ATDx: Building an Architectural Technical Debt Index

Roberto Verdecchia[1], Patricia Lago[1], Ivano Malavolta[1], and Ipek Ozkaya[2]
[1]*Vrije Universiteit Amsterdam, The Netherlands*
[2]*Software Engineering Institute, Carnegie Mellon University, USA*
{*r.verdecchia, p.lago, i.malavolta*}*@vu.nl, ozkaya@sei.cmu.edu*

Keywords: Software Architecture, Technical Debt, Software Analytics, Software Metrics, Software Maintenance

Abstract: Architectural technical debt (ATD) in software-intensive systems refers to the architecture design decisions which work as expedient in the short term, but later negatively impact system evolvability and maintainability. Over the years numerous approaches have been proposed to detect particular types of ATD at a refined level of granularity via source code analysis. Nevertheless, how to gain an encompassing overview of the ATD present in a software-intensive system is still an open question. In this study, we present a multi-step approach designed to build an ATD index (*ATDx*), which provides insights into a set of ATD dimensions building upon existing architectural rules by leveraging statistical analysis. The *ATDx* approach can be adopted by researchers and practitioners alike in order to gain a better understanding of the nature of the ATD present in software-intensive systems, and provides a systematic framework to implement concrete instances of *ATDx* according to specific project and organizational needs.

1 INTRODUCTION

Architectural Technical Debt (ATD) in a software-intensive system denotes architectural design choices which, while being suitable or even optimal when adopted, lower the maintainability and evolvability of the system in the long term, hindering future development activities [Li et al., 2015b]. With respect to other types of debt, e.g., test or build debt [Li et al., 2015a], ATD is characterized by being widespread throughout code-bases, mostly invisible [Kruchten et al., 2012], and of high remediation costs.

Due to its impact on software development practices, and its high industrial relevance, ATD is attracting a growing interest within the scientific community and software analysis tool vendors [Verdecchia et al., 2018]. Notably, over the years, numerous approaches have been proposed to detect, mostly via source code analysis, ATD instances present in software intensive-systems. Such methods rely on the analysis of symptoms through which ATD manifests itself, and are conceived to detect specific types of ATD by adopting heterogeneous strategies, ranging from the inspection of bug-prone components [Xiao et al., 2016], to the analysis of dependency anti-patterns [Arcelli Fontana et al., 2017], maintainability issues [Malavolta et al., 2018], and the evaluation of components modularity [Martini et al., 2018b]. Additionally, numerous static analysis tools, such as NDepend[1], CAST[2], and SonarQube[3], are currently available on the market, enabling to keep track of technical debt and architecture-related issues present in code-bases. Both academic and industrial approaches currently focus on fine-grained analysis techniques, considering *ad-hoc* definitions of technical debt and software architecture, in order to best fit their analysis processes. Nevertheless, to date, how to gain an encompassing overview of the (potentially highly heterogeneous) ATD present in a software-intensive system is still an open question.

In order to fill this gap, in this study we present *ATDx*, an approach designed to provide **data-driven, intuitive, and actionable insights on the ATD present in a software-intensive system**. *ATDx* consists of a theoretical, multi-step, and semi-automated process, concisely entailing (i) the inclusion of architectural rules supported by pre-existing analysis tools, (ii) the identification of outlier values (after normalization) via statistical analysis, and (iii) the aggregation of analysis results into a set of customizable ATD dimensions. *ATDx* is meant to serve two types of stakeholders: (i) *researchers* conducting quantitative studies on source-related ATD and (ii) *practitioners* carrying out software portfolio analysis and management, to suitably detect ATD items and get actionable

[1] `https://www.ndepend.com`
[2] `https://www.castsoftware.com/`
[3] `https://www.sonarqube.org`

insights about the ATD present in their systems according to their organizational and technical needs.

We discuss a set of benefits and drawbacks which characterize the current *ATDx* definition, which were identified from a viability investigation of *ATDx*. Among other, the most relevant characteristics of *ATDx* are: analysis tool and programming language independence, (ii) data-driven results at multiple levels of granularity, (iii) extensibility, and (iv) customizability to specific application domains.

The main contributions of the paper are: (i) the ***ATDx* approach** for building a multi-level index of ATD, (ii) the description of the **process for building an instance of *ATDx***, and (iii) a thorough **discussion** of the limitations and drawbacks of *ATDx*.

The remainder of the paper is structured as follows. In the next section, we present the theoretical framework underlying the *ATDx* approach. In Section 3 we document the steps entailed in the implementation of a concrete instance of *ATDx*. Section 4 discusses the main benefits and drawbacks of the *ATDx* approach, which emerged from a viability investigation. In Section 5 the related work to our study is presented. Finally, Section 6 reports on the conclusions of the study and the future work.

2 THE ATDx APPROACH

In this section we present the definitions on which the calculation of *ATDx* relies (Section 2.1), followed by the description of the approach (Section 2.2).

2.1 Definitions

Definition 1. Architectural rule. Given a source code analysis tool T and its supported analysis rules R^T, the *architectural rules* AR^T supported by T are defined as the subset of all rules $R_i^T \in R^T, i = \{1,\ldots,|R^T|\}$ such that:

- R_i^T is relevant from an architectural perspective;
- R_i^T is able to detect a technical debt item, i.e., a design or implementation construct that is expedient in the short term, but set up a technical context that can make future changes more costly or impossible [Avgeriou et al., 2016].

We consider every AR_i^T as a function $AR_i^T : E \rightarrow \{0,1\}$, where E is the set of architectural elements according to a granularity level (see below). In case that an element $e \in E$ violates rule AR_i^T, then $AR_i^T(e)$ will return 1, and 0 otherwise.

For example, a rule AR_i^T checking if the `java.lang.Error` is extended in a Java system is (i) architectural since it predicates on the high-level design of Java-based systems, and (ii) related to technical debt since the design of the Java Virtual Machine (JVM) assumes that `java.lang.Error` and its subclasses are managed only by the JVM.

Definition 2. Granularity level. Given an architectural rule AR_i^T, its granularity level Gr_i^T is defined as the smallest unit of the system being analysed which may violate AR_i^T, e.g., a class, a method, or a line of code. As an example, if we consider a rule which deals with cloned classes, its corresponding granularity level is "class".

The ad-hoc mapping of design rules to different granularity levels enables us to evaluate and compare the occurrence of rules violations across different software projects at a refined level of precision, instead of trivially adopting a single metric for the size of software projects for all the rules in AR^T, e.g., source lines of code (SLOC).

Definition 3. ATD Dimension. Given a set of architectural rules AR^T, the set of ATD dimensions $ATDD^T$ represents a set of clusters of architectural rules $AR_i^T \subseteq AR^T$ with similar focus. One architectural rule AR_i^T can belong to one or more ATD dimensions $ATDD_j^T \subseteq ATDD^T$ and the mapping between AR_i^T and $ATDD_j^T$ is established by generalizing the semantic focus of AR_i^T.

For example, if an architectural rule AR_i^T deals with the conversion of Java classes into Java interfaces, the AR_i^T could fall under the general "*Interface*" ATD dimension (for further details on the identified ATD dimensions and their establishment, refer to Section 3.2).

We use the 3-tuple $\langle AR_i^T, ATDD_j^T, Gr_i^T \rangle$, as the mapping of each architectural rule AR_i^T to its granularity level Gr_i^T, and one or more ATD dimensions $ATDD_j^T$. It is important to note that, while an AR_i^T can be associated to one and only one granularity level Gr_i^T, an AR_i^T can be mapped to multiple $ATDD_j^T$s, and vice versa.

2.2 Approach Description

ATDx aims to provide a birds-eye view of the ATD present in a software-intensive system by analyzing the set of architectural rules AR^T supported by an analysis tool T, and subsequently aggregating the analysis results into different ATD dimensions $ATDD^T$. Intuitively, starting from an already available large-scale dataset of AR^T executions $NORM^T$ and a System Under Analysis *SUA*, *ATDx* performs a statistical analysis of all elements in *NORM* for detecting anomalous occurrences of architectural rule violations in *SUA*. If the SUA exhibits an anomalous number of violations of a certain rule $AR_i^T \in AR^T$, then the corresponding ATD dimension $ATDD_i^T \in ATDD^T$ mapped to AR_i^T is incremented. In a sense, *ATDx* represents the synthetized facets of ATD (derived from

the architectural rules in AR^T), rather than the total ATD present in the system.

ATDx leverages the calculation of the number of architectural rule violations of a System S over the size of S in order to compare the occurrence of rule violations across projects of different sizes. Specifically, for each AR_i^T, we first calculate the set of violations of AR_i^T in a system by defining $AR_i^T(S)$ as the union of all violations of AR_i^T in S, i.e.,

$$AR_i^T(S) = \bigcup_{e \in Gr_i^T(S)} ar_i^T(e) \quad (1)$$

where $Gr_i^T(S)$ is the set of all elements e in S according to the granularity Gr_i^T (e.g., the set of all classes in a Java-based system), and $ar_i^T(e)$ is a function returning e if the element e violates the architectural rule AR_i^T, the empty set otherwise.

Subsequently, we calculate $NORM_i^T(S)$, defined as the normalized number of architectural rule violations $|AR_i^T(S)|$ over the total number of elements e according to granularity Gr_i^T, i.e.

$$NORM_i^T(S) = \frac{|AR_i^T(S)|}{|Gr_i^T(S)|} \quad (2)$$

where $|Gr_i^T(S)|$ is the cumulative size of S expressed according to granularity level Gr_i^T, and $|AR_i^T(S)|$ is the total number of violations of rule R_i^T (see Formula 1).

Once the $NORM_i^T(S)$ for S is calculated, we statistically detect if it exhibits an anomalous value. In order to do so, we require the set $NORM_i^T$, which contains the values of $NORM_i^T(S)$ for each software project belonging to a pre-defined set of software projects, i.e.,

$$NORM_i^T = \{NORM_i^T(S_1), \ldots, NORM_i^T(S_n)\} \quad (3)$$

where n is the total number of projects belonging to the dataset of software projects considered. Given the calculation of $NORM_i^T$, we can identify if $NORM_i^T(S)$ constitutes an anomalous measurement by comparing its value with the other ones contained in $NORM_i^T$. Specifically, given the set of values $NORM_i^T$ and the value of $NORM_i^T(S)$, we define the function *outlier* as: $outlier : X^n \times [0,1] \rightarrow \{0,1\}$, where $X = [0,1]$; *outlier* returns 1 if $NORM_i^T(S)$ is greater than the upper inner fence of $NORM_i^T$, and 0 otherwise.[4]

In order to provide an overview of the ATD dimensions of a system S, for each ATD dimension $ATDD_j^T \subseteq ATDD^T$ we define the value of $ATDD_j^T(S)$ as the average number of outlier values of the AR_i^T mapped to it, i.e.,

$$ATDD_j^T(S) = \frac{\sum_{i=1}^{n} outlier(NORM_i^T, NORM_i^T(S))}{n} \quad (4)$$

where n is the total number of rules in AR^T mapped to $ATDD_j^T$.

Finally, we also define an overall value $ATDx^T(S)$, embodying the overall architectural technical debt of S calculated via our algorithm, as the average value of all the defined ATD dimensions $ATDD^T$, i.e.

$$ATDx^T(S) = \frac{\sum_{j=1}^{n} ATDD_j^T}{n} \quad (5)$$

where n is the total number of ATD dimensions $ATDD^T$ considered.

3 ATDx BUILDING STEPS

In this section, we document the steps required in order to build *ATDx*. Specifically, the implementation process documented in this section is presented in a generic fashion, i.e., it is not bound to any specific analysis tool or technology. The described process can be performed by both (i) researchers investigating ATD phenomena, and (ii) practitioners analyzing their software portfolios. In fact, following the described theoretical steps allows to implement the instance of *ATDx* which best fits the specific technical, organizational, and tool-related needs considered.

Fig. 1 presents the main steps of the ATDx approach. Given an analysis tool T, five steps are required to build an instance of *ATDx*, namely: (i) the identification of the architectural rules belonging to AR^T, (ii) the formulation of the 3-tuples of the form $\langle AR_i^T, Gr_i^T, ATDD_j^T \rangle$, (iii) the execution of T on a set of already available software projects to form the dataset of $AR_i^T(S)$ measurements, (iv) the execution of the *ATDx* analysis on the constructed dataset, and (v) the application of the *ATDx* approach on the specific system under analysis (SUA).

3.1 Step 1: Identification of the AR^T set

The first step of the *ATDx* building process consists in the identification of a set of architectural rules AR^T on which the calculation of *ATDx* can be based. Specifically, given an analysis tool T and its supported analysis rules R^T, a manual inspection is carried out in order to assess if the rules qualify as AR^T according to the criteria presented in Definition 1. This process can be carried out either by inspecting the concrete implementation of the rules R^T under scrutiny, or by consulting the documentation of T, if available.

[4] If Q1 and Q3 are the lower and upper quartiles of a set of observations, respectively, then the upper inner fence lies at Q3 + 1.5(Q3 - Q1) [Frigge et al., 1989]. Informally, the upper inner fence is the theoretical value lying at the top of the upper whisker of a boxplot.

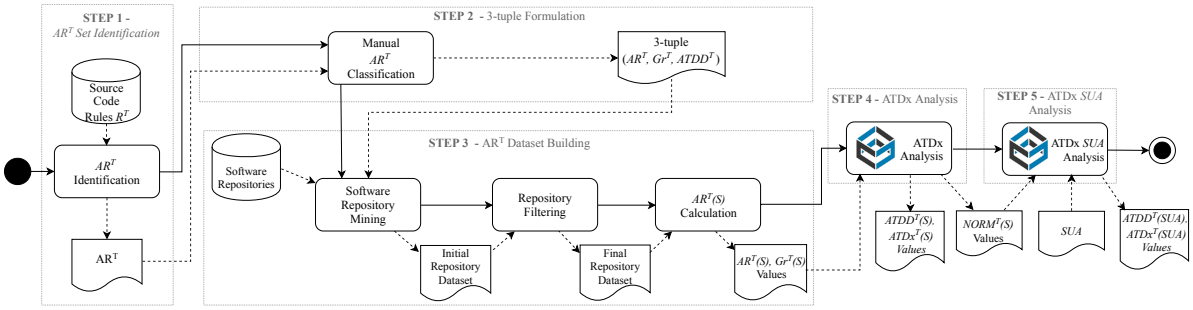

Figure 1: Overview of the ATDx process

3.2 Step 2: Formulation of 3-tuples $\langle AR_i^T, ATDD_j{}^T, Gr_i^T \rangle$

After the identification of AR^T, the 3-tuples $\langle AR_i, ATDD_j, Gr_i \rangle$ are established by mapping each AR_i^T to (i) one or more architectural technical debt dimensions $ATDD_j^T$ and (ii) its granularity level GR_i^T.

The process of mapping an AR_i^T to its corresponding architectural dimensions $ATDD_j{}^T$ is conducted by performing iterative *content analysis* sessions with open coding [Lidwell et al., 2010] targeting the implementation or documentation of the rule in order to extract the semantic meaning of the rule. More in details, once the semantic meaning of each rule is well understood, the AR_i^T under scrutiny is labeled with one or more keywords expressing schematically its semantic meaning. Such analysis is carried out in an iterative fashion, i.e., by continuously comparing the potential $ATDD_j^T$ associated to the AR_i^T under analysis with already identified dimensions, in order to reach a uniform final $ATDD^T$ set.

Similarly, the process of mapping an architectural rule AR_i^T to its corresponding level of granularity Gr_i^T is also carried out via manual analysis of the architectural rule, and subsequently identifying the unit of analysis that the rule considers (e.g., function, class, or file level).[5]

3.3 Step 3: Building the $AR^T(SUA)$ dataset

After the identification of the AR^T set (Step 1), it is possible to build the dataset of $AR^T(S)$ measurements. This process consists of (i) identifying an initial set of projects to be considered for inclusion in the dataset, (ii) carrying out a quality filtering process in order to ensure the soundness of the included projects, and (iii) identifying the $AR^T(S)$ sets and extracting the $|Gr^T(S)|$ values of each project included in the dataset. The selection of the initial dataset of projects to be considered for inclusion is a design choice specific to the concrete instance of *ATDx*. In other words, such choice is dependent on the analysis goal for which *ATDx* is adopted, the availability of the software projects to be analyzed, and the tool T adopted to calculate the $AR^T(S)$ sets. It is important to bear in mind that, given the statistical nature of *ATDx*, having a low number of projects in this step would not lead to meaningful *ATDx* analysis results (for more insights on this point see Section 4).

As for the selection of projects to be considered, the step of carrying out a quality-filtering process on the initial set of projects depends on the setting in which *ATDx* is implemented. In case that *ATDx* is deployed in an experimental setting, e.g., by considering open-source software (OSS) projects, this step has to be carried out in order to ensure that no toy software-projects, such as demos or software examples written for educational purposes, are included in the final dataset of projects [Kalliamvakou et al., 2016].

After the identification of a final set of projects to be considered for analysis, the $AR^T(S)$ sets are calculated for each system S in the dataset. The execution of such process varies according to the adopted analysis tool T. In addition to the $AR^T(S)$ calculation, during this phase also the values $|Gr_i^T(S)|$, calculated for each considered Gr_i^T, are extracted for each project S in the dataset, as such values will also be needed to carry out subsequent *ATDx* steps.

3.4 Step 4: ATDx Analysis

Once the $AR^T(S)$ and $Gr^T(S)$ sets are calculated for the whole dataset of projects, the analysis presented in Section 2 can be executed. Specifically, the *ATDx* analysis takes as input $AR^T(S)$ and $Gr^T(S)$ sets for a set of software projects, and outputs, for each project, its $ATDD^T(S)$ and $ATDx^T(S)$ values. It is worth noticing that, after a first execution of the *ATDx* ap-

[5] In the fortunate instance in which the GR_i^T mapped to AR_i^T is explicitly specified in the source from which the rules R^T are gathered, such information should be preferred over a manual inspection of the rule.

proach, it is possible to carry out further *ATDx* analyses on additional projects by relying on the previously formulated 3-tuples $\langle AR_i^T, ATDD_j^T, Gr_i^T \rangle$, and the precalculated intermediate values of the *ATDx* analysis $NORM_i^T$, as further documented in Algorithm 1).

3.5 Step 5: Applying ATDx to a SUA

Finally, after the execution of *ATDx* on all projects in the dataset, the resulting $ATDD^T$ and $ATDx^T$ values of a specific *SUA* can be computed.

Algorithm 1: Computing $ATDD^T$ dimensions and $ATDx^T$ value for a single SUA

Input: $SUA, AR^T, NORM^T, ATDD^T$
Output: $ATDD^T(SUA), ATDx^T(SUA)$

```
1  dimensions ← empty dictionary
2  atdx ← 0
3  for all dimensions j in ATDD^T do
4      dimensions[j] ← 0
5  for all rules AR_i^T in AR^T do
6      violations ← |AR_i^T(SUA)|
7      normalizedViolations ← NORM_i^T(SUA)
8      if outlier(NORM_i^T, normalizedViolations) then
9          dimensions[j] ← dimensions[j] + 1
10 for all entries j in dimensions do
11     dimensions[j] ← dimensions[j] /
       getNumRules(j)
12     atdx ← atdx + dimensions[j]
13 atdx ← atdx / |ATDD^T|
14 return dimensions, adtx
```

As shown in Algorithm 1, the computation takes as input 4 parameters: (i) the SUA, the set of rules AR^T, the values computed in the previous step $NORM^T$, and the set of dimensions $ATDD^T$. The output of the algorithm is the set of ATD values of the *SUA* $ATDD^T(SUA)$ (one for each dimension) and $ATDx^T(SUA)$. The $ATDD^T$ values provide support in gaining more insights in the severeness of the ATD according to the identified ATD dimensions, while the $ATDx^T$ value provides a unified overview of the relative ATD present in the SUA. After setting up the initial variables for the output to be produced (lines 1-2), the algorithm proceeds by building a dictionary containing an entry for each dimension in $ATDD^T$, with the name of the dimension as key and 0 as value (lines 3-4). Then, the algorithm iterates over each rule in AR^T (line 5) and collects the number of its violations, both raw (line 6) and normalized by the level of granularity of the current rule (line 7). If the normalized number of violations is an outlier with respect to the values collected for the whole dataset of projects (line 8), then the entry of the `dimensions` dictionary corresponding to the dimension of the current rule is incremented (line 9). For each dimension j we normalize its current value according to the total number of rules belonging to j in order to mitigate the potential effect that the number of rules belonging to the dimension may have, and increment the current $ATDx^T$ with the computed score (lines 10-12). Finally, the $ATDx^T$ value is normalized by the total number of dimensions supported by all AR^T rules (line 13) and both dimensions and *ATDx* values are returned (line 14).

4 DISCUSSION

In order get further insights into the viability of *ATDx*, we implemented a prototype of *ATDx*, built by considering a set of predefined SonarQube rules [Ernst et al., 2017], and a large-scale dataset of over 6.7K software projects. The raw data, analysis scripts, and results of such investigation, accompanied by a companion technical report, is made available online for further consultation.[6] Following, the benefits and drawbacks characterizing *ATDx*, and/or emerging from the prototype implementation, are reported.

4.1 Approach Benefits

4.1.1 Tool independence

ATDx is a tool-independent approach by design. This allows its future adopters to tailor it according to the specific development context considered, by basing the *ATDx* calculation on the analysis tools which are deemed more fitting. Specifically, the adoption of *ATDx*, rather than imposing a technological constraint its own tools and practices, is intended to leverage already deployed analysis tools, hence lowering the effort required for its adoption.

4.1.2 Language independence

ATDx allows combining the results of different language-specific analyses via the *ATDx* building process discussed in Section 2.2. This enables its adopters to aggregate analysis results into a single comprehensive overview of the ATD present in a software-intensive system, encompassing architectural components implemented in heterogeneous programming languages.

4.1.3 Tool composability

In our viability investigation we considered a single analysis tool, namely SonarQube. Nevertheless, the *ATDx* approach, thanks to its entailed abstraction and normalization steps, allows to aggregate analysis results gathered via heterogeneous tools, e.g., static *and* dynamic analyzers, in order to gain a simple, unified, overview of the ATD present in a software system.

[6] `https://github.com/ATDindeX/ATDx`

4.1.4 Multi-level granularity results

ATDx provides the means to inspect ATD issues at various levels of granularity. At the highest level of abstraction, the *ATDx* value, representing the total ATD present in a software-intensive system, enables to swiftly gain knowledge of the ATD present in a software system, and to compare its normalized value with the ones of other projects. At a more refined level of granularity lies the set of architectural technical debt dimensions $ATDD^T$. Such semantic decomposition provides the possibility to intuitively gain an overview of the ATD items of different nature which are present in the considered system. The $ATDD^T$ values enable adopters to effectively compare the ATD status across different projects and to visualize it, supporting the communication of related concepts and a deeper understanding of the root causes of ATD. Finally, at the finest level of granularity, the derived data utilized to build the overview can be analyzed. This provides the lowest level of the *ATDx* calculation, namely the normalized AR^T outlier values, in order to gain a deep, actionable, and data-driven guidance on where the architectural debt is rooted. This last level of granularity enables developers to pinpoint the precise location of the source of ATD.

4.1.5 Exstensibility

After the implementation of an *ATDx* instance, it is possible to add new rules to the existing AR^T set by re-executing Steps 1 to 4 of *ATDx* exclusively on the newly added rules. This accounts for scenarios such as the support of new rules added by tool vendors, or the inclusion *a posteriori* of an an additional architectural rule. Additionally, it is possible to include additional tools in the *ATDx* analysis by integrating the new results with the ones obtained from already adopted tools. This process is carried out by expanding the previously identified $ATDD^T$ dimensions, and possibly including new emerging ATD dimensions.

4.1.6 Data-driven

Instead of relying on predefined thresholds to identify outlier values of architectural rules, *ATDx* takes advantage of normalization and statistical analysis of data gathered from the analysis of a dataset of software projects. Hence, *ATDx* is not prone to design problems related to the definition of metric thresholds, as it identifies severe architectural rule violations based on the evaluation of empirical values collected from a multitude of (possibly third-party) projects.

4.1.7 Lightweight analysis for additional *SUA*

Once an initial dataset of $NORM_i^T$ values is collected, the evaluation of additional systems via *ATDx* is straightforward. In fact, as a dataset of $NORM_i^T$ values is already available, the analysis can be carried out exclusively on the new *SUA*, without the need to re-execute the analysis for all projects belonging to the dataset. Additionally, as *ATDx* is designed to leverage tools already deployed in a specific industrial setting, pre-computed data can be used as input for *ATDx*, enabling to carry out the analysis by investing a negligible amount of effort.

4.1.8 Domain-specific customization

As *ATDx* relies on the analysis of a dataset of software projects, rather than considering a single one, it is possible to tailor the approach by including in the dataset exclusively software projects belonging to a specific domain, e.g., safety-critical systems, mobile applications. In fact, *ATDx* analysis results may vary according to the application domain considered. Hence, to get more refined and accurate results, a curated dataset of software projects belonging to the same domain can be considered, enabling to fine-tune the detection of severe rule violations according to the specific domain considered.

4.2 Approach Drawbacks

4.2.1 Dataset dependence

Naturally, the *ATDx* approach is dependent, by definition, on a dataset of software projects. Hence, while numerous ATD analysis approaches require exclusively the *SUA* [Verdecchia et al., 2018], our approach needs instead various software systems in order to detect outlier values. This implies that the *ATDx* results of a *SUA* are only "relative" to the other projects considered, and are not representing an universal result. Additionally, to provide accurate *ATDx* analysis results, it is necessary to ensure the quality of the utilized dataset. In order to mitigate this latter drawback, a quality filtering process on a preliminary dataset of software projects should be executed (see also Section 3.3). Regarding the scale of the dataset, we plan to carry out quantitative and qualitative experiments by considering different domains in order to identify the minimum number of projects required to achieve saturation of results (i.e., when the inclusion of additional projects does not lead to statistically significant changes of results).

4.2.2 *ATDx* implementation validation

In order to verify the correctness of an implemented instance of *ATDx*, it is necessary to evaluate and eventually tune it in a real development context. This process entails (i) the selection of a set of software systems, (ii) their analysis via *ATDx*, and (iii) the inspection of the analysis results by means of focus-groups with the developers who implemented the selected software projects. As the focus of our study

is defining a generic approach through which specific instantiations of *ATDx* can be implemented, this process was deemed as out of scope for the *ATDx* prototype implementation carried out for this study.

4.2.3 Emphasis of outlier values

The design of *ATDx* relies on the identification of outlier normalized values of AR^T violations (i.e., values within the upper inner fence and the upper outer fence). Hence, the emphasis of *ATDx* is on *severe* ATD issues present in software-intensive systems. While in our *ATDx* viability investigation we observed numerous projects characterized by outlier values (4179/6706), such process could be deemed as "lossy" if a more fine-grained analysis is expected. To mitigate this potential drawback, it is possible to modify the approach by considering more inclusive strategies, e.g., by considering as "outliers" the values between the third quartile and the upper outer fence, or even mapping a discrete value according to the quartile associated to an $AR(S)$ value.

4.2.4 Balance of *AR* number mapped to *ATDD*

As the different *ATDD* values constituting *ATDx* are computed by considering distinct sets of ARs, it is necessary that the number of rules across the different sets is balanced. In fact, if the distinct sets exhibit notable differences in cardinality, the weight of underrepresented sets could lead to an unfair representation of an *ATDD*. This drawback has to be considered and mitigated while designing an *ATDx* instance, by carefully selecting ARs, considering their recurrence, relevance, and the cardinality of the mapped *ATDD*s.

4.2.5 Potential empirically unreachable *ATDD* maximum values

Reaching a maximum value in a certain dimension $ATDD_i^T$ is by definition theoretically possible, but empirically extremely improbable. By design, a system S reaches the maximum in one dimension if and only if it possesses outlier values in *all* ARs composing $ATDD_i^T$. If S possesses a maximum value of $ATDD_i^T$, this would indicate that S is characterized by exceptionally severe and recurrent issues in that dimension. In our viability investigation with SonarQube such project was not present for all dimensions. A possible heuristic to solve this potential drawback would be to rescale the values of a dimension i to the $[0, max_i]$ range, where max_i is the maximum value in the dimension i across all rules in AR_T mapped to i. Nevertheless, we refrained from such solution for the sake of clarity.

5 RELATED WORK

Numerous software analysis approaches have been proposed to detect ATD in software-intensive systems. Among the most prominent and current ones, the approach of Arcelli Fontana et al. [Arcelli Fontana et al., 2016, Martini et al., 2018a], [Roveda et al., 2018] focuses on the identification of ATD by analyzing dependency architectural smells, which could lead to the emergence of an additional *ATDD* dimension, namely *"Dependency"*. Similarly, Kazman et al. [Kazman et al., 2015, Xiao et al., 2016] [Cai and Kazman, 2017], analyzed ATD by inspecting antipatterns of semantically related architectural components, e.g., via the analysis of bug-prone components. In [Roveda et al., 2018], another ATD index is presented. Differently from our *ATDx*, this concentrates entirely on architectural smells, notably related to dependency violations. Finally, Le et al. [Le et al., 2018] reported on an empirical investigation of architectural decay via the analysis of 8 architectural smells of different nature. Interestingly, in a similar fashion to *ATDx*, the smell violations are evaluated by adopting interquartile analysis [Tukey, 1977]. Nevertheless, such analysis is carried out at an *intra* architectural rule level, and, in contrast to *ATDx*, values are not normalized per system-size. More ATD identification approaches are reported in a secondary study of Verdecchia et al. [Verdecchia et al., 2018]. Regarding the identification of metrics thresholds, similarly to *ATDx*, in [Alves et al., 2010] an interquartile strategy is adopted to identify the severeness of metric values. Differently, this study does not focus on ATD and, while adopting a system-size normalization strategy, it takes only one level of granularity (NCLOC). In a recent work, Ulan et al. [Ulan et al., 2019] proposed a software metric aggregation approach based on distribution. Our approach is different by (i) focusing specifically on outlier architectural rule violations, (ii) considering sizes according to distinct granularities, and (iii) clustering results into semantic dimensions.

6 CONCLUSIONS

In this study we presented *ATDx*, an approach designed to gain an encompassing overview of the source-code detectable ATD present in a software-intensive system. The *ATDx* approach concisely entails (i) the manual inspection of pre-existing source-code rules, (ii) the identification of outlier normalized values via statistical analysis, and (iii) the aggregation of analysis results into a set of ATD dimensions. As future work, we plan to address the identified drawbacks by refining our theoretical framework of *ATDx*. Additionally, we would like to enhance the current approach by including the temporal factor into our anal-

ysis. Subsequently, we plan to implement a concrete instance of *ATDx*, and validate it by conducting a large-scale study involving industrial partners in order to assess (i) the usefulness and actionability of the proposed index, (ii) its sensibility w.r.t. different analysis tools, and its (iii) performance when dealing with larger datasets. As stated in [Nord et al., 2012], ATD is a complex, heterogeneous, and multifaceted problem. Our approach contributes to advance the field towards establishing a holistic, encompassing, view of the ATD present in a software-intensive system.

ACKNOWLEDGMENTS

This material is partially based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM20-0240.

REFERENCES

Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., and Di Nitto, E. (2017). Arcan: A tool for architectural smells detection. In *IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 282–285. IEEE.

Arcelli Fontana, F., Roveda, R., and Zanoni, M. (2016). Tool support for evaluating architectural debt of an existing system: An experience report. In *Annual ACM Symposium on Applied Computing*, pages 1347–1349.

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. (2016). Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Cai, Y. and Kazman, R. (2017). Detecting and quantifying architectural debt: theory and practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion*, pages 503–504. IEEE.

Ernst, N. A., Bellomo, S., Ozkaya, I., and Nord, R. L. (2017). What to fix? distinguishing between design and non-design rules in automated tools. In *IEEE International Conference on Software Architecture (ICSA)*, pages 165–168.

Frigge, M., Hoaglin, D. C., and Iglewicz, B. (1989). Some implementations of the boxplot. *The American Statistician*, 43(1):50–54.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2016). An In-depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering*, 21(5):2035–2071.

Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., and Shapochka, A. (2015). A case study in locating the architectural roots of technical debt. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 179–188. IEEE.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21.

Le, D. M., Link, D., Shahbazian, A., and Medvidovic, N. (2018). An empirical study of architectural decay in open-source software. In *IEEE International Conference on Software Architecture (ICSA)*, pages 176–185.

Li, Z., Avgeriou, P., and Liang, P. (2015a). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.

Li, Z., Liang, P., and Avgeriou, P. (2015b). Architectural technical debt identification based on architecture decisions and change scenarios. In *Working IEEE/IFIP Conference on Software Architecture*, pages 65–74.

Lidwell, W., Holden, K., and Butler, J. (2010). *Universal Principles of Design*. Rockport Pub.

Malavolta, I., Verdecchia, R., Filipovic, B., Bruntink, M., and Lago, P. (2018). How maintainability issues of android apps evolve. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE.

Martini, A., Fontana, F. A., Biaggi, A., and Roveda, R. (2018a). Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In *European Conference on Software Architecture*, pages 320–335. Springer.

Martini, A., Sikander, E., and Madlani, N. (2018b). A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. *Information and Software Technology*, 93:264–279.

Nord, R. L., Ozkaya, I., Kruchten, P., and Gonzalez-Rojas, M. (2012). In search of a metric for managing architectural technical debt. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 91–100.

Roveda, R., Arcelli Fontana, F., Pigazzini, I., and Zanoni, M. (2018). Towards an architectural debt index. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 408–416. IEEE.

Tukey, J. W. (1977). Exploratory Data Analysis. *Reading, Addision-Wesley*, 688.

Ulan, M., Löwe, W., Ericsson, M., and Wingkvist, A. (2019). Towards meaningful software metrics aggregation. In *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop*.

Verdecchia, R., Malavolta, I., and Lago, P. (2018). Architectural technical debt identification: The research landscape. In *IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 11–20.

Xiao, L., Cai, Y., Kazman, R., Mo, R., and Feng, Q. (2016). Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, pages 488–498. ACM.