**Michał ŚMIAŁEK, Wiktor NOWAKOWSKI, Norbert JARZĘBOWSKI, Albert AMBROZIEWICZ**

Warsaw University of Technology

# Methodology for Application Logic Recovery from Existing Systems

***Abstract.*** *Newly emerging software design paradigms necessitate changes in legacy enterprise applications. For many such legacy systems their transition to the new paradigms becomes problematic or even impossible due to obsolescence of technologies they use. Replacement of the old system with the new one, built from scratch, is usually economically unacceptable. Therefore, there is a growing interest in methods for automated migration of legacy systems into a new paradigm. In this paper we propose a methodology for extraction and migration of application logic information from existing systems. The information extracted from a legacy application is stored in the form of precise requirement-level models enabling automated transformation into a new system structure.*

***Streszczenie.*** *Rozwijane obecnie nowe paradygmaty projektowania systemów oprogramowania, wymuszają wprowadzanie zmian w istniejących aplikacjach korporacyjnych. W przypadku wielu takich systemów, ich dostosowanie do nowych paradygmatów jest problematyczne a nawet niemożliwe ze względu na przestarzałe technologie, przy użyciu których zostały zbudowane. Zastąpienie starego systemu nowym, wytworzonym od podstaw, jest zazwyczaj ekonomicznie nieuzasadnione. Wynikiem tego jest rosnące zainteresowanie metodami automatycznej migracji istniejących systemów do nowych paradygmatów. Niniejsza praca przedstawia metodykę odzyskiwania oraz migracji logiki aplikacji z istniejących systemów. Odzyskane informacje są przechowywana w postaci precyzyjnego modelu na poziomie wymagań oprogramowania, który następnie może posłużyć jako źródło automatycznej transformacji do nowej struktury systemu.* ***(Metodyka Odzyskiwania Logiki Aplikacji z Istniejących Systemów)***

**Keywords:** application logic, reverse engineering, model transformation, model-driven software development
**Słowa kluczowe:** logika aplikacji, inżynieria odwrotna, transformacja modeli, wytwarzanie oprogramowania sterowane modelami

## Introduction

Newly emerging software design paradigms (like SOA and cloud computing, for example) necessitate dramatic changes in system design and usage patterns. For many legacy applications it means that their further development and transition to these new paradigms becomes problematic or even impossible due to obsolescence of technologies and architectural patterns on which they are based. The only reasonable way to overcome these problems is to build a new system that would accomplish similar functionality as the old one, yet enabling business and technology agility offered by the new paradigms. The cost of building a new system from scratch is often too high. Therefore, there is a call for methods and tools supporting automated recovery of the knowledge buried inside legacy applications and enabling migration into new system design.

In this paper we propose a methodology for extraction of application logic information from legacy systems that can be further easily migrated into new system design. The understanding of application logic extraction from the system design is fundamental to the effective recovery of business value contained in the legacy system. Application logic carries the information about the user-system dialogue in relation to domain-specific processing and platform-specific user interface appearance. In our solution, such information can be extracted from any legacy system by determining its observable behaviour. It can be then stored in the form of requirement-level models compliant with the RSL language. These models can be then transformed into architectural- and design-level models (eg. in UML or SoaML) or even into code. The proposed method is supported by a tooling framework and is an important supplement to the methods for recovery and migration of architecture from legacy systems, that are being developed within the REMICS project [1].

## Capabilities of RSL for Storing Application Logic Information

Software systems' architecture can be structured conforming to a number of design principles. The popular architectural patterns are multilayered architecture [2], Model-View-Controller [3] and Model-View-Presenter [4]. The common part of these are components responsible for controlling the flows gathered by system's interfaces from "the outside" of the system (as inputs from users or other systems), using them to trigger business processing inside the system and then passing responses to output interfaces. This common part is called application logic or workflow logic [5].

Application logic in a typical layered system is realized in one of the layers and it bridges the gap between the business logic (data handling and processing layers) and the user interface tier. These two latter layers conform to the limitation of calling modules of adjacent layers only and communicate only through the application logic layer. In the MVP pattern the Presenter simply passes flows between the View and the Model. In an MVC-style architecture, most of the application logic processing resides in the Controller part, that handles the inputs captured by the View, makes calls to the Model and then sends signals to the View, so they can be passed on to the users.

The application logic information reflects the observable behavior of any IT system and defines the way in which it is operating internally. The application logic's dynamic aspect is a supplement to the information contained in static architectural models. Application logic analysis gives more in-depth look into the system than observation of the "exteriors" of a system (GUI design and user manual analysis). Also, most of the time, the flow of control contained in the application logic is easier to capture and understand than information contained in the source code.

In our approach the application logic information extracted from legacy systems can be stored in the form of models at the requirements level. For that purpose we adapted the Requirements Specification Language (RSL) which is well suited for storing such information.

The Requirements Specification Language (RSL) is a semi-formal language for specifying software requirements. Figure 1 shows an example of the language notation.

RSL employs use cases for defining the system's behaviour. Each use can be detailed by one or more textual scenarios consisting of sentences in constrained natural language. Every scenario is a numbered sequence of actions that are performed either by an actor or the system and lead to success or failure in reaching the use case goal. Every

such action is expressed by a single sentence in the SVO grammar. Sentences in this grammar are composed of a subject, a verb and an object, optionally followed by a second indirect object. The subject indicates who performs the action (the “user” or the “system”). The objects in a sentence represent notions from the business domain (eg. “book”) or user interface elements (eg. “book details window”). The verb, in turn, is strongly relevant to the direct object – it describes an operation that can be performed in association with that object (eg. “validate book”, “display book details window”). The indirect objects can represent detailed data that is passed while performing actions (eg. “displays book list window with book list”).

In addition to action sentences, also condition and “invoke” sentences can be used, which organizes the flow of control between scenarios. Condition sentences are used to define alternate sequences of actions that are performed according to the condition defined. Conditions relate to the system state or actor’s decision. “Invoke” sentences, in turn, are used to denote that another use case (more precisely: one of its scenarios) can be invoked from within currently performed use case.

Notions and phrases used in scenario sentences are linked to elements of the domain model. These elements can have their definitions specific to the system’s domain.

Such notation, separating descriptions of the system’s behaviour from descriptions of the domain problem, is easily understandable to different audiences, including end-users, thus allowing them to discuss the application logic of the system to be built. On the other hand, to facilitate automatic transformations from requirements to design-level models and code, the RSL syntax is precisely defined through a meta-model in MOF [6]. The full RSL specification, including abstract syntax, concrete syntax and semantics, can be found in [7].

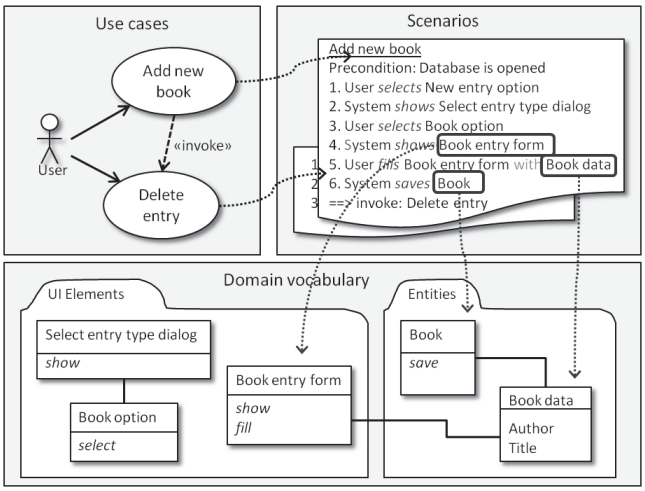


Fig. 1. Summary of the RSL notation

**Process for Application Logic Recovery**

Figure 2 shows an overview of the process for extraction of application logic information from the existing systems. The recovery phase encompasses the idea of semi-automatic reverse engineering while the migration phase is based on model-driven forward engineering techniques. Throughout this process we use the “essential” specifications according to the presented RSL language.

The first step of the process is performed semi-automatically using a GUI-ripping tool (see a discussion on this in [8]). This step involves manual traversal through system’s user interface – a user simply interacts with the legacy system sequentially performing individual functionalities (use cases). An example of such user-system interaction for the JabRef reference manager system (jabref.sourceforge.net) is shown in Figure 3. During this, the GUI-ripping tool records the flows of interaction representing the system’s application logic. This includes the user inputs (buttons clicked, data entered, focus gained, etc.) and respective system responses (windows displayed, messages shown to the user or even textual console behaviour). In order to capture the most extensive application logic knowledge, it is important to traverse through all possible functional paths, including exceptional system’s behaviour resulting, for example, from entering invalid data, operation cancellation etc. The GUI-ripping tool stores all this information in XML-based scripts. In our tool chain we use IBM Rational Functional Tester as the GUI-ripping tool because it supports wide range of UI technologies including those based on textual consoles. However, any tool allowing interaction recording to some form of structured text files may be integrated with our tooling framework.

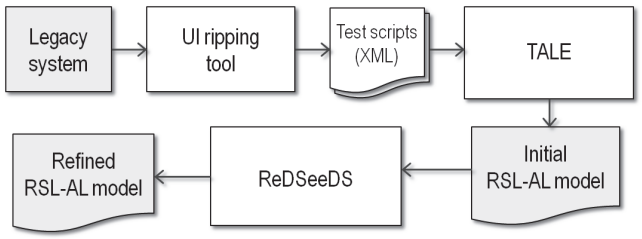


Fig. 2. Overview of the recovery and migration process and tools

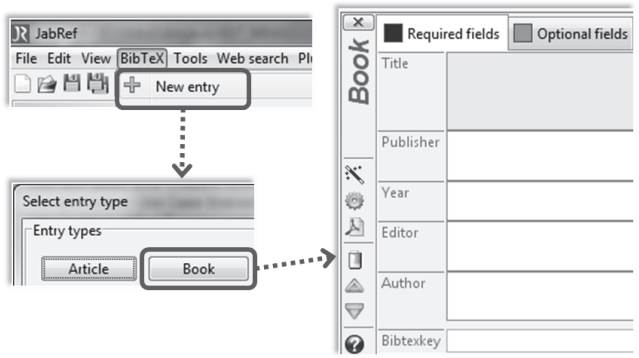


Fig. 3. Elementary example for GUI interactions

The next step is to transform scripts obtained from the GUI-ripping tool into an RSL model. This is done with the TALE tool (Tool for Application Logic Extraction). This novel tool automatically extracts sequences of user-system interactions producing scenarios with SVO sentences. Figure 1 shows an example of extracted scenario representing the interaction illustrated in Figure 3.

Furthermore, the TALE tool also re-creates the domain vocabulary containing domain notions (created mainly based on data passed to and from the user) and UI elements (windows, buttons, input fields, etc.) used in the recovered scenarios. What is important, the tool is able to extract information about the composition of specific notions. For example, when there is a form displayed to enter book data (such as author, title, etc.) a composite notion for "Book data" is created (see Figure 1). Such notion contains descriptions for every field filled on the form, instead of a number of unrelated notions reflecting these fields. This reduces the amount of simple notions created from the GUI recordings and therefore reduces the unnecessary complexity of the recovered model.

The extracted use case scenarios linked to a domain vocabulary form the initial RSL model. This model is human-readable thus giving the possibility of its easy extension and modification. Some modifications are necessary due to not all of the application logic information can be automatically retrieved. This includes sentences that control flow of scenario execution (conditions and ≪invoke≫ sentences) and sentences expressing internal system operations (eg. calls to business logic operations), such as "System verifies data", "System stores information" etc. Also domain vocabulary usually needs renaming some notions. Moreover, changes can be done to cater the migrated system for new or changed functionality or just to optimize some scenario flows.

All these modifications can be made in the ReDSeeDS tool, which offers a comprehensive RSL editor. It allows for writing use case scenarios in SVO grammar and managing of domain specification elements. Switching between TALE and ReDSeeDS is seamless since both tools are integrated within a single framework and they share common data model which is an implementation of the RSL metamodel.

The refined RSL model, containing both the still relevant "legacy" requirements and the "new" ones, is a starting point for the migration phase in which a new system structure is generated. The generation is realised through a model transformation within the ReDSeeDS tool that has a built-in transformation engine for the MOLA language [9]. The structure and notation of the target model depends on the chosen transformation profile as shown in Figure 4.

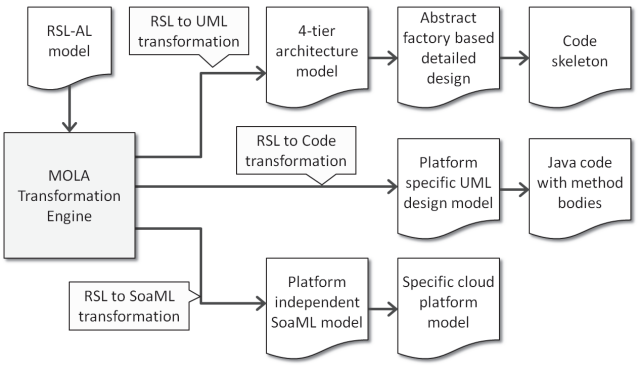


Fig. 4. Transforming RSL-AL model into different target models

Currently "RSL to UML" and "RSL to Code" transformations profiles are ready to use. "RSL to UML" transformations implements the MDA concepts and is able to generate 4-layer solution architecture (Platform Independent Model) and detailed design based on abstract factory in Java (Platform Specific Model) (for details refer to [10]). The "RSL to Code" transformation generates full structure of the system following the MVP architectural pattern, including complete method contents for Presenter and View layers. It also provides a code skeleton for the Model layer.

According to current trends, service-oriented solution is the expected target of the migration process. Our ongoing work currently focuses on the development of transformations from the RSL models into platform independent architectural SoaML models and, possibly, full application logic code for a selected novel web technology.

Because SoaML and UML have the common metamodel, transformations into SoaML and into UML are expected to be similar. The output model of both groups of transformations is UML based logical system design at different levels of abstraction, relevant to the structure of the source requirements specification (use cases, notions and packages). The "RSL to SoaML" transformations is expected to generate target models showing the structure of services their behaviour based on use case scenarios.

**Summary**

Since the presented solution combines some existing approaches, it has been already partially validated. The results presented in [11] prove very good acceptance of the RSL as a specification language. Moreover, evaluation results of the ReDSeeDS approach (see [12]) have shown the feasibility of transformation-supported path from semiformal requirements to code in a model-driven way. A nontrivial part of a software system can be generated by transformations from appropriately defined RSL models.

A comprehensive evaluation of the presented approach in the industrial context is currently ongoing. A larger case study based on a legacy corporate banking system is performed. The main objective of this case study is to modernize the legacy system by migrating it into a cloud in the SaaS model in order to provide uniform access to the system functionality by the customers' external systems.

*This research has been carried out in the REMICS project and partially funded by the EU (contract number IST-257793 under the 7th framework programme), see http://www.remics.eu/.*

REFERENCES

[1] Mohagheghi P., Berre A.J., Sadovykh A., Barbier F., Benguria G.: Reuse and Migration of Legacy Systems to Interoperable Cloud Services- The REMICS project. Proceedings of Mda4ServiceCloud'10 at ECMFA 2010.
[2] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, Chichester, UK, 1996.
[3] Reenskaug T.: Models-views-controllers. Technical note, Xerox PARC, 1979.
[4] Potel M.: MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. Taligent Inc., 1996.
[5] Fowler M.: Patterns of Enterprise Application Architecture Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
[6] Object Management Group: Unified Modeling Language: Superstructure, version 2.2. formal/09-02-02, 2009.
[7] Kaindl H. et al.: Requirements Specification Language Definition. Project Deliverable D2.4.2, ReDSeeDS Project, 2009. www.redseeds.eu.
[8] Memon A., Banerjee I., Nagarajan A.: GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. Proceedings of the 10th Working Conference on Reverse Engineering, pp. 260–269, Nov. 2009.
[9] Kalnins A., Barzdins J., Celms E.: Model Transformation Language MOLA. Lecture Notes in Computer Science, 3599:14–28, 2004.
[10] Bojarski J., Straszak T., Ambroziewicz A., Nowakowski W.: Transition from precisely defined requirements into draft architecture as an MDA realisation. Model Reuse Strategies. Can requriements drive reuse of software models?, pp. 35–42, 2008.
[11] Mukasa K.S. et al.: Requirements Specification Language Validation Report. Project Deliverable D2.5.1, ReDSeeDS Project, 2007. www.redseeds.eu.
[12] Jedlitschka A., Mukasa K.S., Weber S.: Case Creation Verification and Validation. Project Deliverable D6.1, ReDSeeDS Project, 2009. www.redseeds.eu.

***Authors:*** Michał Śmiałek, Wiktor Nowakowski, Norbert Jarzębowski, Albert Ambroziewicz, Institute of Theory of Electrical Engineering, Measurement and Information Systems, Faculty of Electrical Engineering, Warsaw University of Technology, ul. Koszykowa 75, 00-662 Warszawa, Poland, email: *smialek@iem.pw.edu.pl*