphs, and other is for performing copy propagation. The latter allows tracking the value of variable assigned directly, over enumeration or derived from database, within the data flow along the execution paths of the control flow. A definition of the form *x=y* is called a *copy* because it copies the value from *y* to *x*, and when such definition reaches a use of *x*, and no other definition of *x* reaches that use then the use of *x* can be replaced by *y*. In the knowledge discovery process, we treat a *copy* as a synonymous candidate to business term. Once we have agreed on the derivation of *y*, in the further analysis we can operate on the meaning of *y*.

2.5.1.3. Available Expressions Analysis

The available expression analysis determines for each program point, which expressions must have already been computed and not later modified in all paths from the program point to the start (Nielson *et al*. 2004).

The analysis operates on information which is derived as follows:

```
context ActionElement::genExps : Collection(action::ActionElement)
derive:
let expressionKinds : Set (action::ActionElement) = Set {expression kinds corresponding
to MicroKDM}
in self.codeElement -> select (e | e.oclIsKindOf(action::ActionElement) and
expressionKinds -> exists(e.kind))


context ActionElement::killExps : Collection(action::ActionElement)
derive:
let expressionKinds: Set (action::ActionElement) = Set {expression kinds corresponding
to MicroKDM},
```

```
availableExp: Sequence (action::ActionElement) =
if self.succ -> notEmpty() then
        self.succ -> select(a | expressionKinds -> includes (a.kind)) -> asOrderedSet()
else
        Sequence {}
endif
in
availabeExp -> iterate (e : action::ActionElement; killedExp : Sequence
(action::ActionElement) = Sequence {} |
        if (self.killVars -> intersection(e.genVars) -> notEmpty()) then
                killedExp -> append(e)
        else
                killedExp
        endif)
```

ActionElement.genExps contains generated within *ActionElement* expressions whereas *ActionElement*.killExps contains all expressions whose operands are modified in *ActionElement*.

Definition 3:

An expression $e \in \mathbb{E}xp$ is available at a program point u if all paths from *Start* to u contain a computation of e which is not followed by an assignment to any of its operands.

The data flow equations of available expression analysis in the context of KDM may be constructed as follows:

$\mathbf{AvExp_{In}}$

```
context ActionElement::availInExps : Sequence (action:ActionElement)
derive:
if self.pred -> isEmpty() then
  Sequence{}
else
  self.pred -> iterate(a:action::ActionElement; avExp: Se-
quence(action::ActionElement) = Sequence{} | avExp -> intersec-
tion(a.availOutExps))
endif
```

(2.7)

$\mathbf{AvExp_{Out}}$

```
context ActionElement::availOutExps: Sequence (action::ActionElement)
derive: (self.availInExps - self.killExps) -> union(self.genExps)
```

(2.8)

For a given expression e represented as *ActionElement*, available expression analysis discovers a set of availability paths. Each availability path is a sequence of compounded *ActionElements* (*ActionElement*$_1$, *ActionElement*$_2$, …, *ActionElement*$_k$) – a potential execution path starting at *ActionElement*$_1$ such that:

- *ActionElement*$_1$ contains a downwards exposed computation of expression e,
- *ActionElement*$_k$ is either *ActionElement*$_{End}$, or contains a computation of e, or an assignment to some operand of e,

- no other *ActionElement*$_i$ in the path contains a computation of *e*, or an assignment to any operand of e, and
- every path ending on *ActionElement*$_k$ is an availability path for *e*.

Available expression analysis is useful technique in program optimization as it allows avoiding re-computation of the same expressions. From the business knowledge extraction point of view, the information obtained by this technique enables refinement of term and fact units as well as identification of their actual implementations within the source code.

2.5.1.4. Dominators

In order to accurately apply data flow analysis computations we have to define appropriate way to handle loops within a body of procedure. So far, we have introduced their representation just as any other kind of control flow. However, performing data flow analysis may be inefficient if loops are not treated specially. For this reason, we refer to special representation of flow graph, called dominator tree, to speed up a convergence of iterative data-flow analysis.

Definition 4:

A node *d* of flow graph dominates node *n*, denoted as *d dom n*, if every path from the entry node of flow graph to *n* passes node *d*. Each node dominates itself that is *d dom d*.

Dominator information is typically represented by dominator tree, where entry node is the root and each node *d* dominates its descendants in the tree. The dominators are computed for every node *n* in a flow graph, based on the principle that if $p_1,p_2,\ldots,p_k$ are all the predecessors of *n*, and $d \neq n$, then *d dom n* if and only if *d dom* p_i. The dominant information may be calculated using the following forward data flow equations:

Dom$_{In}$
```
context ActionElement::domIn : Sequence (action:ActionElement)
derive: self.pred.outgoindDoms
```
(2.9)

Dom$_{Out}$
```
context ActionElement::domOut: Sequence (action::ActionElement)
derive:
if self.succ -> isEmpty() then
 Sequence {self}
else
 self.incomingDoms ->intersection(self)
endif
```
(2.10)

This suggest relationship between the ordered Dom set and an auxiliary data structure called dominator tree, which may be computed using simple iterative algorithm proposed by Cooper *et al*. (Cooper *et al*. 2001).

2.5.2. Interprocedural Analysis

Interprocedural analysis is challenging activity because the behaviour of each procedure is dependent upon the context in which it is called (Aho *et al*. 2006)**.** There are several approaches for performing intraprocedural analysis:

- *Context insensitive*: easy to implement but extremely inaccurate, as calling is interpreted using goto instructions.
- *Context sensitive*:
  - *Call-strings*: calling context is defined by the contents of the entire call stack.
  - *Cloning-based*: clone procedure body for each context of interest.
  - *Summary-based*: each procedure is represented by a concise description that encapsulates some observable behaviour of procedure.

The summary-based approach avoids re-analysing a procedure's body at very call site that may invoke the procedure. Therefore in our method we select this approach for dealing with interprocedural problems. The approach involves two general parts:

- computing summary flow information for each procedure;
- using this information as the flow function for a call block.

One of inter-procedural forms is a call graph. The call graph of a program is a set of nodes and edges such that (Aho *et al*. 2006):

- There is one node for each procedure in the program.
- There is one node for each call site, that is, a place in the program where a procedure is invoked.
- If call site *c* may call procedure *p*, then there is an edge from the node for *c* to the node for *p*.

In general, during inter-procedural analysis we are concerning in the following side effects produced by call-return mechanism:

- Are the variables passed as parameters by value being modified within the called procedure?
- Are the variables passed as parameters by value being used before their modification?

Let we define side effect properties within the KDM *CallableUnit* which represents a procedure. *CallableUnit*.mustKillVars is a set of variables that are being defined within a procedure, *CallableUnit*.mayKillVars is a set of variables that may be defined along some path of execution of a procedure, *CallableUnit*.mustUseVars is a set of variables that are along some path during procedure execution, and *CallableUnit*.mayUseVars is a set of variables that may be used along some path of execution of a procedure. Clearly, *must* influence anal-

ysis of all paths (i.e. operation union) whereas *may* determines analysis of some paths (i.e. operation intersection).

For the computing flow sensitive side effects of *CallableUnit*.mustKillVars the following data flow equations are constructed:

LiveInter$_{In}$ (2.11)

```
context ActionElement::liveInterInVars : Sequence (code::DataElement)
derive:
if self.succ -> isEmpty() then
        self.getCallableUnit().getGlobalVariables()
else
        self.pred -> iterate(a:action::ActionElement;
mustKill:Sequence(code::DataElement)=Sequence{} | mustKill -> inter-
section(a.liveInterInVars))
endif
```

LiveInter$_{Out}$ (2.12)

```
context ActionElement::liveInterOutVars : Sequence
(code::DataElement)
derive:
if self.kind='Call' then
        self.liveInterInVars -> union (self.getOutbound() -> select(o |
o.oclIsTypeOf(action::Calls)).oclAsType(action::Calls).to.mustKillVars)
else
        self.liveInterInVars -> union (self.genVars)
endif
```

MustKill (2.13)

```
context CallableUnit::mustKillVars : Sequence (code::DataElement)
derive: self.end.liveInterOutVars
```

Where *getGlobalVariables()* is the query operation which returns both the global and local variables. It is easy to observe that for computing *MayKill* property, the meet operation for incoming variables is union rather than intersection (2.11), and the initial values are empty sets rather than all variables.

Computing of *MayUse* information is very similar to variable liveness computation, except that in this case a call statement is being considered.

LiveInter$_{In}$ (2.14)

```
context ActionElement::liveInterInVars : Sequence
(code::DataElement)
derive:
let calledProc : code::CallableUnit =
if self.kind = 'Call' then self.getOutbound() -> select(o |
o.oclIsTypeOf(action::Calls)) ->first().to else OclUndefined endif
if self.kind='Call' then
        (self.liveInterOutVars – (calledProc.mustKillVars)) -> inter-
section(calledProc.mayUseVars))
else
        (self.liveInterOutVars - self.killVars) -> union(self.genVars)
endif
```

LiveInter$_{Out}$ (2.15)

```
context ActionElement::liveInterOutVars : Sequence
(code::DataElement)
derive:
```

```
            if self = self.getOwnerCallableUnit().end then
                    OrderedSet{}
            else
                    self.succ.liveInterInVars
            endif
MayUse      context CallableUnit::mayUseVars : Sequence (code::DataElement)
            derive:
            self.entryFlow.to.oclAsType(action::ActionElement).liveInterOutVars
```
(2.16)

For a call statement, the variables in *mustKillVars* set of the callee cease to be live whereas the variables in *mayUseVars* set of the callee become live. In case of *mustUseVars*, we should change the operation from *intersection* to *union* thus involving "all-paths" analysis and for a call statement check for *mayKill-Vars* instead of *mustKillVars*.

2.6. Conclusions of the Second Chapter

In this chapter we analysed the representation and analysis of knowledge about existing software systems using the Knowledge Discovery Meta-model (KDM). We have observed that KDM lowers the effort required for definition of existing software representation, and that the knowledge extraction and analysis techniques may be employed for automatic or semiautomatic obtainment of particular KDM models. Moreover, higher-level representations may be incorporated to facilitate the analysis of existing software systems.

From this study we can conclude, that:

1. Although the KDM refers to as "knowledge discovery" meta-model, it is restricted to represent the knowledge only from the software assets, and there is no standard way to include representation of knowledge from other resources, such as the software specification or other kind of documentation.
2. The KDM extension mechanism limits the expressiveness of the representation as it does not support any kind of relationships between stereotypes used to extend meta-model elements, thus aggravating introduction of precise domain or platform specific representations.
3. The specification of KDM does not provide formally defined well-formedness constraints on meta-model elements; therefore, discovered models cannot be validated for conformance to the KDM. For this reason, a partial set of well-formedness rules for Code and Data models has been provided.

As the KDM has been selected to define the intermediate representation (IR) of existing software system, we formulated the principles of representing source code and database schema within KDM models. A number of well-

formedness constrains were defined in order enable validation of produced models. For the purpose of knowledge discovery within the source code, we revised data flow analysis techniques and defined them in the context of the KDM. For this reason we extended KDM model elements with properties that derive information computed using of data flow equations. Although we revised fundamental data flow analysis techniques that we use in our method, we believe that their application for the KDM will form a foundation for the further research.

3

A Method for Business Knowledge Extraction from Existing Software Systems

3.1. Introduction

In this chapter we present a method for Business Knowledge Extraction from existing Software Systems – BKESS (Normantas, Vasilecas 2011; Normantas, Vasilecas 2012a; Normantas, Vasilecas 2012b). This method is based on the reverse engineering process that involves several iterative activities and several roles responsible for performing these activities. The high level view of this process is presented in the Fig. 3.1.

A system analyst is responsible for preparing the different architectural views from various artefacts of a software system. The source code, resource configurations, and database schema definitions are parsed to abstract syntax trees (AST) and transformed to intermediate representations within KDM models. These models are further processed by applying the data flow analysis techniques to establish various dependencies between different components of the system.

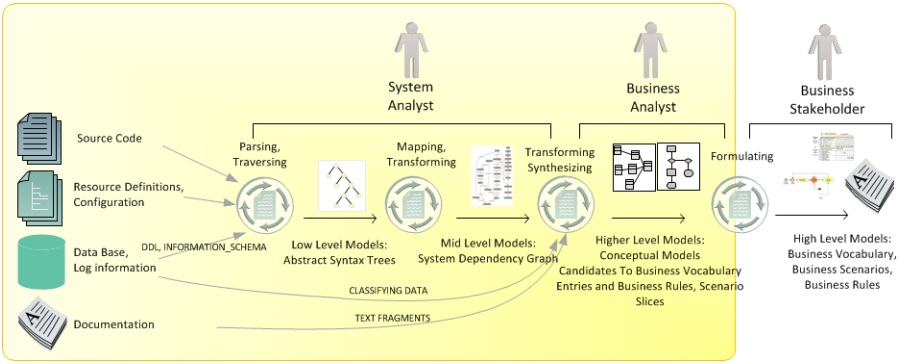

Fig. 3.1. Business knowledge extraction process

Once the dependencies are established, intermediate models may be further refined in order to identify candidates for the business vocabulary (i.e. terms and facts) entries, business scenario slices (i.e. use-cases), and business rules (i.e. conditions that influence a certain direction or termination of the flow, or computation of certain values).

The business analyst reviews and validates the extracted knowledge and, using automated transformations, formulates it in a form that is negotiable with the business stakeholder (vocabulary entries, rule statements, decision tables, process models).

It is important to notice, that the scope of this work is the extraction of candidate elements to business vocabulary, rules, and scenarios. This scope is depicted as rectangular zone within the process representation in the figure above. In the following section, we provide a detailed discussion on the extraction process. The remaining part of the extraction process we leave for the further research.

3.2. Business Knowledge Extraction Process

As it was already introduced, the business knowledge extraction process obtains intermediate representation of different aspects of the software system into the KDM models and abstracts business logic from these models. The process consists of the three main stages that include various automatic, semi-automatic, or manual activities, depicted with corresponding stereotypes as shown in the Fig. 3.2.

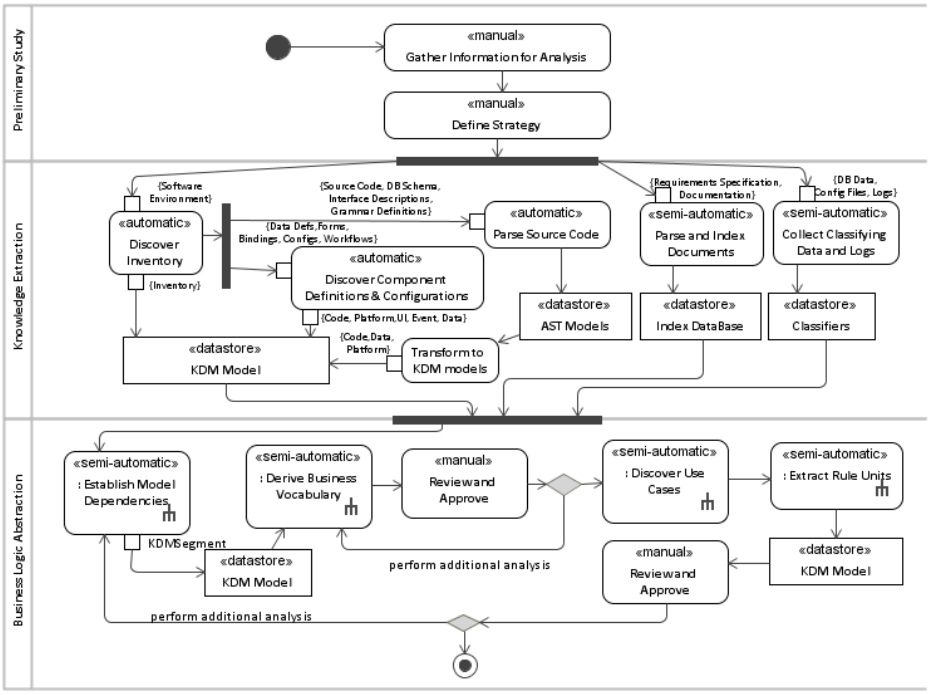

Fig. 3.2. The process of business knowledge extraction from existing software systems

By “automatic”, we mean that the activity requires minimal user interaction (i.e. setup directory for discovering software environment or start processing of the activity). By “semi-automatic”, we mean that the activity is performed by a user with aid of supporting tools (i.e. some steps are automatically performed, e.g. a computation of derived properties). By “manual”, we mean that activity is performed by a user without any automation (e.g. manually create document or model).

3.2.1. Preliminary Study

The first phase of business rules discovery process involves preliminary study of the existing software system and it consists of the following two steps: gathering initial information and defining a strategy for knowledge extraction and representation within KDM.

During the first step, the present architecture of the software system is reviewed, the architectural components are identified, and the high-level depend-

encies between them are established. A set of KDM extension families for specifying domain or platform specific concepts should be considered and manually created within this step.

Based on the acquired initial knowledge about the software system, a strategy for obtaining the representation within KDM is defined. The strategy establishes the list of software artefacts that will be processed, the ways they will be processed, and the time expected for delivery of each representation.

3.2.2. Knowledge Extraction

The knowledge extraction phase involves several steps whose purpose is to build knowledge base used as the main source for business logic abstraction. The knowledge base consists of a set of KDM models that represent software system (referred thereafter as KDM representation), the data base of indexed software documentation, and the data base of classifiers (i.e. lookup table values) and system log information.

As it was already introduced, the KDM defines representation of the software system at several layers of abstraction: Infrastructure, Program elements, Runtime resources, and Conceptual layers. Within each layer, one or more architectural views on certain aspects of software system are being produced.

Discovering Software Inventory

The Infrastructure layer involves the Inventory model that is intended to represent software system's physical artefacts: containers, folders, source files, resource definitions, etc. The Inventory model (Fig. 3.4, left side) serves as a bridge between physical artefacts and their representation within higher level of abstraction. Therefore it is naturally to discover this model in the early steps, and we employ file system scanning or querying version control system to obtain required information. It is important to note that discovered Inventory model elements are supplemented with audit information by creating specific attributes (i.e. user, created, modified) in order to allow tracking changes of software artefacts and to ensure up-to-date software representation.

Creating Code Models

The Program elements layer involves a set of Code models that represent the structure and behaviour of software system implemented within the source code, the platform specific application programming interfaces (API), and the programming language or platform resources specific data types.

Code models representing software platform API may be created automatically by discovering API definitions (if it is allowed and possible) or created manually from the software platform API documentation.

Although the representation of the composition of source code might be transformed directly from abstract syntax trees (AST), the behaviour representation requires numerous AST traversals to discover the data and control flow. For this reason, we follow the principles for representing data and control flow that we have defined within the previous chapter.

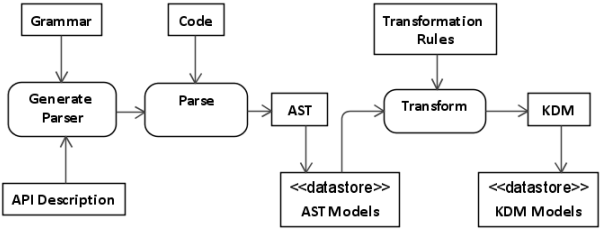

Fig. 3.3. Process for obtaining the source code representation within KDM Code model

The AST is generated by the parser which is built from the language grammar definition (Fig. 3.3). The grammar is supplemented with the software API definition to facilitate identification of API usage and to allow establishing dependencies between source code and API models. As it was already introduced in the previous chapter, transformation rules are defined according to the *MicroKDM* semantics, allowing obtainment of the fine-grained view of source code.

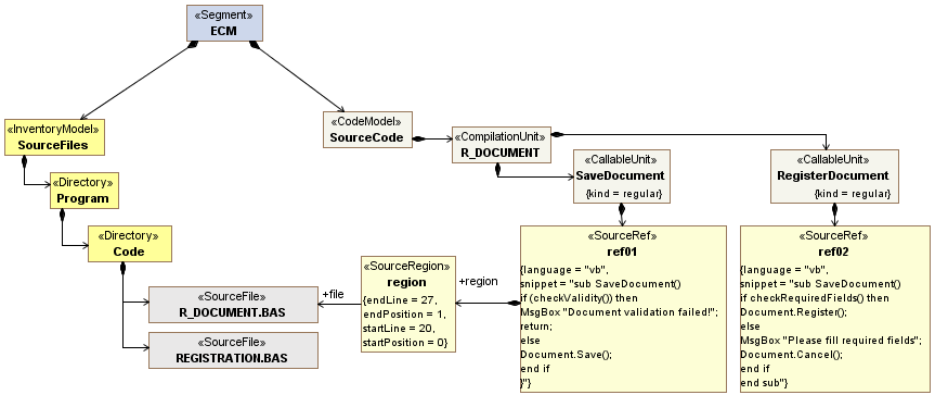

Fig. 3.4. An example presenting relationship between different levels of abstraction that allows to preserve traceability between model elements and the source code that they represent

In order to preserve traceability between software artefacts and their representation within the KDM, model elements are supplemented with *SourceRef* and *SourceRegion* elements. These elements allow to capture the snippet of corresponding content (e.g. code or xml snippet) as well as to indicate the precise location of this snippet within the content of artefact (see Fig. 3.4). This information is automatically obtained from the AST generated by the parser.

Creating Resource Models

The Runtime Resources layer involves several kinds of models to represent various system resources: UI, Data, Platform, and Events models. Depending on the software system, there may be different types of resource definitions, including user interface, data, workflow, system job, task, or component definitions.

During runtime, these definitions are processed by the software platform to create runtime objects (e.g. form instances) that might be manipulated by the application code using software API. The content of resource definition files is structured according to particular schema definition. However, the definition of schema not always might be available; for this purpose it can be reversed automatically from the content (Necasky 2009) or defined manually by considering only relevant parts. Then, according to the predefined set of mapping rules between the schema elements and elements of particular KDM model, the content of resource definitions is parsed and corresponding KDM representation is created.

KDM runtime resource models are augmented with representation of content of configuration files. While discovering their content, it is possible to discover platform resources other than previously identified. It should be noted that such information does not necessary mean that they are actual, because the configuration data may be obsolete, written by resources that are changed or removed in time.

Collecting Available Documentation

The creation of database of software documentation is built in several steps. Digital documents are parsed using specialized document parsing libraries to retrieve trees representing logical structure of document content. The logical structure of the document is defined using the meta-model represented in the Fig. 3.5.

Depending on the type of document, its logical structure may be retrieved directly from the structure definition (e.g. TOC in word documents), bookmarks that links to different pages within a PDF, or considering a set of rules established regarding the properties of physical content (i.e. blocks) of document. However, in order to be indexed by indexing engine, this structure must be flattened. For this reason, we analyse extracted structure, identify the depth of the tree, and define appropriate fields of meta-data schema (Fig. 3.6).

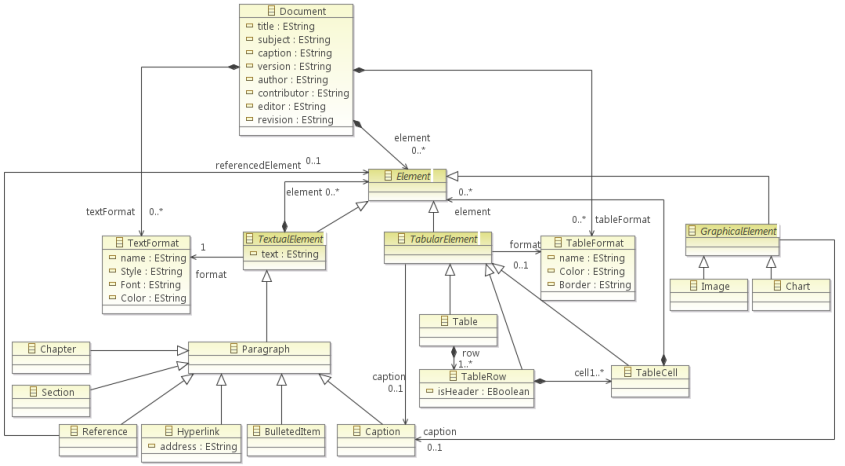

Fig. 3.5. A meta-model for representing the structure of document

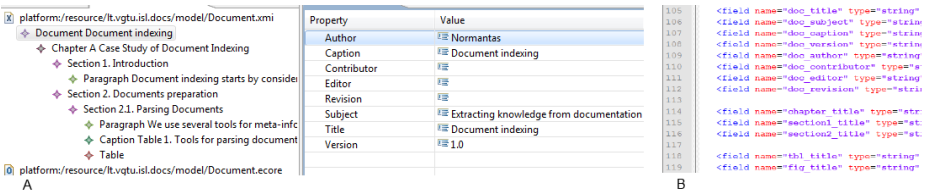

Fig. 3.6. Creating meta-data schema for documentation indexing: (A) presents model fragment extracted from document, (B) presents corresponding fields definition fragment

Following this approach, each "leaf" of document tree, e.g. a paragraph or bulleted list, is indexed as a separate document with meta-data that corresponds to the schema definition. Such indexing allows supplementing vocabulary entries with the corresponding definitions.

Collecting Data

Finally, a database of classifiers and log information is built by reviewing known lookup tables, files containing classifying data definitions, log files or tables. For each resource, a local copy of data is created and stored in the database to be available for further analysis. The data in this database is later used to define base facts from the established business terms.

3.2.3. Business Logic Abstraction

Having extracted all the available and relevant knowledge about the software system into the KDM representation, the next phase of the recovery process involves activities to separate KDM model parts that represent business logic implementation from the infrastructure related ones. For this reason, we first of all establish dependencies between model elements. Then, we derive initial set of term and fact units, and validate it with maintainer of the system. We follow the SBVR standard (OMG 2008) as guidance to classify the business rules and to enable their formal definition. We further apply static source code analysis techniques to refine this set and to identify the implementation logic of business rules, and to extract particular business scenarios.

Establishing Model Dependencies

The aim of dependencies establishment step is to obtain an exhaustive representation of various dependencies between different software artefacts. It includes both horizontal (i.e. within the same level of abstraction) and vertical (i.e. between different levels of abstraction) dependencies.

For this purpose, the step includes the following sub-steps: create system dependence graph (SDG), establish bindings, and create code relationships with various resources, as shown in the figure below.

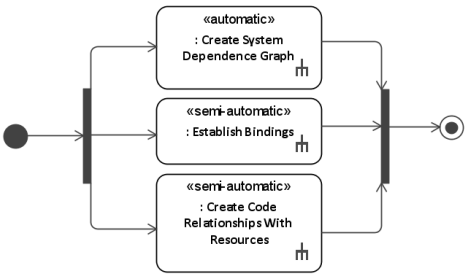

Fig. 3.7. Establish model dependencies activity

Creating system dependence graph (SDG) involves several iterative activities (Fig. 3.8). For each *CallableUnit* that represents a procedure or function, it creates a control flow graph (CFG), which is supplemented with def-use relationships. Then it computes Summary flow information and adds CFG to SDG. The *Def-Use* and *Use-Def* chains and *CallableUnit* Summary information are calculated according to the equations specified in the Chapter 2. The calculation is performed using iterative Worklist algorithm (Khedker *et al*. 2009), until the sets of information under analysis converge within certain iteration.

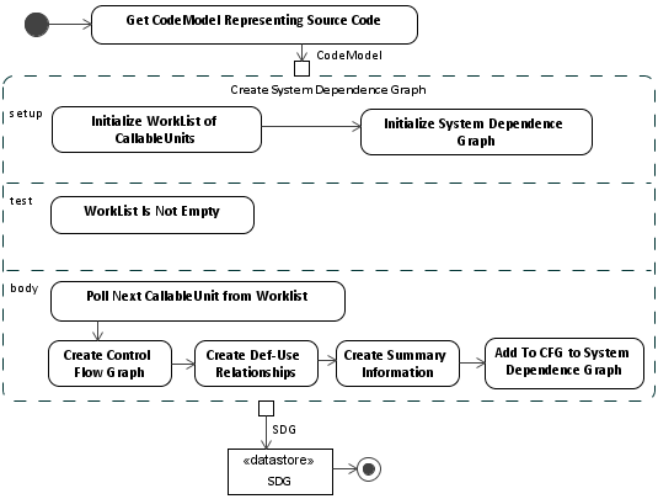

Fig. 3.8. High-level view of process for creating System Dependence Graph

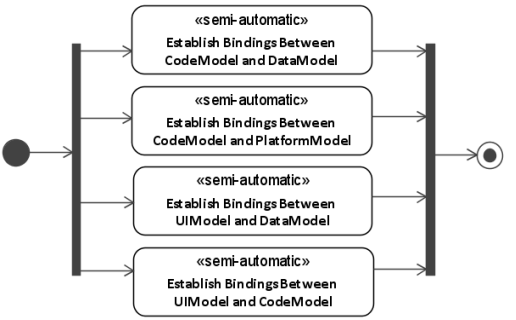

Fig. 3.9. Establishing bindings between structural elements of software systems

Establishing bindings between various structural elements of software system representation is performed in several activities as shown in Fig. 3.9. As there typically are multiple run-time resource specifications (e.g. user interface definitions, data definitions, platform configurations), these activities include analysis of their representation. This activity requires a knowledge from system analyst about the infrastructure (or platform) of software system. Establishing bindings activity results in KDM representation supplemented with binding relationships (Fig. 3.11) that are important in extracting vocabulary, as it will be discussed in the next section of this chapter.

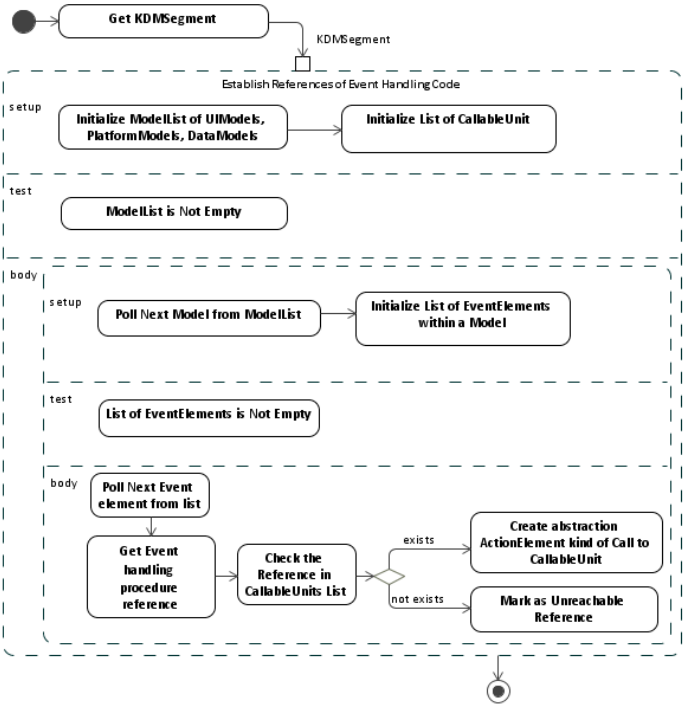

Fig. 3.10. High-level view of process for establishing run-time resource references with event handling code

References between run-time resources and source code are established considering the following various kinds of events provided by the resources:

- Source code handling software *platform events* (e.g. periodical event, application starting/stopping);
- Source code handling *user interface events* (e.g. form event or form control event);
- Source code handling *object instantiation or access events* (e.g. create, update, delete);
- Source code handling *events produced due to access to particular software platform functionality* (e.g. user login, user management event);
- Source code handling *events of workflow activities and transitions between them* (e.g. on enter to activity or exit from it, on transition);
- Source code accessing runtime resources that produce particular kind of event (source code that sends signal to runtime resource invokes execution of source code that handles the event of signal receive, e.g. setting form field value

programmatically invokes execution of source code that handles *OnSetFieldText* event).

Having established dependences and built a SDG, source code analysis techniques may be applied to identify the business logic implementing source code and related resources and represent it within the KDM Conceptual model. The KDM enables mapping with the Semantics of Business Vocabulary and Business Rules (SBVR) specification by providing the following Conceptual model elements – *TermUnit*, *FactUnit* and *RuleUnit*. The *TermUnit* element represents an implementation of the SBVR noun concept; the *FactUnit* represents an implementation of the SBVR verb concept (or fact type), which depicts a characteristic of the noun concept or a kind of relationship between two or more noun concepts; the *RuleUnit* represents an implementation of the SBVR business rule. Moreover, the KDM provides "concept" classes to represent the overall behaviour (*BehaviorUnit*) of the software system or the particular use case (*ScenarioUnit*) of the software's behaviour.

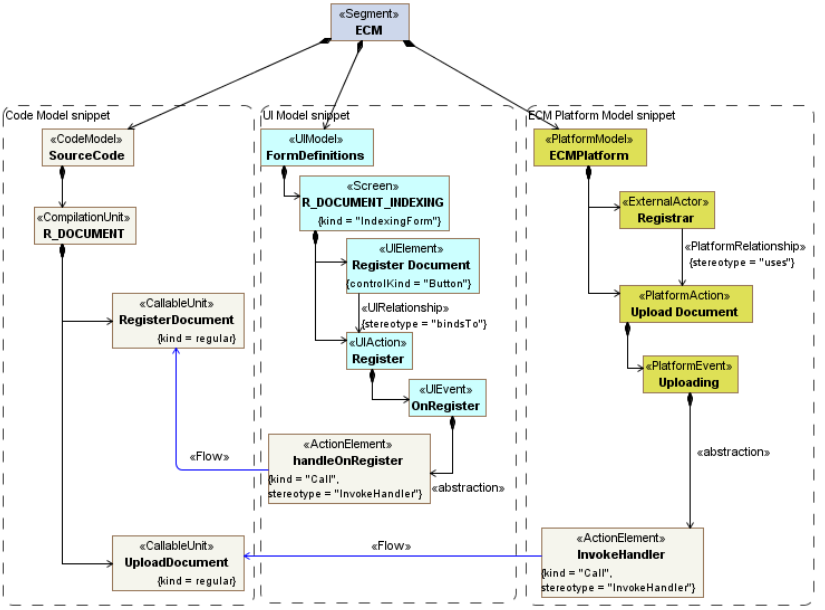

Fig. 3.11. Establishing relationships between different layers of abstraction of representation

The main goal of the business knowledge extraction phase is to create a conceptual model that would include as much as possible business domain rele-

vant information and that would facilitate development of other, more abstract, kinds of knowledge representation.

Deriving Business Vocabulary

Business vocabulary derivation consists of two main steps. In the first step, a set of candidates to noun concepts is derived from the elements of the KDM representation of run-time resources and software source code. In the second step, a set of candidates to fact types is derived considering the relationships between the candidates to noun concepts.

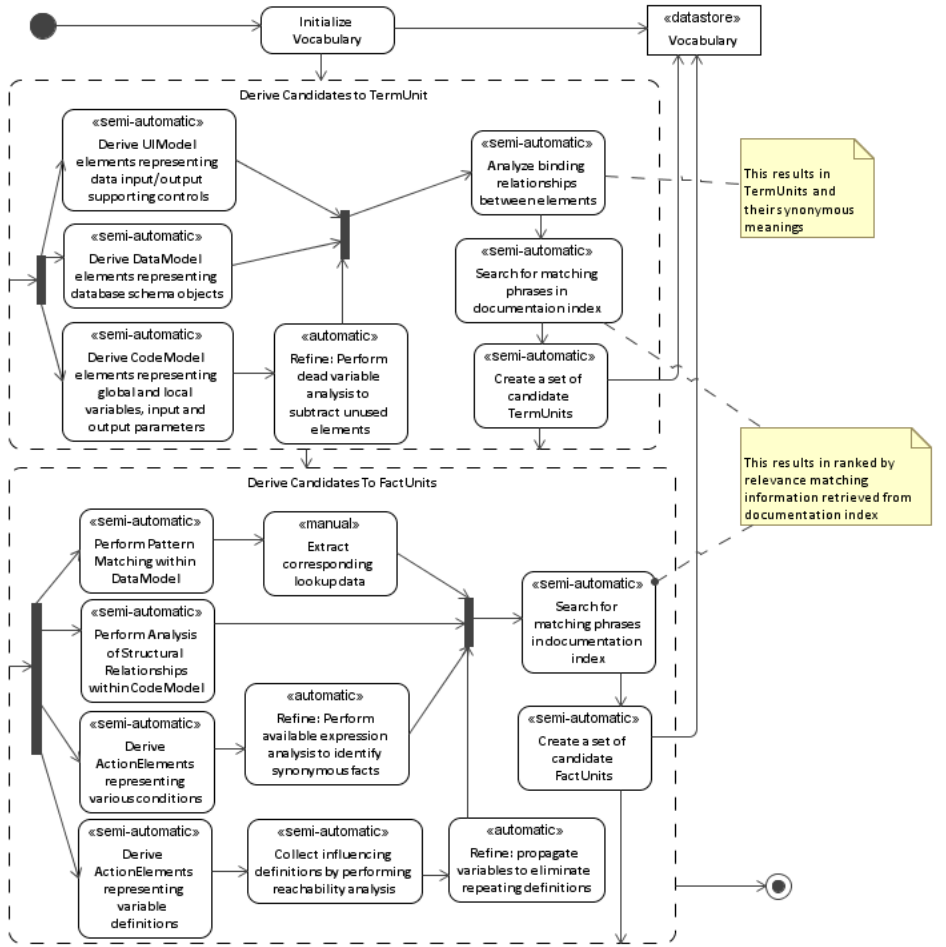

Fig. 3.12. A process of deriving business vocabulary

Deriving Candidates to Noun Concepts

In order to derive candidates to noun concepts we consider various KDM representation elements that define the data that software operates on. There are many ways how data may be entered, stored, and represented within the software system and different ways how it can be organized during various intermediate computations. For this reason, we perform analysis on run-time resources model and source code model.

We first of all analyse the user interface (UI) models. The UI models reflects data input and output functionality available for a user of the system. The UI model elements represent software dialogs, forms and reports that contain different controls labelled in the form understandable for the user. We consider these labels as primary source for identifying noun concepts. As the next source for discovering candidates to noun concepts, we refer to Data models that represent organization of data structure either in relational (relation schemas, tables, views, and columns), or in the hierarchical form (e.g. XML). From the code models we consider data elements that represent global and local variables as well as input and output parameters of procedures or methods.

The data elements from the code models are further refined by applying dead variable analysis and constant propagation thus removing unused and repeatable elements. After having derived initial set of candidates to business terms we analyse bindings between these elements in order to identify implementation of the same concept within different software artefacts. Having initial set of candidate variables, we tokenize identifiers into corresponding noun concepts (split variable names if they are complex, e.g. isDocReceivable → 'is doc receivable', or RECEIVABLE_DOCUMENT → 'receivable document').

Then we search for matching phrases in documentation repository and create definitions of candidate terms from relevant text fragments. These definitions are stored as *TaggedValue* elements corresponding to *TagDefinition* "dictionary basis" defined within the stereotype *VocabularyEntry*, as it will be discussed in the Section 3.2.4. If the candidate has more than one matching fragment, then we create multiple definitions ordered in regards with the relevance rank produced by the search engine.

Deriving Candidates to Fact Types

Derivation of candidates to fact types is based on analysing relationships from data structures implemented in the database or program code of the software system. As a basis for identifying and categorizing different kinds of fact types, we refer to the SBVR categorization scheme:

- The *Characteristic Fact Type* is a fact type that has exactly one role (i.e. unary fact type).

- The *Associative Fact Type* relates two or more concepts in a non-hierarchical way. This category involves the following subcategories:
  - The *is-property-of fact type* relates two noun concepts where the first concept defines essential quality of the second concept.
- The *Partitive Fact Type* defines the whole-part relationship between instances of two given concepts (part and whole) such that each instance of a part is in the composition of the given whole.
- The *Specialization Fact Type* determines that an instance of a specific concept is also an instance of a more general concept.
- The *Assortment Fact Type* defines that an instance of an individual concept is an instance of a general concept.

The primary sources for discovering candidates to fact types are Data and Code models. The following table summarizes the main derivation principles of different kinds of fact types.

Table 3.1. Fact types derivation principles

Source	Derivation principles
Characteristic fact type	
DataModel	• Analysis of table columns that are type of Boolean (i.e. corresponding SQL BIT data type).
CodeModel	• Analysis of class members or Global variables that are type of Boolean.
Association fact type	
DataModel	• Analysis of relational tables that consist of primary key and foreign keys (association tables), such that no other
CodeModel	• Analysis of class members that are not of primitive data types. • Analysis of predicate expressions used as the conditions in control statements (e.g. if/while/switch condition). • Analysis of SQL DML query expressions that form database cursors. • Analysis of SQL DML query expressions used in database views, functions or procedures.
Is-property-of fact type	
DataModel	• Analysis of table columns that are of primitive types, such as text, date, numerical types, etc.
CodeModel	• Analysis of class members or Global variables that are of primitive types, such as string, date, integer, float, etc.
Participation fact type	
UIModel	• Analysis of master-detail structures within the form or report.
DataModel	• Analysis of foreign key constraints with “on cascade delete” option. The partitive fact type defines the whole-part relationship between

Table 3.1. (Continued)

Source	Derivation principles

	instances of two given types (part and whole), such that each instance of a part is in the composition of the given whole. This kind of relationship may be implemented in the databases either using the nested table data types (as supported by Oracle DBMS), or by controlling the existence of a "part" instance, i.e. cascading foreign key constrains. For example:
	`CREATE TABLE Organization (OrganizationID INT PRIMARY KEY, …)`
	`CREATE TABLE Department (DeptID INT PRIMARY KEY, OrganizationID INT REFERENCES Organization(OrganizationID) ON DELETE CASCADE …)`

CodeModel	• Analysis of nested (inner) class declaration (an instance of parent class can exists on its own but an instance of nested class cannot). • Analysis of list data structures consisting of elements that are of custom type (i.e. custom classes), and instances that are stored in the list are created and destroyed only in the context of the owner class instance.
Specialization fact type	
DataModel	• Analysis of foreign key constraints between tables. • The specialization fact type determines that an instance of a specific concept is also an instance of a more general concept. This kind of fact type, depending on how it is implemented in the software system, may have further specified by the following properties: incomplete/complete, and disjoin/overlapping. In an *incomplete specialization*, also called a *partial specialization*, only certain instances of the general type are specialized (i.e. have unique attributes). In contrast, a complete specialization indicates that all instances of the general type must be specialized. In a *disjoint specialization*, also called an *exclusive specialization*, an instance of general type may be specialized by only one subtype, while an overlapping specialization allows an instance of general type to be specialized by more than one subtype. From the implementation point of view, specialization is typically realized by using either certain inheritance mechanism or specific properties (fields, columns) that assign an instance to a certain category. While the latter may be identified by analysing referencing constraints and by applying heuristics on name of referenced lookup tables (e.g. testing for keywords "type" or "kind"), the former may be derived by identifying the referencing constraints that references primary key columns from base and referenced tables (the cardinality of relationship between base and child table is one-to-one). For example, one the simplest way of implementing specialization relationship in databases may be achieved by the following pattern:

Table 3.1. (Continued)

Source	Derivation principles

	`CREATE TABLE Person(PersonID INT PRIMARY KEY, SSN …)`
	`CREATE TABLE Student(PersonID INT PRIMARY KEY REFERENCES Person(PersonID),…)`
	`CREATE TABLE Lecturer(PersonID INT PRIMARY KEY REFERENCES Person(PersonID),…)`
	This pattern uses primary keys as foreign keys thus ensuring that an instance of subtype must be an instance of the general type. However, from such implementation we do not know the actual type of an instance of the Person. An instance of type Person may be either Student, or Lecturer, both of them, or none of these subtypes. Therefore, this kind of implementation possesses the properties of *incompleteness* (a Person may be none of the subtypes) and *overlapping* (a Person may be of both these types). It is important to note, that unless we do not identify explicitly defined disjoint we will treat specialization as overlapping.
	Consider another example of pattern for specialization implementation on the same relational tables:
	`CREATE TABLE Person(`
	`PersonID INT PRIMARY KEY,`
	`Type CHAR(1) NOT NULL CHECK(Type IN('S','L')),`
	`UNIQUE(PersonID,Type))`
	`CREATE TABLE Student(`
	`PersonID INT PRIMARY KEY,`
	`Type AS CAST('S' AS CHAR(1)) PERSISTED,`
	`FOREIGN KEY(PersonID,Type) REFERENCES Person(PersonID,Type))`
	`CREATE TABLE Lecturer(`
	`PersonID INT PRIMARY KEY,`
	`Type AS CAST('L' AS CHAR(1)) PERSISTED,`
	`FOREIGN KEY(PersonID,Type) REFERENCES Person(PersonID,Type))`
	The following pattern ensures that an instance of Person type must be specialized either by a Student or Lecturer subtypes. Such implementation ensures both *completeness* and *disjointness* properties of the specialization between the type Person and its subtypes. It is easy to notice, that by releasing the column Person.Type from the mandatory constraint (i.e. `Type CHAR(1) NOT NULL -> Type CHAR(1) NULL`), the specialization will allow incomplete set of instances of general type Person.

CodeModel	• Analysis of inheritance relationships between classes. For example, `class Student extends Person {}`
Assortment fact type	

Table 3.1. (Continued)

Source	Derivation principles

DataModel	• Analysis of column data values domain specifications.
	• Analysis of check constraint specifications.
	The assortment fact type defines that an instance of an individual concept is an instance of a general concept (i.e. Lithuania is a Baltic State). Typically, the assortment is implemented as a set of categorizing table records (i.e. lookup table values); therefore, identification of this kind of fact type and retrieval of related values (individual concepts) requires manual work by a system analyst. Another case of assortment implementation is Check constraints that enforce certain column values to be in a specific collection of values defined within the IN predicate. For example:
	`CREATE TABLE Department(DeptId … Region VARCHAR(20) CHECK (Region IN ('Baltic states','Benelux','Nordic','British Isles',…)))`
	Or the check constraint over a set of values from another table:
	`CREATE TABLE Department(DeptId … Region VARCHAR(20) CHECK (Region IN (SELECT Name FROM Region)))`

CodeModel	• Analysis of enumeration data structures.
	For example,
	`enum Region{"Baltic states","Benelux","Nordic","British Isles",...}`
	• Analysis of list data structures containing elements that are of primitive data types.
	For example,
	`List<String> regions = new List<String>(){"Baltic states","Benelux","Nordic","British Isles",...};`

Although the is-property-of and characteristic fact types may be derived in relatively straightforward way, the other kinds, as we can see, requires additional analysis of data structures that may be defined either explicitly over specific definitional constructs, or implicitly by data manipulation constructs. As it would be impossible to cover all the cases of implementing different fact types within the software system, the given table formulates summarizes basic ideas used in our study for discovering different fact types within the KDM representation.

Discovering Use Cases

The main goals of use cases discovery step is to identify functionality provided by the software system, identify external actors who initiates uses and has access to this functionality and explore possible flows of control covered by the use cases (i.e. understand their implementation) (Fig. 3.13).

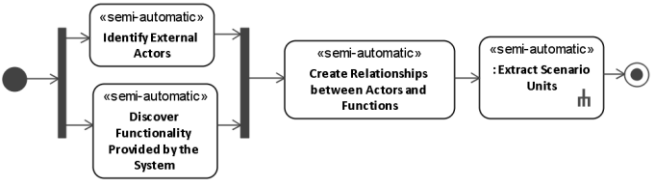

Fig. 3.13. High level view of use case discovery

The primary source for discovering this information is the platform model representing software system configuration, which defines the system actions provided by the main application dialog (menu items, toolbars, etc.), user profile definitions and a mapping between the actions and profiles. Moreover, configurations include definitions of periodical tasks performed internally by the system. The secondary source for discovering functionality provided by the system is the code model representing a set of exposed interfaces. External systems that use these interfaces must be identified by analysing system architecture specification.

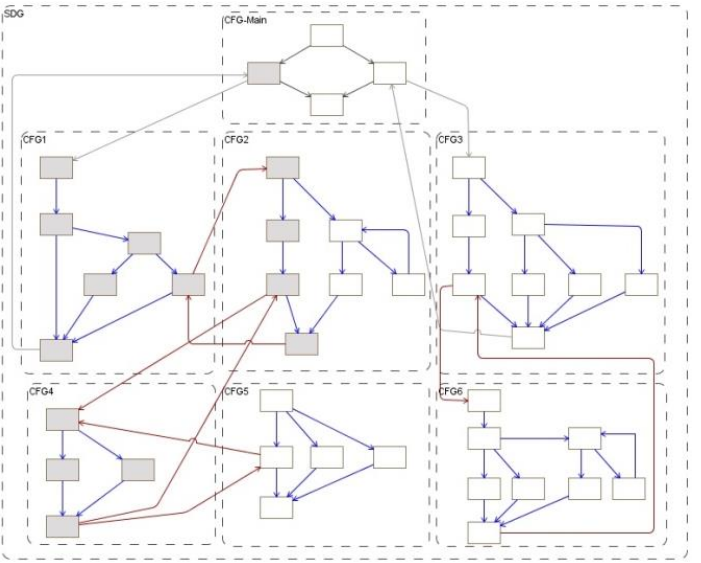

Fig. 3.14. System dependence graph: abstracted view of control flows between control flow graphs. CFG-Main is synthesized from resource events: a grey relationships present call-return between event and it handling procedure; a blue relationship denotes internal control flow; a red relationship presents call-return between procedures. Grey colored path represents extracted particular *ScenarioUnit*

ScenarioUnits are derived to represent behavioural flow paths from platform or user interface events through the PDG. They are gathered by traversing SDG in forward direction along the control flow and call edges and considering whether the nodes (i.e. *ActionElements*) of the graph relate with vocabulary entries identified in previous step. The Fig. 3.15 provides an algorithm for extraction of scenario units. The following figure (Fig. 3.14) illustrates high level view of SDG and extracted scenarios by applying this algorithm.

As we can see in the given example, extracted scenario units represent slices of SDG. We can read this example as: CFG-Main – represents application dialog; the first block (grey) within the CFG-main represents opened form; the flow from first block to the first block of CFG1 represents invocation of form event handling source code.

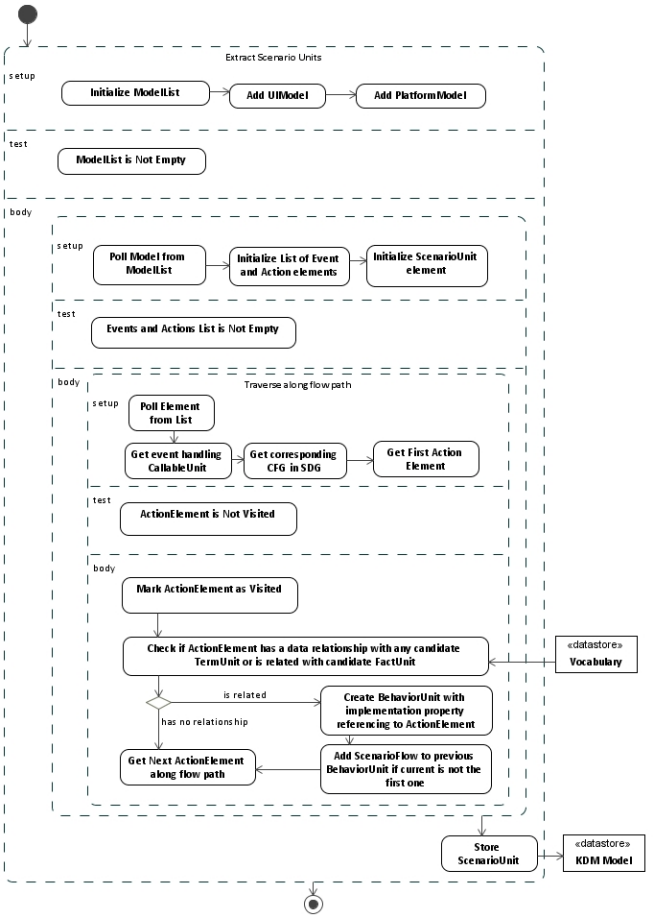

Fig. 3.15. A process of extraction of scenario units

Extracting Candidates to Business Rules

There are two general kinds of business rules: structural and operative. According to Ross (Ross 2003), structural business rules are about how the business organizes its basic knowledge, while operative business rules are about how the business operates its activities. Determining particular kind of business rule from the source code of software system may not be trivial task as it can be seen from the first point of view, because it depends on the context where and when the rule is being evaluated and on the facts that rule depends on evaluation. One of the primary distinctions of these kinds states that structural business rules cannot be violated by the business, while operative can be.

Structural rules by their nature are definitional. They can define constraints on population of a type or define how an instance of a type may be derived or calculated. From the implementation point of view, structural business rules may be ensured both by constraining data values declaratively (e.g. properties of form fields or constraints on columns of relational tables) and by terminating the behavioural flow of control when certain conditions that represents the business rule are not met (e.g. if form field value is empty then raise an alert and return back to the form). The following statement illustrates this kind of business rule: *It is obligatory that each order has at least one order item.*

It can be easily observed that there are at least two ways to ensure this rule: one is by checking on order submit event whether there were inserted order items in order form; other is by setting up mandatory constraint (i.e. NOT NULL) on the column ITEM_ID referencing order item in the table ORDER_ITEM (ORDER_ID NOT NULL, ITEM_ID NOT NULL).

In order to formulate principles for discovering structural business rules, we distinguish them in several subcategories. The following table summarizes these subcategories and explains main principles of discovering candidates to rules from different sources.

Table 3.2. Structural rules discovery principles

Source	***Discovery principles***
Mandatory constraints	
UIModel	Discover constraints that require form field data to be provided before submitting the form.
DataModel	Discover NOT NULL constraints on table columns.
CodeModel	Discover control structures that terminate the flow of control when certain value is not provided. Discover control structures that terminate the flow of control

Table 3.2. (Continued)

Source	*Discovery principles*
	when certain conditions are not met.
Uniqueness constraints	
DataModel	Discover UNIQUE constraints implemented in the database schema.
	Discover UNIQUE INDEX implemented in the database schema.
CodeModel	Discover control structures that terminate the flow of control when there are already created the same instance in a population (e.g. EXISTS).
Value constraints	
UIModel	Discover constraints that require form field data to be in particular range before submitting the form.
DataModel	Discover database constraints that require column values to be in a particular range.
CodeModel	Discover control structures that terminate the flow of control when certain variable is not in specific range.
Derivation rules	
DataModel	Discover computed column specifications and default value constraints.
CodeModel	Discover a path along use-def chains within the SDG (backward traverse).
Calculation rules	
CodeModel	Analyse derivations: each action element within discovered path is of arithmetical kind.

As we can see, some categories of business rules require analysis of elements representing different kinds of constraints on data structures, while the other require analysis of scenario units identified in the previous step. The following figure (Fig. 3.16) presents a high level view of the process of extraction of candidates to business rules.

The process iterates over a list of extracted scenario units and traverses each unit by analysing whether the behaviour unit matches certain rule pattern. When the match is identified, then behaviour unit element is replaced with the rule unit element. After that, the process checks whether the behaviour unit on which the current unit is control dependand was marked as a candidate to business rule, and if it was, then the dependency relationship between two rule units is being created.

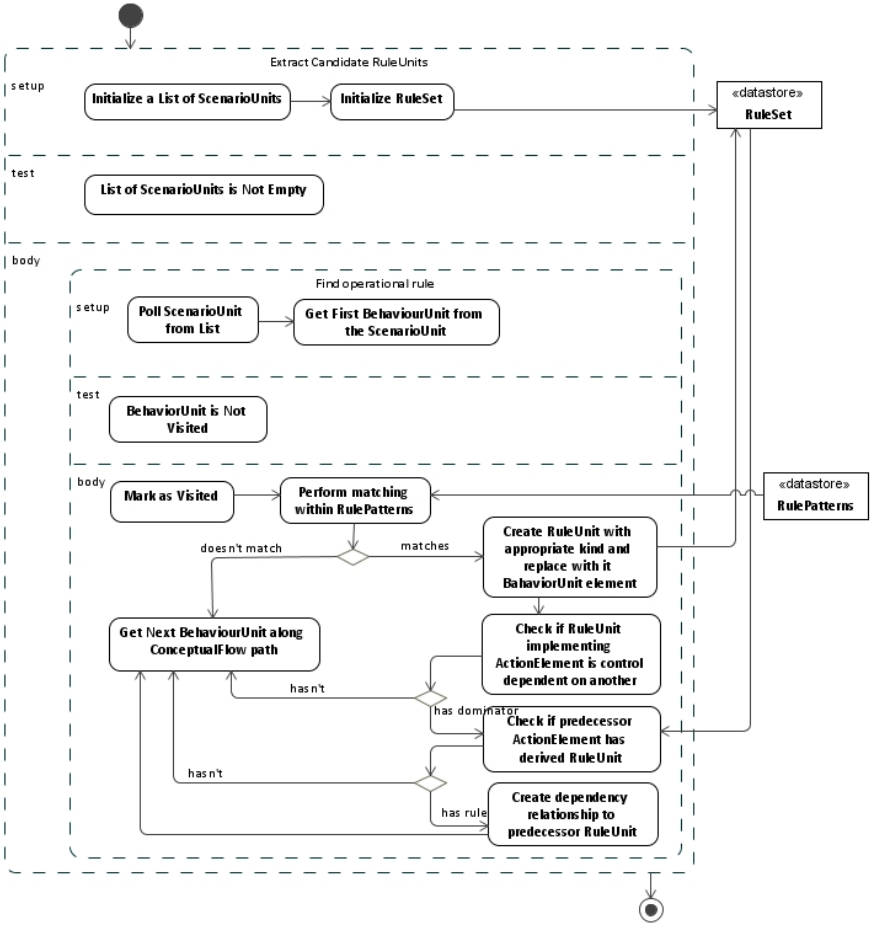

Fig. 3.16. Extraction of candidates to business rules from the scenario units

It is important to note that this approach allows introducing different patterns in matching step; therefore the extraction of candidates to business rules is not limited with the set identified within this work.

Patterns for Discovering Candidates to Structural Rules

Mandatory, uniquenes and value constrants are assertions that must always be true and that is considered to have immediate enforcement power. In the behavioural flow this kind of rule means the termination of flow if cerain condition is not met. The following figure (Fig. 3.17) presents a basic pattern when assertion is enforced in if-then-else statement which results in terminating the flow.

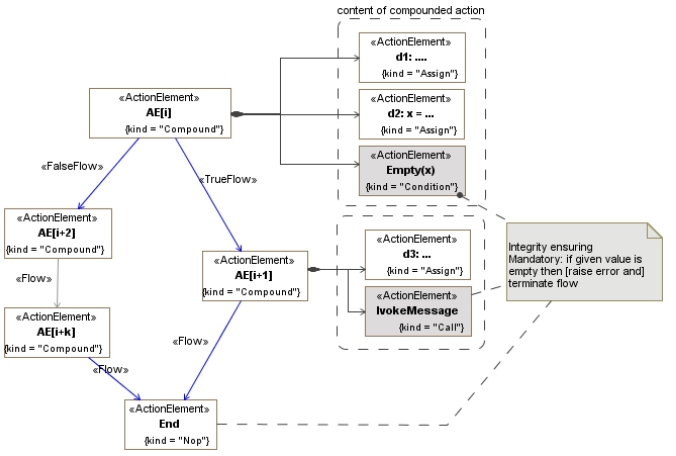

Fig. 3.17. Pattern for discovering constraints from if-then-else statement with immediate termination of the control flow

Pattern 1: *Direct successor of compound action element within behaviour flow contains return statement and direct flow to the end of the flow.*

The following example provides a code snippet that corresponds to this pattern:

```
x = ...;
if(x==""){
 sendMessage("Value cannot be empty!");
 return;
}else{
 ...
}
```

In the following we present a basic pattern for identifying constrains within the loop statement (Fig. 3.18). This pattern follows the same principles as the given above, except that it is applied in order to identify constraints on predefined population of instances (e.g. insertion of a set of items to database or checking existing set of items).

Pattern 2: *Compound action element representing a loop statement has a direct successor that consist of action element representing evaluation of loop iterator and having direct successor that consists of return statement.*

A code snippet corresponding to this pattern is as follows:

```
objects = getObjects();
while(objects.hasNext()){
 if (ojects.next().Title==""){
  sendMessage("Object title cannot be empty!");
  break;
 }}
```

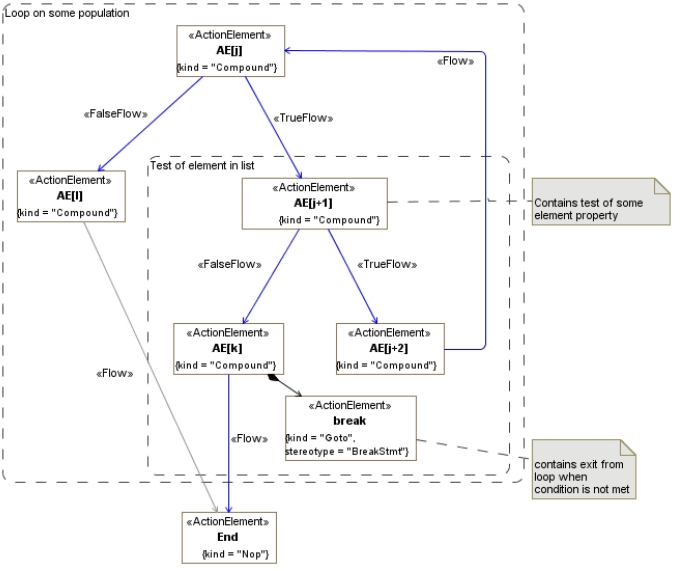

Fig. 3.18. Pattern for identifying constraints from loop statements with immediate termination of the control flow

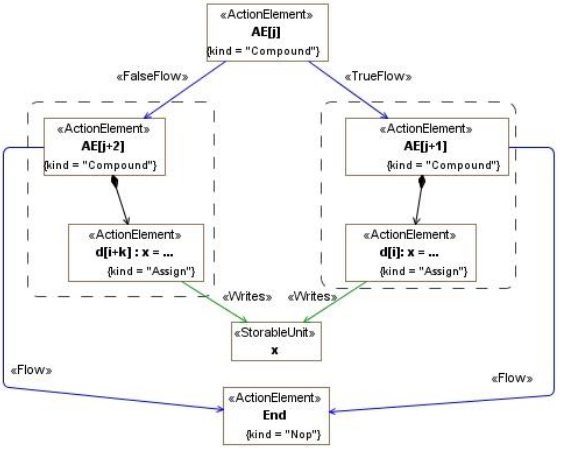

Fig. 3.19. Pattern for discovery derivation from if-then-else statement

In order to identify derivations within the behavioural flow, we define several patterns that correspond to the conditional statements: if-then-else and

switch. The figure above (Fig. 3.19) presents a pattern for identifying derivation rule in if-then-else statement.

Pattern 3: *If direct successors of compound action element, which is representation of condition if-then-else statement, contain different definitions of the same variable then this action element and its direct successors may be treated as a candidate derivation rule*.

A corresponding example for this pattern is represented in the following code snippet.

```
if (regData < GetDate()){
    ...
    isLate = true;
    ...
} else {
    ...
    isLate = false;
    ...
}
```

The following figure (Fig. 3.20) presents a pattern for identifying derivation rule in switch statement.

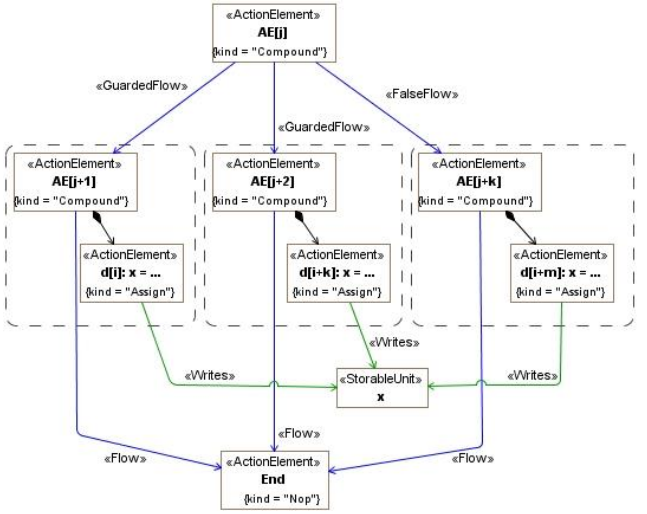

Fig. 3.20. Pattern for discovery derivation from switch statement

Pattern 4: *If more than one direct successors of compound action element representing switch statement contain different definitions of the same variable, then treat this action element and its direct successors as candidate derivation rule.*

The following example illustrates this pattern:

```
switch (objectKind){
    case closed:
      operation = notifyCurator;
    case preparing:
      operation = notifyOwner;
    default:
      operation = doNothing;
}
```

Patterns for Discovery Candidates to Operative Business Rules

Operative business rules are most likely to be implemented by controlling the behavioural flow of the business use case. However, not every every if-then-else statement may be treated as an operative rule because in certain cases actions performed when the condition is satisfied may define different derivation of the same variable (i.e. derivation rule).

In order to avoid treating derivation rules as operative, we introduce a basic pattern that checks whether the conditional statement (i.e. if-then-else or switch) is compound of statements that invoke procedures, define different variables, or another conditional statements. The following figure presents basic pattern for the operative rule discovering.

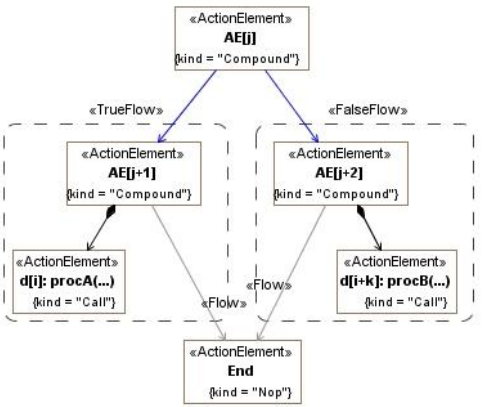

Fig. 3.21. Pattern for identifying candidate to operative rule

Pattern 5: *Direct successors of compound action element representing evaluation of conditions containing action elements represent call to procedure, do not write to the same variable which is alive in the current context, or represent another condition statement.*

An example of this pattern is as follows:

```
if(operation==notifyCurator){
 sendMessage(curator, message1);
}else{
 sendMessage(curator, message2);}
```

Complex operative rules are typically defined either as nested if-then-else statements or as switch statements common in the most programming languages. The following pattern presents a case of switch statement usage.

Pattern 6: *Direct successors of compound action element representing switch statements containing action elements that represent call to procedure, do not write to the same variable which is alive in the current context, or represent another condition statement.*

Corresponding example:

```
switch (objectState){
    case closed:
      notifyCurator();
    case preparing:
      notifyOwner();
    default:
      notifyResponsible();
}
```

3.2.4. Representing Extracted Knowledge

In order to represent extracted knowledge within the KDM, we developed an extension family (Fig. 3.22) containing elements that reflect general properties of the SBVR elements.

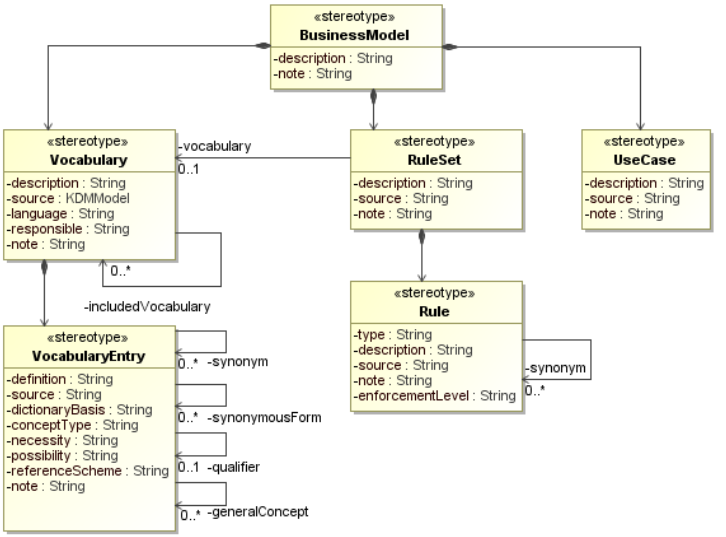

Fig. 3.22. An extension family for representing extracted knowledge

It is important to note, that the associations depicted within this illustration are intended only for demonstration purposes (as it was already mentioned, the

KDM does not allow defining any kind of relationship between stereotypes). For example, the recursive association "includedVocabulary" actually represents tagged reference from one KDM element to another one, which is signed with the Vocabulary stereotype.

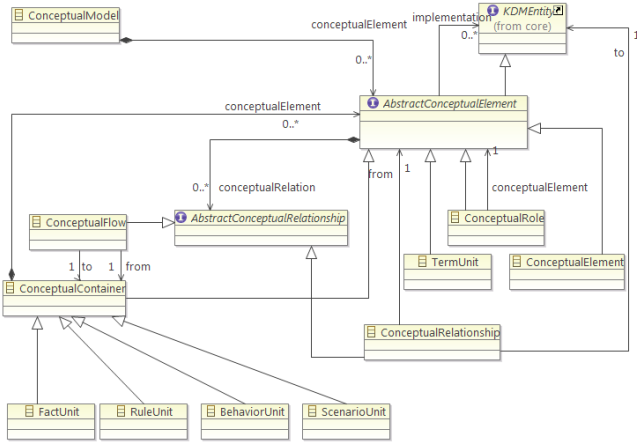

Fig. 3.23. A fragment of KDM meta-model (OMG 2011a) representing elements used to represent extracted knowledge

The remainder of this section provide definitions of the extension family for representing extracted knowledge. The stereotypes within extension family are applied to the elements of KDM Conceptual model, which corresponding fragment is presented in the Fig. 3.23.

Stereotype *BusinessModel* applied to conceptual model element indicates that this model contains elements representing business logic abstracted from the KDM representation. The following table provides explanation of the tag definitions that hold for the stereotype *BusinessModel*.

Table 3.3. Tag definitions applied to BusinessModel stereotype

Tag	Type	Description
description	String	The *Description* tag is used to describe the purpose and scope of the model.
note	String	Explanatory notes.

Stereotype *Vocabulary* applied to conceptual container element indicates that the content of the container is composed of conceptual elements representing vocabulary entries. Each element aggregated within this container must be

marked with the *VocabularyEntry* stereotype. The following table provides explanation of the tag definitions that hold for the stereotype *Vocabulary*.

Table 3.4. Tag definitions applied to Vocabulary stereotype

Tag	Type	Description
description	String	The "Description" tag is used to describe the purpose and scope of the vocabulary.
source	Reference	Reference to KDMModel which elements are being identified as vocabulary entries. If the main source of vocabulary is documentation of the software, then we propose to use InventoryModel with existing documentation as inventory items.
language	String	This tag is used to name the language that is the basis of the vocabulary.
includedVocabulary	Reference	This tag is a reference to another ConceptualContainer stereotyped as Vocabulary, indicating that another vocabulary is fully incorporated into the vocabulary being described.
responsible	String	This tag is used to name who controls and is responsible for the vocabulary (i.e. Speech Community).
note	String	The note tag is used for explanatory notes.

Stereotype *VocabularyEntry* indicates that the conceptual element (term unit, fact unit, or conceptual element representing some value) represents a business logic unit abstracted from the KDM representation. The following table provides explanation of the tag definitions that hold for the stereotype *VocabularyEntry*.

Table 3.5. Tag definitions applied to VocabularyEntry stereotype

Tag	Type	Description
definition	String	A definition is formulated as an expression that can be logically substituted for the primary representation. Definition patterns for identified noun concept: A <TermUnit> is a <definition>. A <TermUnit> is a <definition> [or <definition>]*. Definition pattern for identified individual concept: [The] <ConceptualElement> is the <definition>. Definition patterns for identified fact types Characteristic fact type: <TermUnit> <characteristic> Association fact type: *is-property-of* <TermUnit> has <TermUnit>

Table 3.5. (Continued)

Tag	Type	Description
		<TermUnit> is property of <TermUnit> *binary* <TermUnit> <verb> <TermUnit> <TermUnit> is related to <TermUnit> <TermUnit> is {greater | less} than [or equal to] <TermUnit> <TermUnit> {is equal to | equals} <TermUnit> *n-nary* <TermUnit> <verb><TermUnit>[<verb><TermUnit>]* Participation fact type: <TermUnit> includes <TermUnit> <TermUnit> is composed of <TermUnit> <TermUnit> is included in <TermUnit> Specialization fact type: <TermUnit> is a | is an <TermUnit> <TermUnit> specializes <TermUnit> <TermUnit> generalizes <TermUnit> <TermUnit> is category of <TermUnit> <TermUnit> is of category <TermUnit> Assortment fact type: <ConceptualUnit> is a | is an<TermUnit> <TermUnit> is one of the following: <ConceptualUnit> [,<ConceptualUnit>]* *Note*: ConceptualUnit with tagged value {conceptType,'Individual concept'> represents literal value identified within the KDM representation.
source	String	The source tag indicates the source of definition (i.e. a requirements specification or a user manual).
dictionaryBasis	String	The dictionary basis tag captures possible definition of vocabulary entry extracted from the available documentation and presented as a relevant statement or even a paragraph. If there are more than one matching indices in the document repository, then matching items are presented ordered by relevance (for each matching a corresponding tag is created). If there are no matching indices in document repository, then the definition is entered by a system analyst.
generalConcept	Reference	The general concept tag references the vocabulary entry with another entry which generalizes the entry concept.
conceptType	String	The concept type tag specifies a type of the entry concept: Certain value identified within code or database is tagged as individual concept.

Table 3.5. (Continued)

Tag	Type	Description
		TermUnit is implicitly identified as noun concept. FactUnit is implicitly identified as fact type.
kind	String	The kind tag represents a kind of fact type: characteristic, associative (incl. is-property-of), partitive, specialization, and assortment.
necessity	String	The necessity tag is supplemental to the entry definition. It claims that something is necessarily true. E.g. Each student has exactly one student id.
possibility	String	The possibility tag is supplemental to the entry definition. It claims that something is possibly true. E.g. It is possible that a student is participating in more than one program.
referenceScheme	String	The reference scheme tag is used to state how things denoted by the term can be distinguished from each other based on one or more facts about the things.
note	String	The note tag is used for explanatory notes.
synonym	Reference	The synonym tag references vocabulary entry (noun concept) with other vocabulary entry (noun concept), representing the same concept.
synonymousForm	Reference	The synonymous form tag references vocabulary entry (fact type) with other vocabulary entry (fact type), representing the same concept but formulated in different form.
qualifier	Reference	The qualifier tag references vocabulary entry that might not be unique within the vocabulary with another entry which qualify the first one uniqueness.

Stereotype *RuleSet* applied to conceptual container indicates that the content of this container is nothing but a set of rule unit elements, marked with Rule stereotype. The following table provides explanation of the tag definitions that hold for the stereotype *RuleSet*.

Table 3.6. Tag definitions applied to RuleSet stereotype

Tag	Type	Description
description	String	The description tag is used to describe the scope and purpose of the rules.
vocabulary	Reference	This vocabulary tag references the ConceptualContainer representing vocabulary used to construct rules in the rule set.
source	String	The source tag indicates the primary source, if the rules within the rule set were defined outside the KDM representation.
note	String	The note tag is used for explanatory notes.

Stereotype *Rule* applied to RuleUnit element represents a candidate to SBVR rule statement formulation. The following table provides explanation of the tag definitions that hold for the stereotype *Rule*.

Table 3.7. Tag definitions applied to Rule stereotype

Tag	Type	Description
type	String	The type tag is used to indicate the kind of candidate to business rule and may be: • structural business rule; • operative business rule.
description	String	The description tag is used to explain informally the rule statement
synonym	Reference	The synonym tag references additional, equivalent candidate rules.
note	String	The note tag is used for explanatory notes.
enforcementLevel	String	The enforcement level tag is used to define enforcement level of operative business rule: • strict – rule cannot be violated; • deferred – enforcement may be delayed; • pre-authorized – exceptions allowed prior approval from actors; • post-justified – if not approved after the fact, the sanctions or other consequences will ensue; • override – can be violated with explanations; • guideline – suggested, but not enforced.

In the following figure (Fig. 3.24) we present an example of vocabulary containing several entries: noun concepts represented as KDM *TermUnit* elements. Each vocabulary entry is supplemented with tagged values to provide the business domain specific information and also has traceability link to its actual implementation (attribute implementation). For the purpose of clarity, we provide an example where conceptual element has only one link to its implementation, though there may be multiple links, for example, table for persisting data, class for manipulating data within the code, form definition for representing or gathering data from the user. As it was described in the previous sections, this information is obtained by analysing bindings between different resources of software system.

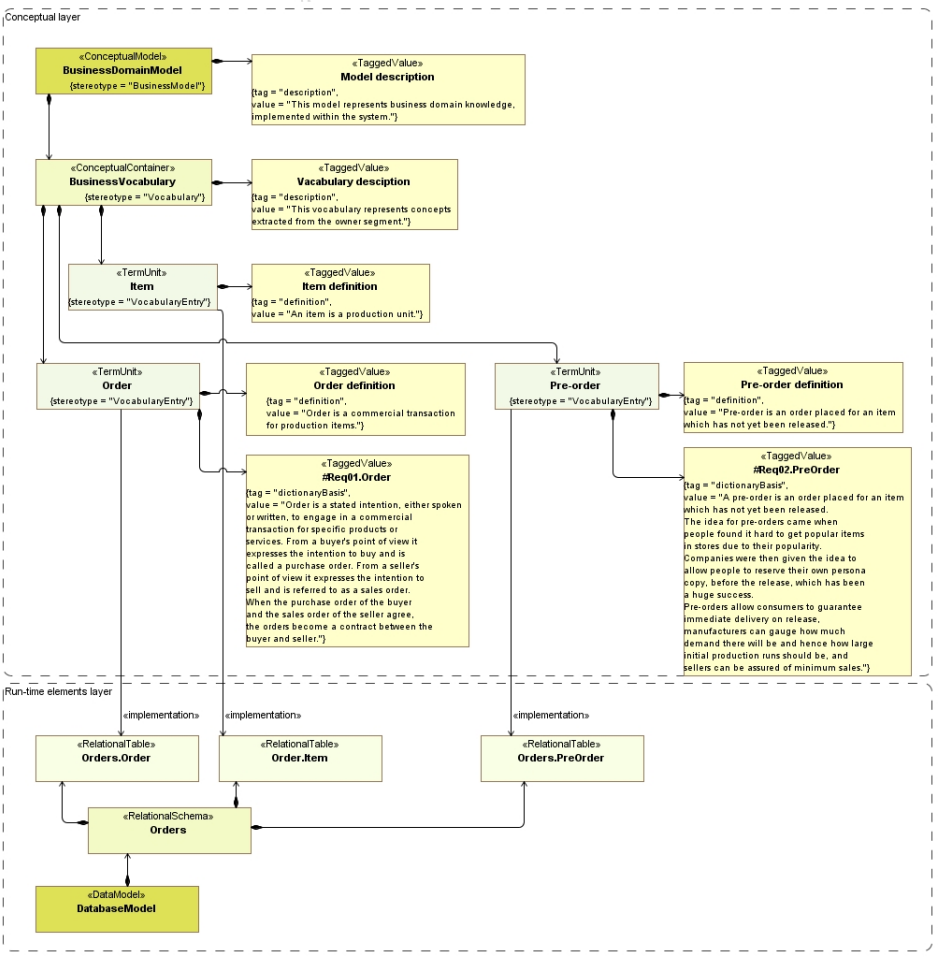

Fig. 3.24. Logical view of vocabulary representation

In the following examples we present the logical views of representation of fact types (Fig. 3.25) that involve noun concepts introduced above and representation of business rule (Fig. 3.26) that involves these fact types. The relationship between fact types and noun concepts is represented by *ConceptualRole* element. The same principle applies to the representation of the business rule. In the example of business rule, we also show the traceability of rule to its implemented that is the *ActionElement* representing conditional statement within a procedure.

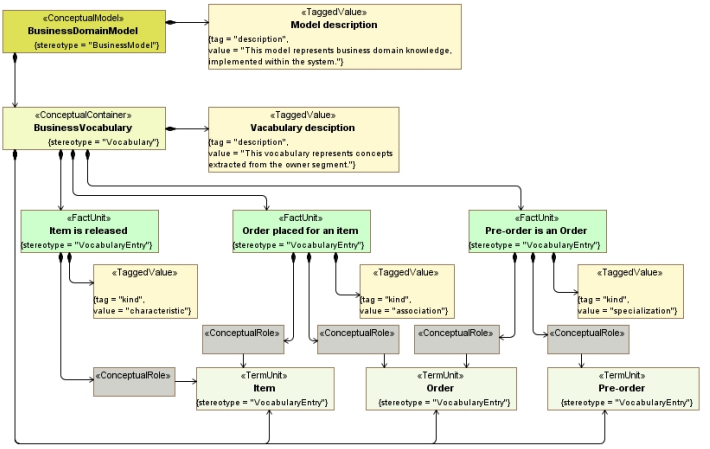

Fig. 3.25. Logical view of fact types composition

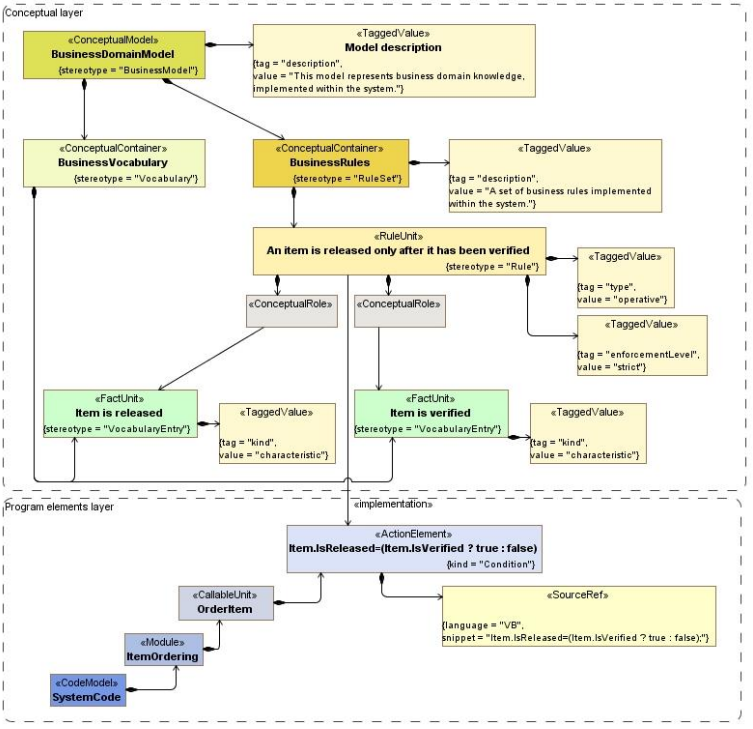

Fig. 3.26. Logical view of KDM representation for business rule composition and its implementation

3.3. Implementation

For the implementation of supporting tools we chose Eclipse Modeling Framework as a basis, because it is an open-source, supports model-driven engineering (i.e. model driven parser generators, meta-model based transformation tools, model graphic editors, etc.), and it is extendable allowing integration of additional tools. For more details on this framewok, we suggest to refer to (Budinsky *et al*. 2003) or to the project website[1]. The high-level view tool framework architecture is presented in the figure below (Fig. 3.27).

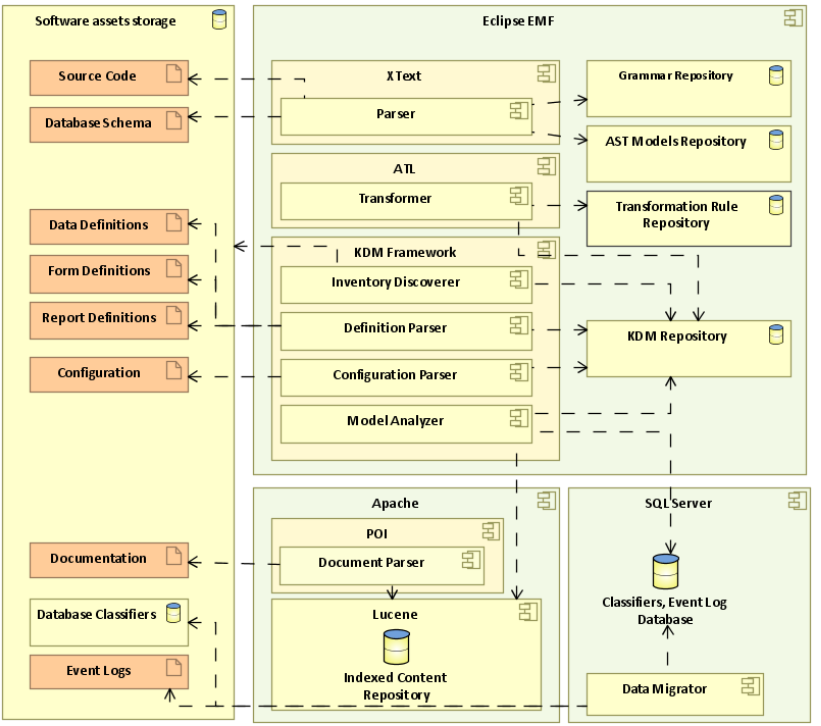

Fig. 3.27. Architecture of tool framework for proposed method

In order to create custom parsing tools, we use *XText* parser generator, that is compatible with EMF Ecore and provides parsed AST from source code as XMI models. This allows us define transformations from AST to KDM models. Transformations are defined using ATL transformation tool and processed by custom transformer. The KDM framework is implemented in the EMF

[1] http://www.eclipse.org/modeling/emf/

infrastructure. The processing of resources and configurations is performed by custom parsers.

To process documentation of the software system and store it in full-text search database, we selected Apache Lucene and Solr technologies in combination with PDF and DOC(X) document parsers. In order to store retrieved data, we selected MS SQL Server database management system.

3.4. Conclusions of the Third Chapter

In this chapter we presented the method for business knowledge extraction from existing software systems. The method consists of three main phases: preliminary study, knowledge extraction and business logic abstraction. The first phase is devoted to get acquainted with software architecture and prepare for the project. The next step involves preparatory tasks, including discovery of software physical inventory, parsing and transforming source code to KDM Code models, parsing and transforming platform resource specifications and configuration files, extracting lookup table values from the database, gathering available documentation and event log information. After having created KDM representation of software artefacts, we apply static analysis and patterns matching techniques to abstract business logic from this representation. Firstly, we establish dependency (both vertical and horizontal) relationships between model elements. Then, we extract business vocabulary by concerning the organization of data structure, corresponding documentation fragments and lookup data. Extracted vocabulary is being used during discovery of use-case scenarios and business rules in order to separate business logic implementing segments (i.e. code/resourse defs/configs) from the remaining parts.

The construction of the method allows us to draw the following conclusions:

1. By applying this process-centric and model-driven method, we are allowed to ensure traceability between extractable software artefacts and their representation within several layers of abstraction: inventory, code, run-time, and conceptual.
2. Usage of patterns based matching technique for business knowledge identification within the software artefacts ensures ability to extend the set of patterns by introducing new ones.
3. Obtained intermediate representation is extremely valuable as it enables model-based search and analysis as well as possibility to transform these models to other forms of representation, including decision tables/trees, reports, templates (e.g. SBVR), or diagrams (UML/OCL, BPMN).

4

Evaluation of the Proposed Method

4.1. Introduction

In order to evaluate the proposed method, we have performed a case study on the Enterprise Content Management (ECM) system (Normantas, Vasilecas 2012a; Normantas, Vasilecas 2012b). The system is being used for more than 5 years to support document management, records management, web content management, and collaboration processes in several governmental organizations of Lithuania.

This system was built on top of commercial off-the-shelf (COTS) platform that enables business specific customizations by proving integrated development environment. In order to meet the organizational requirements and electronic document management rules, the system has been greatly customized (Table 4.1 summarizes the artifacts contained by this software system). However, over the time requirements change and the software system must be adopted to reflect those changes. It leads to many undocumented modifications and even worse, because the impact of these modifications to other parts of the system is difficult to evaluate, they usually tend to be not fully tested. The maintenance cost

exceeds its budged; therefore, the method for automated system comprehension is essential to reduce the maintenance and evolution costs.

The main goals of this study were to identify whether the proposed method is feasible and if so, whether it is efficient enough to be further improved and used in practice to support software comprehension activities. In this chapter we describe the application of the method on ECM component which implementation involves all the architectural layers of the system, present the results obtained by applying the method, evaluate these results in respect with the efficiency metrics, and provide a discussion on the lessons learned from this study.

4.2. A Case Study on Enterprise Content Management System

As it was already introduced, the ECM is built on top of platform designed as client-server architecture (Fig. 4.1). The platform provides integrated development environment for the development of business specific solutions. In general the platform allows designing and deploying data object definitions that are binded with database tables or views, data editing and searching forms, reports, and workflows; create and configure application menu actions, platform actions and periodic jobs.

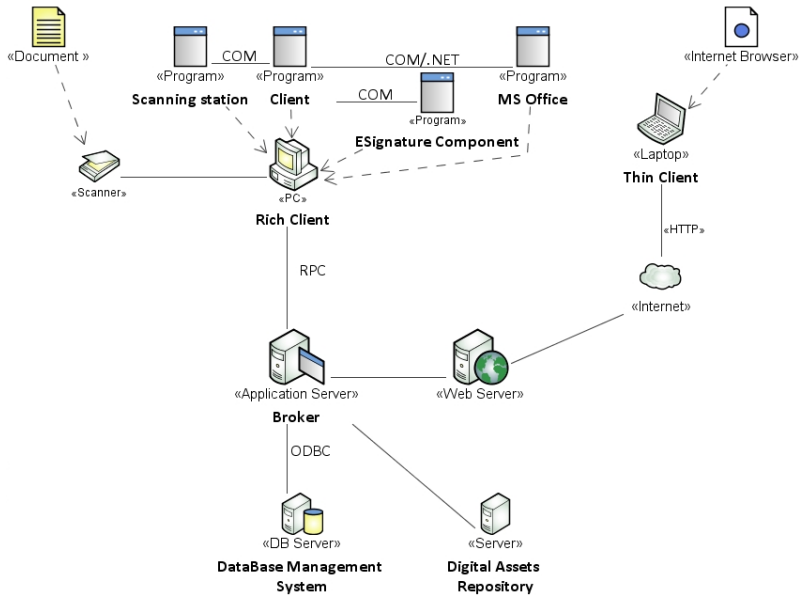

Fig. 4.1. Architecture of Enterprise Content Management System

It also provides component object model (COM) based application programming interface (API) allowing automation of particular system events using a dialect of the Visual Basic for Application (VBA) language and integration with external software systems. Internally the system may be considered as a black-box: the logic behind the interface may be understood only from software technical documentation, or from experience gained by using API in the development of business specific solutions. Therefore one of the main issues that should have been solved was the generation of the API representing code models.

Table 4.1. The overview of gathered information about the artefacts of the software system

Component	***Resources***	***Description***	***Format***	***Qnt***
Client	*Local configs*	Client specific configurations: defined actions, application window setting, etc.	INI	200
-is a desktop application that interacts over network with broker server using RPC; with external applications over COM API.	*Log files*	Client specific log information.	CSV	200
	Folders	User specific collection of documents.	XML	155
Broker Server	*Application files*	Platform components	DLL, EXE, etc.	132
-is a service component containing subcomponents for interaction with clients (by distributing its resources to client applications), database management system (ODBC), web server (Gateway Server), and stores document objects in repository.	*Macros*	Visual Basic for Application script modules containing module level constants, external library declarations, procedure and function declarations (that handles certain resource event).	VBA	57 (>40KLOC)
	Defs	Document, lookup and audit table definitions.	XML	187
	Reports	Data representation and data source definitions that are based on one or more Defs.	XML	16
	Forms	Definitions of forms for data editing or querying; are based on one or	XML	83

Table 4.1. (Continued)

Component	*Resources*	*Description*	*Format*	*Qnt*
		more Defs.		
	Workflows	Workflow definitions: activities, transitions, decision points; events corresponding to these workflow elements (on entry, on exit, etc.)	XML	5
	Config. files	Platform specific configuration.	INI, XML	3
	Log files	Server log information.	CSV	3
Database Management System	*DB Tables*	Database tables storing document metadata, lookup table values, audit values, or custom data.	T-SQL	80
-used to store and retrieve document metadata, lookup table values, audit, and other system data.	*DB Views*	Stored SELECT statements that associate one or more tables to retrieve data from them, applies predicates for data filtering, calls functions for output data formatting.	T-SQL	92
	DB Procedures	Stored procedures that are called from VB script over application provided interface to database connectivity component.	T-SQL	30
	DB Triggers	Procedures that are executed in response to particular events on table; used for internal audit, temporary tables.	T-SQL	10
	DB Functions	User Defined Functions that process input data to produce required output, and are called from stored procedures or views.	T-SQL	26
	Lookup table data	Information classifying data, such as document kind, type, status and etc.	Data	20
	Log	Database specific log.	Data	2
ScanStation	*Templates*	Template definitions for optical content recognition.	XML	23

Table 4.1. (Continued)

Component	*Resources*	*Description*	*Format*	*Qnt*
-scans physical document, performs optical recognition, and transfers textual information to client.	*Config. files*	Fields mapping definitions	INI, XML	2
Office Integration	*Config. files*	Configuration mapping Fields mapping definitions	INI	1
-transfers user filled document templates to client.	*Document templates*	Word document templates containing fields mapped to document fields.	doc, docx	20

4.2.1. The Strategy

Having identified the artifacts of the software system and its integration components, we defined the strategy for the knowledge extraction. Within the strategy it was identified that 8 custom parsers and 13 transformation rules sets must be developed in order to obtain the "as-is" KDM representation of the system.

However, as the main goal of this case study was to identify whether method is feasible and how efficient it is, we chose to analyse only a part of the system. This part part was Received documents registration component which included artefacts described in the following table.

Table 4.2. Content of Received documents registration component

Category	*Artefact*	*Description*
Run-time resources		
User Interface	Registration form	Form for providing meta-data about document to be indexed within the document repository.
	Search form	Form for search and retrieval of receivable documents.
Data Definitions	Document definitions (1)	Data structure definitions for programmatically accessing document meta-data.
	Lookup definitions (10)	Data structure definitions for programmatically accessing data used to classify documents
Configuration Files	Client configuration (3)	Configuration file parameterizing application client (were selected for different profiles).

Table 4.2. (Continued)

Category	*Artefact*	*Description*
	Application configuration (1)	Configurations file parameterizing application settings.
Database Schema	Tables (15)	Relational tables that stores meta-data about the documents and their classifiers, including transitional tables (i.e. implementation of many-to-many relationships).
	Indexes (113)	Indexes defined on columns of the tables.
	Constraints (23)	Foreign key constraints ensuring data integrity.
	Views (11)	Views that are used to project data from tables and provide to search form, lookup fields (combo-boxes).
Program elements		
Script Modules	Forms module (3890 LOC)	Functions that handle form events: open, close, button click, etc.
	Registration module (598 LOC)	Functions that implement registration logic.
	Signature module (1017 LOC)	Functions that implement document signing logic.
	Forwarding module (892 LOC)	Functions that implement functionality of document forwarding to responsible persons.
	Utilities and other supporting modules (>5 KLOC)	General purpose functions that are used by other modules.
Data		
	Document kind	Values from the table.
	Document tag	Values from the table.
	Reception kind	Values from the table.
Documentation		
	Requirements specification	Initial requirements specification.
	Platform documentation	Documentation of enterprise content management system API.

In order to represent extracted model elements in the terms familiar to software maintainers, we created a set of KDM extension families consisting of platform and domain specific stereotypes. As the KDM does not support relationships between stereotypes, we named extension families by following the namespace mechanism as it is represented in the figure below.

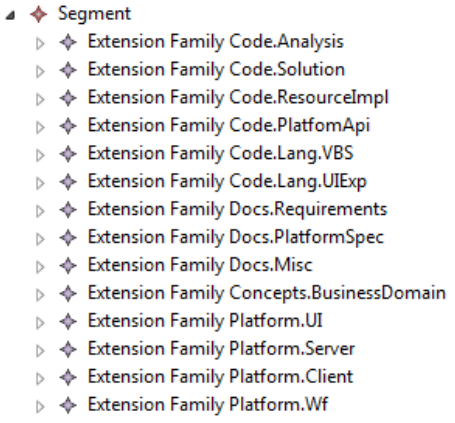

Fig. 4.2. A Set of extension families created for the study

4.2.2. Discovering Knowledge about the System

Receivable document registration component is partially represented in the figure below (Fig. 4.3).

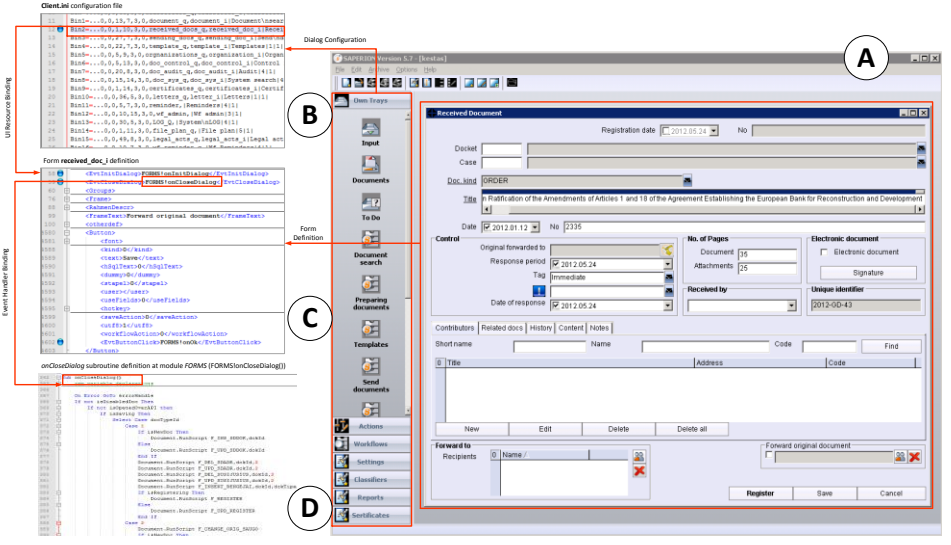

Fig. 4.3. The main dialog of ECM and the form to fill document metadata (A); a snippet of configuration file that defines left side tray bar (B); a snippet of form definition (C); a fragment of code module containing subroutine that handles form event (D)

The figure presents the main application dialog and the form to fill metadata of receivable document (A). The structure of document object is defined within the data definition file that maps to database object (table or view). The configuration of trays for the particular client is predefined within client.ini configuration file (B). A configuration of tray is a key-value pair that reference to form definitions, which are opened due to action performed by a user of the system (drag-and-drop document object and a double click). The form definition (C) contains a description of UI elements within the form and references to the form or its controls event handling code procedures declared within macro modules (D).

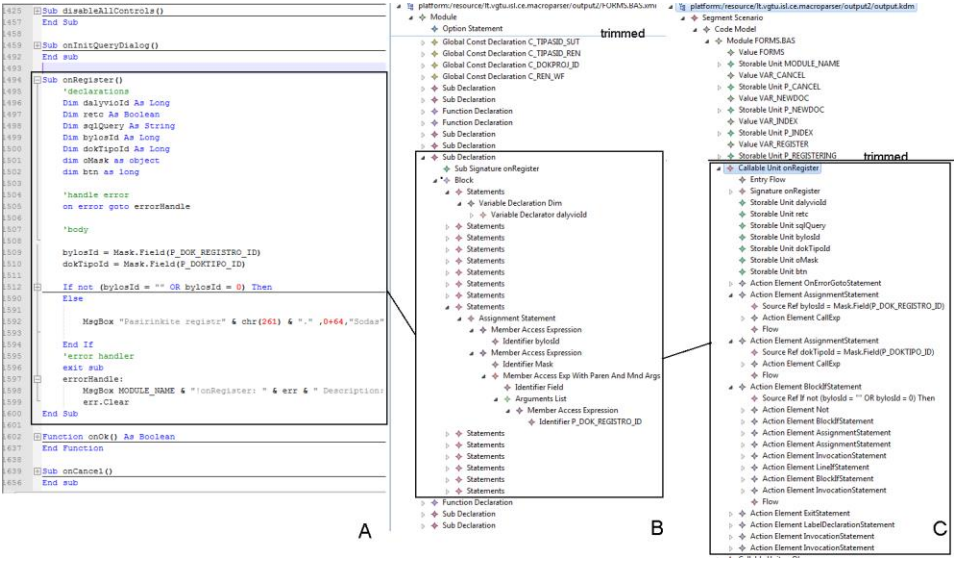

Fig. 4.4. A fragment of VB script module (A), its parsed AST in ECore format (B), and transformed KDM Code model (C)

In order to obtain software artefacts representation within the KDM, we first of all had to parse the content of artefacts into the intermediate form (i.e. AST) that would be possible to transform to KDM. As it was already mentioned, to parse programing code (grammar definition and SQL meta-model are presented in the Appendix C and Appendix D respectively) we used XText parser generator which allowed us to create parsers that produce AST in the ECore format. For the structured content, such as configurations, form and data definitions, we created separate parsers. The following figure (Fig. 4.4) shows a fragment of script module, its parsed AST, and transformed KDM Code model.

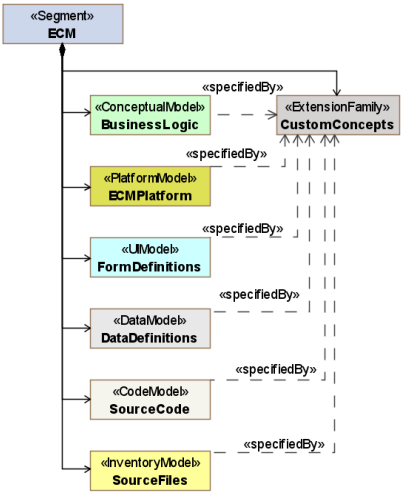

Fig. 4.5. Logical view of extracted segment for representing knowledge about the software system

During the knowledge extraction step, the following models were obtained (Fig. 4.5):

- InventoryModel – to represent software physical artefacts and their organization;
- CodeModel – to represent software source code;
- UIModel – to represent software user interface definitions;
- DataModel – to represent database structure;
- PlatformModel – to represent configurations and platform actions.

We have also extracted classifiers used as lookup tables and field range specifications in form definitions (i.e. values of drop-down list controls). After that, we have parsed available documentation in the structure that we have defined in the previous chapter. Based on the obtained structure we created full-text indexing schema definition and store parsed data into the repository.

4.2.3. Business Logic Abstraction

Having extracted intermediate representation of knowledge about the software system artefacts, we applied business logic extraction principles and techniques presented in the previous chapter. Below we provide a summary of extracted business logic with some translations we have made considering the SBVR formulation guidance. These translations may be treated as suggestions

for analysts how to further formulate definitions of corresponding conceptual elements.

Table 4.3. Summary of extracted business vocabulary

Elements	***#***	***Examples***
Vocabulary		
Noun concepts	239	Document; receivable document; registration number; registration date;
Individual concepts	1145	"Organizational course", "Control", "Law", "Record-keeping", "Finance and account", "Human resources" (i.e. from the classifier Document folders)
Fact types		
Characteristics	29	document is electronic; document is registered; document is signed;
Associations	15	receivable document is composed by Contributor; document is related to document; receivable document is forwarded to document receiver;
Is-property-of	171	Document has document title; Document has document registration number; Document has document registration date; Receivable document has date of response;
Specializations	4	Receivable document is a document (disjoint, complete); Document is of category document kind (overlapping, incomplete); Document contributor is an organization (overlapping, incomplete); Responsible person is a person;
Participations	1	Receivable document contains of document contributors;
Assortments	8	Document receiving method is one of the following: "By e-mail"," By mail", "By courier", "By fax".

Table 4.4. Summary of extracted rule set

Elements	***#***	***Examples***
Rule Set		
Mandatory constraints	27	It is necessary that each document has a document title; It is necessary that each receivable document has a document date and has a document number.
Uniqueness constrains	8	It is necessary that each document has unique document registration date and document registration number. It is necessary that each document has unique document unique identifier.
Value constraints	11	It is impossible that document registration date is greater than current date. It is impossible that document pages number is less or equal to zero.
Derivations	63	It is necessary that each document is registered if and only if it has assigned document order number. It is necessary that each document unique identifier is derived from document type code and *current year* and document counter.
Calculations	0	
Operative rules	23	It is obligatory that each document that is electronic document and is signed document must be signed by registrar during document registration. It is obligatory that each document must be added to documents folder (case) during document registration. Editing document that is registered is permitted only to registrar.

Table 4.5. Summary of extracted use cases

Use cases	*9*	Receivable document search and it extending use cases: View, Edit, Delete, Sign; Receivable document register and it extending use cases: Forward to responsible, Sign, Add to folder (case); Receivable document save and it extending use cases: Forward to responsible, Sign;

4.3. Evaluation

In order to evaluate the results of this study, we use several most frequent measures for establishing effectiveness of information retrieval and classification. These measures are precision, recall, and accuracy:

- *Precision* (*P*) is the fraction of retrieved results that are relevant.
- *Recall* (*R*) is the fraction of relevant results that are retrieved.
- *Accuracy* (*A*) is the fraction of results that are correct.

The following criteria must be identified to calculate these measures:

Table 4.6. Information retrieval metrics

	actual class (observation)	
predicted class (expectation)	**tp** – Correct Results	**fp –** Unexpected Results
	fn – Missing Results	**tn –** Correct absence of result

Where *tp* – true positive, *fp* – false positive, *fn* – false negative, and *tn* – true negative.

Such that, the precision is calculated:

$$P = \frac{tp}{tp + fp} \quad (4.1)$$

Recall is calculated as follows:

$$R = \frac{tp}{tp + fn} \quad (4.2)$$

Accuracy is calculated as follows:

$$A = \frac{tp + tn}{tp + fp + fn + tn} \quad (4.3)$$

A single measure that trades of the precision versus recall is the *F measure*, which is the weighted harmonic mean of precision and recall. It is calculated as follows:

$$F = \frac{1}{\alpha \frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \text{ , where } \beta^2 = \frac{1-\alpha}{\alpha} \quad (4.4)$$

The method was applied to gather required information. System analyst was responsible for knowledge extraction and verification in respect with the specified extraction rules, while the business analyst was asked to approve the ex-

tracted results for relevance. The results of this evaluation are presented in the following tables.

Table 4.7. Evaluation of candidates to business vocabulary entries extraction efficiency

Candidate Element	*tp*	*fp*	*fn*	*tn*	*Precision*	*Recall*	*Accuracy*	*F-measure*
Noun concept (TermUnit)	239	35	10	1110	0,87	0,96	0,97	239
Individual concept (ConceptualElement)	1145	60	-	-	0,95	1,00	0,95	1145
Fact types (FactUnit)								
Characteristic	29	4	1	-	0,88	0,97	0,85	29
Association	15	4	2	18	0,79	0,88	0,85	15
Is-property-of	171	-	2	151	1,00	0,99	0,99	171
Specialization	4	-	1	-	1,00	0,80	0,80	4
Participation	1	2	-	6	0,33	1,00	0,78	1
Assortment	8	2	-	-	0,80	1,00	0,80	8
	Average				0,83	0,95	0,87	0,83
	Median				0,88	0,98	0,85	0,88
	Deviation				0,22	0,07	0,08	0,22

Table 4.8. Evaluation of candidates to business rules extraction efficiency

Candidate Element	*tp*	*fp*	*fn*	*tn*	*Precision*	*Recall*	*Accuracy*	*F-measure*
Rule (RuleUnit)								
Operative	23	6	2	7	0,79	0,92	0,79	23
Structural								
Mandatory constraint	27	8	2	10	0,77	0,93	0,79	27
Uniqueness constraint	8	2	1	0	0,80	0,89	0,73	8
Value constraint	11	3	0	8	0,79	1,00	0,86	11
Derivation	63	9	2	13	0,88	0,97	0,87	63
	Average				0,81	0,94	0,81	0,81
	Median				0,79	0,93	0,79	0,79
	Deviation				0,04	0,04	0,06	0,04

As it could be observed from the efficiency evaluation, high precision and recall rates indicate that extracted results were relevant enough in respect with the defined extraction principles; therefore we could treat the method as an efficient. On the other hand, looking at the concrete numbers, it could be noticed that within the relatively small population of candidates to business rules, there were still many unexpected and missing results. For this reason, a set of patterns for identifying rules should be reviewed.

In order to consider the plausibility of the evaluation results, we have to concern the threats for internal, external, and construction validity.

Internal validity. As it was already mentioned, there is no large population of input data that would make it possible to obtain statistically representative results. Moreover, if this study would be replicated with other cases involving different system and business analysts, the results concerning Precision and Recall measures may have some deviations due to analysts' subjective point of view.

Construct validity. The measures for the evaluation of the efficiency of results obtained within the case study were chosen from the information retrieval field, in which they have an adequate maturity level. These measures allowed us to check whether extraction knowledge is that what we had expected and identify issues that we should consider.

External validity. The external validity is concerned with the generalization of the results. This research considers existing enterprise software systems as the whole population. For this reason, the standard for intermediate knowledge representation (KDM) has been chosen and commonly accepted business knowledge classification scheme has been considered to formulate knowledge extraction principles. In this respect, the results could be generalized for the whole population, though certain principles may have to be revised depending on specific software systems.

4.4. Discussion

Application of the method in the case study of the enterprise content management system revealed some issues that are worth to discuss. First of all, as it could have been expected, development of a method for business logic discovery from software systems inevitably leads to dependence on certain architectural and technological factors; yet, it is impossible to create universal method that would cover all possible technological aspects that may be met in different software systems. In this study we chose enterprise level software systems that are most often offered as products (also called commercial-off-the-shelf, COTS) that provide core functionality extendable according to business needs.

Although we applied the method on the system which is only one of many possible systems, we have observed that the same implementation decisions can be found in other similar purpose systems. This study has shown that in order to operate on the precise and rigorous knowledge about the software system, different parsers and transformers have to be created and that is not a trivial task. The main reason for this is a large diversity of specific programming (or scripting) languages without a clear grammar definition.

Moreover, the study has shown that business logic may be distributed across different software artefacts covering multiple abstraction layers. It also endorsed that large part of program code within the software system is dedicated to support platform specific activities, including data type conversions, user interface control, or certain kinds of work-around for specific system behaviour.

At the same time, this study allowed to identify multiple implementations of the same rules. Extracted set of candidates to business rules may be adjusted with business stakeholders and considered for validness, migration or separation (e.g. to business rules management systems) to other system.

4.5. Conclusions of the Fourth Chapter

In this chapter we presented the evaluation of the proposed method for business knowledge extraction from existing software systems. We described a case study from our research project, and discussed on obtained results. We can conclude that:

1. The results of case study endorse the feasibility and efficiency of the proposed method. The extracted vocabulary was of an average of 92% of precise and of an average of 94% of recall. The extracted set of candidates to business rules was of an average of 85% and of an average of 89% of recall. Both forms of knowledge representation preserved traceability links to their implementation allowing system analysts or maintainer more quickly identify software parts of interest.
2. However, KDM representation is only intermediate format valuable for automated analysis. Seeking to produce more comprehensive representations of views of particular software system aspect, the conversion to static and dynamic UML models should be considered in the further research. In order to be able to validate discovered knowledge with stakeholders, candidates to business rules should be transformed to SBVR, decision tables or trees, or other format negotiable for business related people.

General Conclusions

1. In the first chapter we presented the systematic literature review of related methods for business knowledge extraction from existing software systems. Selected studies were published in peer-reviewed scientific papers in journals, conference proceeding and collections. From this review we can conclude that:
   a) there is insufficient attention paid to extraction of business vocabulary and no concern is made for widely used business rule classification schemes;
   b) the proposed methods rarely concern knowledge extraction as a complex problem which involves more than program code analysis, thus aggravating their application in knowledge extraction from software systems developed using multiple technologies;
   c) the most common techniques for knowledge extraction are program slicing, patterns matching and transformations, although certain methods use the combination of them;
   d) a minority of proposed methods were evaluated in industrial case studies on large enterprise software systems, considering various software artefacts and other available knowledge sources;
   e) the Knowledge Discovery Meta-model (KDM), as an intermediate representation of knowledge about existing software systems, is

not studied enough and there is a lack of algorithms for creating representations within the KDM and performing analysis on the models.

2. It has been established that the KDM supported stereotype-based extension mechanism is limited in producing rigorous and precise representations and in performing analysis on its models, as it does not allows to define various kinds of relationships between stereotypes.
3. It has been identified that the KDM does not provide formally defined well-formedness constraints on meta-model elements; therefore, discovered models cannot be validated for conformance to the KDM. For this reason, a partial set of well-formedness rules for Code and Data models has been provided within this thesis.
4. It has been observed that the KDM does not support meta-model elements for representing and evaluating information required for data-flow analysis. Therefore a number of extensions in form of derived properties of KDM elements were proposed and their derivation has been formally defined.
5. It has also been established that the KDM is insufficiently adjusted to represent extracted business knowledge. For this reason, an extension family of stereotypes corresponding to fundamental SBVR concepts was proposed. It is supposed that this extension would allow further transformation of extracted knowledge to more abstracts formats.
6. The results of case study prove the feasibility and show the efficiency of the proposed method. The extracted vocabulary was of an average of 92% of precise and of an average of 94% of recall. The extracted set of candidates to business rules was of an average of 85% and of an average of 89% of recall. Both forms of knowledge representation preserve traceability links to their implementation allowing system analysts or maintainer more quickly identify software parts of interest.

References

Abran, A.; Bourque, P.; Dupuis, R.; Moore, J. W.; Tripp, L. L. 2004. *Guide to the Software Engineering Body of Knowledge − SWEBOK*, 2004th edn, IEEE Press, Piscataway, USA.

Aho, A. V.; Lam, M. S.; Sethi, R.; Ullman, J. D. 2006. *Compilers: Principles, Techniques, and Tools*, 2nd edn, Boston: Addison-Wesley Longman Publishing.

Ambler, S. W.; Sadalage, P. J. 2006. *Refactoring Databases: Evolutionary Database Design*, Boston: Addison-Wesley Professional.

Andersson, M. 1994. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, in *Proceedings of the 13th Int. Conference of the Entity Relationship Approach*. Berlin: Springer Berlin Heidelberg, 403−419.

Antoniol, G.; Canfora, G.; Casazza, G.; Lucia, A. D. 2000. Information retrieval models for recovering traceability links between code and documentation, in *International Conference on Software Maintenance*, San Jose, USA, 40−49.

Antoniol, G.; Canfora, G.; Casazza, G.; Lucia, A. D.; Merlo, E. 2002. Recovering Traceability Links between Code and Documentation, *IEEE Transactions on Software Engineering* 28(10): 970–983.

Arevalo, G.; Ducasse, S.; Nierstrasz, O. 2005. Lessons learned in applying formal concept analysis to reverse engineering, in *Proceedings of the Third international conference on Formal Concept Analysis*. Berlin: Springer Berlin Heidelberg, 95−112.

Bajec, M.; Krisper, M. 2001. Managing Business Rules in Enterprises, *Electrotechnical Review* 68(4): 236–241.

Bajec, M.; Krisper, M. 2005. A methodology and tool support for managing business rules in organisations, *Information Systems* 30(6): 423−443.

Baxter, I. D.; Mehlich, M. 2000. Reverse engineering is reverse forward engineering, *Science of Computer Programming* 36(2): 131−147.

Bennett, K.; Rajlich, V. 2000. Software maintenance and evolution: a roadmap, in *Proceedings of the Conference on The Future of Software*. New York: ACM, 73−87.

Bergeretti, J. F.; Carre, B. A. 1985. Information-flow and data-flow analysis of while-programs, *ACM Transactions on Programming Languages and Systems* 7(1): 37−61.

Booch, G.; Rumbaugh, J.; Jacobson, I. 2005. *The Unified Modeling Language User Guide*, 2nd edn., Boston: Addison-Wesley Professional.

Brereton, P.; Kitchenham, B. A.; Budgen, D.; Turner, M.; Khalil, M. 2007. Lessons from applying the systematic literature review process within the software engineering domain, *Journal of Systems and Software* 80: 571−583.

Budinsky, F.; Steinberg, D.; Ellersick, R. 2003. *Eclipse Modeling Framework: A Developer's Guide*, Boston: Addison-Wesley Professional.

Canfora, G.; Cimitile, A. 1995. Software Maintenance, in *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, June 22-24, 1995, Rockville, Maryland, USA, 478−486.

Chaparro, O.; Aponte, J.; Ortega, F.; Marcus, A. 2012. Towards the Automatic Extraction of Structural Business Rules from Legacy Databases, in *19th Working Conference on Reverse Engineering* (WCRE), Kingston, Ontario, Canada, 479−488.

Chapin, D. 2008. SBVR: What is now Possible and Why? *Business Rules Journal* 9 (3).

Chiang, C. C. 2006. Extracting business rules from legacy systems into reusable components, in *IEEE/SMC International Conference on System of Systems Engineering*, IEEE, 1-6.

Chiang, C. C.; Bayrak, C. 2006. Legacy Software Modernization, in *IEEE International Conference on Systems, Man and Cybernetics*, Taipei, Taiwan, 1304−1309.

Chikofsky, E. J.; Cross, J. H. 1990. Reverse engineering and design recovery: a taxonomy, *Software* 7(1): 13−17.

Columbus, L. 2013. Roundup of Cloud Computing & Enterprise Software Market Estimates and Forecasts, 2013, [cited 12 February 2013], Available from Internet: <http://www.forbes.com/sites/louiscolumbus/2013/02/01/roundup-of-cloud-computing-enterprise-software-market-estimates-and-forecasts-2013/>.

Cooper, K. D.; Harvey, T. J.; Kennedy, K. 2001. A simple, fast dominance algorithm, *Software Practice and Experience* 4: 1−10.

Earls, A. B.; Embury, S. M.; Turner, N. H. 2002. A method for the manual extraction of business rules from legacy source code, *BT Technology Journal* 20(4): 127−145.

Flores, N.; Aguiar, A. 2005. Reverse engineering of framework design using a meta-patterns-based approach, in *IEEE Computer Society*, Cairo, Egypt, 941−946.

Gang, X. 2009. Business Rule Extraction from Legacy System Using Dependence-Cache Slicing, *IEEE Computer Society*, Washington, USA, 4214−4218.

Glass, R. L. 2012. A Study About Software Maintenance, *Information Systems Management* 29(4): 338–339.

Hay, D. 2000. Defining Business Rules: What Are They Really. Final Report, [cited 24 August 2011], Available from Internet: <www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf>.

Hay, D. C. 2002. *Requirements Analysis Architecture*, Upper Saddle River: Prentice Hall PTR.

Halle, B. V. 2001. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*, 1st edn., San Francisco: Wiley.

Hevner, A. R.; March, ST.; Park, J.; Ram, S. 2004. Design science in information systems Re-search, *Society for Information Management and The Management Information Systems Research Center* 28(1): 75−105.

Huang, H. 1996. Business Rule Extraction from Legacy Code, in *Proceedings of the 20th Conference on Computer Software and Applications*, IEEE Computer Society, Washington, USA, 162−168.

Hung, M.; Zou, Y. 2007. Recovering Workflows from Multi Tiered E-commerce Systems, in *15th IEEE International Conference on Program Comprehension*, Banff, Alberta, Canada, 198−207.

ISO/IEC, 2008. *Information technology – Database languages − SQL − Part 2: Foundation (SQL/Foundation)*, [cited 03 December 2012], Available from Internet: <http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53682>.

ISO/IEC/IEEE, 2008. *ISO/IEC/IEEE Standard for Systems and Software Engineering − Software Life Cycle Processes*.

Kalibatiene, D.; Vasilecas, O. 2008. Formal transformation of ontology axioms to application domain rules, in *Proceedings of the 9th International Conference on Computer Systems and Technologies*. New York: ACM, 63: V.1−63:1.

Kalibatiene, D.; Vasilecas, O. 2010. Ontology-based application for domain rules development, *Computer Science and Information Technologies* 38(4): 271−282.

Kalsing, AC.; Nascimento, GS. D.; Iochpe, C.; Thom, LH. 2010. An Incremental Process Mining Approach to Extract Knowledge from Legacy Systems, in *IEEE 14th Enterprise Distributed Object Computing Conference*, Vitoria, Brazil, 79−88.

Khedker, U.; Sanyal, A.; Karkare, B. 2009. *Data Flow Analysis: Theory and Practice*, 1st edn. Boca Raton: CRC Press, Inc.

Kitchenham, B.; Charters, S. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*, Technical report, Keele University and Durham University Joint Report.

Kleppe, A.; Warmer, J.; Cook, S. 1999. Informal Formality? The Object Constraint Language and Its Application in the UML Metamodel. *First International Workshop*, Mulhouse, France. 148−161.

Kohavi, R.; Sommerfield, D. 1998. Targeting Business Users with Decision Table Classifiers, *AAAI Press*, 249-253.

Lakhotia, A.; Gravley, J. M. 1995. Toward Experimental Evaluation of Subsystem Classification Recovery Techniques, *IEEE Computer Society*, 262−269.

Lehman, MM. 1980. Programs, life cycles, and laws of software evolution, in *Proceedings of the IEEE* 68(9): 1060−1076.

Lientz, B. P.; Swanson, B. E. 1980. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Boston: Addison-Wesley.

Lucia, A. D. 2001. Program slicing: methods and applications, *First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 142−149.

Maqbool, O.; Babri, H. 2007. Hierarchical Clustering for Software Architecture Recovery, *IEEE Transactions on Software Engineering* 33(11): 759−780.

Morgan, T. 2002. *Business Rules and Information Systems: Aligning IT with Business Goals*, Pearson Education.

Nascimento, G. S.; Iochpe, C.; Thom, L.; Kalsing, A. C.; Moreira, A. 2009. A Method For Rewriting Legacy Systems Using Business Process Management Technology, in *Proceedings of 11th Internaltional Conference on Enterprise Information Systems*, Milan, Italy, 57−62.

Nascimento, G. S.; Iochpe, C.; Thom, L.; Kalsing, A. C.; Moreira, A. 2012. Identifying Business Rules to Legacy Systems Reengineering Based on BPM and SOA, in *B Murgante, O Gervasi, S Misra, N Nedjah, AC Rocha, D Taniar, B Apduhan (eds.), Computational Science and Its Applications ICCSA 2012*. Berlin: Springer Berlin Heidelberg, 67-82.

Necasky, M. 2009. Reverse engineering of XML schemas to conceptual diagrams, *Australian Computer Society*, Inc., Darlinghurst, Australia, 117−128.

Nelson, M. L. 1996. A Survey of Reverse Engineering and Program Comprehension, in *ODU CS 551 – Software Engineering Survey*, 1-8.

Nemuraite, L.; Ceponiene, L.; Vedrickas, G. 2008. Representation of Business Rules in UML/OCL Models for Developing Information Systems, *Lecture Notes in Business Information Processing*. Berlin: Springer Berlin Heidelberg, 182−196.

Nielson, F.; Nielson, H. R.; Hankin, C. 2004. *Principles of Program Analysis*, Corrected edn, Berlin: Springer.

OMG, 2008. *Semantics of Business Vocabulary And Business Rules v1.0*, Available from Internet: <http://www.omg.org/spec/SBVR/1.0/>.

OMG, 2009. *Production Rule Representation*, [cited 1 October 2011], Available from Internet: <http://www.omg.org/spec/PRR/1.0/PDF>.

OMG, 2011a. *Knowledge Discovery Metamodel Specification Version 1.3*, Available from Internet: <http://www.omg.org/spec/KDM/1.3/PDF/>.

OMG, 2011b. *Business Process Modeling Notation v1.0*, Available from Internet: <http://www.omg.org/spec/BPMN/2.0/PDF/>.

OMG, 2012. *Architecture Driven Modernization Task Force*, Available from Internet: <http://adm.omg.org>.

Paradauskas, B.; Laurikaitis, A. 2006. Business Knowledge Extraction from Legacy Information Systems, *Information Technology And Control* 35(3): 214−221.

Paradauskas, B.; Laurikaitis, A. 2011. Extracting conceptual data specifications from legacy information systems, *Electronics and Electrical Engineering* 1(107): 46−50.

Perez-Castillo, R.; Cruz-Lemus, J.; Guzman, I.; Piattini, M. 2012. A family of case studies on business process mining using MARBLE, *Journal of Systems and Software*, 85(6): 1370−1385.

Perez-Castillo, R.; Cruz-Lemus, J. A.; Rodriguez, I. G.; Piattini, M. 2011a. Business process archeology using MARBLE, *Information & Software Technology* 53(10): 1023−1044.

Perez-Castillo, R.; de, I. G-R.; Piattini, M. 2011b. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems, *Computer Standards & Interfaces* 33(6): 519−532.

Perez-Castillo, R.; Guzman, R.; Garcia, O.; Piattini, M. 2009. MARBLE: a modernization approach for recovering business processes from legacy systems, *International Workshop on Reverse Engineering Models from Software Artifacts*, 671–676.

Perez-Castillo, R.; Weber, B.; de, I. G-R.; Piattini, M. 2011c. Toward Obtaining Event Logs from Legacy Code, in *M Muehlen, J Su (eds.), Business Process Management Workshops*. Berlin: Springer Berlin Heidelberg, 201–207.

Putrycz, E.; Kark, A. 2007. Recovering Business Rules from Legacy Source Code for System Modernization, in *A Paschke, Y Biletskiy (eds.), Advances in Rule Interchange and Applications*. Berlin: Springer Berlin Heidelberg, 107–118.

Putrycz, E.; Kark, A. 2008. Connecting Legacy Code, Business Rules and Documentation, in *N Bassiliades, G Governatori, A Paschke (eds.), Rule Representation, Interchange and Reasoning on the Web*. Berlin: Springer Berlin Heidelberg, 17–30.

Rodrigues, N. F.; Barbosa, L. S. 2010. Slicing for architectural analysis, *Science of Computer Programming* 75(10): 828−847.

Ross, R. G. 2003. *Principles of the Business Rule Approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, USA.

Sartipi, K. 2003. Pattern-based Software Architecture Recovery, in *Proceedings of the Second ASERC Workshop on Software Architecture*, Banff Center, Alberta, Canada, 1-7.

Sartipi, K.; Yousefi, A. 2010. Scenario-driven Model Transformation in Reverse, *ACT Transactions on Software Engineering and Methodology*, 645-652.

Seacord, R. C.; Plakosh, D.; Lewis, G. A. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*, Boston: Addison-Wesley Longman Publishing.

Skersys, T.; Tutkute, L.; Butleris, R.; Butkiene, R. 2012. Extending BPMN business process model with SBVR business vocabulary and rules, *Information technology and control* 41(4): 356−367.

Sneed, H. M. 2001. Extracting business logic from existing COBOL programs as a basis for redevelopment, in *Proceedings of 9th internation workshop on Program Comprehension*, Toronto, Ontario, Canada, 167−175.

Sneed, H. M.; Erdos, K. 1996. Extracting Business Rules from Source Code, *IEEE Computer Society*, Washington, USA, 240–246.

Sosunovas, S.; Vasilecas, O. 2007. Tool-supported method for the extraction of OCL from ORM models, in *Proceedings of 10th International Conference on Business Information Systems*, Lecture Notes in Computer Science, Vol. 4439. Berlin: Springer-Verlag, 449−463.

Tip, F. 1995. A Survey of Program Slicing Techniques, *Journal of Programming Languages* 3: 121–189.

Tutkute, L.; Butleris, R.; Uzdanavičiutė, V.; Sinkevicius, E.; Skersys, T.; Kapocius, K. 2013. Improving quality of business models using a business vocabulary-based synchronization method, *Electronics and Electrical Engineering* 19(6): 125−130.

Tzerpos, V. 1998. Software botryology: Automatic clustering of software systems, in *Proceedings of 9th International Workshop on Database and Expert Systems Applications*, Vienna, Austria, 811–818.

Ulrich, W. M.; Newcomb, P. 2010. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*, San Francisco: Morgan Kaufmann Publishers Inc.

Wang, X.; Lai, G.; Liu, C. 2009. Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering, *Electronic Notes in Theory Computer Science* 243:121−137.

Wang, X.; Sun, J.; Yang, X.; He, Z.; Maddineni, S. 2004a. Application of information-flow relations algorithm on extracting business rules from legacy code, *Intelligent Control and Automation*. WCICA 2004. Fifth World Congress on, 3055–3058.

Wang, X.; Sun, J.; Yang, X.; He, Z.; Maddineni, S. 2004b. Business Rules Extraction from Large Legacy Systems, *IEEE Computer Society*, Washington, USA, 249−258.

Warmer, J.; Kleppe, A. G. 1998. *The Object Constraint Language: Precise Modeling with UML*, 1st edn. Boston: Addison-Wesley Professional.

Warmer, J.; Kleppe, A. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd edn. Boston: Addison-Wesley Longman Publishing Co., Inc.

Weiden, M.; Hermans, L.; Schreiber, G. ZS. 2002. Classification and Representation of Business Rules, in *Proceedings of European Business Rules Conference*, 1-14.

Weiser, M. 1981. Program slicing, *ICSE '81 Proceedings of the 5th international conference on Software*, IEEE Press, Piscataway, USA, 439−449.

Wiggerts, T. A. 1997. Using Clustering Algorithms in Legacy Systems Remodularization, *IEEE Computer Society*, Washington, USA, 33−45.

XText, 2013. *Xtext 2.4.3 Documentation*, [cited 30 October 2013], Available from Internet: <http://www.eclipse.org/Xtext/documentation/2.4.3/Documentation.pdf>.

Zou, Y.; Hung, M. 2006. An Approach for Extracting Workflows from E-Commerce Applications, in *IEEE International Conference on Program Comprehension*, IEEE Computer Society, Washington, USA, 127−136.

Zou, Y.; Lau, TC.; Kontogiannis, K.; Tong, T.; Mckegney, R. 2004. Model-driven business process recovery, in *11th Working Conference on Reverse Engineering*, Delft, Netherlands, 224−233.

The List of Author's Scientific Publications on the Subject of the Dissertation

Publications in peer reviewed periodical scientific journals

Normantas, K.; Vasilecas, O. 2013. A systematic review of methods for business knowledge extraction from existing software Systems. *Baltic Journal of Modern Computing,* 1(1–2): 29–51. ISSN 2255-8942.

Normantas, K.; Vasilecas, O. 2012a. Business rules discovery from existing software Systems. *International journal of scientific & engineering Research,* 3(10): 1–7. ISSN 2229-5518. (EBSCO, DOAJ)

Normantas, K.; Vasilecas, O. 2009a. Modelling of the business rules using UML/OCL. *Innovative Technologies for Science, Business and Education,* 1(6): 2.1–2.10. ISSN 2029-1035.

Publications in other editions

Normantas, K.; Vasilecas, O. 2012b. Extracting business rules from existing enterprise software systems. *Information and software technologies: 18th International Conference. Communications in computer and information science,* Vol. 319. New York: Springer, 482–496. (ISI Proceedings)

Normantas, K.; Sosunovas, S.; Vasilecas, O. 2012c. An overview of the knowledge discovery meta-model. *Proceedings of the 13th international conference on Computer Systems and Technologies: CompSysTech'13*. New York: ACM, 52–57.

Vasilecas, O.; Normantas, K. 2011. Deriving business rules from the models of existing information systems. *Proceedings of the12th International Conference on Computer Systems and Technologies: CompSysTech'12*. New York: ACM, 95–100. (Scopus, ACM Digital Library)

Vasilecas, O.; Normantas, K. 2010a. Decision table based approach for business rules modelling in UML/OCL. *Proceedings of the 11th International Conference on Computer Systems and Technologies: CompSysTech'10*. ACM international conference proceedings series, Vol. 471. New York: ACM Press, 77–82.

Normantas, K,; Vasilecas, O. 2010b. Business rules approach to e-learning systems development. *Proceedings of the 6th International Conference on E-learning and the Knowledge Society: e-Learning'10*. No 6. Riga: RTU, 113–118.

Normantas, K.; Vasilecas, O.; Sosunovas, Sergejus. 2009b. Defining well-formedness constraints with OCL. *Information technologies'2009: proceedings of the 15th international conference on Information and Software Technologies*. Kaunas: Technologija, 355–364. (ISI Proceedings)

Normantas, K.; Vasilecas, O.; Sosunovas, S. 2009c. Augmenting UML with decision tables technique. *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing: CompSysTech'09*. The Academic Society of Computer Systems and Information Technology. No 71. Varna: ACMBUL&UAI, 21.1–21.6.

Normantas, K.; Vasilecas, O.; Sosunovas, S. 2009d. Management of business rules using decision tables. *Information technologies'2009: 15th international conference on Information and Software Technologies*. Kaunas: Technologija, 144–150.

Normantas, K.; Vasilecas, O. 2009e. On expressing business rules with combination of UML and OCL. *Information technologies'2009: 15th international conference on Information and Software Technologies*. Kaunas: Technologija, 135–143.

Normantas, K.; Vasilecas, O. 2009f. OCL as a specification language for business rules. *Akademinio jaunimo siekiai: ekonomikos, vadybos ir technologijų įžvalgos: 6-oji studentų mokslinė konferencija*. Klaipėda: Klaipėdos universiteto leidykla, 152–157.

Normantas, K. 2009g. Decision tables for the specification of business rules in object-oriented analysis and design. *Respublikinė jaunųjų mokslininkų konferencija „Fundamentiniai tyrimai ir inovacijos mokslų sandūroje“*. [Republic youth scientists conference: Fundamental Research and Innovations in the Joint of Sciences]. Klaipėda: Klaipėdos universiteto leidykla, 57–65.

ANNEXES

Annex A. Well-formedness Constraints for Code Model

This Annex presents a set of well-formedness constraints for verifying and validating the Knowledge Discovery Meta-model (KDM) Code models (See Chapter 2). These constraints are applied to specific KDM elements, defined by context of OCL invariants.

Table A.1. Well-formedness constraints for representing Modules

#	*Constraint*	*OCL*
M1	Module class and its subclasses should not own SourceRef elements.	**context** Module **inv** M1_ModuleHasNoSourceRef: **self**.source->isEmpty()
M2	Code Model cannot directly own any code elements other than the subclasses of the Module class.	**context** Model **inv** M2_CodeModelCanOwnOnlyModule: **self**.codeElement.oclIsKindOf(code::Module)

Table A.1. (Continued)

M3	Every code element should be owned by some instance of the Module class or its subclasses. No other code element should own Module elements and its subclasses.	**context** AbstractCodeElement **inv** M3_ShouldBeOwnedByModule: **not self**.oclIsKindOf(code::Module) implies **self** -> closure(e | e.getOwner()) -> exists(o | o.oclIsKindOf(code::Module)) Note: closure is transitive; getOwner() – is the method of KDM Framework (see KDM, 10).
M4	Instance of the Module element should have at least one stereotype.	**context** Module **inv** M4_MustBeStereotyped: **self**.stereotype->notEmpty() Note: stereotypes are separately defined within Extentensions container;

Table A.2. Well-formedness constraints for representing Procedures

#	*Constraint*	*OCL*
CE1	ControlElement should have at least one stereotype	**context** ControlElement **inv** CE1_ShouldHaveStereotype: **self**.stereotype->notEmpty()
CE2	ControlElement should own a Signature.	**context** ControlElement **inv** CE2_MustOwnSignature: **self**.type.oclIsTypeOf(code::Signature)

Table A.3. Well-formedness constraints for representing Statements

#	*Constraint*	*OCL*
SC1.1	ActionElement kind of Assign must have single Reads relationship to a DataElement representing the value	**context** ActionElement **inv** SC11_AssignInput: **self**.kind='Assign' **implies self**.actionRelation -> one (r | r.oclIsTypeOf(action::Reads) **and** r.oclAsType(action::Reads).to.oclIsKindOf(code::DataElement))
SC1.2	ActionElement kind of Assign must have single Writes relationship to a DataElement, except a ValueElement, representing the value	**context** ActionElement **inv** SC12_AssignOutput: **self**.kind='Assign' **implies self**.actionRelation -> one (r | r.oclIsTypeOf(action::Writes) **and** r.oclAsType(action::Writes).to.oclIsKindOf(code::DataElement) **and not** r.oclAsType(action::Writes).to.oclIsTypeOf(code::ValueElement))
SC1.3	ActionElement kind of Assign may have single Flow relationship to another ActionElement	**context** ActionElement **inv** SC13_AssignOptFlowToNext: **let** flows: Collection(action::Flow) = **self**.actionRelation ->select(ar | ar.oclIsTypeOf(action::Flow)).oclAsType(action::Flow) **in self**.kind='Assign' **and** flows -> size() > 0 **implies** flows -> forAll(f | f.to <> **self**)
SC2.1	ActionElement kind of Condition must have single Reads relationship to a DataElement repre-	**context** ActionElement **inv** SC21_CondInput: **self**.kind='Condition' **implies self**.actionRelation -> one (ar | ar.oclIsTypeOf(action::Reads) **and** ar.oclAsType(action::Reads).to.type.oclIsTypeOf(code::BooleanType))

Table A.3. (Continued)

#	Constraint	OCL
	senting the Boolean value	
SC2.2	ActionElement kind of Condition must have TrueFlow and FalseFlow to another ActionElements	**context** ActionElement **inv** SC21_CondInput: **self**.kind='Condition' **implies self**.actionRelation -> one(ar | ar.oclIsTypeOf(action::TrueFlow) **and** ar.oclAsType(action::TrueFlow)<>**self**) **and self**.actionRelation -> one(ar | ar.oclIsTypeOf(action::FalseFlow) **and** ar.oclAsType(action::FalseFlow)<>**self**)
SC3.1	ActionElement kind of Switch must have single Reads relationship to DataElement representing selector value and none of Writes relationships.	**context** ActionElement **inv** SC31_SwitchIO: **self**.kind='Switch' **implies self**.actionRelation -> one (r | r.oclIsTypeOf(action::Reads) **and** r.oclAsType(action::Reads).to.oclIsKindOf(code::DataElement)) **and not self**.actionRelation -> exists(r | r.oclIsTypeOf(action::Writes))
SC3.2	ActionElement kind of Switch must have at least one GuardedFlow relation to an ActionElement kind of Guard. A single FalseFlow relationship represent default branch.	**context** ActionElement **inv** SC32_SwitchFlows: **self**.kind='Switch' **implies self**.actionRelation -> select (r | r.oclIsKindOf(action::GuardedFlow)) -> size() >= 1 **and self**.actionRelation -> select (r | r.oclIsKindOf(action::FalseFlow)) -> size() <= 1
SC3.3	ActionElement kind of Guard must have single Reads relationship to DataElement representing the guard value.	**context** ActionElement **inv** SC33_GuardInput: **self**.kind='Guard' **implies self**.actionRelation -> one(r | r.oclIsKindOf(action::Reads))
SC3.4	ActionElement kind of Guard must have single Flow relationship the next ActionElement	**context** ActionElement **inv** SC34_GuardFlow: **self**.kind='Guard' **implies self**.actionRelation -> one(r | r.oclIsKindOf(action::Flow))

Table A.4. Well-formedness constraints for representing Expressions

#	Constraint	OCL
EC1.1	ActionElement which kind is one of the Binary Comparison action kinds {'Equals', 'NotEqual', 'LessThenOrEqual',	**context** ActionElement **inv** EC11_CompAEInput: **if self**.kind = 'Not' **then** **self**.actionRelation -> one (r | r.oclIsTypeOf(action::Reads) **and** r.oclAsType(action::Reads).to.type.oclIsTypeOf(code::BooleanType)) **else** **if Set{**'Equals', 'NotEqual', 'LessThenOrEqual', 'LessThan',

Table A.4. (Continued)

#	*Constraint*	*OCL*
	'LessThan', 'GreaterThan', 'GreaterThanOrEqual', 'Not', 'And', 'Or', 'Xor'} must have two Reads relationships to DataElements representing values of the same data type. Except is Unary Comparison kind {Not}, which has a single Reads relationship.	'GreaterThan', 'GreaterThanOrEqual', 'Not', 'And', 'Or', 'Xor'} -> exists(bop | bop = **self**.kind) **then** **self**.actionRelation -> select(r | r.oclIsTypeOf(action::Reads)) -> size() = 2 **and self**.actionRelation -> select(r | r.oclIsTypeOf(action::Reads)).oclAsType(action::Reads) -> forAll(r1,r2| r1.to.type = r2.to.type) **else true** **endif** **endif**
EC1.2	ActionElement which kind is one of Comparison action kinds may have Writes relationship to DataElement of a BooleanType.	**context** ActionElement **inv** EC12_CompAEOutput: **Set{**'Equals', 'NotEqual', 'LessThanOrEqual', 'LessThan', 'GreaterThan', 'GreaterThanOrEqual', 'Not', 'And', 'Or', 'Xor','Not'**}** -> exists(bop | bop = **self**.kind) **and** **self**.actionRelation -> select(r | r.oclIsTypeOf(action::Writes)) -> size() > 0 **implies** **self**.actionRelation -> one (r | r.oclIsTypeOf(action::Writes) **and** r.oclAsType(action::Writes).to.type.oclIsTypeOf(code::BooleanType))

EC2.1	ActionElement which kind is one of the Binary Arithmetical action kinds {'Add', 'Multiply', 'Substract', 'Divide', 'Remainder'} must have two Reads relationships to DataElements representing values of the same data type. ActionElement which kind is one of the Unary Arithmetical kinds {'Negate','Successor'} must have single Reads relationship.	**context** ActionElement **inv** EC21_ArithAEInput:
		if Set{'Negate','Successor'**}** -> exists(uop | uop = **self**.kind)
		then
		self.actionRelation -> one (r | r.oclIsTypeOf(action::Reads))
		else
		if Set{'Add','Multiply','Substract','Divide','Remainder'**}** -> exists(bop | bop = **self**.kind) **then**
		self.actionRelation -> select(r | r.oclIsTypeOf(action::Reads)) -> size() = 2 **and**
		self.actionRelation -> select(r | r.oclIsTypeOf(action::Reads)).oclAsType(action::Reads) -> forAll(r1,r2| r1.to.type = r2.to.type)
		else true
		endif
		endif

Table A.6. Well-formedness constraints for representing Variables, Values, and PrimitiveTypes

#	*Constraint*	*OCL*
DC1	DataElement class instance should have at least one Stereotype.	**context** ActionElement **inv** DC1_ShouldHaveStereotype: **self**.stereotype->notEmpty()
VC1	ValueElement and its subclasses should not have owned code elements.	**context** ValueElement **inv** VC1_ValueCanotHaveOwnedCodeElems: **self**.codeElement->isEmpty()
VC2	ValueElement and its subclasses cannot be used as the target of relations Writes.	**context** Writes **inv** VC2_ValueCanotBeWritten: **not self**.to.oclIsTypeOf(code::ValueElement)
VC3	ValueElement class instance should have at least one Stereotype.	**context** ValueElement **inv** VC3_ShouldHaveStereotype: **self**.stereotype->notEmpty()
TC1	PrimitiveType should have at least one stereotype.	**context** PrimitiveType **inv** TC1_ShouldHaveStereotype: **self**.stereotype->notEmpty()

Annex B. Well-formedness Constraints for Data Model

This Annex presents a set of well-formedness constraints for verifying and validating KDM Data models (See Chapter 2). These constraints are applied to specific KDM elements, defined by context of OCL invariants.

Table B.1. Well-formedness constraints for representing Data model elements

#	*Constraint*	*OCL*
DR	DataResource should have at least one stereotype.	**context** DataResource **inv** DR_ShouldHaveStereotype: **self**.stereotype->notEmpty()
IE1	Index owned by a data element should group elements that are owned by that data element.	**context** IndexElement **inv** IE2_TheSameOwner: **self**.implementation.getOwner()=**self**.getOwner()
IE2	IndexElement should have a stereotype.	**context** IndexElement **inv** IE2_ShouldHaveStereotype: **self**.stereotype->notEmpty()
UK	UniqueKey owned by a data element should group ItemUnit elements that are owned by that data element.	**context** IndexElement **inv** UK_TheSameOwner: **self**.implementation.getOwner()=**self**.getOwner()

Table B.1. (Continued)

RK	ReferenceKey owned by a data element should group ItemUnit elements that are owned by that data element.	**context** IndexElement **inv** RK_TheSameOwner: **self**.implementation.getOwner()=**self**.getOwner()
I1	Index owned by a data element should group ItemUnit elements that are owned by that data element.	**context** Index **inv** I1_TheSameOwner: **self**.implementation.getOwner()=**self**.getOwner()
DCR	Relationship should not be used in Code models.	**context** AbstractDataRelationship **inv** DCR_CannotBeUsedInCodeModel: **not self**.getModel()->oclIsKindOf(code::CodeModel)

Annex C. Cypress Enabled Script Grammar

This annex presents specification of grammar used to generate parser for parsing Cypress Enabled script modules (Visual Basic for Application dialect). This grammar is defined in the XText grammar definition language which is similar to EBNF. The XText itself is based on another parser generator ANTLR; however, in opposite to ANTLR, it allows to directly generate abstract syntax tree (AST) instances in XMI format.

```
grammar lt.vgtu.isl.bkess.CypressEnabled
hidden(WS,LC,SP_COMMENT,REM_COMMENT)
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate cypressEnabled "http://www.vgtu.lt/isl/bkess/CypressEnabled"
Module:
        member+=Member*;
//not every possible member is included there, only those that were inspeted during analysis
Member:
        LineTerminatorOpt
        (OptionStatement | GlobalConstDeclaration | LibFuncOrSubDeclaration | FunctionDeclaration |
SubDeclaration);
OptionStatement: {OptionStatement}
        'Option' 'Explicit' explicitOption=('On' | 'Off')? LineTerminatorOpt;
GlobalConstDeclaration:
        'Const' name=ID EQUALS value=(Literal|ConcatExpression) LineTerminatorOpt;
LibFuncOrSubDeclaration:
        'Declare' ('Function'|'Sub') name=ID 'Lib' library=StringLiteral 'Alias' alias=StringLiteral ('('
        parameters=ParametersList')')? ('As' type=Type)?;
FunctionDeclaration:
        signature=FunctionSignature LineTerminatorMnd
        body=Block? 'End' 'Function' LineTerminatorOpt ;
FunctionSignature:
        'Function' name=ID '(' (parameters=ParametersList)? ')' ('As' type=Type)?;
ParametersList:
        parameter+=Parameter (',' parameter+=Parameter)* ;
Parameter:
        ('ByVal'|'ByRef')? name = ID (isArray?='('')')? ('As' type=Type)? ;
SubDeclaration:
        signature=SubSignature LineTerminatorMnd body=Block? 'End' 'Sub' LineTerminatorOpt;
SubSignature:
        'Sub' name=ID '('(parameters=ParametersList)? ')';
Block: {Block}
        statements+=Statements*;
Statements:
        statement+=Statement (':' statement+=LineStatement)* LineTerminatorMnd ;
```

```
//not every possible statement is included here, only those that were identified during analysis
Statement:
          LabelDeclarationStatement | LocalDeclarationStatement | AssignmentStatement | InvocationStatement
| ConditionalStatement | LoopStatement | BranchStatement | ArrayHandlingStatement;
LineStatements:
          statement+=LineStatement (':' statement+=LineStatement)*;
LineStatement:
          LocalDeclarationStatement| AssignmentStatement | InvocationStatement |BranchStatement | Array-
HandlingStatement;
LocalDeclarationStatement:
          LocalConstDeclaration | VariableDeclaration;
LocalConstDeclaration:
          'Const' name=ID EQUALS value=(Literal|ConcatExpression);
VariableDeclaration:
          VariableDeclarationDim | VariableDeclarationNoDim;
VariableDeclarationDim:
          'Dim' var+=Declarator (',' var+=Declarator)* ;
VariableDeclarationNoDim:
          var+=Declarator (',' var+=Declarator)* ('As' type=Type);
Declarator:
          (ArrayRef | VariableDeclarator) ('As' type=Type)?;
ArrayRef:  name=ID '(' arguments=ArgumentsList ')';
VariableDeclarator:   name=ID;
LabelDeclarationStatement:       label=ID ':';
AssignmentStatement:
          'Set'? assignable=Assignable EQUALS assignmentExp = Expression //LineTerminator;
Assignable:
          ArrayRef | AccessExpression;
//call InvocationExpression: Just a name of a sub or function,  object member access with/without return and
with/without passed arguments
InvocationStatement:
          {InvocationStatement}
          'Call'? expression=AccessExpression //LineTerminator;
ConditionalStatement:
          IfStatement |
          SelectStatement;
IfStatement:
          BlockIfStatement |
          LineIfStatement;
//logicalExpression
BlockIfStatement:
          'If' condition=LogicalExpression 'Then' LineTerminatorMnd
           thenPart=Block?
           elseIfPart+=ElseIfStatement*
           elsePart=ElseStatement?
          'End' 'If';
ElseIfStatement:
          'ElseIf' elseIfCondition = LogicalExpression 'Then' LineTerminatorMnd
           elseIfBlock=Block?;
ElseStatement:
          'Else' LineTerminatorMnd
            elseBlock=Block?;
LineIfStatement:
          'If' condition=LogicalExpression 'Then' thenPart=LineStatements ('Else' else-
Part=LineStatements)?;
SelectStatement:
          'Select' 'Case' caseExp = Expression LineTerminatorMnd
           (caseStatement+=CaseStatement)*
           (caseElseStatement=CaseElseStatement)?
          'End' 'Select';
CaseStatement:
          'Case' clauses+=(ConcatExpression|PrimaryExpression) (',' clau-
ses+=(ConcatExpression|PrimaryExpression))* LineTerminatorMnd
          block=Block?;
CaseElseStatement:
          'Case' 'Else' LineTerminatorMnd
           block=Block?;
LoopStatement:
          DoLoopStatement |
          WhileStatement  |
          ForEachStatement |
          ForStatement ;
WhileStatement:
          {WhileStatement}
          'While' condition=LogicalExpression LineTerminatorMnd
```

```
            block=Block?
           'Wend';
DoLoopStatement:
           DoTopLoopStatement |
           DoBottomLoopStatement;
DoTopLoopStatement:
           {DoTopLoopStatement}
           'Do' ('While'|'Until') loopCondition=LoopCondition LineTerminatorMnd
            block=Block?
           'Loop';
LoopCondition:
           {LoopCondition}
           exp=LogicalExpression;
DoBottomLoopStatement:
           {DoBottomLoopStatement}
           'Do' LineTerminatorMnd
            block=Block?
           'Loop' ('While'|'Until') loopCondition=LoopCondition;
ForStatement:
           {ForStatement}
           'For' loopCondition = LogicalExpression 'To' upperBound=AdditiveExpression ('Step'
step=AdditiveExpression)? LineTerminatorMnd
            block=Block?
           'Next' (loopNextCounter=PrimaryExpression)?;
ForEachStatement:
           {ForEachStatement}
           'For' 'Each' collectionMember=Expression 'In' collection=Expression LineTerminatorMnd
            block=Block?
           'Next';
BranchStatement:
           OnErrorResumeNextStatement
           | OnErrorGotoStatement
           | GotoStatement
           | ExitStatement
           | ContinueStatement
           | StopStatement ;
GotoStatement:
           {GotoStatement}
           'GoTo' label=Identifier //LineTerminator;
OnErrorGotoStatement:
           {OnErrorGotoStatement}
           'On' 'Error' 'GoTo' label=Identifier //LineTerminator;
OnErrorResumeNextStatement:
           {OnErrorResumeNextStatement}
           'On' 'Error' 'Resume' 'Next' //LineTerminator;
ExitStatement:
           {ExitStatement}
           'Exit' (kind=('Do' | 'For' | 'While' | 'Select' | 'Sub' | 'Function'))? //LineTerminator;
ContinueStatement:
           {ContinueStatement}
           'Continue' (kind=('Do' | 'For' | 'While'))? //LineTerminator;
StopStatement:
           {StopStatement}
           'Stop' //LineTerminator;
ArrayHandlingStatement:
           RedimStatement | EraseStatement;
RedimStatement:
           {RedimStatement}
           'ReDim' array=ArrayRef;
EraseStatement:
           {EraseStatement}
           'Erase' array=ArrayRef;
Expression:
           ParameterAssignExp | LogicalExpression | PrimaryExpression;
LogicalExpression returns Expression:
           BooleanAndExpression ({LogicalExpression.left=current} op='Or' right=BooleanAndExpression) *;
BooleanAndExpression returns Expression:
           EqualityExpression ({BooleanAndExpression.left=current} op='And' right=EqualityExpression)*;
EqualityExpression returns Expression:
           IsNothingExpression | RelationalExpression ({EqualityExpression.left=current}
op=(EQUALS|NOTEQUALS) right=RelationalExpression)*;
RelationalExpression returns Expression:
           AdditiveExpression ({RelationalExpression.left=current} op=(LT|LTEQ|GT|GTEQ)
right=AdditiveExpression)*;
AdditiveExpression returns Expression:
```

```
            MultiplicativeExpression ({AdditiveExpression.left=current} op=(PLUS|MINUS)
right=MultiplicativeExpression)* ;
MultiplicativeExpression returns Expression:
            PowerExpression ({MultiplicativeExpression.left=current} op=(MULT|DIV|MOD)
right=PowerExpression)*;
PowerExpression returns Expression:
            UnaryExpression ({PowerExpression.left=current} op=POW right=UnaryExpression)*;
UnaryExpression returns Expression:
            ConcatExpression | NegationExpression | MinusExpression | PrimaryExpression;
NegationExpression returns Expression:
            {NegationExpression}
            'Not' exp=LogicalExpression;
MinusExpression returns Expression:
            {MinusExpression}
            MINUS exp=PrimaryExpression;
PrimaryExpression returns Expression:
            ParenthesizedExpression | ConvertExpression | StringFunctionExpression | MathFunctionExpression |
ObjectExpression | AccessExpression | Literal;
ParenthesizedExpression returns Expression: {ParenthesizedExpression}
            '(' expression = Expression ')';
AccessExpression:     MemberAccessExpWithParenAndMndArgs | MemberAccessExpWithParenAndOptArgs |
MemberAccessExpWithoutParen | MemberAccessExpression;
MemberAccessExpression:          member+=Identifier ('.' member+=(MemberAccessExpWithParen|Identifier))*;
MemberAccessExpWithoutParen:     member+=Identifier ('.' member+=(MemberAccessExpWithParen|Identifier))*
argumentsList=ArgumentsList;
MemberAccessExpWithParen:        MemberAccessExpWithParenAndMndArgs | MemberAccessExpWithParenAndOptArgs;
MemberAccessExpWithParenAndOptArgs:         member+=Identifier ('.' mem-
ber+=(MemberAccessExpWithParen|Identifier))* '(' argumentsList=ArgumentsList? ')';
MemberAccessExpWithParenAndMndArgs:         member+=Identifier ('.' Member += (MemberAccessExpWithParen |
Identifier))* '(' argumentsList=ArgumentsList ')';
ArgumentsList:        argument+=ArgumentExpression (',' argument+=ArgumentExpression)*;
ArgumentExpression returns Expression:       Expression;
ConcatExpression:     concated+=PrimaryExpression ('&' concated+=PrimaryExpression)+;
IsNothingExpression:  obj = AccessExpression 'Is' value=NullableLiteral;
ConvertExpression:    functionName=ConvertFunction '(' arguments=ArgumentsList? ')';
StringFunctionExpression:        functionName=StringFunction '(' arguments=ArgumentsList ')';
MathFunctionExpression:          functionName=MathFunction '(' exp=Expression ')';
ObjectExpression:     functionName=ObjectFunction '(' arguments=ArgumentsList ')';
Literal:   NullableLiteral | StringLiteral | NumberLiteral | BooleanLiteral | HexLiteral;
NullableLiteral:value='Nothing';
NumberLiteral:value = (IntLiteral | DoubleLiteral) ;
DoubleLiteral:INT '.' INT;
IntLiteral:INT;
StringLiteral:value=STRING;
BooleanLiteral:value=('True'|'False');

HexLiteral:value=HEX;
//value should be assigned in subtypes
//in order to distinguish what types are used
Identifier:value = (MessageBoxFunction | ObjectFeature | ID);
//used when passing parameters to word document
ParameterAssignExp:parameter = ID ':=' value=Literal;
MessageBoxFunction:'MsgBox';
ConvertFunction:'Chr' | 'Hex' | 'Oct' | 'Str' | 'CDbl' | 'CInt' | 'Clng' | 'CSng' | 'CStr' | 'CVar' | 'CVDa-
te' | 'Asc' | 'Date' | 'DateSerial' | 'DateValue' | 'Format' | 'Fix' | 'Int' | 'Day' | 'Weekday' |'Month' |
'Year' | 'Hour' | 'Minute' | 'Second' | 'TimeSerial' | 'TimeValue';
StringFunction:'Let' |'Len' | 'InStr'| 'Left'| 'Left$'| 'Mid'| 'Asc'| 'Chr'| 'Right' |'LCase'| 'Ucase'|
'InStr'|'LTrim' |'RTrim'| 'Trim' |'Len' |'StrComp'| 'Format' | 'String';
MathFunction:'Exp' | 'Log' | 'Sqr' | 'Rnd' | 'Abs' | 'Sgn' | 'Atn' | 'Cos' | 'Sin' | 'Tan' | 'Int' | 'Fix';
ObjectFunction:'CreateObject' | 'GetObject';
ObjectFeature:Object | Property | Method;
Object:    'Dialog' | 'Mask' | 'Document' | 'Docflow';
Property:  'cursor';
//put here language infrastructure specific or platform API specific methods that are duplicated with gram-
mar
Method:'first()' | 'next()' | 'Date()' | 'Date' | 'Time' | 'End';
Type:(StringType | IntegerType | ObjectType | BooleanType | LongType | DoubleType | DateType | TimeType);
StringType: name='String';
IntegerType: name='Integer';
ObjectType: name='Object';
BooleanType: name='Boolean';
LongType: name='Long';
DoubleType: name='Double';
DateType: name='Date';
TimeType: name='Time';
```

```
LineTerminatorOpt:    NL*;
LineTerminatorMnd:    NL+;
terminal EQUALS: "=";
terminal NOTEQUALS: "<>";
terminal LT: '<';
terminal LTEQ: '<=';
terminal GT: '>';
terminal GTEQ:'>=';
terminal PLUS: '+';
terminal MINUS: '-';
terminal MULT: '*';
terminal DIV: '/';
terminal MOD: '%';
terminal POW: '^';
terminal SP_COMMENT: ("'") !('\n'|'\r')* ; //'\r'? '\n'?
terminal REM_COMMENT: ('Rem '|'rem '|'REM ') !('\n'|'\r')* ;      //'\r'? '\n'?
terminal LC: (' '|'_') (' '|'\t')* ('\r' '\n')? ('\n' '\r')?;
terminal WS: (' '|'\t')+;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal NL: ('\r' '\n')? ('\n' '\r')?;
terminal ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal STRING:'"' ("\u0020"|"\u0021"|"\u0022" "\u0022"|"\u0023".."\u007e"|"\u0100".."\u017e" | "\t")*
'"';
terminal HEX: '&H' ('a'..'f'|'A'..'F'|'0'..'9')+;
```

Annex D. Structured Query Language ECore Meta-model

This Annex presents several fragments of SQL meta-model created by simplifying the ISO/IEC SQL 2008 grammar. This meta-model was used to define grammar for creating parser that would allow parsing SQL expressions.

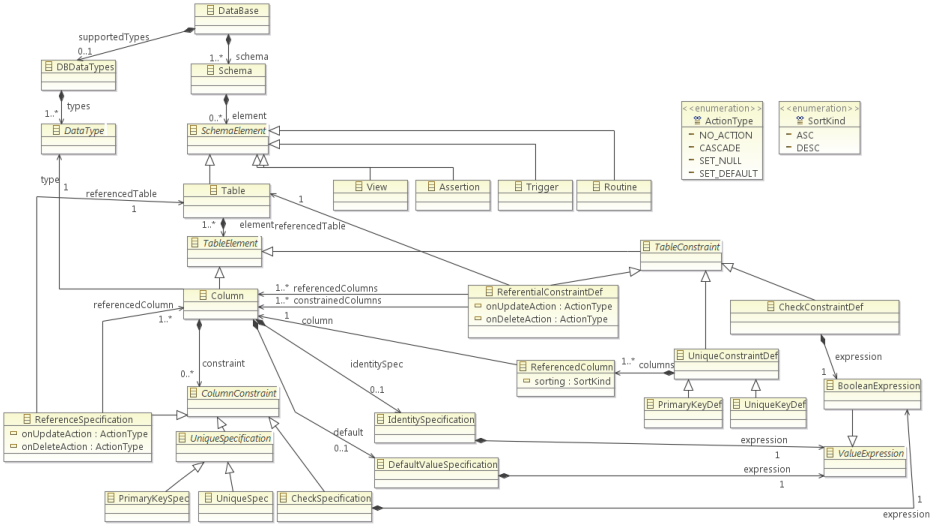

Fig. D.1. SQL Structural elements

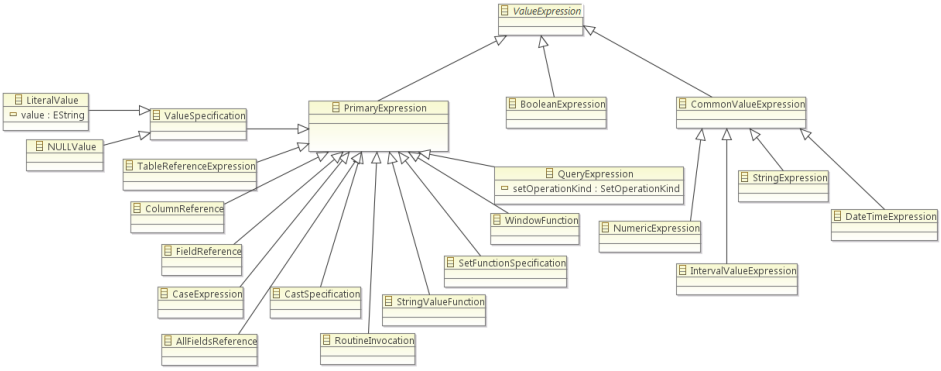

Fig. D.2. SQL Expressions

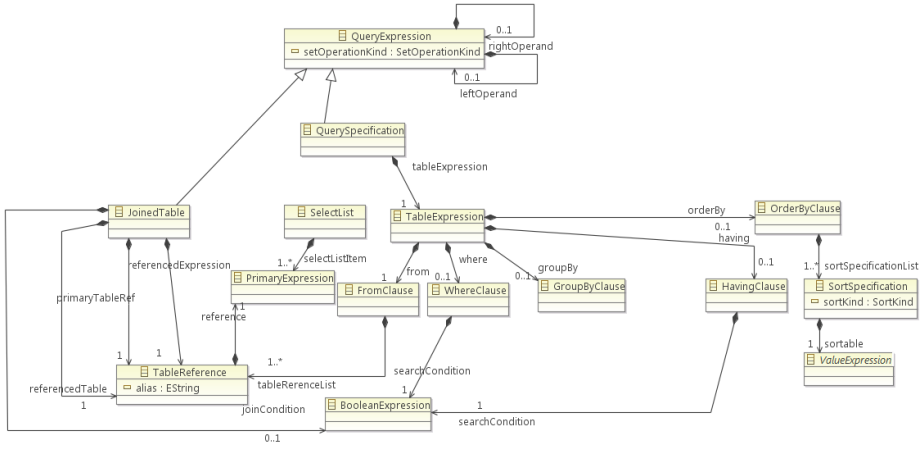

Fig. D.3. SQL Query Expression

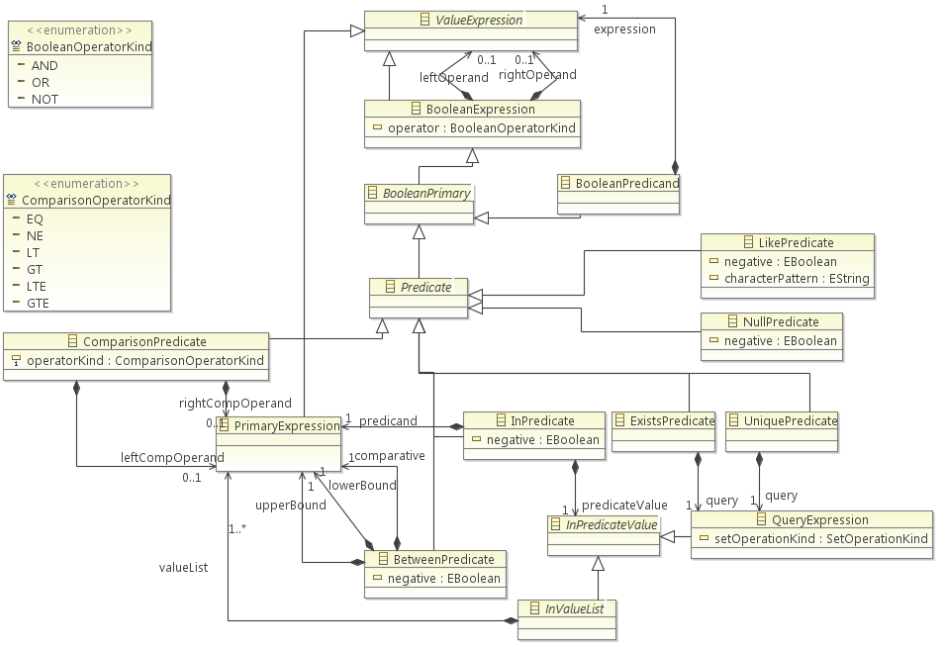

Fig. D.4. SQL Boolean Expression

Annex E. Implementation of Algorithms for Extraction of Business Vocabulary

cm: conceptual::ConceptualModel = ConceptualModel stereotyped by BusinessModel
vocabulary: conceptual::ConceptualContainer = ConceptualContainer stereotyped by Vocabulary
uiExtStereotypes: Collection(kdm::Stereotype) = Collection of stereotypes intended for ui elements that represent labeled data manipulation controls
dataExtStereotypes: Collection(kdm::Stereotype) = Collection of stereotypes intended for data elements that represent structural organization of data
contentExtStereotypes: Collection(kdm::Stereotype) = Collection of stereotypes intended for content elements that represent structural organization of data **in** Xml format
codeExtStereotypes: Collection(kdm::Stereotype) = Collection of stereotypes intended for code elements that represent types, global variables, members, **in**/out parameters
concExtStereotypes: Collection(kdm::Stereotype) = Colleciton of stereotypes intended for conceptual elements that represent extracting business knowledge
--helper function
--creates term unit for the KDMEntity and adds it to the vocabulary or returns corresponding TermUnit if it was already created for this entity
function TermUnit(impl:KDMEntity)

```
termUnits: Collection(conceptual::TermUnit) = conceptual::TermUnit.allInstances() -> col-
lect(ce|ce.implementation=impl)
if termUnits -> notEmpty() then
        return termUnits -> first()
else
        termUnit:TermUnit = TermUnit.newInstance()
        termUnit.implementation = impl
        if (impl.GetModel().oclIsTypeOf(ui::UIModel)) then
                termUnit.name = impl.name
        else
                termUnit.name = ExtractName(impl.name)
        endif
        return termUnit
endif
end function

function DiscoverTermUnits()
        uiElements : Collection(ui::AbstractUIElement) = ui::AbstractUIElement.allInstances() ->
select(u | u.name <> '' and uiExtStereotypes->includes(u.stereotype))
        dataElements : Collection(data::AbstractDataElemement) = data::AbstractDataElement() ->
select(d | d.name <> '' and dataExtStereotypes -> includes(d.stereotype))
        contentElements: Collection(data::AbstractContentElemement) = da-
ta::AbstractContentElement() -> select(c | c.name <> '' and contentExtStereotypes -> inclu-
des(c.stereotype))
        codeElements: Collection(code::AbstractCodeElemement) = code::AbstractCodeElement() -
> select(c | c.name <> '' and contentExtStereotypes -> includes(c.stereotype))
        for all e:core::KDMEntity in uiElements->union(dataElements)->union(contentElements)-
>union(codeElements)
                termUnit: conceptual::TermUnit = TermUnit(e)
                termUnit.stereotype = concExtStereotypes -> select( s | s.name = 'Vocabulary-
Entry' )
                termUnit.taggedValue.add(TaggedValue(conceptType,'Noun concept'))
                for all text:DocumentText in GetMatchingText(termUnit)
                        termUnit.taggedValue.add(TaggedValue(dictionaryBasis,text.Snippet))
                        termUnit.taggedValue.add(TaggedValue(source,text.Source))
                endfor
                vocabulary.add(termUnit)
        endfor
end function
function FactUnit(impl:KDMEntity, roles:OrderedSet(AbstractConceptualElement), kind:String)
        factUnit:FactUnit ← FactUnit.newInstance();
        att:Attribute ← Attribute.newInstance();
        att.tag ← 'kind'; att.value ← kind; factUnit.attribute ← att;
        not impl.OclIsUndefined() implies factUnit.implementation ← impl;
        for all role ∈ roles do
                concRole:ConceptualRole ← ConceptualRole.newInstance();
                concRole.implementation ← role;
                factUnit.conceptualElement ← concRole;
        end for
```

```
            return factUnit;
end function
Vocabulary extraction. Step 1
INPUT
segment:Segment ← a segment consisting of a set of models representing the system
GLOBAL
concModel:ConceptualModel ← ConceptualModel.newInstance();
segment.model ← concModel;
uniqueKeys:Collection(UniqueKey) ← UniqueKey.allInstances();
foreignKeys:Collection(ReferenceKey) ← ReferenceKey.allInstances();
fkRelations:Collection(KeyRelation) ← KeyRelation.allInstances();
step1:traverse data model and create term units and IsPropertyOf or Assortment fact units
for all dm ∈ segment.model -> select(m|m.oclIsTypeOf(DataModel)) do
 for all sc ∈ dm.dataElement -> select(s|s.oclIsTypeOf(RelationalSchema)) do
  for all rt ∈ sc.dataElement -> select(t|t.oclIsTypeOf(RelationalTable)) do
   --create term unit from table
   tblTu:TermUnit ← TermUnit(rt);
   --search for unconstrained columns
   for all col ∈ rt.itemUnit do
    if column is not constrained by unique/foreign key constrain
     --create term unit from table column
     --create is-property-of fact type and add it to conceptual model
      if uniqueKeys.implementation -> union(foreignKeys.implementation) -> exists(i | i <> col) then
       clmnTu:TermUnit ← TermUnit(col);
       concModel.conceptualElement ← FactUnit(tblTu,OrderedSet{tblTu,clmnTu},'IsPropertyOf');
      end if
    end for
    --create assortment fact type from check constraints
   for all ev ∈ rt.dataElement -> select(e|e.oclIsT ypeOf(DataEvent)) -> select(e|e.kind='Insert') do
     ae:ActionElement ← ev.abstraction;
    if ae.kind = 'Check' then
    if check constraint consits of InPredicate
     if ae.codeElement -> exists(ae|ae.kind='InPredicate') then
       inp:ActionElement ← ae.codeElement -> select(p|p.kind='InPredicate') -> first();
       iu:ItemUnit ← inp.actionRelation -> select(r|r.oclIsTypeOf(Read)) -> collect(r|r.from);
       --create container for value list
       for all v 2 ae.codeElement -> se
lect(vl|vl.oclIsTypeOf(ValueList)).oclAsType(ValueList).valueElement do
        concModel.conceptualElement ← FactUnit(ae,OrderedSet{iu, ConceptualEle-
        ment(v)},'Assortment');
      end for
     end if
    end if
   end for
  end for
 end for
end for
Vocabulary extraction. Step 2
step2 analyze foreign key relations and identify Specialization,Partitive,or Relationalship fact
```

```
units
for all rel ∈ fkRelations do
        pkTu:TermUnit ← TermUnit(rel.to.implementation);
        fkTu:TermUnit ← TermUnit(rel.from.implementation);
        if foreign key relation relates two primary keys
        if uniqueKeys.implementation -> exists(f | rel.from.implementation) then
                create specialization fact unit
                concModel.conceptualElement ← FactU-
nit(rel,OrderedSet{TermUnit(rel.from.implementation),
                TermUnit(rel.to.implementation)},'Specialization');
        else
                check if foreign key within key relation has attribute that specifies cascading
delete action,
                i.e. candidate to composition
                if rel.from.attribute -> select (a|a.tag='OnDeleteAction' and a.value='CASCADE')
-> notEmpty() then
                create Partitive fact unit
                concModel.conceptualElement ← FactUnit(rel,OrderedSet{pkTu,fkTu},'Partitive')
                else
                create relationship fact unit
                concModel.conceptualElement ← FactU-
nit(rel,OrderedSet{pkTu,fkTu},'Relationship')
                end if
        end if
end for
```

Annex F. List of Extensions for Dataflow Analysis

This Annex summarizes and describes a list of extensions used to compute equations for dataflow analysis. These extensions are implemented as derived properties (OCL), their formal definitions is presented in the Chapter 2.

Table F.1. A list of extensions implemented as derived ActionElement properties

ActionElement derived property	***Description***
genVars	Collection of data elements (variables) being read by the current action element
killVars	Collection of data elements being written by the current action element
pred	Collection of action elements preceding the current action element along the control flow
succ	Collection of action elements succeeding the current action element along the control flow
liveInVars	Collection of data elements being alive (i.e. not over/written) on entering the current action element along the control flow
liveOutVars	Collection of data elements being alive on exiting the current action

Table F.1. (Continued)

	element along the control flow
deadInVars	Collection of data elements being dead (i.e. over/written) on entering the current action element along the control flow
deadOutVars	Collection of data elements being dead on exiting the current action element along the control flow
genDefs	Collection of generated definitions compounded within the action element
killDefs	Collection of definitions being redefined within the compound action element
reachInDefs	Collection of definitions on entering the action element
reachOutDefs	Collection of definitions on exiting the action element
genExps	Collection of action elements representing expressions being read in the current action element
killExps	Collection of action elements representing expressions being overwritten in the current action element
availInExps	Collection of action elements representing expressions being alive on entering the current action element
availOutExps	Collection of action elements representing expressions being alive on exiting the current action element
domIn	Collection of action elements that dominates the current action element
domOut	Collection of action elements that the current action element dominates on
liveInterInVars	Collection of action elements representing expressions being alive in respect with inter-procedural summary flow information on entering the current action element
liveInterOutVars	Collection of action elements representing expressions being alive in respect with inter-procedural summary flow information on exiting the current action element

Table F.2. A list of extensions implemented as derived CallableUnit properties

CallableUnit derived property	***Description***
mayUseVars	Collection of data elements that may be used (read) within the body of procedure along some path of the control flow
mustUseVars	Collection of data elements that must be used within the body of procedure along all paths of the control flow
mayKillVars	Collection of data elements that may be killed (over/written) within the body of procedure along any path of the control flow
mustKillVars	Collection of data elements that must be killed (over/written) within the body of procedure along all paths of the control flow

Table F.3. A list of extensions implemented as ActionRelation stereotypes

ActionRelation stereotype	***Description***
Use-def	Use-def chain associates with each use of a variable, a list of statements containing definitions of the variable that reach the use.

Table F.3. (Continued)

Def-use	**From**: an action element which uses data element **To**: a set of action elements which defines data element and no other definition exists from the defining action element to the first action element of control flow ($ActionElement_{Start}$) Def-use chains can be computed by extending liveness analysis: the data flow information is a set of tuples <*x*,*n*> where *x* is Var and *n* is a basic block and it is assumed that each statement forms a basic block by itself.

RESEARCH ON BUSINESS KNOWLEDGE EXTRACTION FROM EXISTING SOFTWARE SYSTEMS

Doctoral Dissertation

Technological Sciences,
Informatics Engineering (07T)

Kęstutis NORMANTAS

VERSLO ŽINIŲ IŠGAVIMO IŠ EGZISTUOJANČIŲ PROGRAMŲ SISTEMŲ TYRIMAS

Daktaro disertacija

Technologijos mokslai,
informatikos inžinerija (07T)

2013 12 02. 13,5 sp. l. Tiražas 20 egz.
Vilniaus Gedimino technikos universiteto
leidykla „Technika“,
Saulėtekio al. 11, 10223 Vilnius,
http://leidykla.vgtu.lt
Spausdino UAB „Ciklonas“
J. Jasinskio g. 15, 01111 Vilnius