ULRR

Towards flexible automated software architecture erosion diagnosis and treatment

Item Type	Meetings and Proceedings
Authors	Mair, Matthias;Herold, Sebastian;Rausch, Andreas
Citation	Working IEEE/IFIP Conference on Software Architecture (WICSA);article 9i
Publisher	Association for Computing Machinery
Download date	2026-06-08 17:28:42
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/3980

Towards Flexible Automated Software Architecture Erosion Diagnosis and Treatment

Matthias Mair
Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4
38678 Clausthal-Zellerfeld, Germany
matthias.mair@tu-clausthal.de

Sebastian Herold
Lero–The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
sebastian.herold@lero.ie

Andreas Rausch
Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4
38678 Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

ABSTRACT

Uncontrolled software architecture erosion can lead to a degradation of the quality of a software system. It is hence of great importance to repair erosion efficiently. Refactorings can help to systematically reverse software architecture erosion through applying them in the system where architectural violations have been detected. However, existing refactoring approaches do not address architecture erosion holistically.

In this paper, we describe and formalize the theoretical problem of repairing eroded software systems by finding optimal repair sequences. Furthermore, we investigate the applicability and limitations of existing refactoring approaches. We argue, true to the motto "more knowledge means less search" that using formalized and explicit knowledge of software engineers—modeled as fault patterns and repair strategies—combined with heuristic search techniques could overcome those limitations.

This paper outlines a new approach—analog to a patient history in medicine—we have been starting to investigate in our recent research and also aims at stimulating a discussion about further research challenges in repairing eroded software systems.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

software architecture erosion, heuristic search, repairing architecture erosion, fault pattern

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
WICSA'14 April 07 - 11 2014, Sydney, NSW, Australia
Copyright 2014 ACM 978-1-4503-2523-3/14/04 ...$15.00
http://dx.doi.org/10.1145/2578128.2578231.

1. MOTIVATION

The longer a software system evolves and the higher its complexity is, the more likely it is that software architecture erosion might occur [22]. This term designates the progressive divergence of the intended software architecture and its realization. The reasons for this phenomenon are manifold, for example, coding workarounds through time pressure, employee turnover or adapting to new requirements [10] might let the realization diverge from the intended software architecture. In general, architecture erosion leads to a degradation of system quality attributes like maintainability and adaptability. In the long run, heavily eroded software systems become too costly to be maintained and might need to be replaced by expensive re-developments [17].

According to de Silva et al. [2] one way to control software architecture erosion is to detect and repair architecture erosion. For this purpose, approaches for conformance or consistency checking are used to detect architecture violations, i. e. inconsistencies of the intended architecture and its realization. Approaches to repair these violations consist mostly of reengineering and refactoring techniques [5]. However, the task of repairing architecture erosion requires a broad and profound understanding of the often rather complex software system at hand. Finding a good or even optimal way to repair the software system is a difficult task even for experienced software engineers.

In this paper, we will investigate the problem of how to support software engineers in repairing eroded software systems. We will discuss how the problem is addressed by the state of the art in refactoring. Furthermore, we will propose an outline of a new approach based on the consideration of architectural knowledge about fault patterns and repairs of architecture violations to automatically recommend a good sequence of refactorings for an eroded software system.

The remaining paper is structured as follows. Section 2 illustrates the theoretical problem that this work addresses. In Sec. 3, we take a look on related work tackling the problem of finding sequences of refactorings in complex systems. The proposed approach is presented in Sec. 4. Section 5 describes future work and the paper is concluded in Sec. 6.

2. PROBLEM STATEMENT

In an eroded software system, the intended architecture is inconsistent with the realization of the system. Repairing,

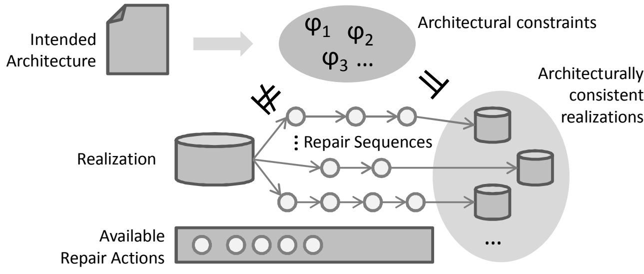

Figure 1: The problem of finding optimal architecture erosion repair sequences (cf. [12]).

or reversing, the erosion process can be understood as the task of transforming and manipulating the realization such that consistency is re-established. Finding the optimal way to do that is the task of finding a sequence of repair actions that is optimal regarding a measure of costs, for example, least actions to perform, required time to perform, etc.

According to Eden et al. [4], a software architecture can be represented as set of logical statements about possible realizations. A realization conforms to an architecture, if and only if it fulfills that set of logical statements. Eden et al. lists many different examples of architectural aspects, such as patterns, reference architecture, naming conventions, metrics that can be interpreted this way and expresses them as first-order logic statements.

Figure 1 illustrates the theoretical problem of repairing an eroded software system by finding optimal repair sequences. Following Eden et al., a given intended software architecture can be understood as a set of first-order logic statements

$$\Phi = \{\varphi_1, \varphi_2, ..., \varphi_n\}$$

about structures and behaviors of systems. A realization r of the system, formalized as finite structure, conforms to the intended architecture if and only if

$$r \models \Phi$$

In reverse, an eroded software system has architecture violations ($r \not\models \Phi$). To re-establish architecture consistency, a realization has to be transformed in a realization r' with $r' \models \Phi$. Therefore, predefined repair actions

$$T := \{t_1, t_2, ..., t_m\}$$

are available, for example, a set of refactorings. The repair actions

$$t_i : L \rightarrow R$$

can be understood as replacement rules. Corresponding, t_i indicates that by application a substructure L from the realization is replaced through another substructure R.

Efficient architecture erosion repair is understood as an optimal sequence of repair actions regarding to a cost function $C : T^* \rightarrow \mathbb{N}$. For example, the minimized cost function $C(t) := |t|$ would express, that preferable short sequences of repair actions are considered as efficient. The search for an efficient architecture erosion repair, for a realization r ($r \not\models \Phi$), can be recognized as search problem for a sequence $s = t_{i_1} t_{i_2} ... t_{i_k} \in T^*$ $(1 \leq i_j \leq m)$, such that applies

$$r \overset{t_{i_1}}{\rightarrow} r_1 \overset{t_{i_2}}{\rightarrow} ... \overset{t_{i_k}}{\rightarrow} r_n = r' \text{ with } r' \models \Phi \text{ and} \quad (1)$$

$$C(s) \leq C(s') \text{ for all } s' \in T^* \text{ with (1),} \quad (2)$$

if C is to minimize (or rather maximize $C(s) \geq C(s')$). More specific, the sequence of transformations/repair actions $t_{i_1} ... t_{i_k}$ are applications of t_i.

Due to the undecidability of first-order logic formulas Φ, the set of realizations conforming to the given architecture is not computable. Consequently, the set of sequences of repair actions, with the above property (1), is not computable as well as far as the set of repair actions is complete, i. e. given an arbitrary realization; any other realization can be generated by executing a certain sequence of available repair actions. This implies that we have to perform an exhaustive search for the optimal sequence of repair actions which fails in reasonable time in a realistic scenario, because of the large search space in combination with first-order logic model checking of potential solutions.

There are several practical solutions which reduce the problem at least in one of the following possibilities:

- Reduction of the complexity of the considered architectural aspects and violations. Either the expressiveness of the language used to describe the intended software architecture is reduced, or the approach focuses on certain aspect of the architecture only. This might simplify consistency checking to something less complex than first-order logic model checking but naturally reduces the cases of erosion that can be detected.

- Reduction of the considered repair actions. This decreases the size of the search space, or to allow certain search strategies to do the search more efficiently.

- Apply heuristic search techniques to find approximate solutions in reasonable time.

In the following section, relevant related approaches and their techniques to tackle the problem of finding refactorings/repair actions in complex system are described.

3. STATE OF THE ART OF COMPLEX REFACTORING

The following approaches all aim at providing automatic tool support for finding places for refactorings in a software system. They vary in the way of detecting architectural erosion[1], supported refactorings, and the used search-based techniques to determine refactorings to be applied.

As discussed in Sec. 2, one practical solution is to reduce the complexity of the considered architectural aspects. For example, many approaches are concentrating on the estimation of metrics to detect refactoring opportunities in software systems. Ouni et al. [16] use metrics for a multi-objective approach for Fowler's catalog [5] of refactorings. In [6], Ghannem et al. and in a similar way Jensen et al. [9] and O'Keeffe et al. [15] restrict their approaches to metrics regarding class structures and support the relevant refactorings from the same catalog. All these approaches have in common that they use, in addition to the reduction of considered architectural aspects, heuristic search techniques

[1]Although most the approaches do not explicitly name architectural erosion as motivation, they are related since their goal is to restore intended or desired structures in a software system.

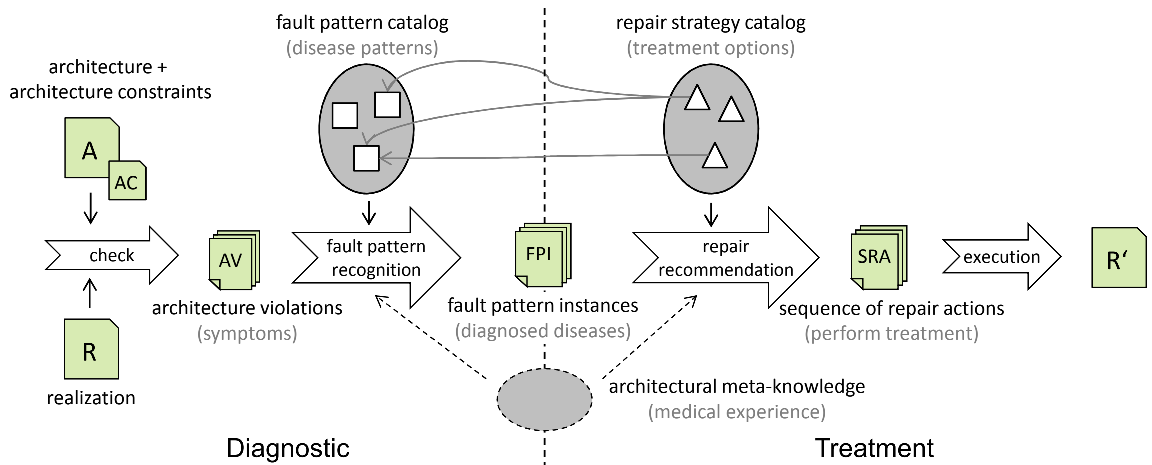

Figure 2: Overview and process of the proposed approach.

to determine a sequence of refactoring operations. Furthermore, these approaches are applicable by focusing on quantifiable architectural principles through metrics. They do not consider architectural aspects like dependency constraints or other structural constraints.

Other approaches focus on single structural constraints such as simple graph patterns on program dependency graphs [3], or cycles in such graphs [18]. In [7], Griffith et al. use a control flow graph to detect code smells. While the formalism—graphs—provides great expressiveness, the set of actual expressions they investigate is limited.

Moghadam et al. describe an approach to refactor source code according to a model which they call the "desired design" [13]. A design model of the current source code is reconstructed and search-based techniques are applied to find a sequence of refactorings to transform the system towards the desired design. Mapped to the theoretical problem, this approach simplifies the general problem by providing a higher-level model of the target realization which simplifies searching and consistency checking.

In contrast to the previous approaches, the work of Terra et al. considers the intended architecture written in a language called *Dependency Constraint Language* (DCL) [20]. An architecture written in DCL defines basically coarse-grained architecture components that are mapped to source code elements and specifies dependency constraints between them. A tool called *dclcheck* can automatically detect violations of these constraints in source code of the software system. In [21], Terra et al. complement dclcheck by a recommendation system which recommends refactorings to solve violations. However, recommendations are made only locally, i. e. for each single violation. This also means that the approach does not provide sequences of refactorings which takes into account interdependencies.

It can be observed that the current state of the art provides appropriate support for determining a sequence of refactorings for single kinds of architectural erosion. Most approaches focus on single architecture aspects as discussed in Sec. 2. Furthermore, it is not possible or difficult in most cases to extend the set of repair actions to adapt the approach to user- or domain-specific architectural knowledge. Moreover, it is unclear if the selected meta-heuristic search methods are still useful after adapting the approaches. Only the work of Terra et al. allow to define user-specific repair actions but does not search for optimal sequences.

4. SOLUTION OUTLINE OF THE PROPOSED APPROACH

4.1 Overview of the Approach

Existing approaches are unsatisfactory by addressing the general problem due to their strong restrictions, thus a new and more powerful approach with the following properties is needed:

- Support of many architecture aspects.
- Support of many, possibly user-specific, atomic and composite repair actions.
- Search for and recommend optimal repair sequences in reasonable time.

To be able to provide these properties, we outline an approach that takes into account the knowledge of software architects about architecture violations and repair options. The proposed approach is depicted in Fig. 2 from a bird's eye perspective. The main ideas and phases of the approach are motivated by analogies between repairing software architecture erosion and the medical treatment of a diseased human patient.

In both domains, the overall process is divided into the phases of `diagnostics` and the actual `treatment`. To start the diagnostic an `architecture`, `architecture constraints` and a `realization` of a software system is required. In analogy to medicine, for example, would a body temperature constraint of a human be 36.0 to 37.5 degree Celsius and a realization, of course, represents a patient. Afterwards, a consistency `check` between the intended architecture and its realization can be done to detect `architecture violations` that can be compared to `symptoms` in medicine detected by a doctor. The next step for a doctor is to `diagnose diseases` by looking at all symptoms and trying to match them to `disease patterns`. For instance, a doctor could diagnose influenza when a patient shows the typical symptoms like high fever, body aches and headaches. The proposed approach follows this analogy by defining and collecting fault

patterns that formalize software engineering expertise about the reasons for the common, contextual occurrence of architectural violations `fault pattern recognition`. The result of the diagnosis is a set of `fault pattern instances` detected in the investigated software system.

Based on the fault pattern instances, the treatment can begin. In medicine, there are often many `treatment options` for one disease, and the doctor has to decide which treatments should be performed. In a similar way, there are possibly different options to repair architecture erosion. Knowledge about options to repair fault patterns is stored in a `repair strategy catalog`. It is important to be able to compose strategies out of atomic repair actions and existing strategies to form user- or domain-specific strategies. Like a combination of treatments in medicine ("use cold compress against fever" and "use antiviral drugs"), repairing architecture erosion for single fault patterns can be composed of more generally usable actions. For example, resolving dependency violations can be composed of (among others) "Move class" and renaming refactorings.

The challenge of the `repair recommendation` process is to recommend a `sequence of repair actions` which is optimal regarding a cost function as described in Sec. 2. Due to the reduced search space—through the use of fault patterns and their mapping to repair strategies—it will be possible to search efficiently for repair sequences. Even if the search space in general might be too large for exhausting search techniques, heuristic search techniques will benefit from the aggregation of violations to fault patterns instances and the knowledge about repair actions for them. Finally, the `execution` transforms the realization to an architecturally consistent realization ($r' \vDash \Phi$).

In addition to the specified fault patterns and repair strategies this approach considers `architectural meta-knowledge`, i. e. formalized experts knowledge like experiences, strategies and tactics to improve the fault pattern recognition and the repair recommendation process, as well. For example, such as medics know about the interdependencies between certain diseases and therapies, software architects might provide knowledge about classes of fault patterns and repair strategies. Useful information, for example, is to direct the search for repair actions to search for repairing's of structural dependency violations in which a component is involved, before trying to repair inconsistencies in the component regarding interaction protocols. These might not be necessary anymore after the component is moved to a different subsystem because the protocol must not be followed in that subsystem.

4.2 Example

In the following, we will illustrate the process by example focusing on the diagnosis phase of the overall process as described in the previous section.

To keep the example handy, we assume a simple architecture model as used in reflexion modeling [14]. The intended software architecture is captured as modules and relationships between modules that models which dependencies are allowed between them. Modules are mapped to source code elements, for example, packages or namespaces. The actual dependencies—such as usage relations or inheritance—in the source code are analyzed and compared to the allowed ones as modeled in the intended architecture.

For checking such dependency violations, we apply the

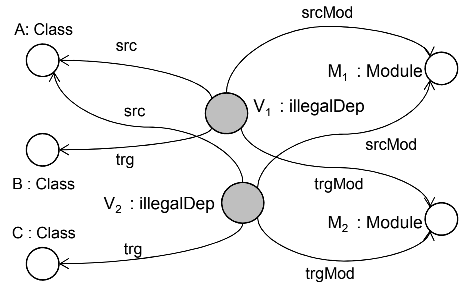

Figure 3: Example of fault pattern graph with dependency violations.

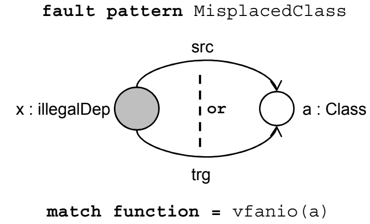

Figure 4: Fault pattern for misplaced classes.

Architecture Checker (ArCh) tool developed in [8]. ArCh is able to check dependency violations as required for reflexion modeling; it is based on an expressive query language which enables us to define arbitrary first-order logic constraints on object-oriented software system structures.

As result of checking a software architecture, ArCh returns the violated constraints with the binding of variables causing the constraint to evaluate to true. For example, it exists a constraint `illegalDep(srcMod,trgMod,src,trg)`, which evaluated to true for two classes `src` (contained in module `srcMod`) and `trg` (contained in module `trgMod`) if and only if `src` depends on `trg` although `srcMod` is not allowed to depend on `trgMod`.

The system structure together with the detected violations can be represented as typed and directed graph. This graph is called fault pattern graph. The example graph in Fig. 3 shows a graph with dependency violations as described above. As described in the previous section, architecture violations are aggregated to form fault pattern instances which represent reasons for violations. Known fault patterns can hence be described as graph patterns; the goal of the diagnosis phase is to find a set of instances of those graph patterns that cover as many of violation nodes in the fault pattern graph as possible.

One possible cause for dependency violations as described are misplaced classes, i. e. classes mapped to the wrong module. It is likely that this might the case for many violations if a class or interface has a high relative number of violations compared with its total numbers of dependency with other classes. For example, if a class has 20 incoming or outgoing dependencies and participates in 15 dependency violations occurrences, the probability that the class is in a wrong module is high.

Figure 4 shows a very simple graph pattern to model this knowledge about misplaced classes. The pattern named `MisplacedClass` states that we look for system elements that participate either as source or target of a dependency vi-

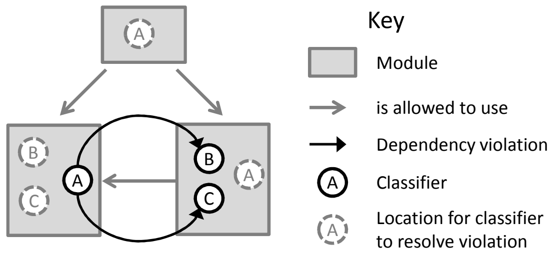

Figure 5: Example of a software system realization and its architecture violations.

olation. If this is the case for a substructure of a fault pattern graph, i. e. a corresponding structure is found, a matching function evaluates "how good" the pattern matches, i. e. how likely it is that the identified fault pattern is the causes for the violations in the fault pattern instance. In the example, the matching function is computed by the metric `vfanio`. It relates the violation fan-in VFanIn—which is the number of incoming dependency violations—and the fan-in which is the number of incoming dependencies; the same relationship is computed for the violation fan-out and the total fan-out:

$$vfanio(c) = \frac{\frac{VFanIn}{(FanIn+1)} + \frac{VFanOut}{(FanOut+1)}}{2}$$

Misplaced classes are of course not the only possible cause for dependency violations. A dependency causing a violation that is analyzed and found not to be used at all should be dealt with in a different way than an obviously misplaced class. General symptoms like these—similar to bad smells [7]— but also architecture- or domain-specific causes can be captured as fault patterns.

Goal of the diagnosis phase is to find fault pattern instances covering the identified architecture violations and to optimize the average matching function value. The result set of fault pattern instances are the starting point for the treatment phase; through the connection of fault patterns with repair strategies, the search space for possible repairing's is reduced. In the example of misplaced classes, a possible repair strategy would be to apply the move class refactoring to the misplaced class to move it to module which would resolve the violations aggregated by the fault pattern instance. Figure 5 illustrates for a small system possible places to move classes (identified as misplaced) to resolve dependency violations.

5. FUTURE WORK

In the future, we plan to realize and implement the proposed approach, as described in Sec. 4, in a prototypical tool. Thereafter, industrial case studies are planned, to evaluate this approach and show its efficiency on real software systems. However, a lot of preliminary work has to be done. Each module in the overall process has its own research challenges, where solutions have to be worked out, before these modules can be combined in one approach.

Modules are discrete parts of the proposed approach with the aim to allow simultaneous work on it. Furthermore, modules have dependencies among each other, for example, the fault pattern recognition has to deal with the formalization of the fault patterns. Hence, clearly defined interfaces are needed. In the following are the modules briefly described with their research focus and challenges, applicable techniques, first ideas and how we plan to address it. In other words, every module can be understood as a work package which has to be finished, before the integration can start.

The core idea of the proposed approach is that architecture violations cannot be considered independent but need to be analyzed in context. Thus, architecture violations are concluded to fault patterns which describe the knowledge of software architects about the fact that combined or exclusive occurrences of these architecture violations point to a common cause. The aim of this module is to develop a meta-model for modeling fault patterns on architecture violation graphs as inexact graph patterns. The challenge hereby is that fault patterns might regard to domain specific architectural aspects and violations and hence must be specifiable by software architects.

The fault pattern recognition module has as its goal to recognize as preferably comprehensive and consistent the modeled fault patterns. It should be noted that the assignment from violations to fault patterns is not unique, i. e. one architecture violation can be part of more than one detected fault pattern. The recognition cannot be done manually due to the complexity of large software systems that an algorithm is needed to automatically recognize fault patterns. Therefore, approaches are useful which recognize inexact graph patterns, as mentioned in the subcategory tree search in inexact graph matching by Conte et al. [1]. The result of this work package is the development of an algorithm for automated fault pattern recognition.

The module repair strategy catalog represents engineer's expertise, how recognized fault pattern instances can be repaired or resolved. Thus, repair strategies are formalized and modeled as automatic executable graph transformation rules which can be executed on the realization of a software system. Strategies are composite repair actions and have dependencies to existing fault patterns, i. e. one strategy consists of a set of repair actions and can repair at least one fault pattern. The aim of the work package is to develop a meta-model for modeling repair strategies. Furthermore, the meta-model should allow the composition of existing repair strategies to new and more complex strategies. In this case, it is possible to model a set of more common or atomic repair strategies like a catalog of refactorings which can be reused to define complex or if necessary domain specific repairs.

The challenge of the repair recommendation module is to find an applicable heuristic search algorithm [19] and to specify its optimization function, if possible the best, to recommend a sequence of repair actions. The heuristic search algorithm should handle dynamic changing search spaces due to the possibility that repaired architecture violations can bring up new violations or repair more than one violation at once. The optimization function, most likely, consists of many individual functions which are combined by weight or Pareto optimal [11]. The repair recommendation result is a sequence of repair actions which serves as foundation for a software architect or developer to eliminate software architecture erosion.

The architectural meta-knowledge module is a cross-cutting module and supports the fault pattern recognition and the repair recommendation. The aim is to integrate and consider more experts knowledge and know-how to improve

the efficiency of the recommendation of repair actions, for example, tactics to order the execution of repairs. The challenges are how to get this knowledge from experts, how to formalize it and how can it be integrated in the proposed approach.

6. CONCLUSION

During the evolution of a complex software system it is most likely that software architecture erosion happens. In general architecture erosion cannot be avoided completely and hence it is an important task to repair eroded software systems. Refactoring techniques, heuristic search and formalized engineers expertise are in combination a major step to tackle this task.

Existing approaches reduce the complexity of the general problem to come up with practical solutions by limiting the set of considered architectural aspects or the set of possible repair actions. Both limitations, to deal with the complexity of the task, lead to approaches that do not address the overall problem holistically.

Instead, we suggest to formalize more existing engineering knowledge and know-how, true to the motto "more knowledge means less search" [23]. This knowledge as part of every module in the overall process combined with heuristic search techniques should lead to good architectural repair actions in complex eroded systems, and outline a new possible approach to control architecture erosion.

In the future, we will further elaborate this approach and plan to conduct extensive industrial case studies to show the applicability to "real life" erosion cases and the usefulness for dealing with software architecture erosion.

7. ACKNOWLEDGEMENTS

This work was supported, in part, by Science Foundation Ireland grant 12/IP/1351 to Lero - The Irish Software Engineering Research Centre (www.lero.ie).

8. REFERENCES

[1] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):265–298, 2004.

[2] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *J. Syst. Softw.*, 85(1):132–151, 2012.

[3] J. Dietrich, J. McCartin, E. Tempero, and S. M. A. Shah. On the existence of high-impact refactoring opportunities in programs. In *Australasian Computer Science Conf.*, volume 122, pages 37–48. ACS, 2012.

[4] A. Eden, Y. Hirshfeld, and R. Kazman. Abstraction classes in software design. *IEE Proc. - Softw.*, 153(4):163–182, 2006.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[6] A. Ghannem, G. Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In G. Ruhe and Y. Zhang, editors, *Search Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2013.

[7] I. Griffith, S. Wahl, and C. Izurieta. Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility. In S. Aljahdali, editor, *Proc. of the ISCA 24th Int. Conf. on Computer Applications in Industry and Engineering.* ISCA, 2011.

[8] S. Herold. *Architectural Compliance in Component-Based Systems. Foundations, Specification, and Checking of Architectural Rules.* PhD thesis, Clausthal University of Technology, 2011.

[9] A. C. Jensen and B. H. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In M. Pelikan, editor, *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, page 1341. ACM, 2010.

[10] M. Lindvall and D. Muthig. Bridging the software architecture gap. *IEEE Computer*, 41:98–101, 2008.

[11] H. Lipson, M. Harman, and L. Tratt. Pareto optimal search based refactoring at the design level. In H. Lipson, editor, *Proc. of the 9th conf. on genetic and evolutionary computation*, page 1106. ACM, 2007.

[12] M. Mair and S. Herold. Towards extensive software architecture erosion repairs. In *Proceedings of the 7th European conference on Software Architecture*, ECSA'13, pages 299–306. Springer-Verlag, 2013.

[13] I. H. Moghadam and M. O. Cinneide. Automated refactoring using design differencing. In *Proc. of the 16th Europ. Conf. on Softw. Maintenance and Reengineering*, pages 43–52. IEEE, 2012.

[14] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.

[15] M. O'Keeffe and M. Cinneide. Search-based software maintenance. In *Proc. of the 10th Europ. Conf. on Software Maintenance and Reengineering*, 2006.

[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20:47–79, 2013.

[17] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Softw.*, 26(2):28–35, 2009.

[18] S. M. A. Shah, J. Dietrich, and C. McCartin. Making smart moves to untangle programs. In *Proc. of the 2012 16th Europ. Conf. on Software Maintenance and Reengineering*, pages 359–364. IEEE, 2012.

[19] E. A. Silver. An overview of heuristic solution methods. *Journal of the Operational Research Society*, 55(9):936–956, 2004.

[20] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exp.*, 39:1073–1094, 2009.

[21] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *Proc. of the 16th Europ. Conf. on Software Maintenance and Reengineering*, pages 335–340. IEEE, 2012.

[22] J. van Gurp and J. Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.

[23] P. H. Winston. *Artificial intelligence.* Series in computer science. Addison-Wesley, 2 edition, 1984.