Towards the Optical Character Recognition of DSLs

Jorge Perianez-Pascual*
jpery@unex.es
Quercus SEG, Universidad de Extremadura
Cáceres, Spain

Roberto Rodriguez-Echeverria*
rre@unex.es
Quercus SEG, Universidad de Extremadura
Cáceres, Spain

Loli Burgueño†
lburguenoc@uoc.edu
Open University of Catalonia
Barcelona, Spain

Jordi Cabot‡
jordi.cabot@icrea.cat
ICREA – UOC
Barcelona, Spain

Abstract

OCR engines aim to identify and extract text strings from documents or images. While current efforts focus mostly in mainstream languages, there is little support for programming or domain-specific languages (DSLs). In this paper, we present our vision about the current state of OCR recognition for DSLs and its challenges. We discuss some strategies to improve the OCR quality applied to DSL textual expressions by leveraging DSL specifications and domain data. To better support our ideas we present the preliminary results of an empirical study and outline a research roadmap.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Applied computing → Optical character recognition**.

Keywords: optical character recognition, domain-specific languages, text recognition

ACM Reference Format:
Jorge Perianez-Pascual, Roberto Rodriguez-Echeverria, Loli Burgueño, and Jordi Cabot. 2020. Towards the Optical Character Recognition of DSLs. In *Proceedings of Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Optical Character Recognition (OCR) aims to identify and extract text strings from images (scanned documents, pictures of handwritten text, video frames, etc.). OCR is still a very active research area [18] that benefits from advances in computer vision, neural networks, machine translation, etc.

Current efforts focus mostly on mainstream natural languages for which there is ample available data to be used for training, predefined language models and public dictionaries that help achieve high levels of accuracy in the OCR process. Instead, specific OCR support decreases considerably when targeting programming languages—although some works to extract source code from programming video tutorials have appeared lately [14, 24, 26]—and even more when addressing Domain-Specific Languages (DSLs) where, due to their own nature and unlike general-purpose languages (GPL), we do not count on predefined dictionaries or pretrained recognition algorithms. For instance, unlike GPLs, we cannot assume that code repositories like GitHub have enough good examples of any DSL we can think of to train a OCR model for the language from scratch.

Nevertheless, good OCR support for DSLs could bring significant benefits and open the door to interesting applications in the field of DSLs. For instance, one could parse old manuals of legacy DSLs (or even conference proceedings from past or related SLE conferences) to automatically extract examples, which could be later used as test data for new parsers or to train any machine learning-based algorithms. In these cases, numerous examples are needed, and common solutions such as the generation of synthetic [21] data may not be optimal. When possible, our approach would be an additional source of data. From a teaching perspective, such OCR for DSLs could help in processing student assignments for automatic assessment[1]. Additionally, DSLs are currently also documented by means of video tutorials, as in the case of general programming languages. Furthermore, there is the specific case of graphical DSLs, whose graphical notation is complemented with textual languages. In those cases, the textual DSL expressions often appear as annotations next to the referenced graphical elements. Those annotated diagrams are usually stored, published, or shared as images. OCL [6] is an example of a textual language that complements UML [23] models. While complex OCL expressions are better defined in separate files, short ones are usually depicted as notes in UML class diagram (or other UML diagrams).

In the same way that Language Workbenches [9] offer features such as the generation of parsers and autocompletion from a DSL definition, we believe that they should also offer OCR support. For instance, they could automatically generate a tailor-made OCR configuration or post-processing for any given language.

SLE '20, November 16–17, 2020, Virtual, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]There are still many courses where tests are completed with pen and paper

In this paper, we provide our vision about what are the main challenges of OCR support for textual DSLs and discuss several alternatives to improve recognition quality by leveraging the DSL specification and available domain data.

The rest of the paper is structured as follows. Section 2 presents the main challenges that motivate our work. Section 3 discusses the main quality improvement strategies and Section 4 an empirical study to assess our ideas. In Section 5, related works are briefly summarized. Section 6 presents a research roadmap and, finally, Section 7 concludes our work.

2 Challenges in OCR support for DSLs

Unlike the recognition of natural language (NL), DSL snippet recognition presents additional challenges. First, since DSL snippets need to be eventually processed by a language IDE, error-free recognition is essential. For instance, missing a "." character at the end of a sentence written in English does not prevent the reader from understanding its meaning or make a text processor fail, but missing a "." in a piece of code will make the language parser not be able to build the expression abstract-syntax tree (AST). Second, when the recognition of a sequence of characters that forms a word has a low-accuracy, pre-trained OCRs try to return the closest word in their dictionary. While pre-trained OCRs have been trained for Natural Language (NL), whose vocabulary is closed (e.g., English dictionary), the quality of their results decreases when used to recognize DSL expressions, since DSLs possess specific lexicons and grammars. Furthermore, punctuation signs are applied in a very different way in a DSL compared to NL (e.g., "." characters are used as separators between two words in addition to whitespaces hence OCR models for NL tend to insert a whitespace after each "." recognized). For this reason, which is also supported by our own experience, most of the recognized DSL snippets present some kind of syntax error, which prevents their proper load into language IDEs. Regarding the way an OCR works, assuming the original DSL snippet was correct, the types of syntax errors can be reduced to the following two: (1) not found symbol, e.g., the OCR changed, merged or split one of the symbols; and (2) punctuation or operator missing or appearing in an unexpected position, e.g., missed ")" or expected """.

Although the easiest way to address these issues might be to (re)train an OCR engine with DSL images of code, usually the amount of data needed is much larger than the data available. For instance, Tesseract, which is one of the most popular OCR engines, has been trained with 400,000-800,000 rendered lines of NL text[2]. While this amount of data could be collected from GitHub repositories for well-known GPLs such as Java or Python, we believe this is hardly (if not impossible) achievable for most DSLs.

For DSL recognition, as an alternative method, we propose to use the knowledge we have about the specificity of the DSL. We distinguish five fundamental groups: (1) DSL Punctuation (punctuation signs and operators), (2) DSL lexicon (i.e. keywords, functions, types), (3) domain names (when applicable[3] – e.g., a domain name is a table name in a database schema or a class name in a metamodel), (4) user-defined names (e.g., variables), and (5) literal values (e.g., 'John Smith'). While the former 3 groups depend on the DSL and can be captured in a dictionary (i.e., they can be derived from language grammar and additional existing artifacts), the last 2 contain information that may vary from snippet to snippet. We propose to use the available knowledge about the DSL to improve the OCR quality for the first 3 groups. Note that we use the term `domain names` to mean different information sources, which could appear in different application domains, e.g., the names of a database schema for SQL, the names of a metamodel for OCL or the names of an API definition for a programming language.

Finally, an additional challenge appears when proposing any method for OCR quality improvement: OCR independence. Any method should be seamlessly applicable to different OCR engines without introducing any overhead.

3 Towards a better OCR support for DSLs

The most basic workflow for OCR recognition of DSLs has only two steps: (1) give an image containing a piece of DSL code to an OCR system, and (2) load the result into a language IDE. As this presents the problems previously discussed, two different approaches enable the improvement of the recognition quality by leveraging the knowledge aforementioned: (1) defining different custom languages as extensions of a base language to better fit the DSL (this would only be applicable to groups 1–3 of Section 2); and (2) repairing the OCR output by means of a post-processing algorithm (which applied to groups 1–4). In the following, we further present both strategies.

3.1 Custom languages

OCR engines usually provide pre-trained models for specific languages (e.g., English). They provide mechanisms, called *custom languages*, to retrain the OCR model, which fully replaces the pre-trained model with a standard/default dictionary. The default configuration is often optimized for the English language (**D: Default**), which should not be problematic given that most DSLs' syntax are written in English and its domain names are usually in English, too. However, in order to improve the OCR output quality for DSLs, according to the aforementioned groups 1–3, we propose to define the following custom languages:

[2] https://github.com/tesseract-ocr/tesseract/issues/654

[3] Note that not all DSLs support domain names

3.1.1 DP: Default + Punctuation. The punctuation signs of the English language (default) are modified to be consistent with the DSL. All punctuation signs of the English language not present in the DSL grammar are removed from the custom language and all punctuation signs of the DSL not present in the English language are added. Additionally, if a DSL operator is composed by a sequence of punctuation signs (e.g., "!="), we propose to include them, too.

3.1.2 DPL: Default + Punctuation + Language. We extend the previous custom language by including all the lexical information of the DSL language such as keywords, functions and types (e.g., "Integer", "this", "self", "substr", etc.).

3.1.3 DPLD: Default + Punctuation + Language + Domain. This language is an extension of the previous one, which additionally includes the known domain names. Since different expressions or snippets written in the same DSL may have different domain name sets, e.g. SQL queries are written for different data schemes (each schema containing different table names, attribute names, etc.), it is necessary to generate a specific custom language DPLD for each pair (DSL, domain) and retrain the model accordingly. Note that this may cause some performance overhead (see Section 4.3 for more details).

3.2 Post-processing

Although custom languages may introduce some improvements, they also pose some limitations.

The improvements of custom languages are highly dependant on the OCR engine and its prioritisation mechanisms. For instance, if an OCR does not prioritize DSL-specific keywords over similar English words, the quality of the result will be unsatisfactory. The situation is even worse if, apart from not improving the quality, the performance decreases, as in the case of DLPD.

Once a custom language is defined, it does not imply that it can be used in different OCR systems since each OCR engine usually defines its own specific method to load custom languages.

DSL snippets with an intensive use of user-defined names (e.g. "p1" or "maxVal") will easily become error-prone since these names cannot be anticipated, and thus, are not part of the custom languages' dictionaries.

We propose to address all these limitations by means of a generic and OCR-agnostic post-processing algorithm.

As previously stated, in DSL recognition, the type of syntax errors can be reduced to two: (1) missing symbols, and (2) punctuation errors. In the literature, we may find a great number of approaches to deal with the automatic repairing of syntax errors for different kinds of parsers [17, 19], which can been applied to particular programming languages. Some of these approaches could also be successfully applied herein to solve those syntax errors introduced by the recognition process, but they would be DSL-specific solutions. Given that our goal is to count on a generic and OCR-agnostic algorithm, in this paper, we explore a preliminary solution to solve the former type of error (missing symbol) by means of a name-repairing algorithm, which is inspired by the work on correction of spelling errors in [8].

First of all, we need to define a dictionary containing all the names of the DSL lexicon and the domain (groups 2–3). Once the dictionary is defined, our proposed algorithm iterates along the sequence of tokens of the recognized expression to compare each token with the symbols in the dictionary. If the algorithm detects that one is missing, the algorithm replaces it by the closest word in the dictionary. This "closeness" is measured using a normalized edit distance.

Additionally, the algorithm can include user-defined names into its dictionary. In such case, the automatic extraction of these names from DSL expressions requires knowledge about the DSL grammar to locate where they are defined and used. Let us assume that an expression defines a variable named "p1". This variable can present different recognition errors, such as "pl" or "pi". The post-processing algorithm locates the places where variables are defined and used and adds to its dictionary the recognized name of the variable declaration. Then, when recognizing variable usage, if the recognized variables do not match with a defined variable (i.e., which are part of the dictionary), it tries to repair all of the usages with the name that is in the dictionary. Note that we are not considering herein how to deal with merged and split words—e.g., the string "b|a" may be recognized as "bla"—which may need specific repairing techniques.

Finally, the second type of syntax error, i.e. punctuation errors, usually requires syntax knowledge for a full repairing. Here, we make a difference between a deep or a light knowledge of the DSL syntax, because each option clearly entails different levels of success as well as effort. In the first case, approaches for automatic syntax error repairing are helpful but they come with the need of a DSL-specific configuration, which makes them effective but more time-consuming. In the second case, we understand light knowledge as knowing only the punctuation rules of the DSL syntax. With such information, an algorithm is able to review the faulty recognized expressions to detect violations of rules, e.g. ": :" instead of ":", and define appropriate heuristics to find and repair them. Although less effective than the previous approach, it can still fix a good number of errors in a more efficient manner.

4 Empirical Study

To get a first evaluation of the aforementioned ideas about OCR support for DSLs, we conducted an empirical study with OCL [6] as a particular case of DSL, and Tesseract[4], which is one of the most popular and accurate OCR systems and broadly used in programming languages transcription [15].

[4] https://tesseract-ocr.github.io/tessdoc/tesseract-4

F1: **Courier Prime Bold**	F2: Courier Prime Regular	F3: Dejavu Mono
F4: Dejavu Serif	F5: Roboto Medium	F6: Roboto Regular
F7: UbuntuMono	F8: Ubuntu Regular	F9: Delius Swash
F10: Handlee		

Figure 1. Collection of fonts considered.

4.1 Dataset of OCL expressions

4.1.1 Data extraction and curation. OCL is a declarative language to write constraints that apply to a metamodel. Due to the problem of data shortage that most DSLs suffer, we could not obtain an appropriate dataset of repository of images with OCL expressions. Therefore, we synthesized our own dataset as follows. We took the dataset from [22], which contains 4,774 OCL expressions from 504 EMF[5] metamodels coming from 245 systematically selected GitHub repositories. This dataset contained faulty or invalid metamodels and/or OCL expressions. In order to filter them out and keep only those that were correct, we transformed the EMF metamodels into USE-compliant metamodels using the ATL [12] transformation from [4], and we loaded and validated the metamodels and OCL expressions with the USE tool [10]. After curating the original dataset, we finally obtained 325 valid OCL expressions. These expressions contain an average number of 174 ± 274 characters.

4.1.2 Automatic image generation. For each of those 325 expressions within our dataset, we automatically generated 10 images using the 10 fonts that Figure 1 shows, resulting in a final dataset with 3,250 (325x10) images.

4.2 Research Questions and Method

With this empirical study, we plan to answer the following research questions:

RQ1: Which is the best strategy to improve recognition quality for OCL?

To answer this question, we defined the custom languages presented in Section 3.1 for the particular case of OCL and the metamodels of our dataset[6]. We configured Tesseract to use these custom languages and for each of them, we recognized the content of our 3,250 images. Furthermore, we created a prototype implementation of our proposed name-repairing algorithm (R). For each strategy (DP, DPL, DPLD, R), we counted the number of correctly recognized expressions and compared them w.r.t. our baseline model (Default configuration). We assume that an expression is correct if it can be loaded in USE[7][8].

[5] https://www.eclipse.org/modeling/emf/

[6] We created a tool to automate the extraction the domain names from each metamodel

[7] Alternatively, we could have used the Normalized Levenshtein Distance (NLD) between the original expression and its recognized version to measure the quality of the recognition, but we believe that the number of loaded expressions in USE provides a more pragmatical view of our results.

[8] In future experiments we plan to study not only the number correct expressions but also the correctness within a wrongly recognised expression, i.e., once an expression is marked as incorrect, study how serious the problem was. We can do this, for instance, studying the number of incorrectly recognised characters/words within an expression

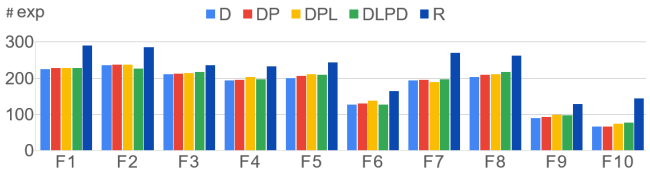

Figure 2. Number of expressions correctly recognized

RQ2: How long do the different strategies take to extract OCL expressions from images?

To answer this question, for each strategy, we computed the running time of the OCR engine during the recognition process. For (R), the times we report include the recognition time of (D) and the time the post-processing algorithm takes. Our experiments were executed in a machine with an AMD Ryzen 9 3900X 12-core 3.80GHz processor, 32GB of RAM and Ubuntu 20.04.

4.3 Results

This section answers our research questions. All the results of our executions are available in our Git repository[9]

4.3.1 Recognition quality. Figure 2 presents for each strategy (D, DP, DPL, DPLD and R), the number of recognized OCL expressions for each font (F1–F10). Table 1 shows for each font and strategy the percentage of expressions recognized and the improvement w.r.t. the baseline model (in brackets). It also shows in the last two rows, for each strategy and in a font-agnostic way, the average percentage and standard deviation of recognized expressions.

	D	DP	DPL	DPLD	R
F1	69.54	70.15 (0.62)	70.46 (0.92)	70.46 (0.92)	89.23 (19.69)
F2	72.62	72.92 (0.31)	72.92 (0.31)	69.85 (-2.77)	88.00 (15.38)
F3	65.23	65.54 (0.31)	65.85 (0.62)	66.77 (1.54)	72.62 (7.38)
F4	59.69	60.00 (0.31)	62.77 (3.08)	60.92 (1.23)	71.69 (12.00)
F5	61.85	63.38 (1.54)	64.92 (3.08)	64.62 (2.77)	75.08 (13.23)
F6	39.38	40.31 (0.92)	42.46 (3.08)	39.08 (-0.31)	50.46 (11.08)
F7	59.69	60.00 (0.31)	58.46 (-1.23)	60.92 (1.23)	83.08 (23.38)
F8	62.77	64.62 (1.85)	64.92 (2.15)	66.77 (4.00)	80.92 (18.15)
F9	27.69	28.62 (0.92)	30.46 (2.77)	29.85 (2.15)	39.69 (12.00)
F10	20.31	20.62 (0.31)	22.77 (2.46)	24.00 (3.69)	44.31 (24.00)
Avg.	53.88	54.62 (0.74)	55.60 (1.72)	55.32 (1.45)	69.51 (15.63)
Std.	18.13	18.16 (0.57)	17.45 (1.49)	17.47 (1.97)	18.19 (5.52)

Table 1. % of expressions recognized and % of improvement w.r.t. D (baseline model)

Our experiments show that the default configuration of Tesseract produces a significant percentage of correct expressions for computer fonts, over 60%, while the use of custom languages entail only small improvements: 4% is the higher

[9] https://github.com/JPery/img2DSL

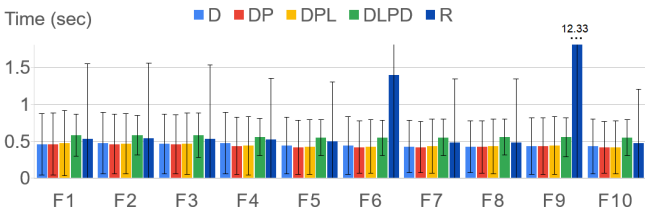

Figure 3. Performance

improvement, obtained in the DPLD case. These results could have been slightly improved by forcing Tesseract to prioritize DSL symbols over English words, but unfortunately, the parameters for such configuration are not operative in its current version[10]. The best results were obtained with our prototypical name-repairing algorithm. Note that, on average, it performs 15.63% better than the baseline model, which entails a significant improvement. Finally, we have observed that all strategies work better for computer-like fonts (F1–F8) than for hand-written fonts (F9–F10). Even in this case, our algorithm obtains the best improvement (24%) for the Handlee font (F10).

4.3.2 Performance (time). Figure 3 shows the average time and standard deviation that we have obtained per font and per strategy.

The performance of D, DP and DPL is, on average, very reasonable (around 0.5 sec), although their standard deviation is significant (around 0.5 sec). This is because the time needed to process an image highly depends on the number of characters it contains, and our dataset is composed by images with expressions of all sizes. Regarding DPLD, the necessity of defining dynamically the custom language according to the metamodel of each OCL expression implies a slight overhead (+0.1 sec). Finally, R (which is the time of default OCR configuration plus the time of repairing algorithm) also entails a small overhead, which in most cases is below the overhead of DLPD. We analyzed the particular cases of fonts F6 and F9 to conclude that the repairing algorithm takes longer because the OCR outputs expressions with more punctuation errors than it does for the other fonts. This makes our algorithm to have to repair many punctuation errors, for which it is not particularly optimized.

To conclude, all the strategies present reasonable execution times, therefore time performance may not hinder the inclusion of OCR support in language workbenches.

5 Related Work

5.1 Extracting Source Code from Videos

Programming screencasts have been used as repositories to extract source code from video and several authors approached this problem by applying OCR techniques. Khormi et al. [15] presented an empirical study to evaluate which OCR engine performs best in source code extraction. Ponzanelli et al. [24] proposed an approach that first identifies fragments including source code and then applies Tesseract on each frame to find and extract code. A similar approach is presented in [14], which briefly comments on two general heuristics to repair OCR errors (no language-based). Yadid et al. [26], in addition to applying Tesseract, proposed the definition of statistical language models for applying corrections at lexical (token model) and syntactical level (line and fragment model).

All these works only consider general-purpose languages, such as Java or Python, with large bodies of code publicly available. In our work, we focus on DSLs—whose available body of code is significantly smaller—and look for alternatives to improve OCR quality by leveraging the available information, i.e. DSL specification and domain data.

5.2 Graphical DSL recognition

Although we may find numerous approaches in the literature for on-line or off-line diagram recognition in the last two decades [16], few of them propose a method to properly extract the text or DSL code, e.g. OCL expressions, embedded into them [2, 13, 20]. Basically, once the text fragments are identified, they are just processed by means of an OCR engine. Therefore, specific OCR support in diagram recognition might also become a use case of the ideas herein proposed.

5.3 Syntax error repairing

With that kind of parsers they can provide engineers with additional features, such as refactoring or code completion. A plethora of works have been published about syntax error repairing algorithms for LR or LL parsers, as for example [5, 7]. Furthermore, the use of statistical languages models for code was suggested by [11], which have been successfully applied in [26]. More recently, Mesbah et a. [19] proposed a Neural Machine Translation network [1] for learning how to repair compilation errors. Medeiros et al. [25] presented an approach to automatically annotate a Parsing Expressions Grammars with labels, and to build their corresponding recovery expressions.

In this work, we have built a simple but effective name-repairing algorithm to correct spelling errors, inspired by [8], which produced a significant improvement in the OCR quality, and which shows that this is line of work that deserves to be explored. In the future, we plan to apply syntax-based repairing technique and study their effectiveness in this context which, as of today, is unknown.

6 Research roadmap

Throughout the development of this work, we have identified several areas for further research.

[10] https://github.com/tesseract-ocr/tesseract/issues/2391#issuecomment-540228564

Integration of OCR support in language workbenches. Although our preliminary results are promising, so OCR engines with the help of repairing algorithms seem to perform reasonably in time and quality, there are still some open issues w.r.t. its integration in language workbenches. All of these issues can be summarized as the necessity of defining a systematic process to automatically derive OCR support for any DSL given its specification and domain data.

OCR output repairing for DSLs. When applying an OCR engine to extract DSL code from an image, we may face two main errors in its output: spelling errors (e.g. a name or a keyword is misspelled) or punctuation errors (e.g. a punctuation sign is missing). As commented in Section 5, there is a plethora of approaches to deal with syntax and compilation errors. However, they have not been broadly studied in the context of OCR support for DSLs. In particular, the definition of statistical languages models for languages without large bodies of code, as DSLs, may play a significant role in this matter. Furthermore, it might be also interesting to evaluate general NL model languages such as GPT-3 [3].

Graphical DSL recognition. Text recognition in diagrams has not been yet properly studied, as aforementioned. One of the main challenges is to identify the mission of each block, i.e., shape, color, a code snippets, NL annotations, etc. A proper text recognition will help towards this goal as well more mature techniques able to identify general shapes (not library-predefined). Finally, these graphical DSLs could come either from computer sources or could be hand-written drawings. Special support for both of them is needed.

7 Conclusions

This work outlines different strategies that can be applied to improve the recognition quality of OCR engines for DSL expressions. By means of an empirical study with OCL, we have studied how these strategies help improve the OCR quality and the overhead that they introduce. The results show that current OCR engines empowered with a repairing strategy have potential to be integrated in language workbenches. We have identified and described a research roadmap showing aspects that promisingly deserve further study.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[2] Martin Bresler, Truyen Van Phan, Daniel Prusa, Masaki Nakagawa, and Vaclav Hlavac. 2014. Recognition System for On-Line Sketched Diagrams. In *Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR*, Vol. 2014-Decem. Institute of Electrical and Electronics Engineers Inc., 563–568. https://doi.org/10.1109/ICFHR.2014.100

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (may 2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[4] Loli Burgueño, Manuel Wimmer, Javier Troya, and Antonio Vallecillo. 2013. TractsTool: Testing Model Transformations based on Contracts. In *Proc. of the MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with MODELS'13 (CEUR Workshop Proceedings)*, Vol. 1115. CEUR-WS.org, 76–80.

[5] Michael G. Burke and Gerald A. Fisher. 1987. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 2 (1987), 164–197. https://doi.org/10.1145/22719.22720

[6] Jordi Cabot and Martin Gogolla. 2012. Object Constraint Language (OCL): A Definitive Guide. In *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. 58–90. https://doi.org/10.1007/978-3-642-30982-3_3

[7] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. 2002. Repairing syntax errors in LR parsers. *ACM Transactions on Programming Languages and Systems* 24, 6 (2002), 698–710. https://doi.org/10.1145/586088.586092

[8] Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7, 3 (1964), 171–176. https://doi.org/10.1145/363958.363994

[9] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Comput. Lang. Syst. Struct.* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[10] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. *Sci. Comp. Prog.* 69 (2007), 27–34.

[11] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. *Proceedings - International Conference on Software Engineering* (2012), 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[12] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Science of Computer Programming* 72, 1-2 (2008), 31–39.

[13] Bilal Karasneh and Michel R.V. Chaudron. 2013. Img2UML: A System for Extracting UML Models from Images. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 134–137. https://doi.org/10.1109/SEAA.2013.45

[14] Kandarp Khandwala and Philip J. Guo. 2018. Codemotion: Expanding the Design Space of Learner Interactions with Computer Programming Tutorial Videos. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3231644.3231652

[15] Abdulkarim Khormi, Mohammad Alahmadi, and Sonia Haiduc. 2020. A Study on the Accuracy of OCR Engines for Source Code Transcription from Programming Screencasts. In *MSR 2020 Technical Papers (Preprint)*. https://doi.org/10.1145/3379597.3387468

[16] Edward Lank, Jeb S. Thorley, and Sean Jy-Shyang Chen. 2000. An interactive system for recognizing hand drawn UML diagrams. In *Proceedings of CASCON 2000, Toronto, Canada*. 7. http://dl.acm.org/

citation.cfm?id=782034.782041

[17] Sérgio Medeiros and Fabio Mascarenhas. 2018. Syntax error recovery in parsing expression grammars. *Proceedings of the ACM Symposium on Applied Computing* (2018), 1195–1202. https://doi.org/10.1145/3167132.3167261

[18] Jamshed Memon, Maira Sami, and Rizwan Ahmed Khan. 2020. Handwritten Optical Character Recognition (OCR): A Comprehensive Systematic Literature Review (SLR). arXiv:cs.CV/2001.00139

[19] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to repair compilation errors. *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), 925–936. https://doi.org/10.1145/3338906.3340455

[20] Valentín Moreno, Gonzalo Génova, Manuela Alejandres, and Anabel Fraga. 2016. Automatic classification of web images as UML diagrams. In *Proceedings of the 4th Spanish Conference on Information Retrieval - CERI '16*, Vol. 14-16-June. ACM Press, New York, New York, USA, 1–8. https://doi.org/10.1145/2934732.2934739

[21] Sergey I. Nikolenko. 2019. Synthetic Data for Deep Learning. arXiv:cs.LG/1909.11512

[22] Jeroen Noten, Josh G. M. Mengerink, and Alexander Serebrenik. 2017. A Data Set of OCL Expressions on GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, 531–534. https://doi.org/10.1109/MSR.2017.52

[23] Object Management Group. 2015. *Unified Modeling Language (UML) Specification. Version 2.5.* OMG document formal/2015-03-01.

[24] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano DI Penta, Sonia Haiduc, Barbara Russo, and Michele Lanza. 2019. Automatic Identification and Classification of Software Development Video Tutorial Fragments. *IEEE Transactions on Software Engineering* 45, 5 (may 2019), 464–488. https://doi.org/10.1109/TSE.2017.2779479

[25] Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. 2020. Automatic syntax error reporting and recovery in parsing expression grammars. *Science of Computer Programming* 187 (2020), 102373. https://doi.org/10.1016/j.scico.2019.102373

[26] Shir Yadid and Eran Yahav. 2016. Extracting code from programming tutorial videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2016.* ACM Press, New York, New York, USA, 98–111. https://doi.org/10.1145/2986012.2986021