ISSN 1744-1986

Technical Report N° 2008/ 08

Recovering Problem Structures to Support the Evolution of Software Systems

Thein Than Tun
Yijun Yu
Robin Laney
Bashar Nuseibeh

5th April, 2008

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

Recovering Problem Structures to Support the Evolution of Software Systems

Thein Than Tun Yijun Yu Robin Laney Bashar Nuseibeh
Department of Computing, The Open University, UK
{T.T.Tun, Y.Yu, R.C.Laney, B.Nuseibeh}@open.ac.uk

Abstract

Software systems evolve in response to changes in stakeholder requirements. Lack of documentation about the original system requirements can make it difficult to analyse and implement new requirements. Although the recovery of requirements from an implementation is usually not possible, we suggest that the recovery of problem structures, which in turn inform the problem analysis of new requirements, is feasible and useful. In this paper, we propose a tool-supported approach to recover and maintain structures of problems, solutions, and their relationships, for specific new features in an existing system. We show how these recovered structures help with requirements assessment, as they highlight early in the evolutionary development whether it is feasible to implement a new requirement. We validate our approach using a case study of a medium-sized open-source software system.

1 Introduction

Many software systems evolve to accommodate new stakeholder requirements. Without accurate documentation about the structure of an existing software system, it is difficult to address the problems imposed by the new requirements. For example, it is difficult to assess early in the development whether the new requirement is feasible, and can be implemented on the basis of extending existing solutions. One manifestation of this difficulty in many long-lived systems is that these systems often behave in unexpected and undesired ways when new features are implemented. In many application domains, this is often called the 'feature interaction' problem [8, 16].

This paper aims at recovering and maintaining the problem structures of existing system, so that the analysis of new requirements can be more informed about their feasibility and relation to the existing solutions. Towards this end, we have developed a tool-supported approach to recover solution structures of existing software systems in order to inform problem analysis of new requirements. We applied reverse engineering tools to automatically locate the solution structures relevant to a user requirement. On the basis of the recovered solution structures, we extended the reflexion model [26] to create the abstract problem structure for the existing solutions. Then we introduced structural similarity metrics to compare the extracted solution structures of different requirements and that of different versions in the evolutionary repository. The similarity metrics can also be used to compare related problem structures.

We applied this tool to study the evolution of an open-source text editor – Vim. Solution structures in terms of callgraphs form the basis to retrospectively analyze features, the implemented requirements, in various releases. For the requirements analysis, we use Problem Frames [21] to structure the context of the features in the problem space. Although applied to a system with particular characteristics, we believe our general approach may be easily adapted to systems in different implementation settings. The main contribution of the paper is a systematic tool support for the recovery of problem structures from the solution structures that can inform the requirements assessment.

The remainder of the paper is organized as follows. Section 2 uses example feature interaction scenarios of Vim to motivate the need for assessing new requirements. Section 3 overviews our approach to recover the problem structures to support for the evolution of software systems. Section 4 discusses our techniques and their application to the selected features of the Vim case study. Evaluation of our approach is presented in Section 5. Related work is surveyed in Section 6, whilst Section 7 provides some concluding remarks.

2 Motivating Examples

Before discussing the details and application of our approach, we first present a scenario of problem analysis in a typical development of an evolving software system, called Vim. Vim is selected for our study for three reasons: (i) it is representative of medium-sized open source software developments, (ii) it has a long history of feature-based evolution, and (iii) it has previously been studied by the reverse engineering community [28, 12].

2.1 Development history of Vim

Vim [1] is a popular open-source 'modal' text editing software, first released in 1991. The software has been evolving since then, and every release may introduce new features. For example, Vim 6 has more than 140 features, the next major release Vim 7 added 19 new features. These new features were incrementally developed and improved over minor releases 6.1, 6.2, 6.3 and 6.4. The recent major release Vim 7 has approximately 173K lines of C code, and recently a bug fix version Vim 7.1 has been released.

Features in Vim generally refer to user accessible system functionality in the solution space, and they can be identified from several sources. For example, features in a particular build of Vim can be identified from the preprocessing macros in the header `feature.h`, in which features have the prefix `FEAT_`. Besides the source code, documentation such as test cases, user manuals and release notes provide more detailed descriptions of the features. However, we found the preprocessing macros were most structured and therefore amenable to automated extraction.

Although there is some documentation about Vim features, the same cannot be said about its structure. Despite this, Vim is widely recognized as a good quality legacy software system, and has won several awards [2]. As in many instances of evolutionary development of legacy systems, introducing new features to Vim is difficult. According to the reference manual in Vim 7, it contains over 400 bug fixes: several fixes are related to minor errors but there were also serious bugs, many of which caused the system to crash. For example, there was a feature interaction between the Spell Completion, Spell Checking and Browsing features in Vim 6.4, which was a fatal bug[1].

There are, of course, several dimensions to addressing such problems: perhaps, if there were more programmers, more exhaustive test cases, more testing, some of these errors may not have entered the software. One such measure developers can take, this paper proposes, is to perform early problem analysis of new features. The paper illustrates such an approach using the following problem as running examples in this paper.

2.2 System crashes due to feature interaction

Vim has a feature "Completion", which can be used, for example, to match a partially completed file name to one of the existing files in a directory. This feature is triggered when a user presses Ctrl-X after keying "i", which is called "i_Ctrl-X" in the user documentation.

Vim has another feature "Spell checking", which highlights misspelled words according to the dictionary. This feature is triggered when a user enters the command ":set spell".

These two features were then put together into a new feature "Spell Completion". This composition seems straightforward enough: replace the directory look-up with a dictionary look-up, and introduce a new user command "i_Ctrl-X_s" to invoke the new feature by keying "s" after "Ctrl-X".

However, Vim crashes after a user invokes this new feature in a particular context: when the word under the cursor is at the beginning of a blank line, and then a third feature, "Window scrolling", is triggered by pressing "Ctrl-E". The complexity of the interaction among these features generates a bug, which could have been discovered earlier if one could anticipate the potential conflicts from the problem analysis.

2.3 The history of the involved features

To find the cause of the problem retrospectively, we looked up the evolutionary history of Vim development. The "Spell checking" feature had been requested since 2000, but the repository of Vim shows that developers did not implement this fully until version 7.0 in May 2006 although it was a high-priority pending feature request. This is an indication of the difficulty in implementing this feature. Even after it had been implemented in version 7.0, the above usage scenario still reveals a severe interaction with other existing features, including "Completion", "Window scrolling". The composed feature "Spell completion" was introduced as a subfeature of "Spell checking" in version 7.0.

In this paper, we present an approach to assess how a new requirement is related to existing solutions by systematically measuring the similarity between their structures.

3 Overview of Our Approach

A schematic view of this work is shown in Figure 1: we explain in the following subsections each of the four steps in our process to (1) extract solution structures at different levels of abstraction from various sources including test cases, usage scenarios and source code; (2) perform problem analysis by problem structures reflected from the abstract solution structures and the domain analysis; (3) compute similarity metrics between structures; and (4) assess new requirements using the similarity metrics of problem structures. Since the subject software systems are evolutionary, all artifacts shown in the figure can be associated with a release, such as N and $N + 1$.

[1] See the `runtime\doc\version7.txt` file of Vim 7.1.

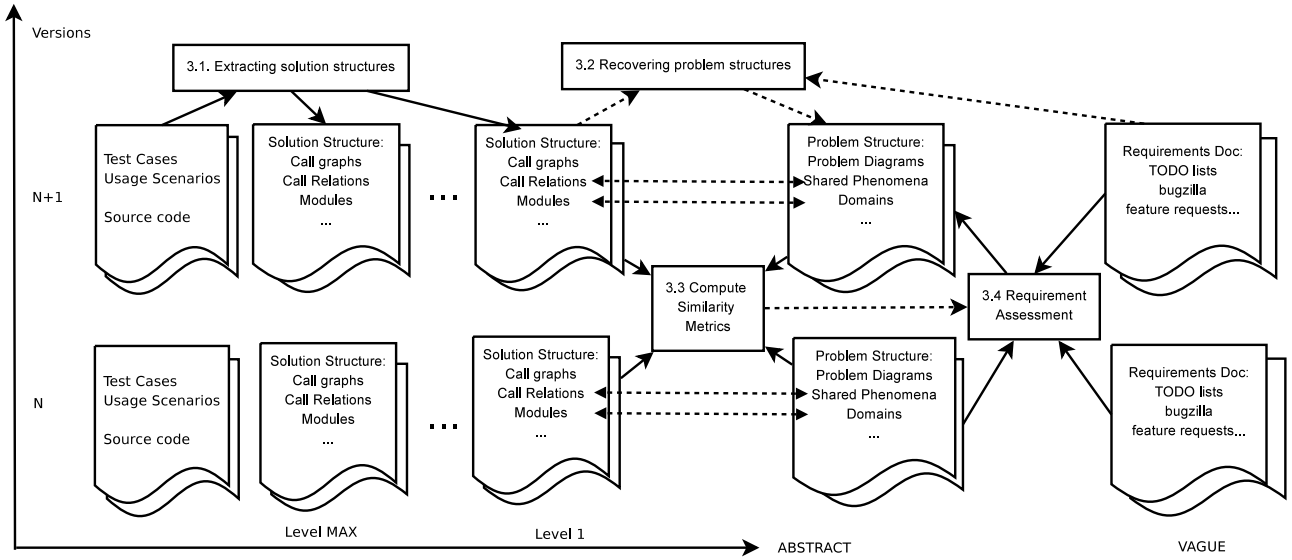


Figure 1. Overview of our approach

3.1 Extracting solution structures from features

Features refer to units of user accessible system functionality in the solution space. A release of evolutionary software system typically consists of a number of new features which address some users requirements. Meanwhile, development of such systems leaves behind for each release in the repository various solution artifacts, such as source code and test cases. However, the relation between the changes in the solution artifacts to the new features are often implicit. In order to relate user requirements to the solutions, it is therefore necessary to make the traceability between features and solutions explicit.

Given a particular feature, in our approach, we extract its related solution structures as callgraphs from the source code, test cases and usage scenarios. Since problems are often more abstract than solutions, in order to obtain problem structures in the end, we need to create abstraction from the extracted solution structures. Therefore, we systematically compute increasingly more abstract callgraphs from the concrete ones.

3.2 Recovering problem structures from abstract solutions

In many software development projects, requirements for new features can be recovered from documents such as "feature request list" (user requests for new features), "to do" list (a list of tasks developers intend to carry out) and the bug reports (errors reported by users). The Problem Frames approach is our chosen technique for problem analysis, but we envisage that other early requirements analysis technique with an emphasis on problem structuring may be used instead.

For each of the user requirements (features), we obtain an initial sketch of the problem structure in a *problem diagram*. In terms of problem frames, such a problem structure includes a list of *domains* that are connected to the requirement. These domains are also connected to each other through shared properties, also known as *shared phenomena*. In relation to the abstract callgraphs, a domain typically corresponds to an existing module whereas a shared phenomenon corresponds to an abstract call relation between the modules. This mapping helps with enriching the problem analysis by uncovering domains and phenomena hidden in the initial problem diagram.

3.3 Computing similarity metrics of structures

In order to assess the difficulty in employing existing solutions to implement a new requirement, we need to compare it with the solution structures of existing features. Yet the new requirement has not been implemented, therefore we need to compare the problem structures of the existing features with that of the new requirement. Such a comparison of problem structures should be informative, that is, predicting possible similarities between the solution structures.

In this study, we define a suite of similarity metrics on the problem/solution structures. Then we use the concrete examples from our case study to choose from these metrics

the most consistent ones. These similarity metrics will then be used for the requirements assessment.

3.4 Requirement assessment informed by similarity metrics of problem structures

In the assessment of a new requirement, we use the similarity metrics between its problem structure and that of other existing features. If a new requirement has a substantially higher degree of similarity to the existing structures than another new requirement, then we can inform developers that the former is more feasible to implement. If two features have no similarity, then it is an indication that they do not interact through function calls. On the other hand, if they share a large number of domain properties, then it indicates a high potential to have feature interaction problems. The similarity metrics may also help to investigate the patterns in the requirements evolution.

The four processes of our approach are supported by engineering tools. The extraction of solution structures and the computation of similarity metrics are fully automated; the recovery of the problems structures and the assessment of the requirements are interactive, supported by several graph visualization tools to highlight the characteristics of problem/solution structures.

4 The Case Study

In this section, we apply an instance of our approach to analyze the evolution of Vim features in our case study.

4.1 Extracting solution structures from features

Using callgraphs as solution structures, we aim to recover for each feature the relevant functions and modules systematically.

First, we instrument each release of Vim to record a trace of executed functions for the test cases of a certain feature. The instrumentation is done by using a tracing aspect through an aspect-oriented programming (AOP) tool, ACC[2]. The tool weaves into every function a statement to report the caller and the callee of the function call (e.g. *call* relation) together with the name of the module that contains the function being called (e.g., *contain* relation). A pseudo code of the aspect is shown as follows.

```
before(): call($ $(...)) {
/* outputting a call relation between
   this->funcName and this->targetName */
}
before(): execution($ $(...)) {
/* outputting a contain relation between
    this->funcName and this->fileName */
}
```

[2]http://research.msrg.utoronto.ca/ACC/WebHome

This tool is C-specific, but similar AOP tools for C++ and Java are also widely available.

Many software developers write test cases to regressively test the software. For example, Vim 7.0 has 62 test cases written by its developers[3]. These test cases are often specific to the key features of the software. For example, test cases 58 and 59 in Vim are related to the "Spell checking" feature.

For example, the "Spell completion" feature was tested by running the woven version of Vim 7.0. As a result, 367 distinct functions were called by 583 distinct call relations. Figure 2 illustrates the concrete callgraph at the lowest abstraction level. These functions are contained by 34

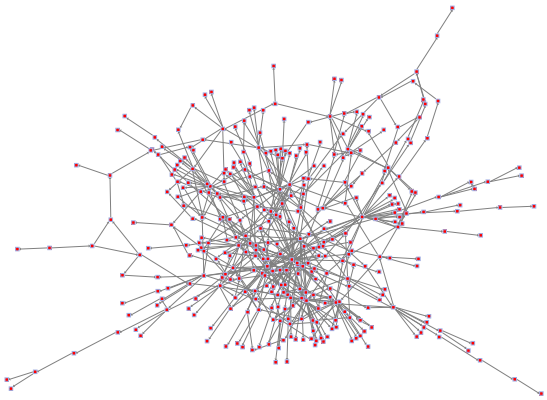


Figure 2. The callgraph at detailed level for the "Spell Completion" feature

file modules with 115 abstract calls between these modules. By abstract call, we mean that there is at least a call between functions contained in the caller and callee modules. Though more abstract, this callgraph is still too complex to be analyzed at the problem level. Therefore, we focused on high-level functions at level l, where l is the maximal length of the paths from the root to the leaf functions in the selected callgraphs. Figure 3 shows the abstraction of the same callgraph with only functions called within 2 levels. This callgraph now has only 19 modules and 16 abstract call relations.

4.2 Recovering problem structures

From the requirements document, we can give an initial sketch of the problem diagram for a user requirement. For the "Spell Completion" feature, the requirement is: "When user issues a command to automatically complete a word, replace the incomplete word with a matching word in the dictionary." The requirement also indicates the problem

[3]Test cases are typically found in the directory `src\testdir`.

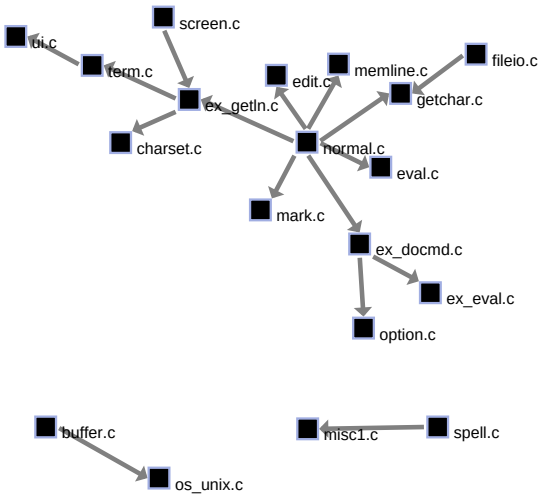


Figure 3. The abstract callgraph at level 2 for the "Spell Completion" feature

context with references to the *user*, *incomplete word*, *dictionary*, and so on. The requirement is rather vague: it does not say what happens if there is no word to complete. Such concerns can be addressed after considering the richer context of the problem reflected from the solution structures. By examining the abstract solution structures, we identify potential domains of the new feature and their relationships to existing ones.

In Figure 5, we present an example problem diagram for the "Spell completion" feature. In the diagram, the *requirement* is shown inside the dotted oval. The Keyboard User, Directory and Buffer represent the *domains* the feature interacts with. The *machine*, Spell Completion Feature, should bring about an appropriate change in the buffer at the user command so that a partial word is completed correctly according to the dictionary. The solid lines are decorated

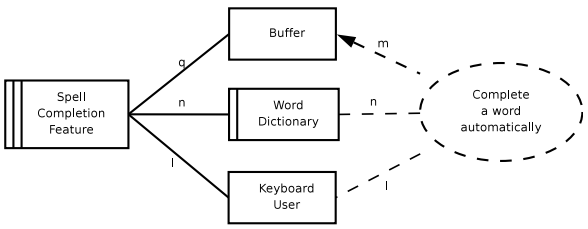


Figure 4. High-level Problem Diagram for Spell Completion Feature

by the labels of *shared phenomena*, elaborated below the diagram.

Informed by the solution structures, we can identify hidden domains and phenomena, for example, the "Screen" domain connecting to the "Buffer" was not considered in the initial problem analysis. The phenomena such as replaceWordAtCursor, getWordAtCursor were thus also uncovered. In addition to them, "i_Ctrl-X" and "s" are two hidden phenomena of the existing "Keyboard User" domain.

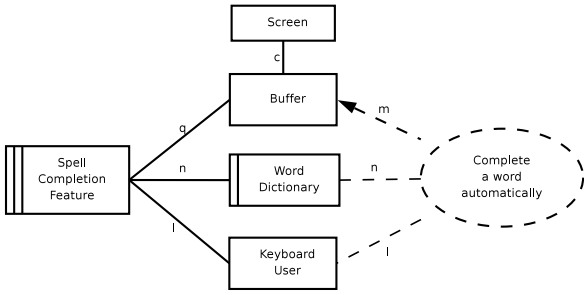
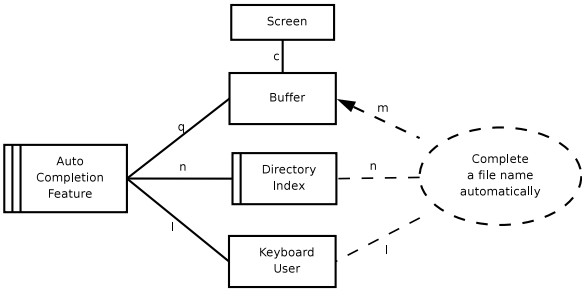


Figure 5. Enriched Problem Diagram for Spell Completion Feature

However, when two subproblems with a shared domain are composed, it is necessary to examine, for example, whether one subproblem will leave the domain in a valid state after it has accessed it. This is called a *composition concern* [21]. Vim developers, initially overlooked one *composition concern* that arises in this problem compositions. The phenomenon ReplaceWord leaves a null pointer in the Buffer when it is called to replace an empty word. An implicit assumption of the Screen feature is that the Buffer does not have a null pointer when the screen is refreshed. When these two features are invoked in that sequence, these features interact, resulting in a system crash. We draw a conclusion from this analysis: such errors could have been detected early in the development through problem analysis.

Figure 6 shows a problem structure similar to that of the "Spell Completion" feature. Comparing with Figure 5, the "Dictionary" domain has a phenomenon "FileList" instead of "WordList", and the "Keyboard User" does not have the phenomenon "s". We use this diagram to explain how we compute the similarity metrics.



c: B!{getWordAtCursor,replaceWordAtCursor}
l: KU!{WCCommand, i_Ctrl-X}
m: SCF!{PartialWord, CompletedWord}
n: D!{FileList}
q: ACF!{ReplaceWord}

Figure 6. Enriched Problem Diagram for Auto Completion Feature

4.3 Computing similarity metrics

As mentioned in the solution structure extraction step, two relations are created from the trace, namely *call* and *contain*. Such relations are kept in Rigi format [31] such that an efficient relation calculator such as crocopat [6] is used to compute the similarity metrics. A similarity is defined by the ratio of the size of the intersection set over the size of the union set.

$$R = \frac{|I|}{|U|}$$

where R, I, U stand for the ratio of intersection size over union size.

We define a similar metric ratio Rkind for each kind of set being compared, while kind is one of the following:

- **func**: a set of functions;
- **call**: a set of call relations;
- **file**: a set of file modules; and
- **acall**: a set of abstract call relations.

The above sets are derived from the two given relations *call* and *contain*, expressed as derived relations by the following rules in the Crocopat syntax:

```
function(x) := EX(y, call(x, y) | call(y, x));
file(x) := EX(f, contain(x, f));
acall(x, y) := EX(f, g, call(f, g)
            & contain(x, f) & contain(y, g));
```

where EX is an existence operator.

For any two versions v1 and v2, the above relations are expanded with one term such that they can be computed together. For example, function(x) is expanded to function(v, x).

The above metrics are thus computed as follows:

```
Ifunc(v1,v2,x):= function(v1,x) & function(v2,x);
Ufunc(v1,v2,x):= function(v1,x) | function(v2,x);
Rfunc(v1,v2):= #(Ifunc(v1,v2,x))/#(Ufunc(v1,v2,x));
```

Here # is an operator in crocopat to compute the size of tuple sets.

Furthermore, we compute subcallgraphs at different abstraction levels, as defined inductively by the depth from the root functions: a function is at the depth level l, if and only if it is at the depth level $l-1$ or being called by a function at the depth $l-1$, when $l > 1$. When $l = 0$, only the root functions, those without callers are considered. The MAX level degenerates to the whole call graph.

```
root(MAX, v, x) := function(v, x);
root(0, v, x) := function(v, x) &
   ! EX(y, call(v, y, x));
root(n+1, v, x) := root(n, v, x) &
   ! EX (y, !root(n, v, y) & call(y, v, x));
call(l, v, x, y) := call(v, x, y)
   & root(l, x) & root(l, y);
acall(l, v, x, y) := EX(f, g, call(l,v,f,g)
   & contain(v, x, f) & contain(v, y, g))
```

On the basis of these subcallgraphs, we can compute the similarity metrics again. The aim is to check whether an abstract problem structure can be mapped onto a concrete solution structure, without affecting the similarity metric. If the similarity metric is useful to establish the problem structure to solution structure, then the two abstractions will not look much different when more detailed functions are included.

When a module is not added or deleted, the previous metrics cannot tell whether the functions inside the module are changed. Therefore we define two more relations to show the exact similarity ratios for the modules *file* and the module relations *acall*:

```
IfileExact(l, v1, v2, f) := Ifile(l, v1, v2, f)
   & ! EX(x, root(l, v1, x) &
     contain(v1, x, f) & !contain(v2, x, f))
   & ! EX(x, root(l, v2, x) &
     !contain(v1, x, f) & contain(v2, x, f));
IacallExact(l,v1,v2,f,g) := Iacall(l,v1,v2,f,g)
        & ! EX(x,y,call(l, v1, x, y) &
          contain(v1, x, f) & contain(v1, y, g) &
          ! (contain(v2, x, f) & contain(v2, y, g)))
        & ! EX(x,y,call(l, v2, x, y) &
          contain(v2, x, f) & contain(v2, y, g) &
          !(contain(v1, x, f) & contain(v1, y, g)));
```

Informed by the similarity metrics between different requirements in the same version and between different versions of same requirement, one can see whether it is useful to recover an initial problem structure.

In terms of problem frames, a problem domain corresponds to a module, and a shared phenomenon corresponds to an abstract call between two modules. For example, the similarity metric between Figure 5 and Figure 6 problem structure is 1.0 for the exact shared phenomena and 0.6 for

the exact domains. This shows that the two features are similar.

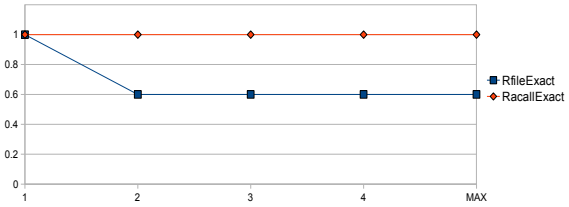


Figure 7. The similarity metrics for diagrams in Figures 5 and 6

5 Evaluation of Our Approach

We evaluated our approach[4] by looking both horizontally at different requirements within the same version as well as vertically at the evolution of requirements over different versions.

We first instrumented every released binary of Vim by the tracing aspect. Then we extract execution traces for each feature by automated testing. We excluded the functions beyond the scope of testing, such as the initialization and finalization parts of the log, such that only function calls relevant to the functionality of the feature are recorded.

For any pair of the traces, we can measure their similarity at abstraction levels ranging from 1 to MAX, where MAX is the longest depth of the refinements.

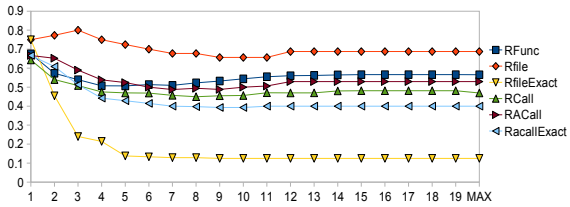


Figure 8. All similarity metrics for the ":set spell" feature between versions 6.4 and 7.1.

Figure 8 presents all similarity metrics we defined in Section 3.3 for the spell checking feature ":set spell" between releases 6.4 and 7.1. According to the documentation of Vim, the spell checking feature is introduced in 7.0 and it is one of the biggest features that distinguish the major release 7 with 6. The test case thus should invoke different functionalities: when entering ":set spell" command, a command error is reported in 6.4, whilst a spell check is called in 7.1. Yet both test cases should share some phenomena in the "Keyboard User" domain. The similarity metrics confirm this fact. Moreover, from Figure 8, one can see that the ratios Rfile and Racall are sometimes above 0.5, indicating an over-estimate of the similarity. Two metrics we defined RfileExact and RacallExact would exclude the files and abstract calls when they contain different sets of functions or function calls respectively. Therefore, in the remaining presentation, we only present these two metrics for brevity.

For ":set spell" feature, Figure 9b compares releases 7.0 with 7.1 additionally. Also in addition, we show some comparison of different features for the same release 7.0. Conceptually, the RfileExact metric reflects domain similarity in the solution structures where the RacallExact metric reflects shared phenomena similarity in the solution struc-

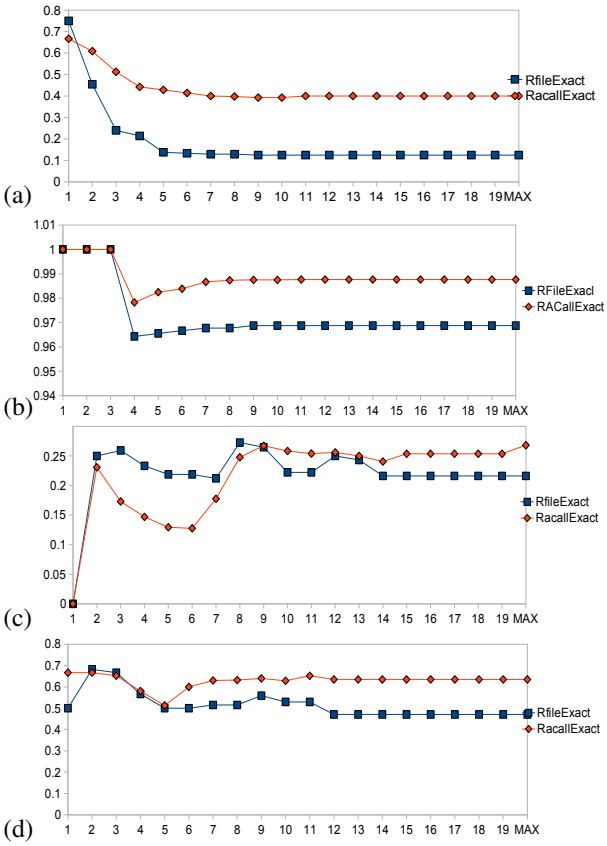


Figure 9. Exact similarity metrics: for the same ":set spell" feature compare similarity betwen (a) versions 6.4 and 7.1 (b) versions 7.0 and 7.1; for the same release 7.0, compare similarity between (c) ":set spell" and "i_Ctrl-X" (d) ":set spell" and "i_Ctrl-X_s" .

[4]All data and analysing scripts are available for download at http://mcs.open.ac.uk/yy66/vim-analysis.html.

tures. The similarity in the first case is as low as 0.11 in terms of RacallExact, indicating that other than the common subproblem of issuing the ":set spell" command, there is little in common between the two versions. On the other hand, the similarity between 7.0 and 7.1 is above 0.95, indicating that the feature implemented in 7.0 is very similar to that of of 7.1, which confirms the Vim documentation that "7.1 is a bug fix release of 7.0".

On the other hand, the comparison of two features in the same release is also interesting, which may show how similar two requirements may look like. In Figure 9c and 9d we show two pairs of such comparison. The similarity between the features "Spell Completion" (i_Ctrl-X_s) and "Spell checking" is significantly higher than that between "Completion" (i_Ctrl-X)and "Spell Checking" (":set spell").

Comparing horizontally the different abstraction levels, Figure 9 reveals that starting from the most abstract structure (with only one level function call), until the MAX concrete solution structure, all the metrics are smoothly converging. And the variation from the converged MAX metric by metrics of abstract level 3 is already very narrowed. This observation indicates that by analyzing the solution structures at a high level, one may have a better idea about the similarity of the detailed solution structures. Informed by this observation, therefore one may consider mapping the abstract solution structures onto initial problem domains for a problem analysis.

Finally, we perform a pair-wise comparison for the features we explained in our earlier running example. The results are compiled into correlation matrices in Table 1.

From Table 1, one can observe that the similarity metrics do provide a good indicator whether two features are similar in the solution structures. Therefore, we believe that one may apply these metrics on the problem structures for early assessment of new requirements. If the requirements have large similarity in terms of the metrics to the existing solution structures, then there is good chance that the implementation could share largely with existing solutions.

The metrics on the problem domains can be similarly defined as RdomainExact and RsharePheonomenaExact where each domain is a node (similar to the file modules in the case study) that may contain a number of domain properties (phenomena). When the phenomena are shared by domains, a link is formed similar to the call relation in this case study. We have measured these metrics on the problem diagrams early to find similar patterns as the solution structure. It is future work to measure them for more case studies and different types of solution structures to see if the hypothesis of correlation in problem/solution structure holds in general.

Table 1. Correlation matrices between feature similarity metrics

(a) Abstraction Level = 3

RfileExact	i_Ctrl-X	setspell	i_Ctrl_E	i_Ctrl-X_s
i_Ctrl-X	1.00	0.26	0.20	0.38
setspell	0.26	1.00	0.15	0.67
i_Ctrl_E	0.20	0.15	1.00	0.18
i_Ctrl-X_s	0.38	0.67	0.18	1.00

(b) Abstraction Level = MAX

RfileExact	i_Ctrl-X	setspell	i_Ctrl_E	i_Ctrl-X_s
i_Ctrl-X	1.00	0.22	0.11	0.22
setspell	0.22	1.00	0.19	0.47
i_Ctrl_E	0.11	0.19	1.00	0.18
i_Ctrl-X_s	0.22	0.47	0.18	1.00

(c) Abstraction Level = 3

RacallExact	i_Ctrl-X	setspell	i_Ctrl_E	i_Ctrl-X_s
i_Ctrl-X	1.00	0.17	0.25	0.46
setspell	0.17	1.00	0.29	0.65
i_Ctrl_E	0.25	0.29	1.00	0.23
i_Ctrl-X_s	0.46	0.65	0.23	1.00

(d) Abstraction Level = MAX

RacallExact	i_Ctrl-X	setspell	i_Ctrl_E	i_Ctrl-X_s
i_Ctrl-X	1.00	0.27	0.16	0.44
setspell	0.27	1.00	0.22	0.63
i_Ctrl_E	0.16	0.22	1.00	0.15
i_Ctrl-X_s	0.44	0.63	0.15	1.00

6 Related Work

A detailed survey on reverse engineering has been done in [10], so we will focus here on approaches using dynamic analysis [5] for feature extraction [14]. The approach of Wilde and Scully [30] uses dynamic analysis as a way of identifying code belonging to a given feature. Wong et al. [31] give automated metrics for determining the relationships between features and code. Eisenberg et al. [13] uses dynamic analysis to rank how likely code is to belong to a given feature. Kothari et al. [22] use a similarity analysis to identify canonical sets of features. Antoniol and Gueheneuc [4] use a mixture of dynamic and static analysis and combine several dynamic approaches to deal with uncertainty. Rohatgi et al. [3] combine static and dynamic information to measure the impact of a code change associated with a feature on a given component, on the basis that this suggests how likely the component is to be associated with the feature. All of these approaches are aimed at classifying code rather than constructing a requirements model.

The work of Lui et al [25] focuses on decomposing a program into a base system and features determined by mathematical transformations of that base. The strength of their approach is in providing a formal algebraic basis to un-

derpin feature extraction. However, their work is aimed at support for program transformation rather than the capture of requirements artifacts. One of the few works using dynamic analysis to reverse engineer source code with a view to establishing traceability between problem and solution domains is the PhD work of Greevy [14]. That work differs from ours in that the goal is to extract a meta model of the source code expressed using their language Dynamix.

Our work is related to the concept assignment problem associated with program comprehension [7], which addresses the discovery of human oriented concepts associated with code. Biggerstaff proposes an approach to this problem based on machine reasoning that is opportunistic and non-deterministic rather than algorithmic. Somewhat in contrast to our work they assume the need to rely on a priori knowledge of the problem domain.

There are a number of RE approaches that can be used to investigate problem structures. The goal-oriented approach KAOS refines high-level goals into sub-goals and then into operational requirements [29]. Operational requirements are then assigned to agents in the solution space [24]. Using a similar notion of goals and goal-refinement, the NFR and i* framework discuss how goals may contribute to achieving software quality [11, 32]. Based on refactoring technique, a reverse engineering technique was applied to source code to recover goal model structures[33], where the focus is mostly on the structures in requirements. The work presented here focusses on recovering structures in the problem context which is complementary to the requirements structure as exhibited in goal refinements. It is our view that the problem structure can also be informed by the structure of requirements and vice versa.

In this paper, we chose the the language and techniques of the Problem Frames approach (PF) [20] to describe problem structures because (i) it provides mechanisms to consider solution components in the problem space [15], (ii) it allows known solution structures to influence problem decomposition [27] and (iii) it can recompose subproblems using a composition operator [23].

Acknowledging that the term feature has specific meanings in specific areas of research such as product-line engineering, and feature interactions, we adopted the general notation of feature as a unit of system functionality that is "user accessible" [19]. There is a considerable literature on feature interaction problems in telecommunication [9] and other application domains such as email [17]. Hay and Atlee [18] treat feature interactions as a more general software problem and discuss a feature composition approach to resolve them.

7 Discussions and Conclusions

This paper examined the difficulty of adding new features to legacy software systems with arbitrary structures. We showed that some of the fatal bugs in software result from structural mismatch between the existing software structure and that of the new feature. Many legacy systems do not have accurate documentation about system structures and this paper proposed a tool-support approach to recover such information. Furthermore, the paper also showed that analysis of the recovered problem structures can be useful not only in identifying structural conflicts early in the development, but also in an assessment of how difficult implementation of a feature could be. We proposed a requirements metric which can be used to measure similarities between problem structures. Our study of Vim has shown that there is indeed significant relationships between problem structures of new features, and the amount of code restructuring and coding their implementation involve. These results persuaded us that the approach we have proposed has practical usefulness.

This work, however, can be improved in several ways. We have applied the approach to examine the historical artefact of Vim. Our analysis has been retrospective, and we have not explored in detail how predictive this approach is. This could be done, for example, by identify bugs in the latest release of Vim, and having them validated by developers of Vim (to see if what we identified as structural conflicts are recognized as bugs), and to examine the current "to do" list to assess the difficult of implementing some of these features. To enable such a study, recovered models need to be managed with the assistance of tool-support.

In order to show wider applicability of this approach, applications with different characteristics other than those of Vim need to be studied too. There are several open-source candidates for this line of work, such as Eclipse and Firefox. We are currently perusing these research questions.

References

[1] http://www.vim.org/.

[2] http://www.vim.org/about.php.

[3] R. A., H.-L. A., and R. J. Feature location based on impact analysis. In *Proceeding (591) Software Engineering and Applications*, 2007.

[4] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: A novel approach and a case study. In *ICSM*, pages 357–366. IEEE Computer Society, 2005.

[5] T. Bell. The concept of dynamic analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234, London, UK, 1999. Springer-Verlag.

[6] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *TSE*, 31(2):137–149, 2005.

[7] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. The concept assignment problem in program understanding. In *ICSE*, pages 482–498, 1993.

[8] E. Cameron, N. Griffeth, Y.-J. Lin, M. Nilson, W. Schnure, and H. Velthuijsen. A feature-interaction benchmark for in and beyond. *Communications Magazine, IEEE*, 31(3):64–69, 1993. TY - JOUR.

[9] E. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, 31(8):18–23, 1993.

[10] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[11] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[12] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos. Improving the build architecture of legacy c/c++ software systems. In *FASE*, pages 96–110, 2005.

[13] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[14] O. Greevy. *Enriching Reverse Engineering through Feature Analysis*. PhD thesis, University of Bern, 2007.

[15] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 137–144. IEEE Computer Society, 2002.

[16] R. J. Hall. Feature interaction in electronic mail. In M. Calder and E. H. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, Glasgow, Scotland, UK, 2000.

[17] R. J. Hall. Feature interaction in electronic mail. In M. Calder and E. H. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, Glasgow, Scotland, UK, 2000.

[18] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119, New York, NY, USA, 2000. ACM Press.

[19] I. Hsi and C. Potts. Studying the evolution and enhancement of software features. In *16th IEEE International Conference on Software Maintenance (ICSM'00)*, pages 143–151, 2000.

[20] M. Jackson. *Problem Frames*. ACM Press & Addison Wesley, 2001.

[21] M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. ACM Press & Addison Wesley, 2001.

[22] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 93–102, Washington, DC, USA, 2006. IEEE Computer Society.

[23] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proceedings of 12th IEEE International Conference Requirements Engineering (RE'04)*, pages 122–131. IEEE Computer Society, 2004.

[24] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 83–93, New York, NY, USA, 2002. ACM Press.

[25] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM.

[26] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *TSE*, 27(4):364–380, 2001.

[27] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *12th IEEE International Conference on Requirements Engineering (RE 2004)*, pages 80–89, 2004.

[28] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architectural repair of open source software. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, page 48, Washington, DC, USA, 2000. IEEE Computer Society.

[29] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of Fifth IEEE International Symposium on Requirements Engineering, 2001*, pages 249–262, 2001. TY - CONF.

[30] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[31] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, 2000.

[32] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 159–168, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[33] 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 *RE*, pages 363–372, 2005.