RSLingo-Studio: A Tool for Rigorous Requirements Specification, Validation and Transformation

Miguel Pedro da Custódia Conceição

***Abstract*—Any software system project entails a thoroughly performed requirements specification during its early stages of conception. With the purpose of defining and communicating requirements between all the stakeholders, the document resulting from this requirements analysis is often produced in natural language. This is a reasonable approach, given that natural language is expressive and universal. However, it is also error and ambiguity prone. Additionally, producing graphical representations of these requirements – namely, UML diagrams, and validating them against their correctness requires a significant amount of human effort. To tackle this issue, this document introduces the development of the RSLingo-Studio tool, which serves the purpose of automating validation and streamlining the process of per-forming transformations of textual requirement specifications into different formats (MS Word or Excel) and multiple types of UML diagrams. RSLingo-Studio makes use of a rigorous Domain Specific Language (DSL) - Requirement Specification Language (RSLingo RSL) defined with the Xtext frame-work and takes full advantage of the power and flexibility of the Xtend programming language to achieve this goal. With the results obtained by developing and applying RSLingo-Studio to a specific case study, we prove that this tool can be a valuable asset for any Requirements Engineering (RE) related activity of a software project.**

***Keywords*—RSLingo; Requirement Engineering; Requirements Specification; Validation of Requirements; Transformations; Model-Driven Engineering; Eclipse; Xtext**

I. INTRODUCTION

This first section introduces the RSLingo-Studio project. We begin by describing the problem domain of the project, followed by the contributions this work offers and end by describing the outline of the remainder of the document.

A. Problem Description

For any engineering project, a clear understanding of the problem-domain is necessary before beginning to implement a solution [1]. As such, requirement specification is where any project truly begins and efforts should be made to ensure we have a thorough and coherent specification defined in a language understood by the major stakeholders. The resulting system requirements specification (SRS) is a valuable resource used throughout many different stages of the project's life-cycle, serving the purpose of facilitating communication between the stakeholders [2]. Evidently, mistakes in the production of this document will lead to undesirable problems that would require a significant amount of human effort to detect and repair, forcing the project's development to teeter back and forth [3]. A substantial amount of these errors stem from the use of natural language in the creation of the SRS, despite its vast expressiveness and ease of use [1]. This is due to the fact that its flexible nature can also result in documents produced with ambiguity, incorrectness, incompleteness or inconsistencies [6]. Additionally, to further achieve a shared understanding of the problem-domain, different representations of the requirements are created. Performing these text-to-model transformations is already taxing work by itself and it is made only worse by having to be subject to the difficulties inherent to the interpretation of natural language by humans, who have to consider the ambiguity, incoherence, incorrectness and incompleteness that natural language could potentially come with, as previously mentioned. Nonetheless, natural language is still the best option when specifying requirements [1, 5], so, an approach that would ensure the resulting SRS is an unambiguous and rigorously validated description of the problem-domain would be ideal. As a model-driven engineering approach [4], RSLingo aims to address this issue by defining a rigorous requirements specification language – RSL (Requirements Specification Language) and by offering a toolset that automatically validates RSL specifications and performs transformations of its contents into different graphical representations (i.e., UML Diagrams), such as Use Case Diagrams or Class Diagrams, for example. Supporting this toolset language are two documents: Excel and Word templates, which are fully compliant with the RSL language structure. This work covers the development of RSLingo-Studio, the tool that implements the RSLingo approach by allowing users to import these Excel files in order to validate its contents for correctness, completeness and coherence; perform the aforementioned transformations into UML diagrams; and generate a more formal, readable, Word document. A total and properly performed computer automatic processing of natural language text is a very complex task, hard to accomplish even by the current state-of-the-art technologies [8]. So, despite the fact that a certain degree of human validation is still necessary, the solution provided by the RSLingo-Studio tool significantly reduces the necessary human effort necessary during the requirements analysis phase of a project's life cycle by automating the validation of SRS documents and the process of generating UML diagrams.

B. Contributions

Considering the previously mentioned limitations of natural language when producing SRS documents and the current state-of-the-art of tools for RE purposes, the need and motivation for a tool like RSLingo-Studio becomes apparent. RSLingo-Studio is meant to be used as an Eclipse1 Rich Client Application, making use of the vast development toolset it offers – the Eclipse RCP (Rich Client Platform). Essentially, the usage of RSLingo-Studio aimed to guarantee two main goals:

1) **Validation** – RSL specifications that are edited using the RSLingo-Studio workspace are automatically checked for errors, inconsistencies, and incompleteness. This validation should consist of the displaying of errors (for structural flaws) and warnings (for errors that don't mean incorrectness, but might generate future issues), with adequate quick-fix suggestions. While RSLingo-Studio isn't designed for automatically correcting everything wrong with poorly produced SRS, this validation should significantly reduce the human effort necessary for identifying errors in such cases. A discussion of the validation checks that RSLingo-Studio supports is provided in section VI.
2) **Transformations** – RSLingo-Studio allows users to perform automatic text-to-model transformations of requirement specifications in the Excel format (respecting RSLingo's Excel template – section IV) into RSLingo's RSL specifications, which can, in turn, be used to generate UML diagrams. Our solution allows users to generate anything from UML Use Case to State Machine diagrams. These diagrams can also be imported back to RSLingo-Studio by performing the reverse operation, whenever further editing of a diagram is required. A discussion of the UML transformations that RSLingo-Studio produces is provided in section VI. Furthermore, RSL specifications can be exported back to the Excel format or into a Word document, for easier readability.

Automating these two steps represents a significantly reduced human effort necessary for completing these otherwise manual tasks. The resulting artefacts can be used as input whenever necessary during a software development process following the MDE paradigm, resulting in a considerable streamlining of the whole RE process.

II. RSL Overview

The Requirements Specification Language (RSL) used in RSLingo is implemented using the Xtext framework and the grammar defining language it provides. It was developed by Professor Alberto Silva prior to the beginning of this work and has since been subject to a number of changes. The current version13 of RSLingo's RSL is based on the multi-view architecture and inspired on the design of some specific former languages such as RSL-IL [7], XIS [10] and SilabREQ [11]. RSL provides a large set of constructs organized into several views according to two abstraction levels: business and system level. As such, Figure 2 contains a table that shows how an RSL specification is defined as a set of RSL packages. These

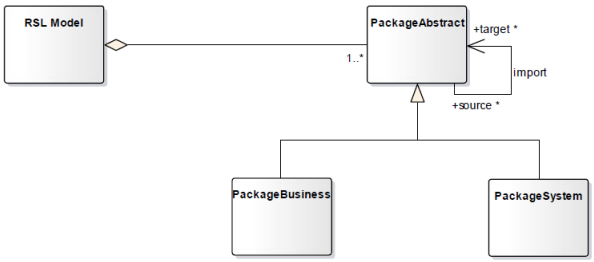

Fig. 1. RSL model as a set of business and/or system level packages

RSL packages can be either of Business or System level and are specific instances of PackageAbstract. By using the import relation, these packages can access and use the constructs defined in other packages. It is noteworthy that RSL also features PackageVariability and PackageVariabilityResolution elements, but that is out of the scope of this work. Table 1 summarizes the classification of the RSL constructs as a bi-dimensional framework.

Concerns / Levels	Package	Context	Active Structure (Subjects)	Behavior (Verbs, Actions)	Passive Structure (Objects)	Requirements
Business	package-business	Business SystemRelation BusinessElement Relation	Stakeholder	BusinessProcess (BusinessEvent, BusinessFlows)	GlossaryTerm	BusinessGoal
System	package-system	System Requirement Relation	Actor	StateMachine (State, Transition, Action)	DataEntity DataEntityView	SystemGoal QR Constraint FR UseCase UserStory

Fig. 2. RSL viewpoints

RSL's Business level includes the views that support constructs that provide a general understanding of the system's business context, while the System level contains the views that define the specification of both static and dynamic concerns of the system, namely the RE specific concerns.

III. RSLingo's Excel Template

RSLingo features a Microsoft Excel SRS-like document template that makes for an easier mapping of its contents and RSLingo's RSL while minimizing combinatorial effects [8, 9]. RSLingo's Excel template [16] is organized into several sheets, with relations and dependencies between them, which allow for having distinct perspectives and views of the system, as each sheet corresponds to a different view mapped in RSLingo's RSL [14]. As such, this template is fully compliant with RSLingo's RSL. The usage of this template is aligned with the notion that SRSs should have

templates that minimize combinatorial effects, making a set of clear distinctions and practical recommendations, such as [9]: Separating business level from system level chapters – denoted by the "b." or "s." prefixes on each sheet's name (ex: the sheet "s.systemgoals"), representing a business or system related section, respectively;

- Decoupling Actors from Stakeholders;
- Decoupling Functional Requirements from Use Cases;
- Defining a Glossary of Terms.

Additionally, the templates should be customizable to the extent that they can be adapted to the needs of the organization using it and the project it is applied to, so RSLingo-Studio features the option to choose which Word/Excel files to use as templates when performing the exporting of an RSL specification. Editing and customizing the template is possible, so long as the changes are not made on a structural level.

IV. RSLingo's Word Template

Unlike the Excel template, which serves as both an entry point (for importing) and an end product (for exporting) on the RSLingo-Studio workflow (detailed in section VI), the sole purpose of the Word template is providing the functionality of representing the information contained in the Excel template or an RSL specification in a more readable and formal fashion. Each Chapter in the template features a title and a sub-title (when applicable) for element the RSLingo-Studio exporting algorithm finds, with paragraphs that contain tags (denoted by the '@' character), that are replaced by an RSL specification's contents when exporting to Word. In other words, each paragraph with tags aims to represent the information that a row in the Excel template contained. In the case of the *DataEntity* View of an RSL specification, the template will automatically generate three tables for each *DataEntity*. These tables are used for representing an entity's Attributes, Foreign Keys and constraint Checks/Rules, being arbitrary in size. The Word template allows customization, so long as the aforementioned tags are not removed, which could potentially produce errors when using RSLingo-Studio's export to Word functionality. Therefore, layout, style or structural changes are easily performable.

V. PlantUML

PlantUML[1] is an online tool that uses human readable text descriptions - a language called PlantUML, to generate UML diagrams. It uses Graphviz[2], a graph visualization software, to draw these diagrams and export them in PNG, SVG, LaTeX and ASCII art. It supports the following UML diagram types:

- Sequence diagram
- Usecase diagram
- Class diagram
- Activity diagram
- Component diagram
- State diagram
- Object diagram
- Deployment diagram
- Timing diagram

The PlantUML language was chosen as the language for representing UML diagrams with RSLingo-Studio, due to its ease of use, expressiveness, and high level of modifiability, contributing to RSLingo-Studio's future-proofness.

VI. RSLingo-Studio

This section presents a more detailed description of this work, the technical background for its implementation, as well as constraints and different approaches that were considered regarding implementation. Figure 3 depicts what RSLingo-Studio is able to accomplish. The main

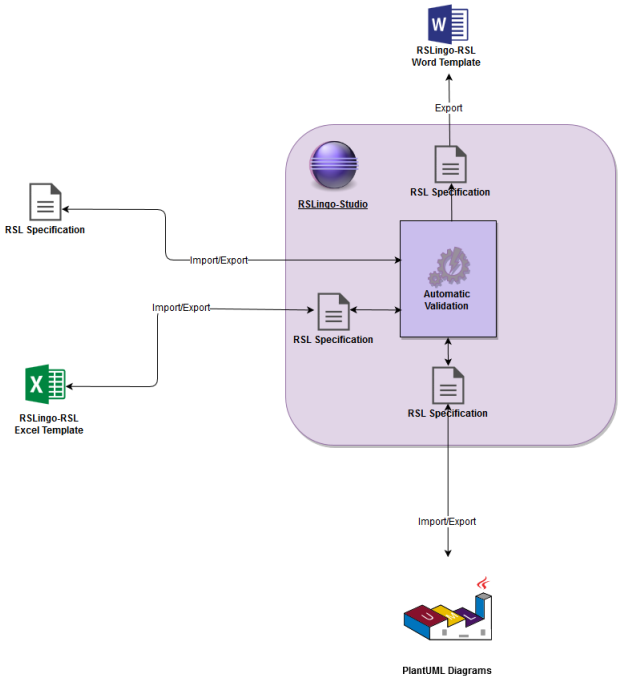

Fig. 3. RSLingo-Studio general overview

idea behind the functioning of the RSLingo-Studio tool is to allow users to begin their interaction from three different starting points, with the purpose of creating RSL specifications for validation. Users can:

1) create RSL specifications directly on the RSLingo-Studio editor;
2) generate them automatically by importing the RSL-Excel file (compliant with the pre-defined template);
3) import UML diagrams created directly in the PlantUML language. These RSL specifications are files with the ".rsl" format.

Once a RSL project is created and the desired (imported or created) RSL specification is present, the automatic validation takes place. After the user is satisfied with

[1]http://plantuml.com/
[2]http://www.graphviz.org/

the validation on the RSL specification, it is possible to export this specification into a Microsoft Word document, producing an easier to read natural language document. It is also possible to export an RSL specification back to the Microsoft Excel template. Finally, RSLingo-Studio's biggest asset is the ability to generate PlantUML diagrams from these RSL specifications.

A. Xtext

Xtext is a framework for developing programming and domain specific languages that automatically generates a parser and a class model for the abstract syntax tree (AST), while providing an Eclipse-based IDE [12]. Xtext simplifies the process of specifying a DSL by featuring a specific language for defining grammars. A grammar defines how an Ecore model is derived from a textual notation. This definition allows Xtext's code generator to automatically produce an ANTLR (Another Tool for Language Recognition) parser. A language specified with Xtext features a customizable validator for implementing additional constraint checks on a DSL, requiring only the definition of possible errors or warnings. As such, critical constraints like cyclical hierarchies or unresolved references can be easily implemented. Validation runs automatically while the user of the DSL is typing in the editor. As mentioned in section II, RSLingo's RSL has been implemented using Xtext's grammar language.

B. Xtend

Xtend is an expressive language with a high level of abstraction that, when compiled, generates Java code. It provides features such as type inference, extension methods, dispatch methods and lambda expressions. Xtend is also completely interoperable with Java, allowing one to reuse any number of Java libraries. Semantically, Xtend is highly similar to Java, which gives it a gentle learning curve as opposed to what one would expect having to learn new programming languages. The Eclipse editor for Xtend offers features like syntax highlighting, code completion, refactoring, navigation and debugging. In the context of this work, Xtend was used for implementing the classes which define the validation rules for RSLingo's RSL and the classes that perform model-to-text transformations. Additionally, all the stub classes generated by Xtext for an RSLingo specification are Xtend classes.

C. Eclipse RCP Development

RSLingo-Studio is built upon the Eclipse IDE, but is usable as a standalone rich client application. This is achieved by having making use of Eclipse's Rich Client Platform, a toolkit for Eclipse that contains the minimal set of plug-ins necessary for developing rich client applications. An example of the application's interface can be found on Figure 4. The legend for Figure 4 is as follows:

1) *Project Explorer* View – provides a hierarchical view of the artefacts in the Workbench.

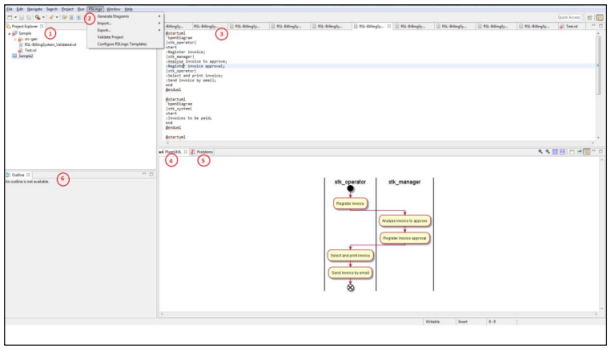

Fig. 4. RSLingo-Studio Interface

2) *RSLingo* Menu – contains all of RSLingo's commands:
   - *Generate Diagrams* is the submenu that contains the UML generation commands;
   - *Import…* branches out into the commands that allow importing from Excel or UML diagrams;
   - *Export…* contains the commands for exporting to Excel or Word formats;
   - *Validate Project* is a command that simplifies the validation process by prompting the user to select an RSL specification to be validated. If no errors/warnings are found in that specification, an "OK" message is shown. Otherwise, the Problems View is opened and highlighted and the user is shown an info-box that identifies all the errors/warnings in the chosen specification, with their respective line numbers for easier tracking;
   - *Configure RSLingo Templates* serves the purpose of allowing the user to configure the used Excel and Word templates to their liking, providing the option to link to the chosen ".xlsx" or ".docx" files to serve as templates.
3) RSL Editor – textual editor for RSL specifications.
4) PlantUML View – displays the UML Diagram corresponding to the selected snipped in the editor.
5) Problems View – displays existing errors or warnings in the current RSL project. Opened automatically when the Validate Project command is executed.
6) Outline View – provides a hierarchical view of the current RSL specification's model.

Importing and exporting between different document formats works by using Apache POI, a Java API that provides libraries for reading and writing files in Microsoft Office formats, namely Excel and Word, in the context of this work. RSLingo-Studio features handlers that execute the corresponding actions to the chosen commands in the RSLingo Menu, such as the ImportExcelHandler.java class, which serves as the default handler for the "Import from MS Excel" command. The process of importing from the MS Excel SRS template is done by running through each row of each sheet of an ".xls" or ".xlsx" file. For each

row, we extract the content of each cell while checking if it's empty and/or correctly filled. Each of these elements is saved as instances of Sheet, Row or Cell classes, provided by the API. As we progress through the sheet, its contents are stored in a Java StringBuilder instance, which is then translated into a String type and set as the content of our newly generated ".rsl" file. Figure 5 contains an example of a code snippet (RSL) generated when importing an Excel file, specifically the Constraints sheet.

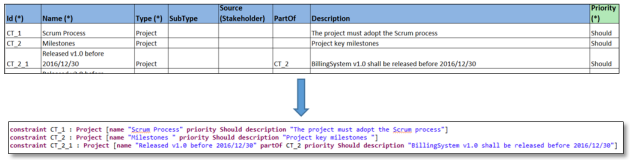

Fig. 5. Example of importing Excel file

In order to inspect a spreadsheet's contents, RSLingo-Studio possesses several different algorithms for the various sheets, ranging from complex recursive functions (such as the StateMachine or DataEntities sheets, which contain sub-rows) to more straightforward iterative ones (such as the FunctionalRequirements or Glossary sheets, in which each row corresponds to a different FR or GlossaryTerm). Exporting an RSL specification to an MS Excel file works by taking advantage of the AST that the Xtext framework automatically produces for each RSL specification, providing methods that allow the handling of elements in that specification as if they were Java classes. As such, exporting to Excel works by defining a series of template rows on the Excel template with pre-defined values. For each sheet and for each element found in the RSL specification, our handler makes a copy of the template row we defined for that given sheet and shifts it one index down. The pre-defined values that were present there are then overwritten with the information we collect from the RSL specification's AST. When a sheet is fully populated, a function is called to delete any remaining template rows. Exporting to MS Word works in the same way, where a different handler progresses through the AST for preparing the information present in the RSL specification. All this information is then placed on the resulting Word document by replacing a series of tags (denoted by the '@' character, as stated in section IV), with the contents of the AST, respecting the linguistic patterns [13, 15]. For each chapter, the tags in each paragraph are copied before being overwritten, in order to allow an arbitrary number of elements to be represented. On the other hand, the process of creating diagrams uses the Xtend language and is detailed in section VI, where we also describe the process of importing previously created PlantUML diagrams.

D. Validation

The usage of RSLingo's RSL Excel template is, in itself, an automatic form of validation provided by the RSLingo approach. Filling out a template restricts the user to only use the correct types of values, without losing any significant amount of expressiveness. As such, RSLingo-Studio is able to feature a two-step validation process, with the first consisting of this "pre-validation". When the user opens RSLingo-Studios' editor, every RSL specification present in each open project in the workspace is subject to our previously defined Xtext's Validation checks. This process works automatically as the user rewrites the specification in the editor. This implementation follows the best practice of performing as little validation as possible in the grammar and as much as possible with Xtext's validation [12], allowing the grammar to maintain its expressiveness without becoming too complex. Validation takes place in the background while the user is writing in the editor. It consists of highlighting errors on the specification, providing the user with error or warning messages and suggesting quick-fixes to remove these messages. It should be noted that the quick-fixes that are suggested do not actively correct what was initially wrong with the SRS document, but rather perform semantic and - to a certain extent, syntactic fixes to remove the error message. In other words, while automatic, RSLingo's validation needs a slight level of human input for fixing errors. Validating UML Diagrams works the same way, given that, when imported, these diagrams are translated back to RSLingo's RSL.

E. Transformation

Once the imported document is represented in a specification in RSLingo's RSL, a user can perform automatic transformations into graphical representations of information contained on the document. These diagrams can then be imported back to RSLingo-Studio for further editing. We opted to use the previously mentioned PlantUML tool as the means of specifying our UML diagrams. PlantUML is able to generate all types of UML diagrams and is constantly being updated and improved with an active community supporting it. The process of producing these diagrams takes advantage of the Xtend language for traversing the specification's model, capturing the relevant information for the selected diagram type and establishing the relations between the various elements, when applicable. This process results in a text file (".txt") that contains the respective UML Diagram specified using the PlantUML language. The nomenclature for the generated ".txt" file follows a specific pattern for consistency purposes: "<systemID>_<diagramType>_diagram.txt" (ex: S_Billing_Class_Diagram.txt). The generated text files follow a well-defined structure in which each diagram begins with a @startuml tag and ends with a @enduml tag. The second line contains a comment – for example, *'statemachinediagram*, that identifies the type of diagram represented in that particular .txt file. The following lines will contain the specification of the diagram itself, namely, the declaration of the elements (actors, stakeholders, etc.)

and relations between them, when relevant. If, for example, a system contains two state machines in its specification we will have something with a structure similar to what is shown on Figure 6.

```
@startuml
'statemachinediagram
'stateMachine1
State1 :
(…)
State2 :
(…)
@enduml

@startuml
'statemachinediagram
'stateMachine2
State1 :
(…)
State2 :
(…)
@enduml
```

Fig. 6. Example of PlantUML diagram

RSLingo-Studio progresses through an RSL specification's AST recursively using Xtend, writing the output into a Java StringBuilder instance that is, in the end of this process, set as the content for generated ".txt" file. In order to prevent overwriting the same file throughout this recursive loop, a new file is generated for each package-abstract (i.e., a different file for each System and/or Business). When the PlantUML View is open in the RSLingo-Studio editor, the displayed diagram corresponds to the one where the mouse cursor is on, anywhere between the two start/end tags. The resulting diagrams can be edited in RSLingo-Studio by changing the corresponding specification in the editor, as if it were PlantUML's online editor. Selecting a diagram allows users to store them in standalone image formats, such as PNG, for example. Importing UML Diagrams is an additional functional that RSLingo-Studio supports, albeit in a partial way, given the fact that not all diagrams can be imported successfully, as of the writing of this document. This process begins by choosing a ".txt" that corresponds to a PlantUML diagram specification. Since any text file can be imported, a validation is made to check if the chosen file is a PlantUML diagram or not, by searching for that diagram's commentary line that specifies its type (mentioned previously in this chapter). The algorithm for importing PlantUML diagrams has a different behavior for each diagram type, but the rationale is the same, regardless. The idea is to process the text contained in the ".txt" files, line by line, searching for certain keywords that represent specific bits of information, while adding this information to a StringBuilder which will be set as the content of the resulting ".rsl" file. For example, considering a simple FR diagram, as shown on Figure 7: In this case, the "'frDiagram" diagram type comment serves as an indicator as to what the algorithm should be looking

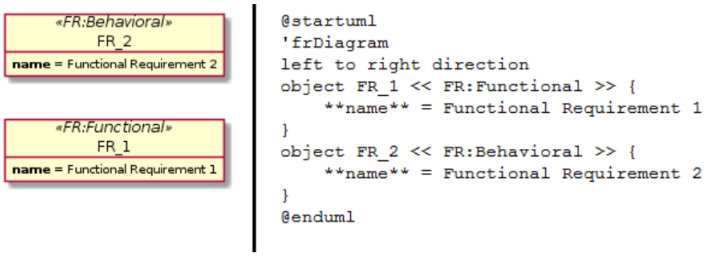

Fig. 7. FR diagram specified in the PlantUML language

for (FRs). Progressing through the next lines, when the "object" keyword is found, the algorithm "knows" it found an RSL FunctionalRequirement element, so the respective keywords are added to the StringBuilder. Afterwards, the following word will be the FR's ID, followed by its type. The following lines will contain the remaining attributes of this FR instance, until the bracket closing character ('}') is found. The "left to right direction" is merely a command for PlantUML to display the diagram with a vertical layout. When applicable, if the "->" and other similar relational keywords are found, they will be processed as relations between the various elements.

VII. Evaluation

In this section, we describe how we evaluated our work and the chosen criteria to do so, as well as the results we obtained. It also presents the BillingSystem, a fictional system used as a working example throughout this evaluation process. Given that we had two main goals: **automatic validation of requirement specifications** and **transformation of these requirements into UML Diagrams**, it felt logical to test these functionalities independently.

A. The BillingSystem Case Study

The Billing System case study simulates a business information system for managing customers, products and invoices of an organization. It contains a main system, S_Billing, and four subsystems, namely: Billing_Admin, Billing_Customers, Billing_Products, and Billing_Invoices. It also depends on two external systems: ERP_Accounting and Portal_FinanceInstitute. For the purpose of testing our solution, we added several errors/inconsistencies to the BillingSystem Excel specification.

B. Validation

Our main metrics for evaluating RSLingo-Studio's validation is the number of these warnings/errors detected and the number of errors we could correct with the assistance of our QuickfixProvider. Opening the generated ".rsl" file on the RSLingo-Studio editor, we can see that a number of errors were detected. By using the Validate Project option in the RSLingo menu, we are given the

message that 14 problems were detected. Figure 8 depicts the displayed window.

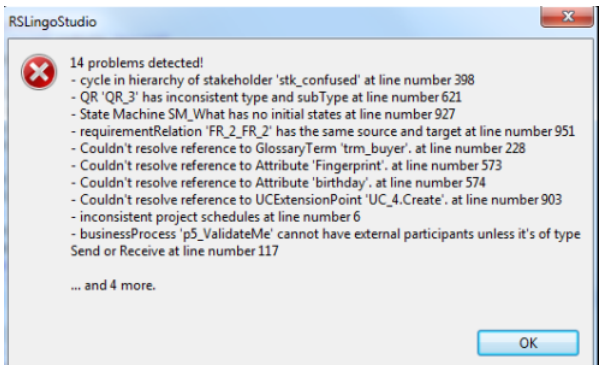

Fig. 8. RSLingo-Studio Project Validation

We defined 15 problems (cyclical hierarchies count as two problems, since they are detected on both affected elements), as described in Chapter 7.1, 14 of which were detected. However, one of the errors is omitted (state machine with no final state), due to being overlapped by the "state machine with no initial state" error, so, by clearing that warning, the missing error is shown. As such, we can state that our solution detected 15 of the 15 errors present in the BillingSystem specification.

The following step is performing the corrections that our QuickfixProvider suggests. Going over each error and performing the suggested quick fix, we could remove every problem that was detected. While some of these solutions are immediately automatic, the QuickfixProvider is able to, at least, suggest to the user what should be done to fix each problem. Consequently, we were able to fix 15 out of the 15 problems detected by the Validator with the assistance of the QuickfixProvider, which can be considered a most favourable result.

C. Transformations

Evaluating the transformations functionality can be sub-divided into three steps:

1) RSL-Excel Transformations: where we tested the *Export to Excel* and *Import from Excel* functionalities for the **completeness** of the results. In this test, we expected both the generated RSL specification from importing and the Excel file generated from exporting to contain the exact same information.
2) RSL-Word Transformations: with the focus being on the *Export to Word* functionality, whose metrics were, similarly to the previous step, the **completeness** of the end product (a MS Word document, in this case).
3) RSL-UML Transformations: where we tested the generation of UML diagrams from RSL specifications. The main metric associated with these tests was the **correctness** of the generated diagrams. We also tested the functionality of generating RSL specifications from PlantUML diagrams by assessing the **completeness** and **correctness** of the resulting RSL specifications.

The results we obtained on all three of these steps were quite satisfactory, having met our expected values regarding the metrics we defined. The only issues we found were mostly aesthetic.

VIII. Conclusions

In the early stages of a project's life-cycle, the specification of requirements has a crucial impact on how successful a project will be. To communicate information between all the stakeholders, an SRS document is produced in natural language text. Despite being the best option, there are still problems inherent to the use of natural language, namely ambiguity, incompleteness, inconsistency and incorrectness that need to be checked through extensive human endeavour. This work proposes the RSLingo approach, which aims to solve these issues by offering an automation of this validation and generation of design models from requirements representations. After analysing the current state-of-the-art, what the RSLingo-Studio toolset offers appears to be unmatched. Allowing the automatic validation of a natural-language-based textual specification of requirements while providing the means to generate multiple types of UML Diagrams could drastically reduce the necessary human effort for a project's development. The technologies employed to develop this work revolve around the Xtext framework and the Xtend language. Ideally, everything will be kept within the Eclipse IDE, resulting in a reduced dependency on different tools. The level of accomplishment of this work was evaluated by applying the RSLingo-Studio tool to a case study with issues that are quite common in a real-world context, with the results being quite favourable. As such, we find that this solution has the potential to be a true asset to any team developing a software project and to be a notable contribution to a project's success.

References

[1] Davis, A.: Just Enough Requirements Management: Where Software Development Meets Marketing, 1st ed. Dorset House Publishing, May 2005.
[2] Silva, A.R.: Quality of Requirements Specifications: A Preliminary Overview of an Automatic Validation Approach, In Proceedings of ACM SAC'2014 Conference, ACM. 2014.
[3] Pohl, K: Requirements Engineering: Fundamentals, Principles, and Techniques, 1st ed., Springer, 2010.
[4] Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structure 43 (139-155), 2015.
[5] Harry D Foster, Adam C Krolnik, and David J Lacey. Assertion-based design. Springer Science & Business Media, 2004.
[6] Donald Firesmith: Specifying Good Requirements. Journal of Object Technology, vol. 2(no. 4): pp. 77-87, 2003.
[7] D. Ferreira, Silva, A.R.: RSL-IL: An interlingua for formally documenting requirements. 2013 3rd International Workshop on Model-Driven Requirements Engineering, MoDRE 2013 - Proceedings, pages 40-49, 2013.
[8] Silva, A.R., Serrão, D. and Catarino, T. RSL-IL Excel Template: A System Requirement Specification based on the RSL-IL Language, V1.1, 2015.

[9] Silva, A.R., Ferreira, D.: Towards a System Requirements Specification Template that Minimizes Combinatorial Effects, 2014, 9th International Conference on the Quality of Information and Communications Technology.
[10] Ribeiro, A. and Silva, A.R. (2014) XIS-Mobile: A DSL for Mobile Applications. Proceedings of 29th Symposium on Applied Computing (SAC'14), Gyeongju, 24-28 March 2014, 1316-1323.
[11] Silva, A.R., Savic, D.: Use Case Specification using the SilabREQ Domain Specific Language, Computing and Informatics, Vol. 34, 2015, 1001–1034, V 2015-Oct-11
[12] Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd ed., Packt Publishing, 2013.
[13] Alberto Rodrigues da Silva, Linguistic Patterns and Styles for Requirements Specification: The RSL/Business-Level Language, Proceedings of the European Conference on Pattern Languages of Programs, 2017, ACM.
[14] Alberto Rodrigues da Silva, RSL-IL Excel Template: A System Requirement Specification based on the RSLIL Language, V1.1, 2015, ACM.
[15] Alberto Rodrigues da Silva, A Rigorous Requirement Specification Language for Data-Intensive Enterprise Information Systems: Focus on RSL's Data Entities, Use Cases and State Machines, 2017, ACM.
[16] RSLingo's Excel Template, v4.0, October 2017: https://www.researchgate.net/publication/320256323_RSLingo_RSL_Excel_Template_v40