Towards specifying pragmatic software reuse

Josip Maras, Maja Štula
University of Split
Rudera Boškovića 32
Split, Croatia
{josip.maras}, {kiki}@fesb.hr

Ivica Crnković
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
crnkovic@ericsson.com

ABSTRACT

Software reuse has numerous benefits, including reduced development time, defect density, and increased developer productivity. Numerous approaches to software reuse have been developed and we can divide them into two categories: preplanned approaches, where software artifacts are developed to be reused; and pragmatic approaches, that facilitate the reuse of software artifacts not necessarily designed for reuse. In this paper, we specify the general approach to pragmatic software reuse, which consists of three steps:*i) feature location*, which identifies the source code of an individual feature; *ii) code analysis and modification*, which fixes conflicts that can happen when achieving reuse; and *iii)* feature integration, which achieves reuse by integrating code into the target system. We also discuss how certain steps in the process are used in current state-of-the-art pragmatic reuse approaches. In addition, based on the experience of developing an approach to pragmatically reusing web application features, we identify general challenges in pragmatic reuse approaches.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Theory

Keywords

Reuse, Web Applications

1. INTRODUCTION

Software reuse is the process of creating software systems from parts of already existing systems, rather than custom developing every individual part from scratch. There are numerous benefits to reuse, among other things, it reduces development time and defect density, while increasing developer productivity [16], [12]. For these reasons, a number of software reuse approaches have been developed. We can divide these approaches into two categories: *i) Preplanned approaches*, in which software artifacts are explicitly developed with reuse in mind, e.g. object-oriented inheritance [6], software components [16], and software product lines [17]; and *ii) Pragmatic approaches* [9], which facilitate the reuse of software artifacts that were *not* necessarily designed for reuse.

While significantly more popular in the research community, preplanned reuse approaches suffer from three main drawbacks [11]: *i)* developing software artifacts in a reusable fashion is significantly more expensive, and it is economically infeasible to develop *all* artifacts to be easily reusable [8]; *ii)* since not every artifact can be developed in a reusable fashion, we often have to choose which should be developed as reusable, and making this decision is a difficult task [21]; and *iii)* even if a software artifact is designed and built as reusable, it is often developed with a certain set of assumptions about how it is supposed to be reused, which can decrease the number of contexts in which it can be deployed and reused [4].

In contrast, pragmatic approaches [9] such as code scavenging [12], ad-hoc reuse [18], opportunistic reuse [20] specifically target the reuse of software artifacts that were not explicitly designed for reuse. At the same time, pragmatic reuse is often labeled as a *wrong* way to reuse code, mostly due to its often non-systematic, ad-hoc nature [11]. However, by facilitating such reuse, and making it more automatic and systematic, we can access the untapped potential of existing code, thereby increasing developer productivity and lowering the overall costs of developing new software systems.

When performing reuse, our ultimate goal is not to reuse a piece of code, but to introduce an already developed feature into our target system, instead of building the feature from scratch. The term *feature* often has different meanings, depending on the context. For example, the IEEE [1] defines a feature as a distinguishing characteristic of a system item that includes both functional and nonfunctional attributes such as performance, security or reusability; while in the program comprehension community, a feature is a specific functionality, defined by requirements, and accessible to developers and users [7]. Notice how the feature definition from the program understanding community does not include non-functional requirements (e.g. performance, maintainability).

ECSAW '15, September 07 - 11, 2015, Dubrovnik/Cavtat, Croatia
Copyright 2015 ACM. ISBN 978-1-4503-3393-1/15/09 ...$15.00
DOI: http://dx.doi.org/10.1145/2797433.2797489.

In this paper, we specify the general approach to pragmatic software reuse; we identify three distinct steps in the process: *i) feature location*, identifying the source code of an individual feature; *ii) code analysis and modification*, fixing the conflicts that can happen when achieving reuse; and *iii)* feature integration, achieving reuse by integrating code into the target system. We also discuss how they are used in current state-of-the-art approaches. Based on our experience of developing an approach to pragmatically reusing web application features [13], we identify general challenges, and discuss advantages and disadvantages of pragmatic reuse approaches.

2. OVERVIEW

Let α and β be two applications, where α is not necessarily designed with reuse in mind. Both applications are implemented with a number of program elements: application α with $E^{\alpha} = \{{e_1}^{\alpha}, {e_2}^{\alpha}, \ldots, {e_g}^{\alpha}\}$ and application β with $E^{\beta} = \{{e_1}^{\beta}, {e_2}^{\beta}, \ldots, {e_h}^{\beta}\}$, and they offer a number of features: $F^{\alpha} = \{{f_1}^{\alpha}, {f_2}^{\alpha}, \ldots, {f_n}^{\alpha}\}$, represents the features of application α, and $F^{\beta} = \{{f_1}^{\beta}, {f_2}^{\beta}, \ldots, {f_m}^{\beta}\}$, the features of application β. A feature ${f_i}^{\alpha}$, in application α, is implemented by a subset of source elements from α: $E^{\alpha}_{f_i} \subseteq E^{\alpha}$, a subset which, in general, is not known at the start of the pragmatic reuse process. The idea of pragmatic reuse is to reuse the feature f_i^{α} into application β, which will then have a set of features $F'^{\beta} = \{{f_1}^{\beta}, {f_2}^{\beta}, \ldots, {f_m}^{\beta}\} \cup \{f_i^{\alpha}\}$.

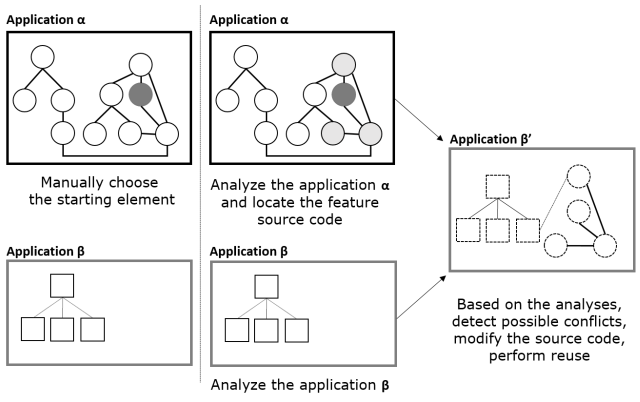
Application α
Manually choose the starting element
Application β
Application α
Analyze the application α and locate the feature source code
Application β
Analyze the application β
Application β'
Based on the analyses, detect possible conflicts, modify the source code, perform reuse

Figure 1: **The reuse process.**

Figure 1 shows a typical case of pragmatic reuse. We have an application α with a feature that we want to reuse in an application β. In most cases, the user selects some entry point in the application that represents the feature; most commonly, a class, a method, or a function, but it can also be a part of the UI (*Manually choose the starting element*, from Figure 1). Most often, this program element is not self-contained and has important dependencies to other parts of the system. So in the first step we have to correctly identify all necessary dependencies of the selected program element (*Analyze the application α and locate the feature source code*, Figure 1). Our final goal is to reuse the program elements of the feature into the application β. When doing this, we change both the environment in which the target feature code is executed, as well as the environment of the target application β. Since this can lead to subtle bugs, we have to detect and resolve all possible conflicts that can occur when performing reuse (*Based on the analyses, detect possible conflicts, modify the source code, perform reuse*, Figure 1). Finally, after the conflicts have been resolved we can actually merge the feature code into the target application and achieve reuse. In general, pragmatic reuse is composed of three distinct steps:

1. *Feature Location* – the code that implements a feature is identified in the originating system.
2. *Code Analysis and Modification* – often, both the feature code, as well as the source code of the target system will have to be analyzed and modified for the reuse to be possible.
3. *Feature Integration* – the feature code is integrated into the code of the target system.

In the remainder of the paper, we will give an overview of each step, and describe how they are implemented in current, state-of-the art approaches.

3. FEATURE LOCATION

The primary difference between pragmatic reuse and preplanned reuse approaches (such as component-based development) is the fact that pragmatic reuse is centered around reusing a feature not designed for reuse. The code that implements a particular feature is most often not well encapsulated in a self-contained entity that can be easily copied from one system into the other. Instead, the code of a feature is spread throughout the system. The main problem in such reuse cases is correctly identifying the code responsible for the implementation of a feature – *feature location* [5], [19].

Feature location techniques are based on different types of analyses designed to establish a traceability between the targeted feature and the implementing artifacts. Common approaches include: *i) textual analysis* – analyzes the source code on a textual level, guided by the idea that words used in the source code and comments encode domain knowledge, and that a feature may be implemented using a similar set of words throughout the system; *ii) static analysis* – examines data and control dependencies, for all possible program inputs, without executing the application; and *iii) dynamic analysis* – analyzes the execution of an application, it is often used for feature location when features can be invoked and observed during runtime. All of these approaches have their limitations. Textual approaches are often not precise enough and sometimes the source code is not in a state that allows for meaningful keyword analysis, static analysis has a tendency to overestimate since it analyzes the application for *all* possible program inputs, especially when dealing with highly dynamic systems, while dynamic analysis is limited to analyzing the execution traces (and it is often very difficult to obtain a representative set of execution traces that cover the complete feature behavior).

In the context of pragmatic reuse, there are different approaches to feature location. For example, Holmes et al. [10], have defined an approach and a tool – Gilligan, in which a developer has to manually choose the starting point (a class or a method) for reuse. The tool then statically analyses the application's source code dependencies and suggests other structural elements which should also be included for the code to be able to work. By accepting or refusing these suggestions, or by including additional elements, the developer exactly specifies/identifies the code for reuse. Gilligan is a tool and an approach for general purpose Java applications.

In a similar fashion, Automated Software Transplantation, by Barr et al. [3], is an approach that starts with the user marking the starting point (in their current implementation, a *C* function). Then they use static backward slicing [22] to identify a path from the main function to the starting point function, and forward static slicing to identify all code that is actually necessary for the implementation of the selected function. Since static slicing tends to overestimates the amount of code that needs to be included, they use a genetic algorithm driven by unit tests, to prune out unnecessary code.

In our approach to pragmatic reuse [13], we have focused on client-side web applications – a UI domain in which features have observable behaviours such as user-interface (UI) modifications. In that case, the user chooses the part of the UI on which the feature of interest manifests and triggers the feature, either manually, by exercising certain user-triggerable actions (such as mouse clicks, moves, or keyboard presses) or by running tests associated with the feature. During feature execution, we analyze the execution of the application, identify all dependencies that exist in the execution, and locate the points in the application execution in which the application behavior can be observed from the outside, for example: when changes to the selected parts of the UI are made, or when communications with the server-side are performed. These points in the application execution capture the behavior of a feature, and we call them *feature manifestation points*. Then, by dynamically slicing the application with feature manifestation points as slicing criteria, we are able to identify the code responsible for the implementation of the feature [15].

4. RESOLVING CONFLICTS AND INTEGRATING FEATURES

When reusing source code from one system into another, we are changing the environment in which the source code of the feature is executed in, as well as possibly modifying the environment on which the target application relies on. At the same time, both the functionality of a certain feature, as well as the functionality of the target system, can also depend on certain global state, provided by the platform. Since these expectations can be in conflict, there exists a possibility that certain modifications to the source code of the feature and the source code of the target application will have to be made. This is usually done by performing source code analysis and identifying potential conflicts.

Most of the approaches [3], [10], modify only the extracted source code. For example, [10] modifies the extracted source code in order to minimize the amount of compilation errors, while [3] analyzes the target system for identifier definitions, and in order to avoid conflicts, performs necessary renamings.

In our approach to pragmatic reuse [13], we are dealing with client-side web applications, which compared to most other domains, have a greater emphasis on global state. The global state can have a wide range of influences on the execution of both the feature code and target application code, and often these problems cannot be resolved with simple renamings in the feature code. For this reasons, we have to detect all modifications and dependencies to the global state from both applications. Next, we have to detect possible conflicts, and make the necessary adjustments, possibly to both the feature code, but as well to the target application code.

It is important to emphasize that this part of the process is very domain dependent. In most domains, the isolation of the feature source code can be relatively easily achieved by placing the source code in special namespaces, and therefore often, only simple renamings of global identifiers are required. While, in other domains, for example, the web application domain, that kind of isolation is often not possible, and more advanced analysis and code fixes have to be applied.

4.1 Feature Integration

Finally, after the feature code has been identified, extracted, and both the feature code and the target system code prepared for integration, we can perform the actual reuse by embedding the source code of the feature into the target application.

This phase is often accompanied by a *verification* stage in which we test whether the reuse was performed successfully or not. For example, our approach [13], as well as the software transplantation approach [3], test the newly produced system by running both the original tests of the target system as well as the feature tests. The tests of the original system test for regression failures – does the inclusion of a new feature break something in the remainder of the code. Feature tests, on the other hand, test whether the feature has the same behavior in the system, as it had in the originating system. As with any tests, performing this verification step does not guarantee the absence of bugs and reuse problems, but it gives the developers the same level of confidence as they had with the current state of the two systems.

5. LESSONS LEARNED

During our research on pragmatic software reuse, we have noticed a number of advantages and disadvantages inherent in the process. The greatest advantage of pragmatic reuse is that it enables us to access the untapped potential of already existing code bases, which were not necessarily designed with reuse in mind. This has the potential of significantly speeding up development time and improving developer productivity. On the other hand, all currently developed pragmatic reuse tools are research prototypes developed to test the viability of a pragmatic reuse approach. This means that so far, possibly due to the immatureness of the tools, there were no large scale studies with developers that test these benefits in close to real-world settings. In our opinion, developing more mature tools and empirically testing the benefits of pragmatic reuse is of paramount importance for pragmatic reuse to become a standard part of software application development.

There is also the problem of validating the correctness of reuse, especially since problems might arise from the interplay of feature code and the target system code. For this we require high quality tests for both the feature and the target system. These tests might not always be available. In that case, automatically generated tests could be used [14], [2].

It is also important to emphasize that current approaches to pragmatic reuse do not really fully solve the general reuse problem. Since reuse is usually done on source code level, the act of reuse is limited only to similar systems developed in compatible technologies. The fact that currently performed research has not explored this option does not mean that this problem could not be tackled – once the code to be reused is identified and extracted it could potentially be hosted within containers that could achieve reuse across various environments.

The ultimate goal of pragmatic reuse is to be as automatic as possible – the developer performing the reuse should only specify the targeted feature and the target location, and the reuse tool should perform the rest automatically. However, the current feature location approaches suffer from some limitations: textual approaches from imprecisions, static analysis from overestimation, and dynamic analysis from the inability to cover all possible execution traces. For these reasons, further research should be invested in improving the precision of feature location techniques. This is one of the most important steps in the pragmatic reuse approaches that has a great influence on the usefulness of pragmatic reuse.

6. CONCLUSION

In this paper we have specified the general approach to pragmatic software reuse – achieving reuse from systems that were not necessarily designed with reuse in mind. We have presented different steps in the process: *i)* feature location, *ii)* code analysis and modification, and *iii)* feature integration. Next, we have discussed each of these steps in the context of current, state of the art approaches. In the end, we have summarized the lessons learned about the different advantages and disadvantages of pragmatic software reuse, that we have gained by developing a state of the art pragmatic reuse approach.

7. REFERENCES

[1] IEEE standard for software and system test documentation. *IEEE Std. 829-2008*, 2008.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of Javascript web applications. In *Software Engineering, ICSE 2011, 33rd International Conference on*, pages 571–580. ACM, 2011.

[3] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *International Symposium on Software Testing and Analysis*, 2015.

[4] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 102–109. IEEE, 1994.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[6] O.-J. Dahl and K. Nygaard. Simula: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.

[8] J. E. Gaffney Jr and R. D. Cruickshank. A general economics model of software reuse. In *Proceedings of the 14th international conference on Software engineering*, pages 327–337. ACM, 1992.

[9] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th international conference on Software Engineering*, pages 447–457. IEEE Computer Society, 2007.

[10] R. Holmes and R. J. Walker. Semi-automating pragmatic reuse tasks. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 481–482. IEEE Computer Society, 2008.

[11] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4):20, 2012.

[12] C. W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[13] J. Maras, J. Carlson, and I. Crnković. Towards automatic client-side feature reuse. In *Web Information Systems Engineering–WISE 2013*, pages 479–488. Springer, 2013.

[14] J. Maras, M. Štula, and J. Carlson. Generating feature usage scenarios in client-side web applications. In *Web Engineering*, pages 186–200. Springer, 2013.

[15] J. Maras, M. Stula, J. Carlson, and I. Crnkovic. Identifying code of individual features in client-side web applications. *Software Engineering, IEEE Transactions on*, 39(12):1680–1697, 2013.

[16] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98. sn, 1968.

[17] D. L. Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, (1):1–9, 1976.

[18] R. Prieto-Díaz. Status report: Software reusability. *Software, IEEE*, 10(3):61–66, 1993.

[19] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[20] M. B. Rosson and J. M. Carroll. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253, 1996.

[21] W. Tracz. Where does reuse start? *ACM SIGSOFT Software Engineering Notes*, 15(2):42–46, 1990.

[22] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.