Improving Diagram Assessment in Mooshak

Helder Correia(✉), José Paulo Leal, and José Carlos Paiva

CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Porto, Portugal
{up201108850,up201200272}@fc.up.pt, zp@dcc.fc.up.pt

Abstract. Mooshak is a web system with support for assessment in computer science. It was originally developed for programming contest management but evolved to be used also as a pedagogical tool, capitalizing on its programming assessment features. The current version of Mooshak supports other forms of assessment used in computer science, such as diagram assessment. This form of assessment is supported by a set of new features, including a diagram editor, a graph comparator, and an environment for integration of pedagogical activities. The first attempt to integrate these features to support diagram assessment revealed a number of shortcomings, such as the lack of support for multiple diagrammatic languages, ineffective feedback, and usability issues. These shortcomings were addressed by the creation of a diagrammatic language definition language, the introduction of a new component for feedback summarization and a redesign of the diagram editor. This paper describes the design and implementation of these features, as well as their validation.

Keywords: Automated assessment · Diagram assessment
Feedback generation · Language environments · E-learning

1 Introduction

Mooshak [5] is a web-based system that supports assessment in computer science. It was initially designed in 2001 to be a programming contest management system for ICPC contests. Later, it evolved to support other types of programming contests. Meanwhile, it was used to manage several contests all over the world, including ICPC regional contests and IEEExtreme contests. Eventually, it started being used as a pedagogical tool in undergraduate programming courses.

Recently, the code base of Mooshak was reimplemented in Java with Ajax GUIs in Google Web Toolkit. The new version[1] has specialized environments, including a computer science languages learning environment [7]. Although the core of Mooshak is the assessment of programming languages, other kinds of languages are also supported, such as diagrammatic languages. This is particularly important because diagram languages are studied in several computer science

[1] http://mooshak2.dcc.fc.up.pt.

© Springer Nature Switzerland AG 2018
E. Ras and A. E. Guerrero Roldán (Eds.): TEA 2017, CCIS 829, pp. 69–82, 2018.
https://doi.org/10.1007/978-3-319-97807-9_6

disciplines, such as theory of computation – Deterministic Finite Automaton (DFA), databases – Extended Entity-Relationship (EER), and software modeling – Unified Modeling Language (UML), thus it is useful for teaching those subjects. Diagram assessment in Mooshak relies on two components: an embedded diagram editor and a graph comparator. The experience gained with this diagram assessment environment in undergraduate courses revealed shortcomings in both components, that the research described in this paper attempts to solve.

Enki is a web environment that mimics an Integrated Development Environment (IDE). Thus, it integrates several tools, including editors. For programming languages, Enki uses a code editor with syntax highlight and code completion. The diagram editor Eshu [4] has a similar role for diagram assessment. Code editors are fairly independent of programming languages since programs are text files. At most, code editors use language specific rules for highlighting syntax and completing keywords. A diagram editor, such as Eshu, can also strive for language independence since a diagram is basically a graph, although each diagrammatic language has its own node and edge types with a particular visual syntax. Nevertheless, the initial version of Eshu was targeted to Entity-Relationship (ER) diagrams and, although it could be extended to other languages, it required changes to the source code, in order to define the visual syntax.

Diagrams created with Eshu on a web client are sent to a web server, converted into a graph representation and compared with a standard solution. The assessment performed by the graph comparator [12] can be described as semantic. That is, each graph is a semantic representation of a diagram and the differences between the two graphs reflect the differences in meaning of the two diagrams. However, the differences frequently result from the fact that the student attempt is not a valid diagram. A typical error is a diagram that does not generate a fully connected graph, which is not acceptable in most diagrammatic languages. Other errors are language specific and refer to nodes with invalid degrees, or edges connecting wrong node types. For instance, in an EER diagram, an attribute node has a single edge and two entity nodes cannot be directly connected. Hence, feedback will be more effective if it points out this kind of error and refers the student to a page describing that particular part of the language. To enable this kind of syntactic feedback, Kora provides a diagrammatic definition language, that can also be used to relate detected errors with available content that may be provided as feedback.

Another issue with reporting graph differences is the amount of information. On one hand, it provides too much information, that can actually solve the exercise to the student if applied systematically. On the other hand, it is sometimes too much and may confuse some students, as happens with syntactic errors reported by a program compiler invoked from the command line. In either case, from a pedagogical perceptive, detailed feedback in large quantities is less helpful than concise feedback on the most relevant issues. For instance, when assessing an EER diagram, a single feedback line reporting n missing attributes is more helpful than n scattered lines reporting each missing attributes. For

the same EER diagram, a line reporting n missing attributes (i.e. condensing n errors on node type) is more relevant than one on m missing relationships (i.e. condensing m errors on another node type), if $n > m$. Nevertheless, if the student persists on the errors, repeating the same message is not helpful. The progressive disclosure of feedback must take into account information provided to the student, to avoid unnecessary repetitions. Thus, new feedback on the same errors progressively focus on specific issues and provides more detail. Also, this incremental feedback must be parsimonious to discourage students from using it as a sort of oracle and avoid thinking for themselves.

This paper reports on recent research to improve diagram assessment in Mooshak and is organized as follows. Section 2 surveys existing systems for diagram edition and assessment. Then, Sect. 3 introduces the components of Mooshak relevant to this research. Three main objectives drove this research: to support a wide variety of diagrammatic languages, to enhance the quality of feedback reported to the student, and to improve usability. The strategy to attain these objectives follows three vectors, each described in its own section: the development of a component to mediate between the diagram editor and the graphs comparator, responsible for reporting on syntactic errors, in Sect. 4; the reimplementation of the diagram editor, to enable the support of multiple diagrammatic languages and mitigate known usability errors, in Sect. 5; and a diagrammatic language definition, capable of describing syntactic features and of configuring the two previous components, in Sect. 6. The outcome of these improvements is analyzed in Sect. 7 and summarized in Sect. 8.

2 Related Work

This research aims to improve Mooshak 2.0 by providing support to the creation and assessment of diagram exercises of any type, with visual and textual feedback. To the best of authors' knowledge, there is only a single tool [14], in the literature, that ensembles most of these features. This tool provides automatic marking of diagram exercises, and it has been embedded in a quiz engine to enable students to draw and evaluate diagram exercises. Although this tool supports the assessment and modeling of multiple types of diagrams, by using free-form diagrams, its feedback consists only of a grade, which is not adequate for pedagogical purposes. Hence, the rest of this section enumerates several works focusing on assessment, editing or critiquing of diagrams.

Diagram Assessment. Most of the existent automatic diagram assessment systems are designed for a specific diagram type. Some examples of these systems are deterministic finite automata (DFA) [2,9], UML class diagrams [1,11,15], Entity-Relationship diagrams [3], among others.

Diagram Editors. Many diagramming software exists from desktop applications, such as Microsoft Visio[2] or Dia[3], to libraries embeddable in web applications, such as mxGraph[4] or GoJS[5]. There is also a growing number of editing tools deployed on the web, such as *Cacoo*[6] and *Lucidchart*[7]. However, most of these tools do not provide validation of the type of diagram being modeled.

Critiquing Systems. From the diagram assessment viewpoint, critiquing features are an important part of diagram editing and modeling tools. A critiquing tool acts on modeling tools to provide corrections and suggestions on the models to be designed. These mechanisms are important, not only to check the syntactic construction of a modeling language, but also to support decision-making and check for consistency between various models within a domain. Much research has been devoted to critiquing tools and they are incorporated in systems such as *ArgoUML* [8], *ArchStudio5*[8] and *ABCDE-Critic* [13].

3 Background

The goal of this research is to make use of new and existing tools to provide support to the creation and assessment of diagram exercises of any type in Mooshak 2.0. Thus, new tools will be created and integrated with those already existent, creating a network of components as depicted in Fig. 1.

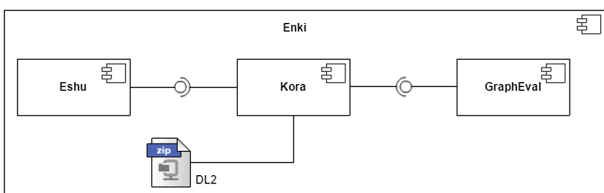

Fig. 1. UML diagram of the components of the system

The next items describe the tools already developed in previous researches that compose the system presented in Fig. 1.

[2] https://products.office.com/en/visio/flowchart-software.
[3] https://wiki.gnome.org/Apps/Dia.
[4] https://www.jgraph.com/.
[5] http://gojs.net/latest/index.html.
[6] https://cacoo.com.
[7] https://www.lucidchart.com/.
[8] https://basicarchstudiomanual.wordpress.com/.

Diagram Editor – Eshu 1.0. The corner stone of a language development environment is an editor. For programming languages, several code editors are readily available to be integrated in Web applications. However, only few editors exist for diagrammatic languages. In project Eshu [4], the authors develop an extensible diagram editor, that can be embedded in web applications that require diagram interaction, such as modeling tools or e-learning environments. Eshu is a JavaScript library with an API that supports its integration with other components, including importing/exporting diagrams in JSON. In order to validate the API of Eshu, an EER diagram editor was created in *Javascript*, using the library provided by Eshu and HTML5 canvas. The editor allows to edit EER diagrams, import/export a diagram into JSON format, apply EER language restrictions in diagram editor (constraints on links) and display visual feedback on EER diagram submissions. The editor has been integrated into Enki [7] (described later on this article) with a diagram evaluator, and validated with undergraduate students in a Databases course.

Diagram Evaluator – GraphEval. Diagrams are schematic representations of information that, ignoring the positioning of its elements, can be abstracted in graphs. Based on this, structure driven approach to assess graph-based exercises was proposed [12]. Given two graphs, a solution and an attempt of a student, this approach computes a mapping between the node sets of both graphs that maximizes the students grade, as well as a description of the differences between the two graphs. Then, it uses an algorithm with heuristics to test the most promising mappings first and prune the remaining when it is sure that a better mapping cannot be computed.

Integrated Learning Environment – Enki. [7] is a web-based IDE for learning programming languages, which blends assessment (exercises) and learning (multimedia and textual resources). It integrates with external services to provide gamification features and to sequence educational resources at different rhythms according to students' capabilities. The assessment of exercises is provided by the new version of Mooshak [5] – Mooshak 2.0, which, among other features, allows the creation of special evaluators for different types of exercises.

4 Kora Component

Kora aims to improve and make feedback extensible to new diagrammatic languages. This tool acts on the diagram editor, by providing corrections and suggestions to submitted diagrams, to help the student solving the exercise. It also makes the bridge between Eshu and Diagram Evaluator.

The Kora component is divided into two parts, `client` and `server`. The `client` part is integrated on the web interface, as shown in Fig. 2, and is responsible for running the Eshu editor, as well as handling user actions and presenting feedback. The `server` part is responsible for evaluating diagrams, generating feedback, and exchanging information with the client side, such as language configurations.

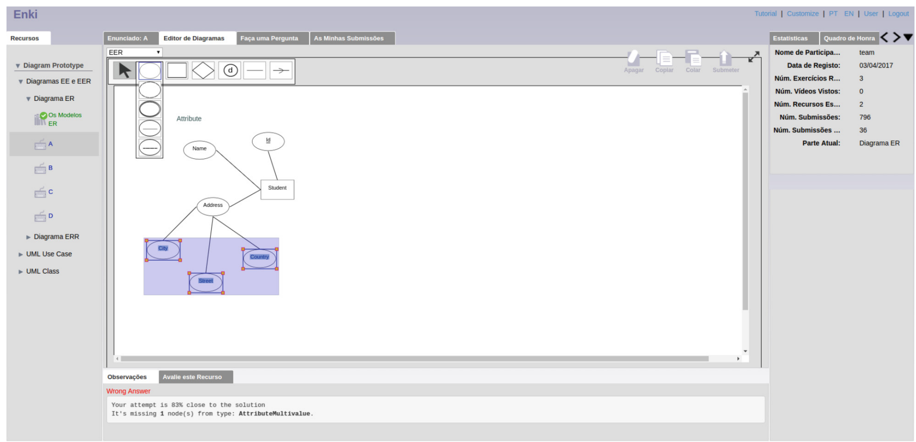

Fig. 2. User interface of Enki integrated with Kora

A diagram is a schematic representation of information. This representation has associated to itself elements that have certain characteristics and a positioning in the space. By abstracting the layout (the position of the elements), the diagrams can be represented as graphs. The approach that is intended to follow for the assessment of the diagrams is the comparison of the graphs. Thus, it is possible to analyze the contents of the diagram without giving relevance to its positioning or graphic formatting.

In Eshu 1.0, the types of connections are checked during creation and editing, that is, if source and target nodes could not be connected it would be reported immediately. However, during the validation of Eshu 1.0, it was noticed that the editor was getting slower as the number of nodes increased, although not all syntactic issues were actually covered. Also, syntactically incorrect graphs were causing problems in the generation of feedback by the evaluator. Due to these issues, syntactic verification was moved to Kora.

The diagram assessment in the system is split into two parts: syntactic assessment and semantic assessment. The syntactic assessment involves the conversion of the JSON file to a graph structure, and validation of the language syntax. It consists of validating the structural organization of the language, based on the set of rules defined in the configuration file. In this phase, the following tasks are done: validation of the types for the language; validation of the edges – for each edge it is checked if the type, source and target are valid; validation of the nodes – check if in and out degree are valid; validation of the number of connected components in the graph. The semantic assessment consists of comparing the attempt and the solution diagrams, following the graph assessment algorithm [12]. The evaluator receives a graph as an attempt to solve a problem and compares it with a graph solution to find out which mapping of the solution nodes in attempt nodes minimizes the set of differences, and therefore maximizes the classification. The feedback is generated based on these differences, and pre-

sented in Eshu, both in visual and textual form. However, when the student's attempt is far from the solution, it reports too many differences.

To cope with this problem Kora uses an incremental feedback generator to generated a corrective feedback [10]. The generator uses several strategies to summarize a list of differences in a single message. The most general message that was not yet presented to the user is then selected as feedback.

Kora uses a repertoire of strategies to summarize a list of differences. Some strategies manage to condense several differences. For instance, several differences reporting a missing node of the same type may be condensed in the message "n missing nodes of type T". Another strategy may select one of these nodes and show its label. An even more detailed strategy may show the actual missing node on the diagram. A particular strategy may not be applicable to some list of differences. In this case no message is produced.

The resulting collection of feedback messages is sorted according to generality. General messages have precedence over specific messages. However, if a message was already provided as feedback than it is not repeated. The following message is reported instead. Using this approach, messages of increasing detail are provided to the student if she or he persist on the same exact error.

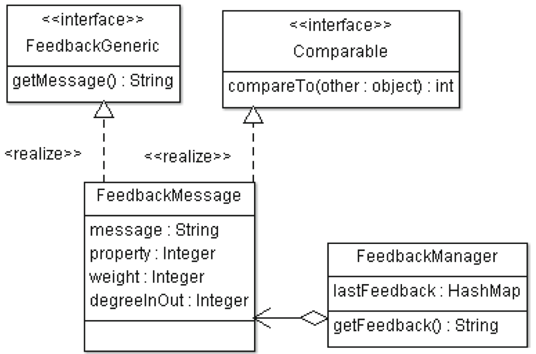

Fig. 3. UML class diagram of the feedback manager

Figure 3 presents the UML class diagram of the feedback manager implementation. The class `FeedbackMessage` contains the feedback information, including message, property number, weight, and in/out degrees (if it is a node). The property number indicates the property to which the message refers, the weight defines how much important is the mistake of the student, the degree of input/output allows to determine the importance of the node comparing to other nodes (i.e. higher degree, generally, means higher importance), and the message is the message itself. The class `FeedbackManager` generates and selects the feedback to be sent to the student. From the list of differences that is returned by the graph evaluator, it is generated a list of `FeedbackMessage`. From this list, the feedback already sent to the student is removed, and the remaining is sorted based on the

fields of the FeedbackMessage class. The first FeedbackMessage from the list is selected and sent to koraClient with FeedbackMessage (id,properties). In the KoraClient, the FeedbackMessage is converted to text and its text is presented, according to the selected language (Portuguese or English) and when possible it is presented with visual feedback in Eshu.

5 Eshu 2.0

A diagram is composed of a set of Node and a set Edge; each Node has a position and a dimension; each Edge connects a source and a target node. Although Eshu 2.0, similarly to Eshu 1.0 [4], follows an object-oriented approach for *Javascript*, it separates the data part from the visualization and editing part.

Eshu 2.0 consists of three packages: eshu, graph and commands. The package graph has the classes responsible for creating nodes and edges, storing the graph (Quadtree) and operating on the data of the graph (insert, remove, save changes and select an element). Package eshu contains the classes responsible for the user interface, including handlers for user interaction, methods to export and import the diagram in JSON format, methods to present visual feedback in the diagram editor, among many others. The package commands contains the classes that are responsible for the implementation of operations, such as undo, redo, paste, remove or resize.

One of the main improvements of Eshu 2.0 is the extensibility of nodes and edges. In Eshu 1.0, the creation of a new type of node (or edge) involves the creation of a new class that extends Vertice (or Edge for edges) and defines the method draw. With Eshu 2.0, a new type of node (or edge) can be inserted by only adding a nodeType (or edgeType) element to diagram, in the configuration file. This element contains general information for a node (or edge), such as its SVG image path (used to represent it in the UI), type name, constraints on connections, among others.

Eshu is a pure JavaScript library, hence it can be integrated in most web applications. However, some frameworks, such as Google Web Toolkit (GWT), use different languages to code the web interfaces, in this case Java. To enable the integration of Eshu in GWT applications, a binding to this framework was also developed. The binding is composed of a Java class (that is converted to JavaScript by GWT) with methods to use the API, implemented using the JavaScript Native Interface (JSNI) of GWT.

The undo and redo commands are very important to the user while editing the graph. These two operations were not included in the first version of Eshu [4], but were now added. To facilitate the integration of these operations, a set of classes that implement the command design pattern were developed. Now, operations, such as insert, delete and paste, are encapsulated as an object allowing to register them in a stack, and thus pop or push them.

Also, the API allows the host application to send feedback in the form of changes to the existing diagram. For example, if a change is an insertion of an element, then it is presented in the editor, selected, and its size is increased to

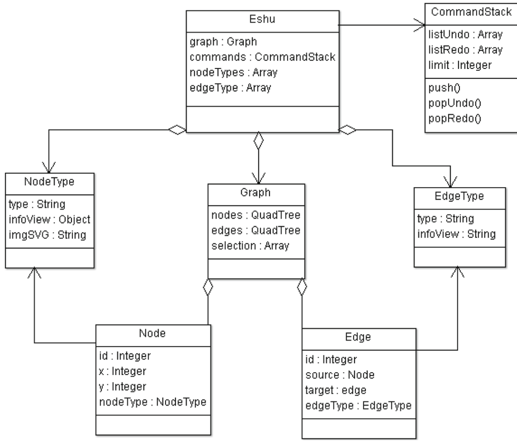

Fig. 4. UML class diagram of Eshu

highlight. If these changes are deletions, modifications or syntax errors, they can be rendered by displaying that nodes and edges with a lower transparency and selected (Fig. 4).

6 Diagrammatic Language Configuration

Both Kora and Eshu were designed to be extensible, to be able to incorporate new kinds of diagrams. A new kind of diagram is defined by an XML configuration file following the Diagrammatic Language Definition Language (DL^2). This file specifies features and feedback used in syntax validation, such as types of nodes, types of edges and language constraints. They also include editor and toolbar style configurations to be used in Eshu.

The language configuration files are set in Mooshak's administration view and must be valid according to DL^2 XML Schema definition. Figure 5 summarizes this definition in an UML class diagram, where each class corresponds to an element type. It should be noted that some element types are omitted for the sake of clarity.

The configuration file has two main types: `Style` and `Diagram`. An element of type `Style` contains four child elements, namely `editor`, `toolbar`, `vertice`, and `textbox`. The element `editor` contains the styles of the editor, such as height, width, background, and grid style properties. Element `toolbar` defines the styles of the toolbar, such as height, width, background, border style, and orientation. The `textbox` element contains attributes to configure the style of the labels for nodes and edges, such as font type and color, text alignment, among others. Finally, `vertice` contains general styles of the nodes, particularly the width, height, background, and border.

Type `Diagram` specifies the syntax of the language, including a set of `nodeType`, which describes the allowed nodes, a set of `edgeType`, that details

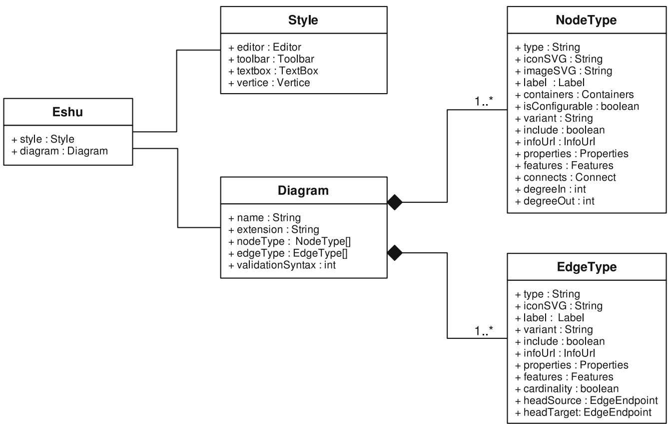

Fig. 5. UML class diagram of DL^2 XML Schema definition

the supported edges, and three attributes: name of the language (`name`), extension associated with the language (`extension`) and type of syntax validation (`validationSyntax`), which can be 0 to disable validation, 1 to validate syntax only in Kora, 2 to validate in Kora and Eshu, or 3 to validate only in Eshu.

Each `nodeType` has a path for the SVG image in the toolbar (`iconSVG`), a path for the SVG image of the node (`imageSVG`), the name of the group to which the node belongs in the toolbar (`variant`), the default properties of the label (`label`), a group of parts of the container that can have labels (`containers`), a set of properties available in the configuration window (`properties`), an `infoURL` with information about the node, a set of possible connections that the node can have (`connects`), the `degreeIn` and `degreeOut` of the node, and a boolean attribute `include` which indicates if an overlap of two nodes should be considered a connection between them.

An `edgeType` contains the configuration of an edge. The majority of its properties are similar to the existent in a `nodeType` (`type`, `iconSVG`, `label`, `variant`, `include`, `properties`, `features` and `infoUrl`). However, it has specific attributes, such as `cardinality`, that indicates whether the edge should have cardinality, and `headSource` and `headTarget` which specify how are both endpoints of the edge.

7 Validation

The goal of features presented in the previous sections is to improve diagram assessment in Mooshak. In particular, the new features are expected to enable the support of multiple diagrammatic languages, enhance feedback quality and solve usability issues. The following subsections present the validations performed to assess if these objectives were met.

7.1 Language Definition Expressiveness

An important objective of this research is to enable the support of new diagrammatic languages. For that purpose, a new XML norm for the specification of diagrammatic languages, named DL^2, was developed. To validate the expressiveness of the proposed specification language, several diagrammatic languages were configured with it.

This language defines the syntactic features of a diagrammatic language and it is instrumental in the configuration of the diagram editor, in the conversion to/of the diagram to a graph representation, and in the generation of feedback.

Mooshak already supported the concept of language configuration for programming assessment. However, the available configurations were designed for programming languages. They include, among other, compilation and execution command lines for each language. To support the configuration of diagrammatic languages, an optional configuration file was added. In the case of diagrammatic languages this field contains a DL^2 specification.

The previous version supported only EER diagrams. Hence, this language was the first candidate to test DL^2 expressiveness. It has twelve types of nodes and three types of edges, and none of them has posed any particular difficulty. In particular, all the node types were easy to draw in SVG and both the node and edge types have a small and simple set of properties. In result, the ZIP archive with the DL^2 specifications contains SVG files of nodes and edges, and an XML with configuration of elements.

UML is a visual modeling language with several diagram types that are widely used in computer science. To validate the proposed approach we selected class and use case diagrams since these are frequently used in courses covering UML. Each of these two languages has characteristics that required particular features of DL^2. Use cases diagram define relationships among nodes without using edges: the system is represented as a rectangle containing use cases. The `include` element of DL^2 allows the definition of connections between overlapping nodes and was used to create these implicit relationships. Classes in class diagram are also particularly challenging since they have complex properties, such as those representing attributes and operations. The `container` element of DL^2 definitions proved its usefulness in structuring these lists of complex attributes.

7.2 Usability and Satisfaction

The experiment conducted to evaluate the usability and satisfaction of the previous version consisted of using the system in the laboratory classes of an undergraduate Databases course, at the *Department of Computer Science of the Faculty of Sciences of the University of Porto (FCUP).* After the experiment, the students were invited to fill-in an online questionnaire based on the Nielsen's model [6], in Google Forms. The answers have revealed deficiencies in speed, reliability and flexibility. Students complained mostly of difficulties on building the diagrams, and the high delay when evaluating their diagrams.

To check impact of the changes, the validation of the usability of the current version followed a similar approach. The experiment took place on 16th and 19th of June of 2017, also with undergraduates enrolled in same course. The number of participants was 21, of which 7 were females, and the mean of their ages was 20.83 years. They attempted to solve a set of 4 ER exercises and 2 EER exercises.

The questionnaire was very similar to the used before but, this time, it was embedded in Enki, as a resource of the course. Also, the new questionnaire includes a group of questions specifically about feedback, to evaluate whether Kora helps the students in their learning path while not providing them the solution directly.

Figure 6 shows the results grouped by Nielsen's heuristics of the previous and new versions. The collected data is shown in two bar charts, with heuristics sorted in descending order of user satisfaction.

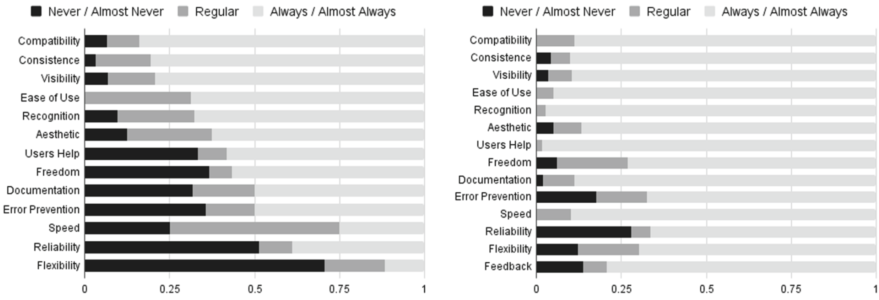

Fig. 6. Acceptability evaluation - on the left side the results of the previous version, and on the right side the results of the new version

It is clear that the usability of the system and satisfaction of the users have improved. In fact, all the heuristics got better results. Also, the results show that, with the new version, the heuristics with higher satisfaction are users' help, recognition, and ease of use. On the other side, reliability, error prevention, and flexibility were the areas with worse results. Some students complained that the feedback with Kora is too explicit, which can allow them to solve the problem by trying several times while following the feedback messages.

The last question of the questionnaire is an overall classification of the system in a 5 values Likert-type scale (very good, good, adequate, bad, very bad). The majority of the students (57.1%) classified it as very good, while the rest (42.9%) stated that it was good.

8 Conclusion

Mooshak is a system that supports automated assessment in computer science and has been used both for competitive programming and e-Learning.

Recently, it was complemented with the assessment of Entity-Relationship (ER) and Extended ER (EER) diagrams. Diagrams in these languages are created with an embed diagram editor and converted to graphs. Graphs from student diagrams are assessed by comparing them with graphs obtained from solution diagrams. The experience gained with this tool revealed a number of shortcomings that are addressed in this paper.

One of the major contributions of this research is the language DL^2. The XML documents using this configuration language decouple syntactic definitions from the source code and simplify the support of new diagrammatic languages. Configurations in the DL^2 are used both on client and server sides. On the client side, they are used by the Eshu diagram editor to configure the GUI with the visual syntax of the node and edge types of the selected languages. On the server side, they are used by the Kora component to perform syntactic analysis as a prerequisite to the semantic analysis. These configurations are also instrumental in the integration with static content describing the language syntax, that can be used as feedback when errors are detected. The expressiveness of DL^2 was validated by reimplementing ER and EER editors, as well as a couple of UML diagrams, namely class and use case.

Another contribution of this research are the approaches used by the Kora component on the server side. In complement to those related to diagram syntax and driven DL^2, mentioned in the previous paragraph, feedback message summarization also contributes to improving feedback quality. The graph comparator used for semantic analysis produces a large amount of errors that confuse the students as much they help. The proposed summarization manages to generate terse and relevant messages, starting with general messages aggregating several errors, and advancing to more focused and particular errors if the student's difficulty persists. In the latter case, feedback is generated in the diagram edition window using the diagrammatic language visual syntax.

In an upcoming version of Mooshak, this work may be used in a new assessment model that transforms the diagram of the student into program code and executes the standard evaluation model. This would allow students to "code" their solutions using diagrams, and the evaluation to be based on input/output test cases. Another assessment model could do the opposite (i.e., transform program code into a diagram) to evaluate the structure of the program, thus improving the feedback quality.

Last but not least, Mooshak with Kora is available for download at the project's homepage. A Mooshak installation configured with a few ER exercises in English are also available for online testing[9].

Acknowledgments. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme, and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project POCI-01-0145-FEDER-006961.

[9] http://mooshak2.dcc.fc.up.pt/kora.

References

1. Ali, N.H., Shukur, Z., Idris, S.: A design of an assessment system for UML class diagram. In: International Conference on Computational Science and its Applications, ICCSA 2007, pp. 539–546. IEEE (2007). https://doi.org/10.1109/ICCSA.2007.31
2. Alur, R., D'Antoni, L., Gulwani, S., Kini, D., Viswanathan, M.: Automated grading of DFA constructions. IJCAI **13**, 1976–1982 (2013). https://doi.org/10.5120/18902-0193
3. Batmaz, F., Hinde, C.J.: A diagram drawing tool for semi-automatic assessment of conceptual database diagrams. In: Proceedings of the 10th CAA International Computer Assisted Assessment Conference, pp. 71–84. Loughborough University (2006)
4. Leal, J.P., Correia, H., Paiva, J.C.: Eshu: An extensible web editor for diagrammatic languages. In: OASIcs-OpenAccess Series in Informatics, vol. 51. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/OASIcs.SLATE.2016.12
5. Leal, J.P., Silva, F.: Mooshak: a web-based multi-site programming contest system. Softw. Pract. Exp. **33**(6), 567–581 (2003). https://doi.org/10.1002/spe.522
6. Nielsen, J.: Finding usability problems through heuristic evaluation. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 373–380. ACM (1992)
7. Paiva, J.C., Leal, J.P., Queirós, R.A.: Enki: a pedagogical services aggregator for learning programming languages. In: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, pp. 332–337. ACM (2016). https://doi.org/10.1145/2899415.2899441
8. Ramirez, A., et al.: ArgoUML user manual a tutorial and reference description. Technical report, pp. 2000–2009 (2003)
9. Shukur, Z., Mohamed, N.F.: The design of ADAT: a tool for assessing automata-based assignments. J. Comput. Sci. **4**(5), 415 (2008)
10. Shute, V.J.: Focus on formative feedback. Rev. Educ. Res. **78**(1), 153–189 (2008)
11. Soler, J., Boada, I., Prados, F., Poch, J., Fabregat, R.: A web-based e-learning tool for UML class diagrams. In: 2010 IEEE Education Engineering (EDUCON), pp. 973–979. IEEE (2010). https://doi.org/10.1109/EDUCON.2010.5492473
12. Sousa, R., Leal, J.P.: A structural approach to assess graph-based exercises. In: Sierra-Rodríguez, J.-L., Leal, J.P., Simões, A. (eds.) SLATE 2015. CCIS, vol. 563, pp. 182–193. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27653-3_18
13. Souza, C.R.B., Ferreira, J.S., Gonçalves, K.M., Wainer, J.: A group critic system for object-oriented analysis and design. In: The Fifteenth IEEE International Conference on Automated Software Engineering, Proceedings of ASE 2000, pp. 313–316. IEEE (2000). https://doi.org/10.1109/ASE.2000.873686
14. Thomas, P.: Online automatic marking of diagrams. Syst. Pract. Act. Res. **26**(4), 349–359 (2013). https://doi.org/10.1007/s11213-012-9273-5
15. Vachharajani, V., Pareek, J.: A proposed architecture for automated assessment of use case diagrams. Int. J. Comput. Appl. **108**(4) (2014). https://doi.org/10.5120/18902-019