On Software Modernisation due to Library Obsolescence

Simos Gerasimou
Department of Computer Science
University of York
York, UK
simos.gerasimou@york.ac.uk

Maria Kechagia
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
m.kechagia@tudelft.nl

Dimitris Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

Richard Paige
Department of Computer Science
University of York
York, UK
richard.paige@york.ac.uk

Georgios Gousios
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

ABSTRACT

Software libraries, typically accessible through Application Programming Interfaces (APIs), enhance modularity and reduce development time. Nevertheless, their use reinforces system dependency on third-party software. When libraries become obsolete or their APIs change, performing the necessary modifications to dependent systems, can be time-consuming, labour intensive and error-prone. In this paper, we propose a methodology that reduces the effort developers must spend to mitigate library obsolescence. We describe the steps comprising the methodology, i.e., source code analysis, visualisation of hot areas, code-based transformation, and verification of the modified system. Also, we present some preliminary results and describe our plan for developing a fully automated software modernisation approach.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; *Software evolution*; *Maintaining software*;

KEYWORDS

application programming interfaces; software libraries; library evolution; software modernisation; visualisation

1 INTRODUCTION

Modern software development practices advocate the use of third-party libraries as a means of improving maintainability, usability and dependability of software systems [8]. Many of these libraries aggregate functions and services, and expose endpoints, i.e., Application Programming Interfaces (APIs), that enable interacting with the logic of the libraries. An API is considered an implicit agreement between a software system (client) and a third-party library (provider). Based on this agreement, software developers can integrate the library within their client systems and use the provided functions without needing to be aware of the underlying implementation logic.

Despite the benefits accompanying the use of third-party libraries and their APIs, a strong dependency link is created with software systems using these libraries [2, 3]. Since software systems and the libraries they are based on evolve independently, maintaining the system to a fully-functional state requires additional effort when changes to the libraries' APIs violate the agreement. For instance, library evolution caused by new requirements or architectural changes can lead to API modifications that break compliance with client systems. Likewise, end-of-support of a software library can cause client systems to fall into "technology stagnation" unless they switch to another functionally-equivalent and actively maintained library. Similarly, the introduction of a competing library with improved functionality, better features and reduced overheads might urge software developers to consider adopting this new library in subsequent system versions.

In order to avoid the imminent risks that arise from using obsolete libraries (e.g., unresolved bugs, susceptibility to cyber attacks due to security issues), developers typically sustain the effort of modernising their systems [14]. Nevertheless, modifying the system to start using a new library is a time-consuming, error-prone and, to a great extent, developer-driven task [2, 6]. Performing this task correctly includes identifying system instructions that must change, doing the necessary code changes and, finally, checking that the modernised system preserves its original functionality. As the size of the client system and the legacy third-party library increases, the effort required for performing these steps increases significantly [8].

We argue that developers can reduce significantly the manual effort spent on mitigating library obsolescence by adapting concepts and approaches from the areas of software visualisation and bug finding [6, 13]. For instance, visualisation paradigms can help developers to understand better the architecture of their systems and perform impact analysis. Software visualisation is an active research area, but with rather limited adoption in software development [9, 13]. Also, code-based and pattern-based transformation approaches can assist with automating code refactoring [2].

In this paper, we propose an end-to-end methodology that not only helps developers to assess the required effort, but also automates a great part of the modernisation task. In particular, the

WAPI'18, June 2–4, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5754-8/18/06…$15.00
https://doi.org/10.1145/3194793.3194798

Figure 1: GitHub issue for TinyXML

methodology involves analysis of the software system's source code to extract *hotspots*, i.e., source code elements that invoke the legacy library, visualisation of these hot areas using the city metaphor [13], automatic source code transformation based on templates (patterns), population of the generated templates with suitable code, and, finally, verification of the software system and the transformed code to establish that the functional and non-functional requirements of the resulting system are maintained.

The remainder of the paper is organised as follows. In Section 2 we present examples for motivating our approach. In Section 3 we introduce the proposed approach for mitigating library obsolescence and describe a set of preliminary results. Finally, in Sections 4 and 5, we present related work and outline our conclusions and plans for future work, respectively.

2 MOTIVATING EXAMPLES

We motivate our work using two examples in which library evolution introduced compatibility-breaking changes. The first example comes from TinyXML2[1], the new version of the popular C++ XML parsing library. Even though the evolved library is more CPU and memory efficient than its predecessor, it is heavily redesigned and does not maintain backward compatibility. Figure 1 shows a relevant issue[2] from the GitHub repository of the TinyXML project. These obsolescence issues caused modernisation problems to developers of client systems. Some developers performed the migration manually while others preferred to switch to another functionally-equivalent library. For instance, the FileZilla client (https://filezilla-project.org) started using pugiXML from version 12 onwards. Irrespective of the chosen solution, the effort required for updating the system was substantial (e.g., see FileZilla development diary[3]).

Second, we refer to the *Lighttpd* web-server library which is used by several high-traffic websites (e.g., Wikipedia, YouTube). Research in [5] found that a one-line change to one of the methods in the updated library broke support for HTTP compression and crashed client applications. Despite the minimal API changes, this new issue could push developers to undertake the effort and migrate their systems to alternative and more reliable libraries.

Clearly, library and API changes affect the stability of client systems. We aim to assist developers by identifying the affected source code in client systems and simplifying the migration process.

[1] http://www.grinninglizard.com/tinyxml2/index.html
[2] https://github.com/leethomason/tinyxml2/issues/440
[3] https://tinyurl.com/y7styrn4

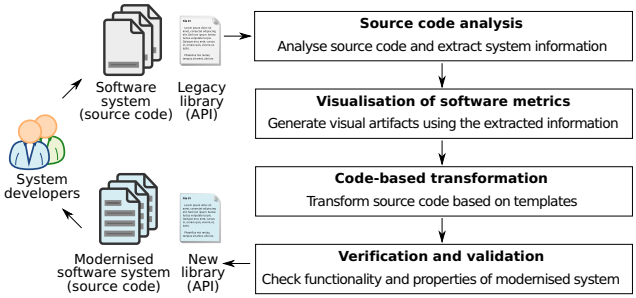

Figure 2: High-level overview of our methodology for mitigating library obsolescence.

3 SOFTWARE MODERNISATION

3.1 Modernisation Methodology

The high-level overview of our methodology is shown in Figure 2. Given as inputs the source code of the client software system and the API of the obsolete library, our methodology operates as follows. Initially, it starts a set of extraction transformations that enable source code analysis, and extraction of suitable metrics and abstract specifications from the legacy code. These metrics (e.g., total number of API invocations) are very important as they provide the means to establish and visualise the dependency level of the software system on the obsolete library. Also, the specifications are in a form suitable for reengineering. Then, through a code-based transformation step, our methodology uses the specifications to generate code based on templates (patterns). These templates are subsequently populated with suitable code that permit to start exercising the new library. Finally, verifying and validating the transformed code and the software system as a whole (e.g., running unit and integration tests) enables to check that both the functionality and properties of the modernised system are preserved.

Stage 1: Source code parsing and analysis
This automated stage focuses on parsing and analysing a set of input files, i.e., the source code corresponding to the client system and the API of the obsolete library. The parsing step comprises a series of actions including source code scanning and preprocessing, identification of language tokens, semantic analysis, name resolution and binding, and, finally, generation of various types of internal models, indexes and Abstract Syntax Trees (ASTs).

Once parsing finishes, the extracted models and ASTs are analysed. This step enables to identify elements, i.e., files, classes, methods, and instructions, of the software system that access the obsolete library. Orthogonal to this process is the extraction of a similar set of elements from the library's API that are used by the system.

By combining and analysing the generated sets of elements we gain insight into the system and establish several dependency metrics. These metrics capture information about the system architecture including interconnections between software modules and dependencies with external libraries and components. Also, through impact analysis of the affected system elements, we can establish a coupling degree between the software system and the obsolete library.

The effectiveness of this stage depends on using a suitable scanning and parsing software component. High-level programming

languages (e.g., C/C++, Java, C#) come with publicly available and ready-to-use infrastructures that help with this task; see, for instance, Eclipse CDT for C/C++ and Roslyn for C#. We use Eclipse CDT for our prototype implementation (Section 3.2).

Stage 2: Visualisation of software metrics
Manual source code inspection is challenging for complex systems with multiple dependencies, packages, classes, and methods (e.g., FileZilla has more than 300 C/C++ files and more than 130KLoC). To facilitate reasoning and effort estimation, we automatically generate interactive visual artifacts using the metrics produced during analysis. These artifacts provide a pictorial view of the system and help with system understanding.

The benefits of using suitable visual metaphors both for assisting with software exploration and for reducing the cognitive load is widely accepted [1, 9]. Several visual metaphors have been proposed for representing static aspects of software (cf. Section 4). To the best of our knowledge, however, there is no prior research on visualising the dependencies of a software system on third-party libraries and using this information to guide software modernisation.

We bridge the gap in existing research by adapting the semantics of available visualisation metaphors (e.g., Treemap, Code City [13]) to this problem. In our preliminary realisation (Section 3.2), we use Code City to represent software systems as three-dimensional cities with districts and buildings, and associate the extracted dependency metrics with visual properties of city components.

The development team can study the produced interactive visual artifacts and carry out risk analysis. The outcome of this study indicates how to address the obsolescence issues best. An estimate of the effort required for completing the modernisation can also be made. For instance, if extensive coupling is identified, it might be more sensible to investigate how to reduce its degree (e.g., by improving the system architecture) before proceeding with the remaining modernisation stages.

Stage 3: Code-based transformation
During this stage, the software system undergoes a three-step transformation to become compliant with the new library. The first step includes (1) the automated generation of an abstraction layer (e.g., using the adapter pattern) and its population with elements that delineate the usage of the obsolete library by the software system; and (2) the automated modification of commands in the affected system files to delegate the work to the generated abstraction layer instead of the obsolete library. In this way, our methodology minimises changes in the system's source code.

The next step involves the inference of mappings between the obsolete and new libraries [4]. A mapping rule comprises two sets of API commands that perform the same task, one from the obsolete library and the other from the new library. The outcome of this inference step is a list of mappings.

Inferring likely API mappings is challenging and time-consuming, and its automation has been the focus of recent research [12]. The selection of a suitable technique depends on the characteristics of the considered libraries and the expertise of developers. For instance, static analysis and textual similarity techniques [10] can be used when the APIs of the libraries are, to a degree, similar. Alternatively, developers can resort to inspecting the documentation of these APIs and generating the list of mappings manually.

Listing 1: Code fragment that maps the get attribute method in TinyXML to the equivalent methods in PugiXML)

```
const char* XMLElement::Attribute(const char* name,
                                  const char* value) const {
  return pugiXMLNode.attribute(name).value(); }
```

The final step involves populating the generated abstraction layer with suitable code fragments that invoke the new library. To achieve this, developers use the extracted list of mappings and write code within the placeholders in the abstraction layer. For instance, Listing 1 shows the TinyXML method (placeholder) for extracting the attribute from an XML element and the corresponding code fragment in PugiXML (https://pugixml.org); this relationship is one-to-many. Once this is done, the functionality that was previously done by the obsolete library is now undertaken by the new library.

The level of difficulty for writing the relevant code fragments depends on the correspondence between the obsolete and the new library. Investigating the time and effort required for completing this task based on this correspondence is part of our future work.

Stage 4: Verification and validation
The next stage involves checking that both the functional and non-functional requirements of the evolved system (transformed and unaffected code) continue to hold. This includes running unit, integration and system tests, or any other type of formal verification (e.g., use model checking to verify the absence of concurrency bugs).

Special focus should be given to the amended system parts, i.e., the abstraction layer and affected components, as these parts are most likely to have introduced erroneous behaviour. The visual artifacts generated earlier can help developers to extract traceability information and decide where to spend most of their efforts.

3.2 Preliminary Realisation

We present a prototype realisation of our methodology which is currently under development as an Eclipse plugin. To evaluate our approach, we use the FileZilla client v.11, a mature open-source C/C++ application with a non-trivial code base and several functional and non-functional requirements. FileZilla uses a number of third-party libraries, including TinyXML for reading and updating XML files that keep server sites and interface-related properties.

At first, we obtained the ASTs of the FileZilla source code using the parsing facilities provided by Eclipse CDT[4]. The automated analysis of these ASTs enabled the identification of *hotspots*, i.e., FileZilla code elements that access the legacy XML library. We also gained insight into the usage level of this library by the system.

We adapted the Code City metaphor [13] to visualise the information extracted during the analysis stage. Figure 3 shows a representation of FileZilla using our adapted city metaphor. We employ the basic metaphor semantics, i.e., represent an application as a city, a package/sub-package as a district/sub-district, and a class as a building. However, we introduce new semantics that help visualising the dependency level between the software system and the obsolete library, and capturing the hotspots. First, we use red-coloured buildings to show a class that invokes the obsolete library, i.e., an affected class. Second, we set the height of a red-coloured

[4]https://www.eclipse.org/cdt

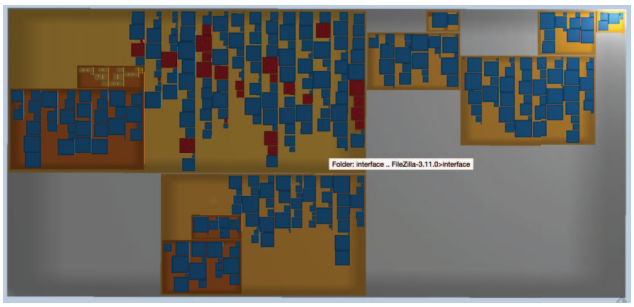

Figure 3: FileZilla visualisation using our adapted code city metaphor. The white box is the system, districts/sub-districts with yellow to orange colours are packages/sub-packages, blue buildings are unaffected classes and red buildings are classes that use the obsolete XML library.

building as the number of commands from the obsolete library present in an affected class; taller buildings have more commands that need to change. Third, we set the width of a red-coloured building as the number of distinct commands used from the obsolete library; wider buildings make heavier use of the obsolete library, and hence the corresponding classes have higher coupling.

Next, we executed the generation transformations that enabled to automatically produce an abstraction API layer and modify the affected FileZilla files (i.e., link them with the abstraction layer instead of the legacy library). We produced the list of mappings manually by inspecting the specifications of the obsolete and new libraries. We used this mappings list to populate the elements comprising the abstraction layer with suitable code fragments. Our preliminary experience showed that some code fragments were straightforward and easy to derive, whereas others were more difficult and required more effort. Supporting automated mapping inference and code population is part of our future work.

Given the modernised system, we used the test suite coming with FileZilla and confirmed that the functionality and properties of the system still hold. This assessment provided insight that the abstraction layer has been generated and populated correctly, and also that the system started exercising the new library.

4 RELATED WORK

API Evolution. Recent research has been focused on identifying issues that can break client applications' source code, because of changes in used APIs. Jazek et al. studied this problem in real word programs and argued that better tools and methods need to support library updates in client applications [6]. Xavier et al. also investigated API changes that may break previously established contracts, resulting in software crashes [14]. They conducted empirical studies with real software projects, and they reported lessons for better library support and maintainability of client applications. Raemaekers et al. proposed a framework of metrics that can be used to measure the stability of a software library [11]. Based on insights from previous work and considering the challenges associated with mitigating library obsolescence, we develop a visualisation approach that can assist developers of client applications to improve software robustness.

Visualisation Techniques. Software visualisation techniques are widely used in program comprehension; see for instance the recent survey in [9] about existing software visualisation approaches. This survey highlights the number of visualisation tools for supporting developers' needs on dependency management. A few studies have attempted to provide visualisations of evolving software systems and their library dependencies. Kula et al. developed a heat-map metaphor, which can help maintainers to navigate to library dependencies and gives an overview of the users across the different versions of a library [7]. Wettel et al. have used the city metaphor to represent software projects and they conducted a controlled experiment with developers to examine how the subjects comprehend the structure of a program [13]. In addition to the existing related work, we are using the city metaphor to highlight areas of software projects using third-party libraries.

5 CONCLUSIONS

The increasing dependency of software systems on third-party libraries challenges system maintainability. Developers need guidance to identify and modify client source code that can be affected due to library obsolescence. We proposed a modernisation methodology that can help developers of client systems to maintain their source code when these obsolescence issues occur. Our methodology applies to programming languages that support library linking and the analysis of their implementations permits the detection of library use in client source code. Also, we introduced a visualisation approach, using the *city metaphor*, for visualising source code areas that invoke the legacy library. We presented a prototype realisation of our methodology and applied it to the FileZilla client. We plan to provide a mechanism to automatically adapt client source code to new third-party libraries. We will also explore the automated test generation for changed parts of client source code.

REFERENCES

[1] C. R. B. de Souza and D. L. M. Bentolila. 2009. Automatic evaluation of API usability using complexity metrics and visualizations. In *ICSE'09 - Companion Volume*. 299–302.
[2] D. Dig and R. Johnson. 2005. The Role of Refactorings in API Evolution. In *ICSM'05*. 389–398.
[3] S. Gerasimou, D. Kolovos, R. Paige, and M. Standish. 2017. Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems. In *STAF'17*. 385–393.
[4] A. Gokhale, V. Ganapathy, and Y. Padmanaban. 2013. Inferring likely mappings between APIs. In *ICSE'13*. 82–91.
[5] P. Hosek and C. Cadar. 2013. Safe Software Updates via Multi-version Execution. In *ICSE'13*. 612–621.
[6] J. Kamil, D. Jens, and B. Premek. 2015. How Java APIs break - An empirical study. *Information and Software Technology* 65 (2015), 129–146.
[7] R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue. 2014. Visualizing the Evolution of Systems and Their Library Dependencies. In *VISSOFT'14*. 127–136.
[8] R. Lämmel, E. Pek, and J. Starek. 2011. Large-scale, AST-based API-usage Analysis of Open-source Java Projects. In *SAC'11*. 1317–1324.
[9] L. Merino, M. Ghafari, and O. Nierstrasz. 2017. Towards actionable visualization for software developers. *JSEP'17* (2017).
[10] H. A. Nguyen, T. T. Nguyen, and et al. G. Wilson. 2010. A Graph-based Approach to API Usage Adaptation. In *OOPSLA'10*. 302–321.
[11] S. Raemaekers, A. van Deursen, and J. Visser. 2012. Measuring software library stability through historical version analysis. In *ICSM'12*. 378–387.
[12] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API Property Inference Techniques. *TSE'13* 39, 5 (2013), 613–637.
[13] R. Wettel, M. Lanza, and R. Robbes. 2011. Software Systems As Cities: A Controlled Experiment. In *ICSE'11*. 551–560.
[14] L. Xavier, A. Brito, A. Hora, and M. T. Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *SANER'17*. 138–147.