IJSAR, 4(5), 2017; 39-45

International Journal of Sciences & Applied Research

www.ijsar.in

Prioritizing software components for realistic reuse

J. M. S. V RaviKumar* and Katta Subba Rao

CSE Department, BVRIT, Narsapur, T.S., India.

Corresponding author: *Dr. J. M. S. V RaviKumar, CSE Department, BVRIT, Narsapur, T.S., India.

Abstract

Practical software reuse, in which existing software components are invasively modified for use in new projects, involves three main activities – selection, alteration and integration. Most of the academic research into practical research to date has focused on the second of these activities, alteration, especially the definition of reuse plans and verification of persistent changes, even though the selection activity is arguably the most important and effort-intensive of the three activities. There is therefore a great deal of scope for improving the level of support provided by software search engines and recommendation tools to practical re-users of software products. Search engines are particularly promising in this regard since they possess the intrinsic ability to "evaluate" products from the viewpoint of users' reuse scheme. In this paper we discuss some of the main issues involved in prioritizing the selection support for practical reuse provided by test-driven search engines, describe some new metrics that can help address these issues, and present the outline of an approach for grade software products in search results.

Keywords: Software components, Realistic reuse

Introduction

The practical reuse of software revolves around the invasive alteration and reuse of code for purposes for which it was (probably) not originally intended [2]. Depending on the amount of code that needs to be changed and the level of quality desired, pragmatic reuse can be very invasive, so when a variety of possible reuse candidates is available it is important that developers select the most suitable product. However, the selection process is itself very important since, by its very nature, it requires every applicant to be analyzed even though only one of them will probably end up being reused. Indeed, if no sufficiently suitable product is found, developers will often decide to build the required working nature completely from scratch. Today, this selection process is highly performed manually using heuristics.

Obviously software search engines and recommendation tools [3], [4], [5] (broad overview in [6]) can in principle play a key role in pointing this problem. However, to date no search engine has focused on supporting practical reuse. The majority of software search engines use basic, text-matching algorithms to identify search results, and if they provide any *metrics* at all, they are generic, predefined metrics simply calculated on the total of the code personified by components without any concern for the user's reuse approach. Therefore, even when working with a conventional code search engine, users still have to do the best part of the work needed to evaluate modules appropriateness for practical reuse.

There is one search technology, however, which has the potential to offer much better selection support for practical software reuse – search technology [7]. Test driven search engines such as Merobase [4] build on traditional software search technology by filtering the initial search results using a test supplied by the user. This test in effect represents the query against which the engine searches, and essentially characterizes how the sought after product will be used in the user's new approach. Test-driven search engines therefore have the advantage that they are supplied with an running representation of the usage approach for which the desired product is intended. At the time of writing, however, only two test-driven search engines have been described in the literature (S6 [3] and Merobase [4]), and neither has been optimized to support the selection process for practical reuse. This is the goal of the work outlined in this paper. In the next section we briefly discuss general-purpose software metrics and their shortcomings with respect to component reuse. Section 3 then presents our proposed approach for helping to rank search results according to their overall suitability for a realistic reuse scenario. The ranking approach and related challenges are discussed in section 4. Finally, section 5 concludes the paper with some final observations.

Categorize strategy

In the context of pragmatic reuse of software components, a huge variety of criteria influence the degree to which products "match" the requirements of a developer's reuse context, some functional and others non-functional. Software metrics clearly afford the basis for quantitatively measuring this "degree of similarity ". However, there are two major challenges that have to be overcome to use software metrics effectively to finalize a component's appropriateness for specific reuse approach.

The first challenge is the sheer number of software metrics proposed by academia and used in industry [6] to measure the "worth" of code. This makes it impossible to define a single metric or set of metrics that best capture a modules's "degree of match" in all usage scenarios across all software domains. At best, developers usually have their own individual interpretation of software metrics [7], and at worst, they have little or no idea what "qualities" raw metrics indicate. To be useful, therefore, software component search engines should, on the one hand, allow developers to specify the software metrics they believe best capture their individual approach and on the other hand, apply some predefined, generic strategy for prioritize modules based on a sophisticated combination of base metrics. However, combining metrics is reputably difficult, even when the usage scenario is exactly known, and supporting the efficient individual selection of metrics by search engine users will almost certainly require some kind of complete software metrics catalog [9].

The second challenge is measuring the use of metrics to take into account a developer's specific usage scenario when ranking software components for practical use. This affects not only the functional suitability of components, but also their non-functional suitability based on the quality goals of the developer. By its very nature, practical reuse usually involves the invasive adaptation of components by re-users, with the attendant risks of error introduction.

A consequence, raw metrics measured in a simple way on the entirety of the functionality encapsulate by a component may give a misleading view of its suitability for a particular circumstances because they are usually prejudiced by functionality that is not needed for that scenario. To provide a realistic measure of a component's appropriateness for pragmatic reuse, metrics need to focus on the actual functionality of relevance. This applies to metrics selected by search engine users, as well as predefined, aggregate metrics and other approaches for ranking mechanism. In short, therefore, to provide effective relative measures of a component's appropriateness for pragmatic reuse scenarios, search engines need to find some way of distinguishing relevant from irrelevant functionality and of adapting/combining standard metrics in a way that best procedures the overall effort that their potential reuse will rivet.

Concepts and metrics

For our purposes we define the pragmatic reuse of a component as the reuse of a component in a usage scenario for which it was not (or may have not been) planned.

This definition encompasses the original definition of Holmes et al. [1] and ties it to the notion of a usage scenario. Since there is no universally agreed definition of the term *component* in literature, we regard a component as any collection of inter-dependent compilation units written in a mainstream object-oriented language (e.g. Java, C# and Objective-C/Swift) that delivers a coherent package of functionality. In object-oriented languages, the methods of compilation units such as classes and interfaces represent the most natural unit of *functionality*. As a practical measure of the "work done" by a method we use the static instruction count metric [12], since this accounts for the varying complexity and calculations performed inside method bodies. Technically, in Java for example, a component is regarded as a collection of pragmatically reusable compilation units that deliver functionality for a specific usage scenario. The reusable functionality of a component is assumed to only be accessible through the component's interface methods which, by definition, have to be externally visible to client software. To overcome the limitations of general-purpose metrics for component ranking, we propose several concepts and accompanied metrics made to the pragmatic reuse of components, each providing a unique perspective on the component, guided by the question of what a developer doing practical reuse wants to know from a search engine about potential candidates. Figure 1 provides a conceptual overview of the different kinds of metrics proposed to support the decision making process in pragmatic reuse of software components for developers. Each individual concept (highlighted in bold) is explained in the following

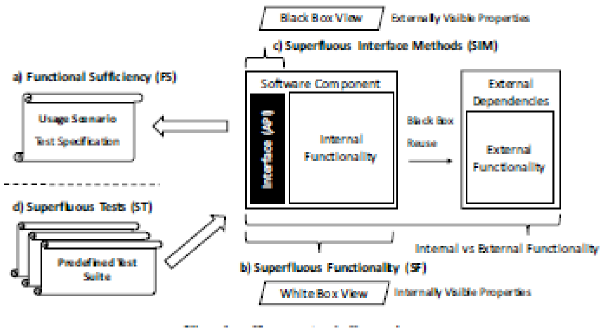
Black Box View
Externally Visible Properties
c) Superfluous Interface Methods (SIM)
a) Functional Sufficiency (FS)
Usage Scenario
Test Specification
Software Component
Interface (API)
Internal Functionality
Black Box Reuse
External Dependencies
External Functionality
d) Superfluous Tests (ST)
Predefined Test Suite
Internal vs External Functionality
b) Superfluous Functionality (SF)
White Box View
Internally Visible Properties

Fig1. Conceptual overview

A. Functional Sufficiency

In the practical reuse of a component, a developer usually aims to find reusable candidates that at least partially deliver sub-functionality for the envisioned usage scenario. The concept of functional adequacy (cf. Figure 1 a) measures the extent to which a component delivers the desired functionality for a certain usage scenario, with a value in the range [0, 1]. At the extreme values, if a component has a FS of 0 for a certain usage scenario it is said to be unsuitable, whereas if it has a FS of 1 it is said to be functionally sufficient. Components with a value in between are said to be partially functional sufficient.

The functional sufficiency of a candidate component allows a developer to judge how suitable it is, from a purely functional perspective, for a certain usage scenario. We assume that all other factors being equal, developers would pick a functionally-sufficient component over a partially sufficient component since it already fulfills the requirements of the usage scenario, so theoretically could be reused without invasive changes to the component's code base.

However, when additional non-functional properties are taken into account (e.g. higher code quality in terms of maintainability), a partially sufficient candidate may be the most attractive. Nevertheless, the lower the *FS* the more changes typically need to be made to adapt the component. This, in turn, results in higher modification effort and risk of introducing errors [1], [11]. Developers therefore have to make a careful trade-off between the functional sufficiency of a component and the potential effort required to fulfill non-functional properties. At the end of the adaptation process, the *FS* should be 1. Our approach for measuring *FS* assumes that the usage scenario of a developer is encoded in test specifications (e.g. unit tests) using standard test-driven technology [6]. A comparison of the successful and failed test cases provides a reasonable approach for measuring functional sufficiency. However, we focus at the slightly finer level of granularity given by individual assertions, measuring the ratio of successful/failed test assertions (we assume each assertion indirectly maps to a desired piece of sub-functionality). Using this basic information *FS* can then be defined as the number of successful assertions defined in the usage scenario for a component divided by the total number of assertions specified for the usage scenario.

B. Superfluous Functionality

The functional sufficiency of a component measures the extent to which the functionality provided by a component fulfills the requirements of a particular usage scenario. But what about the functionality "wrapped up" in a component that is not needed by the usage scenario? It is also useful for a developer to know how much irrelevant or superfluous functionality would be included into the new code base without additional invasive changes, since this could impact the maintainability and verifiability of the component in the new application. To measure this we introduce the concept of *superfluous functionality* [12] (cf. Figure 1 b) which divides a component's delivered functionality into two parts: the needed functionality and the superfluous functionality with respect to a certain usage scenario. *Needed functionality* (*NF*) denotes the functionality which is required by a usage scenario, whereas *superfluous functionality SF* denotes all the offered functionality that does not play a role for the usage scenario. In other words, superfluous functionality (or "extraneous" functionality [1]) can be regarded as code that would "pollute" the new project's code base and therefore theoretically needs to be removed in order to achieve an ideal candidate for pragmatic reuse. We define the following set of superfluous functionality metrics on the internal functionality of a component for a certain usage scenario (details in [12]): *NF, SF, leanness, balance*. *NF* and *SF* denote absolute metrics and measure the amount of needed functionality and superfluous functionality respectively (i.e. aggregating method body weights, by the default "work done" which is defined as the count of instructions). Assuming a particular usage scenario, the *leanness* of a component is then defined as the ratio of the *NF* to the entire functionality, while the *balance* of a component is defined as the ratio of *SF* to the *NF*. Knowledge about the needed functionality can also be used to calculate adjusted versions of standard metrics which are calculated only on the functionality actually used in the scenario of interest, rather than on the entire functionality of a component. Figure 2 illustrates the various relationships that can exist between a component and a usage scenario in terms of functional sufficiency and superfluous functionality. In practice, whereas *FS* indicates the basic effort required to adapt a (partially) functional sufficient component for a certain usage scenario and level of quality, superfluous functionality indicates the potential level of polluting code, and thus the additional effort which would be required by the developer to remove it in the new project's code base. Thus, although a component may have a *FS* of 1, and thus needs no adaptation from a functional perspective, it may still be advisable to make invasive changes to it to reduce the *SF*.

C. Superfluous Interface Methods

Superfluous functionality is essentially a white box metric since it relies on the internal structure of a component with respect to the usage scenario. We complement this with a black

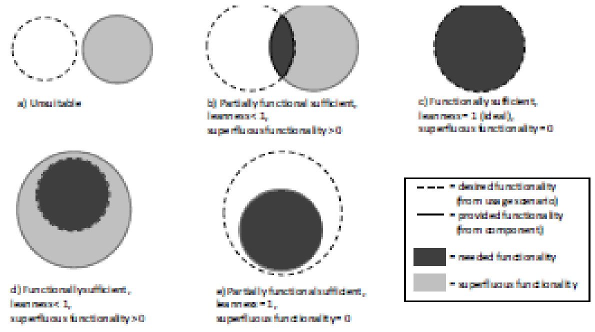
a) Unsuitable
b) Partially functional sufficient, leanness< 1, superfluous functionality >0
c) Functionally sufficient, leanness= 1 (ideal), superfluous functionality = 0
d) Functionally sufficient, leanness< 1, superfluous functionality >0
e) Partially functional sufficient, leanness = 1, superfluous functionality= 0
= desired functionality (from usage scenario)
= provided functionality (from component)
= needed functionality
= superfluous functionality

Fig. 2. Functional Sufficiency and Superfluous Functionality [12]

box metric, *superfluous interface methods* (*SIM*) (cf. Figure 1 c) which measures how many of the externally visible inter-face methods are superfluous. This is important, since future maintainers of the new code base will have to make decisions about how to call the reused component, and the availability of publicly visible but irrelevant methods in a component's interface not only raises the complexity of this task but also raises the likelihood of errors. The basic idea is to assess the functional cohesiveness of a component's external interface (i.e. the coherence of its exposed roles and responsibilities) relevant to the usage scenario. Cohesion metrics are a widely discussed topic in the area of software metrics and a variety of metrics have been proposed (e.g. [13]). Informally, we exploit the concept of cohesion by determining the ratio of the set of cohesive interface methods offered by a component for a certain usage scenario to the total number of interface methods offered by the component. For this, we pick a concrete cohesion metric and define a *min Shared Attributes* constraint, representing the minimal number of directly/indirectly shared attributes between two interface methods of a component.

The *SIM* metric is helpful input to the developer when deciding whether to remove superfluous functionality during the adaptation process. In general, publicly visible superfluous functionality is more dangerous than hidden superfluous functionality since it can lead to future errors in client code, not just errors in future versions of the component code.

D. Superfluous Tests

The code repositories used to populate software search engines frequently contain tests for software components as well as the components themselves. These can offer a rich source of information about the component since they embed information about its intended semantics, usually written by the developers of the components. An approach for using this information to help validate the correctness of the adaptation process has already been published in the literature [11]. How-ever, if only a portion of a component is needed (i.e. *SF* > 0), unrelated parts of the available test suite for the component become superfluous as well. The authors of [11] present a validation approach for pragmatic reuse tasks that actually make use of portions of the predefined test suites of the reused component to retain code quality. Unfortunately, when using this approach to prevent decreasing code quality and to decrease the likelihood of more failed test cases, increased effort is needed to fix and adapt related tests to functional code changes. We propose the concept of *superfluous tests* (*ST*) (cf. Figure 1 d) to measure the ratio of superfluous tests in the predefined test cases to the superfluous functionality delivered by the component for a certain usage scenario. The value of *ST* indirectly points to the effort needed to invasively adapt related test cases to cope with modifications to the original component's code base to retain the original level of quality (i.e. reliability). In other words, *ST* complements *SF* by considering predefined tests shipped with a component.

Like superfluous functionality, *ST* may be defined in numerous ways. At an intermediate level of granularity, one basic approach is to determine the number of predefined test cases related to the component's needed functionality for a certain usage scenario. This information could be then compared to the number of predefined test cases related to the component's overall functionality to determine the ratio of superfluous tests. Similar to superfluous functionality, a more fine-grained approach may be achieved by assigning a weight to each predefined superfluous test case to indicate its relevance.

Realizing ranking of software components

The metrics outlined in the previous section all indirectly indicate, in the early selection phase of pragmatic reuse, the relative effort needed to achieve certain levels of quality when reusing (partially) sufficient components for a certain usage scenario. The overall decision support offered to estimate the effort for the pragmatic reuse tasks [14] for each potential component candidate is illustrated in Figure 3. Each metric may be chosen, individually, as a separate ranking criterion to select a component candidate that best fits the developer's requirements. This selection must be made subjectively by each developer since there is no universally "correct" ranking criterion. Alternatively, the metrics can be used in combination to provide a multi-criteria approach to ranking components which delivers a more realistic indication of the potential effort involved in reusing a component for a certain usage scenario. In this case, a ranking approach needs to be capable of weighing each metric according to the user's preference (e.g. weighted sum). However, such user preferences may be hard to achieve, leading to conflicting objectives [12] and/or the ignoring of selected metrics. One possible way of overcoming this problem is to return a set of non-dominated component candidates based on Pareto efficiency [15], since this is a useful "tool" for analyzing potential trade-offs (conflicting objectives) between a set of metrics. Our vision is to let the developer explore component candidates interactively on the Pareto frontier by offering an intuitive visualization tool. Apart from Pareto-based rankings, a single-criterion ranking may also include information from additional metrics by providing extended visualization options. For instance by visualizing and adapting graphical properties of each result based on metrics.

Conclusion

In this short position paper we have presented our vision for new metrics and analysis techniques that can be built into

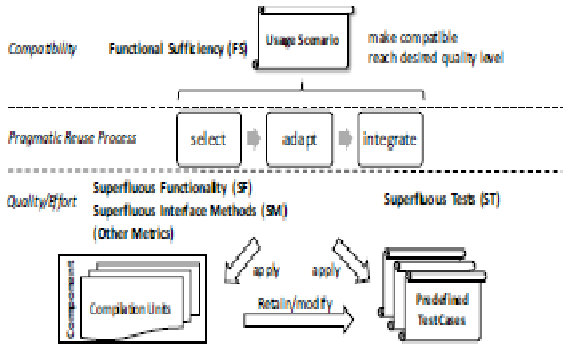

Fig. 3. Decision Support for Pragmatic Reuse Process

test-driven software search engines to help developers select software components for pragmatic reuse. The main novel contribution of this approach compared to previous work on software search engines is the explicit focus on supporting pragmatic reuse involving the possible invasive adaptation of components for new usage scenarios. The overall goal is to rank (partially) sufficient components according to the degree of effort that will likely be needed to adapt the component to fulfill the functional requirements to a certain level of quality. Our work is still at an early stage but preliminary results [12] support the utility of the approach and will be elaborated in future publications.

References

[1] S. Makady and R. J. Walker, "Validating pragmatic reuse tasks by leveraging existing test suites," Software: Practice and Experience, vol. 43, no. 9, 2013.

[2] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," Science of Computer Programming, vol. 79, pp. 241–259, 2014.

[3] S. P. Reiss, "Semantics-based code search," in Software Engineering, 2009. ICSE 2009. IEEE 31st Conference on. IEEE, 2009.

[4] O. Hummel, "Semantic component retrieval in software ngineering,"Dissertation, 2008.

[5] S. E. Sim and R. E. Gallardo-Valencia, "Introduction: Remixing snippets and reusing components," in Finding Source Code on the Web for Remix and Reuse. Springer, 2013, pp. 1–14.

[6] O. Hummel and W. Janjic, "Test-driven reuse: Key to improving precision of search engines for software reuse," in Finding Source Code on the Web for Remix and Reuse. Springer New York, 2013.

[7] B. Kitchenham, "What's up with software metrics?–a preliminary map-ping study," Journal of systems and software, vol. 83, no. 1, 2010.

[8] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in METRICS 2004. IEEE CS, 2004.

[9] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay, "Accurately interpreting clickthrough data as implicit feedback," in Proceedings of the 28th annual ACM SIGIR conference on Research and development in information retrieval. ACM, 2005.

[10] Roshan, D. Rohith, and K. Subba Rao. "LEAST-MEAN DIFFERENCE ROUND ROBIN (LMDRR) CPU SCHEDULING ALGORITHM." *Journal of Theoretical and Applied Information Technology* 88.1 (2016): 51.

[11] Kumar, Ravi, et al. "Reverse Engineering A Generic Software Exploration Environment Is Made Of Object Oriented Frame Work And Set Of Customizable Tools." *International Journal of Advanced Research in Computer Science* 2.5 (2011).

[12] M. Kessel and C. Atkinson, "Measuring the superfluous functionality of software components," 18th ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE), 2015, (to be published).´

[13] Kumar, Ravi, et al. "Active Scrutiny Techniques for the Reconstruction of Architectural Views." *International Journal of Advanced Research in Computer Science* 3.1 (2012).

[14] C. W. Krueger, "Software reuse," ACM Computing Surveys (CSUR), vol. 24, no. 2, pp. 131–183, 1992.

[15] Kumar, Ravi, et al. "System Testability Assessment and testing with Micro architectures." *International Journal of Advanced Research in Computer Science* 2.6 (2011).