Improving Testing Efficiency through Component Harvesting

Oliver Hummel, Colin Atkinson, Daniel Brenner and Sinan Keklik

Chair of Software Technology – University of Mannheim
68159 Mannheim – Germany

`{hummel, atkinson, dbrenner, keklik}@informatik.uni-mannheim.de`

Abstract. *Although code and component search engines have improved significantly in recent years, and the number of reusable components accessible over the Internet has increased dramatically, the "not invented here syndrome" and other human factors are still standing in the way of a significant increase in the levels of software reuse. However, it is still possible to use components discovered via search engines on the Internet to improve the efficiency of the development process, even if they are not embedded into final products. By providing an efficient way of obtaining multiple alternative implementations of a given piece of functionally, component search engines greatly reduce the effort involved in using "multi-version programming" and "back-to-back" testing techniques to enhance the testing process. These techniques have been proven effective in improving testing efficiency, but the effort involved in creating the multiple versions has hitherto been prohibitive. After explaining how these ideas can be combined with test-driven component harvesting approaches, the paper outlines some test hypothesis that can be used to evaluate their practical usability.*

1. Introduction

For many years the lack of effective search engines and sizeable component repositories has been a major obstacle to component-based reuse in mainstream software engineering [Seacord 1999]. However, recent developments in Internet search engines, and the open-source revolution have significantly changed the viability of component reuse [Brown & Booch 2002] and have for the first time tipped the balance in the "build or buy" tradeoff in favor of the "buy" option. For instance, websites like Sourceforge.net host more than 50.000 open-source projects with literally millions of components that can be discovered with the help of specialized search engines like merobase.com or krugle.com or even with general purpose search engines like Google or Yahoo as shown in [Hummel & Atkinson 2004]. For example, the number of Java files available through the search engines just mentioned is estimated to be more than 3 million [Hummel & Atkinson 2006].

Despite the increased ease with which components can be found on the Internet, however, systematic reuse of significantly sized components is still the exception rather than the rule. Where it does occur, reuse tends to be performed in an ad-hoc, copy-and-paste style with developers for example copying small code snippets from the results of web searches. There are many subtle reasons for the reluctance to reuse larger scale components, but they can all be wrapped up under the banner of the famous not "not-invented-here"-syndrome (NIHS). The basic problem is that developers inherently regard foreign software as difficult to understand and adapt, of dubious quality and potentially untrustworthy, and therefore prefer to implement the functionality from scratch. Another issue, particular with regard to open-source components in proprietary software systems are licensing restrictions. For example, a component released under a strong copyleft license forces developers to publish the whole system as open-source.

Since these are social rather than technical problems it is likely to be some time before they are resolved and the reuse of harvested components in new applications becomes more widespread. In the meantime, however, it is still possible to derive significant value from components harvested from the web by improving the efficiency of the process by which self-developed components are tested. This is the topic that we focus on in this paper. After outlining the basic principles of testing and describing the main related work in the next section we then proceed to outline different ways in which harvested components can improve the efficiency of the component testing process. We then identify some experimental hypotheses and techniques which might be used to validate their efficiency.

2. Component Testing

At its core component defect testing is no different to the testing of other forms of software [Beizer 1990]. An installed version of the component is executed with certain input values and the results are compared to correct results. However, herein lies one of the biggest and to date most intractable problems of testing - the problem of establishing what the "correct result" should be. Without the availability of an entity that can define what the "correct result" is for a given set of inputs, defect testing is not possible. In every day software engineering terminology this entity is referred to as an "oracle". Initially, human beings almost always play the role of the oracle in defining test cases for new applications or components. They do this by calculating by hand what the result of an operation invocation or sequence of operation invocations should be based on the required functional behavior of the component. Because this is expensive, however, only a limited number of test cases can usually be generated in this manner. This is why a large number of different heuristics and techniques are used to maximize the number of defects that a given set of test cases are likely to find.

Since human oracles are so expensive any technique that can make test cases (i.e. input data and correct results) available without the need for human intervention can drastically cut the cost of testing. Once a software application or component has matured and its functionality has become stable test cases from previous versions of the component can be reused. This is known as regression testing. Moreover, if the functionality of the new version is identical to that of the old version, the latter can be used as an oracle to generate the correct results corresponding to automatically generated test inputs. By definition, however, this technique will not work for new functionality. In this case the only way to create an oracle which can automatically generate "correct results" is to create an alternative implementation of the system. Multi-version programming [Avizienis 1995] popularized the idea of using multiple diverse versions of a software. The different versions are simultaneously executed and their results are compared. If there is no agreement among all the oracles multi-version programming utilizes voting approaches to determine the correct value. Back-to-back testing [Vouk 1990] is a more conservative technique applied at development time that requires a human to choose the correct result and to discover the fault that caused the deviation in any of the results. However, these approaches involve significant effort. Instead of writing test cases, human engineers have to develop the different versions of the system, with three being regarded as the minimum number required. Clearly, this is a cumbersome task that consumes a lot of time and effort so this approach tends to be used only for software with high reliability requirements.

2.1 Component Harvesting

The basic premise behind the testing approaches proposed in this paper is that it is possible to gain the advantages of multi-version programming and back-to-back testing without their overhead. The main idea is that instead of developing the multiple versions from scratch implementations can be "harvested" from large component repositories such as the Internet. The practically of harvesting software from the web even using general purpose search engines such as Google and Yahoo has already been reported [Hummel & Atkinson 2006]. Furthermore, in the last year several commercial search engines have emerged which specialize in discovering software on the Internet (e.g. krugle.com, koders.com, merobase.com). However, with the exception of the latter these all support only text-based searches for software based on the matching of search strings to source code. However, this is not sufficient for our purposes.

In order to harvest components to enhance the testing process it is necessary to be able to search for components with a given syntactic signature AND appropriate behavior specified with the help of test cases. This is the goal of the Extreme Harvesting (EH) approach [Hummel & Atkinson 2004] that enhances syntactic searching techniques with an additional filtering step to exclude all components which do not have the required semantics. This is done by executing test cases on each of the discovered components, and rejecting those that do not pass. Initial experiments have demonstrated that it is possible to obtain multiple implementations (sometimes more than 50) for numerous test examples. [Hummel & Atkinson 2006] also contains a more detailed explanation of the approach itself which we cannot discuss here for a lack of space. In order to be clear about different kinds of test cases in the following discussion, we will refer to the set of test cases that are used to harvest components using EH as the "search" test cases.

3. "Multi-Version" Harvesting

The simplest way of using EH in conjunction with a multi-version technique for oracle generation is to use an EH engine to harvest multiple versions rather than to build them from scratch. Thus, the tester would create the search test cases by hand and would then use them to harvest components with the desired functionality.

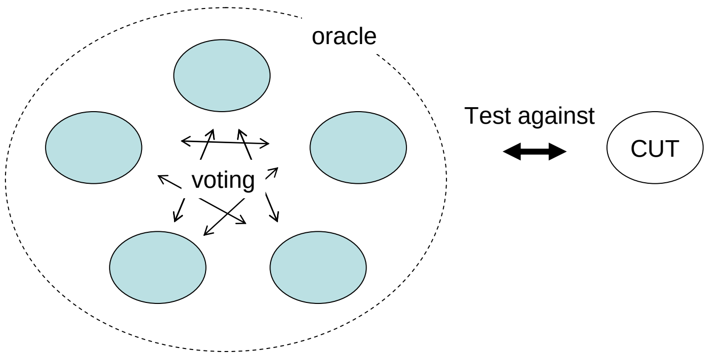

Figure 1. Multi-Version Harvesting

Figure 1 shows five harvested components (the shaded ones) which have been harvested using a hand generated set of search test cases. They are used according to the multi-version programming style to generate "correct results" for randomly generated test input data. The results generated by the component under test (CUT) are then checked against those generated by this pseudo-oracle. Although this approach still suffers from the problem that the faults in the multiple versions may not be independently distributed [Knight & Leveson 1986] in practice the chance that all of the different versions vote for the same wrong result can be significantly reduced by increasing the number of versions used in the voting process. With traditional multi-version programming this is extremely costly because each new version has to be developed by hand. With harvested versions it should be easy and cheap (provided that enough candidates are discovered).

4. Discrepancy-driven Testing

Nevertheless, even with a large number of harvested versions it is impossible to be 100% sure that the results generated by the voting process are correct. Thus, a more conservative strategy, as in back-to-back testing, is to refer back to the human "golden" oracle to arbitrate between disputed results. This approach, which we refer to as discrepancy-driven testing, is illustrated schematically in Figure 2.

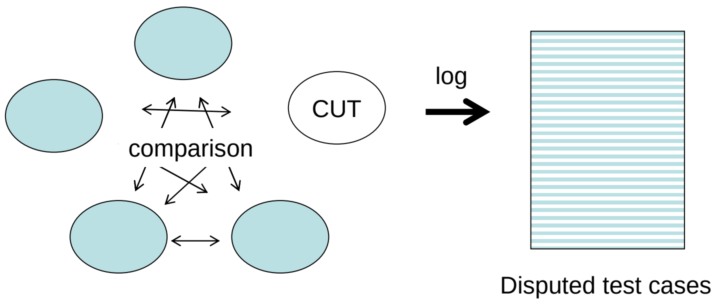

Figure 2. Discrepancy-driven Testing

As shown in the figure, harvested versions are tested with random input data as before, but we do not rely on them as oracles. Instead, whenever the candidates do not agree on one result and so-called discrepancies occur it is obvious that some anomaly has been uncovered and someone somewhere must have made a mistake. In other words, when two functionally identical components do not return the same result for identical input, at least one of them must be wrong and we have discovered a useful test case to execute with our own implementation. Disputed test cases are therefore logged and then are later inspected by a human engineer who decides which of the test cases the CUT failed to answer correctly. If the goal is to improve the quality of the CUT, as we are assuming here, the causes of erroneous results from the CUT are tracked down and fixed. Although this approach has the advantage that it reduces the possibility of erroneous “correct results” (beyond those due to human error), it has the disadvantage that the human engineer is once again involved and must spend effort in inspecting disputed test results. The fundamental question that must be answered with this approach, therefore, is whether a human engineer’s effort is more effectively invested in inspecting disputed test cases generated in discrepancy-driven testing or in generating new test cases in the traditional way. This is a question we shall return to in Section 5.

4.1 Discrepancy-driven Test Case Generation

The goal of the previous approach was basically to use ideas from back-to-back testing in conjunction with EH to try to uncover as many defects as possible in a CUT. Going further with this idea, however, it is possible to use this approach to improve the test cases for use in future tests of new version of the component, or of components with similar functionality. Rather than focusing on the CUT, therefore, with discrepancy-driven test case generation (DDTG) the focus is on the disputed test cases themselves. The basic premise is that test input data that results in a discrepancy between the oracles have uncovered a fault in one developers understanding or implementation of a component, and thus have identified a “hot spot” where developers tend to make mistakes. Together with a “golden” human oracle we expect to have an effective way to create good test cases. The following table shows some examples components, collected with our harvesting engine, and how they perform in comparison with a golden oracle.

Table 1. Various harvested components in comparison to a golden oracle.

Component	**n**	**c**	**f**	**c/n**
Greatest common divisor	25	2	23	8.00 %
Fibonacci	27	11	16	40.74 %
Leap year check	19	1	18	5.26 %
Bubblesort	50	46	4	92.00 %
Selectionsort	24	23	1	95.83 %
Mergesort	21	16	5	76.19 %
Quicksort	10	10	-	100 %
Sum up an array of numbers	6	4	2	66.66 %
Palindrome check	38	8	30	21.05 %

The table’s first column describes the type of component, the second contains the number n of components discovered, while the third and the fourth contain the number of correct (c) and faulty (f) components, respectively, in comparison to our golden oracle. Eventually, the last column shows the fraction c/n. We have taken two of the above examples, namely the greatest common divisor and the palindrome checking components to investigate them more closely and experienced encouraging results. We were interested in the quality of test cases generated with the above strategy and although, both of them have a rather low proportion of candidates that are always right there was at least one component for each test case (generated in relation to a discrepancy) that delivered the correct result. Under this assumption it would be guaranteed that discrepancy-driven test case generation would find all faults in the component under development. However, these results require more investigations and hence we are currently working on the empirical validation as described below.

All practical test case generation approaches need some exit criteria for determining when to stop the test data generation process. The discrepancy-driven test case generation approach can be used with three different test case generation exit criteria. The first exit criterion is simply based on the time consumed. New test cases are generated as long as more time is still available. However, for the tester it might be difficult to determine how long the execution of all components will take and thus how many test cases can be generated within a time period. Therefore, the second test case generation exit criterion is to specify the number of generated test cases directly. This is totally independent from the consumed time for the creation and independent from the number of occurred discrepancies. Finally, the last exit criterion is to specify the number of discrepancies. Test cases are generated until the given number is reached. In the best case a large number of test cases can be generated before the specified number of discrepancies is achieved. Clearly, all decisions depend on the available resources. The creation of new test cases is cheap since it can be automated, but the review of the discrepancies is expensive since it involves humans.

5. Experimental Evaluation

As mentioned in the previous sections, although discrepancy-driven testing and test case generation remove the risk of “false positives” in the testing process, they also require human beings to analyze the disputed test cases. The fundamental question which naturally arises therefore is whether human effort is better spent analyzing disputed test cases in the discrepancy-driven techniques or whether it is better spent developing traditional test cases. Fortunately, there are experimental techniques available for assessing the efficiency of testing techniques and test cases. The simplest approach is to test the same implementation of a given component using the two different approaches (discrepancy-driven versus classical) and analyze how many defects are discovered. Obviously, if there is a clear difference in the number this is an indicator of the greater effectiveness of one approach w.r.t. the other. A more sophisticated variation of this approach is so-called “fault injection” in which a version of the component is built with a number of faults deliberately added. Again, the basic idea is to compare the number of these faults uncovered using each technique. Another useful variation of the approach is “mutation testing” [Andrews et al. 2006] in which faults are inserted automatically by applying so-called mutation operations to a base version of the CUT. This technique has the advantage that many more errors can be inserted and thus the effectiveness of the different approaches can be better evaluated.

The basic hypotheses which need to be tested to determine the value of the discrepancy-driven approach are of the following form.

H1. Discrepancy-driven testing is more effective than Classical Testing (CT) in uncovering defects in a software component

For a given amount of human effort (i.e. number of test cases created or analyzed) the number of defects uncovered is greater when following a DDT approach than a CT approach.

H2. Discrepancy-driven test cases are more effective than classical tests cases in uncovering defects in a software component.

A given number of test cases, derived from disputed test cases in DDT, is able to uncover more faults than the same number of classic test cases.

In addition, the following hypothesis would be of interest.

H3. A hybrid mixture of discrepancy-driven and classical testing is better then either individually at uncovering errors in software component.

For a given amount of human effort (i.e. number of test cases created or analyzed) the number of defects uncovered in the hybrid approach is greater than in either the DDT approach or the CT approach alone.

A secondary goal of the experimental work would be to measure the effects of various parameters on the effectiveness of the discrepancy-driven approach, and to try to identify the best mix of DDT and CT in a hybrid approach. The key parameters are:

1. Number and quality of search test cases
2. Number of components in the pool

6. Conclusion

In this paper we explained how third-party components harvested from the Internet can be used to boost the efficiency of mainstream development projects even if they are not actually embedded into the final product. The basic idea is to use them to increase the efficiency of the testing process by reducing the human effort involved in creating effective test cases. Using techniques similar to multi-version programming and/or back-to-back testing, a group of harvested components can be used as a pseudo-oracle to calculate the "correct value" that should be returned for automatically generated input data or can flag "disputed values" where the results of one or more of the group deviate from the results of the self-built components. The notion of harvesting multiple components that are then compared against a self-written component also has the potential to improve the harvesting process itself. All searches engines, whether for code or for normal web pages, invariably try to rank the results that match a given query in order to try to offer the best components first. Using a form of voting system of the kind used in multi-version programming it should be possible to rank the components according to their statistical performance compared to all of the other discovered components. For example, the component that had the highest number of "correct" results when compared to the others (i.e. was in the majority voting subset the greatest number of times) would seem to be the best and so could be ranked highest.

The notion of discrepancy-driven testing also has interesting applications beyond simple defect testing (where the goal is simply to uncover as many defects as possible). Disagreements about the value that should be returned can arise because of misunderstandings in the contract to be fulfilled rather than in mistakes in the implementation. Discrepancy-driven testing may therefore be a good way of generating test cases focused on uncovering contract misunderstanding rather than implementation errors such as in Built-in Testing [Brenner et al. 2006]. Finally, as suggested by the third hypothesis above, the most likely scenario is that a hybrid mixture of discrepancy-driven and classic test case generation techniques will be the most effective. Thus, one of the goals of the experimentation program proposed in Section 5 is to identify the best mix.

References

Andrews, J.H., Briand, L.C., Labiche, Y. & Namin, A.S. (2006) “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”, forthcoming in: IEEE Transactions on Software Engineering.

Avizienis, A. (1995) “The Methodology of N-Version Programming“, In: Software Fault Tolerance, John Wiley & Sons.

Beizer, B (1990), Software Testing Techniques, International Thomson Press.

Brenner, D., Atkinson, Paech, B., Malaka, R., Merdes, M. & Suliman, D. (2006) “Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing”, to appear in: Proceedings of the Intern. EDOC Conf., Hong-Kong.

Brown, A.W. and Booch, G. (2002) “Reusing Open-Source Software and Practices: The Impact of Open-Source Software on Commercial Vendors”, In: LNCS 2319, Edited by C. Gacek, Springer.

Hummel, O. and Atkinson, C. (2004) “Extreme Harvesting: Test Driven Discovery and Reuse of Software Components”, in: Proceedings of the International Conference on Information Reuse and Integration (IEEE-IRI), Las Vegas, USA.

Hummel, O. and Atkinson, C. (2006) “Using the Web as a Reuse Repository”, In: Proceedings of the International Conference on Software Reuse (ICSR), Turino, Italy.

Knight, J.C. and Leveson, N.G. (1986) “An Experimental Evaluation of the Assumption of Independece in Multi-Version Programming”, in: IEEE Transaction on Software Engineering, Vol. 12, Iss. 1.

Seacord, R. (1999) “Software Engineering Component Repositories”, in: Proceedings of the International Conference of Software Engineering, Los Angeles, USA.

Vouk, M.A. (1990) “Back-to-Back Testing”, Information and Software Technology, Vol. 32, Iss. 1.