Computer Languages, Systems & Structures 54 (2018) 337–357

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Automatic production of end user documentation for DSLs

Gwendal Le Moulec*, Arnaud Blouin, Valérie Gouranton, Bruno Arnaldi

Univ Rennes, INSA Rennes, Inria, CNRS, IRISA, France

A R T I C L E   I N F O

Article history:
Received 28 November 2017
Revised 19 June 2018
Accepted 18 July 2018
Available online 26 July 2018

Keywords:
Software documentation
Domain-specific language
Model slicing

A B S T R A C T

Domain-specific languages (DSL) are developed for answering specific problems by leveraging the expertise of domain stakeholders. The development of DSLs requires a significant software engineering effort: editors, code generators, etc, must be developed to make a DSL usable. Documenting a DSL is also a major and time-consuming task required to promote it and address its learning curve. Recent research work in software language engineering focus on easing the development of DSLs. This work focuses on easing the production of documentation of textual DSLs. The API documentation domain identified challenges we adapted to DSL documentation. Based on these challenges we propose a model-driven approach that relies on DSL artifacts to extract information required to build documentation. Our implementation, called *Docywood*, targets two platforms: *Markdown* documentation for static web sites and Xtext code fragments for live documentation while modeling. We used *Docywood* on two DSLs, namely *ThingML* and *Target Platform Definition*. Feedback from end users and language designers exhibits qualitative benefits of the proposal with regard to the DSL documentation challenges. End user experiments conducted on *ThingML* and *Target Platform Definition* show benefits on the correctness of the created models when using *Docywood* on *ThingML*.

1. Introduction

According to Mernik et al., domain-specific language (DSLs) are software languages that "*provide notations and constructs tailored toward a particular application domain*" [1]. DSLs are increasingly being developed to leverage specific domain expertise of various stakeholders involved in the development of software systems [2]. Although DSLs are usually small, their development requires a significant software engineering effort [3,4]. Concrete syntaxes, editors and compilers are examples of core components that compose a DSL ecosystem. Current research efforts focus on easing the development of specific parts of DSLs to reduce their development cost and maintenance [5–8].

The work proposed in this paper aims at opening new research questions and proposing new tools for DSL maintenance with a focus on the user documentation of DSLs. As advocated by Fowler, a "*kind of generator* [that comes with DSLs]*would define human readable documentation – the language workbench equivalent of javadoc.* [...]*There will still be a need to generate web or paper documentation*" [9]. Documenting a DSL is indeed another major and time-consuming development task [1,3]. This task, however, is required to promote DSLs, address their learning curve [3], and limit the "*language cacophony problem*": languages are hard to learn and the use of many languages will be much more complicated than using a single language [9].

* Corresponding author.
E-mail addresses: gwendal.lemoulec@irisa.fr (G. Le Moulec), arnaud.blouin@irisa.fr (A. Blouin), valerie.gouranton@irisa.fr (V. Gouranton), bruno.arnaldi@irisa.fr (B. Arnaldi).

https://doi.org/10.1016/j.cl.2018.07.006
1477-8424/© 2018 Elsevier Ltd. All rights reserved.

By studying parallels made between DSLs and APIs (*Application Programming Interface*) [1,10,11], we identified four technical properties that end-user DSL documentation tools should possess:

(1) Documentation must be complete, i.e., all the DSL concepts must have an up-to-date documentation; (2) Documentation has to be contextualized according to the current need of the DSL users; (3) Maintaining documentation over several platforms is a complex task that can lead to documentation obsoleteness; (4) Providing code examples to illustrate each concept of a DSL is a time-consuming task.

Documentation is known as improving the usability of the documented artifact [12]. So, these four properties aim at answering a more general challenge that is improving the usability of DSLs by improving their end user documentation.

The proposed approach focuses on textual and grammar-based DSLs. The approach produces end user documentation from artifacts of the implementation phase of DSLs: the metamodel, the grammar, and models that cover all the concepts of a DSL. For each concept of a DSL, the metamodel, the grammar, and a model are sliced [13,14] to keep their elements that focus on this concept. A piece of documentation, dedicated to this concept, is then produced and is composed of: an illustrative example; explanations about the concept and its possible parameters, in natural language. These explanations are not fully synthesized by our approach: metamodel documentation is extracted from the metamodel to be used in the generated documentation. One benefit of the approach is its ability to capitalize on existing DSL artifacts to produce documentation for different platforms. For example, the documentation are currently generated for two platforms: (1) in a *Markdown* format to be easily integrated in wikis; (2) as *Java* code to be seamlessly integrated in the Xtext [5] DSL editor and then to provide DSL users with live documentation by content assist.

Our generative process follows coverage criteria: the produced end user documentation explains all the concepts of the DSL domain model (e.g., all the classes, attributes, and references of the DSL metamodel).

The proposal has been prototyped in *Docywood*,[1] built on top of the Eclipse Modeling Framework (EMF) [15] and Xtext. We validated the proposal through an experiment that involves: 17 subjects; two third-part modeling DSLs, namely *ThingML*[2] [16] and *Target Platform Definition*,[3] designed for a computer science audience; two language designers of *ThingML*. The results of the experiment exhibit qualitative benefits of the proposal with regard to the five DSL documentation challenges. Both subjects and language designers identified several possible improvements. The quantitative results exhibit benefits regarding the correctness of the created models when using the generated documentation in addition to the official one for ThingML.

The paper is structured as follows. Section 2 introduces an example used throughout the paper to illustrate the approach. Section 3 explains the approach. Section 4 details the evaluation of the approach. Section 5 discusses related work. Section 6 concludes the paper and gives insights for future work.

2. Problem statement

First, we introduce an illustrative example use throughout this paper to illustrate the approach. Then, we formalize the problem to solve.

2.1. Illustrative example: a DSL for moving robots

We define a simple language, called *Robot*, for moving robots. Fig. 1 describes the documented metamodel of the *Robot* DSL. A user can define a program (*ProgramUnit*) to move a robot with commands (*Command*). The commands are: move forward (*Move*); rotate on itself following a given rotation angle (*Turn*); a specific *while* loop to execute commands while no obstacle is in front of the robot (*WhileNoObstacle*). All the metamodel elements are documented (Fig. 1 shows the embedded documentation of three elements, namely *ProgramUnit* , *ProgramUnit.commands*, and *Move*). The documentation, written by language designers, is embedded in the metamodel. For example with *EcoreTools*, such a metamodel documentation consists of annotations on the metamodel elements.

Listing 1 shows a *Robot* code snippet that follows the grammar of Listing 2. A *ProgramUnit* surrounds its commands with the *begin* and *end* tokens. The *Move, Turn*, and *WhileNoObstacle* commands respectively match the tokens *move, turn*, and *whileNoObstacleAt*, followed by their parameters declared between parentheses. A *WhileNoObstacle* command defines sub-commands between brackets.

2.2. Overall objectives

Provide a support to automate the production of DSL documentation.

2.2.1. Technical properties

Software languages are software too [17] and relations between APIs (*Application Programming Interface*) and DSLs have been established [1,10]. So, the relationship can be drawn between DSL and API documentation to precise the overall

[1] See: https://github.com/arnobl/comlanDocywood
[2] See http://thingml.org/
[3] See https://github.com/mbarbero/fr.obeo.releng.targetplatform