Business Rules Discovery from Existing Software Systems

Kestutis Normantas, Olegas Vasilecas

Abstract—Software maintenance consumes a large amount of its total life cycle costs. In fact, maintainers spend a lot of time analyzing source code, configurations and resource definitions referring to the documentation in order to gain a deeper understanding of the logic of business rules implemented in the system. To facilitate these activities, we propose a model-driven approach on business rules discovery from existing software systems. We describe the process for obtaining of standard based intermediate representation of knowledge about the software system and for abstraction of business logic from this representation. We believe that our on-going research on discovering business rules will decrease the efforts required for maintenance and evolution of software systems.

Index Terms—Business Knowledge Extraction, Business Rules Discovery, Knowledge Discovery Meta-Model, Architecture-Driven Modernization, Model-Driven Reverse Engineering

1 INTRODUCTION

According to the definition of IEEE [1], software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment. Software maintenance consumes a large amount of its total life cycle costs. Canfora and Cimile [18] reveal that the cost of maintenance consumes 60% to 80% of the total life cycle costs while Seacord et al. [13] observes that the relative cost for maintaining and evolving the software has been steadily increasing and reached more than 90 percent of the total cost.

Recently, the Object Management Group (OMG) within the Architecture-Driven Modernization (ADM) Task Force initiative [8] provides a number of standards [10] for representation and analysis of existing software systems in order to support modernization, including the maintenance, activities. The Knowledge Discovery Meta-model (KDM) [9] is the fundamental meta-model in this set of representations as it defines representation of all aspects of the software system and enables interoperability for tools that captures and analysis information about the existing system. A number of modernization projects [15] report significant cost savings by applying architecture-driven approaches in the modernization of large scale information systems.

Motivated by cost-effective modernization projects, we believe that by employing modern architecture-driven technologies with source code analysis techniques and a business rules (BR) approach, the costs of maintenance and evolution of information systems may be reduced significantly. In this paper we address related issues and propose an approach for the discovery of BR from the existing software systems. Our contributions are as follows. We describe reverse engineering activities that must be involved to build KDM representation of discovered knowledge about the software system. We define how various kinds of business rules may be implemented in the system, and analyze what source code analysis techniques could be applied to abstract the business logic from KDM representation.

The paper is structured as follows. We first of all, introduce an example software system that will be considered in the discussion of the approach. Then, we overview related work of BR extraction from the source code. After, we explain our approach by presenting the process of BR discovery. Finally, we provide conclusions and discuss further research.

1.1 An Example Software System

In order to present general ideas of the approach for business rules discovery, we refer to example enterprise resource planning (ERP) system. The ERP system is designed using multi-tiered Client/Server architecture (figure 1).

Fig. 1. Architecture of example Enterprise Resource Planning System
The system acts as a platform for development of business specific solutions: it provides customization capabilities by enabling to define specific data objects (over data definitions); using data definitions create forms and reports; specify workflows that may be assigned to data objects. It also provides application programming interface (API) allowing automation of particular system events using a dialect of the Visual Basic for Application (VBA) language and integration with other software systems using the Component Object Model (COM) interface. 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.

Such kind of software systems may be easily adapted to the business requirements; however, over time, requirements change and software system must be updated to reflect those changes. It leads to undocumented modifications of the system, and even worse, because the impact of these modifications to other parts of the system is very difficult to evaluate, they usually tend to be not fully tested. Maintenance and upgrade costs often exceed the cost to initially adapt the software system; therefore, the approach for automated comprehension of the business logic implemented in the software system is very important to reduce the maintenance costs.

Figure 2 depicts a simplified fragment of example ERP, emphasizing business rules embedded in the source code. The top side of the picture presents user interface: the main application dialog and the form “Order” opened over menu item “New order”. The bottom side of the picture shows the snippet of the form definition (XML) and the fragment of source code (VBA) implementing the business logic – calculation of a discount value.

2 Related Work

Numerous methods and techniques for BR discovery from existing software systems have been contributed in the field of reverse engineering. Chiang [3] presents an approach that uses static program slicing [[14]] of control flow graph (CFG)
of a program code to obtain code slices representing BR, and to transform them into the reusable CORBA components. Huang et al. [5] define a number of heuristic rules for domain variables identification, slicing criteria identification, and slicing algorithm selection. Code slices indicate BR implemented in the system. Three representations are considered: code-view – rules are represented as code fragments; formula-view – rules are represented as three parts formulae (left hand side for variable, right hand side for expression that modifies variable, and conditions under which modifications are applied); and input-output dependence view (bidirectional data flows between input and output parameters).

An extension to Huang et al. [5] solution is proposed by Wang et al. [16]. The approach proposed by them consists of five steps as follows. First, the program is sliced into multiple slices in order to be understandable for the further analysis. Then, two types of domain variables are identified: pure domain variables that represent system’s input and output; and derived domain variables that depend on pure domain variables. Dependency is established by applying information-flow relations computing algorithm proposed by Bergeretti and Carre [2]. Having extracted domain variables and their dependencies, the next step, called data analysis, identifies business items that are actually implemented in the selected slice. According to the obtained information, a set of business rules is extracted and represented using multiple views in order to be validated with stakeholders. Another improvement of Wang et al. work is proposed by Gang [4]. 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. 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. Putrycz and Kark [12] emphasize the fact that business analysts require more than just code snippets referring the business rules. For this reason, they propose an approach that use document (in HTML format) content extraction and key phrase analysis to link the source code implementing business rules with technical and other related documentation. To separate business processing logic from infrastructure related, they focus on single program statements that carry a business meaning, such as calculation and branching since they most often represents high level processing. Resulting production rules in the form of \(<\text{Condition}>\neq \text{Action}>\) are represented using business vocabulary and business rules (SBVR).

In contrast to related works, our research concentrates on software systems that are built using heterogeneous technologies, and aims to gather any kind of information about the software system to facilitate the comprehension of business logic implemented in the system. We therefore rely on the KDM standard to represent the knowledge about the software system and apply source code analysis to abstract the business logic.

### 3 The Approach for Business Rules Discovery in Existing Software Systems

The approach for business rules discovery in existing enterprise software systems is based on reverse engineering process that obtains intermediate representation of different aspects of the software system using the Knowledge Discovery Meta-Model (KDM). The process consists of the following phases: preliminary study, knowledge extraction, and business logic abstraction. In this section we will give an overview of each of these phases.

#### 3.1 Preliminary Study

The first phase of business rules discovery process involves preliminary study of the existing software system. It aims to define the scope and costs of such kind of modernization project. This phase involves the following two steps: gather initial information and define the strategy for knowledge extraction and representation within KDM. During the first step, the present architecture of software system is reviewed, the architectural components are identified, and the high-level dependencies between them are established. 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 artifacts that will be processed, the ways they will be processed, and the time expected for delivery of each representation.

#### 3.2 Knowledge Extraction

This 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) used within the software system. It should be mentioned that depending on concrete project, the knowledge base may include other existing knowledge resources, for example, system log information to provide more clarity on software resources usage.

The KDM representation of the software system is built by discovering its inventory at the first step. This step produces the KDM Inventory model representing system as it is deployed: model elements represent containers, folders, files and their types. Having discovered software inventory, the content of identified software resource definitions and configurations may be extracted and represented within KDM runtime resource models. Typically runtime resources are structured files such as form definition, data definition, report definition, workflow definition files, etc. These artifacts are processed by
and considering MicroKDM [[9]] semantics. The latter allows obtain KDM representation at the sufficiently low level of granularity – statements and expressions of given programming language are represented using different kinds of KDM ActionElements.

Creating the database of software documentation consists of the following steps: the digital documents are parsed using specialized document parsing libraries to retrieve trees representing logical structure of document content (document, chapter, section, subsection, and body), considering a set of rules established regarding the properties of physical content (i.e. blocks) of document; retrieved information is further tokenized, supplemented with corresponding attributes and indexed with full-text index engine to be available for linking with elements of KDM representation in order to facilitate comprehension of the software system artifacts.

A database of classifiers is built by reviewing known lookup tables and files containing classifying data definitions. 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.

3.3 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. To categorize business rules we refer to the BR formalization provided by the GUIDE Business Rules Project [[17]]. The GUIDE classifies BR into the following four categories: business terms, facts, constraints (action assertions), and derivations. After a brief introduction into preparation activities, we will discuss the approach for separation that kind of rules.

BR implemented in the system may cover different system resources. Therefore, we first of all establish dependencies between inter-related elements of representation. The main aim of this step is to build a system control flow graph (CFG) from the KDM representation in order to be able to apply data flow analysis techniques, such as variable reachability and liveness analysis [[6]], and extract the business processing logic. We therefore construct a code-level CFG, and supplement it with higher abstraction level, i.e. runtime resources, dependencies. The CFG, obtained from the code model of example introduced in the previous section is given in the picture below.

Dependencies between runtime resources and source code are established considering runtime resources that produce events and procedures that handle these events. The example of such kind of dependency has been shown in figure 3.
Having established dependences, initial set of term units and fact units (in the GUIDE classification referred as structural assertions) is derived by primarily considering the representation of structural elements of runtime resources. We first refer to UI models, because they contain elements that convey business terms explicitly and therefore may be merely understood by the maintainer. Thus, for each KDMEntity which is an instance of specific type of AbstractUIElement a TermUnit is created and added to ConceptualModel. A reference to that entity is added to the collection of elements representing implementation of TermUnit (property “implementation”). The mUnit elements is augmented with elements that correspond to instances of specific types of AbstractPlatformElement (extracted from the configuration files).

To facilitate further refinement of derived TermUnit elements, we reference them with indices from the data base of software documentation. For each term unit, we construct several types of search queries: the first one limits the search scope to a title of structural elements of indexed documents.
(i.e. chapter, section, subsection); the second one limits search scope to a body of document’s structural element. Search results ranked by relevance are referenced with TermUnit element by creating annotation element for each matching index.

Derivation represents a particular computation of term unit - inference or mathematical calculation. Term units that are defined over complex computations - assignment statements that involve more actions than a simple assignment - are being considered as derivations. We do not, however, separate inference from mathematical calculations, and refer to a derivation as a set of actions taken to obtain the value of given term unit (i.e. a set of control and data flow dependent source code statements used to compute particular variable). Therefore, a derivation is a slice SDF, a set of ActionElements, computed using backward slicing of CFG with criterion CDF=⟨DE, AE⟩, where DE is a set of KDM DataElements representing implementation of particular term units, and AE is a kind of KDM ActionElement representing output of these data elements to user interface, database or other kind of data repository. Considering our example, it would result in a set of control and data flow dependent ActionElements that are represented as colored in grey CFG nodes. From this set, we can further produce the following derivations:

“A discount of Gold member may be applied when the netto price is greater than 2000.00 € and the customer is loyal”
“A discount of Silver member may be applied when the netto price is greater than 2000.00 €”
“A discount of Bronze member may be applied when the netto price is less than 2000.00 €”

Having extracted derivations, the maintainer becomes able to comprehend the logic of computation of particular business term or a set of terms, and validate it with stakeholders.

According to GUIDE, action assertions may be classified in several ways. The first classification that we use in our approach distinguishes action assertions in to the condition, integrity constraint, and authorization. A condition is an assertion that if something is true, another business rule will be applied. We consider this kind of assertion as KDM ActionElement kind of Condition has a direct data flow relationship with an element representing implementation of particular term unit from identified set of term units. This kind of business rule helps the maintainer to understand how many and which conditions evaluate particular business term and evaluate the impact of source code modification to computation of this term. Integrity constraints define assertions that must be always true. They are derived considering properties of term units (i.e. they are defined within data definition files as mandatory fields or unique indices; or within form definitions as required fields). The authority rules represent the configuration and usage of access control lists.

The second classification distinguishes action assertions into the action controlling and action influencing assertions. An action controlling assertion describes what must or must not happen. In the systems this kind of action assertions typically appears as error messages (e.g. VBA function MsgBox kind of critical) after which a control flow terminates. Therefore, we derive it as a slice SAC, a set of ActionElements, computed using backward slicing of CFG with criterion CAC=⟨DE, AE⟩, where DE is a kind of DataElement, defined within a set of elements that implement particular term unit, representing error message text or variable used to produce a text are defined within a set of term units, and AE is an ActionElement representing raise of error message after which an exit from loop or procedure follows. Action influencing assertion describes what should or should not happen. In the systems they typically appear as warning messages (e.g. VBA function MsgBox kind of warning or question) after which a control flow may continue. We derive it as a slice SAU, a set of ActionElements, computed using backward slicing of CFG with criterion CAU=⟨DE, AE⟩, where DE is a DataElement, defined within a set of elements that implement particular term unit, representing error message text or variable used to produce a text, and AE is an ActionElement representing raise of questioning or warning message. Having derived this kind of action assertions, the maintainer is able to quickly find the required error, warning or questioning messages and examine the trace of control flow that influence raise of the message.

4 CONCLUSIONS AND FURTHER RESEARCH

In this paper we have presented the process-centered model-driven approach for business rules discovery from existing software systems. We have shown which of reverse engineering activities must be involved to discover representation of the knowledge about the software system. We also have discussed how the source code analysis techniques can be applied for the representation to abstract the business logic implemented in the system. We have observed that employing standard representation of discovered knowledge about the software system facilitate reverse engineering activities by enabling independence from the implementation platform. 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 must be considered. In order to be able to validate discovered rules with stakeholders, transformations to business specific representations such as SBVR, Decision Tables and Trees, must be considered as well.

5 REFERENCES


