Towards a Circular Economy of Industrial Software

Kutscher, Vladimir; Ruland, Sebastian; Müller, Patrick et al.

(2020)

Procedia CIRP 90 (2020) 37–42

Contents lists available at ScienceDirect

Procedia CIRP

journal homepage: www.elsevier.com/locate/procir

27th CIRP Life Cycle Engineering (LCE) Conference

Towards a Circular Economy of Industrial Software

Vladimir Kutscher[a,*], Sebastian Ruland[b], Patrick Müller[c], Nathan Wasser[d], Malte Lochau[b], Reiner Anderl[a], Andy Schürr[b], Mira Mezini[c], Reiner Hähnle[d]

[a] Department of Computer Integrated Design, TU Darmstadt, Otto-Berndt-Str. 2, Darmstadt 64287, Germany
[b] Real-Time Systems Lab, TU Darmstadt, Magdalenenstr. 4, 64289 Darmstadt, Germany
[c] Software Technology Group, TU Darmstadt, Hochschulstr. 10, Darmstadt 64287, Germany
[d] Software Engineering Group, TU Darmstadt, Hochschulstr. 10, Darmstadt 64287, Germany

ARTICLE INFO

Article history:
Received 30 June 2019
Revised 17 January 2020
Accepted 28 January 2020

Keywords:
Software reengineering
Industrial software
Digital twin
Industry 4.0
Legacy system
Computerized numerical control
Circular economy

ABSTRACT

Software has become an indispensable part of industrial production and thus influences the life cycle of manufacturing systems, as many of these systems have to be replaced or evolved due to changing requirements. Software adaptation through continuous evolution extends the service time of these systems and thus saves valuable resources. In this paper we present an interdisciplinary methodology for reengineering legacy software to increase the productive lifetime. Our proposed approach systematically reuses implicit domain knowledge to evolve, validate and commission new software from legacy code. The approach is evaluated on a CNC machine as a special type of industrial system.

1. Introduction

Software has an end of life. Among others, this aging process is a result of transformations of the environment (Grottke et al., 2008; Parnas, 1994). In industry this change is induced by new technologies, business models and changing requirements (i.e. Industrie 4.0) (Anderl, 2015).

However, since existing systems often do not meet the requirements, an acquisition of new systems or an adaptation of the software is usually necessary. New acquisition causes financial expenses in addition to ecological costs, as manufacturing new systems requires considerable resources. Furthermore, new systems must be integrated into production. This can lead to compatibility problems and a long production downtime. An alternative is reengineering existing software, which enables recycling of systems but causes major challenges. First, existing software must be analyzed efficiently and knowledge only implicitly present in the source code must be obtained. Second, the modified software needs to be functionally verified and validated. Finally, it must be configured and commissioned on the production system.

In order to meet the introduced challenges and utilize the full potential for continuous improvement of existing software, we present an interdisciplinary concept. Fig. 1 shows legacy software of a laser plotter (a) recycled not only within an evolved system (b), but also used for other systems with comparable functionality, e.g., a milling machine (c).

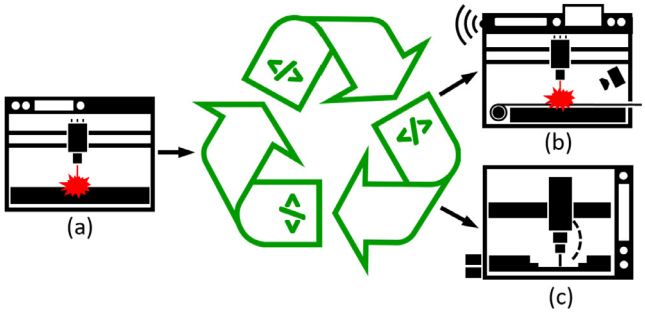

Fig. 1. Recycling of legacy software (a) in evolved (b) and similar systems (c)

Paper structure First, a running example is presented. Subsequently, the required multidisciplinary knowledge is covered in Sect. 3. Based on this, Sect. 4 presents the concept, which is validated in a use case (Sect. 5). Finally, the conclusion and outlook on future work is given in Sect. 6.

* Corresponding author. Tel.: +49 6151 16-21868; fax: +49 6151 16-21793.
E-mail address: kutscher@dik.tu-darmstadt.de (V. Kutscher).

https://doi.org/10.1016/j.procir.2020.01.133

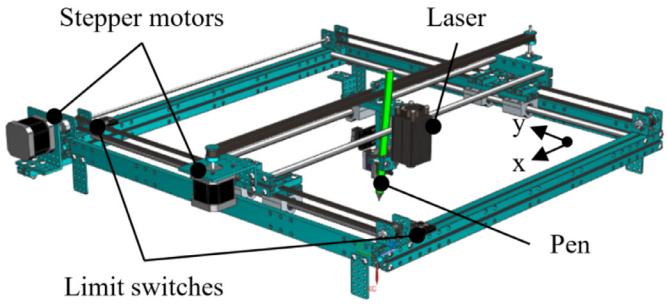

Fig. 2. CNC laser plotter

2. Running Example

Figure 2 depicts a computerized numerical control (CNC) machine which can cut or plot paper. This system is used as a running example to motivate and illustrate the approach presented in this paper. The mechanics of the CNC machine consist of an x- and y-axis which are controlled by stepper motors. A laser- or pen-holder can be moved along both axes. Limit switches detect the limitations of the working area. The machine is operated by numerical control (NC) software which interprets *G-code*, the machine commands standardized in DIN ISO 2806 (1996). Exemplary G-codes are: `H` which triggers the homing procedure and `G0 X-100 Y-50 F1000` which defines movement mode (`G0`), a target location (`X,Y`) and a feed rate (`F`).

An exemplary source code (C language) for a simplified NC software is given in Fig. 3. Using the input of the introduced G-code instructions (`G` or `H`), the corresponding source code is designed as follows. The NC control interprets incoming machine commands in the function `parse_command`. The transmitted commands are then processed in different `{in}`-statements. If the input starts with `G`, the configured engineering unit is set (`unit`) and the remaining instruction is parsed and stored in the value `axis_command`. The extracted coordinates are then processed (`move_x, move_y`). The `Z` coordinate is only handled if it was previously activated (`move_z` is inside an `#ifdef` macro). For example, the Z-axis is deactivated in the laser mode of the presented machine (Fig. 2). If the incoming command line is an `H`, a homing procedure (`home_axes`) is activated and the axes move to their home position defined by the limit switches. If the input starts with neither `G` nor `H`, a failure (`UNSUPPORTED_COMMAND`) will be triggered in the `else`-branch. Subsequently, further interpretation of the transmitted command takes place. The used NC software is multi-functional and can be deployed on various manufacturing systems. This legacy software can therefore serve as the basis for different software products.

3. Background

The reengineering of a system is characterized by interdisciplinarity, which is an essential aspect of this project. In this section the state of the art for each individual discipline involved is explained independently, before we unify them in Sect. 4. We start by describing Industrie 4.0, which motivates the necessary software adaptation of industrial systems. The virtual commissioning and the aspect of the digital twin as an important part of reengineering is described next. Subsequently, the methodology of knowledge extraction is described, with which the existing software is analyzed and extended. Lastly, we will describe quality assurance measures to validate software changes.

3.1. Industrie 4.0

Industrie 4.0, the fourth industrial revolution currently taking place, is transforming the industrial world (Anderl, 2015) based on intelligent *cyber-physical systems* (CPS) that enable new applications and business models (Kagermann et al., 2013). Since many machines, especially older production systems, do not meet CPS requirements, these systems must be replaced or upgraded before they can participate in the Industrie 4.0 world.

Virtual Commissioning A modified system needs to be put back into operation. Configuration and debugging of the software occupies a large part of the commissioning (Reinhart and Wünsch, 2007). Using our running example (Sect. 2), the behavior of the CNC plotter must be validated if the NC software is modified. In order to reduce the load on the physical machine, methods and tools have already been developed which are summarized under the term *virtual commissioning* (Lee and Park, 2014). However, the focus of virtual commissioning is usually on automation software according to IEC 61131 IEC 61131-1 (2014) which is executed on programmable logic controllers (PLC). Holistic virtualization of simple systems has already been performed (Beghi et al., 2017), whereby the simulation of machine software that is not part of the PLC is still limited.

Digital Twin In order to virtually test and commission software of a machine, a *digital twin* (Anderl et al., 2018) (a detailed virtual representation of the machine) is necessary. Since the first description of a digital twin (Grieves and Vickers, 2017) the possible use-cases in different domains were investigated (Kritzinger et al., 2018; Negri et al., 2017). A holistic, executable model is complex, but the combination of different domains enables novel applications (Wagner et al., 2019). In our use case the digital twin is utilized to virtually commission software which is not part of the control according to IEC 61131.

3.2. Knowledge Extraction

Our running example represents a legacy system whose software has to be recycled. As legacy software often has limited documentation, most information is implicitly inside the source code. Therefore, techniques to extract this knowledge and make it explicit are highly relevant for evolving legacy software, enabling developers to easily integrate new functionality. The two most relevant pieces of information in the software are software modules and module interaction as explained in the following. *Software Modules* Software can be split into multiple modules, each containing functionality highly relevant to other functionality inside the same module. In our running example one module is the *axis-control* executed by calling the functions `move_x`, `move_y` or `move_z`. While software modules can be distributed over many parts of the source code, the closer they are the easier it is to integrate new modules or change specific functionality. To further facilitate reuse and reduce maintenance efforts, software can be engineered as a Software Product Line (SPL) (Clements and Northrop, 2002; Weiss and Tau, 1999). This means that modules can be (de)selected by configuration, making it possible to create several different software products from a single code base (e.g., implementing separate control modules for pen and laser, leading to source code that can control a CNC plotter with either). A single configuration specifying the (de)selected modules is called a product.

Module Interaction and Software Product Line Models Software modules often interact with each other. This leads to dependencies and even incompatibilities between modules. To specify this kind of interaction for SPLs, a so called feature model is often used. Fig. 4 shows a small feature model matching our running example. There is an optional feature `Z\_ENABLED` and another optional

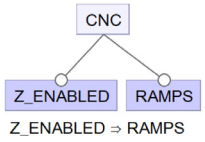

Fig. 4. SPL Feature Model

feature called RAMPS, which when activated signifies the presence of a specific hardware variant. In addition there is a cross-tree constraint, which symbolizes that the CNC machine cannot operate with a z-axis without the RAMPS hardware (and enabling the corresponding feature). This example shows that a feature model can convey information about the source code, as well as the surrounding build environment.

3.3. Quality Assurance

Quality assurance (QA) is imperative during the creation and modification of source code. QA has many different goals during development. However, in this paper we focus on backwards compatibility and correctness of changes. For both goals there are two main techniques to reach them, namely *testing* and *verification*. In the case of our running example, QA is intended to ensure that the NC software can continue to run on the original machine and that the changes are error-free. *Testing* Testing tries to cover a large set of possible program runs by executing the program with specific input (and optional output) values. A test is successful if the program executes without error and provides the correct output. Different *coverage criteria* exist for testing, specifying which parts of the program need to be covered. For industrial software the chosen criterion is often *branch-coverage*, which is often mandatory for safety-relevant source code (e.g., RTCA/DO-178B standard). This means that for each branch in the software there exists at least one test traversing the branch during execution

(e.g., for our running example 3 test cases are needed with inputs starting with G, H and an arbitrary character such as Z). Since this is usually infeasible for non-trivial software, the coverage criterion is often accompanied by a coverage-value specifying the percentage of branches the test cases have to cover. *Verification* Verification is often used in addition to testing and tries to prove certain program behaviors or properties. While quite powerful, verification tends to be much slower and more costly than testing, often requiring manual inputs from highly trained users; for this reason verification is often most useful only for smaller parts of the program which have already been established to be hotspots. Certain types of verification also allow automating much of the process, e.g., LLRêve (Felsing et al., 2015) uses symbolic execution to automate *regression verification*, a technique to prove equivalence of two program fragments (potentially with constraints on the inputs).

4. Industrial Software Recycling

We now present our interdisciplinary methodology for reengineering legacy software. The focus is on evolution of software by recycling implicit knowledge contained in source code of legacy systems. This way functionality and therefore sustainability of production machines can be increased. The concept consists of the reengineering process in Fig. 5, with source code of an existing machine as initial input. This code is then ① analyzed and broken down into software modules that are ② linked by a feature model. The resulting modules are then ③ validated and verified and subsequently ④ virtually commissioned using a simulation-based digital twin. After virtual commissioning, the evolved software can be ⑤ used in production or ⑥ adapted to additional requirements. Additional adaptions require QA and virtual commissioning as well, to ensure validity of the changes made. Obviously, after using the evolved software in production it can be further adapted to meet upcoming requirements. In the following we describe the steps and methodologies needed to perform steps ① - ⑥. Retrofitting by upgrading the hardware and electronics and connecting new technologies to existing systems is out of scope in this paper but considered in other works, e.g., (Guerreiro et al., 2018; Lins et al., 2018; Moctezuma et al., 2012).

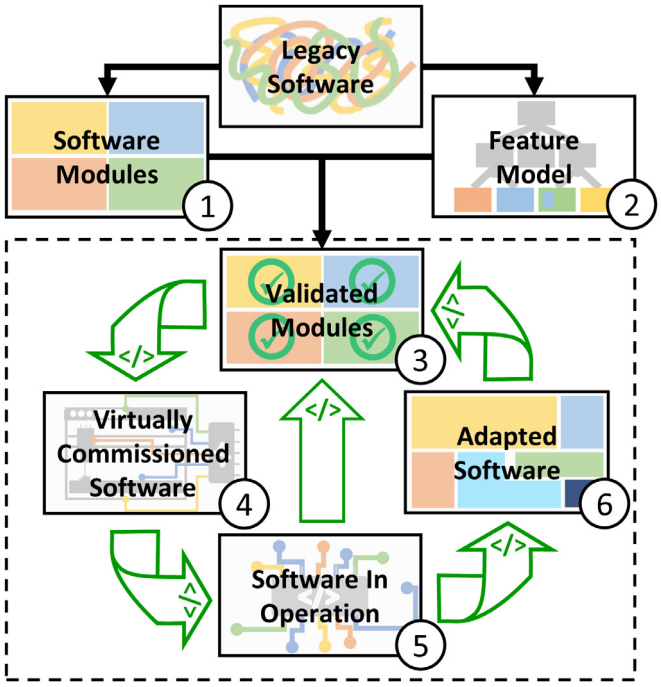

Fig. 5. Software Reengineering Process

```
1  void parse_command(char* input) {
2    axis_command = NULL;
3    if (input[0] == 'G') {
4      axis_command = parse_axis_command(input);
5      move_x(axis_command);
6      move_y(axis_command);
7      #ifdef Z_ENABLED
8        move_z(axis_command);
9      #endif
10   } else if (input[0] == 'H') {
11   } else { FAIL(UNSUPPORTED_COMMAND); }
12   if (axis_command) {...}
13 }
```

Fig. 6. Sliced version of Fig. 3

① *Module Extraction.* We first extract modules that are present in the source code but are not clearly separated from code that has other functionalities. The modules are extracted via slicing. *Program slicing* (Weiser, 1981) in general is the reduction of a program to the parts that are relevant for the value of a given variable at a given location. This location and the variables we are concerned with is called *slicing criterion* (or criteria). Then the given program is reduced to the parts that may have an effect on the slicing criterion. As an example consider Fig. 3: choosing axis_command in line 14 as our slicing criterion, the resulting slice consists of the code in Figure 6. All code that has no effect on the value axis_command is removed, but all instructions that can have an effect are retained. Note that control flow constructs

```
1  void parse_command(char* input) {
2    axis_command = NULL;
3    if (input[0] == 'G') {
4      unit = parse_unit(input); // mm or inches
5      axis_command = parse_axis_command(input);
6      move_x(axis_command);
7      move_y(axis_command);
8      #ifdef Z_ENABLED
9        move_z(axis_command);
10     #endif
11   } else if (input[0] == 'H'){
12     home_axes();
13   } else { FAIL(UNSUPPORTED_COMMAND)  }
14   if (axis_command) {...}
15 }
```

Fig. 3. Exemplary CNC machine source code

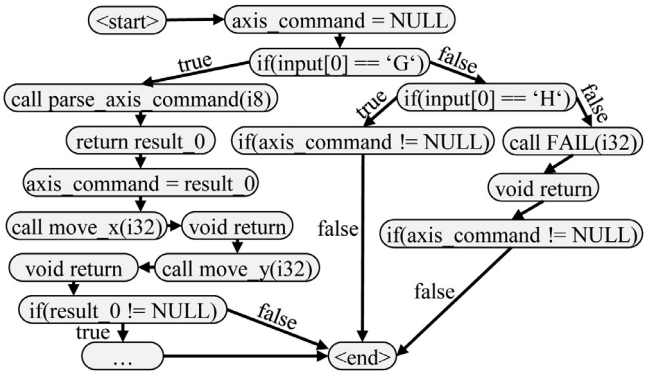

Fig. 7. Visualization of executing the sliced function `parse_command`

(such as the if-else-statements and the `FAIL` function call) are retained as well. In order to find relevant slicing criteria, we utilize information retrieval techniques (Marcus et al., 2004; Shao and Smith, 2009) to decide on program locations to slice. This means we pre-process the source code — including comments — to create a corpus that maps terms to locations. We then make queries to this corpus, resulting in locations that correspond to the given search term, which is supplied by a domain expert. We use the locations retrieved from the corpus as slicing criteria for a slicing approach crafted for the extraction of modules. *Program visualization*, e.g., using CSED, a modification of the *symbolic execution debugger* (SED) (Hentschel et al., 2019) we developed for C code, can also be used to help understand the source code, find features and locate hotspots for regression analysis. Fig. 7 shows a visualization of executing the sliced function in Fig. 6. Nodes (other than start and end) correspond to program statements with outgoing edges to the next statement to be executed. If-statements usually have two labeled outgoing edges, as the next statement depends on the evaluation of the condition. However, unreachable nodes and their attached edges are removed, resulting in the two if-nodes with only one edge (labeled `false`). This analysis can lead to dead code elimination and program simplification.

② *Reverse Engineering of Feature Models.* In order to retrieve the feature model that is already implicitly present in the legacy source code, we need to analyse both the build system and the source code itself. Using configuration options from both sources we automatically create an initial feature model. However, this inital feature model may contain configurations that will not produce correct code (as incompatible modules might be selected). We therefore prune it in several ways. We first evaluate the preprocessor for any errors. In the second step, we check the source code in all possible configurations for possible syntax errors, using a special variability aware parser. Building on that parser, we employ a variability aware type checker. Finally, possible linker errors are evaluated. Using information from all these checks we remove invalid configurations from the intermediate feature model (Nadi et al., 2014). Up to this point all tasks are fully automated and result in a feature model which is subject to optional improvements, some of which can be manual. By obtaining both feature model and extracted modules, a fully fledged SPL has been extracted from the legacy software.

③ *Quality Assurance.* After extracting and reorganizing the software modules of the legacy software (or after enhancing its functionality), QA is the next step to validate correctness of the changes. There are dependencies and incompatibilities between modules, as described by the extracted feature model. Therefore, testing the modules based on that information is necessary. To this end modules need to be (de)selected to build a single product for a test run. However, testing individual products is usually infeasible because of the sheer number of products. For this reason we utilize tools that have already been developed to generate test-cases on SPLs (Bürdek et al., 2015). During transformation of software (e.g., module extraction or adaptation), faults might be introduced. These faults can be new (e.g., incorrectly implemented new functionality) or reemerging faults (e.g., faults that have been fixed but reappear due to copy&paste or incorrect merges during version control) (Simpson, 2010). To generate test-cases revealing differences in program versions, a methodology called differential testing can be utilized (McKeeman, 1998). Differential testing generates test-cases that guarantee different output values if executed on the different versions used to generate the differential test. In case of transforming legacy software to an SPL no behavior of the program should be changed. This can either be proven by utilizing regression verification to prove functional equivalence or using differential testing to validate that no faults have been introduced, if no differential test could be generated. In case of behavior modifying changes (e.g., new functionality), differential testing can also be used, however differential testing is often restricted to generating one test-case revealing a difference. This is also due to the fact that generation of differential test-cases is often very time-consuming. Another strategy is to define a specific coverage criteria for the program under test. For example, NC software often contains specific error messages, resulting from non-valid inputs (e.g., for G-code in our running example inputs with no leading `G` or `H` result in an error). Generating test-cases to reach all error messages can be used to check the error handling of following versions, which is especially important for safety reasons.

④ *Commissioning using Digital Twins.* After QA the software is principally ready for operation. However, commissioning on a machine is very time-consuming and involves risks, as errors not found during testing and therefore still present in the software now also affect the physical world. In addition, the productive machine may not be available for extensive downtime while commissioning. An alternative way is to use a digital twin to virtually commission the software. This enables a assessment of the software behavior, which meets the requirements of industrial software in terms of correctness, reliability and security. The application of a digital twin of a system to investigate the effects of the control software requires combination of the electronics and the mechanics of a system. For this purpose, a simulation chain is used which includes the simulation of a micro controller (Kutscher et al., 2019) and a multi-body simulation of the system. The modified and tested software is installed in the micro controller simulation environment and coupled with a multi-body simulation (Anderl and Binde, 2018). The control signals coming from the micro controller simulation form the boundary conditions for the multi-body simulation and therefore enable mapping of the effects of a control

```
void parse_command(char* input) {
  axis_command = NULL;
  if (input[0] == 'G') {
    axis_command = parse_axis_command(input);
    unit = parse_unit(input); // mm or inches
    move_axis(X_AXIS,axis_command);
    move_axis(Y_AXIS,axis_command);
    #ifdef Z_ENABLED
      move_axis(Z_AXIS,axis_command);
    #endif
    #ifdef A_ENABLED
      move_axis(A_AXIS,axis_command)
    #endif
  } else (input[0] == 'H'){
    home_axes();
  } else { FAIL(UNSUPPORTED_COMMAND) }
  if (axis_command) {...}
}
```

Fig. 8. Evolved version of Fig. 3

software on the physical behavior of a system. This combination can now be used to visually test the behavior of the software with the mechanics of a machine and to configure the software. Once the software has been tested and configured using the digital twin, it can be installed onto the physical twin, which can be the original, the modified or another system that recycles functionality of the initial system.

⑤ *Software in Operation.* After QA and virtual commissioning of the software, it can either be directly used in production or further adapted. Obviously, after using the software in production for some time, it can also be further adapted later.

⑥ *Adaptation.* The extracted modules and the feature model then enable developers to create and extend the source code more easily. As an example consider the code in Figure 8. The code from our running example has been extended to enable an `A` axis in addition to the `X,Y,Z` axes using a generic method `move_axis` that belongs to an extracted axis module instead of axis specific ones (`move_x,move_y,move_z`). Using this additional `A_AXIS` is entirely optional and becomes an additional new feature in the model. Visualization of the original and adapted code with CSED allows easy comparison and helps localize hotspots for regression verification in the next QA cycle.

During each of these steps it is possible to restart our approach if feature extraction should be improved (e.g., new search terms have been identified). Software that already passed steps ① and ② is reused as input upon restarting.

5. Evaluation

Our reengineering approach allows to evolve legacy software by recycling implicit knowledge already contained within the system. However, to use this methodology in practice there are two key aspects that need to be answered. The first research question is if the approach is *applicable* to real systems, while the second is if the approach *scales*. As proof of concept, we demonstrated our approach on a concrete application case, which is presented below. The use case is based on the running example introduced in Sect. 2. The software and hardware of the machine are extended by a new function, which enables automated paper conveying, while the paper had to be inserted manually in the initial system. The use case is shown in Fig. 9.

The initial situation is a legacy system with legacy software and hardware. As described in Sect. 4, the software is analyzed and broken down into modules. Afterwards, the changes are validated.

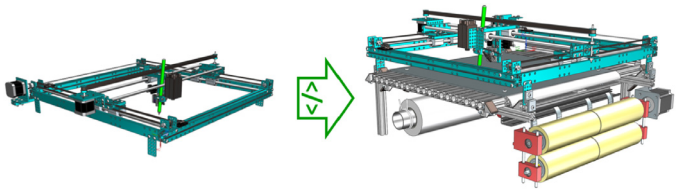

Fig. 9. Enhancement of a CNC plotter by a paper conveyor

Subsequently, functions for controlling existing axes are reused for adding a degree of motion to drive the paper feed. Modification and validation are carried out iteratively until the desired function has been implemented without errors. The modified and validated software is configured and put into operation using a digital twin of the modified physical twin. Once this step has been successfully completed, the finished software is loaded onto the physical machine. The result is a machine with extended software by recycling the implicit knowledge of the legacy system. During these steps we noticed that there still exists a gap in requirements and availability of fully automated tool support for the whole process. During feature extraction there are still tasks which have to be manually executed, while for testing and generation of digital twins the system needs to be abstracted as the full system is often hard to handle. Nonetheless, we managed to reengineer the legacy system and evolved it to handle an additional axis for paper conveying. Therefore, we managed to demonstrate the applicability of our approach to real systems. To achieve scalability it will be necessary to align our approach with general modularization principles in Software Design. These are increasingly used in industrial software as witnessed, for example, by the AUTOSAR[1] standard. Theoretically one can process large software systems using the same input, e.g. extract all modules for given search terms from the whole source code of an automobile. The question whether there are search terms that are relevant for a whole system remains to be answered.

6. Conclusion and Future Work

The presented concept shows an interdisciplinary and methodical approach, which enables the extension of existing production machines with new functions by using the implicit knowledge contained in the existing software. With the help of this software reengineering process, the efficient recycling of software becomes an attractive alternative to implementing new software from scratch. As future work, we plan to extend our approach in multiple directions. First, we want to apply our approach to different domains, such as additive manufacturing and milling machines. Second, we want to extend our QA approach by means of non-functional QA, to validate that non-functional properties (e.g., energy consumption) do not deteriorate during reengineering. Additionally, we want to fully integrate testing into the virtual commissioning to further automize the validation step. Lastly, we want to further incorporate tools into our approach to further improve automatism and scalability.

CRediT authorship contribution statement

Vladimir Kutscher: Conceptualization, Methodology, Software, Resources, Writing - original draft, Writing - review & editing, Visualization, Project administration, Validation. **Sebastian Ruland:** Conceptualization, Methodology, Software, Resources, Writing - original draft, Writing - review & editing, Visualization, Validation. **Patrick Müller:** Conceptualization, Methodology, Software,

[1] https://www.autosar.org/

Resources, Writing - original draft, Writing - review & editing, Validation. **Nathan Wasser:** Conceptualization, Methodology, Software, Resources, Writing - original draft, Writing - review & editing, Validation. **Malte Lochau:** Conceptualization, Methodology, Writing - original draft. **Reiner Anderl:** Supervision, Project administration, Funding acquisition. **Andy Schürr:** Supervision, Project administration, Funding acquisition. **Mira Mezini:** Supervision, Project administration, Funding acquisition. **Reiner Hähnle:** Supervision, Project administration, Funding acquisition.

Acknowledgements

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

Anderl, R., 2015. Industrie 4.0 – technological approaches, use cases, and implementation. at-Automatisierungstechnik. De Gruyter Oldenbourg.
Anderl, R., Binde, P., 2018. Simulations with NX/Simcenter 3D: Kinematics, FEA, CFD, EM and Data Management, 2nd Hanser.
Anderl, R., Haag, S., Schützer, K., Zancul, E., 2018. Digital twin technology – an approach for Industrie 4.0 vertical and horizontal lifecycle integration. it - Inf. Technol., 60.
Beghi, A., Marcuzzi, F., Martin, P., Tinazzi, F., Zigliotto, M., 2017. Virtual prototyping of embedded control software in mechatronic systems: A case study. J. Mechatron., 43.
Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D., 2015. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. FASE. Springer.
Clements, P., Northrop, L., 2002. Software product lines: practices and patterns, vol. 59. Reading: Addison-Wesley. IEEE.
DIN ISO 2806, 1996. Numerical control of machines.
Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M., 2015. Automating regression verification. ASE. IEEE/ACM.
Grieves, M., Vickers, J., 2017. Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. Transdisciplinary Perspectives on Complex Systems, 89. Springer.
Grottke, M., Matias, R., Trivedi, K.S., 2008. The fundamentals of software aging. ISSRE Workshops. IEEE.
Guerreiro, B.V., Lins, R.G., Sun, J., Schmitt, R., 2018. Definition of smart retrofitting: First steps for a company to deploy aspects of Industry 4.0. Adv. in Manuf.. Springer.
Hentschel, M., Bubel, R., Hähnle, R., 2019. The symbolic execution debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. Int. J. on Softw. Tools for Technology Transfer 21 (5).
IEC 61131-1, 2004. Programmable controllers-part 1: general information.
Kagermann, H., Wahlster, W., Helbig, J., 2013. Recommendations for implementing the strategic initiative Industrie 4.0.
Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W., 2018. Digital twin in manufacturing: A categorical literature review and classification. IFAC, 51.
Kutscher, V., Anokhin, O., Anderl, R., 2019. Enhancing digital twin performance through simulation of computerized numerical control firmware. To appear in: Procedia Manuf..
Lee, C.G., Park, S.C., 2014. Survey on the virtual commissioning of manufacturing systems. JCDE, 1.
Lins, T., Augusto Rabelo Oliveira, R., Correia, L.H.A., Sa Silva, J., 2018. Industry 4.0 retrofitting. SBESC. IEEE.
Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I., 2004. An information retrieval approach to concept location in source code. WCRE.
McKeeman, W. M., 1998. Differential testing for software 10
Moctezuma, L.E.G., Jokinen, J., Postelnicu, C., Lastra, J.L.M., 2012. Retrofitting a factory automation system to address market needs and societal changes. INDIN. IEEE.
Nadi, S., Berger, T., Kästner, C., Czarnecki, K., 2014. Mining configuration constraints: Static analyses and empirical results. ICSE. ACM.
Negri, E., Fumagalli, L., Macchi, M., 2017. A review of the roles of digital twin in cps-based production systems. Procedia Manuf.. Elsevier.
Parnas, D.L., 1994. Software aging. ICSE. IEEE.
Reinhart, G., Wünsch, G., 2007. Economic application of virtual commissioning to mechatronic production systems. J. Prod. Eng., 1.
Shao, P., Smith, R.K., 2009. Feature location by IR modules and call graph. ACM-SE. ACM.
Simpson, P.A., 2010. FPGA Design. Springer.
Wagner, R., Schleich, B., Haefner, B., Kuhnle, A., Wartzack, S., Lanza, G., 2019. Challenges and potentials of digital twins and industry 4.0 in product design and production for high performance products. Procedia CIRP.
Weiser, M., 1981. Program Slicing. ICSE. IEEE.
Weiss, D., Tau, C., 1999. Software Product-Line Engineering: A Family-Based Software Development Process, 1. Addison-Wesley.