DOMAIN SPECIFIC LANGUAGE FOR SPECIFYING FORMAL SOFTWARE REQUIREMENTS

Maria COLȚA, Cristian CREȚU*, Alexandrina GORCEA, Irina NEDEALCOVA, Daniel SCHIPSCHI,

Department of Software Engineering and Automation, group FAF-223, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Crețu Cristian, cristian.cretu@isa.utm.md

Tutor/coordinator: **Mariana CATRUC**, university lector

Abstract. *This article represents a comprehensive analysis of developing a Domain Specific Language for specifying formal software requirements. The paper describes the process of the DSL's creation, focusing on its objectives, key features, and governing principles. This analysis outlines the essential components and capabilities that the DSL incorporates, including syntax, semantics, and the grammar's vocabulary. Through the utilization of the ANTLR tool, a parse tree was constructed that precisely illustrates the DSL's structure, offering a clear view of its configuration. The DSL is crafted to facilitate the expression of both non-functional and functional software requirements, through a structured format that encompasses interfaces, specifications, and detailed requirements and functionalities specifications. This new DSL strives to simplify the specification process, bringing forth innovative methods and perspectives for the effective articulation of software requirements.*

Keywords: *DSL, grammar, semantics, syntax, specification, software requirements*

Introduction

Domain-Specific Languages (DSLs) for specifying formal software requirements represent a focused approach to overcoming the traditional challenges encountered in software development. These languages are engineered to offer a high degree of specificity, enabling developers and stakeholders to communicate requirements with unparalleled precision. The essence of DSLs lies in their ability to encapsulate complex software functionalities and requirements within a more intuitive and accessible linguistic frame- work. This specificity facilitates a deeper understanding among all parties involved in the development process, from business analysts and project managers to the software engineers themselves. By tailoring the syntax and semantics to fit the unique needs of a particular domain, DSLs bridge the gap between the conceptual models of a system and the technical specifications required for its implementation [1, 2].

Domain Analysis

In a world where software is becoming increasingly complex and integrated into the core processes of enterprises and users' lives, it is necessary to ensure a high level of precision and predictability in the solutions being developed.

According to the most recent data from Statista 2024, the software market is expected to witness consistent growth, surpassing $1 trillion in revenue by 2027. This expanding tendency emphasizes the value of software development economically as well as how society is becoming more and more dependent on technical solutions [3].

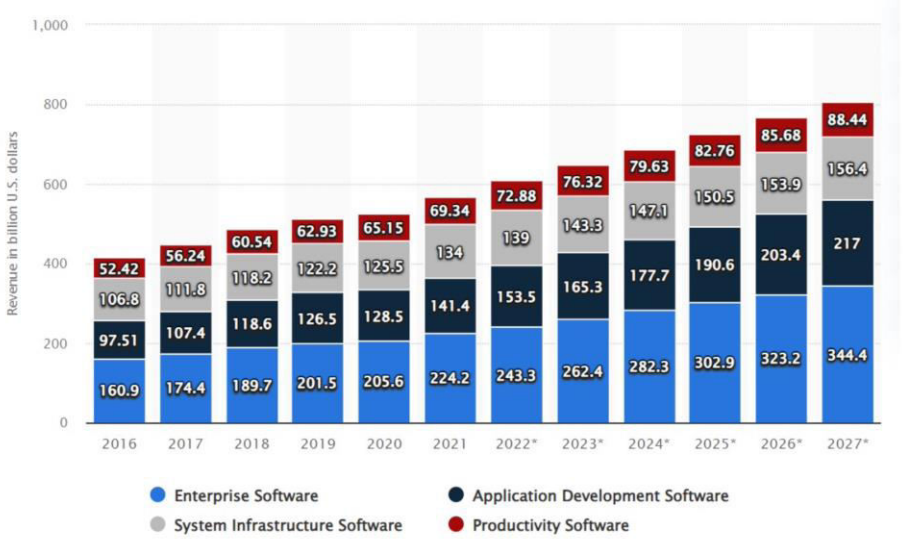

Figure 1. Revenue of the software market worldwide from 2016 to 2027 [3]

A key factor in the creation of a successful software product is the accuracy and clarity in the definition of its requirements.

Software Requirements Specification (SRS) serves as a foundational element in the software development lifecycle, offering a comprehensive description of the software's intended functionalities and operational conditions. Despite its critical role, current SRS practices often encounter significant challenges that can hinder the development process and impact the quality of the final product. These challenges primarily revolve around ambiguity, inconsistency, and a lack of formality, each contributing to potential misunderstandings and misalignments between stakeholders' expectations and the software development outcomes.

Language Overview

"Winx" is proposed as a Domain-Specific Language (DSL) designed to bridge the gap between formal, precise specifications and natural language software requirements. Its methodical development plan began with the establishment of essential technologies, including Angular, and the use of JetBrains' IntelliJ IDEA, complemented by the Antler plugin for advanced grammar development [4].

Users interact with *Winx* by typing in programs using a markdown-like syntax, enabling them to express software requirements clearly and efficiently. The computational model of *Winx* focuses on providing a structured language for requirements expression, akin to markdown, with logical operators like ”and” and ”or”. Basic data structures in *Winx* include variables that can be replaced with arrays or custom data types, enhancing flexibility in data manipulation [5].

Winx enhances the specification and implementation process through its support for control structures like packages, interfaces, and inheritance, which enable effective program flow and structure management. The requirement for complete code input for implementation, coupled with the system's ability to check and output the implementation order, adds a layer of precision and error detection. Moreover, Winx's error handling mechanism is designed to be robust, with a parser that identifies and communicates errors during the compilation phase for timely resolution [6]. The DSL extends its utility with tool support, including a VS Code extension for error highlighting, which aids in a smoother development experience. By leveraging Java as the host language, Winx aligns closely with widely used programming paradigms, offering a user-friendly, efficient, and error-resilient platform for formalizing software requirements, thereby facilitating improved communication among stakeholders and streamlining the software development lifecycle.

Grammar

An example of grammar that could be used for a DSL for specifying formal software requirements is given below:

<winx> ::= (<package> | <DESCRIPTION>)+
<body> ::= (<interface> | <specification> | <DESCRIPTION>)+

<package> ::= package <ID> { <body> }
<interface> ::= <importance> <access_modifiers> interface <ID> { <interface_body> }
<specification> ::= specification <ID> (implements <ID>)* { <specification_body> }
<interface_body> ::= (<requirementSpec> | <functionSpec>)+
<specification_body> ::= (<requirementSpec> | <functionSpec>)+

<requirementSpec> ::= <description> <importance>? <ID> { <req_specification>* <result_specification>* }
<req_specification> ::= <importance> @ <ID> (<logical_op> <ID>)* ;
<result_specification> ::= result <importance> <ID> ;
<logical_op> ::= AND | OR

<functionSpec> ::= <description> <importance> <access_modifiers> <ID> (<input_types>) (implements <ID>)* <functionBody>
<functionBody> ::= { <specificationEntry>* <return_types> }
<input_types> ::= <variable> (, <variable>)*
<return_types> ::= return <variable> (, <variable>)* ;
<specificationEntry> ::= @ <ID> : <STRING> ;

<variable> ::= <type> []? <ID>
<importance> ::= critical | optional
<type> ::= INT | FLOAT | DOUBLE | STRING | BOOLEAN | CHAR | VOID
<access_modifiers> ::= public | protected | private | default
<description> ::= <DESCRIPTION>

<LPAREN> ::= (
<RPAREN> ::=)
<COLON> ::= :
<SEMICOLON> ::= ;
<COMMA> ::= ,
<LBRACE> ::= {
<RBRACE> ::= }
<TILDE> ::= ~
<EXCLAM> ::= !

The key components for this grammar include:

1. Packages: Organizational units containing interfaces or specifications.
2. Interfaces and Specifications: Define contracts for software components, detailing requirements and functionalities.
3. Requirement and Function Specifications: Describe specific software needs and functions, incorporating logical operators (AND, OR) and importance levels (critical, optional).
4. Data Types and Variables: Support basic data types and variables for defining function inputs and outputs.

5. Access Modifiers: Control visibility of interfaces and functions (public, protected, private, default).

Code Example

An illustration of code crafted in the described Domain-Specific Language (DSL) is presented within Fig. 2. It defines a high-priority interface in a "Database" package, with a critical method GetUserList that processes an array of floats and is expected to execute within 10 seconds. The method returns a custom data type, highlighting its essential role in the database operations.

```
package Database {
    ~An interface holds an abstract structure~
    critical interface Database {
        // comment
        critical public GetUserList(FLOAT[] input)
{           @ExecTime : "10s";
            return "CustomDataType" x;
        }
    }
}
```

Figure 2. Code example in Winx

Derivation Tree

The use of ANTLR for parsing, applied to the code example provided earlier, resulted in the generation of a parse tree, which adheres to the established grammar rules (Fig. 3). It shows the hierarchical structure of the language constructs as they would be presented internally by parser after lexing and parsing the input text.

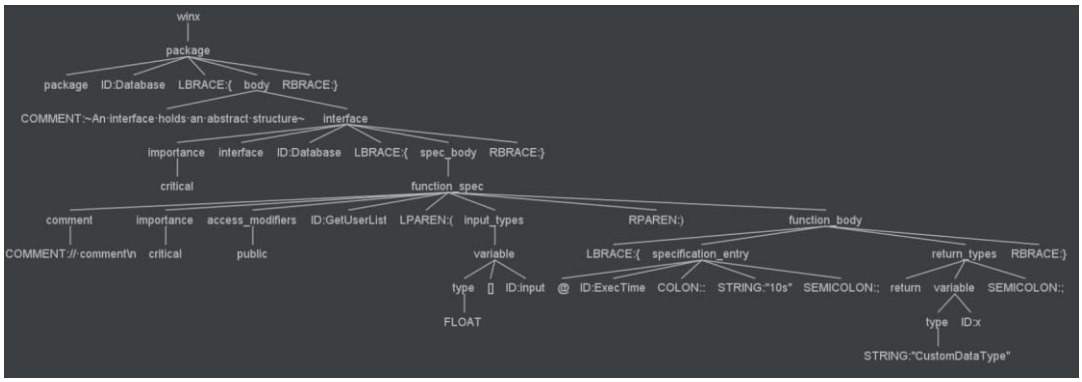

Figure 3. Derivation tree

Conclusions

The development of Winx, a Domain-Specific Language (DSL) for formal software requirements, marks a substantial advancement in converting natural language requirements into precise, actionable specifications. Leveraging tools such as Angular, IntelliJ IDEA, and the Antler plugin, Winx offers a markdown-like syntax for clear and efficient articulation of software needs.

Winx's design is notably user-friendly for developers, closely resembling Java to ensure ease of understanding and use. This similarity facilitates a smooth transition for developers familiar with Java, enhancing comprehension and execution of software requirements. Enhanced by a robust error handling mechanism and tool support, including a VS Code extension, Winx ensures a seamless development process by quickly identifying and rectifying errors.

The initiative behind Winx aims to streamline the software development process, improve software quality, and foster better collaboration among team members. By aligning closely with

Java's semantics, Winx not only improves communication among stakeholders but also significantly benefits the software development lifecycle from requirements gathering to deployment.

References

[1] "On the Construction of Programs: An Advanced Course by R. M. McKeag, A. M. Macnaghten" [Online].
[2] "On the Construction of Programs: An Advanced Course by Olajubu, Olawande" [Online]. Available: https://cs.unibg.it/esecfse_proceedings/fse15/p1060-olajubu.pdf
[3] "Revenue of the software market worldwide from 2016 to 2027, by segment" [Online]. Available: https://www.statista.com/forecasts/954176/global-software-revenue-by-segment
[4] "Model-Driven Development of Domain-Specific Languages: Challenges and Opportunities" [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-98464-9_8
[5] Available:https://books.google.md/books?hl=en&lr=&id=0_I8AAAAIAAJ&oi=fnd&pg=PA343&dq=specification+languages&ots=GlvXO3wS_O&sig=xSWK1NxtdT9Cf2HIggvrBK0-qoM&redir_esc=y#v=onepage&q=specification%20languages&f=false
[6] "An Examination of Requirements Specification Languages" [Online]. Available: https://dl.acm.org/doi/10.1145/3238147.3241538