An Automated Hint Generation Approach for Supporting the Evolution of Requirements Specifications

Eya Ben Charrada
Department of Informatics
University of Zurich, Switzerland
charrada@ifi.uzh.ch

Martin Glinz
Department of Informatics
University of Zurich, Switzerland
glinz@ifi.uzh.ch

ABSTRACT

Updating the requirements specification during software evolution is a manual and expensive task. Therefore, software engineers usually choose to apply modifications directly to the code and leave the requirements unchanged. This leads to the loss of the knowledge contained in the requirements documents and thus limits the evolvability of a software system. In this paper, we propose to employ the co-evolution of the code and its test suite to preserve or restore the alignment between implementation and requirements: when a change has been applied to the code, subsequent changes in the test suite as well as failing tests are analyzed and used to automatically generate hints about the affected requirements and how they should be changed. These hints support the engineer in maintaining the requirements specification and thus ease the further evolution of the software system.

1. INTRODUCTION

Documentation associated with software is considered to be part of the software itself. Among various types of software documents, the requirements specification plays a key role for maintenance and evolution: first, requirements facilitate program comprehension, which is a crucial part of the evolution process. In fact, requirements give a high-level view of the functioning of a system which is easier to understand than code. Requirements also provide the rationale behind an implementation. Understanding how a system works from the structure of its parts only is very complex because the purpose or intent is omitted in the implementation [10]. Second, if the rationale is missing, important design and implementation decisions can be inadvertently undone during maintenance. Third, requirements specifications are accessible to all stakeholders as they do not require a technical background to be understood. Thus they can be used to discuss and negotiate change with stakeholders.
To remain useful, the requirements specification has to be maintained when the software system evolves. However, updating the requirements specification is a manual task. In the case of large requirements specifications, it is time-consuming and error-prone. Therefore, maintainers usually choose to apply modifications to source code directly and leave the requirements specification unchanged [15]. When the software knowledge contained in the requirements specification is lost, it becomes increasingly difficult to apply well-considered changes to the software [2].

In this paper we present a technique for automatically generating hints that guide the maintainers to efficiently update the requirements specification when the source code is modified. Our idea is based on using modifications in tests to get hints about changes in requirements.

The remainder of the paper is organized as follows. In the next section we motivate our work by presenting the limitations of current requirements update techniques. Then we present our idea of using tests as intermediate between requirements and implementation in Section 3. In Section 4 we present a classification of software change and discuss the type of hints needed in each case. The approach for hint generation is detailed in Section 5. Future work is presented in Section 6 while related work is discussed in Section 7.

2. LIMITATIONS OF CURRENT REQUIREMENTS UPDATE TECHNIQUES

Updating requirements specifications is a manual and expensive task. Ideally, when software engineers receive a change request, they should analyze the impact of the change on requirements, update the specification, then modify the implementation. This task is time-consuming because engineers have to do impact analysis twice: once at the requirements level and then at the source code level. In fact, as source code includes many details which are not present in the specification, analyzing the change impact on requirements only is not sufficient to detect all the parts of the code that need to be updated. Therefore, engineers usually choose to do all the maintenance work (analysis and modification) at the source code level directly and leave requirements specifications unchanged.

Another option to manage requirements evolution is to modify the implementation first, then propagate changes backward to the requirements specification. In this case, it is essential to have dependable traceability links between the requirements specification and source code. This type of traceability links, however, is difficult to define and to maintain. This is mainly due to the difference in structure between

IWPSE-EVOL '10, September 20-21, 2010 Antwerp, Belgium

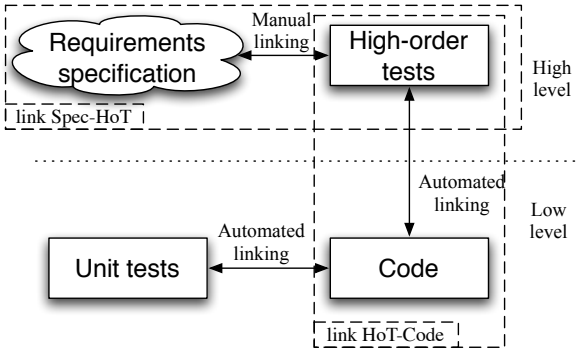

Figure 1: Relating requirements and code using high-order tests

requirements and implementation [5]: requirements usually represent end-user needs, while the implementation reflects many design and implementation details. There is also the problem of scattering (the implementation of a requirement is distributed over many classes) and tangling (one class contributes to the implementation of many requirements). This results in a very large number of links and makes the task of change propagation time-consuming and error-prone.

Another limitation of this approach is that modifications in source code are not always related to changes in requirements (e.g., refactorings). This makes the change propagation task more complex. Consequently, requirements documents are usually not maintained and briefly become obsolete and unreliable.

3. USING TESTS TO LINK REQUIREMENTS AND IMPLEMENTATION

In this paper we present a new approach for supporting the update of the requirements document during software evolution. Our approach builds on two observations:

- Tests are usually maintained with the implementation. For example, Lethbridge et al. [9] found that testing and quality documents are usually updated within a few days after changes are applied to a software system.
- Tests meant to check the external behaviour of the system are derived from requirements thus changes in these tests are usually related to changes in requirements.

Our basic idea is to use the test suite as an intermediate between requirements and implementation during software maintenance and evolution (Figure 1). We focus on tests meant to check the external behaviour of the system (e.g., acceptance and system tests). In the rest of the paper we use the term high-order tests, which was introduced by Myers [13] to refer to this type of tests. We analyze modifications applied to the test suite when the source code has been changed and use them to generate hints about changes in the requirements. Guided by these hints, a maintainer can update the requirements specification with little additional effort.

As high-order tests are derived from requirements, defining traceability links between the requirements specification and these tests is straightforward and can be done manually (Figure 1: Link spec-HoT). On the other hand, we can obtain the relation between tests and source code automatically when tests are executed against the implementation (Figure 1: Link HoT-Code). This makes high-order tests suitable for being an intermediate for propagating changes from implementation to requirements.

In the rest of the paper we assume that the following two conditions are satisfied in the considered software projects: (1) existence of a well-maintained high-order test suite with good requirements coverage and (2) existence of dependable traceability links between requirements and high-order tests.

4. CLASSIFICATION OF SOFTWARE CHANGE

In this section we explore different types of software change and discuss the types of hints needed for each of these changes. We base our work on the classification of software maintenance and evolution developed by Chapin et al. [4]. They distinguish twelve types of software change which are grouped into four clusters: (1) *support interface*, (2) *documentation*, (3) *software properties* and (4) *business rules*. In clusters (1) and (2), we find all maintenance tasks that do not affect source code. Examples of maintenance tasks in these clusters are software evaluation (*support interface*) and documentation update (*documentation*). These tasks affect neither the software implementation nor its tests, thus we do not address them in our work. Table 1 contains changes in *software properties* and *business rules*.

Table 1: Types of software maintenance

Cluster	Type of maintenance	Do we consider it?
Software properties	Groomative	No
	Preventive	No
	Performance	Yes
	Adaptive	No
Business rules	Reductive	Yes
	Corrective	Yes
	Enhancive	Yes

In the next paragraphs, we go through these seven maintenance types and briefly discuss the type of hint we expect to provide. The approach for hint generation is detailed in section 5.

4.1 Reductive Maintenance

A maintenance task is considered as reductive when it removes or limits existing system functionalities. If a functionality is suppressed, tests meant to check the functionality are removed, otherwise they fail. To update the specification, engineers need to know the requirements that have to be removed.

4.2 Corrective Maintenance

The goal of corrective maintenance is to fix existing functionalities or make them more precise. Corrective maintenance

improves the conformance of the system to the specified requirements; thus it does not imply changes in the requirements document. At the test level, we might add a few tests to check that a fix works correctly. Adding new tests can also be due to the addition of a new functionality (enhancive maintenance). Our hint generation approach needs to differentiate between these two types of maintenance. This point is further discussed in section 5.3.

4.3 Enhancive Maintenance

Enhancive maintenance is the most common type of maintenance in the business rules cluster. It includes replacing, adding and extending system functionalities. We treat the cases of addition and replacement separately.

Replacing functionality. When a functionality is modified or replaced, tests related to the functionality have to be modified correspondingly because they would fail otherwise. The hint we expect here is the identification of the requirements to be updated in the specification.

Adding functionality. Addition of functionality is usually followed by the addition of tests covering the new functionality. To update the requirements specification when new functionality is added, the maintainer needs to know what the new requirements are about. It is also interesting to know the relation between the new requirements and old ones because this gives information about the context of the new requirements and helps understanding them. This might also be helpful when establishing the traceability links between the new tests and the requirements specification.

4.4 Groomative and Preventive Maintenances

Both groomative and preventive maintenance affect the maintainability of the software. Although maintainability might be required by stakeholders, it is usually not possible to test it. Therefore these types of maintenance have no effect on the test suite and are not considered in our work.

4.5 Performance Maintenance

Performance maintenance changes the system performance, which is a non-functional requirement. Similarly to the replacement of functionalities (enhancive maintenance), the hint we expect here is the identification of the requirements affected by a change.

4.6 Adaptive Maintenance

A maintenance task is adaptive when it affects the technologies or resources used by the system. Adaptive maintenance might not require modifications in tests. For example, it is usually possible to check that a system works correctly on two different platforms, by running the same tests on each of these platform. Therefore, generating automated hints about adaptive maintenance from tests might not be feasible. In our current research we do not consider adaptive maintenance, but we will address it in future work.

5. APPROACH FOR HINT GENERATION

In this section we present our approach for hint generation. We detail the approach in the case of enhancive maintenance, which is the most common type of software change among the ones we are considering [4]. Generating hints for other types of maintenance is discussed briefly in Section 5.3 and will be elaborated in future work.

5.1 Automatically Identifying Requirements Affected by Change

In this section we define rules for generating the set of requirements affected by a change. The rules are based on the modifications applied to tests and on the traceability links between tests and requirements. We formulate the addressed problem as follows:

R is a set of requirements, T is a set of high-order tests and L a set of links relating elements in R to elements in T. After applying change to the software, T_c and R_c are subsets of T and R containing elements affected by the change. Our goal is to derive R_c based on T_c and L.

The main challenge faced here is that we usually do not have a one-to-one mapping between requirements and tests. In fact, one requirement might be checked by many tests and one test might check many requirements.
An intuitive way to get the set of requirements that need to be updated is to look for all requirements related to tests that have been changed and include them in R_c. The algorithm is the following:

for $t \in T_c$
 for $r \in R$
  if r is related to t then $R_c \leftarrow R_c \cup r$

This algorithm is very simple, but yields results having low precision: it generates many false positives resulting in much unnecessary work for the maintainer. A more sophisticated way to address the problem is to evaluate, after each change, the likelihood of a requirement to be affected. We have developed four rules for estimating this likelihood. The likelihood for a requirement to change (LC(r)) is represented using a number from 1 to 5, where each of these numbers represent one of the following categories:

5- The requirement is affected by the change
4- The requirement is likely to be affected by the change
3- We cannot decide about the requirement
2- The requirement is likely not to be affected by the change
1- The requirement is not affected by the change

Rule A
If a test $t \in T_c$ is related to only one requirement r, then r is affected by change: LC(r)=5.
This case is illustrated in Figure 2 (a), where LC(r_1)=LC(r_2)=5.

Rule B
If requirement r_k is related to a set of tests T_k, and $T_k \cap T_c = \emptyset$ then LC(r_k)=1
Figure 2 (b) is an illustration of the case.

Rule C
Consider a requirement r_k related to a set of tests $T_k = T_{km} \cup T_{ko}$ where T_{ko} are tests related to r_k only and T_{km} are tests related to r_k and other requirements (Figure 2 (c)). *If $T_{ko} \cap T_c = \emptyset$ and $T_{km} \cap T_c \neq \emptyset$ then r_k is not likely to change: LC(r_k)=2*

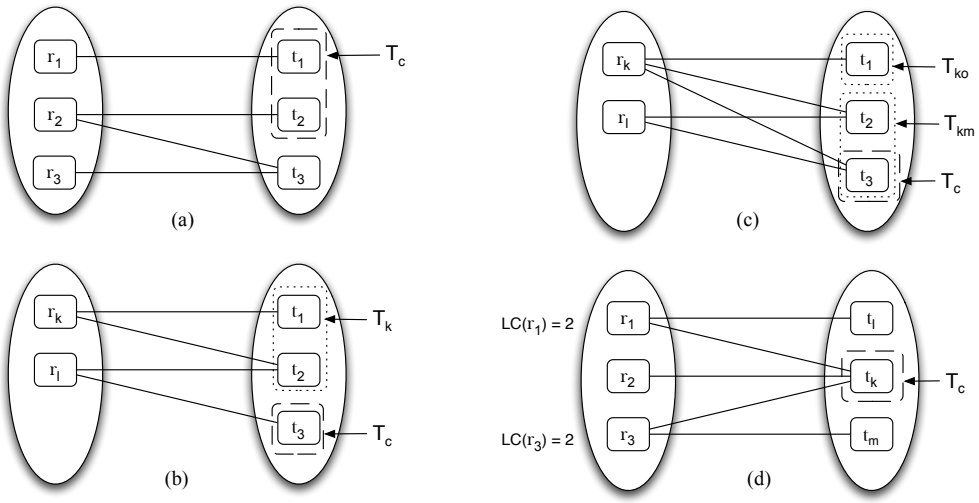

Figure 2: Identifying affected requirements

The rationale behind this rule is that if r_k changes, both T_{km} and T_{ko} should be affected by this change. If only T_{km} changes, then the change in tests is probably related to changes in other requirements (r_l in the case of Figure 2 (c)).

Rule D
Consider a set of requirements R_k = (r_i,..,r_m,..,r_p) related to a test $t_k \in T_c$. If all requirements other than r_m are not affected by change (LC(r)<3 for all r such as $r \in R_k$ and $r \neq r_m$) then r_m is likely to change: LC(r_m)=4.

The rule is illustrated in Figure 2 (d). Modifications in test t_k are due to changes in r_1, r_2 and/or r_3. As r_1 and r_3 are not likely to change (we calculate their likelihood to change by applying the previous rule, but the values might also be set manually by the maintainer) then the modification is probably due to change in r_2: LC(r_2)=4.

Applying the Rules
As the rules are not mutually exclusive, the order of their application matters. We apply the rules in the following order: A, B, C, D. If none of the rules apply for a requirement r, then we cannot decide whether r needs to be modified or not: LC(r)=3.

5.2 Automatically Generating Hints About New Requirements

When a new functionality is added to the system, we need to know what the functionality is about so that we can update the specification. We intend to get this information from the source code. In most cases, source code contains information about system functionalities and behavior. For example, method names usually reflect the purpose of the method, and its documentation contains extended details about the role of the method. However, as source code is usually huge, relevant information about system functionalities may be buried in a large number of design and implementation details. The challenge here is to extract the right information from the source code. We will exploit the information in the source code by analyzing test execution traces. By test execution traces, we mean the methods that are called when the test is running. Tracing all methods doesn not work because it would yield a very large number of irrelevant traces. Therefore we do a selective tracing: we only trace methods having names that are similar to words present in the newly added tests.
We illustrate our idea using an example of a simple library management system that manages borrowing and returning books. Suppose that maintainers add a new functionality for sending reminders to borrowers for returning books. Then they update the test suite by adding a test to check the functionality. In the test we will probably find the words "send" and "reminder". When analyzing the test execution traces, we might find a method called "sendReminder". The code and documentation associated to this method probably contain information about when and how this method is used.

Another relevant hint we can extract from execution traces is the relation between the new requirement and old ones. We propose to do this like in [6]. We consider overlaps between execution traces as indicators about relations between requirements: if tests t_1 and t_2 are related to requirements r_1 and r_2, respectively, and if execution traces of t_1 and t_2 overlap, then we deduce that r_1 and r_2 are related to each other.

5.3 Generating Hints for Other Maintenance Types

In the case of reductive and performance maintenance, identifying the requirements that need to be removed or modified can be done by defining identification rules in a way which is similar to what we did for functionality replacing (section 5.1). We might also use execution trace analysis and look for key methods that have disappeared from the traces to identify functionalities removed during reductive maintenance.

When new tests are added, we need to differentiate between corrective and enhancive maintenance. We to do this by

analyzing the test coverage: in the case of corrective maintenance, the coverage of the added tests is very similar to existing tests that are meant to test the same functionality. On the other hand, when new functionality is added, new tests cover newly added code.

6. STATE OF WORK

We have applied our approach to a simple library management system that we have developed as a testbed. First results show that the approach works, but the yield is still too low: we have many cases where the likelihood of change evaluates to three, i.e. it is not possible to give a hint. We are currently exploring the effect of considering the type of change applied to the test suite and the test coverage in improving the decisiveness of our rules. Our current work focuses mainly on enhancive maintenance. Other types of maintenance will be addressed by elaborating the ideas presented in section 5.3: new rules have to be defined to generate relevant hints for each type. Concerning the type of requirements we are considering, we started by analyzing changes in textual requirements specifications. We intend to cover also requirements expressed in modeling languages such as ADORA [7] or UML in the future.

The validation of our approach will be based on a case study. We will measure the precision and recall of the generated hints as well as the usefulness of these hints for updating requirements.

7. RELATED WORK

Our work relies on traceability links between requirements and tests. Much research has been done in the field of defining and updating traceability links. Antoniol et al. [1] and Hayes et al. [8] developed methods based on information retrieval models to generate traceability links between source code and documentation automatically. Egyed [6] uses trace analysis to semi-automatically find dependencies between requirements and generate links. Mäder and Gotel [11] developed an approach to update traceability links for UML models during software maintenance.

Our research subject is related to the problem of co-evolution of artifacts, which is also addressed in the work of Mens et al. [12] and Reiss [14]. However, these works focus on the co-evolution of design and implementation, while our focus is on requirements and implementation.

In order to solve the problem of obsolete or nonexistent requirements documents, Yu et al. [16] propose to reverse engineer requirements goal models from code. The main limitation of reverse engineering methods is that the generated artifacts are either imprecise or incomplete [3]. Thus we expect our approach to provide more dependable requirements.

8. CONCLUSION

We presented an automated approach to support requirements specification maintenance during software evolution. Modifications applied to tests are analyzed and used to generate hints about changes in requirements. These hints are expected to decrease the costs needed for updating requirements and thus help preserving the valuable knowledge contained in them. We focused in this paper on identifying requirements affected by change and getting hints about newly added requirements in the case of enhancive maintenance. Other types of changes, as well as an evaluation of the efficiency and effectiveness of our approach, will be addressed in future work.

9. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] K. H. Bennett and V.T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, 2000.

[3] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In *Future of Software Engineering*, pages 326–341, 2007.

[4] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.G. Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 325–339, 1999.

[6] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Trans. Softw. Eng. on Software Engineering,*, 29(2):116–132, 2003.

[7] M. Glinz, S. Berner, and S. Joos. Object-oriented modeling with ADORA. *Inf. Syst.*, 27(6):425–444, 2002.

[8] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Trans. Softw. Eng.*, 32(1):4–19, Jan. 2006.

[9] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Softw.*, 20(6):35–39, 2003.

[10] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. Softw. Eng.*, 26(1):15–35, 2000.

[11] P. Mader, O. Gotel, and I. Philippow. Enabling automated traceability maintenance by recognizing development activities applied to models. *International Conference on Automated Software Engineering.*, pages 49–58, 2008.

[12] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.

[13] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 1976.

[14] S.P. Reiss. Constraining software evolution. In *Proceedings of the International Conference on Software Maintenance, 2002.*, pages 162–171, 2002.

[15] E. Yourdon. *Just enough structured analysis.* http://yourdon.com/strucanalysis/, 2005.

[16] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J.C.S. do Prado Leite. Reverse engineering goal models from legacy code. In *Proceedings of the International Conference on Requirements Engineering*, pages 363 – 372, 2005.