RTFM: Towards Understanding Source Code using Natural Language Processing

Maximilian Galanis[a], Vincent Dietrich[b], Bernd Kast[c] and Michael Fiegert[d]
Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81739 Munich, Germany

Keywords: Semantics-based Software Engineering, Information Extraction, Natural Language Processing, Planning.

Abstract: The manual configuration of today's autonomous systems for new tasks is becoming increasingly difficult due to their complexity. One solution to this problem is to use planning algorithms that can automatically synthesize suitable data processing pipelines for the task at hand and thus simplify the configuration. Planners usually rely on models, which are created manually based on already existing methods. These methods are often provided as part of domain specific code libraries. Therefore, using existing planners on new domains requires the manual creation of models based on the methods provided by other libraries. To facilitate this, we propose a system that generates an abstract semantic model from C++ libraries automatically. The necessary information is extracted from the library using a combination of static source code analysis to analyze its header files and natural language processing (NLP) to analyze its official documentation. We evaluate our approach on the perception domain with two popular libraries: HALCON and OpenCV. We also outline how the extracted models can be used to configure data processing pipelines for the perception domain automatically by using an existing planner.

1 INTRODUCTION

Autonomous systems are expected to adapt to changing tasks quickly. A good example for this are industrial assembly tasks with small lot sizes. Such cases demand robots that quickly adapt to new work pieces. Unfortunately, the rapid reconfiguration of these complex autonomous systems is infeasible when undertaken manually. One solution to this problem is the use of planning algorithms for automatic reconfiguration. They can synthesize suitable data processing pipelines automatically and, therefore, reduce the needed engineering and time requirements to reconfigure autonomous systems (Kast et al., 2018).

Planners usually require models that formalize the description of the target domain. The algorithms and data structures used in the models are often provided by code libraries. These libraries are usually domain specific and contain a wide variety of methods and algorithms for their target domain. To add a new domain to the planning algorithm, it is therefore necessary to model the algorithms and data structures used in the library. However, such models are still created manually. The main reason for this is that the needed semantic information is often not explicitly described as part of the source code. It is usually only described in the documentation of the library, which engineers can understand using their background knowledge. Because understanding not only the source code but also the documentation requires significant domain knowledge, the creation of such models is a time consuming task.

We propose a system that can create an abstract semantic model of a given C++ library automatically. Our approach combines source code analysis to extract the application programming interface (API) with state-of-the-art natural language processing (NLP) to understand the code documentation. The source code analysis is used to provide a dependable knowledge base, which can be augmented with additional information but never be falsified. We use this knowledge base and extend it with semantic information that is extracted from the official documentation. Our system creates semantic models of the functions and data structures of the library. It is machine-readable and represents a high-level abstraction of the underlying source code. Thereby, the model eliminates the need to understand and manipulate source code, which has the potential to reduce the time and engineering knowledge requirements. Our model represents the data structures of the library as *concepts*

[a] https://orcid.org/0000-0001-8483-0535
[b] https://orcid.org/0000-0003-0568-9727
[c] https://orcid.org/0000-0001-7838-3142
[d] https://orcid.org/0000-0002-6371-6394

Galanis, M., Dietrich, V., Kast, B. and Fiegert, M.
RTFM: Towards Understanding Source Code using Natural Language Processing.
DOI: 10.5220/0009826604300437
In *Proceedings of the 17th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2020)*, pages 430-437
ISBN: 978-989-758-442-8

and the functions operating on these data structures as *operators* (Kast et al., 2019). More generally, one can also view operators as algorithms and concepts as the (semantic) data structures the algorithms work on. In practice, operators always have concepts as inputs and outputs, with which we model the function parameters and return types.

Throughout this paper, we focus on the perception domain for which excellent libraries, like HALCON[1] and OpenCV[2] exist. We demonstrate the applicability of our approach by extracting models from both computer vision libraries and outline how the extracted model can be used to synthesize perception pipelines.

1.1 Problem Analysis

Extracting abstract semantic models from a C++ library and its documentation is a hard problem, as such a system must adapt to varying documentation and API styles. Therefore, we evaluated the differences in both aspects for three perception libraries: HALCON, OpenCV and the PCL[3]. The analysis shows that the declaration of a function as the sole information source is usually insufficient for the automatic creation of an abstract model. Consider a function that returns *void* and has a pointer as its parameter. In this case, the C++ declaration provides no information on whether this parameter is an input, an output or both, of this function. While the *const* qualifier can define parameters as unchanging and thus as inputs, it is often not available. However, this information is described as part of the function's documentation. Either with explicit labels, or as part of the natural language description. Therefore, the source code but especially its documentation provide additional meta information about the functions, like whether a parameter is an input or an output.

To add semantic knowledge to the model, the parameter's data type is important. Similarly to the input/output information, knowledge about the semantic content is contained in both the API and the documentation. A library could define a class *Image*, which it uses to represent images. Unfortunately, it is common to obfuscate this information by using wrapper classes that can contain a wide variety of data types, like the *HObject*[4] class in HALCON. If this is the case, only the documentation describes the semantic type of a parameter. Similarly to before, this meta information can either be encoded by explicit labels in the documentation (e.g., in HALCON), or in the natural language description of the parameter (e.g., in OpenCV).

Table 1: Comparison of HALCON, OpenCV and the PCL concerning their API and documentation. *NL* stands for natural language.

	Library		
Documentation	**HALCON**	**OpenCV**	**PCL**
input/output	explicit	NL	explicit
semantic type	explicit	NL	NL
availability	custom	doxygen	doxygen
API			
data types	obfuscated	partly obfuscated	explicit
const qualifier	consistent	inconsistent	consistent

Our findings are illustrated in Table 1. The comparison motivates a highly modular design to cope with the heterogeneity in the design of both the API as well as the official documentation of these libraries. Furthermore, the examples of OpenCV and PCL illustrate that it is necessary to understand the natural language documentation in order to obtain, e.g., the semantic types of parameters.

1.2 Contribution

In this paper, we make the following contributions: (1) Proposal of an architecture for knowledge extraction with multiple information sources. (2) Evaluation of the combination of source code analysis in combination with state-of-the-art NLP methods on the task of interpreting source code and its documentation. (3) Demonstration of the applicability of a fully functional model extraction system (MES) on the HALCON and OpenCV libraries.

2 RELATED WORK

In this section, we discuss how NLP and information extraction tools are currently used in the software engineering field to extract information from meta knowledge sources like the official documentation or forum discussions.

Closely related to this work, is the trend in software engineering to explore methods to interpret source code and the corresponding natural language documents. A good example of this is the work presented by (Zhang and Hou, 2013). They use semantic analysis to extract negative mentions of API features from forum discussions and then extract problematic API features. Another way to interpret natural language is presented by (Zhong et al., 2009) who

[1] https://www.mvtec.com/products/halcon
[2] https://opencv.org
[3] http://pointclouds.org/
[4] https://www.mvtec.com/doc/halcon/1811/en/HObject.html

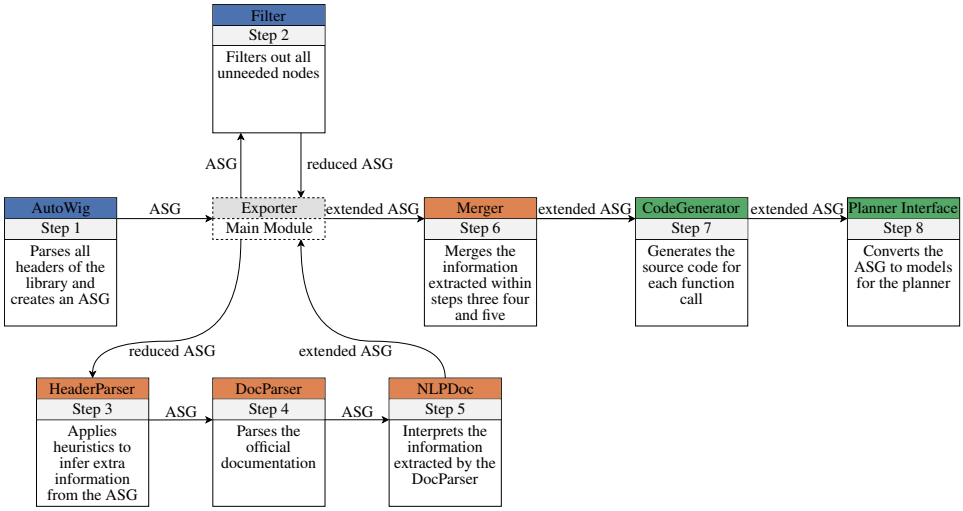

Figure 1: Overview of the different modules used in the harvester. Every block corresponds to a module of the harvester. The structure of this system can be divided into three phases. Preprocessing (blue), knowledge acquisition (orange) and export (green). Each phase is divided into several modules.

use NLP to infer resource specifications from natural language documentation. In their approach, they map class methods to five predefined types of methods based on their associated description. To classify the class methods into the categories, they use word-level classification of the corresponding documentation based on Hidden Markov Models (HMMs) (Rabiner and Juang, 1986). Using these classes, they can then detect bugs in source code by, e.g., detecting that a manipulation method is called before a creation method. Another classification problem is approached by (LeClair et al., 2018), who attempt to do a high-level classification of software projects into predefined categories based on their source code. For this, the authors use word embeddings that were pretrained to encode the similarity of high-level project descriptions and low-level source code.

Probably most similar to our work is presented by (Li et al., 2018). They tackle the API accessibility issue by mining an API knowledge graph from multiple API sources (e.g., official documentation, API tutorials) so that developers can be warned of possible caveats of the APIs before they run into problems. To extract the API caveats from the documentation, (Li et al., 2018) crawl the online API documentation and extract the textual description of the APIs. Then, they construct an API skeleton graph from the reference documentation by exploiting the provided information like the fully qualified names of the entities, inheritance information, data type references, function/method return types, and function/method parameter types. To extract the API caveat sentences, a pattern-based approach (e.g., regular expressions) was used. Finally, using co-reference resolution, sentences containing API caveats are linked to their corresponding API entity in the knowledge graph. Their user study provides evidence that this knowledge graph improves API caveat accessibility among developers.

Like most research in the field of software engineering, our approach focuses on the quest to reduce the amount of work needed to build and maintain software using program analysis and information extraction. However, unlike the existing research, which tries to support the developer, the here presented system goes further and explores ways to automate the role of the software developer partially.

3 METHODOLOGY

Our MES is designed to be modular and expandable. We achieve this by using the abstract semantic graph (ASG) of the target library as our knowledge base throughout the analysis. The ASG is extracted from the header files of the target library using static source code analysis and represents all the code components that are part of the source as nodes and the semantic relationships between the nodes as edges. Our system is designed in such a way that all modules have access to the current knowledge in the form of the ASG. Any information that is extracted by the modules, like the extracted semantic value for each parameter or whether a parameter is an input or output, is then added directly to the correct nodes in the ASG themselves. Figure 1 illustrates this design and the currently implemented modules.

Out of the modules in the MES, the *Exporter* module is unique in that it does not implement any functionality itself, but acts as the main module and defines the order in which the other modules are called. Therefore, there are no architectural restrictions in the order in which independent modules can be called. Each module has access to all of the information that was harvested so far. This also enables the creation of custom pipelines for different target libraries. This flexibility is fundamental because a fully generic solution to the model extraction problem is not feasible yet, as illustrated in subsection 1.1. The *Exporter* also provides a default pipeline that is used for all following experiments.

Parsing the Library and Filtering. First, we obtain our knowledge base, the ASG of the target library. For this, we use the parser module of AutoWIG[5](Fernique and Pradal, 2017), which internally uses Clang[6] to create one coherent ASG for the header files of the target library. By using their parser module instead of using Clang directly, we obtain an ASG instead of the abstract syntax trees that Clang produces. The decisive reason to rely on only the header files for the source code analysis is that the full source code is not necessarily available, as it is not uncommon to only distribute a binary version. The resulting ASG is then modified to include several additional attributes. These additional attributes allow us to store the information that is extracted in the following modules directly inside the correct nodes of the ASG, which drastically simplifies the data handling.

The next step is to decide which nodes in the ASG should be processed. By default, the ASG includes not only the code components of the given library but also all code components of the library's dependencies, like the C++ standard library. Therefore, we filter the ASG to only include nodes of a given namespace, as it is common practice to have all user-facing code components of a library in a unified namespace.

Applying Heuristics. After obtaining and filtering the ASG, we try to extract the semantic knowledge that is contained the ASG itself. As elaborated in subsection 1.1, the source code (and thus the ASG) can contain some semantic information about whether function parameters are inputs or outputs. Therefore, we extract this information with the *HeaderParser* module, which analyzes both, the parameter qualifiers and their data types. This module then marks all parameters or return values that are represented by fundamental types[7], as well as all parameters that are qualified with the *const* qualifier, as inputs. Like *const* parameters, parameters of fundamental types are also guaranteed to be inputs because they cannot contain any pointers, references or classes.

Extracting and Analyzing the Documentation. After extracting the semantic information of the source code, our system proceeds to analyze the official documentation of the target library. However, to analyze the documentation it must be parsed first. For this task, we use documentation specific parsers that extract the natural language comments of the code components that are contained in the provided ASG. This is necessary as the documentation styles of different libraries varies significantly. After linking the extracted comments to the corresponding nodes in the ASG, there are several ways to analyze it. In the case of HALCON, all the needed information[8] is already explicitly stated in the documentation and thus parsed directly by the *DocParser* and its custom heuristic. Unfortunately, such a well-documented library is rare. Therefore, we use NLP to analyze the documentation. In particular, we analyze the parameter comments, as they contain the information about a parameter's mode (input or output) as well as its semantic content. To extract this knowledge, a pre-trained neural network that is based on the transformer architecture (Vaswani et al., 2017), is used. This network is then fine-tuned on two tasks: (1) Input/Output classification (I/O task). (2) Semantic type classification (Sem_type task). The trained network is then used to classify the parameters and the resulting labels are added to the corresponding node in the ASG.

Merging the Different Knowledge Sources. As the last step during the knowledge extraction, we merge the information from the different modules. Consider the information whether a function parameter is an input or output. During the analysis in the MES, several modules like the *HeaderParser*, or the *NLPDoc* module might classify the parameter into input and output. In case these information sources contradict each other, a way to fuse both information sources is needed. To accomplish this, the *Merger* module uses a weighted majority vote to determine the final result for each of the two classification tasks given the results of several modules. This implemen-

[5]https://autowig.readthedocs.io/en/latest/index.html
[6]https://clang.llvm.org/

[7]In C++, this includes the boolean type as well as integer, float and character types: https://en.cppreference.com/w/cpp/language/types.
[8]This includes whether a parameter is an input or output of the function and the parameter's semantic value.

tation was chosen to be flexible enough to allow for additional modules that might be added in the future.

Generating the Source Code. Once the needed information is extracted and merged, the MES generates the necessary glue code. This code is responsible for calling the underlying function using the inputs provided by the input concepts of the operator as well as passing the outputs of the function to the output concepts. The *CodeGenerator* relies on cppyy (Lavrijsen and Dutta, 2016)[9] to create Python bindings for the underlying library automatically.

By using these bindings, the underlying C++ library can be called directly within Python, which simplifies the code generation process.

Interface to the Planner. The last module of our system is responsible for exporting the harvested information to the used planner. As mentioned before, our planner utilizes a bi-modal approach, which splits the model into procedural knowledge (operators) and declarative knowledge (concepts). Therefore, the last module interfaces with the existing planner (Kast et al., 2018) to create the necessary operators and concepts. If available, concepts are created to reflect the semantic type of the data they hold. In case no semantic type information was extracted, this module falls back to using the data type of the parameter that is being processed. Because of the modularity of our MES, the extracted information could also be exported to other planners by adding an additional planner interface.

4 EVALUATION

Automatically Synthesizing Operator Pipelines. To showcase that the extracted models are suitable for automatically synthesizing data processing pipelines, our planner was presented with a select list of extracted models from the HALCON library, some starting facts, and a goal [10]. Additionally, this experiment motivates the extraction of semantic information during the model extraction.

The initial inputs (facts) are two parameters and an image, as illustrated in Figure 2. The goal of the planner is then to find a suitable chain of operators that produce the given goal, which, in this case, is an instance of the concept *number* with the value 2.3. In this experiment, the paper clip in the input image is rotated by 2.3 degrees. Thus, by solving this task, the planner indirectly solves a range of problems - identifying the angle of single objects in high contrast images. As illustrated in Figure 3, the planner is able to find a suitable operator pipeline that solves the given task. While the provided example is relatively simple, this approach has been shown to scale to more complex perception pipelines (Dietrich et al., 2018). It has also been shown that the necessary fine-tuning of parameters can be addressed with optimization based approaches (Dietrich et al., 2019).

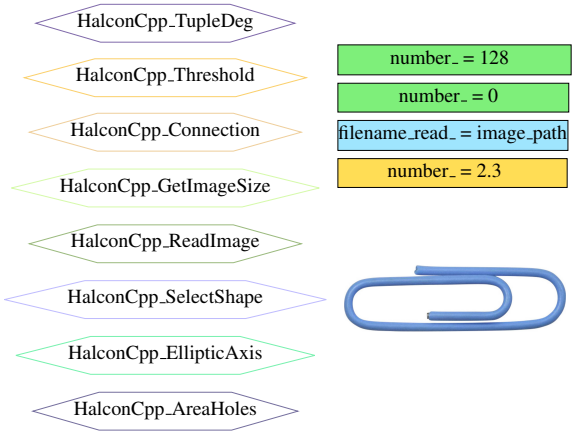

Figure 2: Initially available operators (hexagons) and instances (green and blue) as well as the desired goal instance (yellow). The given image path points to the depicted paper clip.

Table 2: Planning performance with and without semantic type information.

Semantic Information	Visited Nodes	Improvement
No semantic types	324	
With semantic types	115	64.5 %

In their C++ interface, HALCON uses very few data types, such as *HObject* and *HTuple*. However, our planner mainly evaluates the compatibility of operators based on their input and output concepts. HALCON's API design, therefore, results in a vast search space. As illustrated in Table 2, the additional semantic information, which we extracted from HALCON's documentation drastically improves the planning performance. While the semantic type information provided by explicit labels in HALCON's documentation is valuable, this experiment also shows its limits. Because the instance was labeled as a *number* and could not be defined more precisely as e.g., an angle in degrees, defining goals could be problematic in more complex scenarios.

[9] https://cppyy.readthedocs.io/en/latest/

[10] Facts and goals are specific instantiations of concepts. An example of a specific instantiation of a concept is the number 42, which is a specific instance of the concept *number*.

Figure 3: Data processing pipeline synthesized by the planner. Operators are illustrated as ovals, while concepts are illustrated with rectangles. The content of the more complex concepts *Image* and *Region* is not displayed.

Table 3: Obtained NLP accuracy on the different tasks on both libraries. All results are the average of 5 runs with different seeds. Table 7 gives a brief overview of all used datasets.

Task	Dataset	Accuracy
I/O	io	87.2 %
I/O	io_balanced	84.2 %
Sem_type	sem_type	89.6 %
I/O	cv_io	97.9 %
I/O	cv_io_balanced	89.8 %

Natural Language Processing Results. Unfortunately, only few libraries include machine-readable labels in their documentation. Generally, it is necessary to analyze the natural language comment that is associated with the function parameters to obtain the needed semantic information.

To be able to evaluate the contents and meaning of the natural language comments, we use the SciBERT (Beltagy et al., 2019) language model (LM). We then fine-tuned separate SciBERT instances on the two tasks for HALCON as well as OpenCV using the labeled data that is provided by HALCON's documentation and a manually labeled dataset for OpenCV. For the I/O classification task, we provide an additional *balanced* dataset for both libraries, in which the samples of both classes are equally likely. This is needed because the class *input* is much more common in the I/O classification datasets. Table 8 illustrates ten samples of our dataset for each library, while Table 7 summarizes the used datasets. The samples of our dataset show that we use a combination of the parameter comment and the parameter name as the input to the LM. We found that this slightly improves the classification performance of the fine-tuned LM, as illustrated in Figure 4. Finally, we apply similar hyperparameters during our experiments to the ones (Beltagy et al., 2019) use during their evaluation. The most important hyperparameters are listed in Table 6.

As illustrated in Table 3, using the combination of the parameter comments and the parameter name as features to classify the parameters yields promising results on both tasks. Even on balanced datasets, the obtained accuracy only drops by a small margin, which shows that the models indeed correctly learn to discern the different classes.

Table 4: Performance of SciBERT (fine-tuned on HALCON) on the OpenCV dataset. All results are the average of 5 runs with different seeds.

Task	Dataset Train	Dataset Eval	Accuracy
I/O	io	cv_io	90.1 %
I/O	io	cv_io_balanced	84.2 %
Sem_type	sem_type	cv_sem_type	33.9 %

Transferring Knowledge. Creating a labeled dataset for each new library that should be modeled contradicts the basic thought of an automatic MES. Therefore, we examine the performance of the LMs that were fine-tuned on HALCON on OpenCV - a new domain. The results of this evaluation are illustrated in Table 4. The results show that the model fine-tuned on the I/O classification task does generalize well to this unseen library and achieves a 84.2 % accuracy on the *cv_io_balanced* dataset. This result is comparable to the accuracy this model achieved in its training domain: HALCON.

Unfortunately, the classification of the parameters into their semantic types does not perform as well. While the model trained on the Sem_type task achieved 89.6 % in its training domain, it only achieves 33.9 % on OpenCV, which is only barely better than chance in this case. However, images are classified correctly with 85.5 % in the *cv_sem_type* dataset, suggesting that images are described similarly, allowing the classifier to perform well for this class.

The samples from the datasets illustrated in Table 8 help to understand some of the problems concerning the transfer performance of the Sem_type task. Consider the semantic type *image*. All of the samples of this class contain the word *image* making it easier to classify. This is the case for samples in both libraries and might explain the good performance the LM achieved for this particular class. For the other classes, however, classification is more difficult, even for an experienced engineer. One example for this is the semantic type *string*. The samples 6 and 16 in Table 8 are both strings but unlike the image class, the samples do not share common words. Additionally, both parameters could just as well be a number to select pre-defined options in an enum given the parameter comment.

Table 5: NLP performance after merging the information from the *HeaderParser* module. All results are the average of 5 runs with different seeds.

Task	Dataset Train	Dataset Eval	Accuracy
I/O	io	cv_io	93.2 %
I/O	io	cv_io_balanced	90.1 %

Performance after Merging. The I/O classification task is special in that it has two knowledge sources currently: the ASG, and the NLP analysis of the parameter comment. Thus, the knowledge from the ASG can be merged with the result from the NLP analysis. In the case of this experiment, the extra information is used to minimize the reliance on NLP. The current implementation of the *Merger* module uses following weights: (1) *HeaderParser*: 4 (2) *DocParser*: 2 (3) *NLPDoc*: 1 This means that NLP is only used on parameters that could not be classified with certainty by the *HeaderParser* module i.e., parameters that are *not* qualified with the *const* qualifier and are *not* a fundamental type. As shown in Table 5, this method improves the final accuracy.

5 CONCLUSION

In this paper, we presented an approach to extract abstract semantic models of C++ libraries automatically, which we evaluated on the perception domain with two popular computer vision libraries. We also outlined how the extracted models of the perception libraries can be used within a planner to further automate the creation of perception pipelines. It, therefore, lowers the engineering barriers to develop robotics and automation solutions that can adapt to new tasks automatically via planning. Our approach is based on the combination of static source code analysis and NLP, which is used to interpret the corresponding documentation. Because we did not make any domain specific assumptions, we expect our approach to perform similarly on other domains.

Our evaluation shows the benefits of additional semantic information on the planning performance. The required semantic information can be extracted with a heuristics-based parser in case a machine-readable documentation is provided (i.e., HALCON). More generally, however, it is necessary to use NLP to extract semantic knowledge. Therefore, we fine-tuned a state-of-the-art LM on two classification tasks to extract semantic information. Our results show that this approach works well in the training domain. Unfortunately, applying the trained model to another library showed mixed results. While it worked well for the input/output classification task, the semantic type classification task showed the limits of the used LM.

Future work could extend the here described static analysis with dynamic program analysis to validate the extracted labels. Additionally, an interesting research direction would be to take more information into account, like the other parameters of the function or the functions' description. Finally, a more diverse training data set could improve the transfer performance.

REFERENCES

Beltagy, I., Lo, K., and Cohan, A. (2019). SciBERT: A pretrained language model for scientific text. *2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, arXiv:1903.10676.

Dietrich, V., Kast, B., Fiegert, M., Albrecht, S., and Beetz, M. (2019). Automatic configuration of the structure and parameterization of perception pipelines. In *2019 19th International Conference on Advanced Robotics (ICAR)*, pages 312–319.

Dietrich, V., Kast, B., Schmitt, P., Albrecht, S., Fiegert, M., Feiten, W., and Beetz, M. (2018). Configuration of perception systems via planning over factor graphs. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6168–6174.

Fernique, P. and Pradal, C. (2017). AutoWIG: Automatic generation of Python bindings for C++ libraries. *CoRR*, abs/1705.11000.

Kast, B., Albrecht, S., Feiten, W., and Zhang, J. (2019). Bridging the gap between semantics and control for industry 4.0 and autonomous production. In *2019 15th International Conference on Automation Science and Engineering (CASE)*, pages 780–787.

Kast, B., Dietrich, V., Albrecht, S., Feiten, W., and Zhang, J. (2018). A hierarchical planner based on set-theoretic models: Towards automating the automation for autonomous systems. In *16th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*.

Lavrijsen, W. T. L. P. and Dutta, A. (2016). High-performance Python-C++ bindings with pypy and cling. In *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, pages 27–35.

LeClair, A., Eberhart, Z., and McMillan, C. (2018). Adapting neural text classification for improved software categorization. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 461–472.

Li, H., Li, S., Sun, J., Xing, Z., Peng, X., Liu, M., and Zhao, X. (2018). Improving API caveats accessibility by mining API caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193.

Rabiner, L. and Juang, B. (1986). An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–16.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008.

Zhang, Y. and Hou, D. (2013). Extracting problematic API features from forum discussions. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 142–151.

Zhong, H., Zhang, L., Xie, T., and Mei, H. (2009). Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318.

APPENDIX

Table 6: Hyperparameters used to fine-tune the SciBERT language model on the I/O and Sem_type classification tasks.

Learning Rate	2×10^{-5}
Number of Epochs	3
Batch Size	32
Warmup Ratio	6 %
Weight Decay	0.0
Learning Rate Decay	Linear

Table 7: Overview of the used datasets.

Library	Task	Dataset Name
HALCON	I/O	io
	I/O	io_balanced
	Sem_type	sem_type
OpenCV	I/O	cv_io
	I/O	cv_io_balanced
	Sem_type	cv_sem_type

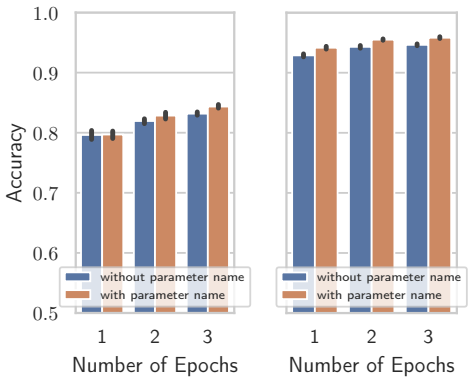

Figure 4: Illustration of the effects of adding the parameter name to the parameter comment on the I/O (left) and Sem_type (right) classification tasks. Drawn with 95 % confidence intervals on ten runs with varying seeds.

Table 8: Illustration of 10 samples from both the HALCON and the OpenCV dataset.

#	Fully Qualified Name	Parameter Name	Parameter Comment prefixed with Parameter Name	I/O	Sem_type
1	::HalconCpp::ConnectGridPoints	Image	Image. Input image.	input	image
2	::HalconCpp::PowerReal	Image	Image. Input image in frequency domain.	input	image
3	::HalconCpp::InpaintingCed	InpaintedImage	InpaintedImage. Output image.	output	image
4	::HalconCpp::ConvertPoseType	OrderOfTransform	OrderOfTransform. Order of rotation and translation.	input	string
5	::HalconCpp::GetIoDeviceParam	GenParamName	GenParamName. Parameter names.	input	string
6	::HalconCpp::CreateShapeModelXld	Optimization	Optimization. Kind of optimization and optionally method used for generating the model.	input	string
7	::HalconCpp::InpaintingCt	Sigma	Sigma. Pre-smoothing parameter.	input	number
8	::HalconCpp::DistRectangle2ContourPointsXld	Length2	Length2. Second radius (half width) of the rectangle.	input	number
9	::HalconCpp::XRangeFunct1d	XMax	XMax. Largest x value.	output	number
10	::HalconCpp::BackgroundSeg	Foreground	Foreground. Input regions.	input	region
11	::cv::calcOpticalFlowFarneback	prev	prev. first 8-bit single-channel input image.	input	image
12	::cv::fastNlMeansDenoisingColored	src	src. Input 8-bit 3-channel image.	input	image
13	::cv::decolor	grayscale	grayscale. Output 8-bit 1-channel image.	output	image
14	::cv::destroyWindow	winname	winname. Name of the window to be destroyed.	input	string
15	::cv::haveImageWriter	filename	filename. File name of the image	input	string
16	::cv::imencode	ext	ext. File extension that defines the output format.	input	string
17	::cv::erode	borderValue	borderValue. border value in case of a constant border	input	number
18	::cv::minEnclosingCircle	radius	radius. Output radius of the circle.	output	number
19	::cv::rectangle	thickness	thickness. Thickness of lines that make up the rectangle. Negative values, like #FILLED, mean that the function has to draw a filled rectangle.	input	number
20	::cv::hconcat	dst	dst. dst output array. It has the same number of rows and depth as the src1 and src2, and the sum of cols of the src1 and src2.	output	other_base_type