Introduction of an Assistant for Low-Code Programming of Hydraulic Components in Mobile Machines

Eva-Maria Neumann[1](✉), Fabian Haben[2], Marius Krüger[1], Timotheus Wieringa[3], and Birgit Vogel-Heuser[1,4]

[1] TUM School of Engineering and Design, Institute of Automation and Information Systems, Technical University of Munich, Boltzmannstr. 15, 85748 Garching, Germany
{eva-maria.neumann,marius.krueger,vogel-heuser}@tum.de

[2] Stadtwerke München, Emmy-Noether-Straße 2, 80992 Munich, Germany
fabian.haben@protonmail.com

[3] HAWE Hydraulik SE, Einsteinring 17, 85609 Aschheim, Germany
t.wieringa@hawe.de

[4] Core Member of MDSI, Member of MIRMI, Munich, Germany

Abstract. The increasing functionality of automation software in complex mechatronic systems such as construction machinery is a major challenge for companies to remain competitive. A major difficulty is that the software development in construction machinery often involves employees from different disciplines who have technological expertise about the process but little software background. Low-code platforms allow software to be developed intuitively without extensive programming knowledge. However, in mechatronics, the resulting programs are often facing the so-called scaling-up problem that occurs in case highly complex technical processes are implemented using graphical programming languages. This paper thus presents an assistant that supports the programming of automation software on low-code platforms to reduce the complexity of the resulting code. Static code analysis and machine learning are combined to enable predictions about software blocks to be used. For the example of the low-code platform eDesign, a graphical programming platform developed by HAWE Hydraulik SE, it is shown how users of the platform can be assisted in creating maintainable, reusable automation software in the construction machinery sector.

Keywords: mobile machines · construction machinery · low-code · visual programming languages · static code analysis · data mining · assistance system

1 Motivation and Introduction

The increasing complexity of automation software in mechatronic systems and the associated problems with code maintainability are a major challenge for companies to survive in the global market in the long term. A major difficulty is that the development of software for mechatronic systems often involves technicians who have technological expertise about the process but little software background [1]. To this end, low-code platforms allow software to be developed via an intuitive graphical interface even without extensive programming knowledge, usually using *Visual Programming Languages (VPL)*. Although such platforms have already found their way into the world of automation technology, the resulting programs are often difficult to understand and maintain due to the high complexity of the technical processes to be controlled, leading to the so-called *scaling-up problem* in graphical languages [2]. In the field of high-level language software, there are already a large number of programming assistants to minimize the complexity of the code already during programming. For automation software in mechatronics, however, such approaches are hardly available so far [3]. This paper therefore presents a programming assistant that supports the programming of automation software on low-code platforms to reduce the complexity of the resulting code. To develop the system, approaches from static code analysis and machine learning were combined to enable predictions about software blocks to be used and optimal assistance for the user. Using the low-code platform eDesign, a graphical programming platform developed by HAWE Hydraulik SE, it is demonstrated how users of the platform can be assisted in creating maintainable, reusable automation software in the construction machinery sector. The assistance is provided on three levels:

- *Calculation and display of complexity metrics:* By quantitatively evaluating various properties of the graphical programs, the user receives direct feedback on which adjustment screws can improve comprehensibility.
- *Encapsulation of recurring function block combinations:* If recurring sub-functions consisting of several function blocks are identified in a project, they can be encapsulated in one function block, thus significantly reducing complexity.
- *Suggestions for function blocks to be used:* Based on machine learning algorithms, the programming assistant learns from the structure of existing projects and can thus generate live suggestions for function blocks to complete the program during programming.

The presented paper enlarges previous results published in [1] by providing more details on the implemented programming assistant and a concrete.

2 State-of-the-Art in Analyzing and Assisting Low-Code Development

Static code analysis is an established lever to identify optimization potentials without executing the code and quantifying specific quality characteristics [2], e.g., using software metrics. However, in the field of VPL, static code analysis is not yet widespread and existing approaches are often tailored to an individual language, such as MathWork's *Model Metrics* [3] for Simulink. Besides the syntactical program composition, also the layout quality strongly influences a VPL program's understandability, i.e., the visual arrangement of blocks and their connections. Taylor et al. [4] propose a set of metrics to quantify the graphical design quality. In the field of IEC 61131-3 compliant automation software, Capitán and Vogel-Heuser [5] use metrics adapted from IEC 61131-3 by Halstead [6] and McCabe [7], as well as metrics by Henry and Kafura [8] and the Module

Size Uniformity Index (MSUI) by Sarkar et al. [9]. Fischer et al. [10] investigate the overall complexity of graphical and textual IEC 61131-3 software. For this purpose, several metrics are used that evaluate different classes of software complexity. A common approach to reducing complexity is the encapsulation of recurring functionalities in reusable units. Duplicated code or so-called *code clones* in the software can be a hint for recurring functionalities that are reused via *Copy, Paste & Modify,* and, thus, can be a starting point for standardization [11]. One of the first algorithms for finding clones in graph-based modeling languages is *CloneDetective* [12], which can be adapted to various textual programming languages and also to VPLs such as Simulink. However, this requires suitable translators for each VPL. Recently, Rosiak et al. introduced an approach capable of identifying code clones in graphical IEC 61131-3 languages based on similarity metrics [13], which, however, does not provide live assistance during programming. There is a variety of approaches from static code analysis to identify highly complex or duplicated code structures in VPL and low-code, but available methods are usually tailored to specific VPL and cannot be transferred to other languages without adaptions. Additionally, assistance or suggestions to compensate high complexity values is usually not included.

Data mining is the process of finding functional structures in existing data and is thus ideally suited for extracting knowledge from code analysis data that can be used to build programming assistants. Bruch et al. [14] use data mining to derive suggestions for the programmer based on existing projects by analyzing the context, the frequency of calls to existing methods, and rules derived from previous projects. Further approaches to formulate suggestions to complete code during programming are based on natural language processing [15] or neural networks [16, 17]. The *SimVMA* system for Simulink [18] is capable of predicting complete systems in an early stage based on partially implemented systems and generates individual next steps as suggestions. The approach from Deng et al. [19] is based on the analysis of subgraphs in available projects in a graph-based representation. On this basis, a structure table is created that includes the subgraph leading to a selected node, the possible subsequent nodes, and the confidence for each combination. When a node is selected by the user, a similarity of the corresponding subgraph is calculated for all subgraphs in the structure table. Potential candidates can then be prioritized by confidence. Due to the promising results, this approach [19] is used as a basis to derive the proposed programming assistant. Regarding commercial tools, the generation of suggestions of elements to be used next has been established since long for textual languages, e.g. Visual Studio's *ReSharper* [20]. For VPL, however, commercial programming assistants to support the programmer are rare. Low code platforms such as Siemens *Mendix* [21] allow the simple programming of applications for different fields, but no assistance, e.g., based on data analysis of existing projects is provided.

In summary, there are different concepts to support programmers during writing the code by providing suggestions. However, to the best of the authors' knowledge, there is up to now no approach for low-code platforms that combines assistance to automatically quantify complexity, identify reuse potentials, and provide live recommendations during programming. Thus, this paper enlarges the previously introduced programming assistant by the authors [1] by providing additional details on the implementation as well as the practical usage of the assistance in a user study.

3 Concept of a Programming to Develop Low-Code

To allow the user to objectively quantify program complexity during programming, find and replace code duplicates within a project, and generate suggestions for blocks to be used next, a general concept for a programming assistant is proposed that can be tailored to any VPL that is representable as nodes connected by edges. The concept involves a two-step approach (see Fig. 1): Before the creation of a new project, i.e., *pre-coding*, knowledge is extracted from previous projects to enable different types of assistance functions for programmers *during coding*.

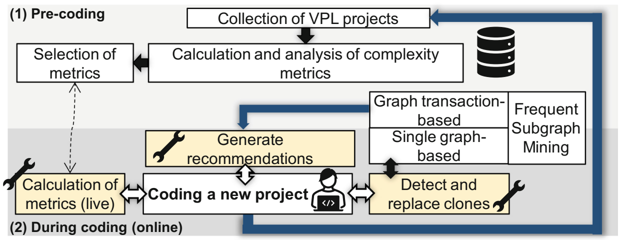

(1) Pre-coding
Collection of VPL projects
Selection of metrics
Calculation and analysis of complexity metrics
Graph transaction-based
Single graph-based
Frequent Subgraph Mining
Generate recommendations
Calculation of metrics (live)
Coding a new project
Detect and replace clones
(2) During coding (online)

Fig. 1. Overview of the two parts to of the low-code programming assistant (adopted from [1])

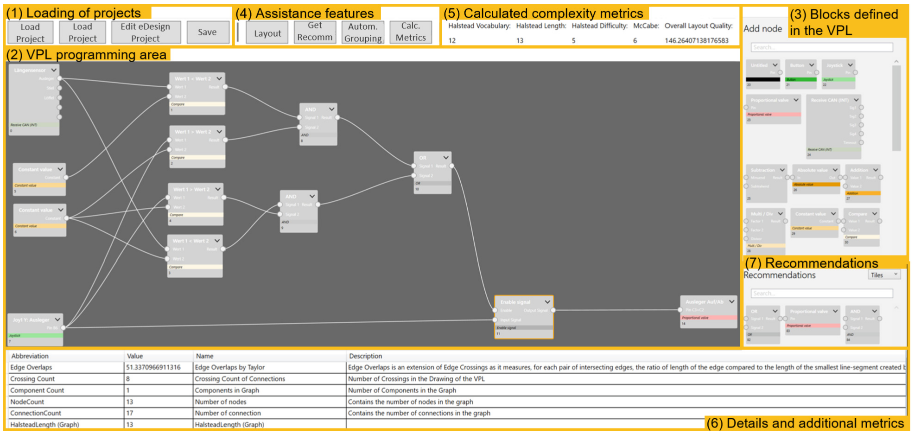
In the *pre-coding*, a data basis of available projects is required that initially need to be transformed to a graph-based representation. This is the basis to select complexity metrics that shall be displayed for the user when developing code. Additionally, *frequent subgraph mining* is performed on the collected projects to generate a basis to derive suggestions during programming. More precisely, a *graph-transaction based* approach is followed. *During Coding,* the user is supported live during programming in a low-code environment based on the data set established in the previous phase: Complexity metrics are calculated for the whole project and updated whenever the project is changed. Additionally, duplicated code parts are identified that can be encapsulated as reusable blocks. Finally, suggestions for blocks to use next are proposed as soon as the developer clicks on an existing block in a given project. For details on the applied code analysis and data mining methods, please refer to [1].

4 Prototype of the Programming Assistant

The following section illustrates the implementation of the programming assistant by means of a prototype on the example of the low-code platform HAWE eDesign. eDesign aims to facilitate the programming of hydraulic components by providing different function blocks connected via ports (cf. Fig. 2).

As a basis for selecting complexity metrics and the suggestion of blocks, a data basis of 1,269 anonymized eDesign user projects has been analyzed during the Pre-Coding phase (cf. Fig. 1). For the pre-analysis of the projects, the metric values are determined with a *Python Jupyter Notebook* [22] based on different sources. In addition to the graphical representation of eDesign, there is a textual intermediate representation of the programs in the high-level C language, which allows the application of analysis techniques for textual languages. In this case, *multimetric* [23] is used to calculate complexity metrics for the C code. Additionally, metrics for the graphical representation are complemented based on the Python library *Network X* [24] and by own implementation of the metrics of Taylor et al. [4] and further metrics in C#. Based on a statistical pre-analysis of the projects, it is concluded that the three Halstead metrics *Vocabulary, Length,* and *Difficulty* as well as McCabe's *Cyclomatic Complexity* and the *Overall Layout Quality* according to Taylor reveal the most significant insights into the projects' complexity and, thus, will be included in the programming assistant.

The programming assistant is developed in C# and allows programming new projects in a low-code environment similar to HAWE eDesign. To evaluate the proposed concept, the different sub-concepts of the programming assistant must be usable during the programming of new projects. This requires an additional connection to the associated web application. To calculate the metrics based on the graphical representation of the program, projects from HAWE eDesign can be loaded into the prototype. During the import, the previously selected metrics are automatically calculated. Thus, all calculated metrics for a project can be bundled by the prototype and exported in a single file (cf. *Area 5* in Fig. 2).



Halstead Vocabulary:	Halstead Length:	Halstead Difficulty:	McCabe:	Overall Layout Quality:
12	13	5	6	146.26407138176583

Abbreviation	Value	Name	Description
Edge Overlaps	51.3370966911316	Edge Overlaps by Taylor	Edge Overlaps is an extension of Edge Crossings as it measures, for each pair of intersecting edges, the ratio of length of the edge compared to the length of the smallest line-segment created b
Crossing Count	8	Crossing Count of Connections	Number of Crossings in the Drawing of the VPL
Component Count	1	Components in Graph	Number of Components in the Graph
NodeCount	13	Number of nodes	Contains the number of nodes in the graph
ConnectionCount	17	Number of connection	Contains the number of connections in the graph
HalsteadLength (Graph)	13	HalsteadLength (Graph)	

Fig. 2. Prototypical implementation of the programming assistant for the example of the low-code environment HAWE eDesign (areas 1–7 highlighted in yellow; adopted from [1]) (Color figure online)

Area 1 allows loading and creating new projects. Once a new project has been loaded or created, programming can be done in *Area 2*. In *Area 4*, various functions can be started manually, such as the calculation of metrics or the automatic grouping. The metrics in *Area 5* are automatically updated once the program structure is changed. In *Area 6*, other metrics can be displayed, such as the individual values of the overall quality of the design. *Area 3* shows all possible blocks that have been implemented in the prototype and can therefore be used for programming. To use them, simply drag and drop them into the programming area. Below, in *Area 7*, the candidates of the proposed assistant are shown. These are calculated as soon as a block is selected. In contrast to the original VPL used in HAWE eDesign, groups encapsulating several blocks can be created as a starting point for standardization in this implementation. The groups thus created can continue to be used in the program in the same way as existing blocks.

5 Evaluation in User Study

The prototypically implemented programming assistant was evaluated in a user study with ten students with a background comparable to that of the focused dedicated specialists (pronounced technical process knowledge but little programming skills). The benefits of the programming assistant were assessed in four programming tasks and a subsequent survey. The participants were divided into two groups – *Group 1* worked on the programming tasks with assistance, *Group 2* without assistance. Figure 3 shows an example for the sample solution of a task to reuse a certain functionality (in this case: limitation based on logical operators) in different parts of a given project. The possibility to encapsulate and reuse functionality with the assistance activated in *Group 1* leads to a significantly reduced complexity compared to *Group 2*.

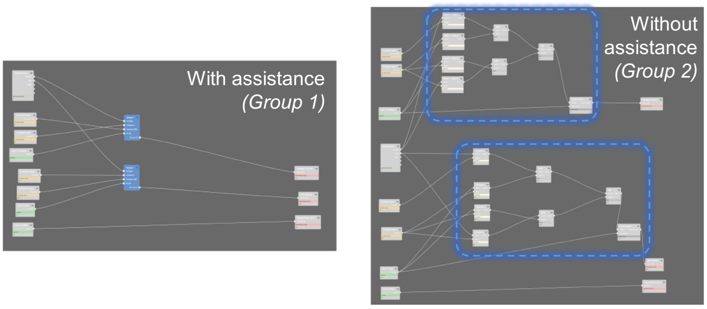


Fig. 3. User support in the programming assistant to reduce complexity by encapsulating reusable functionality in *Group 1* (blue blocks left) compared to reuse without encapsulation via *Copy & Paste* in *Group 2* (blue dotted areas right). Reusable functionality was automatically identified in *Group 1* via clone detection.

The user study confirmed that the use of the programming assistant led to considerable time savings of approx. 54% on average based on time measurements in both groups for the completion of each of the given tasks. Additionally, the features of the programming assistant were perceived as helpful overall. In the future, however, further analyses are required with industrial practitioners using eDesign in their daily development work to program hydraulic components.

The findings of the user study were confirmed in an industry workshop with three developers from HAWE eDesign to evaluate the concept from the perspective of low code platform developers. The workshop confirmed that from the point of view of the interviewed industry experts, the programming assistant is considered helpful and applicable for their customers.

6 Conclusion and Outlook

This paper presents a concept for a programming assistant to support dedicated specialists with sophisticated process knowledge but little programming experience in developing software in low-code platforms by metric-based complexity assessment, encapsulation of code duplicates, and suggestions for blocks to be used live during programming, thus reducing the scaling-up problem in VPL. Using the example of the low code platform HAWE eDesign, the applicability and advantages of the assistance have been successfully evaluated in a study with users emulating dedicated specialists and an additional workshop with industry experts.

Current research in the context of Industry 5.0 highlights the importance of supporting humans with innovative approaches from automation to cope with shortened innovation cycles and the increasing system complexity, especially regarding the increasing scope of functionality implemented via software. Since knowledge of the technical process is becoming increasingly important in software development for mechatronic systems, software will increasingly be written by dedicated specialists without in-depth programming knowledge. Thus, the relevance of low code platforms is expected to increase in the next year. Therefore, future work is required to adopt the implementation of the proposed concept of a programming assistant for further VPL and low-code platforms by considering users with different background and degree of qualification.

Acknowledgment. This work was supported by the Bavarian Ministry of Economic Affairs, Energy and Technology via the project *AIValve* (Grant No. DIK0116/01).

References

1. Neumann, E.-M., Vogel-Heuser, B., Haben, F., et al.: Introduction of an assistance system to support domain experts in programming low-code to leverage industry 5.0. IEEE RA-L **7**, 10422–10429 (2022)
2. Nachtigall, M., Nguyen Quang Do, L., Bodden, E.: Explaining static analysis - a perspective. In: IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), pp. 29–32. IEEE (2019)
3. MathWorks Model Metrics. https://de.mathworks.com/help/slcheck/model-metrics.html. Accessed 14 June 2023
4. Taylor, M., Rodgers, P.: Applying graphical design techniques to graph visualisation. In: 9th International Conference on Information Visualisation, pp. 651–656. IEEE (2005)
5. Capitán, L., Vogel-Heuser, B.: Metrics for software quality in automated production systems as an indicator for technical debt. In: IEEE CASE, pp. 709–716. IEEE (2017)
6. Halstead, M.H.: Elements of Software Science. Elsevier Computer Science Library. Operating and Programming Systems Series, vol. 2. Elsevier, New York and Oxford (1977)
7. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**, 308–320 (1976)
8. Henry, S., Kafura, D.: Software structure metrics based on information flow. IEEE Trans. Softw. Eng. **SE-7**, 510–518 (1981)
9. Sarkar, S., Rama, G., Kak, A.: API-based and information-theoretic metrics for measuring the quality of software modularization. IEEE Trans. Softw. Eng. **33**, 14–32 (2007)
10. Fischer, J., Vogel-Heuser, B., Schneider, H., et al.: Measuring the overall complexity of graphical and textual IEC 61131-3 control software. IEEE RA-L **3**, 5784–5791 (2021)

11. Ain, Q.U., Butt, W.H., Anwar, M.W., et al.: A systematic review on code clone detection. IEEE Access **7**, 86121–86144 (2019)
12. Juergens, E., Deissenboeck, F., Hummel, B., et al.: Do code clones matter? In: IEEE ICSE, pp. 485–495. IEEE (2009)
13. Rosiak, K., Schlie, A., Linsbauer, L., et al.: Custom-tailored clone detection for IEC 61131-3 programming languages. JSS **182**, 1–18 (2021)
14. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. 213. ACM Press, New York (2009)
15. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: O'Boyle, M., Pingali, K. (eds.) Proceedings of 35th ACM SIGPLAN PLDI, pp. 419–428. ACM, New York (2014)
16. Svyatkovskiy, A., Zhao, Y., Fu, S., et al.: Pythia: AI-assisted code completion system. In: Teredesai, A., Kumar, V., Li, Y., et al. (eds.) 25th ACM SIGKDD KDD, pp. 2727–2735. ACM, New York (2019)
17. Kalyon, M.S., Akgul, Y.S.: A two phase smart code editor. In: IEEE HORA, pp. 1–4. IEEE (2021)
18. Stephan, M.: Towards a cognizant virtual software modeling assistant using model clones. In: IEEE/ACM 41st ICSE-NIER, pp. 21–24. IEEE (2019)
19. Deng, S., Wang, D., Li, Y., et al.: A recommendation system to facilitate business process modeling. IEEE Trans. Cybern. **47**, 1380–1394 (2017)
20. Koch, M.: Inspections and Quick-Fixes in ReSharper (2021). https://www.jetbrains.com/dotnet/guide/tutorials/resharper-essentials/inspections-quick-fixes/. Accessed 14 June 2023
21. Siemens Mendix. https://www.plm.automation.siemens.com/global/de/products/mendix/. Accessed 14 June 2023
22. Python Jupyter Notebook. https://jupyter.org/. Accessed 14 June 2023
23. Python multimetric. https://pypi.org/project/multimetric/. Accessed 14 June 2023
24. Python NetworkX. https://networkx.org/. Accessed 14 June 2023