Science of Computer Programming 166 (2018) 89–119

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Towards naturalistic programming: Mapping language-independent requirements to constrained language specifications

Mariem Mefteh [a,*], Nadia Bouassida [a], Hanêne Ben-Abdallah [b]

[a] Sfax University, Mir@cl Laboratory, Sfax, Tunisia
[b] King Abdulaziz University, Jeddah, KSA and Sfax University, Mir@cl Laboratory, Sfax, Tunisia

ARTICLE INFO

Article history:
Received 22 November 2016
Received in revised form 7 May 2018
Accepted 21 May 2018
Available online 29 May 2018

Keywords:
Natural language processing
Patterns
Semantics
Multilingual requirements
Use case scenarios

ABSTRACT

This research paper presents a new approach that constitutes a first step towards programming using language-independent requirements. To leverage the needed programming effort, our approach takes requirements in the form of language-independent use case scenarios. Then, it generates the inputs of a code generator which, in turn, produces the corresponding code. To provide for the language-independence, our approach uses an enriched version of the semantic model, as a means to represent similar ideas possibly in different ways and in different natural languages. The enrichment consists of a set of patterns that it implements as XML code representing the information embedded in the use case scenarios. This intermediate representation can be processed to derive the inputs required by any code generator to produce code in a particular programming language. This paper illustrates the approach and its tool support for use case scenarios written in English and French, and semantic model patterns implemented as XML code that can be processed by the ReDSeeDS code generator. In addition, it presents the results of an experimental evaluation of the approach on use case scenarios (written in English and in French) belonging to five different systems. This evaluation quantitatively shows the ability of our approach: to extract ReDSeeDS inputs conforming to the expert's inputs with a high precision; to generate XML code elements conforming to the input with an encouraging performance as evaluated by the participating students (an F-measure ranging between 87.43% and 92.31%); and to generate Java code judged efficient by the participating programmers (an F-measure ranging between 66.4% and 93.43%).

1. Introduction

As argued by Knöll et al. [1], current programming techniques suffer from four main problems, namely the mental problem, the programming language problem, the natural language problem, and the technical problem. The *mental problem* reflects the obligation of adjusting the program ideas to the conditions of a specific programming language (like restructuring them in the form of classes, methods and attributes in the object-oriented languages). The *programming language problem* reflects the mandatory implementation of the same program ideas and algorithms in various ways depending on each programming language conditions. The *natural language problem* stems from the fact that people, working together from

* Corresponding author.
E-mail addresses: Mariem.mefteh.ch@gmail.com (M. Mefteh), Nadia.bouassida@isimsf.rnu.tn (N. Bouassida), HBenAbdallah@kau.edu.sa (H. Ben-Abdallah).

https://doi.org/10.1016/j.scico.2018.05.006
0167-6423/

different countries, are obliged to document the developed software often in English, which makes the documentation task less productive and error prone. Finally, the *technical problem* occurs when developers spend most of their time thinking about and appropriately describing the main ideas embedded in the program [1]. The above problems result in time loss and lower productivity for software development companies.

Furthermore, due to the rapid progress of programming languages and their increasingly complex development environments, software developers find themselves in an endless race towards acquiring the necessary knowledge to be productive. While the underlying theory of programming languages is important to grasp in order to express the idea of a program, learning their syntax and structures should not be, however, a predicament to productivity. Indeed, instead of facing the actual challenging tasks of programming (describing, modeling and enhancing the actual idea of a program), programmers often end-up dealing with minor issues like choosing the right character set and doing number conversions.

To minimize the time of learning and wrestling with the programming languages' syntax/structures, on the one hand, and to overcome the above four problems of current programming techniques, on the other hand, "*naturalistic programming*" [1] [2] [3] advocates expressing *program ideas* in a natural language and to transform them into programming language structures. By writing computer programs using a natural language, naturalistic programming provides for the generation of programs in various programming languages, as-needed and with low costs. After all, programmers transform their thoughts into particular programming languages, while the ideas are almost the same even when expressed in different languages.

As a first step towards naturalistic programming, we propose, in this paper, an approach and its Code Recovery tool (CRec-tool) for extracting source code from language-independent requirements. More specifically, our approach extracts a code conforming to an existing code generator's syntax from requirements described as use case[1] scenarios, one of the most commonly used forms of requirements specification. Use case scenarios describe the future system functionality, from the user's perspective, with textual details that can be used to deduce/derive the internal system's (code). Our approach has the merit of accepting language-independent use case scenarios: It does not require sentences in a special form (e.g., SVO sentences within the scenarios steps [8]), and it accepts quite complex and ambiguous sentence structures. To handle these challenges, our approach uses the *semantic model* (originally [9] [10]) which we enhanced by structuring it through *patterns*; these describe the semantic model entities and implement them in terms of XML code. We decided to rely on the XML language due to its standardized structure that can be manipulated by several tools. The proposed patterns implementation can be exploited for different purposes, such as automatic text translation, domain analysis for software product lines [11] and source code generation – the focus of this paper. More specifically, the proposed patterns implementation are exploited to extract information from the use case scenarios in order to provide developers with useful code.

It is worth noting that several approaches were proposed to generate code from instructions written in a natural language (e.g., [12], [13], [14]). Some of these approaches follow model driven paths (e.g., [15] [16] [17] [18]). However, their majority is semi-automated and accepts inputs written in a syntax-controlled natural language, often only in the English language. In contrast, our approach removes these constraints thanks to the pattern-structured semantic model that it uses to generate a formal representation of the input language-independent scenarios. In addition, our approach benefits from the important advancement achieved in some projects like ReDSeeDS [8] for the code generation purpose: Our approach implements the semantic model patterns as inputs to any available code generator (ReDSeeDS in this paper). This makes our approach generic; that is, it can be used for any natural language (English and French are illustrated in this paper), and for any code generator (ReDSeeDS is illustrated in this paper). This merit is reached by implementing the semantic model patterns in XML code that is treated to extract the corresponding ReDSeeDS inputs which consist in RSL requirements and domain models. Furthermore, our approach generates Java code following the Model/View/Presenter (MVP) architectural pattern [19] which is a derivation of the Model/View/Controller (MVC) pattern. MVP keeps the same principles as MVC, except for the elimination of the interaction between the View and the Model layers, because it will be done through the Presenter layer. The MVP pattern is used mostly to build user interfaces (UI), and thus interactive systems. An interactive system is composed of GUIs (graphical user interfaces) and functional layers. The GUIs are rendered in the generated View layer and they enable the user to manipulate data. The functional layers contain the system treatment, performed in the Model layer in communication with the Presenter layer which organizes the data to be displayed in the View.

To show the advantages and limitations of our approach, this paper presents an experimental evaluation conducted by employing the CRec-tool on use case scenarios (written in English and in French) belonging to five different systems. This evaluation quantitatively examines the three stages of the approach. First, it shows the ability of our approach to extract the ReDSeeDS inputs conforming to the expert's inputs with a high precision (between 78.38% and 93%). Second, based on students' feedback, it shows that our approach generates XML code elements conforming to the input requirements with an F-measure ranging between 87.43% and 92.31%. Third, based on programmers' feedback, it shows that the efficiency of the Java code produced by our tool on four software applications is encouraging with an F-measure ranging between 66.4% and 93.43%.

The remainder of this paper is organized as follows: in Section 2, we overview approaches for information extraction from texts and source code generation from requirements. In Section 3, we present our approach and its tool for synthesizing

[1] In this paper, we adopt the original definition of "use case" as it was introduced by Jacobson [4]: "*A use case is a specific way of using the system by using some part of the functionality. It constitutes a complete course of interaction that takes place between an actor and the system.*" This notion has been explained by other researchers, such as Cockburn [5], Constantine et al. [6], Bittner [7], etc.