Computer Science • 13 (4) 2012 http://dx.doi.org/10.7494/csci.2012.13.4.53

Wiktor Nowakowski
Michał Śmiałek
Albert Ambroziewicz
Norbert Jarzębowski
Tomasz Straszak

RECOVERY AND MIGRATION OF APPLICATION LOGIC FROM LEGACY SYSTEMS

Abstract *Future Internet technologies necessitate dramatic changes in system design, delivery and usage patterns. For many legacy applications it means that their further development and transition to the Internet becomes problematic or even impossible due to the 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 call for methods and tools supporting the automated migration of legacy systems into a new paradigm. This paper proposes a tool supported method for recovery and migration of application logic information from legacy 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 in a model-driven way. Evaluation of the approach is based on a case study legacy system.*

Keywords application logic, reverse engineering, model transformation, model-driven software development

1. Introduction

It is more than obvious that the functioning of most companies and organizations is heavily dependent on software assets which automate or support most of their business processes. Many of such software applications have been in production for years being constantly evolved in order to adapt them to business changes resulting both from changing market needs as well as emerging new technologies providing new business opportunities. Service Oriented Architecture (SOA) and cloud computing are seen as the most dominant software engineering paradigms nowadays. They have dramatically changed the way software systems are designed, delivered and used, what also implies changes in the way business services are provided [1]. For many organizations the transition of their legacy applications to the new architectural patterns becomes problematic. This is mainly due to obsolescence of technologies, platforms and architectures on which legacy systems are based. Software systems introduced many years ago are often characterized by a complex monolithic structure (eg. without clear distinction between user interface, application logic and business model), technologies with non-common gateways, poor interoperability and lack of support, what makes the refactoring to the new structure (eg. component- or service-based architecture) or integration with other enterprise applications virtually impossible. The evolution is also hampered by the loss of knowledge, both technical and business, caused by insufficient documentation of changes introduced over years, changes in personnel, etc.

Most often, the only reasonable solution to the problems mentioned above is to build a new system that would accomplish the functionality of the old one yet enabling business and technology agility possible to achieve with new software paradigms. However, the cost of replacing the old system with a system built from scratch is often too high. Therefore, there is a call for methods and tools for automated recovery of the knowledge buried inside legacy systems facilitating migration to the new architectures or reuse of essential portions of existing systems.

An important initiative to provide standards for understanding and evolving existing software is the OMG's Architecture-Driven Modernization (ADM). It proposes the Knowledge Discovery Metamodel (KDM) [21] for representing the knowledge obtained from existing software in the form of models. KDM provides constructs for representing knowledge about software systems mainly at the level of code. The constructs for representing domain and application logic abstractions are roughly defined.

In this paper we propose an approach for recovery and migration of application logic information from legacy systems. The understanding of application logic extraction from the system design is fundamental to the effective recovery of business value contained in the legacy system. The application logic of an IT system defines sequences of user-system interactions in relation to the domain logic within which the system operates. In our approach, such information can be extracted from any existing system by determining its observable behaviour and stored in the form of requirements-level models conformant to the RSL-AL language. This language is an

extension of the Requirements Specification Language (described in Section 3.1 in this paper) and it serves as an intermediate language between the recovery and migration steps. The migration step uses the ReDSeeDS approach [25, 26] to generate the target system structure. Specifications in RSL-AL can be transformed to component architectures (eg. UML or SoaML), platform specific design (eg. specific cloud platform) and even to implementation (code). The proposed approach is supported by a tooling framework and is an important supplement to the methods for reuse and migration of legacy systems, that are being developed within the REMICS project [17].

The paper is structured as follows. Section 2 discusses the notion of application logic that is to be recovered. Section 3 describes capabilities of the RSL-AL language for capturing application logic related information. Section 4 presents the process and tools for recovery and migration of application logic from legacy applications. Finally, Section 5 concludes by presenting the results of evaluation performed so far and summarising future work.

2. The notion of application logic

Software systems architecture can be structured conforming to a number of design principles established within the IT industry. The popular architectural patterns are multilayered architecture [6, 7], Model-View-Controller [23] and Model-View-Presenter [22]. The common part of these and many other architectural approaches are components responsible for controlling the flows gathered by system 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 (see Figure 1). This common part controlling internal flows in a software system is called application logic or workflow logic [9].

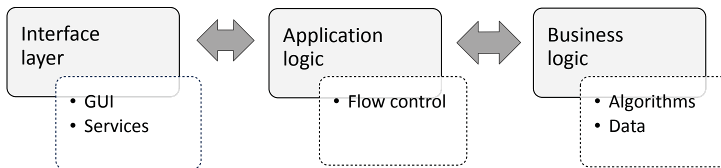

Figure 1. The role of application logic in IT systems.

The notion of application logic varies in the implementation in regard to architectural patterns mentioned above. 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 as depicted in Figure 1. These two latter layers conform to the limitation of calling modules of adjacent layers only (see [7] for discussion on this constraint) 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 carries information about the user-system dialogue in relation to domain-specific processing and platform-specific user interface appearance. Such information reflects the observable behavior of any IT system and defines the way in which it is operating internally. It can be argued that the re-discovery of knowledge residing in the legacy system through the aspect of application logic has advantages over other approaches. The application logic's dynamic aspect is a supplement to the information contained in static architectural models. Application logic analysis gives a more in-depth look into the system than observation the 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.

3. Capturing application logic with RSL-AL

The Requirements Specification Language allows for conceptual modeling based on object-oriented ideas and user-interface specifications in the area of requirements engineering. Software requirements modeled in RSL have their behavioral and structural aspects distinguished: the descriptions of modeled domain elements are separated from the specification of their dynamics. In RSL, links between behavioral and static elements can be assigned roles and responsibilities and can have temporal ordering and variety of conditions specified. This allows for precise flow control in the resulting models.

The RSL notation is human-readable (based on popular notation, understandable to different audiences and using expressions as close as possible to the natural language), but on the other hand is precise enough to allow automated processing (like, for example, MDA-style transformations [16]).

In Sections below we describe principles of RSL and its application to logic extension. For the extended overview of the RSL language please refer to [24] and to [13] for the formal language definition.

3.1. RSL concepts and structure

In RSL, a taxonomy of requirements is formulated. The most general requirement type, called simply "Requirement", is an expression of some feature defined for a software system. The Requirement's subtypes include functional and constraint requirements, as well as use cases specialized from the UML use cases [20].

Relationships from a fixed set defined in RSL may be used to interrelate requirements. This set includes several types for Requirement Relationships as well as use-case-specific relationship "Invoke" used to denote that another Use case (more precisely: one of its scenarios) can be invoked from within the currently performed

Use case. The advantage of this relationship over UML «include» and «extend» mechanisms lies in its precision and unambiguity: the invoked scenario steps are executed in the exact position in the interaction sequence as defined by the relationship usage (see [4] for a discussion on vague semantics of the include/extend relationships).

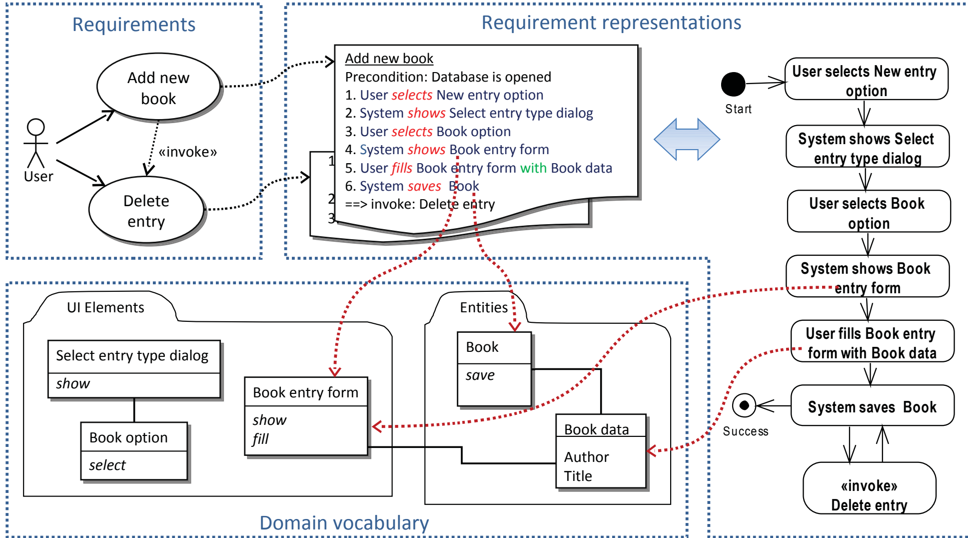

Figure 2. Summary of the RSL notation.

RSL differentiates between requirement entities ("requirements as such") and their representations (their content). Requirements as such are names with identifiers, attributes and relations to other requirements. Content representations carry the information contained within the requirements (e.g. for Use cases, an interaction information in the form of scenarios). This is illustrated in Figure 2 which can be used as an example for the following description of RSL.

Requirements can have multiple representations that carry equivalent information. Given that information included in different representations is the same, the representations can serve different purposes as they present the same requirements content in a disparate manner. While some of the representations are oriented towards human RSL users, the others are intended for automated machine processing. Some of the human-readable representations are easily read and understood by a business audience (for example software users or project sponsors' representatives), the others are more appealing to technical people (e.g. programmers, designers, or architects).

The current RSL version defines two main types of requirements representations: constrained language representations and schematic (or: diagrammatic) representations. Constrained language representations express requirement information as textual scenarios. The diagrammatic representations are based on UML activity diagrams and sequence (interaction) diagrams. All types of representations are defined

in a formal way through a grammar. The grammar for diagrammatic representations is expressed as a MOF [19] metamodel.

The representations defined in RSL contain user-system interaction information and flow of events for a given requirement. This interaction is described by a set of scenarios (leading to one goal or showing a number of ways that fail in reaching that goal). Each scenario has a set of ordered sentences describing signals exchanged between actors and system components participating in the scenario. Special types of sentences, called control sentences, are used to express conditions in the flow of interaction. For more information on requirement representations refer to [27].

In RSL, the container for requirements as well as their representations is called the Requirements Specification. It consists of a hierarchy of requirement packages and a specialized package named the Domain Specification. This last package contains elements pertaining to the vocabulary of the problem domain separated from (or rather: pointed at by) a description of requirement dynamics carried by the requirements representations. The domain specific vocabulary in RSL is centered around nouns. In RSL, a noun ("user", "ticket") is called a notion. An RSL noun may consist of multiple words (like "data form" or "user account"). Each such noun-based notion is a basic vocabulary element and it has a textual definition.

Notion can be part of many different phrases. Each of such phrases contains the notion with a supplement of other parts of speech (e.g. verbs or adjectives). For example "user account" may have phrases "delete user account" (with definition: "to delete from the system an account related to a user") and "expired user account" ("an account that was not renewed in a given period"). Definitions used to describe notions and phrases may contain other notions or phrases. Such use of domain element names in the definitions is a basis for creating associations among domain elements and resulting in a domain vocabulary having the characteristics of a static class-like model.

As it was pointed above, the domain vocabulary elements are used within the descriptions of behavioral aspects of the requirements and are referred to by requirement representations. Such references are called hyperlinks (this is also true for the situation when a textual definition of some domain element uses the name of other notion or other notion's phrase). Hyperlinks are the building blocks of textual representations and the domain element definitions. For the constrained representations, the sequence of hyperlinks used in a given expression conforms to a specified grammar.

RSL uses the SVO(O) grammar [10] [11]. The basic sentence structure for this grammar is Subject – Verb – Object – Indirect Object. In RSL, a sentence in the SVO(O) grammar contains one hyperlink to its subject (a noun phrase, in most cases just a notion) and a hyperlink to a verb phrase. This verb phrase may be a simple one (containing only a single object) or complex (with two objects and an optional preposition). Such sentences are used to precisely describe the dynamic aspect of a requirement: a sentence conforming to the SVO grammar defines an active interaction entity (the sentence subject) performing an action (the verb in the verb phrase) with the use of passive elements (the objects).

3.2. RSL extension for application logic

The core of RSL has been extended with capabilities for creating solution-independent application logic descriptions. The RSL-AL extension is based on RSL concepts like separation of elements' description and their dynamics specification, precise domain vocabulary and rigorous interaction definition, and at the same time it allows for the efficient management of application logic building blocks and application logic patterns.

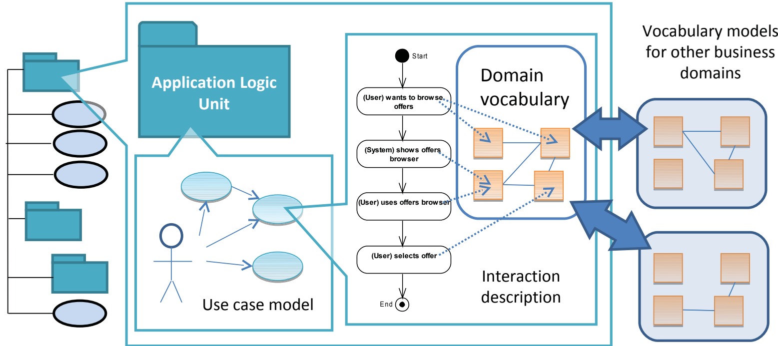

Figure 3. The levels of application logic management.

While RSL is a language with capabilities for application logic specifications, the new elements in the extension are introduced to allow for the efficient management of application logic related information. This management can be done on different levels: on the functionality overview level and on the level of detailed interaction information (see Figure 3).

The upper level of abstracting the application logic information pertains to elements grouped in units collecting areas of functionality. These groupings, called Application Logic Units, are containers for elements of functionality (use cases) and can be interrelated by the use of precisely defined relationships. The granularity of application logic management on this level may be compared to package-level concepts of UML.

Relationships existing at the lower detail level (connecting units of functionality) are already covered extensively in RSL (see the «invoke» relationship above), but RSL-AL introduces the idea of linkages between snippets of functionality. Such partial functionalities do not describe interactions which allow reaching some kind of a goal or a significant value (as opposed to user-system dialogue found in use cases), but are important and/or repeatable elements of control flow descriptions within the software system. They can contain just the partial flow of interaction leading to an intermediate goal and are intended as basic functionality blocks. The RSL extension for application logic allows operating on such partial functionalities: they can be defined and then interrelated (put together, chained) to create more complete functional units. These

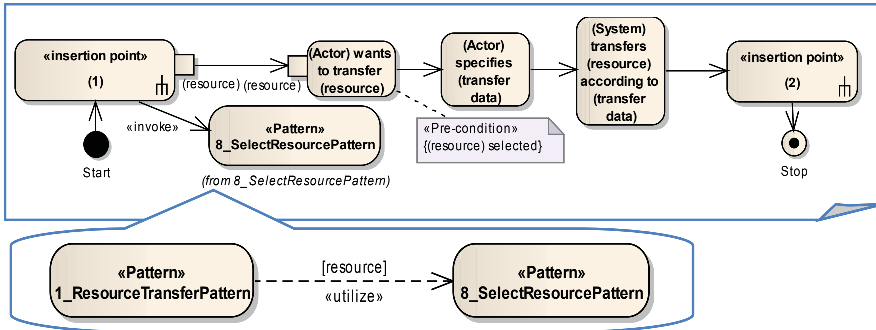

Figure 4. Insertion of application logic "snippet" into flow of interaction describing other functionality.

relationships are represented at two levels of abstraction: at the level of a functional unit (use case level, sub-package level) and at the level of a requirement representation (as a textual scenario or an activity diagram).

In the RSL-AL approach, full utilization of the RSL concept of separating the vocabulary and behavior enables control of consistency of the vocabulary used in the chained snippets of functionality. Each of such elementary application logic building blocks has precisely defined notions it refers to and these notions are used as this functionality parameters. When one partial functionality is inserted into other one, all the uses of parametrized notions in the inserted one are substituted with corresponding notions of the target unit of functionality. This ensures that the resulting interaction description uses consistent wording.

Figure 4 presents an example of inserting an application logic snippet (regarding a resource selection) into the interaction flow for transferring a resource. In the main interaction, a marker (insertion point) indicates the position at which other flows can be inserted (by the «invoke» relationship). UML pin-like notation indicates the vocabulary parameter of the interaction description. The higher-level view of this information (the diagram in the lower part of Figure 4) shows just two interactions and a relationship between them along with the vocabulary parameter used).

The above notion substitution mechanism is also used at other levels of application logic elements management and enables the use of patterns in the domain of application logic modeling. Elements of functionality (packages, units of functionality or atomic building blocks) can be abstracted from the business domain they describe (note interactions in Figure 4 that deal with very general notions like resource). Then, reusing them in different contexts (with different business vocabulary) needs minimal effort. Such a reuse process requires only mapping of abstracted domain vocabulary elements to a target (concrete) domain model. The descriptions of interaction flow are not changed in this process (see right-hand side of Figure 3). For a more detailed explanation of this mechanism refer to [3].

4. Process and tools for application logic recovery and migration

Figure 5 shows an overview of the process and tools allowing for recovery and migration 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-AL language. We first analyse the legacy system's UI by using a GUI-ripping tool. Based on this semi-automatic analysis we generate the initial RSL-AL model which can then be modified by hand to refine it or to cater for new or changed functionality. By the fact, that the model is based on a metamodel, we can use a model transformation engine to generate the target system structure models (both platform independent and platform specific) and code. Subsequent steps of the process are described in detail in the sections below.

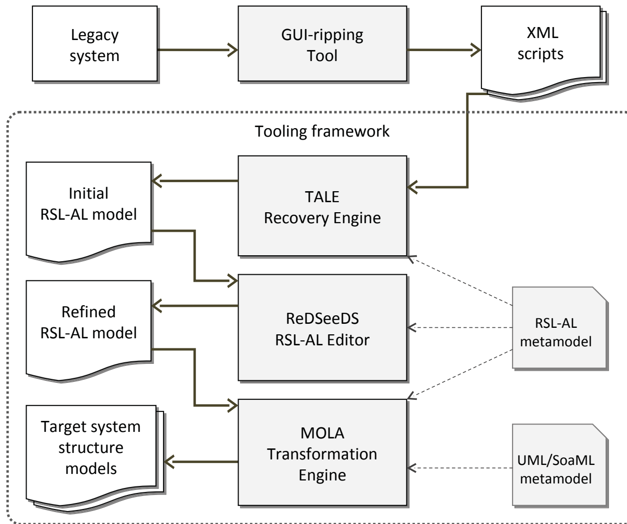

Figure 5. Overview of the recovery and migration process and tools.

4.1. Recovery

The first step of the recovery process is performed using a GUI-ripping tool (see a discussion on this in [15]). This step is performed semi-automatically. It involves manual traversal through a system's user interface during which the system observable behaviour is systematically scanned. A user (preferably a person who normally works with the legacy system and is aware of its behaviour), simply interacts with the legacy system sequentially performing individual functionalities (use cases). An

example of such interaction for searching a client (in Polish: Wyszukiwanie klienta) is illustrated in Figure 6a. During this, the GUI-ripping tool records the flows of interaction representing the system's application logic. This includes user inputs (buttons clicked, data entered, widget focus gained, etc.) and the 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 a 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.

The next step of the recovery process is to transform scripts obtained from the GUI-ripping tool into an RSL-AL model. This is done with the TALE tool (Tool for Application Logic Extraction) developed as part of this work. This novel tool automatically extracts sequences of user-system interactions producing scenarios with SVO sentences. Figure 6b shows an automatically extracted scenario representing the interaction illustrated in Figure 6a. All the extracted scenarios are attached to use cases as their representations and are grouped within the "Functional Requirements" package being a part of the recovered model (see the project tree in Figure 6b).

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 personal data (such as first name, last name, PESEL numer, etc. – see the "Osoba fizyczna" tab in Figure 6a), a composite notion for "Osoba fizyczna data" is created. Such a 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, therefore reduces the unnecessary complexity of the recovered model. All these elements are stored in the "Domain Specification" package.

The extracted use case scenarios linked to a domain vocabulary form the initial RSL-AL model. Thanks to the characteristics of the RSL-AL language, this model is easily understandable to people (even those barely knowledgeable of the original system) thus giving the possibility of its easy extension and modification. First of all, some modifications are needed because of the fact that not all of the application logic information can be automatically retrieved from the recording scripts. 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", "System deletes item from item list" etc. Also domain vocabulary usually needs manual refactoring – mostly renaming some notions. Moreover, changes can be done to cater to the migra-

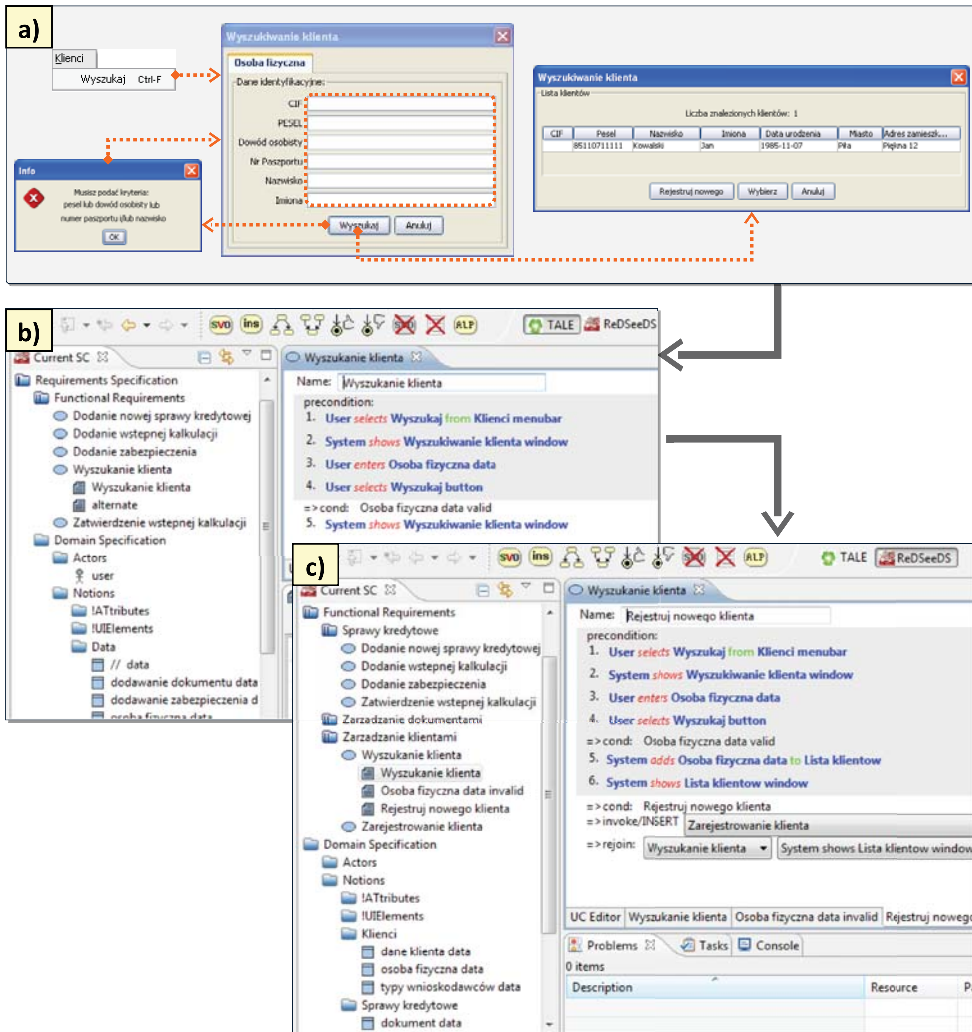

Figure 6. An example of GUI interaction (a), the automatically recovered RSL-AL model (b) and the manually refined final model (c).

ted system for new or changed functionality or just to optimize some scenario flows, eg. by applying standard application logic patterns [3].

All these modifications can be made in the ReDSeeDS tool, which offers a comprehensive RSL-AL editor. It allows for writing use case scenarios in accordance with the rules of the language grammar. Managing of domain specification elements from the level of the use case editor or using tree-like structures is possible as well. Also a basic application logic pattern library is supplied with the tool. Switching betwe-

en TALE and ReDSeeDS is seamless since both tools are integrated within a single framework and they share a common data model which is an implementation of the RSL-AL metamodel. Figure 6c shows the recovered model after refinements.

4.2. Migration

The refined RSL-AL 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 [14]. In order to do this, we need to reorganize the requirements model according to the needs of the transformation rules that are to be applied (as shown in Figure 6c) and choose one of predefined transformation profiles. The structure and notation of the target model depends on the chosen transformation profile as shown in Figure 7.

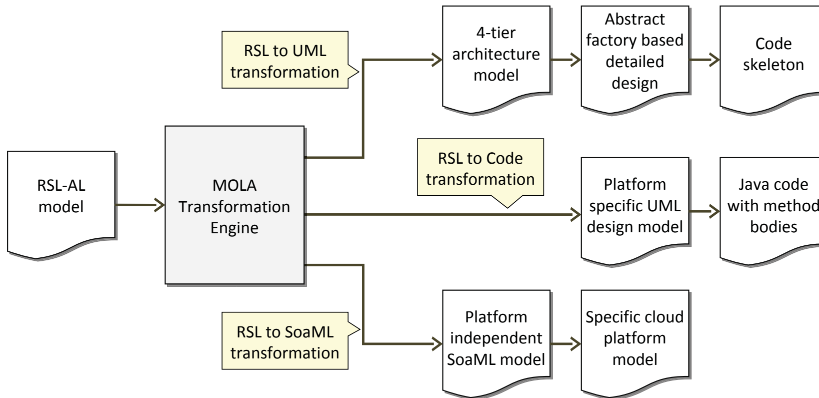

Figure 7. Transforming RSL-AL model into different target models.

Currently "RSL to UML" and "RSL to Code" transformations profiles are ready to use. "RSL to UML" transformation chain implements the MDA concepts with the requirements specification in RSL as the CIM (Computation Independent Model), multi-tier architecture model as the PIM (Platform Independent Model) and detailed design model based on abstract factory design pattern as the PSM (Platform Specific Model) [5]. The "RSL to Code" transformation is able to generate classes forming a full structure of the system following the MVP architectural pattern, including the complete code for the method bodies in the application logic (Presenter) and presentation (View) layers. It also provides a code skeleton with method stubs for the domain logic layer (Model).

According to current trends in internet technologies, an expected target of the migration process is a cloud-enabled system. Thus, the new system structure should

follow service-oriented architecture principles and should be expressed in SoaML [8]. This requires a specific transformation profile, which is under construction now.

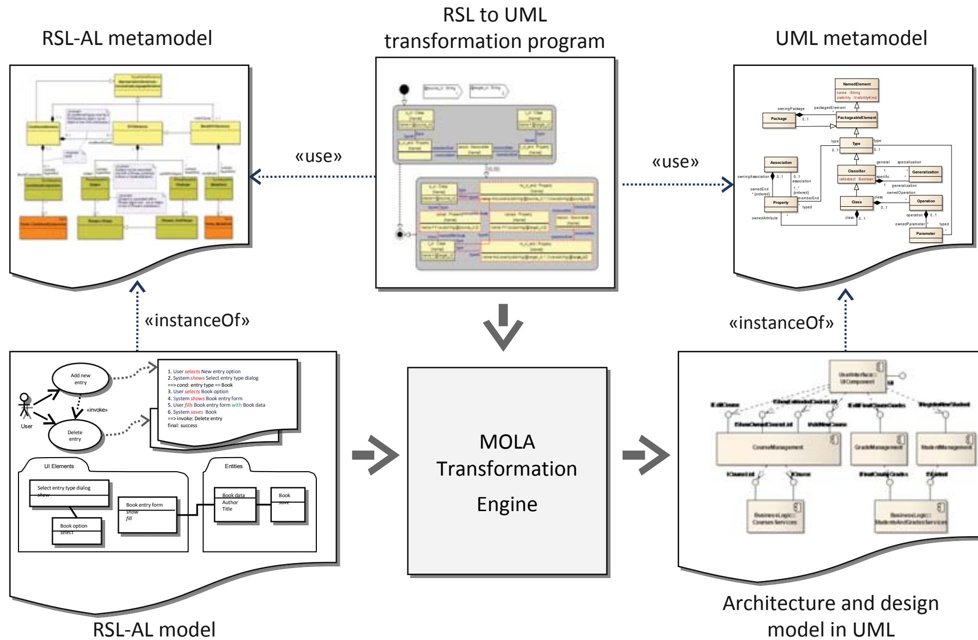

Figure 8. Transformation details.

The migration process, implemented within the ReDSeeDS tool, uses the MOLA engine. An example transformation scheme is shown in Figure 8. The MOLA transforamtion engine uses transformation programs written in MOLA transformation language, which offers very readable graphical notation. Any transformation expressed in MOLA consists of meta-models for the source and target models, along with one or more transition procedures. Source and target meta-models are defined in the MOLA meta-modelling language, which is close in specification to that of EMOF (Essential MOF — see [19]). MOLA procedures form the executable part of the MOLA transformation. Traceability associations, which link elements of the source meta-model and corresponding elements of the target meta-model, facilitate building of natural transformation procedures and document the performed transformations. This is illustrated in Figure 9. There is a trace from the "Wyszukanie klienta" use case leading to the "IWyszukanieKlienta" interface in the application logic tier in the solution architecture and its realisation within the "ZarzadzanieKlientami" component in the detailed design. This structure is the result of the "RSL to UML" transformation, where the target model is an instance of the UML meta-model.

By the fact that SoaML (see [2]) and UML have a common meta-model, transformations to SoaML are expected to be similar to the transformations which generate

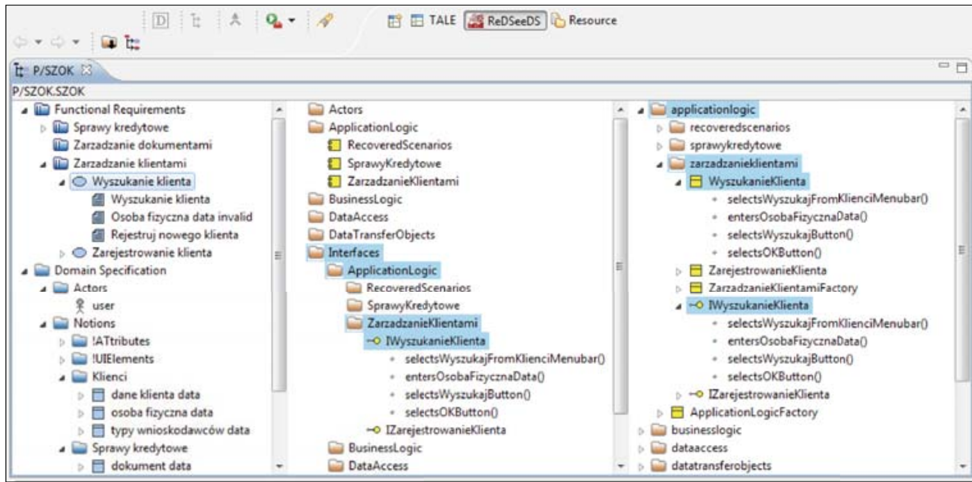

Figure 9. Target system structure generated with the RSL to UML transformation.

UML. The output model of both groups of transformations is an 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 are expected to generate the structured model of services constructed with stereotyped packages, components, interfaces, classes. The target models is also expected to contain sequence diagrams describing the services behaviour based on the use case scenarios. All messages exchanged via services in sequence diagrams will have adequate operations in the corresponding interfaces thus keeping the target model coherent. An example of a sequence diagram generated with "RSL to UML" transformation is shown in Figure 10.

5. Evaluation and future work

Since the presented solution combines some existing approaches, it has been already partially validated. The results presented in [18] prove very good acceptance of the RSL as a specification language. Also usability of RSL-AL constructs when writing application logic specifications have been validated in a controlled experiment (see [3]). 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 (including the recovery phase) in the industrial context is currently ongoing. A larger case study based on a legacy corporate banking system delivered by Infovide-Matrix S.A. (one of the major Polish software providers) is performed. In fact, the examples in this paper are taken

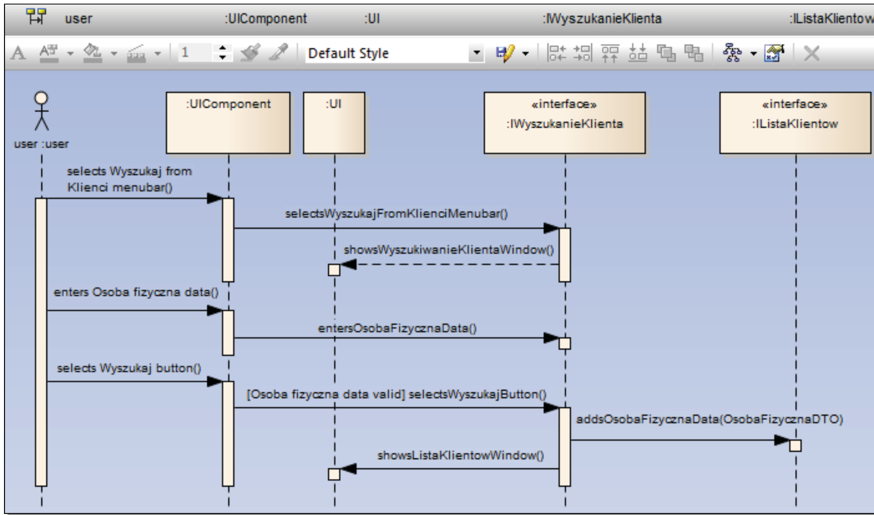

Figure 10. Dynamic architectural model generated with the RSL to UML transformation.

from this case study (a bank credit management system). 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.

Early experiments on the case study system show promising levels of application logic that can be recovered. For a certain part of the system's functionality a set of test scripts were recorded in the GUI-ripping tool. With this input, the TALE tool was able to produce sensible RSL-AL model. It needed some manual modification of the interaction sequence in most of the scenarios (ca. 80 percent) as well as some modification of the domain vocabulary (ca. 50 percent of notions needed to be renamed). Although the recovery phase is not fully automatic, experience shows that it is still much faster than it would be if one had to manually write scenarios from scratch. Moreover, the recovery can be associated with the normal operation of the recovered system. The operators can work normally with the legacy system, at the same time recording its functionality.

The ongoing work on the solution currently focuses on the development of transformations from the RSL-AL models into platform independent architectural SoaML models and, possibly, a full application logic code for a selected novel web technology. To make our approach more holistic, future work will include automatic generation of test cases from RSL-AL models. This will assure that the migrated system will comply with the observable behaviour of the legacy one. Furthermore, it will be possible to perform acceptance testing of the functionality updated during the recovery and migration process.

Acknowledgements

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] *Economic and social impact of software & software-based services, d2 – the european software industry.* Pierre Audoin Consultants Report, July 30 2009.

[2] *Service oriented architecture Modeling Language (SoaML) Specification, version 1.0, formal/2012-03-01*, 2012.

[3] Ambroziewicz A., Śmiałek M.: Application Logic Patterns — reusable elements of user-system interaction. In *Proc. of MODELS'10, Oslo, Norway, LNCS 6394*, pp. 241–255. Springer, 2010.

[4] Astudillo H., Génova G., Śmiałek M., et al..: Use cases in model-driven software engineering. *LNCS*, 3844:262–271, 2006. MODELS'06.

[5] Bojarski J., Straszak T., Ambroziewicz A., Nowakowski W.: Transition from precisely defined requirements into draft architecture as an mda realisation. In M. Smiałek, K. Mukasa, M. Nick, J. Falb, ed., *Model Reuse Strategies, Can requriements drive reuse of software models?*, pp. 35–42. Fraunhofer Verlag, 2008.

[6] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *Pattern-Oriented Software Architecture, vol. 1: A System of Patterns.* Wiley, Chichester, UK, 1996.

[7] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., Stafford J.: *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2003.

[8] Elvesaeter B., Berre A.-J., Sadovykh A.: Specifying services using the service oriented architecture modeling language (soaml) — a baseline for specification of cloud-based services. In F. Leymann, I. Ivanov, M. van Sinderen, B. Shishkov, eds., *Proc. of CLOSER 2011, Aachen, Germany*, pp. 276–285. SciTePress, 2011.

[9] Fowler M.: *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[10] Graham I. M.: Task scripts, use cases and scenarios in object-oriented analysis. *Object-Oriented Systems*, 3(3):123–142, 1996.

[11] Graham I. M.: *Object-Oriented Methods Principles & Practice.* Pearson Education, 2001.

[12] Jedlitschka A., Mukasa K. S., Weber S.: Case creation verification and validation. Project Deliverable D6.1, ReDSeeDS Project, 2009. `www.redseeds.eu`.

[13] Kaindl H., Śmiałek M., Wagner P., Svetinovic D., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T., Schwarz H., Bildhauer D., Brogan J. P., Mukasa K. S., Wolter K., Krebs T.: Requirements specification language definition. Project Deliverable D2.4.2, ReDSeeDS Project, 2009. `www.redseeds.eu`.

[14] Kalnins A., Barzdins J., Celms E.: Model transformation language MOLA. *Lecture Notes in Computer Science*, 3599:14–28, 2004. Proc. of MDAFA'04, Linköping, Sweden.

[15] Memon A. M., Banerjee I., Nagarajan A.: GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proc. of the 10th Working Conference on Reverse Engineering, Victoria, Canada*, pp. 260–269, November 2003.

[16] Miller J., Mukerji J., editors. *MDA Guide Version 1.0.1, omg/03-06-01.* Object Management Group, 2003.

[17] Mohagheghi P., Barbier F., Berre A., Morin B., Sadovykh A., Saether T., Henry A., Abhervé A., Ritter T., Hein C., Śmiałek M.: *European Research Activities in Cloud Computing*, chapter Migrating Legacy Applications to the Service Cloud Paradigm: The REMICS Project. Cambridge Scholars Publishing, 2012.

[18] Mukasa K. S., et al.: Requirements specification language validation report. Project Deliverable D2.5.1, ReDSeeDS Project, 2007.

[19] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.

[20] Object Management Group. *Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02*, 2009.

[21] Object Management Group. *Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), version 1.3, formal/2011-08-04*, 2011.

[22] Potel M.: *MVP: Model-View-Presenter the Taligent programming model for C++ and Java.* Technical Report, Taligent Inc., 1996.

[23] Reenskaug T.: *Models-views-controllers.* Technical Note, Xerox PARC, 1979.

[24] Śmiałek M., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T.: Introducing a unified requirements specification language. In *Proc. CEE-SET'2007, Software Engineering in Progress, Poznań, Poland*, pp. 172–183. Nakom, 2007.

[25] Śmiałek M., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T.: *Od modelu do wdrożenia — kierunki badań i zastosowań inżynierii oprogramowania*, chapter Narzędzie i metodyka dla systematycznego wytwarzania oprogramowania, pp. 23–34. Wydawnictwa Komunikacji i Łączności, 2009.

[26] Śmiałek M., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T., Wolter K., Hotz L., Mukasa K. S., Jedlitschka A., Bildhauer D., Falkowski K., Haas J., Horn T., Riediger V., Schwarz H., Kalnins A., Kalnina E., Sostaks A., Celms E., Rein M., Drejewicz S., Knab S., Falb J., Çetin S., Tüfekçi O., Çokkeçeci I.: Case-driven software development comprehensive approach to produce and reuse model-based software cases. Project Deliverable D8.2.2, ReDSeeDS Project, 2009. `www.redseeds.eu`.

[27] Śmiałek M., Bojarski J., Nowakowski W., Ambroziewicz A., Straszak T.: Complementary use case scenario representations based on domain vocabularies. *Lecture Notes in Computer Science*, 4735:544–558, 2007. Proc. of MODELS'07, Nashville, TN, USA.

Affiliations

Wiktor Nowakowski
Warsaw University of Technology, Warsaw, Poland, nowakoww@iem.pw.edu.pl

Michał Śmiałek
Warsaw University of Technology, Warsaw, Poland, smialek@iem.pw.edu.pl

Albert Ambroziewicz
Warsaw University of Technology, Warsaw, Poland, ambrozia@iem.pw.edu.pl

Norbert Jarzębowski
Warsaw University of Technology, Warsaw, Poland, jarzebon@iem.pw.edu.pl

Tomasz Straszak
Warsaw University of Technology, Warsaw, Poland, straszat@iem.pw.edu.pl

Received: 9.03.2012
Revised: 26.06.2012
Accepted: 3.09.2012