Towards Improving Software Architecture Degradation Mitigation by Machine Learning

Sebastian Herold
Karlstad University
Karlstad, Sweden
Email: sebastian.herold@kau.se

Christoph Knieke, Mirco Schindler, Andreas Rausch
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
Email: firstname.surname@tu-clausthal.de

Abstract—**Mitigating software architecture degradation is a task in evolving large and complex software-intensive systems that is as important as it is challenging. One aspect adding to the complexity of the task is the amount of information in the implementations of most real-world systems to be digested in order to detect, analyse, and remedy degradation. In domains with similar challenges, machine learning techniques have been applied in recent years and partially delivered exciting results. Hence the question arises whether, and to which degree, machine learning can be successfully applied to tackle software architecture degradation. In this paper, we propose a novel combination of existing techniques for different phases of the task of mitigating software architecture degradation from detecting it to repairing it. We outline how these techniques could be complemented by machine learning to increase their accuracy and efficiency over time.**

Keywords–Software Evolution; Software Architecture Degradation; Machine Learning.

I. Introduction

Mitigating Software Architecture Degradation (SAD) plays an important role for the longevity of evolving software-intensive systems. Today SAD is a big challenge in modern architectures like the architecture of software ecosystems and services and leads to a deteriorates of the quality of such systems. Architecture degrades/erodes when the implemented architecture of a software system diverges from its intended architecture [1]. This usually happens during software evolution when the software undergoes changes as a result of bug fixes and further development, but may also happen during initial implementation of the system. Architecture erosion hinders the further development of systems and leads to less reusability, maintainability, understandability and decrease in performance.

There has been a lot of research on how to mitigate SAD [1]. However, studies show that in practice it is still difficult to remedy SAD [2]. The study in [1] concludes that none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

There are many reasons why the reduction of SAD causes so many difficulties. One is the inherent complexity of the task. Modern software systems are highly complex and have a long lifespan. The system experts have to filter and find the information relevant to SAD in the huge amount of data contained in large (and potentially old) repositories of source code and other relevant artefacts. Current approaches to SAD seem not to scale well with this complexity [1].

In other domains with similar challenges Machine Learning (ML) techniques are already used to support maintenance and evolution tasks (“predictive maintenance”). Generally, ML is taken to encompass automatic computing procedures based on logical or binary operations that learn a task from a series of examples, i.e., ML provides systems the ability to automatically learn and improve from experience without being explicitly programmed to so [3]. ML is divided into three subdomains supervised, unsupervised, and reinforcement learning [4].

Over the past decade, ML techniques have been widely adopted in a number of massive and complex data-intensive fields such as medicine, biology, and astronomy, for these techniques provide possible solutions to mine the information hidden in the data [4]. The question motivating this article is whether and how such techniques can be applied to mitigate SAD more effectively and efficiently. In this paper we look at the state of the art in mitigating SAD and propose a combination of existing techniques and how to add ML to them in order to increase the techniques’ accuracy and efficiency over time.

The paper is structured as follows: Section II gives an overview on the related work. Our idea towards a learning environment for mitigating SAD is presented in Section III. Finally, Section IV concludes.

II. State of the Art in Machine Learning for Dealing with SAD

In order to characterize the state of the art in using ML, or related techniques, to mitigate SAD, we conducted a systematic literature review of 26 eventually relevant papers. This review is currently being finalized, however, a few important characteristics can already be noted.

We were particularly interested in which activities of SAD mitigation are covered by research. In an earlier mapping study on SAD in general, activities were distinguished into detection, analysis, repairing, and prevention of degradation [5]. We added architecture recovery as an important subsequent step that is intertwined with detection in many techniques and categorized the relevant papers according to the five resulting activities (multiple categories per paper were possible). The results show that a majority of papers fall into the activities of recovery (7 papers) and detection (9 papers). These can be considered the “early” phases of SAD mitigation as one has to have the intended architecture and to know the present inconsistencies before actions against SAD can be taken. Seven papers were considered to cover the analysis of degradation and only three look at the usage of ML for repairing degradation. Preventing erosion was the motivation for four papers.

The use of ML for architecture recovery appears quite natural as clustering is one of the commonly used techniques used in this activity and also a main application of many unsupervised learning techniques. Hence, these techniques are used relatively frequently in this context [6] [7] [8].

The usage of such techniques as a part of detecting and analysing SAD is very diverse. It covers very distinct activities like automating the creation of architecture-implementation mappings required in most consistency checking techniques [9], the detection of design defects [10], or the analysis of the use of architectural tactics [11]. To our best knowledge, no approach applies ML to help software engineers understand potential causes of instances of SAD to better mitigate it as proposed in literature [12]. In preventing SAD, the identified studies were mainly about preventing architecture smells which are often considered to be an important factor leading to degradation (e.g., Fontana et al. [13]).

Most of the studies report positive results regarding the performance of the applied ML techniques as measured by precision, accuracy, or recall. It can thus be concluded that the use of ML techniques in the context of SAD seems beneficial even though a potential publication bias in favour of positive results cannot be completely excluded. However, a few studies outline that there is space for improvement. Khomh et al., for example, apply Bayesian Belief Networks for code and design smell detection with comparably low precision [14]. Lenhard et al. describe a study in which they investigate whether "smelly" code can be used as an indicator for architectural inconsistencies [15]. They tried to train a classifier for this task and describe the results as unsatisfying as they show low precision and recall.

We can conclude from these preliminary findings that an holistic approach making use of ML techniques, supporting the software engineering coherently from the detection of degradation to counteracting it, is missing. Moreover, ML is less frequently applied directly to the phenomenon of degradation, i.e., the divergence between intended architecture and implementation of a system but with aspects indirectly connected to it, such as code/design/architecture smells, design defects, architecture tactics, etc. A stronger research focus on the analysis activity in general with the goal of providing insights to degradation causes might also provide a better ground for applying ML techniques in the activities of repairing and preventing degradation.

III. The Envisaged Approach

In this section, we present our idea towards a learning environment for mitigating SAD.

A. Overview

The suggested approach follows conceptually the ideas of repairing architecture degradation presented by Mair et al. and extends them [12]. The authors follow the analogy of medical doctors that, for treating a disease, first assess symptoms to exclude and diagnose possible diseases or conditions to eventually suggesting and executing a suitable therapy. According to that metaphor, we propose an approach consisting of three main activities as depicted in Figure 1: *Architecture Recovery and Consistency Checking (ARC)*, *Degradation Cause Analysis (DCA)*, and *Recommending Repair Actions (RRA)*.

Comparable to assessing medical symptoms, the ARC step aims at assessing the status quo of software architecture degradation in the software system at hand. This means it consists of inspecting its intended software architecture and the implementation of a system and detecting inconsistencies between them. It might also involve recovering (parts of) the

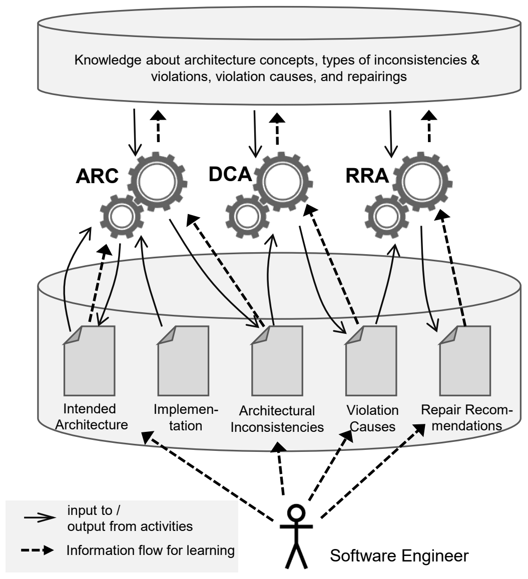
Knowledge about architecture concepts, types of inconsistencies & violations, violation causes, and repairings
ARC
DCA
RRA
Intended Architecture
Implemen-tation
Architectural Inconsistencies
Violation Causes
Repair Recom-mendations
input to / output from activities
Information flow for learning
Software Engineer

Figure 1. Conceptual Overview of the Proposed Approach

intended architecture as its specification might not exist or be outdated. The result of this activity is—beside a potentially updated intended architecture—primarily a set of architectural inconsistencies each of which might either be considered tolerable or to represent an actual, potentially harmful violation of the intended architecture.

Similar to how the presence or absence of several different symptoms might point to a certain medical condition, combinations of several architectural inconsistencies and properties of the implementation fragments connected to them might indicate a deeper problem behind those inconsistencies. The activity DCA is hence about aggregating information about single instances of architectural inconsistencies to form an overall picture of the underlying causes of the degradation which potentially helps to remedy it more efficiently.

Recommending Repair Actions (RRA): In analogy to deciding on the appropriate therapy based on the diagnosed medical condition, decisions on whether and how to repair the present architecture degradation based on the identified causes have to be made. The activity RRA targets at recommending repairing of the implementation and the intended architecture based on the identified degradation causes in the system under investigation.

One essential aspect of the proposed approach is to complement existing techniques for each of these activities by a "learning component" with the aim of making the use of those techniques more accurate and efficient. ML shall be applied to interpret feedback from and interaction with the user, a software engineer, to increase accuracy and efficiency of those techniques over the lifetime of a system and for application in other systems. For example, a classifier could learn over time to distinguish architectural violations from inconsistencies that are considered allowed architectural divergences based on a software architect's manual classification of such exceptions in the set of identified inconsistencies in the past. We postulate the hypothesis that such a classifier could improve the accuracy of an architecture consistency technique.

Most techniques for any of these three activities rely on explicitly specified conceptual knowledge as depicted in the upper part of Figure 1. This knowledge is system-independent and comprises architectural concepts and the constraints they imply in the implementation, types of architectural inconsistencies and violations, etc. This knowledge is either embodied in the technique applied (e.g., reflexion modelling as ARC technique with its fix set of dependency constraints [16]) or to be specified by the user (like in tools like ArCh [17]). In the proposed approach, ML techniques shall be applied to extend such conceptual knowledge over time semi-automatically.

In the following sections, we look at these activities in more detail and the potential and envisaged usage of ML techniques.

B. Architecture Recovery and Consistency Checking

The techniques proposed to be used in this step are those suggested by Schindler and Rausch for architecture recovery [18] and by Herold for architecture consistency checking [17]. In the first approach, architectural concepts (e.g., patterns, conventions, communication paradigms, or architectural tactics) are described as set of properties that source code elements implementing these concepts have to fulfil. Based on these formally specified properties, instances of known architectural concepts can be recovered from a system's implementation. The difficulty lies in the exact definition of the relevant properties for such a concept. The authors instead suggest to make use of ML techniques to classify elements as instances of a particular architectural concept based on the already existing instances and their properties. This classifier could be refined over time and lead to more accurate recovery results.

This approach can be easily connected to the aforementioned approach to architecture consistency checking [17]. In this approach, architectural rules are attached to architecture concepts as first-order logic statements expressing constraints that a consistent implementation needs to fulfil. These rules are very similar to the properties used by Schindler and Rausch [18]. An interesting area for the application of ML techniques is the detection of exceptions from those architectural rules. Often, identified violations of consistency rules are considered acceptable exceptions from the rules (see also Buckley et al. [19]). ML techniques could help to identify common properties among such exceptions to reduce the number of such false positives in future consistency checks.

Moreover, almost all checking techniques and architectural concepts require some form of mapping between architecture elements and implementation elements. This is often considered a cumbersome and time-consuming task [19]. While there exist techniques to semi-automate this task [20] [21], the use of ML techniques has been investigated only for reflexion modelling with promising results [8] [9].

C. Degradation Cause Analysis

For the step of analysing degradation causes, we propose to further extend an approach proposed by Herold et al. [22]. This approach originally complemented reflexion modelling as ARC technique of choice but can be easily adapted to the technique suggested here. Violation causes are expressed as a combination of (1) structural patterns over architectural models, source code, and architectural inconsistencies between them and (2) quantitative properties formulated as metrics and target values to express likely properties of architecture or implementation elements if a specific degradation cause seems to be the reason for an identified architecture violation. The closer the actual metric values are in the context of a specific violation, the more likely the degradation cause is considered to be the actual reason. A recommendation system integrated into a reflexion modelling tool proposes the potential causes in descending order according to their computed probabilities to the user.

We assume that ML techniques can improve these recommendations based on previous user feedback that confirmed or declined suggested recommendations. In particular in cases of competing degradation causes (with similar probabilities), supervised techniques could utilize other features of the relevant architecture and implementation fragments to prioritise certain causes over others based on previous experiences. In a similar way, a system could learn weightings of the "symptoms" expressed in the quantitative properties of a degradation cause, to adapt to system-specific characteristics. If, for example, degradation causes refer to textual similarity metrics (among other metrics) to measure conformance with naming conventions, but users very often decline such causes despite high scores computed by the recommendation system (because the naming convention does not apply in the system at hand), a lower weight decreasing the naming convention's influence might be beneficial. Furthermore, unsupervised techniques could help to identify new types of degradation causes based on existing but yet unclassified inconsistencies and relevant features of the implementation elements related to them.

D. Recommending Repair Actions

For the step of recommending repair actions, we envisage to adapt the approach proposed by Terra et al. who describe a recommendation system that suggests refactorings for violations of dependencies as modelled in an architectural model of a system [23]. Instead of producing recommendations per single violations as in the original approach, the degradation causes identified during DCA will be the units for which recommendations will be made such that they do not look at violations in isolation but consider their semantic context.

ML techniques can help to overcome one of the main limitations of the technique so far which is the fix and predefined priorities of refactorings. If two or more refactorings are applicable to resolve an architecture inconsistency, the one with the higher priority will be recommended. Again, the technique could be improved in this regard through learning from previous actions of the user, having accepted and rejected recommended refactorings, and looking at which recommendation alternatives were chosen in different contexts. Moreover, we envisage to also observe how these recommendations are actually turned into actions. If the recommended (series of) refactorings are frequently extended by additional actions, for example, the recommended refactoring could be adapted, or a new recommendation could be added to the recommendation system. Similarly, unsupervised learning techniques could identify recommendations from repair actions that the user performs without following any of the suggestions of the system at all.

IV. Conclusion and Future Work

In this paper, we sketched a novel holistic approach to counteracting software architecture degradation in software-intensive systems through extending existing techniques by machine learning. Based on the preliminary results of a systematic literature review, we conclude that an holistic approach making use of machine learning techniques is missing. We assume that this new direction leads to improved accuracy and efficiency in mitigating SAD and, hence, to a higher acceptance of the corresponding techniques in practice.

In the immediate future work, we intend to adapt and extend the tools for the three activities identified. This includes extending them by the means to retrieve feedback on suggested architecture violations, degradation causes, etc. from the user and to feed this information into appropriate learning mechanisms.

In addition, this involves gathering and formalising some of the conceptual knowledge as outlined in Section III and Figure 1. This would serve as baseline knowledge in the evaluation of the approach based on which the learning mechanisms would adapt to project or system-specific settings over time. This data can partially come from literature, such as the formalization of patterns as architectural constraints. Given the lack of studies in analysing degradation causes, however, empirical observations from case studies with real-world software projects and potentially from experiments with students or practitioners will be performed. Based on this, we will thoroughly evaluate our approach to identify the strengths, and possible weaknesses.

REFERENCES

[1] L. De Silva and D. Balasubramaniam, “Controlling software architecture erosion: A survey,” *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[2] N. Ali, S. Baker, R. O’Crowley, S. Herold, and J. Buckley, “Architecture consistency: State of the practice, challenges and requirements,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 224–258, 2018.

[3] Y. Zhang, *New Advances in Machine Learning*. BoD–Books on Demand, 2010.

[4] J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng, “A survey of machine learning for big data processing,” *EURASIP Journal on Advances in Signal Processing*, vol. 2016, no. 1, p. 67, 2016.

[5] S. Herold, M. Blom, and J. Buckley, “Evidence in architecture degradation and consistency checking research: Preliminary results from a literature review,” in *Proceedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW ’16. ACM, 2016, pp. 1–7.

[6] A. Grewe, C. Knieke, M. Körner, A. Rausch, M. Schindler, A. Strasser, M. Vogel, and (Keine Angabe), “Automotive software product line architecture evolution: Extracting, designing and managing architectural concepts,” in *International Journal on Advances in Intelligent Systems*, Hans-Werner Sehring, Ed. IARIA, 2017, pp. 203–222.

[7] A. Corazza, S. Di Martino, V. Maggio, A. Moschitti, A. Passerini, G. Scanniello, and F. Silvestri, “Using machine learning and information retrieval techniques to improve software maintainability,” in *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, A. Moschitti and B. Plank, Eds. Springer, 2013, pp. 117–134.

[8] S. M. Naim, K. Damevski, and M. S. Hossain, “Reconstructing and evolving software architectures using a coordinated clustering framework,” *Automated Software Engineering*, vol. 24, no. 3, pp. 543–572, Sep 2017.

[9] T. Olsson, M. Ericsson, and A. Wingkvist, “Semi-automatic mapping of source code using naive bayes,” in *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ser. ECSA ’19. ACM, 2019, p. 209–216.

[10] A. Ghannem, G. El Boussaidi, and M. Kessentini, “On the use of design defect examples to detect model refactoring opportunities,” *Software Quality Journal*, vol. 24, no. 4, p. 947–965, Dec. 2016.

[11] M. Mirakhorli, J. Carvalho, J. Cleland-Huang, and P. Mäder, “A domain-centric approach for recommending architectural tactics to satisfy quality concerns,” in *2013 3rd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*, July 2013, pp. 1–8.

[12] M. Mair, S. Herold, and A. Rausch, “Towards flexible automated software architecture erosion diagnosis and treatment,” in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA ’14 Companion. ACM, 2014, pp. 1–6.

[13] F. Arcelli Fontana, P. Avgeriou, I. Pigazzini, and R. Roveda, “A study on architectural smells prediction,” in *2019 45th Euromicro Conference on Softw. Eng.and Advanced Applications (SEAA)*, Aug 2019, pp. 333–337.

[14] F. Khomh, S. Vaucher, Y. Guéhéneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in *2009 Ninth International Conference on Quality Software*, Aug 2009, pp. 305–314.

[15] J. Lenhard, M. M. Hassan, M. Blom, and S. Herold, “Are code smell detection tools suitable for detecting architecture degradation?” in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, ser. ECSA ’17. ACM, 2017, p. 138–144.

[16] G. C. Murphy, D. Notkin, and K. Sullivan, “Software reflexion models: Bridging the gap between source and high-level models,” *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 4, p. 18–28, Oct. 1995.

[17] S. Herold and A. Rausch, “Complementing model-driven development for the detection of software architecture erosion,” in *2013 5th Int. Workshop on Modeling in Software Engineering (MiSE)*, May 2013, pp. 24–30.

[18] M. Schindler and A. Rausch, “Architectural concepts and their evolution made explicit by examples,” in *Proceedings of The Eleventh International Conference on Adaptive and Self-Adaptive Systems and Applications, ADAPTIVE 2019*. IARIA, 2019, pp. 38–43.

[19] J. Buckley, N. Ali, M. English, J. Rosik, and S. Herold, “Real-time reflexion modelling in architecture reconciliation: A multi case study,” *Information and Software Technology*, vol. 61, pp. 107 – 123, 2015.

[20] A. Christl, R. Koschke, and M.-A. Storey, “Automated clustering to support the reflexion method,” *Information and Software Technology*, vol. 49, no. 3, pp. 255 – 274, 2007, 12th Working Conference on Reverse Engineering.

[21] R. A. Bittencourt, G. J. d. Santos, D. D. S. Guerrero, and G. C. Murphy, “Improving automated mapping in reflexion models using information retrieval techniques,” in *2010 17th Working Conference on Reverse Engineering*, Oct 2010, pp. 163–172.

[22] S. Herold, M. English, J. Buckley, S. Counsell, and M. Ó Cinnéide, “Detection of violation causes in reflexion models,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 565–569.

[23] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, “A recommendation system for repairing violations detected by static architecture conformance checking,” *Software: Practice and Experience*, vol. 45, no. 3, pp. 315–342, 2015.