2019 IEEE International Conference on Software Architecture Companion (ICSA-C)

Applying Social Network Analysis Techniques to Architectural Smell Prediction

Antonela Tommasel
ISISTAN, CONICET-UNICEN. Argentina
antonela.tommasel@isistan.unicen.edu.ar

***Abstract*—As a software system evolves, the amount and complexity of the interactions amongst its components is likely to increase, which negatively affects the system design structure and also its quality. For instance, certain modules might become coupled due to a new user feature being added or to sub-optimal development decisions. Design degradation symptoms are often related to high coupling and unwanted dependencies, such as: cyclic dependencies or violations to design rules, amongst other architectural smells. Thus, the early detection of such symptoms is important for architects to: i) anticipate dependency-related design problems in different parts of the system, ii) assess possible situations of technical debt, and iii) proactively look for solutions to preserve the quality of the system. Although there are approaches that analyse design dependencies in code bases and flag smell occurrences, very few of them have dealt with the prediction of dependency relations amongst software components. This research hypothesises that a predictive approach can warn architects about dependency-related problems before they appear. To this end, a particular graph-based approach is social networks analysis (SNA), which has been used for modelling both nature and human phenomena. Specifically, SNA techniques can predict links that do not yet exist between pairs of nodes in a network. SNA applications have shown evidence that the topological features of dependency graphs can reveal interesting properties of the software system under analysis. Nonetheless, SNA techniques have not yet been extensively exploited in the Software Architecture community. In this context, the question that motivates this research is to what extent SNA can leverage on information from a software design (and its evolution over time) for inferring new dependencies and likely configurations of architectural smells out of those dependencies.**

I. Introduction and Motivation

The high development costs motivate the evolution and adaptation of existing systems, rather than the development of entirely new systems, in order to meet new requirements (e.g., a functional feature) and deal with environment changes (e.g., a need to run on a new platform or to integrate with other system). Nowadays, it is common for systems to be upgraded and maintained over spans of several years. To be successful, software evolution must be carefully managed and executed [11, 23]. However, as a software system evolves, the number and complexity of the interactions amongst its components is likely to increase, which negatively affects the system design structure, but also key system qualities such as: correctness, performance, reliability or maintainability. For instance, certain modules might become coupled in a given system version, and progressively make other modules to be dependent on each other. These dependencies might manifest at the architectural level as cycles amongst components, layer bridging, or hub-like components, among other problems.

The software architecture of a system describes the software elements of that system and their different types of relationships [7]. Furthermore, the software architecture comprises the set of design decisions that shape the solution, in order to satisfy the quality goals posed by the system stakeholders [14]. For example, the elements could be modules and the relationships could be usage relations between those modules. Design decisions for a module view can refer, for example, to a layering pattern or to modifiability principles of cohesion and coupling that the modules must adhere to. The architecture is crucial for software maintenance and evolution [17]. It has been shown that poor architecture choices are an important source of technical debt [7]. Nonetheless, the architecture can become less useful if its prescriptions are not reflected in the code. Since documented architectures are generally non-existing or outdated for many software projects, it is common for architects to rely on the implemented architecture for different analyses, such as the identification of architectural smells. In this context, this research focuses on the view of software architecture as realised in source code.

Architectural smells can be defined as a combination of architectural constructs often leading to system maintainability problems [11] or other quality problems. They manifest as violations of well-known Software Engineering principles and have repeatable forms that enable a potential automated detection. In the last decade, research has been devoted to study how smells are introduced, how they evolve and what their effect is on program comprehension [22]. However, research on how to predict the appearance of architectural smells, given the structural characteristics of systems, has been scarce.

The early detection of architectural smells is important for architects, so that they can plan ahead for actions to stop system degradation and preserve system quality. In this regard, several tools exist for assessing whether the right dependencies amongst system modules are in place, including LattixDSM, SonarQube or SonarGraph [13], amongst others. These tools normally extract dependency graphs from the source code to compute different metrics (e.g, average component dependency, or package cyclicity) that provide indicators of the system health. Some tools can also bring problems, such as architectural violations, cycles or other smells, to the architect's attention. A limitation of this approach is that such tools are only able to analyse the dependencies that exist in the system, meaning that problems are detected once they have appeared. In this situation, evidence suggests that developers can be

978-1-7281-1876-5/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSA-C.2019.00053

IEEE computer society

reluctant to fix problems once they are already in the code [1]. Some of the tools provide "just-in-time" detection capabilities, which might help developers to fix specific smells in a local design context. However, this capability is not always helpful when architects want to assess the different smells of the system with a global design perspective. In a forward-looking scenario, architects would want to know which modules are likely to be coupled in the near future, and which smells are more harmful for the system. This architecture-level analysis requires to anticipate dependency-related problems in order to proactively look for solutions.

In order to make predictions about software design dependencies, a particular graph-based approach is Social Networks Analysis (SNA), which refers to a strategy for analysing social structures through network and graph theories [16]. SNA studies to what extent the evolution of a network can be modelled by using features intrinsic to the network [16]. One particular task is known as Link Prediction (LP), and aims at inferring missing links between two nodes in a network based on the observable interactions (or links) amongst nodes and node attributes, when available. Due to its relevance in different domains, several techniques have been proposed to solve the LP problem, usually based on graph structural features that assess similarity between pairs of nodes. As software architectures comprise components that interact with each other, it is natural to model architectures as graphs. In fact, a module view can be modelled as a dependency graph, in which nodes represent modules and directed edges represent usage relations. These relations can be inferred from the code via static analysis. However, software design networks could have a different dynamic from traditional social networks.

The motivations for this research come from the lack of studies investigating the potential of SNA not only for describing software design structures, but also for predicting their evolution or properties. The analysis then would not just be limited to characterising design elements (e.g. packages, components) and their dependencies, but to how design information can be used to provide insights about future system versions. This knowledge would support predictive capabilities for architects' decisions regarding system refactorings or other strategies for dealing with technical debt in the system. In this context, *the goal of this research is to use SNA techniques along with information from current and past design structures of a system for inferring possible architectural smells in terms of configurations of design dependencies*.

II. Research Questions

This research departs from the hypothesis that software systems and their underlying architectures behave, to some extent, as social networks. Therefore, it is possible to predict some types of changes in the system design structure based on the appearance of dependencies amongst design elements. In particular, the focus is on changes that result in problematic configurations, such as dependency-based architectural smells [15]. This hypothesis poses several research questions.

- ***RQ1***. *How do architectural smells evolve over system versions, in terms of increasing or decreasing their dependency configurations?*

For instance, given a package cycle at version t, it can be analysed whether the cycle will grow in successive versions (due to additional dependencies being added to the smell). A bigger cycle could indicate that the smell got worse, and thus, its prediction is worthwhile. Some initial findings about cycles in Java applications suggest this trend [19]. Also, the causes for cycles getting better (or worse) due to both existing and new dependencies should be tracked. A similar analysis can be performed for other architectural smells.

- ***RQ2.*** *What criteria are useful for assessing similarity of design elements with respect to link prediction?*

Traditional LP techniques are primary based on determining whether two elements are likely to interact in the future based on a similarity score. This score is supported by the homophily principle, which states that similar elements are more likely to interact than dissimilar ones. In a software design context, homophily would mean, for example, that similar modules are more likely to establish dependencies than dissimilar modules. However, given that homophily was inspired by human relations, it remains to be evaluated whether it applies to design structures. For instance, two similar modules can intentionally be designed to not become dependent on each other, based on business logic or modularity reasons. Conversely, dependencies might still appear between dissimilar modules. Thus, there is a need to adjust the homophily principle and learn "exceptions" for software designs.

- ***RQ3***. *Can past system versions affect, and improve the predictions of, the design structure of a future version?*

In principle, the prediction of the appearance of dependencies or architectural smells can rely on characteristics of the current system, such as: structural design aspects, or features of individual design elements. Nonetheless, previous system versions can provide further information for the predictions. This is so because LP techniques often learn both from existing and non-existing relations amongst graph elements. Some of these relations come inevitably from the system history. In fact, there are several studies about the influence of past system versions on phenomena observed in the present system [24]. In particular, the architectural violations for a given system version can be inferred from problems in previous versions [18].

- ***RQ4***. *To what extent Machine Learning techniques can aid in the prediction of architectural smells?*

The LP problem can be cast as a classification problem in which a prediction model is built based on different information taken from the graph. In this setting, the classification model learns about both the existence and absence of relations between the different pairs of nodes in the dependency graph. These types of techniques allow to further leverage on the system history to make predictions, for instance, to deal with alternatives to the homophily principle discussed above. Nonetheless, predictions are probably not going to be perfect. In this regard, the technique might predict dependencies that

are not actually going to exist, or be unable to predict actual future dependencies. In other words, a simple detection of predicted dependencies might not be sufficient to determine how architectural smells will behave on the future. For instance, if two usage relations for modules are to be predicted, it does necessarily mean that a new (non-existing) cycle will be closed. For this reason, each type of smell requires to consider the predicted dependencies in context.

III. Related Works

Social networks can covey different meanings. Often, they represent social media and social networking sites, such as *Facebook* or *Twitter*. From a broader perspective, social networks refer to the graph structures comprising social actors and their connections, thus reflecting the roles of actors and the patterns and anti-patterns of relations. In this sense, SNA has emerged as an interesting and challenging research field [27].

In the Software Engineering domain, social networks can be extracted from the archives of a software project to better understand the project, the ecosystem, or different software engineering practices [12]. This social perspective can be effective for shedding light on the technical aspects of the work. For example, focusing on the people involved in the engineering processes, it has been studied how developers collaborate, the interplay of dependencies amongst technical artifacts and their creators, or how decisions about organizational structure affect the social structure of developers. Other works have used social networks for representing software dependency graphs in order to study software evolution [28, 6], defects, bugs [30], and vulnerable components [21]. Most works are of a descriptive nature, i.e. the analysis is made with a full knowledge of the software structure and there is no intent of predicting future system states.

As regards software evolution and low-level design, Terra et al. [28] used LP techniques to predict the package in which a particular class should be located during a refactoring process. Bhattacharya et al. [6] identified a set of global graph metrics to describe dependency graphs and determining which components one should debug, test or refactor first. Nguyen et al. [20] studied the performance of topological similarity metrics to predict bug proneness. Finally, Zimmermann and Nagappan [30] concluded that metrics assessing the local neighbourhood of nodes contribute better than global metrics to the understanding of software systems. Additionally, they stated that network metrics are better descriptors of software defects than source code metrics. However, SNA techniques have not yet been exploited for the prediction of software dependencies at the design level. Instead, prediction has been traditionally based on domain information or object-oriented metrics. For example, Aryani et al. [3] proposed a source code independent approach based on domain information from user manuals and help documents for predicting predict design dependencies in the current version system.

One of the symptoms of architectural degradation and even technical debt is the appearance of smells affecting the overall maintainability of software systems. Amongst the set of architectural smells, those based on class or package dependencies are of special interest to this research. Several automated smell detectors have been proposed, which differ in the algorithm, the applied heuristics or rules, and the metrics considered [13]. Nonetheless, detection techniques present some limitations [22]. First, the agreement between detectors is low, implying that the detectors return different smell instances according to the considered definitions. Second, some approaches require to specify the detection rules with domain-specific languages. Third, most detectors require the specification of thresholds to differentiate the smells, which affects their performance.

To overcome the limitations and providing a more objective detection, Machine Learning techniques have been adopted. For example, code smell detection using classifiers has been tackled inmostly based on training a classification model based on rules, or object-oriented metrics. The most notable advances were proposed in [2], in which an evaluation of 32 Machine Learning algorithms for detecting four types of code smells on 74 systems was presented. Classifiers were trained based on object-oriented metrics at method, class, package and project levels. The training smells were selected using freely available tools. Experimental evaluation showed a high classification performance, guiding the authors to conclude that Machine Learning is a feasible approach for smell detection. However, the approach presented some drawbacks that might affect the generalisation of results [22]. A first one is the characteristics of smells and non-smells. If the characteristics of smells are significantly different from those of the non-smells, then any Machine Learning technique could distinguish between both. This situation might not properly represent a real system in which the structural characteristics of smelly and non-smelly components might not differ. Second, each type of smell was individually analysed, thus rendering the classification task unrealistic as systems simultaneously comprise different types of smells. Third, evaluation considered a balanced scenario, which does not replicate the unbalanced nature of software systems in terms of the number of components affected by smells. Fourth, the authors selected independent metrics without analysing their correlation, which might lead to a biased evaluation due to over-fitting. Finally, Palomba et al. [25] characterised smells not only by source code metrics, but also by how the source code changes over time. Thus, the authors presented a technique based on change history informationand co-change of artifacts.

As previously stated, this research is interested in dependency-based architectural smells rather than on code smells. Undesired dependencies (e.g., those forming a cycle amongst modules) have been shown to be a possible manifestation of architectural debt. Thus, understanding the complexity of dependencies could help to better understand and conceptualise the debt. Along this line, Le et al. [15] hypothesised that architectural smells may be used to pinpoint the issue-prone parts of a system, even before stakeholders bring up the issues. To this end, the authors proposed an empirical study of architectural decay in open-source systems based on defining and analysing six types of smells. The selection of the smells

was guided by the key software engineering principles and the architectural aspects that were important to engineers, resulting in that five out of the six selected smells were dependency-related. The analysis confirmed that architectural smells have tangible negative consequences, requiring increased maintenance effort throughout a system lifetime.

Overall, some key differences can be identified between the presented techniques and the proposed research. First, the reviewed techniques have only focused on code smells, and have paid less attention to the detection of architectural smells. Considering that the ideas underlying code smells could also apply to architectural smells, it might be tempting to regard architectural smells as higher-level abstractions of code smells [15]. However, code smells may be detected and solved without knowledge of the architecture, which may not be the case of architectural smells. Second, techniques have focused on identifying smells that already exist in the system, instead of predicting which new smells are likely to appear in future system versions based on the system current state. Thereby, smell prediction would allow to anticipate the effect of current architectural decisions for preventing architectural decay. Third, smells have been mostly characterised based on system metrics. Generally, there are different information sources that could be exploited, and the choice depends on the kind of smells to be detected [24]. Conversely, this research bases the prediction on topological and lexical characterisations of systems, and on the evolution of such characteristics across different software versions.

IV. Research Methods

The methodology and modelling for this research is based on the SNA area [16]. Figure 1 schematizes the proposed approach for predicting architectural smells, from the data collection process (Section IV-A), in which dependency graphs are mined from the collected software repositories (Section IV-B), up to the dependency and smell prediction (Section IV-C). Finally, this section also describes how the performance of the predictions will be evaluated (Section IV-D).

A. Data Collection

The performance and effectiveness of the proposed prediction technique will be studied by considering Java open-source projects, in two steps. First, the performance will be evaluated on several Apache systems with more than 10 versions, and more than 10 contributors per version. Apache was chosen as it is one of the largest open-source organizations and it has produced well-maintained systems. Moreover, Apache systems have been already used in the literature for studying architectural smells [15]. Second, performance will be evaluated on other systems, not developed by Apache, to ensure the generalisation of results. At the moment, preliminary research has been conducted on Apache Camel, Apache Ant, Apache Derby and Apache Cxf.

As the prediction of smells involves analysing multiple system versions and the changes between them, it is necessary to select which versions to include in the analysis. For predictions to be made, consecutive versions should yield changes between already existing components. In turn, this selection also includes determining which type of versions (i.e. major, minor, patch or pre-release) and evolution paths to analyse. Nonetheless, determining the accurate evolution path of systems is not a trivial task. Behnamghader et al. [4] identified three frequently occurring patterns that can condition the version selection and evolution paths. Thus, following [4], in the case of major versions, it can be considered the sequence involving all minor versions (from the start of one major version to the start of the subsequent major version), representing the total changes received by the within a single major version. This implies the analysis of versions with extensive changes to system's functionality. Patches and pre-releases will be ignored as they mostly involve bug fixes, whilst pre-releases are merged into a posterior official version (either major or minor ones).

Systems versions will be crawled from their corresponding repositories. For each selected version, its source and compiled codes will be downloaded. Given the lack of publicly available datasets, which poses certain threats to validity, the analysis performed over the selected systems will be publicly stored. As a matter of fact, the processed software versions used in preliminary evaluations are available in a public repository[1].

B. Dependency Graph Representation

A prerequisite for applying LP techniques is to transform the system under analysis into a dependency graph that captures the architecture view under analysis. The focus will be on module views and three dependency-based architectural smells [15], namely: cyclic dependencies, hub-like nodes, and nodes with unstable dependencies. More formally, a dependency graph is a graph $DG\,(V, E)$, where each node $v \in V$ represents a module, and each edge (or link) $e\,(v, v') \in E$ represents a type of dependency from node v to v'. Since the research is oriented to Java systems, nodes correspond to packages while edges represent the usage dependencies between those packages. In principle, each system package can be regarded as a different module. Nonetheless, this assumption might not hold in all systems. For instance, different sub-packages might belong to the same conceptual module. Thus, only top-level packages will be identified as modules.

The dependency graphs for the different software versions are extracted based on the CDA tool[2] that works at the Java bytecode level and extracts high-level information about classes and packages in a human readable form. The tool also identifies the different types of dependencies (e.g. uses, implements, extends), and the package membership of classes. This processing allows to build package dependency graphs.

Dependencies can be characterised by different kinds of information, which could favour the predictions of particular dependencies or types of smells. To begin, this research focuses on topological and content-based information. On one hand, topological information is related to the graph structure (i.e., the package structure in this case) and the role that the nodes (and their edges) play in that structure [16], from either a

[1] https://github.com/tommantonela/icsa-2018

[2] http://www.dependency-analyzer.org

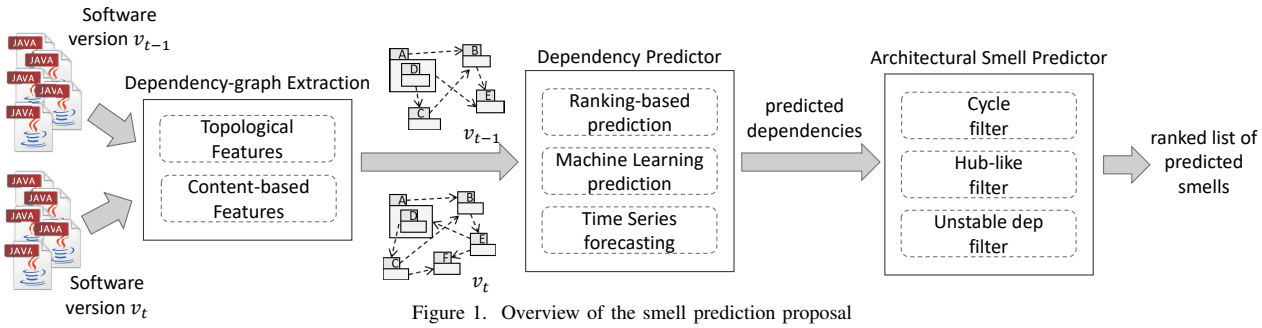

Figure 1. Overview of the smell prediction proposal

local or a global point of view. In this context, several features can be defined to assess the structural or topological similarity of two elements. For example, Common Neighbours is defined as the number of common adjacent nodes (i.e. neighbours) that two nodes have in common, aiming at capturing the notion that two disconnected elements who share neighbours would be "introduced" to each other.

On the other hand, content-based information provides an alternative (and complementary) similarity criterion to topological features. For example, one of the simplest strategies is to compute the lexical overlap between texts. Natural language processing routines are used to transform texts into their bag-of-words representations [26], which can be built by considering different aspects of the original texts. For instance, representations could be restricted to only the appearing nouns, adjective or verbs, or could choose to remove all punctuation. In the software domain, each Java class or package can be regarded as a bag-of-words containing the most representative tokens that characterise its source code, e.g. identifiers, methods' names and comments. According to Palomba et al. [24] lexical properties can help in the identification of linguistic anti-patterns that could be related to the existence of smells by distinguishing between nouns, adjectives and verbs, and identifying negative forms and dependencies between words (e.g. the relations between subjects and predicates). It is worth mentioning that the lexical characterisation of packages conveys information about the system architecture. For example, lexical and linguistic information has been used for "reconstructing" the responsibilities of the system, and thus reconstructing its implemented architecture [5]. Additionally, Corazza et al. [8] defined different types of linguistic information that convey different levels of relevance, and then contribute to an architecture reconstruction process.

C. Dependency and Smell Prediction

Based on the dependency graph and the chosen characterisation of dependencies, the link prediction task takes a $DG\,(V, E)$ at time t, and then infers the edges that will be added to $DG\,(V, E)$ at time $t + 1$. Let U be the set of all possible edges amongst nodes in $DG\,(V, E)$. Then, the link prediction task generates a list R of all possible edges in U−E, and indicates whether each edge (in R) is present in $DG\,(V, E)$ at time $t + 1$. For the purpose of this work, only dependencies between packages are considered. The bases of the prediction task include three alternatives with increasing complexity, which are discussed below.

- *Ranking-based Link Prediction.*

This is the simplest approach and follows directly from the homophily principle, giving a baseline for the research. In this alternative, all non-observed dependencies for a package p are ranked according to their scores. Due to homophily, the dependencies connecting more similar packages are supposed to have a higher appearance likelihood. Despite its simplicity, the definition of similarity is not trivial. As the metrics are defined based on diverse assumptions, it is expected that different metrics will lead to different predictive results. The output of this alternative for a given package p is a ranking with the top-N packages that are likely to connect to p.

This alternative assesses the structural similarity of packages by relying on different topological metrics that provide global or local information of the network [16]. The metrics to be considered are: Adamic-Adar, Common Neighbours, Katz Score, Resource Allocation, SimRank, and Sørensen. The best performing metrics from [28]: Kulczynski, RelativeMatching, and RusselRao, are also considered. In addition, content-based metrics will be included. Particularly, the Cosine Similarity scores for different bag-of-words representations (e.g. comments, method invocations, variable names) can be employed.

- *Machine Learning Link Prediction.*

In this alternative, a binary classifier is trained using the information provided by a number of graph versions. Whilst the previous alternative is solely based on the information of the existing dependencies, this alternative leverages also on the information provided by the absence of dependencies between pairs of packages. In this context, both the dependency graphs corresponding to the current and previous versions (named v_t and v_{t-1} respectively) are required as inputs.

To train a classification model, the dependency graphs are converted into a set of input instances, called dataset. Each instance consists of: a given pair of nodes, a list of features characterizing the pair, and a label that indicates whether a dependency exists between the nodes. In this context, the training set comprises instances belonging to the two system versions: existing dependencies in v_{t-1} (as instances of the positive class), missing dependencies in v_{t-1} (as instances of the negative class) and existing dependencies in v_t. It is worth noting that the information from the system versions

serves to train the classifier for properly learning instances of both the positive and negative classes. This mechanism allows to include information of dependencies in v_{t-1} that are guaranteed not to appear in the next version (v_t). If only one system were considered, no information regarding the negative class could be included in the model training, as it would not possible to guarantee that non existing dependencies are not going to appear in the next version. Similarly as for the previous alternative, the pairs of packages are characterised based on both topological and content-based features.

Once the model is trained, it predicts which dependencies that do not exist in v_t could appear in v_{t+1}. Note that these might not be the only dependencies that will be added in v_{t+1}, as dependencies between packages that did not originally exist in v_t could also appear. Although it is possible to predict that an existing package will depend on an unobserved package, it is not possible to determine what that unknown package will be. This means that only potential dependencies considering the packages already existing in v_t are considered.

It is worth noting that both the training and test datasets present the real class distribution of the system versions, as negative instances are not under-sampled nor artificial positive instances are introduced. This allows to keep the bias by dataset manipulation to a minimum. Classification could be performed using the Weka[3] implementation of the Support Vector Machine (SVM) algorithm, parametrized with a RBF kernel, which is useful for unbalanced datasets.

- *Time Series Forecasting.*

A time series is a sequence of values of a variable (or feature) taken at successive times (system versions in our case), often with equal intervals between them. One of the particularities is that data is not necessarily independent nor necessarily identically distributed, highlighting the importance of the order of the data. This alternative combines dynamic SNA with topological features to learn a robust model able to predict new links. In this case, each version represents a moment in time t. As in the previous alternative, for each system version different features are computed for each pair of packages. For estimating the scores for version v_{t+1}, the scores observed in previous versions (e.g. t_1 to t_n) are used as input for the technique. However, not every historical point might be of interest for the forecasting. For example, if a refactoring greatly affected the system structure, information from prior versions might not be representative of the actual system structure nor future ones. Hence, the performance of this alternative depends on the selection of the window of previous versions. Then, for each feature, time series forecasting models based on the scores of each of the known system versions are used to estimate the feature scores for the next system version. Finally, the estimated feature values are used to predict new dependencies. Predictions are based on a classifier trained with the last known version of the system, v_t. The forecasting was based on the Gaussian Processes model implemented in Weka.

The fact that any of the three alternatives above predicts whether an individual dependency is likely to appear is helpful, but it is not enough to predict the appearance of a new architectural smell, as not every predicted dependency might cause a new smell to emerge. Usually, the emerging smells are the result of a group of new and existing dependencies. Hence, for predicting the occurrence of a new smell, predicted dependencies should be filtered according to some criterion. It is worth noting that the prediction of dependencies might be affected by missed dependencies that should have been predicted or the prediction of mistaken dependencies. The filtering process is dependent on the type of smell to detect.

For example, a cycle filter considers only predicted dependencies that lead to the closure of new cycles in v_{t+1}. The predicted dependencies could be considered all altogether, and simultaneously added to graph at v_{t+1}, before checking for cycles. As another example, a hub filter could consider the nodes incidental to the predicted dependencies that fit with the hub definition in the literature. For each node that is incidental to at least one predicted dependency, all its actual and predicted dependencies are analysed together to compute the hub score of the node. Note that this strategy allows the detection of those nodes becoming hubs due to the addition of new dependencies, but disregards nodes that might become hubs due to changes in the overall structure of the dependency graph (i.e. nodes for which no new dependencies are added).

The described prediction and filtering alternatives foresee that all predicted smells are presented to the architect, which could result in mistaken prediction of some smells (false positives). Hence, once smells are predicted and ranked, it is necessary to define which of them are going to be presented in order to reduce the mistaken predictions. Nonetheless, choosing the number of smells to recommend might not be easy [29]. The simplest method is to set a fixed threshold and always recommend the same number of smells, based on: relevancy scores, a percentage of instances, or the number of predicted items. This method has several drawbacks. First, it ignores the characteristics of the task at hand. For example, it does not consider the history of smell appearance in previous versions. Moreover, a number higher to that of the actual elements to find could be chosen, with the corresponding decrease of precision, as inevitability mistaken recommendations will be made. Second, thresholds based on fixed relevancy scores might fail to acknowledge the possibility of rankings presenting different score distributions. For these reasons, the number of smells to predict will be chosen according to the history of discoverable smells in the previous versions.

D. Evaluation

The evaluation of the proposed prediction techniques is based on traditional evaluation metrics from the SNA area [29]. For all the alternatives presented, performance will be evaluated by comparing the predicted results with the real results of the next system version. In other words, predictions will be made over v_t and their predictive performance will be evaluated considering v_{t+1}. Given that the prediction of architectural smells builds on the prediction of individual dependencies, and the results of the dependency predictions can influence the smells prediction, the evaluation comprises two

[3]https://www.cs.waikato.ac.nz/ml/weka

steps: an evaluation of the quality of dependency predictions, and then an evaluation of smell predictions per se. As regards the dependency prediction, in the case of the ranking-based alternative, the quality of predictions will be evaluated in terms of precision (the percentage of actual dependencies that is discovered by the technique with respect to the total number of predicted dependencies), recall (the percentage of actual dependencies that is discovered by the technique with respect to the total number of new dependencies appearing in v_{t+1}), and the Normalized Discounted cumulative gain (NDCG) considering the top-N dependency predictions. As the actual number of dependencies to discover is known in advance, to reduce the sensitivity of results due to an inadequate selection of N, it is set to the 10%, 20%, 25% and 100% of the number of dependencies to discover. Then, the overall performance of the algorithm will be computed as the aggregation of the scores of the multiple packages for each list of length N.

As regards the other alternatives, in typical binary classification tasks, classes are expected to be approximately balanced, implying that the expectations for baseline classifier performance can be easily computed by traditional accuracy, precision and recall metrics. Nonetheless, classification problems that exhibit class imbalance (as the link prediction problem) do not share this property and the expectation for the classification metrics diverges for random and trivial classifiers. As a result, accuracy is problematic as its value approaches the perfect score for trivial predictors that always return false (in this case, meaning that the dependency or smell is not going to appear). At the same time, the correct classification of positive instances is more important as those instances represent the exceptional cases. When applied to the software design domain, it is more important to accurately determining that a few dependencies or smells will appear, that determining that hundreds of dependencies will not appear. In turn, this situation causes recall to be more important that precision, as it is preferable to detect the highest number of correct dependencies, at the expenses of also identifying some mistaken dependencies or smells to be analysed (and maybe discarded) by the architect. In this regard, the areas under ROC and PR curves are more adequate performance metrics [29].

The evaluation of the predictive capabilities of the three proposed alternatives will also allow to compare the descriptive power of each of the features selected for describing the pairs of packages. Particularly, it could be analysed whether topological similarity metrics are enough for assessing the similarity between software modules or it is necessary to consider also content-based features, amongst other possibilities. In turn, this exercise would allow to answer RQ2.

Regarding the prediction of architectural smells, performance will be evaluated based on precision, recall and PR curves. Similarly as for the prediction of dependencies, the correct identification of the positive instances (i.e. smells) is more important than that of negative instances (i.e. non-occurring smells). As a result, good recall is preferable over good precision. The smell prediction, in combination with the dependency prediction alternatives considering information from past system versions, would allow to study how design structures evolve and to what degree the system versions depend on the past structures and changes. Furthermore, the mistaken predictions could help to understand deviations from the expected design structure that could hint other configurations of smells. In turn, this exercise would allow answering RQ3. In summary, this final evaluation will point to answering RQ4 regarding the feasibility of Machine Learning techniques for predicting architectural smells.

At last, once the technique is refined and a tool is developed, it is expected to evaluate the performance and capabilities of the proposed techniques in real projects with architects.

V. Expected Contributions

The contributions resulting from this research are manifold. It builds on the advances of SNA and Machine Learning techniques to provide insights regarding how software design structures evolve, particularly in terms of degradation symptoms. The primary intended outcome of this research is a predictive approach that would allow architects to spot a set of dependency-related problems that are likely to appear in a given system. These problems are captured as architectural smells. Being able to anticipate architectural smells is important because architects can proactively look for solutions. In this regard, the approach could also allow simulating (dependency-related) decisions affecting the design structure to observe their potential effects in future system versions. Note that this kind of predictions differs from the checks provided by current tools for architecture conformance. The proposed approach provides a global view of several smells that might affect the design structure, while conformance tools usually take an operational perspective and aim at alerting developers about the appearance of local smells in the code. A a related line of research is to apply the predictive techniques for estimating dependency-related metrics of the system under analysis. In general, all these predictions contribute to assess and manage the architecture technical debt of systems.

To be useful for practitioners, as well as for assessing the relevance of recommendations in applications, the technique is planned to be integrated into a tool in the form of an Eclipse plugin. The tool should take as input the Java source code of at least two software versions, and produce as output a list of the predicted dependencies and a ranking of likely smells. In an initial prototype, the tool will provide support for three dependency-based smells (i.e., cycles, hubs, and unstable dependencies). The tool should also allow to compute and visualise traditional software metrics as a complement to the predictions. The user (architect) will have several configuration options, such as: the granularity of the analysis (dependency or smell), the information to consider in the analysis (topological, content-based or even software metrics), the technique for making predictions described in the previous sections, or whether to include software history in the analysis in the case the technique requires it. Furthermore, the user will be able to select components to exclude from the predictive analysis (e.g., third-party libraries or specific components that are not expected to be modified on the next version).

Each time the user runs the analysis, the source code will be assumed to represent a new system version. If predictions have already been made, this new version will be used as feedback to check whether predictions became true (i.e. the dependencies or smells that were predicted actually appeared). Once predictions are made, the user will be able to interact with them. The dependencies or smells will be ranked according to the confidence of the prediction (e.g. the probability given by the classifier). The user will be able to visualise the Java elements involved in each smell, the software metrics of the affected Java elements, and provide feedback about predictions. As it is expected that the predictions will not be perfect, this last type of user interaction will allow to introduce an additional feedback mechanism to adjust the techniques and prevent (future) mistakes.

VI. Research Agenda

The ongoing research has primary focused on the definition of the dependency graph and the evaluation of the dependency predictor. The definition of the dependency graph was complemented with a statistical analysis of software versions and the evolution of SNA metrics (tackling RQ1). In this line, it was analysed how past decisions reflected in the software structure affect the future occurrence of dependencies, and smells thereof (tackling RQ3). In addition, it was analysed the descriptive power of both topological and content-based features for defining the similarity of components (tackling RQ2). The descriptive power of software-related metrics remains to be evaluated. Regarding smell prediction, evaluations focused on defining preliminary filtering strategies for cycles and hubs (tackling RQ4).

At present, the research is oriented to perform a systematic study with more systems (and versions) to corroborate the initial findings reported in [9, 10]. Moreover, considering the errors in which learned models could incur, additional studies are being performed to introduce mechanisms of reinforcement learning to feed the predictions with new information obtained from the environment (or from the architect), as a means to establish the confidence of predictions and reducing the probability of false positives.

Once the prediction technique is fully evaluated, it will be incorporated into a tool to evaluate the capabilities of the approach in a study with subjects in the context of real software projects. Finally, it is expected to include other types of architectural smells [15] in the work.

References

[1] N. Ali, S. Baker, R. O'Crowley, S. Herold, and J. Buckley. Architecture consistency: State of the practice, challenges and requirements. *Empir Soft Eng*, 23(1):224–258, 2018. ISSN 1573-7616.

[2] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empir Soft Eng*, 21(3):1143–1191, 2016. ISSN 1573-7616.

[3] A. Aryani, F. Perin, M. Lungu, A.N. Mahmood, and O. Nierstrasz. Predicting dependences using domain-based coupling. *J Soft Evol Proc*, 26(1):50–76, 2014. ISSN 2047-7481.

[4] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empir Soft Eng*, 22(3):1146–1193, 2017. ISSN 1573-7616.

[5] A. Boaye Belle, G. El Boussaidi, and S. Kpodjedo. Combining lexical and structural information to reconstruct software layers. *Inf Softw Technol*, 74:1 – 16, 2016. ISSN 0950-5849.

[6] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th ICSE*, pages 419–429, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[7] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng. Design rule spaces: A new model for representing and analyzing software architecture. *IEEE Trans. Softw. Eng.*, pages 1–1, 2018. ISSN 0098-5589.

[8] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. Weighing lexical information for software clustering in the context of architecture recovery. *Empir Soft Eng*, 21(1):72–103, 2016. ISSN 1573-7616.

[9] J. A. Diaz-Pace, A. Tommasel, and D. Godoy. Can network analysis techniques help to predict design dependencies? an initial study. In *2018 IEEE ICSA-C*, pages 64–67, 2018.

[10] J. A. Diaz-Pace, A. Tommasel, and D. Godoy. [research paper] towards anticipation of architectural smells using link prediction techniques. In *2018 IEEE 18th SCAM*, pages 62–71, 2018.

[11] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *QoSA*, volume 5581 of *LNCS*, pages 146–162. Springer, 2009. ISBN 978-3-642-02350-7.

[12] M. A. Gerosa, D. Redmiles, P. Bjørn, and A. Sarma. Editorial: Thematic series on software engineering from a social network perspective. *JISA*, 6(1):23, 2015. ISSN 1869-0238.

[13] L. Hochstein and M. Lindvall. Combating architectural degeneration: A survey. *Inf. Softw. Technol.*, 47(10):643–656, 2005. ISSN 0950-5849.

[14] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120, 2005.

[15] D.M. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *IEEE ICSA, 2018*, pages 176–185.

[16] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):1019–1031, 2007. ISSN 1532-2882.

[17] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE ICSA*, volume 2, pages 69–78, 2015.

[18] C. Maffort, M. T. Valente, R. Terra, M. Bigonha, N. Anquetil, and A. Hora. Mining architectural violations from version history. *Empir Soft Eng*, 21(3):854–895, 2016. ISSN 1573-7616.

[19] H. Melton and E. Tempero. An empirical study of cycles among classes in java. *Empir Soft Eng*, 12(4):389–415, 2007. ISSN 1573-7616.

[20] T.H.D. Nguyen, B. Adams, and A.E. Hassan. Studying the impact of dependency network measures on software quality. In *2010 IEEE ICSME*, pages 1–10, 2010.

[21] V.H. Nguyen and L.M.S. Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th MetriSec*, pages 3:1–3:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0340-8.

[22] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th SANER*, pages 612–621, 2018.

[23] I. Ozkaya, A. Diaz-Pace, A. Gurfinkel, and S. Chaki. Using architecturally significant requirements for guiding system evolution. In *2010 14th CSMR*, pages 127–136, 2010.

[24] F. Palomba, A. De Lucia, G. Bavota, and R. Oliveto. Chapter four - anti-pattern detection: Methods, challenges, and open issues. volume 95 of *Advances in Computers*, pages 201 – 238. Elsevier, 2014.

[25] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Trans. Softw. Eng.*, 41(5):462–489, 2015. ISSN 0098-5589.

[26] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[27] S.H Strogatz. Exploring complex networks. *Nature*, 410(6825):268, 2001.

[28] R. Terra, J. Brunet, L. Miranda, M.T. Valente, D. Serey, D. Castilho, and R. Bigonha. Measuring the structural similarity between source code entities. In *25th International SEKE*, pages 753–758, 2013.

[29] Y. Yang, R.N. Lichtenwalter, and N.V. Chawla. Evaluating link prediction methods. *Knowl. Inf. Syst.*, 45(3):751–782, 2015. ISSN 0219-1377.

[30] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th ICSE*, pages 531–540, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.