Assessing Code Decay: A Data-driven Approach

Ajay Bandi
Math., Computer Science & Information Systems
Northwest Missouri State University
Maryville, MO, 64468, USA
ajay@nwmissouri.edu

Edward B. Allen and Byron J. Williams
Computer Science and Engineering
Mississippi State University
Mississippi State, MS, 39762, USA
(allen, williams)@cse.msstate.edu

Abstract

Code decay is a gradual process that negatively impacts the quality of a software system. Developers need trusted measurement techniques to evaluate whether their systems have decayed. This paper aims to assess code decay by discovering software architectural violations. We propose a methodology that uses Lightweight Sanity Check for Implemented Architectures (LiSCIA) to derive architectural constraints represented by can-use and cannot-use phrases. Our methodology also uses a reverse engineering tool to discover architectural violations. We also introduce measures that indicate code decay in a software system. We conducted a case study of a proprietary system (14 versions) to demonstrate our methodology for assessing code decay. Resulting architectural constraints and architectural violations were validated by the expert of the system. The proposed code decay metrics can give managers data-driven insight into the process of software development, the history of the software product, and the status of unresolved violations. Our results and qualitative analysis showed that the methodology was effective and required a practical level of effort for moderate sized software systems.

keywords: architecture constraints, architecture violations, data-driven approach, code decay metrics, reverse engineering, software evolution

1 Introduction

Properly implemented, code should follow specified architectural constraints and conform to the conceptual architecture. However, architectural violations are often due to new interactions between modules that were originally unintended in the planned design [9, 12]. Such violations may be caused by adding new functionality, or modifying existing functionality to implement changing requirements or to repair defects. When such changes are inconsistent with the planned architecture and design principles, the system becomes more complex, hard to maintain, and defect prone [6, 10, 13]. Often, redesign or reengineering of the whole system is the only practical solution for this problem [7]. The phenomenon of gradual increase in software complexity due to unintended interactions between modules that are hard to maintain has been termed architectural degeneration and code decay [6, 8]. Research in software evolution shows that violations of architecture and design rules cause code to decay [6, 7, 9]. Code decay is a gradual process that degrades the maintainability of the software system. In this paper, code decay refers to the rate of violations of architecture and design rules over time that make software more difficult to modify. This research focuses on violations of architectural constraints. Architectural constraints are the architectural rules represented by can-use or cannot-use phrases. Some authors define "code decay" more generally or with respect to non-architectural characteristics [2].

Using a data-driven approach to minimizing code decay is important to software engineering practitioners who are focused on improving software quality during software maintenance. Code decay is an attribute that is evident only in retrospect. It is usually assumed that "decay" is a gradual process that goes unnoticed until a crisis occurs. One can detect decay by comparing measured attributes from the past with current values, and determine that quality has "decayed." A challenge is using data-driven approaches to detect incipient "decay" well before a crisis develops. The main goal of our research is to find ways to derive architectural constraints, to detect architectural violations, and to assess code decay of software over multiple versions. This paper presents results from Bandi's dissertation [1]. The remainder of this paper is organized as follows. Section 2 details our proposed methodology for practitioners. Section 3 describes the case study. Section 4 presents our case study results and its analysis. Section 5 discusses results, and threats to validity. Section 6 presents conclusions and future work.

978-1-943436-01-9 / copyright ISCA, SEDE 2015
October 12-14, 2015, San Diego, California, USA

2 Methodology for Practitioners

To apply our methodology, the practitioner uses tools to extract the architecture of the software, to calculate the architecture dependencies, to identify architectural violations for a given set of architectural constraints represented by can-use or cannot-use phrases, and LiSCIA questionnaire [3, 4]. We used Lattix for extracting the architecture dependencies, and for identifying architectural violations. A similar reverse engineering tool would also work. Given a software system that has multiple versions, the following procedure presents the major steps in our methodology.

(1) While there is an unanalyzed refactored version
   (a) Choose an initial or refactored version that is representative of the architecture.
   (b) Choose the subsequent versions.
   (c) Derive the architectural constraints based on initial or refactored version.
   (d) While not done with all the versions
      i. Discover current architectural violations in a version.
   (e) End while
   (f) Identify new violations and solved violations in each version.
   (g) Assess code decay over multiple versions.
(2) End while

Figure 1 shows the methodology for deriving architectural constraints and discovering architectural violations. The roles of the participants when applying our methodology are evaluator, expert, and analyst. The same person can fulfill multiple roles and multiple people can fulfill the same roles. In order to get the most out of the evaluation, at least two persons should be involved in order to create discussion. One important element of our methodology is that an expert of the system must participate in deriving architectural constraints.

In our methodology we use LiSCIA [3, 4] to derive architectural constraints. LiSCIA has two major phases (start-up phase and evaluation phase). The following are the steps to derive architectural constraints.

(1) Analyst prepares the following software artifacts before the start-up phase of LiSCIA.
   - Source code of a system in an IDE from the repository using the version control system.

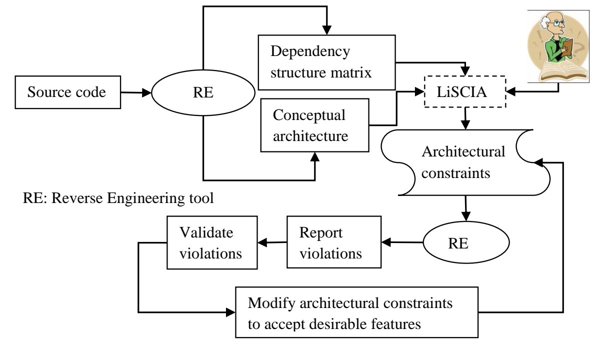

Figure 1: Deriving architectural constraints and detecting architectural violations

   - System's conceptual architecture and the dependency structure matrix derived from the source code using the reverse engineering tool.

(2) Start-up phase: The following are the steps during the start-up phase of LiSCIA.
   (a) Analyst explains the roles to the participants.
   (b) Analyst asks the participants to skim the LiSCIA questionnaire, which is given in Appendix B of Bandi's dissertation [1] and also in [3].
   (c) Analyst provides the artifacts (source code, conceptual architecture, and dependency structure matrix) to the participants.
   (d) The participants review the organization of the source code and draw the high level architecture diagram. During this step, they may use the artifacts provided by the analyst.
   (e) The participants define the components (logical groups of functionality) of the system.
   (f) The participants define the name patterns for each component defined in the above step.
   (g) The participants list the technologies used.
(3) Evaluation phase: Given the overview report from the start-up phase of LiSCIA, the following steps are performed by the participants to derive architectural constraints.
   (a) Answer the LiSCIA questionnaire and note the architectural constraints by evaluating the component dependencies, namely draft constraints. These are represented in can-use/cannot-use phrases.
   (b) The draft constraints are verified by the expert resulting in the final architectural constraints.

Given the architectural constraints of a system, the following are the steps to discover the architectural violations from the derived constraints. Repeat the following until the expert identifies no more desirable features among the violations.

(1) Analyst runs Lattix to identify architectural violations on all versions of interest.

(2) The expert validates the architectural violations identified by Lattix in the above step to determine whether violations are in fact desirable features of the system, rather than violations.

(3) If any architectural constraints are not correct

  (a) Modify the constraints so that desirable features are no longer flagged as violations by Lattix.

The number of discovered violations in each version, i, is called net violations ($V_{net,i}$). The following are the quantities that are calculated from the list of validated violations.

- *New violations ($V_{new,i}$):* The number of unique violations that occurred in a version i, excluding the violations that occurred in the previous version, $i-1$. For the initial version of a system, the number of new violations is equal to the net violations.

- *Solved violations ($V_{solved,i}$):* The number of violations that are missing from the previous version, $i-1$. For the initial version of a system, the number of solved violations is equal to zero.

- *Reoccurred violations ($V_{reoccur,i}$):* The number of solved violations in the previous versions that reappeared in the version i. For the initial version of a system, the number of solved violations is equal to zero.

The term 'decay' emphasizes time, we considered the development time of the systems using the released dates of the versions. The following terms are defined for a version i.

- *time (t_i):* The development time of version i.

- *Time (T_i):* The cumulative development time from the beginning of the project until the version i.

- *Code decay for version i (cd_i):* This is a measure of a type of code decay for a given version i since the last release.

- *Net code decay (CD_i):* For a version i, the net code decay, which is a measure of a type of code decay, is defined as the net violations divided by the cumulative development time from the beginning of the project when decay was zero.

- *Overall code decay (CD_n):* The value of overall code decay, which is a measure of a type of code decay, is calculated from the initial version to the final version.

To assess code decay over multiple versions, the analyst collects the data for net violations ($V_{net,i}$) for all the versions of a system from Lattix. Then according to the above definitions, the analyst calculates the value of code decay cd_i for version by version, net value of code decay CD_i for a version i, and the overall code decay CD_n of the system.

3 Case study

We selected an anonymous proprietary system for our case study. The roles of the participants in our case study are explained in Section 2. We limited our search to a proprietary system because accessibility to the expert (architect or team lead) of the system is an important element of our methodology. Other requirements are frequent short-term commits to the code repository, and compilable source code which is necessary to get all the runtime dependencies. The source code of the system was available in a repository to allow extracting the conceptual architecture and dependency structure matrix using Lattix. The versions of the target system were released and in production.

We selected 14 released versions of the system developed in the Java programming language from August 2008 to May 2014. The participants of this study were the software architect and a software developer for expert and evaluator roles respectively. The Java classes and interfaces grew from 367 to 935 classes and 27 to 101 interfaces over 14 versions respectively.

In this study, the roles of evaluator, expert, and analyst were fulfilled by different individuals. The expert role was fulfilled by the architect of the project. The expert had 8 years of experience in developing several software systems in Java. The evaluator, who participated in deriving the architectural constraints, had 9 years of programing experience in Java. The evaluator role was fulfilled by the GUI and business rules developer of the system. The researcher fulfilled the analyst role.

The case study procedure was interleaved with the research methodology procedure. The researcher asked participants to "think aloud" while drawing the high

level architecture and defining the components of the system. The researcher noted the discussions between the participants. Noting these discussions helped gather insights into the system in the case study. After deriving the architectural constraints, the researcher interviewed the participants to collect the demographic information, session questions, and feedback on the whole process. The researcher in the role of analyst collected architectural violations of different versions using Lattix.

4 Results and Analysis

We executed the methodology given in Section 2.

4.1 Derive architectural constraints

At the beginning of the case study, the analyst provided the artifacts of the first release to the participants (expert and evaluator). These artifacts included Java source code in the Eclipse IDE, a conceptual architecture, and a dependency structure matrix of the first release. The conceptual architecture and the dependency structure matrix were derived from source code by Lattix.

During the start-up phase of LiSCIA the participants drew the high level architecture of the given version. They referred to the conceptual architecture and the dependency structure matrix provided by the researcher. Then, the participants defined the components of the system. The source code of the software was divided into logical groups of functionality. The participants divided the system into six components: Graphical User Interfaces (GUI), input processing, persistence, security, utilities, and reporting.

For each component, the participants determined source files belong to it by defining a pattern on the directory and file names. In general, a single file should only be matched to a single component, but in this case, the architecture was not well defined for the project at the time of development, so the same name-pattern matches three different components. The name-patterns for the components are the following.

- GUI — `*.jsp`
- Input processing — `webapp/*/*/*.xml, *validator.java, validator.xml`
- Persistence
  1. Model — `model/*.java`
  2. Services — `services/*.java`
  3. Lucene — `dao/hibernate/*.java`
- Security — `security/*`
- Utilities — `util.*`
- Reporting — `reporting/*.java`, `reports/*.xml`

The participants listed the technologies they used in developing the system. They used MySQL, Hibernate, Spring, Maven, Lucene, Jackson (JSON), Apache Tiles, Apache Commons, Apache HttpClient, Apache Taglibs, Liquibase, C3PO, JASPR, Castor, OpenLDAP, DWR, JQuery, Ajax, Prototype, Script.aculo.us, DBUnit, Java.x.mail, Log4J, Display tag, SLF4J, Axix, CGLib, WSDL4J, JavaAssist, and DB2.

In the evaluation phase of LiSCIA, the participants used the information from the start-up phase to evaluate the architecture with a goal of deriving the architectural constraints. The participants mostly concentrated on the evaluation of component dependencies. To get the details of the dependencies, they used the dependency structure matrix given by the analyst. The participants discussed circular dependencies, unexpected dependencies, and which component depends on most of the other components. The participants listed the software architectural constraints represented by can-use or cannot-use phrases. They were services can use model, model cannot use services, utils can be used statically anywhere, GUI can use taglibs, taglibs cannot use GUI, taglibs cannot use services, services cannot use converters, framework cannot use services, webservices cannot use reporting, webservices cannot use taglib, webservices cannot use dwr, services can use dao, services can use dao.hibernate, dao cannot use services, dao.hibernate cannot use services, GUI cannot use services, GUI cannot use dao, GUI cannot use dao.hibernate, persistence cannot use services, dao cannot use services package, and genericHibernate class cannot use auditService class.

The only major architecture change occurred before the release of the eighth version. New architectural constraints were added at that point and no constraints were removed. The new constraints were enum cannot use services, enum cannot use dao, and enum cannot use listeners.

The above software architectural constraints were derived by the expert and evaluator. Then, they were interviewed by the researcher for their feedback on the session. Both the expert and the evaluator were very confident when deriving the architectural constraints they did not miss any constraints. The time taken to derive the constraints was 2 to 2.5 hours including the interview sessions. Once again, the constraints were reviewed by the expert to finalize them before discovering violations. They suggested that a formal

Table 1: Violations count over multiple versions of System

Version i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$V_{net,i}$	46	47	50	50	54	54	58	26	28	24	28	29	51	56
$V_{solved,i}$	0	4	3	0	0	0	0	44	0	4	0	0	1	9
$V_{new,i}$	46	5	6	0	4	0	4	12	2	0	4	1	23	14
$V_{reoccur,i}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0

example of LiSCIA would be helpful to derive the constraints more quickly.

The net violations ($V_{net,i}$) for all the versions were discovered by Lattix and the values of $V_{net,i}$ are given in Table 1. The class reference type of violations were the most numerous in various versions (about 40%). Method calls (virtual invocation) were second most numerous type of violation in the system. Null constructor, static, and extends types of violations were highest to lowest percentage of remaining violations.

Table 1 shows that the number of net violations also increased in the first seven versions of the system. Thereafter there was a sudden decrease in number of violations. There was a major architecture change before the eighth version which was created for both the application's framework enhancements and also for the initial development of a new business module called the Trade Services module. These framework enhancements were mainly aimed to reduce the number of classes and the number of lines of code by consolidating classes, and reusing classes.

One such example is the Conversion classes which convert the data encapsulated in an entity to a desired format. Initially the system had a `ConversionService` class for each entity, but the developers had to keep on adding classes or additional lines of code for different types of conversions for the same entity. So during the enhancements between versions 7 and 8, developers refactored the source code by deleting the `ConversionService` and opted for JSP tags to conditionally display the desired format for each entity.

In the eighth version, the outcome of the framework enhancements reduced the number of classes. But at the same time, the initial development of the Trade module brought in its own classes. Later on, a spurt in the number of classes was the outcome of this combined effort to enhance the framework and to add the functionality of the Trade module. The expert explained that the increase in the net violations from the twelfth version to the thirteenth version was due to new developers who worked on the release, the code review system was not strictly enforced, and limited time was allocated to design and analysis of the requirements.

4.2 Find new, solved, and reoccurred violations

After discovering the net violations ($V_{net,i}$), we manually compared these violations from one version to the next version and found the new violations ($V_{new,i}$), solved violations ($V_{solved,i}$), and reoccurred violations ($V_{reoccur,i}$). Table 1 shows the counts of $V_{new,i}$, $V_{solved,i}$, and $V_{reoccur,i}$ in all the versions. We assigned a unique ID for each violation to track the life cycle of that violation in different versions. In this study, once the violations were solved in a version, that violation did not reappear in later versions. Therefore $V_{reoccur,i}$ is zero for all versions in this case study.

In the release of eighth version, due to the framework enhancements, which included changes to the architecture there was an increase in the number of solved violations. Reorganization of several classes solved 44 violations. Table 1 shows that the new violations also increased during the eighth version, due to adding a new business module to the system. In the later versions, the violations were removed in the source code because developers refactored by deleting class references and virtual invocations.

4.3 Assess code decay over multiple versions

The released dates of the versions were collected from the revision control system. For a given version i, the number of working days since the last release, time (t_i) in work weeks, and the cumulative development time from the beginning of the project for each version (T_i), also in work weeks, were calculated. We then computed the code decay values for each version (cd_i), the net code decay (CD_i) using respectively. The values of t_i, T_i, cd_i, and CD_i of the system are given in Table 2. An ideal system follows all the architectural constraints without any architectural violations. Therefore the value of code decay for such system is 0 violations/week.

Figure 2 shows the code decay values (cd_i) for each release. The cd_i measurements in this graph can give a manager insight into the the process of software development. "Did the development of this version cause further code decay?" Before the major change in architecture of the system, the code decay value cd_i

Table 2: Development time and code decay values of System

Version i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Work days	412	12	12	10	16	36	40	440	36	34	17	28	240	122
t_i	82.4	2.4	2.4	2.0	3.2	7.2	8.0	88	7.2	6.8	3.4	5.6	48	24.4
T_i	82.4	84.8	87.2	89.2	92.4	99.6	107.6	195.6	202.8	209.6	213.0	218.6	266.6	291
cd_i	0.56	0.42	1.25	0.00	1.25	0.00	0.50	-0.36	0.28	-0.59	1.18	0.18	0.46	0.20
CD_i	0.56	0.55	0.57	0.56	0.58	0.54	0.54	0.13	0.14	0.11	0.13	0.13	0.19	0.19

t_i and T_i are in work weeks; cd_i and CD_i in violations per week

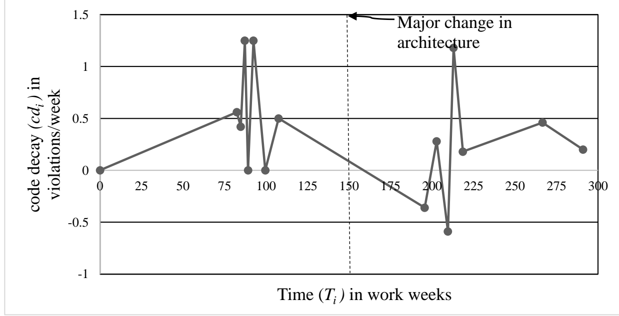

Figure 2: Code decay for each release in System

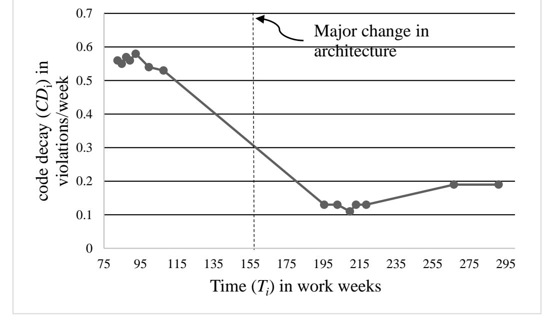

Figure 3: Code decay since beginning of the System

varied from 0 to 1.25 violations/week. When compared to an ideal system, the code decay value (cd_i) was positive because of increase in the number of new violations compared to solved violations. The expert explained that the developers focused on the functionality of the software and not on the architectural constraints.

After the change in architecture, the code decay values (cd_i) varies from -0.59 to 1.18 violations/week. When compared to an ideal system's code decay value of 0 violations/week, the code decay value (cd_i) of two versions after the change in architecture had negative values because there were more solved violations than new violations. This is because the developers concentrated on the enhancements. Figure 3 shows the net code decay (CD_i) values of each version from the start date of the project. The CD_i measurements in the graph give a manager insight into the status of the software product from its start date. "Is the product's average code decay worse or better than the past version?" This graph gives the status of the software product from its start date. The net code decay values are positive because it considers only net violations over the time period since the beginning of the project T_i. The net code decay value decreased after the change in architecture and was stable for later versions. For an ideal system, the net code decay value is 0 violations/week. The overall code decay (CD_n) of the system at the end of the study period was 0.19 violations/week.

5 Discussion

5.1 Results and Lessons Learned

We conducted a case study on a proprietary system. In this case study, we started our code decay assessment by deriving the architectural constraints. We applied Lattix and LiSCIA as a part of our methodology to derive architectural constraints. These constraints are represented by can-use and cannot-use phrases. The expert of the system validated our derived architectural constraints before and after discovering the architectural violations. The subjects who participated in our study had software development experience varying from 8 to 9 years. The participants were confident that they hit all the important constraints of packages or classes of the system. Our participants did not include any constraints regarding extensions of classes from the Spring framework since we were aware that several classes used external libraries. They spent more time in evaluating the components section of the review phase of LiSCIA to derive the constraints. They did not spend much time on other sections in the review phase of LiSCIA. Some of the participant's comments from the interview session are the following.

- One of the participants mentioned that deriving the rules was easier since they used a Model-View-Controller (MVC) framework in implementing their system.

- The start-up phase of LiSCIA helped participants to recall design decisions on other larger projects.

- Participants suggested having a formal example of LiSCIA to help speed the process of learning our methodology for deriving architectural constraints.

- It was difficult for participants to define the components of the system, which are logical functional units of the implemented system.

- They said that if they had followed the evaluation of components section in the review phase of LiSCIA while developing the system, they would have followed all the architectural constraints resulting in no architectural violations in the system.

Other researchers [5, 11] have analyzed the evolution of architectures by considering the early version's 'uses' relationships as their constraints or considering the architecture style rules as their constraints. However, they did not concentrate on recovering architecture constraints from the implemented system. We propose a methodology that uses LiSCIA and a reverse engineering tool to derive architectural constraints from the implemented architectures. Our results and qualitative analysis of the case study showed that the methodology was effective and required a practical level of effort for a moderate sized software system.

Our methodology includes the detection of architectural violations based on constraints derived using LiSCIA. We used Lattix as a part of our methodology to detect architectural violations. We validated the violations with the system experts. Violations may be resolved if the expert modifies the architectural constraints for any desirable features that were flagged as violations. In our case study, the experts did not choose to modify the constraints while validating the violations. We categorized the discovered violations and we found that 'class reference' type violations were the highest percentage among all the violations in our case study. The number of net violations $V_{net,i}$ decreased after changes in the architecture. Our results and qualitative analysis show that our methodology effectively detected and validated the architectural violations for a given list of constraints expressed by can-use and cannot-use phrases.

The net violations ($V_{net,i}$) were discovered using Lattix. Then, we found the $V_{new,i}$, $V_{solved,i}$, and $V_{reoccur,i}$ by comparing the $V_{net,i}$. The term 'decay' emphasizes time, we considered the development time of the systems using the released dates of the versions. Second, we computed the following code decay values in violations per week.

- Code decay for a version (cd_i) — This measure gives a manager insight into the process of software development. "Did the development of this revision cause further code decay?"

- Net code decay (CD_i) — This measure gives a manager insight into the software product from the beginning of the project. "Is the product's average code decay worse or better than the past version?"

- Overall code decay (CD_n) — This measure gives a value of code decay for the current system. "Does the current system have unresolved violations and what has been the average rate of violations, namely code decay?"

For the case study system, the values of cd_i and CD_i were fluctuating in the beginning of the project and decreased at the end of the final version of the study. The increase in $V_{new,i}$ and decrease in $V_{solved,i}$ increased the code decay values. This means the existence of violations in the system increased the code decay values and solving or repairing violations decreased the value of code decay. From the case study we observed that the developer-driven reasons for introducing architectural violations were that inexperienced or novice developers worked on a few releases, and developers focused on pure functionality of the system. The process-driven reasons for architectural violations were the code review system was not strictly enforced, limited time allocation to design and analysis of the requirements, change in requirements from the client, and the project release deadlines.

5.2 Threats to validity

Construct validity: In our case study we were limited to can-use and cannot-use architectural relationships. There are other kinds of architectural rules also. We chose calender time to calculate the rate for code decay rather than other measures of time (for example, level of developers' effort).

Internal validity: Case study does not control factors the way a controlled experiment does. Thus our research questions did not explore cause-effect relationships.

External validity: Our case study system is database intensive, has web browser interfaces and is used by government clients. Our case study results (architectural constraints, architectural violations, and values of code decay) cannot be generalized to all kinds of systems. However, our methodology and code decay measurement techniques can be replicated on other systems.

6 Conclusions and Future Work

Using Lattix and LiSCIA, we developed a method to derive architectural constraints. To evaluate the proposed methodology, we conducted a case study where we validated the derived architectural constraints with experts of the system. Our empirical results and qualitative analysis showed that the methodology was effective and required a practical level of effort for moderate sized software systems. Using Lattix we developed a method to discover architectural violations. To evaluate the proposed methodology, we conducted a case study where we validated the discovered architectural violations with experts of the system. Our results and qualitative analysis showed that the methodology was effective in detecting architectural violations for a given list of constraints that represent can-use and cannot-use rules. 'Class reference' was the major architectural violation category experienced in the case study software system. A large number of class reference violations in the software increases undesirable coupling which makes the system hard to maintain. New violations in a system increases the code decay values (cd_i, CD_i, and CD_n) and solving or repairing violations decreases the value of code decay. Degradation over multiple versions is key to the concept of "decay." This means that over time, the system becomes harder to change than it should be. This also illustrates that systems have various decay histories over time.

Researchers should conduct case studies on deriving architectural constraints using other architectural relationships such as composition of modules, aggregation of modules, and database relationships besides can-use and cannot-use relationships. Researchers could perform research to analyze the code decay of systems by considering the level of effort of the developers on the project instead of calender time.

References

[1] Ajay Bandi. *Assessing code decay by detecting architectural violations.* PhD thesis, Mississippi State University, December 2014.

[2] Ajay Bandi, Byron J. Williams, and Edward B. Allen. Empirical evidence of code decay: A systematic mapping study. In *Proceedings: 20th Working Conference on Reverse Engineering*, pages 341–350.

[3] Eric Bouwers. *Metric-based Evaluation of Implemented Software Architectures.* PhD thesis, University of Delft, January 2013.

[4] Eric Bouwers and Arie van Deursen. A lightweight sanity check for implemented architectures. *IEEE Software*, 27(4), 2010.

[5] Joao Brunet, Roberto Almedia Bittercourt, Dalton Serey, and Jorge Figueiredo. On the evolutionary nature of architectural violations. In *Proceedings: 19th Working Conference on Reverse Engineering*, pages 257–266, 2012.

[6] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[7] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proceedings: 2nd International Symposium on Constructing Software Engineering Tools*, 2000.

[8] L. Hochstein and M. Lindvall. Diagnosing architectural degeneration. In *Proceedings: 28th Annual NASA Goddard Software Engineering Workshop*, pages 137–142, 2003.

[9] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *Eighth IEEE Symposium on Software Metrics*, pages 77–86, 2002.

[10] Steffen M. Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings: 3rd International Symposium on Empirical Software Engineering Measurement*, 2009.

[11] Leonardo Passos, Ricardo Terra, Marco Tuilo Valente, Renato Diniz, and Nabor Mendonca. Static architecture conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 2010.

[12] Roseanne Tesoriero Tvedt, Patricia Costa, and Mikael Lindvall. Does the code match the design? A process for architecture evaluation. In *Proceedings: International Conference on Software Maintenance*, pages 393–401, 2002.

[13] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Proceedings: IEEE International Conference on Software Maintenance*, pages 306–315, 2012.