Software Evolution based on Requirements-Level Programming

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

Abstract. The most important reason for software systems to evolve is the change in user requirements. The ability to react quickly to changing requirements is crucial for any effective software development lifecycle. In this paper we will present a coherent set of technologies for shortening the path from evolving requirements to code. The most important novel element on this path is a language defined at the level of requirements (understandable for non-IT experts) that is equipped with runtime semantics. This means that it is possible to translate specifications written in this language, automatically into partially executable code. The language also allows for easy detection of changes in requirements. This detection can be propagated down to the code structure and appropriate code parts (these that are not automatically generated) indicated for rework. It will be demonstrated that the presented approach is effective and suitable for a wide range of problem domains as opposed to domain-specific approaches. This will be shown through a case study for a typical business software system, performed with a novel tool suite.

1 Introduction and related work

Evolution of software is always associated with performing changes to code. Despite different natures of these changes (extending functionality, refactoring, correcting, etc.), the source is always the same: requirements. Changes to requirements cause changes to the system.

In a rapidly changing world, it is important that software can react quickly to changes. This can be done by reducing the effort to “translate” requirements into the final system. Traditionally, this is done through a manual process of analysing natural language text and based on this analysis - designing the system and producing code. In the recent years, there were introduced prominent approaches to automate this path. In Model-Driven Architecture (MDA; see [1]) there was introduced the concept of several modelling layers at different levels of abstraction. The first level is independent of the computations (cf. requirements), and the subsequent layers add certain design details leading to the final code. Models at each of the layers are produced in a series of automatic transformations. This leads to organising software development around artifacts that are models (see a wide overview in [2]).
Model Driven Software Development introduces at least partial automation in reaching code from higher level models. In Domain-Specific Modelling, this is shifted even further, so that the production of code is fully automated (see eg. [3] for a good overview). This way, models become the actual code. However, this is achieved at a price of reducing the applicability of a given modelling language and associated models, to just one, relatively limited problem domain (cf. domain specificity). This means that models produced for one class of systems cannot be reused for other kinds, and the software developers cannot shift easily from one problem domain to another. Domain-Specific Modelling is closely related to automated Software Product Lines (see [4]) or Software Factories (see [5]). In this approach, software is built in an automated way so that a series of similar (evolving) systems can be developed. Every new system is built as a variant of a base system and only some changes to the intermediate models have to be made. Software product lines and software factories are part of a more general trend to organise software evolution (see [6] for a wide overview of this field). In general, we would like to automate the process in which software changes. Unfortunately, most approaches concentrate on the evolution of design artifacts and code. Requirements are treated as second-class citizens in this process. Requirements-based evolution of software is not a frequent subject of research (see [6] for its position within the overall research area and [7] for the current state-of-the-art in requirements evolution).

In [7] it is proposed that a comprehensive tool infrastructure for requirements evolution is built. This paper replies to this research direction in proposing an infrastructure for treating requirements as first-class artifacts. This means that requirements are introduced directly into the path of automatic transformations. This is achieved by structuring natural language and giving it runtime semantics. The proposed infrastructure includes automatic transition from domain-independent requirements to design and code. It can be noted that also some previous approaches treat the issues of interplay between requirements and design (see [8] and [9] for important insights). In [10], a process for evolving requirements with separate domain models and associated implementation is presented. Unlike for our approach, automatic transformations and pointers to necessary changes in code are not handled. In [11], theoretical foundations for the evolution of requirements (resulting in their well-formedness) are introduced. Traceability of evolving requirements is also discussed, although again, no automation is introduced. Requirements evolution can also be related to product family construction which is based on requirements variability analysis (see eg. [12]).

This paper follows and significantly extends the process presented in [13] which advocates high levels of automation on the path from requirements to code. We present a requirements specification language where the most detailed specifications can be treated as the first step towards specifying the problem solution (as opposed to the problem definition), which is discussed in [14]. This first step is then translated into code that evolves together with requirements (see [15] for a relevant insight). In the following sections we present the syntax
Fig. 1. Application logic and domain logic layers of the presented language

of an appropriate requirements specification language and introduce its runtime semantics. This is an important contribution of this paper where a general-purpose language at a high level of abstraction (much higher than the current general purpose programming languages) has its runtime semantics defined. This definition is conducted by giving precise rules for transforming specifications in this language into compilable code. Based on this runtime capabilities we also present a schema that introduces high levels of automation into software evolution controlled through requirements.

2 Programming at the requirements level

In Domain-Specific Modelling, models become the de-facto code. This approach of shifting the level at which programming is done allows for achieving very significant raise in productivity (as pointed out in [3]). This is, though, achieved at a cost of learning and maintaining self-designed modelling languages. We thus would like to design a general-purpose modelling language suitable for various problem domains. At the same time, this language should be understandable by domain experts (especially including non-IT specialists) who participate in specifying requirements for software systems.

For the above reasons, we will base our design of a requirements-level programming language on a widely accepted approach to specifying functional requirements. We will apply use cases (see [16] for the initial idea) which are also part of the widely used Unified Modelling Language (see [17] for the official specification). Use cases are normally not seen as first-class modelling artifacts and do not normally participate in the model transformation paths mentioned in the previous section. This is because the use case representations have quite vague semantics and approaches to clarify this are sparse (see [18] for one of the few). In [19] there was proposed an extension to the use case model which clarifies the control flow of use cases and facilitates its automatic transformation into other modelling artifacts.
The general idea of requirements-level programming is presented in Figure 1. The use case representations define flows of user-system interactions which specify the application logic of the system. This application logic uses terms and phrases specific for a given problem domain. These terms (domain elements) and phrases (domain-specific actions) constitute the domain logic with appropriately defined processing algorithms. What is important, the domain logic (cf. domain-specific languages) is declared in a language which does not depend on any specific domain. This way we result in a general-purpose language where the application logic is precisely separated from the domain logic. It can be noted that the specifications at the application logic level can be easily applied to any other problem domain by substituting terms and phrases. This is illustrated in an example in Fig. 2. There we can see two identical fragments of application logic applied to two different problem domains with different processing algorithms.

The above presented criteria for a programming language at requirements level are met by the Requirements Specification Language (RSL, see [20] for full specification) developed within the ReDSeeDS project (www.redseeds.eu). The language allows for specifying use case representations, defining flows of events and associating terms used in these events with a central domain vocabulary.
We will illustrate the language through a case study example. In Figure 3 we can see a fragment of the use case model of a Campus Management System (CMS). This model follows standard use case modelling principles with the exception of two ≪\text{invoke}≫ relationships. This type of relationship substitutes the ambiguous ≪\text{extend}≫ and ≪\text{include}≫ relationships (see [21]). Figures 4 and 5 show the details of use case representations. The presented scenarios are written in a simple subject-verb-object (SVO) grammar. They also show how the ≪\text{invoke}≫ relationship is weaved into the flow of events. In Figure 5 we can see two scenarios with their flow being controlled by two conditions (≫\text{cond}). It can be also noted that every SVO sentence is adorned with an indicator of the action “target” (either the system or an actor). This will help in generating code as described in the further sections.

The SVO sentences are linked to a central vocabulary which is presented in Figure 6 (right). Every object in a sentence has an associated domain element (notion) in the vocabulary. Obviously, objects in different sentences can point to the same domain element. This way, the whole specification is coherent with scenarios written in a uniform terminology. At the same time, the vocabulary offers space for specifying the domain logic (placed under domain phrases like “verify course”). It can be noted, that an additional activity from the requirements specifiers it to divide both the use case model and the domain vocabulary into packages. These packages will be used to structure the final code generated from requirements, as presented in the next section.
In order to be able to generate code from the above presented requirements models we need to assign runtime semantics to them. This would mean that each of the RSL constructs would have their meaning explained in the code execution environment. We will supply this meaning by specifying automatic rules for transforming these constructs into Java-like code. This way, the RSL runtime semantics will be transformed onto Java runtime semantics. The following list presents the most important rules (some rules were omitted for brevity), concentrating on the elements defining the application logic.
1. **Package transformation.** Every package in the requirements model is transformed into a package in Java. The packages are divided into the application logic layer (use case packages) and the domain logic layer (notion packages). In addition, a single package for user-interface-related operations is generated.

2. **Use case transformation.** Every use case is transformed into a public interface (with appropriately concatenated name) in the associated package. For every such interface, a realising class is generated.

3. **Notion transformation.** Every notion is transformed into a public interface (with appropriately concatenated name) in the associated package. For every such interface, a realising class is generated.

4. **Phrase transformation.** Every verb phrase in the domain model is transformed into an operation in the interface generated either from the containing notion or from the use cases that use this phrase. The name of the operation is a concatenation of words of the phrase. The detailed rules are as follows.
   (a) Verb phrase used in an SVO sentence where an actor is the subject and the system is the target is generated into an operation of the interface generated from the use case containing the SVO sentence.
   (b) Verb phrase used in an SVO sentence where the system is the subject and an actor is the target is generated into an operation of the interface within the user-interface-related namespace.
   (c) Verb phrase used in an SVO sentence where the system is the subject and the system is the target is generated into an operation of the interface generated from the containing notion.
5. **Scenario transformation.** Every set of scenarios for a use case is transformed into code within the methods of the class realising the interface generated from this scenario. The detailed rules are as follows.
   (a) Code of the class generated from a use case contains methods associated with operations generated according to rule 4a.
   (b) Every method in the above class contains code generated from sentences that follow an SVO sentence from rule 4a and finish at the next such sentence.
   (c) Inside the above method, there are generated method calls. For an SVO sentence from rule 4b, a call to the method realising a user-interface-related operation is generated. For an SVO sentence from rule 4c, a call to the method realising a domain logic operation is generated.
   (d) Alternatives in scenarios are transformed into conditional instructions (“if”). For each of the alternative scenarios, a branch in the conditional instruction is generated. The branching is done by detecting the value returned by the method call generated from the last SVO sentence before the alternative in the scenario.

The above rules are illustrated in the case study example. Figures 7 to 10 show the structure and dynamics of the code generated from the requirements model presented in the previous section. Figure 7 presents the application of rules 1 to 3. It shows a UML component diagram with generated components (representing Java packages) from the requirements model structure shown in
Figure 6. Each of the components of the application logic layer has interfaces generated from use cases (e.g. the “Edit course” use case is transformed into the IEditCourse interface). Similar rule was applied to the domain logic layer (e.g. the “course list” notion transformed into the ICourseList interface). More detailed example for rules 2, 3 and 4 is presented in Figure 8. There we can see the interfaces of the CourseManagement package from the application logic layer presented in a UML class diagram. These interfaces have their operations generated according to rule 4a. Moreover, we can see two classes that realise two selected interfaces (other omitted for brevity). These two classes are associated with the interfaces from the domain logic layer (ICourse and ICourseList) and the single interface of the UI layer (see rules 4b and 4c). Figures 9 and 10 show the code of the above two classes with full dynamics (contents of the methods). The associated UML interaction (sequence) diagrams explain how this code is equivalent to scenarios presented in Figures 4 and 5. Comparison of these four figures should facilitate understanding of the set of rules 5a-5d. It can be noted that in order to generate all the code details (like e.g. the call/method signatures), some additional markings in the requirements model and additional
rules would need to be applied. These simple rules were omitted due to limited space.

4 Software evolution as requirements evolution

Having the automatic code generation mechanisms presented in the previous section we can now define a schema for software evolution. As postulated in the introduction, the schema is driven by changes in requirements. This is illustrated in Figure 11. It shows two iterations in an evolutionary software lifecycle. In the first iteration, the requirements (describing the application logic $R_{AL1}$ and the domain logic $R_{DL1}$) are specified and appropriate code generated. According to what was demonstrated in the previous section, the application logic code $C_{AL1}$ can be generated fully, but the domain logic code $C_{DL1}$ can be generated partially (only the class structure, methods and signatures). In the second iteration, both the application logic and domain logic requirements change ($\delta_{RAL}$ and $\delta_{RDL}$). After the change, the application logic code $C_{AL2}$ can be generated automatically, and no additional effort is needed. However, there remains a crucial issue of determining changes to the domain logic code $C_{DL2}$. We would like to know which methods in the partially generated code $C_{DL2}$ need updates by programmers and to what extent. This can be done by examining changes in domain logic requirements $\delta_{RDL}$. It can be noted that in order to examine these changes (especially in large systems with vast requirements specifications) we need a tool which compares two requirements specifications and shows the $\delta$. By tracing this delta into code, the methods needing rework can be indicated very precisely ($\delta_{CDL}$).

Let us now examine the schema we have sketched above, within our case study example. The changed requirements are presented in Figures 12 and 13. The first diagram shows the updated use case model, with added use cases highlighted. The second diagram presents changes made to the representation of one of the use
Based on these changed requirements, code has been re-built. Its new version now has an updated structure shown in Figure 14. It can be noted that the structure has been extended appropriately to the extension in the use case set and the necessary domain elements (notions). New interfaces have been added, and the existing interfaces have been modified (the interface details not shown for brevity). By comparing Figures 7 and 14 it can be easily determined which additional interfaces have been created for the new use cases in the application logic layer and for the new notions in the domain logic layer.

Figure 15 show updates made to the code of the CShowOwnedCourseList class. Changes can be easily determined by comparing this with Figure 10. It can be noted that an additional method has been created. Now, there are two
methods (instead of one in the previous version) which correspond to two major interactions of the user with the system (see transformation rule 5b in the previous section). It can be noted that the difference in the application logic code also influences the other layers. There have been added a call to the ICursoseFilter interface in the domain logic layer and several additional calls to the user interface layer.

The application logic code (e.g. methods “entersCourseFilter” and “selectsToSeeCourseList”) presented in Figure 15 can be generated using the transformation rules introduced in this paper. However, the domain logic code (e.g. method “verifies” in CCourseFilter) and the user interface code needs to be updated manually. Thus, we need a mechanism to determine code which needs rework. This can be done “manually” by examining diagrams in Figures 10 and 15. The difference is quite obvious and the additional methods can be easily indicated. Also, these methods that can be retained are easily detected.

Unfortunately, for a larger system, this manual detection of differences is very time consuming. Thus, we use the ReDSeeDS tool to do this task automatically. The tool was used to introduce the requirements models and then to perform transformations in the first iteration and then in the second one. The results of the second iteration were then compared with the results of the first one. This is shown in Figure 16. The upper part shows comparison of one of the packages in the requirements domain model. It can be seen that two notions (“course” and “course list”) are retained in an unchanged form. Other notions (including “course filter”) are not very similar to any of the notions used in the previous iteration. Thus, these are good candidates for tracing code which needs to be modified (or here: extended). We can select an added notion and all the influenced code elements will be highlighted. This is illustrated in the lower part
5 Conclusion and future work

The most important aim of this work was to demonstrate that general-purpose requirements models can have runtime semantics and this semantics can be used to achieve high levels of automation in code evolution. As the presented case study shows, our approach automates to significant extent the transition from

Fig. 15. Updated application logic code

of the Figure. As we can see, a DTO class and an interface method are affected by the additional notion “course filter”. 
requirements written in constrained natural language to code. We have demonstrated that the presented semantics allows for generating Java code within the application logic layer.

It is important to note that the presented RSL language fulfills the requirements postulated in [22]. Its understandability to the users was validated in the industrial context and the results presented in [23]. In this paper we have shown that RSL is also precise enough to control code generation. Thus we can claim that the new approach, together with the ReDSseeDS tool, constitute a powerful mean of support for the software developers. The main benefit is that the developers can quickly see the consequences of changes to application logic on the domain logic. This ‘diff’ can be determined for even a yet not existing code right after the requirements are ready. The differences are illustrated by instantly showing places in the domain-specific code where code rework is needed. Another important benefit is that the system refactors the code automatically based on the changed scenarios and domain notions. The developers have clear pointers, as to which methods should be changed (extended, combined, split, etc.). This is clearly shown by comparing visually with the previous version.
A software development organisation can base its software lifecycle around a repository of RSL-based requirements models. Within this repository, different variants of the same system (produced in different iterations) can be stored. The repository can also contain models for other systems, produced in different domains. It can be noted that after conducting several projects, a development organisation obtains a powerful repository of reusable artifacts. These artifacts can be reused by building new versions, but also a wider (cross-domain, cross-system) reuse is possible. This can be done by merging use cases and associated code.

Requirements written in RSL offer specific solutions to specific problems. However, it can be noted that RSL models can be also treated as certain application logic patterns. If we detach the application logic from the specific domain we obtain a generalised model which can be treated as a pattern for visible functionality of software (see. also Figure 2). This seems to be an interesting research direction and is planned to be investigated in the future. Another interesting direction is to introduce into RSL certain constructs that would allow for specifying domain-specific elements in more detail. By combining with the pattern approach, this direction would allow for constructing applications based on pre-defined domain specifications and associated patterns of application-logic functionality. The new domain-specific element of RSL could include an algorithmic language for specifying domain logic processing (as opposed to application logic processing offered already by RSL). Also, certain notation could be introduced for representing domain-specific data objects (structure of the domain). This could be combined with possible integration of RSL with the existing domain-specific languages. Domain-specific languages could be plugged-in at the domain logic level and substitute the above mentioned domain-specific elements of RSL.

References