Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1349–1356

ISBN 978-83-60810-51-4

Using structured grammar domain models to capture software system essence

Michał Śmiałek, Albert Ambroziewicz, Wiktor Nowakowski, Tomasz Straszak and Jacek Bojarski
Warsaw University of Technology
Warsaw, Poland
Email: smialek@iem.pw.edu.pl

***Abstract*—Creation of a precise domain vocabulary is crucial for capturing the essence of any software system, either when recovering knowledge from a legacy system or when formulating requirements for a new one. Software specifications usually maintain noun notions and include them in central vocabularies. Verb or adjective phrases are easily forgotten and their definitions buried inside imprecise paragraphs of text. This paper proposes a model-based language for comprehensive treatment of domain knowledge, expressed through constrained natural language phrases that are grouped by nouns and include verbs, adjectives and prepositions. In this language, vocabularies can be formulated to describe behavioural, declarative and conditional characteristics of a given problem domain. What is important, these characteristics can be used (linked) from within other specifications similarly to a wiki. In particular, the application logic can be formulated through sequences of imperative Subject-Predicate sentences containing only links to the phrases in the vocabulary. The paper presents initial tooling framework to capture application logic specifications and make them available for further automated transformations.**

I. Introduction and related work

AS POINTED out by Brooks back in the eighties [1], software systems possess essential (inherent) and accidental (technological) complexity. The essential complexity cannot be removed without reducing the problem at hand. In order to understand any software system we thus need to "extract" this essential complexity and make it clearly visible. This is especially important when modernising the existing systems. We normally would like to remove all the code, related to the old technology and retain just the problem-related essence. Then, we could transfer this essence (after possible improvement and extension) into a new technology.

An important attempt to enable capturing essential knowledge about software systems is the Knowledge Discovery Metamodel (KDM), as explained by Pérez-Castillo et al. [2]. Unfortunately, KDM operates mainly at quite low levels of abstraction, concentrating e.g. on defining a metamodel for abstract syntax trees capturing the code structure of the system. It also contains structures to represent conceptual-level artifacts but this part of the standard is very roughly defined. Moreover, it can be argued that capturing the detailed internal structure does not reduce the accidental complexity associated with the "twisted" internals of a legacy system. We need means to capture the essence of the system's logic and not e.g. the detailed code breakdown structure as implemented in the legacy system.

By contrast, a very comprehensive approach to capturing essential knowledge (Domain Driven Development - DDD) was proposed by Evans [3]. He postulates organising software development around rigorously defined domain models. These models capture the domain logic of the system at a high level of abstraction. At the same time, the domain logic is the foundational basis to specify the application logic describing the observable interaction of the users with the system (called "workflow logic" by Fowler [4]). This approach was even strengthened in rigour by Bjôrner [5] who advocates mathematical precision in domain engineering. He identifies serious flaws in system specification whenever domain specifications are treated without enough care.

Domain engineering is thus argued as an important element in capturing the essential complexity. Unfortunately, it is normally treated as a second-class citizen in specifying systems. It is equated with a more-or-less complete list of noun-related domain elements with their definitions, placed somewhere close to the end of the overall specification (be it requirements, design or business description) and soon forgotten. Worse still, in many cases the vocabulary is in fact buried in text throughout the whole specification. All the definitions of domain notions are scattered everywhere leading in many places to contradictions (e.g. different definitions of the same term). This all calls for a tooling framework where the various domain notions could be used consistently through referring to a central vocabulary, as postulated by Śmiałek et al. [6].

The tooling for DDD has been developed in the context of the Romulus project (see work by Iglesias et al. [7]). However, the domain models in Romulus are at the level of design models rather than pure domain descriptions. A domain-driven approach was also taken by the creators of the Requirements Specification Language (RSL [8]) within the ReDSeeDS project (www.redseeds.eu). The domain models in this language rely heavily on verbs used within requirements specifications. This is also similar to knowledge engineering approaches like the one described by Chan [9] and also pure ontology languages like RDF [10].

In effect, we result with a constrained language with embedded semantics, capable of representing domains along the proposition by Evermann and Wand [11]. Moreover, the language introduces a very strict relation between the domain logic (expressed through verbs associated with nouns) and the application logic.

In the current work we use RSL to enable capturing the essential complexity at the level of application logic of either existing or new systems. This kind of "essential complexity" is meant as sequences of user-system and system-system/user interactions defining the observable system behavior. We propose to capture it through constrained-natural-language sentences that refer (hyperlink) directly to a domain model based on nouns, verbs and other parts of speech. Similar usage of hyperlinks was proposed by Kaindl [12], but such a comprehensive treatment with an extensive tooling environment is not found in the literature according to our best knowledge. What is more, we propose a method for capturing and migrating the essence from legacy systems. It is unique in generating application logic scripts from UI/GUI-ripping results. The users record their activity in the legacy system and this is transferred to the application logic (essential) specification. Due to precision of such specifications, this can be brought to the level of code in an MDA-style transformation process [13].

II. Phrases: basic building blocks for specifications

In order to describe a domain, people normally use certain natural language phrases. Any entity in a given domain is expressed through a phrase containing prominently a noun. In sentences, nouns are normally used in the role of subjects or objects. Noun phrases are obviously not satisfactory to express the domain logic – its dynamics. We need verbs that can be composed of many words (e.g. phrasal verbs or aggregates of the Dixon's primary and secondary type verbs [14]). In a sentence, a verb occurs as a part of its predicate. It is strongly relevant to the noun: it describes behaviours, functions and events of the entity represented by that noun. These are important elements of domain descriptions as defined by Bjôrner [15]. One noun can have any number of behaviours, functions or events associated ("read book", "write book", "buy book"). Sometimes there is a need to enrich nouns with modifiers or determiners ("single book", "old book").

To capture the application logic we will thus define a language capable of expressing noun-based phrases. This is illustrated in Figure 1. The base "phrase" contains a noun and optionally a determiner and a modifier (most often – an adjective or an adverb). "Verb phrases", in addition, contain verbs ("simple verb phrase") or verbs and other nouns ("complex verb phrase"). Verb phrases can be enriched with prepositions. All the parts of a phrase constitute atomic terms.

The above can be seen as a constrained language and we can define a grammar for it. We want the language to be

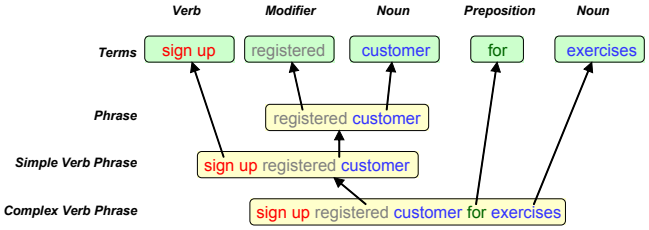

Figure 1. Phrase structure example

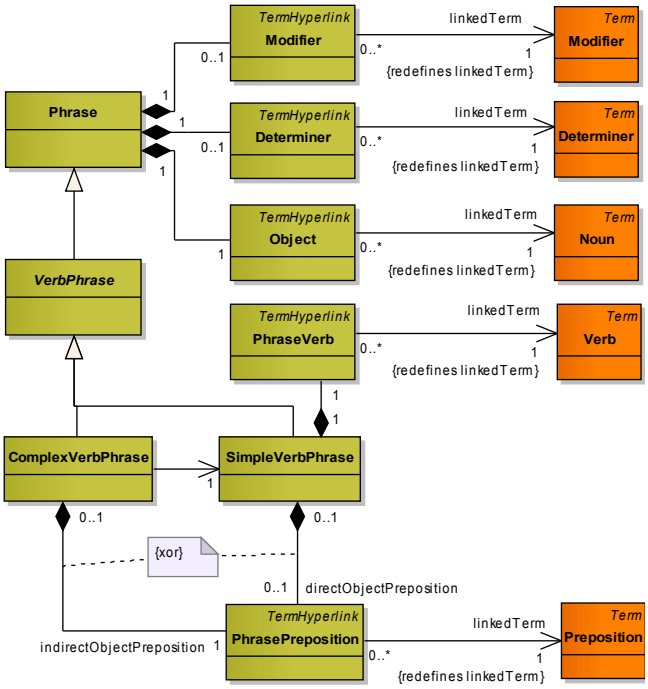

Figure 2. Phrase metamodel

used for automatic transformations and thus we will use a metamodel to define it (work by Kleppe [16] can be used as a good introduction on this). This is shown in Figure 2. A Phrase always contains an Object that points to a specific Noun. Such Phrases are satisfactory for representing entity names. More complex Phrases can optionally contain a Modifier and a Determiner.

A VerbPhrase describes some behaviour that can be performed in association with a Noun. In the metamodel, VerbPhrase is an abstract subtype of Phrase and it exists in two concrete variants: SimpleVerbPhrase and ComplexVerbPhrase. The SimpleVerbPhrase is the basic structure for expressing the Noun's behaviour. It can have all the elements of a Phrase, but it also includes a PhraseVerb pointing to a Verb. A ComplexVerbPhrase describes behavioural relation between two Nouns. It can be noted that it contains an Object (the indirect object) but also points to a SimpleVerbPhrase that contains another Object (the direct object).

It is worth noting that the phrases in fact constitute

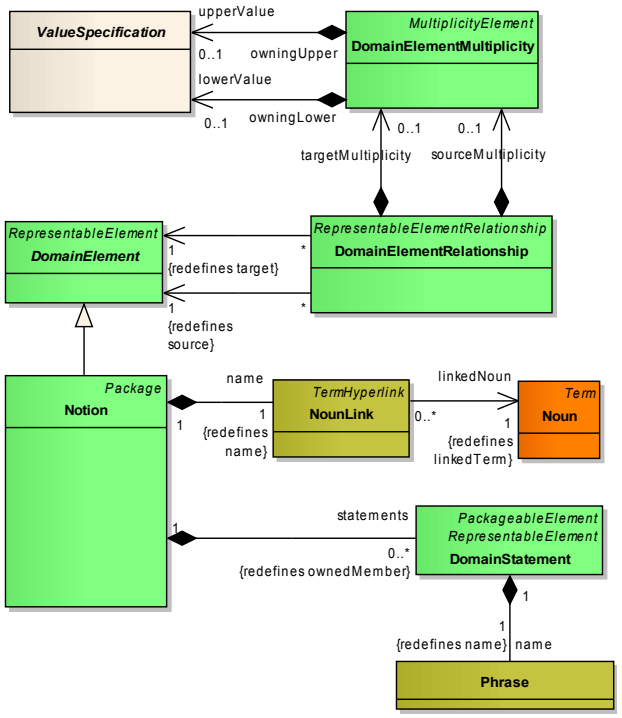

Figure 3. Notion metamodel

Exercises

Form of physical activity performed in a [[n:fitness club]]. Exercises may be [[m:cyclic n:exercises]] or [[m:sporadic n:exercises]]. [[m:Sporadic n:exercises]] must be reserved earlier (in eg. a private session with a [[n:coach]]).

[[m:cyclic n:exercises]]
[[m:sporadic n:exercises]]
[[v:check n:availability p:of n:exercises]]
[[v:submit n:sign-up p:for n:exercises]]
[[v:want to n:sign-up p:for n:exercises]]
[[v:sign-up n:customer p:for n:exercises]]

Figure 4. Example of textual notation

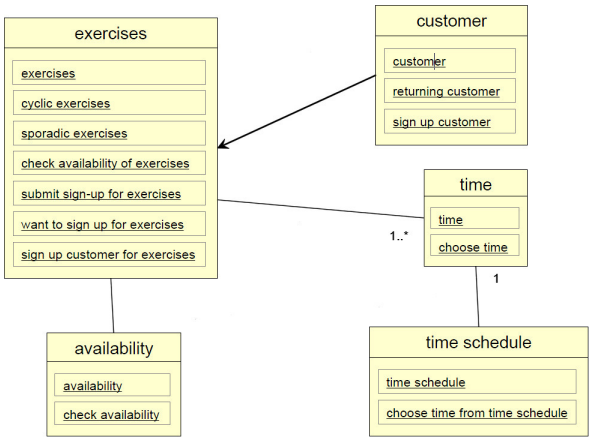

Figure 5. Example of graphical notation

sequences of hyperlinks pointing at external terms. These Terms (with their forms, which depend on the context) can be stored in an external, global structure. This structure, represented by a Terminology, defines the semantics of the Terms through giving relations between them, and can be based on existing dictionaries/ontologies (e.g. WordNet [17]). This way, the phrases can be subject to semantic-based matching, as described by Wolter et al. [18].

III. Structuring domain specifications with notions

To organise the phrases we will group them by the nouns defining the described domain entities. We will call such group a "notion". The appropriate metamodel for this part of the presented language is shown in Figure 3. Every Notion, which is a kind of UML :: Kernel :: Package, can include many DomainStatements referring to the same noun. The DomainStatements can have textual descriptions (see Section IV) and their names are the Phrases described above. The common Noun being an object for all the phrases within a notion is also linked as that notion's name (see the relevant NounLink).

To complete the domain structure, we need to define relationships between notions. This is done through DomainElementRelationshipss which denote relationships between two DomainElements. The relationship can be constrained by bounds of multiplicity and is directed. The source of the relationship has to be different than its target. The DomainElementRelationship can have constrained multiplicity described by the sourceMultiplicity and the targetMultiplicity. Similarly as in the UML metamodel [19], the DomainElementMultiplicity is a range consisting of two ValueSpecifications.

The above abstract syntax calls for appropriate concrete syntactic elements. Our metamodel introduces a special kind of structural domain representation that explicitly focuses on domain elements. It can be seen as possessing some of the key object-oriented principles: domain elements can be connected through domain element associations adorned with multiplicities. We could thus simply use UML class model notation. However, where a graphical notation is necessary, we propose a notation clearly distinguishing domain elements from classes. This is to stress its domain modelling (cf. ontological modelling) purpose. This is illustrated in Figures 4 and 5. The first Figure presents a textual description of a notion (Exercises) with several embedded phrases (e.g. cyclic exercises). It can be noted that the notion description contains phrases (represented by hyperlinks in the description). In Figure 5 we can see a graphical notation that includes the same notion. The phrases have a notation that clearly distinguishes them from e.g. class operations.

IV. Hyperlinking phrases to build specifications

The metamodel we have presented allows to organise the domain definition in the form of a dictionary of phrases. We have already shown that the phrases can be hypelinked from within the domain element descriptions

(see Fig. 4). However, this can be easily extended to any textual specification. For instance, we could organise this way the functional and the quality requirements. Through consistent use of hyperlinks we could significantly raise precision and unambiguity of such specifications. For this purpose we will thus extend the presented language to allow formulating full sentences constructed out of hyperlinks.

We have already seen that all the elements used in phrases link to the Terminology. In fact, phrases consist only of hyperlinks to specific Terms through the TermHyperlink construct (specialised by Modifier, Determiner, Object, PhraseVerb, and PhrasePreposition). This idea is extended to use phrases as targets of hypelinking and to construct specifications as sequences of hyperlinks to phrases. Now, instead of copying the same phrase in many places, we just point to its definition placed in a central domain specification. This provides consistency as every hyperlink may point at exactly one element. This is in line with the findings by Kaindl [20] which indicate that hyperlinks applied in requirements specifications are basic facilitators of coherence. However, the approach is beneficial only with strong tool support, which we will discuss in the following section.

The precision of system specifications is assured by using hyperlinks that link interaction flow descriptions with definitions of phrases. In textual specifications, this leads to the idea of a wiki-like language. Hyperlinks can be inserted into free-form text using the notation presented in the previous section.

Unfortunately, free (although hypelinked) text used in specifications has serious limitations. Namely, it is not suitable for automatic processing (e.g. translation into design or code, comparison, structured editing, semantic operations), and it can be formulated still in an unreadable way. To cater for these two problems we would need to introduce much more rigour and limit the language used. We will now present such a limited language with three types of sentences: SVO sentences, modal SVO sentences and conditional sentences. It can be argued that most of software requirements and descriptions could be covered by these three types. They will use phrases (or rather: hyperlinks to phrases) as their atomic "lexemes".

The overall structure of constrained language sentences is shown in Figure 6. They generally consist of Predicates that point to VerbPhrases and Subjects that point to regular (noun-only) Phrases. In addition, some sentences can contain ModalVerbs and ConditionalConjustions that point directly to appropriate Terms in the terminology.

The simplest SVOSentences contain just one Subject and one Predicate. This together results in a grammar that follows the Subject – Verb – Object (S-V-O) scheme, as proposed by Graham [21]. In such RSL sentence the Predicate is a hyperlink to a SimpleVerbPhrase, and the Subject hyperlinks to a Phrase. These phrases are further hyperlinked to appropriate terms in the terminology.

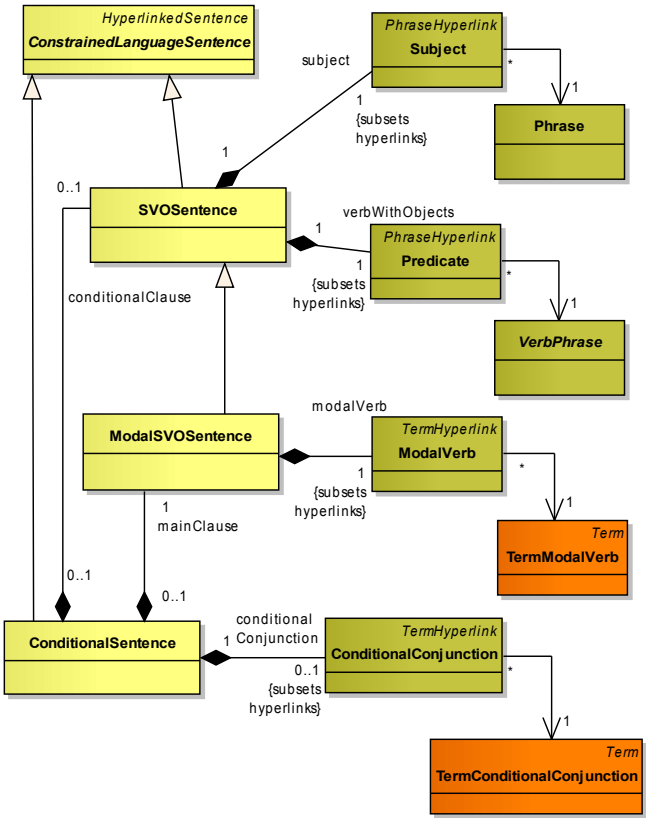

Figure 6. Constrained language sentence metamodel

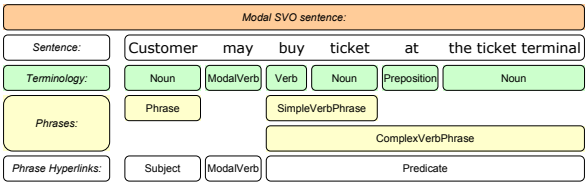

Figure 7. Example of ModalSVOSentence with ComplexVerbPhrase

We should also note that such sentences can contain ComplexVerbPhrases. In such situation, the sentence is extended by an additional indirect object (S-V-O-O) allowing to express more complex behaviour involving more than one noun phrase.

A ModalSVOSentence is an SVOSentence extended with a ModalVerb, as shown in Figure 7. This type of sentence adds a modality aspect to the information carried by an SVOSentence. This allows to express priority of the described activity, an attribute or circumstance that denotes mode or manner of the described activity or an obligation (a possibility) of the subject to perform an action (described by a Predicate). ModalSVOSentences can be used to describe different system's features – they can be used in order to capture non-functional requirement statements or high-level/system vision type statements. In Figure 7 we can additionally see an example of a sentence where a Predicate points to a ComplexVerbPhrase.

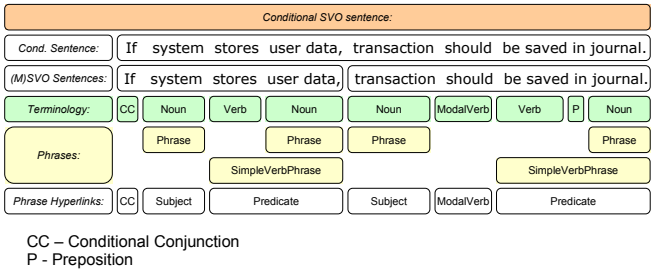

Figure 8. Example of ConditionalSentence

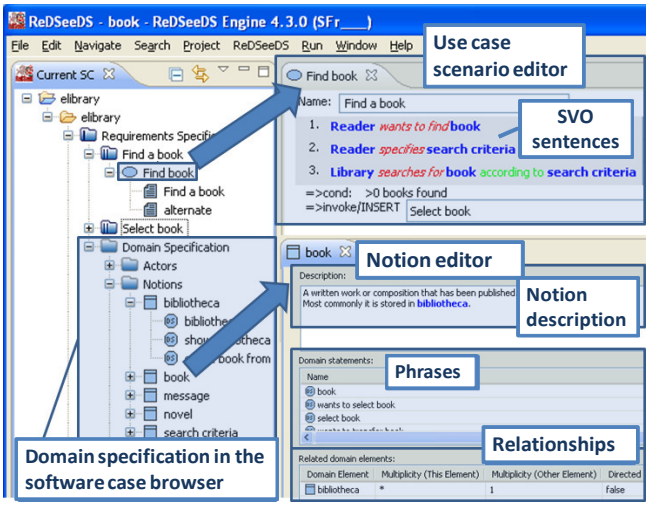

Figure 9. Editors and browsers of the ReDSeeDS tool

A ConditionalSentence is an ordered set of two SVO sentences. The first of these sentences (the conditional clause) describes a state, a possibility or in general – a condition in which the activity expressed in the second sentence (the main clause) can occur. The type of the condition is additionally determined by conditional conjunction linking these two sentences. The Terms like "if", "when", "upon", "during", can be used as conjunctions for conditional sentences. This is illustrated in Figure 8.

V. Tool-based evaluation

In order to evaluate the presented approach, a tool chain was constructed. Its main purpose was to allow for recovering the system essence and storing it using the notation and metamodel presented in the previous section. The central part of this tool chain is the ReDSeeDS tool (redseeds.sourceforge.net), that implements the RSL metamodel ([8], see section I). The tool offers a set of editors dedicated to different types of domain elements (see Figure 9, bottom-right, containing the notion editor with compartments for the notion's definition, phrases and relationships). The central point of the tool is the use case scenario editor (as illustrated in Figure 9, top-right). It allows writing use case scenarios consisting of sentences describing interaction flows within the software system.

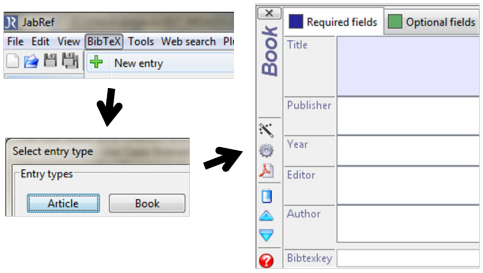

Figure 10. Elementary example for GUI interactions

Such sentences are created within the grammatical rules described above. The sentences are referencing domain specification elements: actors, notions and system elements (see sentence element colouring according to hyperlink types). The tool allows to manage the domain specification elements directly from the use case editor or using a typical tree-like browser (see Figure 9, left-middle).

The first step of the recovery process utilizing our tool chain is performed using a GUI-ripping tool (see a discussion by Memon et al. [22]). While performing this step, the GUI-ripping tool records the interactions representing the system's application logic. This includes the user inputs (buttons clicked, data entered, widget focus gained, etc.) and respective system responses (windows displayed, messages shown to the user or even textual console behaviour). A simple example of such user – system interaction is shown in Figure 10. We use the JabRef reference manager system (jabref.sourceforge.net) but any other example system with extensive user-system interaction could be used.

In our evaluation, for GUI-ripping, we have used a commercial test management tool (Rational Functional Tester, www.ibm.com/software/awdtools/tester/functional/). However, any tool allowing for interaction recording to some form of structured text files can be integrated with our software. The tool we used, records sequences of interactions into XML-based scripts, as shown in Figure 11. The next step is then to transform these scripts into the presented notation. This is done with the TALE tool (Tool for Application Logic Extraction) developed as part of this work. This novel tool can produce scenarios with SVO sentences, as illustrated in Figure 11 (bottom; the script reflects the interaction from Fig. 10). What is important, the tool can also extract information about the composition of specific notions used in the scripts. Namely, all the buttons and fields in the consecutive windows (e.g. "Select entry type" or "JabRef - untitled") can be stored as descriptions of respective noun phrases (discussion of this is out of scope of this paper).

The scenarios generated by the TALE tool can be joined into use cases and further processed in ReDSeeDS. The generated specification can be subject to manual modifications and additions such as use case model re-factoring, additions to the domain model, changes in scenario sen-

Figure 11. Elementary example of a recovered application logic script

tences, etc. After such modifications, the model contains both the still relevant "legacy" specifications and the "new" ones. This constitutes the "essence" of the application logic. We can now use this essence to generate a new system. In order to do this we need to reorganise the model according to the needs of the transformation rules.

The result of the process for the example from Figure 10 is summarised in Figure 12. The scenario has been manually extended with sentence 6 (see top-right). In addition, some of the generated notions (groups of phrases) have been identified as UI elements (see centre-left). We have also added a new domain element ("jabref untitled data"). It should be also noted that the phrases have been linked with relevant terms in the terminology (note the "validate" term in bottom-left). We should also note that most of the resulting "essential" model has been recovered automatically from the legacy system (here: JabRef).

In order to transform the presented model into a software system architecture we can use the rules formulated by Śmiałek [23]. According to one of the rules, each use case is transformed into an application logic (cf. controller/presenter in the Model-View-Controller/Presenter architectural patterns) class. The realisation of this simple rule is shown in Figure 12 (see the CAddBookEntry class at bottom-left). We can even go further and generate important parts of dynamic code, as it was shown recently by Šimko et al. [24] and Śmiałek [25]. For instance, Figure 13 presents a small fragment of application logic code generated automatically from the essential model in Figure 12. As it can be seen, all the "user" sentences (1, 3 and 5) were transformed into operations in the controller class. Furthermore, the "system" sentences (2, 4 and 6) were transformed into operation calls to appropriate "view" (denoted by "v") or "model" (denoted by "m") objects. The resulting code can be fully operational in regard to the application logic, i.e. it can fully control all the flows of user-system interaction. What is important, the code can also contain decisions ("if" statements) that control the interaction flow depending on the user decisions or the

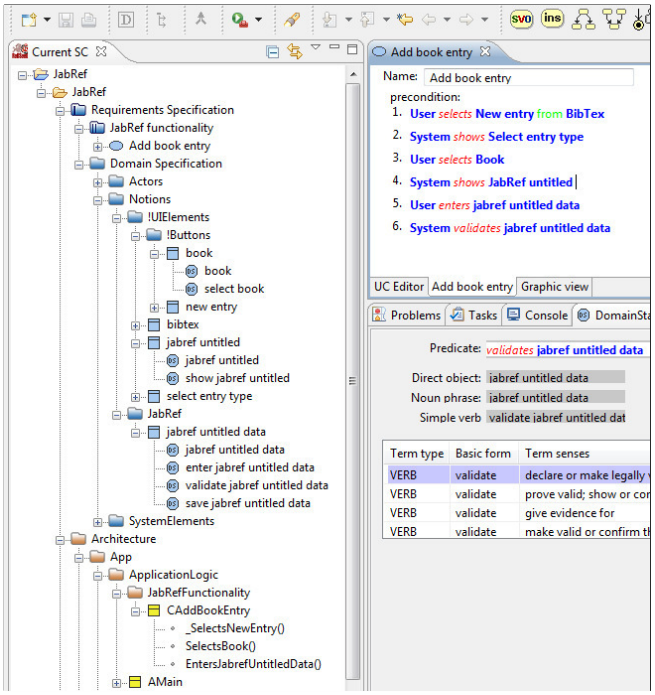

Figure 12. Recovered script and domain specification in the ReDSeeDS editor

CAddBookEntry

```
public void _SelectsNewEntry(){
  vSelectEntryType.show();
}
public void SelectsBook(){
  vJabRefuntitled.show();
}
public void EntersJabRefuntitledData
  (XJabRefuntitledData aJabRefuntitledData) {
  mJabRefuntitledData.validate
   (aJabRefuntitledData);
}
```

Figure 13. Code generated from the example recovered script

current system state. Such decisions can be generated on the basis of alternative scenarios, but a detailed discussion is out of scope of this paper.

By generating code we finalise the process that is summarised in Figure 14. Throughout this process we use the "essential" specifications according to the Application Logic (AL) language presented in the previous section. We first analyse the legacy system's UI by using a GUI ripping tool. Based on this semi-automatic analysis we generate the initial AL model. This model can then be manually updated (edited) at the "essential" level to cater for new or changed functionality. This model is based on a metamodel and thus we can use model transformation tools to generate more detailed (e.g. technology/platform specific) models and code. In the presented tool chain we use the MOLA transformation language [26] and the associated transformation engine.

By using the presented tooling environment several studies are currently undertaken. First, there is performed

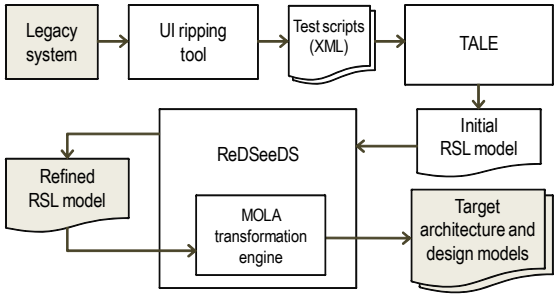

Figure 14. Overview of the application logic handling process

a larger case study based on a legacy credit management system, used currently by a major Polish bank. This study is performed in cooperation with Infovide-Matrix S.A. (large Polish software consultancy/provider). The system's observable application logic has been already semi-automatically translated into RSL scenarios and the new system architecture has been generated. The current results show very promising levels of application logic that can be recovered from a legacy system. What is important, this recovery is to large extent automatic. Furthermore, the recovered logic is brought to the level of requirements understandable to the users. It was already shown by Jedlitschka et al. [27] that such structured specifications with precisely defined domain vocabularies are well accepted as simply being a better way of expressing requirements. While working within such a "discover notions – write structured sentences" framework, the analysts are encouraged to be acquainted with software system's environment and are stimulated to write precise, clearly formulated requirements statements.

Further studies were performed with students attending the "Model Driven Software Engineering" course at the Warsaw University of Technology. During this course, the students were asked to develop applications by using the presented Application Logic language using the ReDSeeDS tool. The students were able to write precise scenarios in the presented notation, for up to 12 use cases within the course of a 4-hour lab session. Furthermore, they were able to generate and implement essentially functional applications for around half of these use cases, within further 10 hours of lab sessions. The applications had all the necessary application logic and were performing elementary data handling (store-read) operations.

VI. Conclusion and future work

The presented Application Logic language aims at capturing the essence of the system's functionality. It can be noted that the specifications are written at the level of detailed functional requirements. What is important, these requirements are written in near-natural language thus making it accessible to the end-users (see relevant work by Śmiałek [28]). At the same time, specifications are based very coherently on the domain definition by pointing to centrally defined domain statements (phrases). To define the application logic, the specifications can contain only pointers (hyperlinks) to centrally defined noun and verb phrases. A sequence of such hyperlinks forms a scenario describing the user-system interactions. Our experience shows that such application logic scenarios are easy to write by inexperienced developers (analysts) and even the end-users. This can be done using any tool that allows for hyperlink management. This prominently includes wiki systems, but also some CASE tools enable this (see e.g. the scenario editor of Enterprise Architect, www.sparxsystems.com).

Writing scenarios hyperlinked to a central vocabulary gives important element of coherence to specifications. However, in order to be able to perform automatic transformations or semantic-based matching [18], we need a tool that implements the presented (or analogous) metamodel. In the current work we have shown that it is also possible to use such a tool as a repository for essential application logic recovered from legacy systems. This repository gives an additional advantage of generating code directly from high-level scenarios. This includes not only the code structure (classes, method signatures) but also the dynamics (method bodies) for the application logic layer.

It can be noted that the presented results can be extended in the direction of creating a more expressive language at the "essential" level. It has to be stressed that the language is not meant for data processing. Thus, it will not possess typical data-processing constructs like loops or variables. Instead, it concentrates on capturing application logic, where loops are implicit through repeated system-user interaction. However, in the currently ongoing work, the scenarios will be extended to include conditional sentences thus adding decision capabilities to the language. Moreover, the language will gain the capability to combine scenarios into UML-style use case models with relationships between use cases. Additional sentences will enable use case "invocations" shifting the flow of interactions from one use case to another. This can be combined with the work by Ambroziewicz [29] on application logic patterns. The presented language can be used as a pattern language where the noun and verb phrases can be abstracted from a particular problem domain. The patterns can operate on a generalised domain and then can be instantiated for a specific domain.

Future work will also include extending the TALE tool to be able to recover scenarios combined into use cases on the basis of analysis of GUI-ripping results. It will also consist in extending the language into a language fully capable of performing "programming" at the level of essential application logic. The goal is to move much of such programming activity to a significantly higher level of abstraction than currently. This way, the application logic programming can become accessible even to the end-users. It has to be noted that this language would not yet capture all the essence of a software system functionality. The domain logic will not be expressed in any way. The

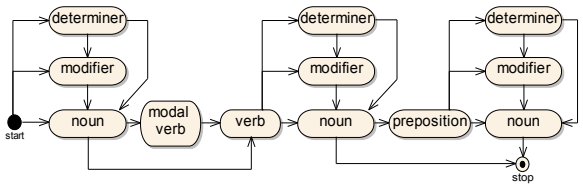

Figure 15. Simple SVO sentence grammar state machine

domain statements would indicate the necessary domain functionality (data processing algorithms etc.), but not define this functionality.

Finally, it has to be noted that the SVO grammar is a kind of controlled language with formal grammar as presented in Figure 15 (see also e.g. work by Fuchs et al. [30] or Sleator and Temperley [31]). As such it does have some difficulties with reflecting different natural languages. Some heavily inflected languages, like Polish, need suffixes and prefixes for words, even in sentences with similar structure and meaning. Another problem is that some languages (e.g. German, Turkish) allow for different order of words in a sentence. This can be solved by adding attributes to sentence classes, indicating word order or language used for this sentence. Phrases themselves pose some problems while considering natural languages. Determiner and Modifier classes can constitute different parts of speech and also multi-word verbs and nouns are constructed in variety of manners in different languages. Handling of multi-language specifications is thus a very interesting challenge for future research.

Acknowledgment

This research has been carried out in the REMICS project (http://www.remics.eu) and partially funded by the EU (ICT-257793 under the 7th Framework Programme).

References

[1] F. P. Brooks, "No silver bullet: Essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, April 1987.

[2] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems," *Comput. Stand. Interfaces*, vol. 33, no. 6, pp. 519–532, Nov. 2011.

[3] E. Evans, *Domain Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

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

[5] D. Bjôrner, "Rôle of domain engineering in software development. why current requirements engineering is flawed!" *Lecture Notes in Computer Science*, vol. 5947, pp. 2–34, 2010, PSI 2009.

[6] M. Śmiałek, J. Bojarski, W. Nowakowski, and T. Straszak, "Writing coherent user stories with tool support," *Lecture Notes in Computer Science*, vol. 3556, pp. 247–250, 2005, xP'05.

[7] C. A. Iglesias, J. I. Fernández-Villamor, D. Pozo, L. Garulli, and B. García, "Combining domain-driven design and mashups for service development," in *Service Engineering*, S. Dustdar and F. Li, Eds. Springer Vienna, 2011, pp. 171–200.

[8] H. Kaindl, M. Śmiałek, , P. Wagner, and et al., "Requirements specification language definition," ReDSeeDS Project, Project Deliverable D2.4.2, 2009, www.redseeds.eu.

[9] C. W. Chan, "Knowledge and software modeling using UML," *Software and Systems Modeling*, vol. 3, no. 4, pp. 294–302, 2004.

[10] Resource Description Framework (RDF). W3C. [Online]. Available: http://www.w3.org/RDF/

[11] J. Evermann and Y. Wand, "Toward formalizing domain modeling semantics in language syntax," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 21–37, January 2005.

[12] H. Kaindl and M. Snaprud, "Hypertext and structured object representation: A unifying view," in *Proceedings of the Third ACM Conference on Hypertext (Hypertext '91)*, San Antonio, TX, December 1991, pp. 345–358.

[13] A. G. Kleppe, J. B. Warmer, and B. W, *MDA Explained, The Model Driven Architecture: Practice and Promise*. Boston: Addison-Wesley, 2003.

[14] R. M. Dixon, *A new approach to English Grammar, on semantic principles*. Oxford University Press, 1991.

[15] D. Bjôrner, *Software Engineering 3: Domains, Requirements, and Software Design*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[16] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008.

[17] C. Fellbaum, Ed., *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[18] K. Wolter, M. Śmiałek, L. Hotz, S. Knab, J. Bojarski, and W. Nowakowski, "Mapping MOF-based requirements representations to ontologies for software reuse," in *CEUR Workshop Proceedings (TWOMDE'09)*, vol. 531, 2009.

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

[20] H. Kaindl, "Using hypertext for semiformal representation in requirements engineering practice," *The New Review of Hypermedia and Multimedia*, vol. 2, pp. 149–173, 1996.

[21] I. M. Graham, "Task scripts, use cases and scenarios in object-oriented analysis," *Object-Oriented Systems*, vol. 3, no. 3, pp. 123–142, 1996.

[22] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, Nov. 2003, pp. 260–269.

[23] M. Śmiałek, *Software Development with Reusable Requirements-Based Cases*. Publishing House of the Warsaw University of Technology, 2007. [Online]. Available: http://bcpw.bg.pw.edu.pl/Content/2098/Smialek_Habil.pdf

[24] V. Šimko, P. Hnětynka, and T. Bureš, "From textual use-cases to component-based applications," *Studies in Computational Intelligence*, vol. 295, pp. 23–37, 2010.

[25] M. Śmiałek, "Requirements-level programming for rapid software evolution," in *Databases and Information Systems VI*, J. Barzdins and M. Kirikova, Eds. IOS Press, 2011, ch. 3, pp. 37–51.

[26] A. Kalnins, J. Barzdins, and E. Celms, "Model transformation language MOLA," *Lecture Notes in Computer Science*, vol. 3599, pp. 14–28, 2004, mDAFA'04.

[27] A. Jedlitschka, K. S. Mukasa, and S. Weber, "Case reuse verification and validation report," ReDSeeDS Project, Project Deliverable D6.2, 2009, www.redseeds.eu.

[28] M. Śmiałek, "Accommodating informality with necessary precision in use case scenarios," *Journal of Object Technology*, vol. 4, no. 6, pp. 59–67, August 2005.

[29] A. Ambroziewicz and M. Śmiałek, "Application logic patterns – reusable elements of user-system interaction," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, 2010, vol. 6394, pp. 241–255.

[30] N. E. Fuchs, S. Höfler, K. Kaljurand, F. Rinaldi, and G. Schneider, "Attempto controlled english: A knowledge representation language readable by humans and machines," *Lecture Notes in Computer Science*, vol. 3564, pp. 213–250, 2005.

[31] D. D. K. Sleator and D. Temperley, "Parsing english with a link grammar," Department of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-91-196, 1991.