Identify Impacts of Evolving Third Party Components on Long-Living Software Systems

Benjamin Klatt[†], Zoya Durdik[†], Heiko Koziolek[*], Klaus Krogmann[†], Johannes Stammel[†] and Roland Weiss[*]
[*]Industrial Software Systems, ABB Corporate Research Ladenburg, Germany
[†]Research Center for Information Technology (FZI), Karlsruhe, Germany
Email: {klatt, durdik, krogmann, stammel}@fzi.de {heiko.koziolek, roland.weiss}@de.abb.com

***Abstract*—Integrating 3rd party components in software systems provides promising advantages but also risks due to disconnected evolution cycles. Deciding whether to migrate to a newer version of a 3rd party component integrated into self-implemented code or to switch to a different one is challenging. Dedicated evolution support for 3rd party component scenarios is hence required. Existing approaches do not account for open source components which allow accessing and analyzing their source code and project information. The approach presented in this paper combines analyses for code dependency, code quality, and bug tracker information for a holistic view on the evolution with 3rd party components. We applied the approach in a case study on a communication middleware component for industrial devices used at ABB. We identified 7 methods potentially impacted by changes of 3rd party components despite the absence of interface changes. We further identified self-implemented code that does not need any manual investigation after the 3rd party component evolution as well as a positive trend of code and bug tracker issues.**

I. Introduction

Integrating 3rd party components, also referred as COTS, has become a common approach in industrial software development aiming to satisfy expectations such as cost or time-to-market reduction. However, when a long-living software system needs to be maintained over several years, it is important to carefully reflect and continuously monitor the long-term impact of this integration. Individual release cycles of self-implemented software and 3rd party components, including discontinued 3rd party components, force the decision of upgrading to a new version or even searching for an alternative component.

Making such a decision is a challenging task due to the need of assessing the change impact as well as the expected improvement of the migration. In this paper, we present an approach which combines code and bug tracker analyses to estimate i) code change impacts, ii) source code quality, and iii) development reliability of 3rd party components. This approach supports decision making for 3rd party component scenarios. The approach enables the identification of potential sustainability problems arising from the use of 3rd party components and the derivation of mitigation strategies to properly handle them. We document our experiences from an industrial case study where we applied this approach to a communication middleware for industrial devices. The case study shows that the approach supports the assessment of migration decisions. It helps identifying possible quality issues, as well as source code that might be affected to reduce the required manual investigation.

Our approach uses static code analysis to identify explicit interface changes and internal code changes in 3rd party components. Afterwards, a dependency analysis identifies potential impacts on the self-implemented code. In addition, we use a problem-pattern-detection and bug tracker analysis to estimate the code quality and reliability of the 3rd party component's evolution. Opposed to the majority of the existing approaches in this area, such as [1], [2], and [3], we accommodate the trend to integrate open source components which provide access to their source code and project information. Our approach explicitly includes additional resources of open source projects, such as their source code or bug tracker.

The contribution of this paper is a combined 3rd party component analysis which is capable to (i) identify explicit and hidden semantic changes, (ii) source code quality issues, (iii) estimate the change's impact on self-implemented source code, and (iv) assess the development's reliability of 3rd party components.

The rest of this paper is structured as follows: Section II summarizes related work in this area. Section III describes our analysis strategies in more detail, followed by a summary of our experiences from the case study in Section IV before we provide our conclusion in Section V.

II. Related Work

The research area of 3rd party evolution impact analysis is embossed by two major trends of comprehensive approaches. The first encompasses approaches assuming a black box view on integrated components, such as Kotonya & Hutchinson [2], Voas [4], or Zheng et al. [3] also developed within ABB research. In the same manner as Kotonya et al., it follows the second major trend: Using an architecture description language (ADL) and process-based approach to manage evolving 3rd party components. Opposed to these approaches, our analysis copes with the trend of integrating open source components that provide access to source code and software management information with further possibilities for the impact and development reliability analysis.

In the area of source-code-based impact analysis, there are existing approaches using dependency analysis, such as Clarksen et al. [1] and Bohner [5]. However, they neither investigate in an explicit internal change analysis nor in a comprehensive analysis considering a general quality trend analysis to further support migration decisions.

Neamtiu et al. [6] presented a change detection approach considering changed method signatures as well as internal

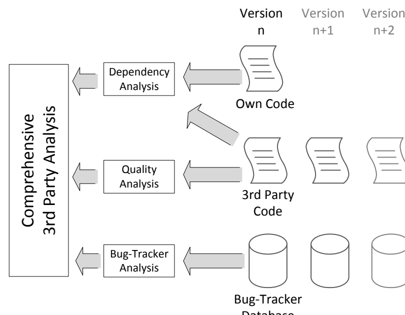

Fig. 1. Analysis Strategy Overview

(hidden) changes. The authors used this change comprehension for dynamic software updates. We use a comparable approach integrated in the SISSy tooling (Structural Investigation of Software Systems) [7] also used for the code quality analysis of the presented approach as described below.

III. Analysis Strategies

Our approach combines code-based dependency and code quality analyses with bug tracker analysis to reduce efforts and risks connected to the long-term co-evolution of 3rd party components. To provide a comprehensive evolutionary view on a 3rd party component, we combine the analysis of source code dependencies, code quality, and bug tracking information, and consider multiple versions of a 3rd party component as shown in Figure 1. The approach is designed to assess the migration to (1.a) a new version of a 3rd party component, (1.b) another 3rd party component with the same functionality, (2) assess the migration impact in terms of affected source code for 1.a/1.b, and (3) estimate development reliability trends.

To estimate the possible migration impact, we performed the same analyses on the version currently integrated, as well as on versions that are candidates to migrate to. The code analysis incorporates a dependency and a general quality analysis. The former analysis included the identification of interface changes as well as internal changes not reflected in them (i.e. potential semantic changes in the interface's implementation). These results were combined with the dependency analysis to identify possibly affected parts in the self-implemented code. The code quality analysis was then used to compare the 3rd party implementation with a quality benchmark (Seng et al. [8]) and to get a trend analysis of the 3rd party component's code quality evolution. With the same purpose, the analysis of the 3rd party bug tracker is used as an additional source of information about the evolution and reliability of the 3rd party component.

For the source code investigation, we use the SISSy tool (Structural Investigation of Software Systems) [7] also used by Seng et al. [8]. SISSy is a free tooling for the automated analysis and assessment of maintainability of object-oriented systems based on static analysis techniques. While there are many different tools providing analysis for coding issues, such as very large classes or dead code, we decided to use SISSy for a better comparison with the quality benchmark published by the Q-Bench project as described in Section III-B1.

As a limitation of the presented analysis, access to the 3rd party source code and bug tracker is important. While the former is easy to get for open source components and often for commercial components as well, the availability and maintenance state of the bug tracker often differs even for open source projects. However, depending on the actual scenario, even parts of the presented analysis can be performed to support the migration process.

In the following, we present details about the analysis with a focus on the dependency analysis and a brief summary of the others.

A. Dependency analysis

For source code dependency investigation we recommend three sub-analyses:

1) Analyzing dependencies of self-implemented code to understand coupling of internal and external components.
2) Detecting 3rd party method signature changes and their usage to identify code that might be affected.
3) Detecting internal changes of 3rd party methods with unchanged signatures to identify self-implemented code that is potentially affected.

1) Dependency Overview: The dependency analysis identifies internal dependencies between system components, and external dependencies between the system and 3rd party components. The number of dependencies reflects the propability of change propagation between them. Hence, the dependency analysis provides an indicator of which components might be affected by changes and with which probability. Central code elements of the system can be identified based on their number of dependencies. Changes to those central components should be done carefully, since such a change might lead to change propagation into some or all of the dependent modules. The considered types of dependencies are method calls, variable accesses and type references.

2) Impact of 3rd party signature changes: When using a 3rd party component, a problem to consider is whether updates to a new version propagate changes into the self-implemented code. The general procedure of the regarding analysis consists of two steps: 1) Identify signature changes in the 3rd party component, 2) Identify dependening code fragments of the self-implemented code.

In step 1, we compare static code models (i.e., generalized abstract syntax trees of the source code, represented in a database) of two versions of the 3rd party component by using a heuristic for identifying modified signatures. Methods are defined as equal if they have the same signature and are located in equal-named files. The signature consists of the name of the method concatenated with the list of its formal parameter types (e.g., methodX(Type1, Type2)). Hence, we determine a list of methods, whose signatures have not been changed from one version to another. We then invert this list to get the changed methods by taking all public methods and excluding the previously identified list.

A manual investigation of the results is required to identify mismatches, for example due to renaming or movement in

the directory structure. Afterwards, the list is reduced by the manually identified false positives.

In step 2, we calculate the direct and indirect callers of these changed methods based on the database representation of the source code. We extend the call-hierarchy until "'we cross the border"' to the self-implemented code (i.e., change propagation inside the 3rd party component is also considered). As a result, we get a list of methods residing in our code that are directly or indirectly dependent on signature-changed methods of the 3rd party library.

3) Impact of hidden 3rd party changes: The second dependency analysis aims at identifying code fragments that are affected by what we call hidden changes in a 3rd party component. Hidden changes are modifications of the logic within a method's body without modifications of the method's signature.

The general procedure of this analysis consists of two steps: 1) Identifying methods with internal but no signature changes, 2) Identifying dependent code fragments. In step 1, we compare the static code models of two versions of the 3rd party component by using a heuristic for identifying modified method bodies. Methods are defined to contain (hidden) implementation changes if they are recognized as corresponding in two versions (since they are in equal-named files and have equal signatures) but differ in number of statements or number of logic lines of code (no comments). This (extendable) heuristic also detects methods with restructurings without semantically changes. Nevertheless, it reduces the number of methods to be investigated in detail. Hence we determine a list of methods whose body has been changed from one version to another.

In step 2, we calculate the direct and indirect callers of these changed methods. Again, we extend the call-hierarchy until we "'cross the border"' to our code. As a result we get a list of methods residing in our code that are directly or indirectly dependent on hidden changes in methods of the 3rd party component.

B. Quality Analysis

For a complementary view on the 3rd party component and its evolution, we consider not only dependencies of our self-implemented code, but also the code quality and information from the bug tracker system of the 3rd party component.

1) Code-Quality: To assess the quality of the code itself and its evolution over multiple versions, we perform a problem-pattern-detection for analyzed versions of the 3rd party component. We use the SISSy tooling, also mentioned above, to detect problem patterns as described by Seng et al. [8] (e.g., complex code structures, dead code, or cyclomatic dependencies).

According to the number of affected quality aspects we differentiate major and minor problem patterns. The assignment of applicable categories and the ranking stems from Seng et al. [8] to ensure objectivity. High-ranked problem patterns should be considered with a higher priority than low-ranked ones. Independent from the actually applied ranking, the presented method is suitable to identify trends in the code quality.

According to Seng et al., we compare the numeric results of the problem pattern analysis to reference values from the Q-Bench research project. These values have been gathered from analysing more than 120 projects written in C++ and Java. The empirical data covers large and medium size projects, as well as open source and commercial software. The empirical values contain minimum, lower quartile, median, upper quartile and maximum regarding the numbers of problem pattern instances normalised for 1000 lines of code for each problem pattern type. Further details about this benchmark are also documented by Seng et al. [8].

This analysis allows i) trend analyses across multiple versions (i.e. how does software's quality evolve) and ii) an evaluation of the absolute code quality of a software product. When selecting a 3rd party library or during its risk assessment, one can use the available information to argue for or against a certain implementation of a 3rd party component. For example, an increasing reaction time for fixing critical bugs might be a threat to self-implemented code which relies on the 3rd party component.

If the 3rd party code does not fall within the lower quartiles of the benchmark's reference values, this does not necessarily mean that it is a risk to sustainability. It can only be inferred that there is an increased probability of a such a risk. Especially, if the software quality decreases from one version to another and many changes are made to poor quality artifacts, the quality indicators should be treated as warnings.

2) Bug Tracker: The second part of the quality analysis is performed on the bug tracker data of the 3rd party component. Snapshots of the bug tracker data, according to the releases of the component, are analyzed to derive the trend of the component's quality.

As opposed to the code quality analysis, there is no benchmark to compare the bug tracker analysis with. However, the results provide an insight on the quality and evolution of the component. Thus, if bugs remain open for a long period of time and over multiple releases, or the amount of severe bugs dominates in the overall bug number or the amount of bugs increases from release to release, this might be an indicator for quality and service problems of the 3rd party component.

Such an analysis is complementary to the code-level-analyses presented above. However, the application of bug tracking, in terms of stored information and bug tracker usage, strongly depends on the project's specific culture as also mentioned by Kim et al. [9].

IV. Case Study

To evaluate the approach, we have applied the proposed analysis in an industrial case study on a communication middleware component for industrial devices used by ABB. Figure 2 presents a high-level view on the components of the ABB system relevant for this case study. Due to intellectual property constraints, we had to perform the dependency analysis between OPC UA and OpenSSL and the code and bug tracker quality analysis based on OPC UA only.

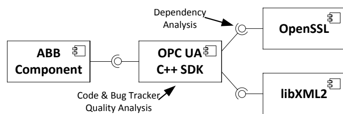

Fig. 2. Components under study

We have analysed three versions of the communication middleware component OPC UA C++ SDK and two versions of OpenSSL employed by those OPC UA versions. The findings of this analysis can be transfered to other 3rd party component integrations, and thus be used as a reference for similar case studies and researches.

In order to investigate the source code, we created a database for static code analysis (i.e., a SISSy database) containing three versions of OPC UA SDK code and their respective OpenSSL full-code version.

A. Analysis Results

1) Dependency Overview: From the SISSy database we determined the number of dependencies between OPC UA C++ SDK modules. We used the OPC UA C++ SDK directories as modules for our analysis.

Based on the number of direct and indirect dependencies, a component named UA Stack has been identified to be a central element of OPC UA C++ SDK code. Due to the high number of dependencies, changes in this component should be done very carefully, since a change might lead to change propagation to some or all of its dependent modules. The usage of OpenSSL functionality has increased from 149 references to 232 references in the analysed OPC UA releases. As a result, the dependency on this 3rd party component has become even more solid. In the following, we describe the results of the dependency analysis for this component.

2) Impact of 3rd party signature changes: In step 1, using the method signature-based impact analysis described in Section III-A, we identified 83 public methods of OpenSSL which have changed from one version to another. Step 2 of the analysis returned no dependencies on these methods. Thus no manual investigation is required for them and the analysis saved unnecessary investigation efforts.

3) Impact of hidden 3rd party changes: The first step of the hidden change impact analysis returned 130 methods of OpenSSL that have been changed internally.

In the second step, we calculated affected methods within OPC UA code based on the list of 130 methods with internal changes. As a result we found seven methods within OPC UA that are potentially affected by these implementation changes. Manual investigations of those methods identified that the source code does not need to be modified. Again, substantial effort could be saved due to the reduction to only seven methods that need to be reviewed. Otherwise all changes and their impact would have had to be identified manually or the decision whether to migrate to the new version could not have been made.

B. Code Quality Analysis

The problem pattern analysis of the OPC UA code returned problem pattern instances in a good range of the benchmark values. They reside in the range of minimum to lower quartile, so they are better than three quarters of the reference projects. No major trends for the quality could be observed for the analysed versions.

C. Bug Tracker Analysis

The bug tracker data analysis over three release snapshots (overall 153 issues) indicated acceptable component quality and a positive trend in its development. For example, the ratio of major and critical bugs to minor bugs decreased from 1.62 to 0 because none of them were left at the release date of the last version.

V. Conclusion

This paper addresses the impact estimation of evolving 3rd party components integrated into long-living industrial software systems. We have presented our approach to use static code dependency analyses to identify explicit and hidden 3rd party component changes that might be propagated into the self-implemented software and need further investigation. Further on, we included code quality as well as bug tracker analysis to assess the quality trend for multiple versions of the 3rd party component.

We presented the results of our case study on an industrial software system used by ABB for the automation domain. The case study identified signatures and hidden changes, and their potential impacts on the self-implemented code. In addition, code has been identified which will not be affected and thus the analysis saved the effort for manual investigation in these parts of the code. Potential impacts that were identified, separated for signature- and hidden changes, as well as code areas that do not require any manual investigation, are valuable results that support the management of dependencies and back up decision making for 3rd party components.

In our current and future work, we extend and improve our code change detection heuristics as proposed by Neamtiu et al. [6] in order to further reduce the required manual investigation. Further on, we work on standardising the queries performed for the analyses to provide them as part of our free tooling. Within ABB, the analysis strategy will be provided to business units and product development to improve and support their decision making.

References

[1] P. J. Clarkson, C. Simons, and C. Eckert, “Predicting change propagation in complex design,” *J. of Mech. Design*, 2004.

[2] G. Kotonya and J. Hutchinson, “Analysing the impact of change in cots-based systems,” in *COTS-Based Software Systems*. Springer, 2005.

[3] B. Zheng, Jiang ; Robinson and K. Williams, L. ; Smiley, “A lightweight process for change identification and regression test selection in using cots components,” in *5th Int. Conf. on COTS-Based Soft. Sys., 2006.*

[4] J. M. Voas, “The challenges of using cots software in component-based development,” *Computer*, vol. 31, 1998.

[5] S. Bohner, “Extending software change impact analysis into cots components,” in *Soft. Eng. Workshop, 2002. Proc.s. 27th Annual NASA Goddard/IEEE*, 2002.

[6] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” *SIGSOFT Softw. Eng. Notes*, vol. 30, 2005.

[7] “SISSy – Structural Investigation of Software Systems.” [Online]. Available: http://www.sqools.org/sissy/

[8] O. Seng, F. Simon, and T. Mohaupt, *Code Quality Management*. dpunkt Verlag, Heidelberg, 2006.

[9] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., “Memories of bug fixes,” in *Proc. of the 14th Int. Symp. on Found. of Soft. Eng.*, 2006.