Architectural Technical Debt Identification: Moving Forward

Roberto Verdecchia*†
*Gran Sasso Science Institute, L'Aquila, Italy
†Vrije Universiteit Amsterdam, The Netherlands
roberto.verdecchia@gssi.it

***Abstract*—Architectural technical debt is a metaphor used to describe sub-optimal architectural design and implementation choices that bring short-term benefits to the cost of the long-term gradual deterioration of the quality of software. Architectural technical debt is an active field of research. Nevertheless, how to accurately identify architectural technical debt is still an open question. Our research aims to fill this gap.**

We strive to: (i) consolidate the existing knowledge of architectural technical debt identification in practice, (ii) conceive novel identification approaches built upon the existing state of the art techniques and industrial needs, and (iii) provide empirical evidence of architectural technical debt phenomena and assess the viability of the conceived approaches.

***Keywords*- Software Architecture; Technical Debt; Software Maintenance;**

I. Introduction

In software development processes, a high number of heterogeneous (and sometimes conflicting) goals have to be considered. Fulfillment of functional requirements, adherence to quality standards, time to market (TTM) and budget management are among the constraints that steer the administration of development processes. This leads to the establishment of tradeoffs in order to deliver software products by meeting the prefixed goals.

Architectural technical debt (ATD) is a metaphor, relating a software engineering phenomenon to economic debt, used to describe sub-optimal decisions resulting in the conceivement of immature architectural artifacts [1]. During the development phases, software architecture plays a crucial role in the implementation of software systems [2]. Hence it can also lead to the introduction of high impact ATD Items (ATDIs). When ATDIs are neglected, software-intensive systems tend to slowly deteriorate through time, often leading to obsolete or even failing system.

To date, how to accurately identify ATDIs is still an open question. In particular, a challenge lies in the avoidance of *ad-hoc* identification approaches and the generalizability of results. Technical debt results to be particular sensitive to context [3], which by itself is difficult to analyze if a well-defined model for contextual analysis is missing [4]. Due to its abstract nature, lack of research, and scarcity of tools, ATD is regarded by Kruchten et al. as one of the most challenging TD to be uncovered [5]. Most of the current methods aiming to identify ATDIs rely on summary code dependency analyses, or interviews with software architects [6].

II. Goal

In order to further the identification of ATDIs, it is vital to identify what information of software systems is already available, and how this can be used to identify ATDIs. By developing methods utilizing pre-existing software artifacts, it is possible to conceive methods for the evaluation of ATD that are cost- and time-efficient. *The ultimate goal of our research is to understand how to efficiently and effectively identify ATDIs present in software-intensive systems.*

The conceivement of (semi-)automated approaches will enable us to utilize as subjects for extensive empirical experimentation Open Source Software (OSS) artifacts, which can be easily accessed through version control repositories. In particular, by adopting approaches based on repository mining, it becomes possible to carry out analyses characterized by being non-intrusive. Such characteristic is particularly interesting when industrial contexts are considered, as the potential chance of adoption result to be much higher w.r.t. intrusive approaches (e.g. based on structured interviews). In addition, it becomes possible also to carry out our analysis processes in an iterative fashion, enabling us to efficiently consider also the temporal dimension, which results to be a crucial aspect of ATD [7]. We identify three research questions underlying our research, namely:

RQ1: *"Do modification summaries, commit log messages, issue trackers and other software artifacts of version repositories provide ATDI related information that cannot be derived from source code analysis alone?"*

$H1_0$ (*null* hypothesis): Artifacts of version repositories contain the same ATDI related information that can be found by exclusively analyzing source code.

RQ2: *"Which ATDI can be identified automatically from artifacts of version repositories?"*

$H2_0$ (*null* hypothesis): "No ATDI can be identified automatically from artifacts of version repositories"

RQ3: *"Which ATDI tend to require additional human input to be identified?"*

$H3_0$ (*null* hypothesis): "There are no ATDIs which tend to require more often human input "

III. Methods

As stated by Martini et al. [8] source code metrics might not be sufficient to convey information at the architectural level. In fact, tools focusing on source code alone appear to be able to identify only a small subset of TDIs. Hence code analysis tools alone are not sufficient to identify ATD. In the majority of the cases, ATD is not directly related to code and its intrinsic qualities, but is rather involving structural or architectural choices and technological gaps [5]. Nevertheless, it is possible to lend metrics and approaches designed for code TD and port them to a higher level of abstraction. This can lead to the conceivement of different strategies that can be adopted to identify ATD, namely:

(i) Self-admitted ATD: This results to be one of the most consolidated approaches of TDI identification through repository mining techniques [9]. The intuition is to explore code bases in search of portions of code mapped to a comment where developers ("self-admittedly") document the presence of TDIs. Being able to identify automatically such instances can lead to a more efficient overview of the current state of TD in systems and potentially lie the groundwork towards its management. A similar approach can be considered in order to process other software artifacts (e.g. repositories of models) or to extract self-admitted ATDI from code bases.

(ii) Abstracted code evolution analysis: This method, appositely conceived for this research project, entails the extraction of architectural information from code bases and its subsequent analysis to identify architectural deterioration. Specifically, subsequent commits of a software repository are considered by analyzing the most important architectural changes, potentially through an evolution analysis of reverse engineered models of the software architecture. A possible approach entails a compliance checking through model comparison between implemented and intended architecture [10]. Existing tools, such as NDepend[1] or Scitools Understand[2], can be utilized to support this strategy.

(iii) Combination of multiple sources: An innovative idea entails gathering information from multiple sources, e.g. source code, commits, user reports, documentation, knowledge markets etc., in order to identify where ATDIs are located in software systems and where these items are discussed upon. By considering multiple sources where ATDIs are mentioned, it is possible to abstract from the source code and identify ATDIs which cannot be found by code analyses alone. Such methods may involve the parsing of issue trackers, questions and answer sites (e.g. Stack Overflow), documentation and other software artifacts where developers report issues relating to the architectural level. Results can then be validated through semi-structured interviews with software developers. The preliminary goal of this approach is to establish a taxonomy of ATDIs.

[1]https://www.ndepend.com/. Accessed 10th April 2018

[2]https://scitools.com/. Accessed 10th April 2018

IV. Envisioned results

Our envisioned contribution resulting from this research is manyfold. It builds upon a comprehensive analysis of the state of the art in ATD identification techniques [6]. This revealed the need for a clear and operational definition of the types of architectural items relevant for technical debt.

Our ultimate result envisioned is the conceivement of efficient and effective approaches for ATDI identification in software-intensive systems.

A second target result is a set of instruments that allow to assess the impact of ATDIs on the overall quality of software-intensive systems. This requires serious experimentation that can focus on estimation, measurement, and/or prediction as validation strategy. We plan to adopt as experimental subjects heterogeneous case studies obtained both from industrial partners and OSS repositories.

When available, such instruments will provide the necessary data to rank the ATDIs according to their relative significance. This is the first step to get a reliable overview on where to invest for ATD elimination.

Ultimately, our aim is to equip software architects with the tools for ATD monitoring, e.g. through documentation, visualization, and communication means. This would provide a definite step forward towards a clear understanding of ATD, and its management.

References

[1] T. Besker, A. Martini, and J. Bosch, "A Systematic Literature Review and a Unified Model of ATD." IEEE, Aug. 2016, pp. 189–197.

[2] H. van Vliet, *Software engineering: principles and practice*, 3rd ed. Wiley, 1993.

[3] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[4] A. Bedjeti, P. Lago, G. A. Lewis, R. D. De Boer, and R. Hilliard, "Modeling Context with an Architecture Viewpoint," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 117–120.

[5] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[6] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural Technical Debt Identification: The Research Landscape," in *International Conference on Technical Debt (TechDebt)*, 2018.

[7] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, Mar. 2015.

[8] A. Martini, L. Pareto, and J. Bosch, "Towards introducing agile architecting in large companies: the CAFFEA framework," in *International Conference on Agile Software Development*. Springer, 2015, pp. 218–223.

[9] A. Potdar and E. Shihab, "An Exploratory Study on Self-Admitted Technical Debt." IEEE, Sep. 2014, pp. 91–100.

[10] R. Verdecchia, "Identifying Architectural Technical Debt in Android Applications through Compliance Checking," in *International Conference on Mobile Software Engineering and Systems*, 2018.