Runtime Semantics of Use Case Stories

Michał Śmiałek, Norbert Jarzębowski, Wiktor Nowakowski
Warsaw University of Technology, Warsaw, Poland
Email: {smialek, jarzebon, nowakoww}@iem.pw.edu.pl

***Abstract*—Direct end-user participation in software system construction necessitates bringing general-purpose programming activities to the level understandable by "laymen". This paper introduces a new software development approach where stories written in commonly understood structured natural language gain runtime semantics. Stories are precisely linked to domain concepts and actions, thus forming the application logic of the system. These constructs are written at a high level of abstraction, very close to detailed software requirements specifications. In fact, they are structured into familiar use case models that include special "invocation" relationships between use cases. At the same time, the paper proposes precise translational semantics for such defined stories. For each story element, equivalent Java code is derived. This semantics has been implemented within a tool containing a story editor and a code generator producing fully dynamic application logic code with Swing-based user interface. Ease of use of the new story language and usefulness of the generated Java code has been evaluated through student assignment projects.**

I. Introduction and related work

Use case modeling is one of the most commonly used visual notations for reaching understanding between the end-users and software developers. Individual use cases within the use case models represent units of observable behavior of the system-to-be-built and thus define the system's functional requirements. Use case descriptions tend to become more and more precise thus improving their applicability in the software development chain (see e.g. [1], [2], [3]).

Introducing precision is associated with automating the translation of use case models and their descriptions into various other models and even code. The first group of attempts at this issue consists in generating analysis models, like conceptual class diagrams and some dynamic diagrams (eg. state machines), as reviewed by Yue et al. [4]. There also exist (quite sparse) approaches to deriving design models. Śmiałek [5] gives a detailed set of rules for transforming into architectural design models, including dynamic sequence diagrams. Šimko et al. [6] propose a simple meta-model for describing textual use case descriptions and transforming them into static component models. A similar approach, but extended with natural language parsing and generating service-oriented component models was proposed by Ding et al. [7] (see also de Castro et al. [8]).

Ultimately, this leads to relating use case descriptions to code. In some approaches, use case models (as such) are derived from code by applying certain reverse-engineering techniques (see e.g. [9]). Recently there have emerged approaches to generate code directly from use case descriptions. Franců and Hnětynka [10] propose a framework to produce three-tier-based code. Textual scenarios are first translated into special trace scripts ("procases") and then into the contents of application logic layer class methods. Similarly, Śmiałek [11] proposes a set of rules to translate constrained natural language scenarios directly into Java code. It can be noted that within these two approaches (although not specified directly), the textual scenarios gain certain runtime semantics, which allows their translation into code (cf. works by Sinning et al. [12] and Liu et al. [13]).

In this paper we propose to unify and summarize the above research directions. We follow precision of previous notations like the Essential Use Cases [14] and we add strict **runtime** semantics. This way, we in fact introduce a new approach to programming, that is done at the level of use case **stories** translatable into executable code. To fulfill this, we use the Requirements Specification Language (RSL) [15] that is already equipped with all the necessary notation and is defined through a precise meta-model. On top of this language we propose detailed well-formedness rules and translational semantics [16] for individual language constructs. We use Java with Swing as the target for our translation. In contrast to previous approaches we attempt at comprehensive treatment of use case models as programmatic entities. This includes clarifying use case semantics as pointed out by Simons [17], also using the results of Metz et al. [18], [19].

II. Story notation

Stories introduced in this work are a variant of Scenario Representations found in RSL. The basic structure is left from RSL, but certain well-formedness constraints are applied. To explain the notation we will use the model in Figure 1. It presents an excerpt containing three use cases of a Campus Management System. Two of the use cases are invoked from the "Show course list" use case. The «invoke» relationship substitutes the «extend» and «include» relationships known from UML (see further description in this section).

Figure 2 shows stories for one of the use cases. Every story consists of a sequence of numbered sentences. There are three basic types of sentences: A). *subject-verb-(direct)object* (s-v-o, see sentences 1, 4, 5, 4.1.1, 4.1.2); B). *subject-verb-object-(indirect)object* (s-v-o-o, see sentences 2, 3); C). *condition* (marked with "cond"). The s-v-o(-o) sentences describe interaction between the actor and the system, and system responses. They can be additionally divided into three types.

- Actor-to-system sentence (see sent. 1, 3, 4.1.2): the subject points to an actor, and the direct object points to a user interface element (button, option, etc.). Every story

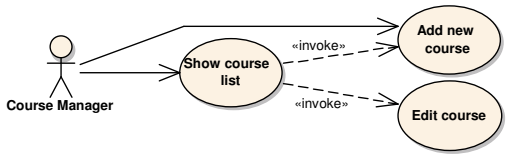

Figure 1. Example use case model of the application logic

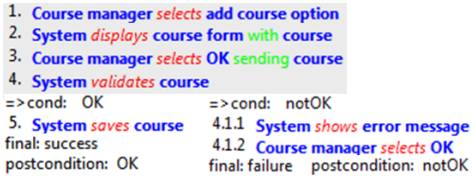

Figure 2. Textual stories for "Add new course"

should start with such a sentence, and such sentences (one or more) should follow any system-to-actor sentence.

- System-to-actor (see sent. 2, 4.1.1): the subject points to the system, and the direct object points to a user interface element (window, form, etc.) . Such sentences can follow any other sentence in a story.
- System-to-system (see sent. 4, 5): the subject points to the system, and the direct object points to a domain element. Such sentences can follow only actor-to-system and other system-to-system sentences.

The optional indirect object in any of the above types of sentences should point to a domain element.

The condition sentences can follow any system-to-system and system-to-actor sentence. They always exist in sets of at least two such sentences causing alternative stories. They consist of the "cond" keyword followed by a single free-text word that needs to be unique in a given set of alternative conditions. For instance, Figure 2 contains in fact two stories that split after sentence 4.

The "Add new course" use case can be treated as "stand-alone" but it is also invoked from "Show course list". This is reflected in the stories that contain special invocation and rejoin sentences. Invocation sentences allow to pass control to the stories of another use case. The rejoin sentences can be used to show the story sentences at which an alternative story (caused e.g. by an invocation) joins the basic scenario. This is illustrated in Figure 3, where the two invoke relationships from Figure 1 are reflected by two invocation sentences denoted by "invoke/INSERT". There are three alternative scenarios, visualized also in Figure 4. The Figure illustrates possible flow of control depending on the actor choosing to invoke a use case or select an option (here: the "OK" option in sentence 3.2.1). It also shows the flow that depends on how the invoked use case ends (compare with the endings in Fig. 2). The flow is controlled by appropriate condition sentences that need to precede the invocation and rejoin sentences.

III. Story Translational Semantics

In order to define runtime semantics for the above story notation we will use Java (with Swing, whenever the user interface elements are involved). It will be treated as an "assembly

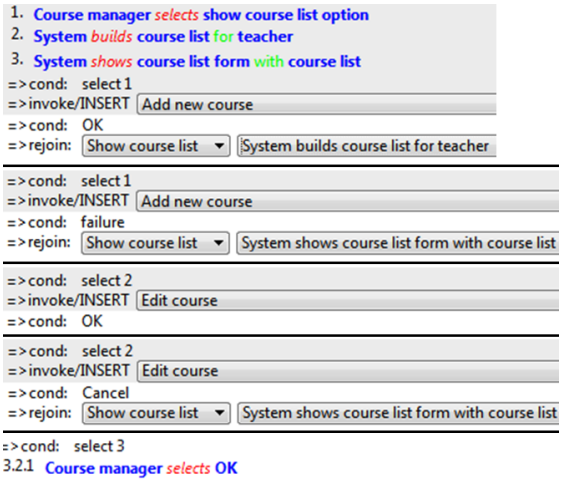

Figure 3. Textual stories for "Show course list"

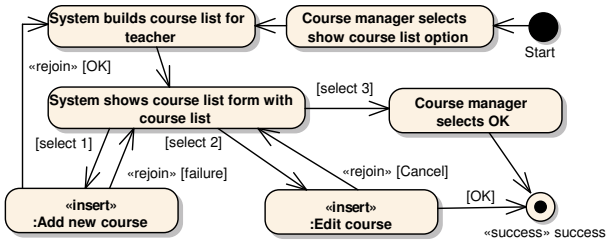

Figure 4. Visualized stories for "Show course list"

language" to which constructs of our new concept-oriented language will be translated. Since the runtime semantics of Java is well understood, such translation will also provide the runtime semantics for stories. Obviously, in implementation, a potential story compiler can use any target language (including machine code) based on the translation from the Java/Swing semantics.

We will also use UML to facilitate understanding the structure of the translated code. The semantics will be organized into a set of 6 semi-formal translation rules. The rules cover the semantics of use cases as such, s-v-o(-o) sentences and condition sentences. This set can be additionally extended with the rules for the invocation and rejoin sentences, but this is out of scope of this paper. This constitutes a complete and minimal set, enabling the construction of a compiler for the presented story language. In Figures 5 and 6, the translation examples are illustrated with arrows adorned with the respective rule numbers.

Rule 1. *Every use case is translated into an application logic ("C" - controller) class. Invocation relationships between use cases are translated into associations (cf. UML) navigable both ways.* This is illustrated in Figure 7 that shows a UML diagram equivalent to a Java code skeleton. The two presented "C" classes are translated from two of the three use cases of Figure 1. Their operations shown in the diagram were created by using the remaining rules.

Rule 2. *Every actor-to-system sentence in a given use case story is translated into an operation (method decla-*

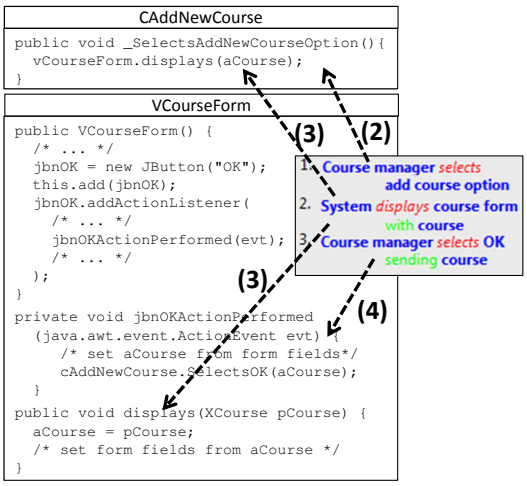

Figure 5. Translation of actor-to-system and system-to-actor sentences

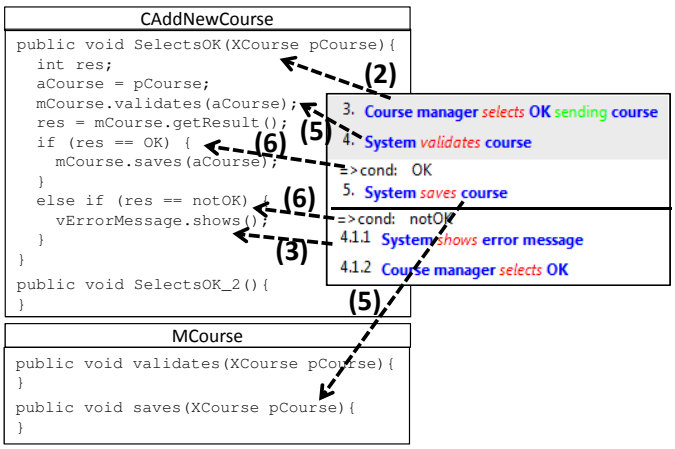

Figure 6. Translation of system-to-system and condition sentences

ration/header) within the respective "C" class (generated according to Rule 1). The operation name is constructed from the verb–direct-object phrase. If the sentence contains also an indirect-object, it is translated into the operation parameter. The parameter is a Data Transfer Object, where its type and name are derived from the respective concept name, associated with the indirect-object.

Rule 3. *Every system-to-actor sentence is transformed into an asynchronous call to a method in a presentation layer ("V" - view) class. The call is placed within the body of a method translated form the preceding actor-to-system sentence. The variable name in the call is taken from the direct-object name, while the method name is taken from the verb name. If the system-to-actor sentence has an indirect object, it is translated into a call parameter similar to that in Rule 2.*

Rule 4. *Every actor-to-system sentence directly following a system-to-actor sentence is transformed into a button declaration with an associated event handler function. The button is declared within the "V" class containing the method where a call to this method was generated according to Rule 3. The event handler contains an asynchronous call to a method generated according to Rule 2.*

Rule 5. *Every system-to-system sentence is transformed into a synchronous call to a method in a domain logic ("M" - model) class. The call is placed within the body of a method translated from the preceding actor-to-system sentence. The result of the call is made available for possible further decisions. The naming of the call and handling of the indirect-object part of the sentence is identical to that in Rule 3.*

Rule 6. *Every set of condition sentences directly following a system-to-system sentence is translated into an "if-else" (or: "switch") set of statements. The conditions within the "if" statements compare the results of the call generated according to Rule 5 with the consecutive condition sentence contents. Further sentences in alternative stories are inserted within the appropriate "if" sections, according to Rules 3-5 until the stories are finished or actor-to-system sentence found.*

The Rules from 2 to 6 result in adding new classes to

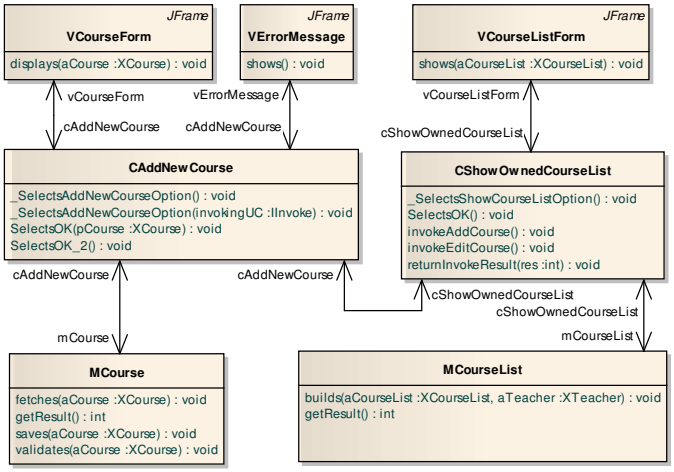

Figure 7. Presentation and domain logic classes generated from concepts

the translated code. An overview of the full code structure (simplified for brevity and clarity) is presented in Figure 7. The structure follows the MVC/MVP three-tier architectural pattern, and the initial letters in the class names indicate their inclusion into each of the tiers. The presented rules go far beyond generating this static structure only. It should be noted that the dynamic code (method bodies) of the middle tier ("C") is complete and does not necessitate any further manual intervention. Also some important elements of the upper tier ("V") are ready and complete.

IV. Conclusion: Evaluation and Future Work

The presented story language abstracts over many technical issues that need to be handled within procedural and object-oriented code. This is in line and significantly extends model-driven approaches to end-user development [20]. With the presented semantics, coding capabilities are shifted to the level of use cases traditionally used for specifying requirements. This way, application logic coding becomes accessible to "ordinary people" simply allowing them to tell stories about possible observable system behaviour. Though, it has to be

noted that the language does not offer domain logic (data processing) capabilities. It allows to generate the "V" and "C" layers of the system but the "M" layer has to be coded in a traditional way. However, the structure of the code to be developed (including operation signatures) is generated and can be easily interfaced with the fully generated part.

To evaluate the presented semantic rules there was created a story translator. It was built on top of an RSL editor, developed within the ReDSeeDS project (www.redseeds.eu, see also [21]). The editor allows for entering stories that comply with the presented rules. The stories can be taken as input to a model-based transformation written in the MOLA language [22] that is native for the ReDSeeDS transformation environment. The developed translation creates full structure of the code (as e.g. in Figure 7) and **complete** dynamic code of the two upper layers. This includes crude UI form layouts based on domain element definitions (a detailed description of specific rules is out of scope of this paper).

The story language and the transformation was used by 28 postgraduate CS students attending a course on Model-Driven Software Development. The students were instructed on RSL story constructs and had previously gained knowledge about constructing MVC/MVP style systems, using UML and Java. During the classes, they were formed into 8 groups consisting of 3-4 students each, with evenly distributed skills. All the groups were assigned a ready use case model of a Campus Management System, containing 12 use cases (some of them shown in Fig. 1). The students were asked to write stories for the use cases using the presented language. Half of the groups used the RSL editor, and half - a standard editor of a UML tool. Regardless of the tool used, the students managed to write more than 10 sentences per use case within a two hour session. Next, the students were asked to develop code for the use cases within 10 hours (5 two-hour sessions). The acceptance criteria for the resulting code included producing debug-style messages that would show flow of control within the system. The groups that used the presented transformation performed significantly better implementing code for 68 sentences on average. The other groups that used manual translation to code and transformation into design models only, produced from 21 to 28 sentences on average.

Thus, the current results show that the presented approach can have significant potential in enhancing productivity of programming. The experiment shows that quite inexperienced programmers and software designers (students) can benefit from removing the need to structure their code and write the application logic part. Instead of finding technological ways to implement the end-user functional requirements, they can concentrate on writing the data processing (domain logic) code. The presented approach allows for generating a ready skeleton for such code. Though, it can be noted that also this code could potentially be developed at the conceptual level together with the end-users. We leave this for future work that could ultimately lead to fulfilling the vision of rapid translation from informal stories to ready systems as proposed in [23].

Acknowledgment

This research has been carried out in the REMICS project and partially funded by the EU (contract no. ICT-257793 under the 7th Framework Programme), see http://www.remics.eu/.

REFERENCES

[1] K. T. Phalp, J. Vincent, and K. Cox, "Improving the quality of use case descriptions: empirical assessment of writing guidelines," *Software Quality Journal*, vol. 15, pp. 383–399, 2007.

[2] M. Kamalrudin, J. Hosking, and J. Grundy, "Improving requirements quality using essential use case interaction patterns," in *Proc. 33rd International Conference on Software Engineering*, 2011, pp. 531–540.

[3] M. Śmiałek, J. Bojarski, W. Nowakowski, A. Ambroziewicz, and T. Straszak, "Complementary use case scenario representations based on domain vocabularies," *Lecture Notes in Computer Science*, vol. 4735, pp. 544–558, 2007, MODELS'07.

[4] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requirements Engineering*, vol. 16, no. 2, pp. 75–99, 2011.

[5] M. Śmiałek, *Software Development with Reusable Requirements-Based Cases*. Publishing House of the Warsaw Uni. Tech., 2007.

[6] 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.

[7] Z. Ding, M. Jiang, and J. Palsberg, "From textual use cases to service component models," in *Proc. 3rd Int. Workshop on Principles of Engineering Service-Oriented Systems*, 2011, pp. 8–14.

[8] V. de Castro, E. Marcos, and J. M. Vara, "Applying CIM-to-PIM model transformations for the service-oriented development of information systems," *Information and Soft. Technol.*, vol. 53, pp. 87–105, 2011.

[9] R. Hirschfeld, M. Perscheid, and M. Haupt, "Explicit use-case representation in object-oriented programming languages," in *Proc. 7th Symp. on Dynamic Languages*, 2011, pp. 51–60.

[10] J. Franců and P. Hnětynka, "Automated generation of implementation from textual system requirements," in *Software Engineering Techniques*, ser. Lecture Notes in Computer Science, 2011, vol. 4980, pp. 34–47.

[11] 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.

[12] D. Sinnig, P. Chalin, and F. Khendek, "LTS semantics for use case models," in *Proc. 2009 ACM Symp. on Applied Computing*, 2009, pp. 365–370.

[13] H. Liu and H. Lieberman, "Programmatic semantics for natural language interfaces," in *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, 2005, pp. 1597–1600.

[14] L. L. Constantine, "Essential modeling: use cases for user interfaces," *interactions*, vol. 2, no. 2, pp. 34–46, 1995.

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

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

[17] A. J. H. Simons, "Use cases considered harmful," in *Proc. 29th Conf. TOOLS Europe*, 1999, pp. 194–203.

[18] P. Metz, J. O'Brien, and W. Weber, "Specifying use case interaction: Types of alternative courses," *Journal of Object Technology*, vol. 2, no. 2, pp. 111–131, March-April 2003.

[19] ——, "Specifying use case interaction: Clarifying extension points and rejoin points," *Journal of Object Technology*, vol. 3, no. 5, pp. 87–102, May-June 2004.

[20] F. Perez, P. Valderas, and J. Fons, "Allowing end-users to participate within model-driven development approaches," in *Proc. 2011 IEEE VL/HCC Symposium*, 2011, pp. 187 –190.

[21] M. Śmiałek, A. Kalnins, A. Ambroziewicz, T. Straszak, and K. Wolter, "Comprehensive system for systematic case-driven software reuse," *Lecture Notes in Computer Science*, vol. 5901, pp. 697–708, 2010, SOFSEM'10.

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

[23] M. Śmiałek, "From user stories to code in one day?" *Lecture Notes in Computer Science*, vol. 3556, pp. 38–47, 2005, XP'05.