On Dynamic Feature Location

Rainer Koschke and Jochen Quante
University of Bremen, Germany
{koschke|quante}@informatik.uni-bremen.de

ABSTRACT

Feature location aims at locating pieces of code that implement a given set of features (requirements). It is a necessary first step in every program comprehension and maintenance task if the connection between features and code has been lost.

We have developed a semi-automatic technique for feature location using a combination of static and dynamic program analysis. Formal concept analysis is used to explore the results of the dynamic analysis.

We describe new experiences with our technique. Specifically, we investigate the gain of information and increase of costs when the system under analysis is profiled at basic block level rather than routine level as in our earlier work. Furthermore, we explore the influence of the scenarios used for the dynamic analysis (minimal versus combined scenarios).

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Experimentation

Keywords

Feature Location, Formal Concept Analysis

1. INTRODUCTION

Suppose you were a new maintainer of a compiler who is not yet fully familiar with its implementation. How would you proceed if a maintenance request asks you to add a new `repeat` loop to the language supported by the compiler? Suppose further that this new loop is similar to the `while` loop that is already implemented by the compiler. You would likely want to look at the implementation of this `while` loop. But, what if the documentation of this compiler is lousy and you do not know which pieces of code implement this kind of loop?

In far too many cases, it is not known to a programmer which parts of the code implement a certain feature or set of features of interest, although this information is essential to any change.

Features are product functions as described in a user manual or requirement specification, something that a user wants the system to do and that is actually implemented. The feature can be observed by the user.

The maintainer wants to know the relevant pieces of code that implement a feature. These can be individual statements – both executable and declarative – as well as routines or coarser elements such as classes or modules. As a unifying term, we will refer to these elements as *units*. *Feature location* is the process of locating units that implement these features.

Several techniques have been proposed to remedy the regrettable state of affairs that the mapping of features onto units is not known. These techniques can be roughly categorized into static, dynamic, and hybrid techniques depending on the kind of information they consider. Static techniques are based on any information that can be retrieved from the system without executing it. Dynamic techniques gather their information through runtime analysis. Hybrid techniques combine the two.

The advantages and disadvantages of static versus dynamic analyses are well known. Briefly, all dynamic techniques are safe only with respect to the input that was actually considered during runtime to gather the information; generalizing from these data may not be safe. On the other hand, static analyses – if being applied conservatively – yield safe information; but because many interesting properties of programs are statically undecidable in general, static analyses are bound to approximative solutions, which may be too imprecise in practice. Dynamic analyses yield the "lower bound of truth" while static analyses yield the "upper bound." Hybrid techniques try to find a middle way.

All dynamic techniques for feature location – including our own which is based on formal concept analysis [8] – take a set of test cases that exercise a feature of interest and gather the units that are executed for each test case. As such, they depend upon the quality of the test suite used. Moreover, these techniques may be applied at different levels of granularity: at the level of single statements, whole routines, or even bigger chunks such as classes. The lower the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'05, November 7–11, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-993-4/05/0011 ...$5.00.

level of granularity, the more detailed the analysis can be. The flip side of the coin is that the computational demand increases, too. This increase may be particularly hurtful for a technique based on formal concept analysis, whose theoretic worst case yields exponentially many concepts.

In this paper, we explore two things. Firstly, we apply our technique for feature location at the basic block level. Previously, we have applied it on routine level only. While conceptually the same, the technique at basic block level promises more detailed results. On the other hand, the number of basic blocks is larger by an order of magnitude over the number of routines. Formal concept analysis might fall short of handling this increase of data. We shall measure benefits, that is, gain of information, and costs, that is, execution time, for concept analysis.

Secondly, we explore the influence of the test data on the results. The assumption of feature location based on dynamic analyses is that we can find an initial set of test cases that is good enough to locate the feature of interest. Finding such a good set may require prior knowledge of the implementation. The scenario for feature location is that we do not know the system in all details – otherwise feature location would not be an issue in the first place – hence we can assume at most a rough understanding of the system. We shall measure the difference between an initial set of test cases (aiming at isolating features in the lattice) and an extended one that combines several features.

While the issue of granularity is an issue specific to feature location using formal concept analysis, the influence of test data pertains to every dynamic feature location technique.

Overview.

Section 2 defines what it means for a routine to be specific to a feature and delves into the approximative character of current dynamic analyses. Section 3 discusses related research and Section 4 describes our own hybrid technique in more detail. Section 5 describes case studies that address the above mentioned issues. Section 6 concludes.

2. ON FEATURE-SPECIFIC UNITS

In this section, we reflect on what it means for a unit to be specific to a feature. Let P be a program, $\mathcal{U}$ the set of units of P, $\mathcal{F}$ the set of features of P, and $\mathcal{S}$ the set of all possible usage scenarios for P. $\mathcal{U}$ is finite. Whether $\mathcal{F}$ is finite, too, depends upon our notion of feature. In non-trivial programs, we can often combine features in arbitrary ways. If we consider such combined features as features of their own, $\mathcal{F}$ is infinite. However, in the context of feature location, one often starts with a finite set of features to be located. $\mathcal{S}$, on the other hand, is infinite for every non-trivial program.

We say a scenario *exhibits* a feature if the feature may be observed when P is run under the scenario. Let $u \in \mathcal{U}$ and $f \in \mathcal{F}$. With this relation and a *use* relation between scenarios and units, we can define *specific* as follows:

$$\begin{aligned} u \text{ is } \textbf{specific} \text{ to } f \Leftrightarrow \\ \forall s \in \mathcal{S} : \text{exhibits}(s, f) \Leftrightarrow \text{uses}(s, u) \end{aligned} \tag{1}$$

In words, whenever the feature is executed, the unit is used and only then. Yet, there is still a hole in this definition: What do we mean by *use*? Dynamic analyses, so far, have answered this question by "the unit is executed for the scenario". However, this definition of *use* is inappropriate. Consider a compiler for C. When input is read during lexical analysis, it is matched against the strings in the symbol table of the lexer, and an appropriate token id is returned for every key word; otherwise the lexer returns an identifier id. The compiler does not know what kind of input can occur, so it must be prepared for all possible tokens. The implementation of the compiler, therefore, adds all key words of the language to this table during initialization.

Now, if we want to find out what is specific to handling `while` loops in this compiler, we would say that the fact that the string `while` is added to the symbol table is something specific to this kind of loop. However, the execution of adding the string is present in every scenario; but it is truly used only if the input program of the compiler contains a `while`.

A way to define *use* better than simply *execution* is the following: A scenario s for a feature f **uses** a unit u of program P if there is no program slice of P that has the same observable behavior as P with respect to s but that does not contain u. That is, one cannot remove u from P without loosing the behavior expected for f under s.

In our compiler example, we could remove the statements that add the `while` string to the symbol table if we consider only those test cases that do not have a `while` loop; but for all test cases that have such a loop, the statements must be present.

With this definition of *use*, we can categorize the remaining units u as follows with respect to a feature f:

$$\begin{aligned} &u \text{ is } \textbf{conditionally specific} \text{ for } f \Leftrightarrow \\ &\forall s \in \mathcal{S} : uses(s, u) \Rightarrow \text{exhibits}(s, f) \\ &\wedge \neg\forall s \in \mathcal{S} : \text{exhibits}(s, f) \Rightarrow \text{uses}(s, u) \end{aligned} \tag{2}$$

$$\begin{aligned} &u \text{ is } \textbf{relevant} \text{ for } f \Leftrightarrow \\ &\forall s \in \mathcal{S} : \text{exhibits}(s, f) \Rightarrow \text{uses}(s, u) \\ &\wedge \neg\forall s \in \mathcal{S} : uses(s, u) \Rightarrow \text{exhibits}(s, f) \end{aligned} \tag{3}$$

$$\begin{aligned} &u \text{ is } \textbf{irrelevant} \text{ for } f \Leftrightarrow \\ &\forall s \in \mathcal{S} : \text{exhibits}(s, f) \Rightarrow \neg uses(s, u) \end{aligned} \tag{4}$$

$$\text{Otherwise, } u \text{ is } \textbf{split} \text{ for } f. \tag{5}$$

As said earlier, dynamic analyses try to approximate *use* by *execution*. Because the two of them are not the same, some real uses may be missed and some units found by dynamic analyses will turn out to be no real uses. Another problem of dynamic analyses is the fact that $\mathcal{S}$ is infinite in general, and they can handle only a finite subset of $\mathcal{S}$.

3. RELATED RESEARCH

This section discusses related research. The dynamic techniques will be described in more detail as they are evaluated in this paper. We will present our hybrid technique in a section of its own and fairly detailed because it is the main target of our measurements.

3.1 Static Techniques

Chen and Rajlich [7] propose a method that allows a user to navigate on the static abstract system dependency graph. A tool supports the navigation. This technique works well if the analyst is familiar with the system. Otherwise the analyst does not know where to start and continue.

Marcus and Maletic [10] locate the points where to start by way of latent semantic indexing, an information retrieval technique that identifies correspondences between terms used in a requirement specification and identifiers in the code. A similar approach was earlier proposed by Antoniol et al. [1] to establish traceability links between requirement documents and source code.

Zhao et al. proposed a technique that combines Chen and Rajlich's idea with information retrieval techniques to guide the navigation [16].

3.2 Dynamic Techniques

Dynamic techniques gather the executed units for each scenario through profiling or another kind of instrumentation.

Wilde and Scully [13] categorize the executed units for a feature f as follows:

- units *commonly involved* (code executed in all test cases, regardless of f),
- units *potentially involved* in f (code executed in at least one test case that invokes f),
- units *indispensably involved* in f (code that is executed in all test cases that invoke f), and
- units *uniquely involved* in f (code executed exactly in cases where f is invoked)

Another approach based on dynamic information is taken by Wong and colleagues [14]. They analyze execution slices of test cases implementing a particular functionality. The process is as follows:

1. The *invoking input set* I (i.e., a set of test cases or – in our terminology – a set of scenarios) is identified that will invoke a feature.
2. The *excluding input set* E is identified that will not invoke a feature.
3. The program is executed twice using I and E separately.
4. By comparison of the two resulting execution slices, the units can be identified that implement the feature.

In both approaches, the idea is to build the difference between sets of executed units.

Zhao et al. propose a set of algorithms to compute simpler information we retrieve from the concept lattice [15]. Essentially, they implement the ideas of the two other dynamic techniques (simple set difference) not leveraging the advantage of concept analysis.

Rajlich and Wilde compared their complementary techniques in a case study [12].

4. HYBRID FEATURE LOCATION BASED ON CONCEPT ANALYSIS

Our own technique for feature location combines static and dynamic information [8]. The dynamic part is an extension of the techniques described in Section 3.2. Instead of simply looking at one feature at a time, we consider a whole set of features and their associated sets of test cases. The simple set differences used in the earlier approaches is replaced by formal concept analysis [2; 9]. Roughly speaking, one can think of formal concept analysis as a technique that repeatedly builds set differences among various test cases, units, and features in order to factor out what is specific to a feature.

A similar approach was taken by Bojic and Velasevic [6; 4; 5] to group classes into packages according to use cases. The difference between the two approaches is described elsewhere [3].

The static part of our technique resembles Chen and Rajlich's browsing on the static dependency graph, although the navigation is informed by the information we gather from the concept lattice produced by concept analysis. The exact details are described elsewhere [8]. In this paper, we focus on the dynamic part.

Next, we will describe the dynamic part in more detail by way of the example in Figure 1. Let us assume, we want to locate features in a compiler; more specifically, we want to find out how a compiler treats a certain set of language constructs, for instance, the ASSIGN operator, IF and WHILE statements and the binary MINUS operator. In general, let $\mathcal{F}$ be the set of features we are interested in.

For these features, we can design a set of test cases (also known as scenarios), such as those in Figure 1. Furthermore, we provide a mapping of features onto these test cases as illustrated in the left part of Figure 2. This mapping is relation *exercise* $\subseteq \mathcal{S} \times \mathcal{F}$ described in Section 2.

```
test1:   a := 10; while a do a := a - 1;
test2:   a := 0; if a then a := 1;
test3:   a := 0;
test4:   a := a - 1;
empty:   empty program
```

Figure 1: Set of test cases S

Next we run the system several times, each time using a different test case. Through profiling we gather the executed units for each test case, which results in a table. Each row in that table describes which units have been executed for each test case. We call this table the invocation table *Exe* (it is the dynamic approximation of relation *use* of Section 2). *Exe* is a binary relation $Exe \subseteq \mathcal{S} \times \mathcal{U}$. The invocation table *Exe* for our running example is shown in the right part of Figure 2. The units in this example are routines of the recursive-descent parser of our compiler.

Scenarios	Features $\mathcal{S} \times \mathcal{F}$				Units $\mathcal{S} \times \mathcal{U}$					
	ASSIGN	WHILE	MINUS	IF	parse	parse_expr	parse_asgn	parse_binop	parse_wl	parse_if
test1	×	×	×		×	×	×	×	×	
test2	×			×	×	×	×			×
test3	×				×	×	×			
test4	×		×		×	×	×	×		
empty					×					

Figure 2: Feature-scenario mapping and invocation table

Feature-scenario mapping and invocation table together allow us to spot those units that are specific to a feature. Where would we look for units related to feature MINUS? We would investigate all test cases that use MINUS and those units these test cases have in common. In Figure 2, these would be the routines `parse`, `parse_expr`, and `parse_asgn`. However, some of them might contribute to other features as well; `parse`, for instance, is called for all test cases no matter whether ASSIGN is actually used. How can we find out what is going on in this table? The answer is *formal concept analysis*.

In order to explain concept analysis (here immediately adjusted to our application), we need some definitions first. We apply concept analysis by defining a set of objects $\mathcal{O} = \mathcal{U} \cup \mathcal{F}$, a set of attributes $\mathcal{A} = \mathcal{S}$, and a relation $\mathcal{I} = \mathit{exercises} \cup \mathit{Exe}$ in between that describes which objects have which attributes.

For a set of objects $O \subseteq \mathcal{O}$, $ca(O)$ is the set of **common attributes**:

$$ca(O) := \{a \in \mathcal{A} \quad | \quad \forall o \in O : (o, a) \in \mathcal{I}\}$$

For a set of attributes $A \subseteq \mathcal{A}$, $co(A)$ is the set of **common objects**:

$$co(A) := \{o \in \mathcal{O} \quad | \quad \forall a \in A : (o, a) \in \mathcal{I}\}$$

A pair (O, A) is a **formal concept** if $A = ca(O) \wedge O = co(A)$. Intuitively, a concept is a maximally large filled rectangle of the table (with row and column permutations). Let $c = (O, A)$ be a concept, then $O = \mathit{extent}(c)$ and $A = \mathit{intent}(c)$.

Between concepts, we can define a **partial order** $\leq$ as follows: let $c_1 = (O_1, A_1)$ and $c_2 = (O_2, A_2)$ be two concepts of the same formal context, then

$$c_1 \leq c_2 \Leftrightarrow O_1 \subseteq O_2 \text{ or dually, } A_2 \subseteq A_1$$

If $c_1 \leq c_2$ holds, c_2 is the **superconcept** of c_1 because it has all attributes of c_1; c_1 is the **subconcept** of c_2. The concepts and the partial order define a lattice, the **concept lattice** $\mathcal{L}$. The concept lattice for the running example is shown in Figure 3(a).

The **infimum** ($\wedge$) of two concepts in this lattice is computed by intersecting their extents as follows:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, ca(O_1 \cap O_2)) \tag{6}$$

The infimum describes a set of common attributes of two sets of objects. Similarly, the **supremum** ($\vee$) is determined by intersecting the intents:

$$(O_1, A_1) \vee (O_2, A_2) = (co(A_1 \cap A_2), A_1 \cap A_2) \tag{7}$$

The supremum yields the set of common objects, which share all attributes in the intersection of two sets of attributes.

The concept lattice can be visualized in a more readable equivalent way by marking only that graph node with an attribute $a \in \mathcal{A}$ whose represented concept is the most general concept that has a in its intent. Analogously, a node will be marked with an object $o \in \mathcal{O}$ if it represents the most special concept that has o in its extent. The unique elements in the concept lattice marked with a and o, respectively, are therefore:

$$\mu(a) = \bigvee \{c \in \mathcal{L}(C) \mid a \in \mathit{intent}(c)\} \tag{8}$$

$$\gamma(o) = \bigwedge \{c \in \mathcal{L}(C) \mid o \in \mathit{extent}(c)\} \tag{9}$$

There may be concepts in the lattice, say c, for which there is no object, o, with $\gamma(o) = c$ and no attribute, a, with $\mu(a) = c$. We call such concepts **empty concepts** (they are empty in the sparse lattice).

Finally, as shown by Simon [11], the classification of units introduced in Section 2 can be recast in terms of concept analysis (illustrated for feature MINUS in Figure 3(b)) as follows:

$$u \text{ is specific (cf. (1)) for } f \Leftrightarrow \gamma(f) = \gamma(u) \tag{10}$$

$$u \text{ is conditionally specific (cf. (4)) for } f \Leftrightarrow \gamma(f) < \gamma(u) \tag{11}$$

$$u \text{ is relevant (cf. (2)) for } f \Leftrightarrow \gamma(u) < \gamma(f) \tag{12}$$

$$u \text{ is irrelevant (cf. (3)) for } f \Leftrightarrow \gamma(f) \not< \gamma(u) \wedge \gamma(u) \not< \gamma(f) \tag{13}$$

$$\text{Otherwise, the unit is split.} \tag{14}$$

The test cases must be designed so that every feature of interest, f, can be isolated from other features; that is, $\neg\exists f' \neq f : \gamma(f') = \gamma(f)$.

An empty scenario, called **bias**, is used to subtract all units that are executed when the system is started and terminated without executing any other feature.

Finally, the information shown by the concept lattice gives you hints where to start your static search for feature-specific units and where to continue. It ranks every unit with respect to its specificity for any given subset of features. For the static search, you can slice the program at routine or statement level. The concept lattices gives you the slicing criteria.

5. VALIDATION OF EARLIER HYPOTHESES

In an earlier case study [8], we successfully applied the technique to a large industrial system at the routine level. The system consisted of about 10,000 statically declared routines (of which about 1,500 were actually executed by our test suite).

This section first reports on beliefs, scepticism, and hopes we had after the initial experience and then on new experiments to validate these hypotheses. Unfortunately, we could not use the same industrial system again. It was no longer available to us.

5.1 Earlier Hypotheses

Hypothesis 1

We chose the routine level for profiling in the earlier experiment because we were sceptical whether a more detailed analysis at the statement level would be feasible. Yet, we may expect statement level profiling particularly effective for routines with large switch statements or `if-then-else`

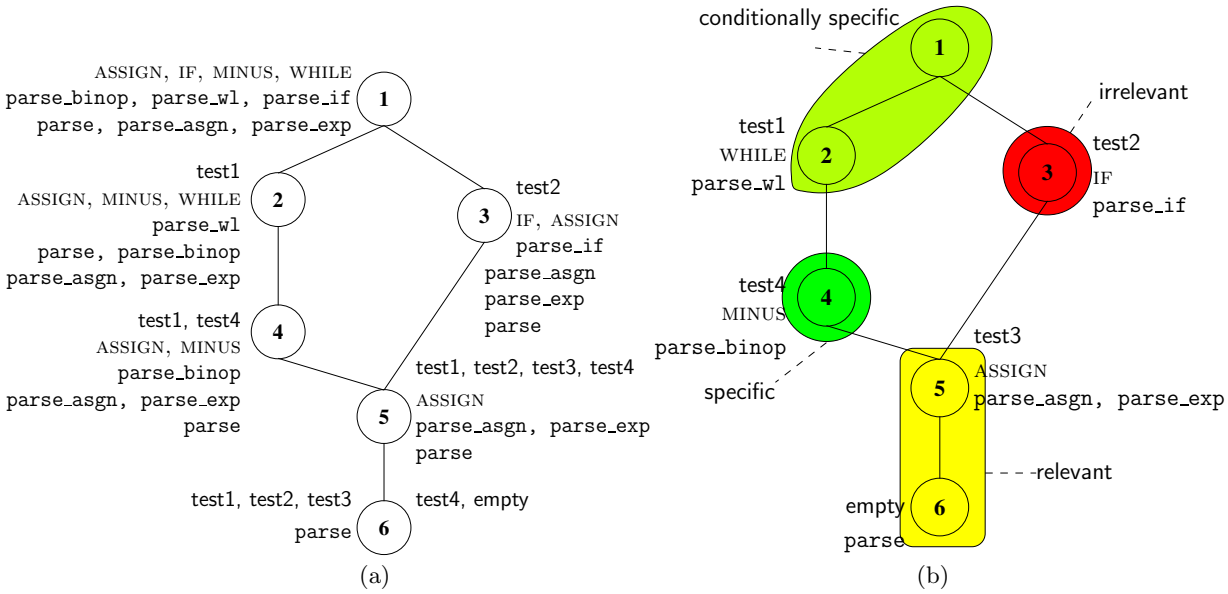


Figure 3: Full (a) and sparse (b) concept lattices for the context from Fig. 2.

cascades of which only one branch is executed for a particular feature. Such routines as a whole would be executed for many features and, hence, appear very low in the lattice, whereas their branches would be closer to the top element and more specific in the lattice. Furthermore, a subsequent static analysis to validate the findings of dynamic feature location can work at a more detailed level. For instance, the dynamic feature location may yield slicing criteria for static program slicing that works at statement level. Recall that we recommend to complement any dynamic analysis with a static validation (cf. Section 2). We formulate our first hypothesis:

> **Hypothesis 1**: Statement level profiling yields more detailed results at acceptable costs.

We test hypothesis 1 by profiling at both the routine level and statement level (basic blocks, more precisely) and measure the gain of information as the refinement of the lattice. We measure cost as the number of concepts and the runtime for concept analysis.

Every routine r consists of a set of basic blocks (sequences of statements with single entry and single exit) $\{b_1, \ldots, b_n\}$; moreover, whenever one basic block of r is executed, r has also been executed, hence: $\forall s \forall 1 \leq i \leq n : (b_i, s) \in \mathcal{I} \Rightarrow (r, s) \in \mathcal{I}$. As a consequence, $\gamma(r) \leq \gamma(b)$ for all basic blocks b of r. If $\gamma(r) = \gamma(b)$, there is no gain of information because the basic block is executed whenever r is executed. If $\gamma(r) < \gamma(b)$, we find a basic block that is executed only under more specific circumstances. The set of features to which a unit u – be it a basic block or a routine – contributes is $FS(u) = \{f | f \in \text{extent}(\gamma(u))\}$. We call this set the **feature set** of u. If $\gamma(r) < \gamma(b)$, the feature set of b may be a subset of the feature set of r; that is, b is more specific than r because it is higher in the lattice.

The cardinality function $|FS|$ can be presented for the basic blocks of a single routine as sketched in Figure 4(b), where the basic blocks appear in ascending order of the cardinality of their feature set. The upper line shows the number of features for the routine that contains them when the analysis is applied at routine level. The grey area can be viewed as the gain of information from routine to basic block level, which can be measured as the relative number of basic blocks one has to look at. More precisely, let $B(r)$ be the set of basic blocks of routine r (when profiling at routine level, one has to look at every basic block; at basic block level only at those reported):

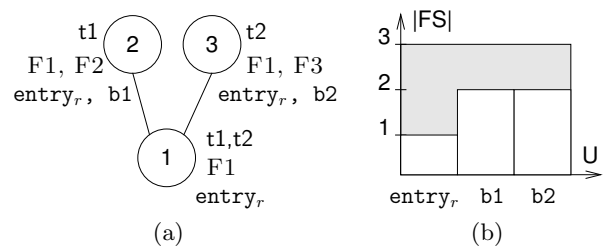


Figure 4: (a) Lattice excerpt, (b) function $|FS|$.

$$G(r) = 1 - \frac{\sum_{b \in B(r)} |FS(b)|}{|B(r)| \times |FS(r)|} \tag{15}$$

There are two notes to make here. First, because we take into account only those basic blocks that are executed at least once after subtracting the bias, $|FS(b)| > 0$ holds for every basic block. Second, one might wonder why $|FS(r)|$ can be greater than $|FS(entry_r)|$ with $entry_r$ being the entry basic block of r in Figure 4(b). Note that we establish FS for r and $entry_r$ through tracing at different levels. Consider Figure 4(a) as an example. Here, $FS(entry_r) = \{F1\}$ when we trace at basic block level. If we trace at routine level, however, the basic blocks will be missing in the lattice and the concepts 1, 2, and 3 in Figure 4(b) collapse into one concept, so that $FS(r) = \{F1, F2, F3\}$.

Hypothesis 2

Ideally, we have positive test cases that exhibit all variations of the features of interests and negative test cases that are useful to subtract anything that is not feature relevant. For economic reasons, the test suites should also be reasonably small.

We found all specific routines in the earlier experiment with only 93 test cases for 76 relevant features. In most cases, there was a one-to-one correspondence between test cases and features. A few test cases exercised two features, in which case we added another test case that exercised only one of them in order to separate the routines specific to these two features in the lattice. However, we did not combine features further because we assumed that the combination would not affect the results.

Hypothesis 2: The combination of features does not yield additionally executed units.

We were able to design a small test suite that actually sufficed to locate all specific features without any prior knowledge of the implementation in the earlier experiment. We hoped that this might work for other systems, too. However, if the combination of features adds information, then the initial non-combined test suite is too simplistic.

We test hypothesis 2 by starting with an initial small set of test cases of which we think exhibit the set of features sufficiently; that is, they are sufficient to isolate the features in the lattice as described in Section 4. If we have n completely independent features, we need minimally $n + 1$ test cases (one more for the bias).

Then we extend the initial set through seemingly redundant combinations of the features. One should think that the lattice remains the same because we are not adding features. We can then measure the changes in the lattice to see whether the sum is more than its parts.

Adding test cases means to move from a subcontext to a supercontext (see also [9]). The lattice for the supercontext is a refinement of the subcontext; that means, every concept in the subcontext may be mapped onto one concept of the supercontext [9]. Hence, we can measure the effect of additional test cases as the number of added non-empty concepts.

Hypothesis 3

Computing the concept lattice in the earlier experiment took just a few minutes on an ordinary PC of these days although the number of concepts may grow exponentially with the number of objects and attributes in the worst case. We were hoping that the approach scales well for larger sets of features in practice.

Hypothesis 3: The approach scales for large feature sets.

We test hypothesis 3 by taking into account a very large number of feature combinations.

5.2 General Test Case Setup

Investigated compilers.

For our new experiment, we use two fairly large C compilers that are both available as open source: *sdcc* (Small Device C Compiler[1]), version 2.4.0, and the C compiler, *cc1*, of GCC[2], version 3.4.3. The number of routines and basic blocks the compilers consist of is shown in Fig. 5.

	sdcc	cc1
#routines	1,325	15,986
#basic blocks	46,699	379,086
#basic blocks in bias	1,095	6,965
#routines executed (expr+loops)	484	1,349
#basic blocks exec'd (expr+loops)	5,970	15,615
#routines executed (switch)	650	2,657
#basic blocks executed (switch)	10,113	34,602

Figure 5: Routine and basic block counts

We selected compilers because they are complex software systems with many features which can be combined in many ways. We looked at variations in program language constructs (loops and expressions) and compiler switches.

Instrumentation.

We used an own prototype for code instrumentation that inserts counting statements at the head of each basic block. When an instrumented program terminates, it outputs the execution counter for each executed basic block.

Test-Case Design.

We concentrated on a certain set of features at one time. For these features, we created a minimal set of test cases which should isolate the desired features. That means, when running concept analysis on the features of the test cases, each feature would have a specific node.

Then we created additional test cases that use the same features in different combinations. We created test cases for combination of features in different arrangements (sequentially, nested, same/different features etc.). Then we ran concept analysis on the results of these test cases combined with the minimal set to find out whether these redundant test cases provide additional information.

The output of running a test case was the list of basic blocks that was executed by the compiler during that run, along with the number of executions for each basic block. Since we were not interested in all the basic blocks that are executed with every run of the compiler (initialization etc.), we did one run with an empty program and took this as the bias. This bias was then subtracted from each test case result, and only basic blocks that were still executed at least once were considered for concept analysis. The size of the bias and the number of executed routines and basic blocks for the test cases can also be seen in Fig. 5.

5.3 Loops

To locate the spots in the compilers' code that have to do with loops, we created several test cases that isolate the features `while`, `do-while`, `for`, and `if-goto` ('minimal' test cases) similarly to Fig. 1.

Hypothesis 1

Figure 6 shows the percentage of routines that was split into more than one concept when refining from routine level

[1] `http://sdcc.sourceforge.net/`
[2] `http://gcc.gnu.org/`

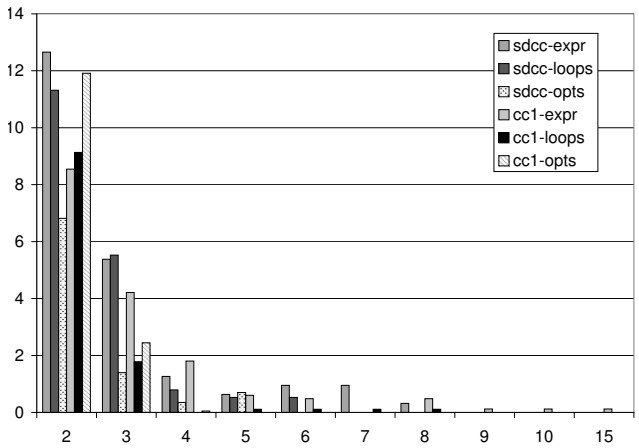


Figure 6: Additional information gained by refining from routine level to basic block level; X = number of concepts that a routine was split into when refining to basic blocks; Y = percentage of routines that split into this number of concepts

to basic block level. Up to 22% of the routines were distributed over more than one concept on basic block level. So we looked into the details of these routines: Do we get any additional information from that split? This would be the case if features that were specific to a routine's node distribute also over several of the new basic block nodes. We measured this distribution through the information gain as defined by equation (15). In these test cases, we could reduce the number of basic blocks that have to be considered for each of the interesting features by 76.4% for *sdcc* and by 83.2% for *cc1*.

Further investigation revealed that this improvement was mainly caused by routines that were using switch statements with many cases or large if-then-else cascades (e.g., `yyparse`, `expand_stmt` in these test cases). Feature location on basic block level helps to directly find the relevant code snippets inside such constructs.

Hypothesis 2

Then we combined these basic constructs to build more complex functions. We compiled those test cases with the instrumented compilers and applied concept analysis to different combinations of the resulting information both on routine and on basic block level. The additional combining test cases are shown in Figure 7. The results are displayed in Figure 8.

	`do-while` + `for` nested	`do-while` + `for` sequential	`if-goto` + `while` nested	all except `for`
L1				
L2	×			
L3		×		
L4	×	×		
L5			×	
L6	×		×	
L7	×		×	×

Figure 7: Combinations of test cases for loops

In case of the nested `do-while` and `for` (L2), and also in the `while/if-goto` test case (L5), on routine level there were no additional routines executed. The concept lattice stays the same, with just the additional test case node inserted. This is largely true for both *sdcc* and *cc1*.

For the same test cases on basic block level, additional blocks were executed and more new concepts were created. So the combination of basic test cases did not produce additional information on routine level, but on basic block level, it did.

5.4 Expressions

A second set of test cases dealt with expressions. The basic test set contains 6 test cases that isolate the 4 basic integer operations $+$, $-$, $*$ and / and integer literals ('initial' test set).

Hypothesis 1

In the expression test cases, the improvements were spread over many more routines than in the loop test cases. Routines like `decorateType`, `algebraicOpts` in *sdcc* and `recog`, `find_reloads`, and `expand_expr_real` in *cc1* showed great improvements. Overall, the number of basic blocks that have to be considered for each of the interesting features could be reduced by 81.6% for *sdcc* and by 78.8% for *cc1*.

As an example, Figure 9 shows the details of this reduction graphically. The cardinality of feature sets FS (see Section 5.1) across all routines and their basic blocks is displayed in a more compact fashion than Figure 4(b). The X axis is the percentage of basic blocks, and the Y axis is the number of features they contribute to. Each diagram shows two curves: The light gray one is the curve on routine level, while the dark gray one shows the corresponding curve on basic block level. The light gray area can be regarded as the amount of information we gained by going from routine level to basic block level.

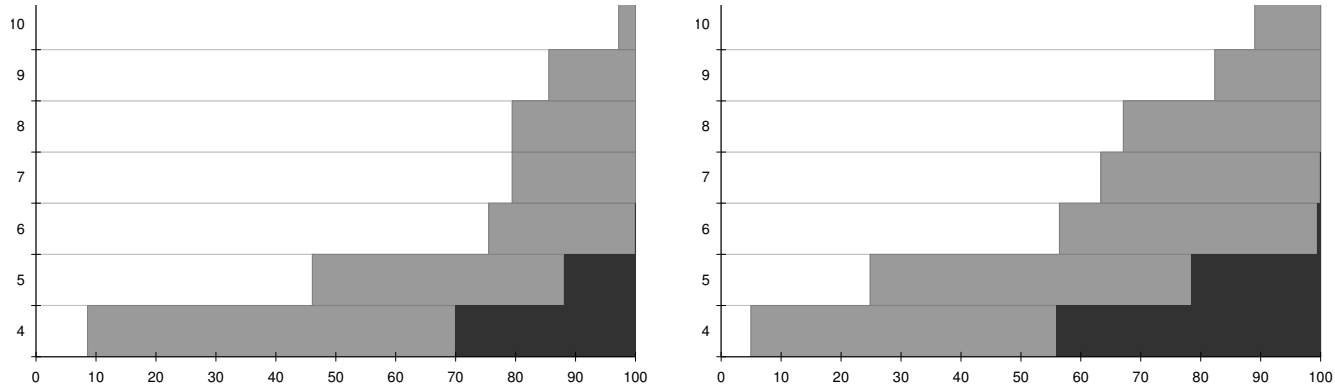
Hypothesis 2

Additional test cases were combined with the initial test set: Addition of a constant, addition of 0 and 1, use of all four operations in one expression (*all-in-one*), *all-in-one* with additional subexpressions of this expression and also with new combinations of operations, *all-in-one* with a more complex expression that contains some of the operations multiple times, and *all-in-one* with multiple assignments.

The results of running concept analysis on these test case combinations is shown in Figure 10. Apparently, the compilers are doing more work for combined operations than just processing every single operation. More routines and basic blocks are executed, and there are more concepts in the lattice. Investigation showed that this increase has mostly to do with register allocation. Another expected observation is that it also matters which operations are combined. Adding a constant other than zero needs more resources than adding a zero – this also results in more concepts. Combining $-$ with / is more complex than combining $+$ with $*$. On the other hand, it does not matter much if we have multiple expressions of similar kind or just one larger expression. Basically, it seems that once a certain number of combinations has been covered and a certain degree of complexity has been reached, we reach some stability. This means that our initial assumption – that we had a minimal set of isolated basic features – was not true; but in fact the set was only

	sdcc								*cc1*							
	Routine level				Basic block level				Routine level				Basic block level			
	C	NE	AE	RT	C	NE	AE (R)	RT	C	NE	AE	RT	C	NE	AE (R)	RT
L1	15	12		10	30	20		654	17	15		35	33	26		2,039
L2	16	13	0	10	60	27	9 (4)	798	18	16	1	40	48	30	3 (2)	2,065
L3	18	14	5	12	116	32	158 (31)	947	24	20	3	48	71	36	29 (7)	2,248
L4	18	14	5	12	147	37	158 (31)	1,034	24	20	3	53	78	37	30 (8)	2,494
L5	16	13	0	10	39	25	27 (10)	778	20	17	0	41	54	33	0 (0)	2,096
L6	17	14	0	11	72	32	30 (11)	943	21	18	1	46	75	37	3 (2)	2,421
L7	19	15	5	12	135	38	158 (31)	1,069	25	21	3	51	92	40	30 (6)	2,875

Figure 8: Results of concept analysis for loops. C is the number of concepts and NE the number of non-empty concepts; AE is the number of additionally executed routines or basic blocks, compared to the initial test suite; for the basic blocks, the value in brackets is the number of routines that the additional basic blocks are located in; RT is the runtime, given in milliseconds on a Pentium IV with 3 GHz and 1 GB RAM.



(a) *sdcc*: reduction by 81.6% (b) *cc1*: reduction by 78.8%

Figure 9: Information gain for expressions. X = basic blocks (sorted by number of features they contribute to), Y = number of features. The curves represent the number of basic blocks that have to be considered when looking for a feature – light gray for routine level, dark gray for basic block level.

a subset. We would also have to consider features such as aggregation of operations to get a sufficient test set.

5.5 Command Line Switches

In a third test series, we investigated different command line switches for code optimization on the same input file. The input file was a variation of the set implementation from Lindig's tool *concepts*[3]. The basic test set consisted of compilation runs with each optimization option switched on separately (depending on the available compiler switches).

Different combinations of compiler switches were combined with the initial test set. Also, one other compiler switch for generating debug information was used.

Hypothesis 1

Looking at the information gain, for this test case group the results were not as promising as for the other groups at first sight. The number of concepts did hardly increase when going from routine level to basic block level. Anyway, when looking at the details, we do in fact have an improvement, since basic blocks are moving upwards in the concept lattice and are assigned to more specific features. The search space reduction reached 88.5% for *sdcc* and 70.0% for *cc1* and was therefore comparable to the other test cases.

[3] `http://www.st.cs.uni-sb.de/~lindig/`

Hypothesis 2

The results of concept analysis on basic block level were a bit more detailed than on routine level (see Figure 11). In this test case, combining certain switches did not generate any additional information, while combining other switches did. Apparently, some of the optimizations have interdependencies with each other, while others do not. This can be observed for both *sdcc* and *cc1*.

5.6 Hypothesis 3

We also tried to apply our analysis to the set of all language constructs of the C language sufficient to isolate the constructs in the test suite. We created more than 100 test cases for about the same number of features.

For *sdcc*, the relation of this formal context consists of 117 rows and 678 columns with 32,314 fields set, resulting in a lattice with 80,279 concepts when calculated on routine level. On basic block level, the relation increases to 8,953 columns with 324,173 fields set. When additionally regarding multiple backends and two other compiler switches, the number of concepts explodes to 4.5 millions on routine level.

In case of *cc1*, the relation of the context for the 100 test cases with one backend and no additional compiler switches contained about 1.3 mio elements, and the lattice could not be calculated even on routine level due to limitation of space resources.

	sdcc								*cc1*							
	Routine level				Basic block level				Routine level				Basic block level			
	C	NE	AE	RT	C	NE	AE (R)	RT	C	NE	AE	RT	C	NE	AE (R)	RT
initial $+ - */$	17	16		7	19	19		353	21	20		28	31	27		1,420
$+C$	19	18	6	7	35	27	88 (16)	415	38	29	6	32	52	43	159 (28)	1,410
$+0$	26	21	4	7	35	31	45 (18)	416	32	25	1	32	44	38	14 (6)	1,387
$+1$	24	19	6	8	35	27	92 (16)	431	38	29	4	33	60	42	107 (26)	1,573
$+C$, $+0$, $+1$	42	26	6	10	80	39	95 (17)	517	56	34	7	41	85	58	172 (34)	1,842
all-in-one (aio)	22	21	19	8	25	25	394 (48)	474	34	28	19	33	54	40	628 (80)	1,708
aio, $+*$	23	22	19	9	28	28	394 (48)	510	46	31	19	38	81	49	628 (80)	1,792
aio, $-/$	24	22	19	9	31	30	395 (48)	536	45	33	22	38	75	52	817 (88)	1,924
aio, $++$	26	22	19	9	34	31	405 (48)	517	35	29	20	37	60	43	642 (83)	1,853
aio, $**$	24	23	24	9	30	28	501 (55)	525	34	28	19	37	63	43	628 (80)	1,904
aio, $+*$, $-/$	26	24	19	10	35	33	395 (48)	618	59	35	22	46	115	60	817 (88)	2,040
aio, complex	23	22	19	9	29	27	425 (53)	525	35	29	21	37	71	45	738 (86)	2,184
aio, two lines	23	22	24	9	29	28	503 (55)	528	35	29	20	38	71	45	663 (82)	2,097

Figure 10: Results of concept analysis for expressions

	sdcc								*cc1*							
	Routine level				Basic block level				Routine level				Basic block level			
	C	NE	AE	RT	C	NE	AE (R)	RT	C	NE	AE	RT	C	NE	AE (R)	RT
minimal	22	12		10	22	12		2,660	9	7		70	9	7		17,200
two on	26	15	0	16	26	15	0 (0)	3,180	12	10	152	110	14	11	1,399 (201)	22,600
other two on	30	16	0	20	34	17	2 (1)	3,190	10	8	0	90	10	8	0 (0)	19,420
all four on	29	18	2	20	36	21	36 (9)	3,130	14	12	222	110	16	13	2,639 (290)	23,380
debug	23	13	14	20	39	14	186 (23)	3,080	11	9	192	100	15	11	1,300 (212)	19,530

Figure 11: Results of concept analysis for optimization options

An interesting observation during these tests was that the number of concepts grew exponentially when adding test cases with additional features, while it grew only linearly when adding redundant test cases.

The overall result of this attempt was that looking at all the features of a program at once is a complex approach which needs a lot of resources and generates complex results. It was hardly possible to calculate the concepts on routine level, and it is currently impossible on basic block level for complex programs.

5.7 Summary

Figure 12 shows the search space reduction for all test cases and the share of routines that this reduction related to, and Fig. 13(a) summarizes the findings for hypothesis 1. In all three cases, the hypothesis was confirmed. So, there is a strong indication that in many cases profiling at statement level does yield more precise results.

The computational costs were mostly acceptable. For every executed routine, there were between 11 and 20 executed basic blocks. The runtime increased by a factor between 50 and 200. That indicates a superlinear growth in the number of routines. Yet, in absolute time, the costs can still be neglected: the longest run took only 23 seconds for *cc1* (consisting of about 400 KLOC) on an ordinary PC.

	sdcc			*cc1*		
	expr	loops	opts	expr	loops	opts
Split	22.2%	18.7%	8.6%	16.5%	11.4%	14.4%
Red.	81.6%	76.4%	88.5%	78.8%	83.2%	70.0%

Figure 12: Share of routines that split into multiple concepts, and search space reduction for those.

Figure 13(b) summarizes the findings for hypothesis 2. Hypothesis 2 is largely falsified. In practice, hence, one should also combine seemingly independent features to capture feature interactions.

Hypothesis 3, finally, was falsified, too. Hence, the number of features one can investigate with concept-based feature location has its limit. This technique works well for a smaller set of features. For many features, in particular, manifold combinations of features, the combinatorial explosion of required test cases and the exponential growth in the number of concepts imposes barriers.

6. CONCLUSIONS

In an earlier case study [8], we successfully applied the technique to a large industrial system (10,000 statically declared routines of which about 1,500 were actually executed and 93 test cases). This time we applied it to two compilers, one of them with a comparable number of routines, and failed with 100 test cases. In the compiler case studies, there was more overlap of the test cases, and the earlier case study did not take features as objects into account.

Dynamic feature location based on concept analysis does allow one to locate the units for a set of features, but the number of features and test cases one can handle has a limit. If many features are to be investigated, the combinatorial explosion of feature compositions does not allow full coverage of features for both economic reasons (creation and execution of test cases) and the computational demand for concept analysis. In such cases, one should probably revert to static analyses.

loops	expressions	options
√	√	√

(a) Hypothesis 1

loops		expressions		options	
routine	statement	routine	statement	routine	statement
√	—	—	—	∼	∼

(b) Hypothesis 2

Figure 13: Summaries for (a) hypothesis 1 and (b) hypothesis 2: √ denotes confirmed, ∼ denotes inconclusive, and — denotes falsified.

If one has a smaller set of features, one can apply concept analysis even at basic block level; the larger system has about 380,000 basic blocks of which 13,000 were actually executed. The gain of information lay in between 70 and 89% at sustainable costs (mostly less than 3 seconds) in cases where different features are implemented by the same routines. If features are implemented by distinct sets of routines, lowering to basic blocks is less effectual.

Lowering the level was particularly effective for routines with switches and if-then-else cascades. One could combine profiling at routine and basic block level by instrumenting such functions in particular. It remains a question for future research whether a similar gain can be achieved for object-oriented systems which replace switches by redefinition of methods and dispatching.

In many cases, the combination of features in test cases does add executed units. This effect hits every dynamic feature location, not just those based on concept analysis. For this reason, one should combine even seemingly independent features. Furthermore, one should be very conscious about the limits of dynamic analyses and combine the dynamic techniques with static ones. The advantage of concept-analysis based feature location over simpler dynamic analyses is that the concept lattice will factor out the differences between atomic and composed feature invocations.

Website

The material that was used and created for this paper can be downloaded from our website:
`http://www.informatik.uni-bremen.de/st/dfl/`

References

[1] Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., and Merlo, E. Recovering traceability links between code and documentation. *IEEE TSE 28*, 10 (Oct. 2002), 970–983.

[2] Birkhoff, G. *Lattice Theory*, first ed. American Mathematical Society Colloquium Publications 25, Providence, RI, USA, 1940.

[3] Bojic, D., Eisenbarth, T., Koschke, R., Simon, D., and Velasevic, D. Addendum to locating features in source code (short paper). *IEEE TSE 30*, 2 (Mar. 2004), 140.

[4] Bojic, D., and Velasevic, D. A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Engineering. In *Proc. of the 4th CSMR* (Feb. 2000), IEEE Press, pp. 23–32.

[5] Bojic, D., and Velasevic, D. URCA Approach to Scenario-Based Round-trip Engineering. In *Proc. of the OOPSLA 2000 Workshop on Scenario-based Round-Trip Engineering* (Minneapolis, MN, USA, Oct. 2000), Unpublished., pp. 51–56.

[6] Bojic, D., and Velasevic, D. A Method for Reverse Engineering of Use-Case Realizations in UML. *The Australian Journal of Information Systems 8*, 2 (May 2001).

[7] Chen, K., and Rajlich, V. Case study of feature location using dependence graph. In *IWPC* (2000), IEEE Press.

[8] Eisenbarth, T., Koschke, R., and Simon, D. Locating features in source code. *IEEE TSE 29*, 3 (2003).

[9] Ganter, B., and Wille, R. *Formal Concept Analysis–Mathematical Foundations*. Springer, 1996.

[10] Marcus, A., and Maletic, J. I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE* (May 2003), IEEE Press, pp. 125–134.

[11] Simon, D. *Lokalisierung von Merkmalen in Softwaresystemen*. Ph.d. dissertation, University of Stuttgart, Germany, 2005. To appear.

[12] Wilde, N., Buckellew, M., Page, H., and Rajlich, V. A Case Study of Feature Location in Unstructured Legacy Fortran Code. In *Proc. of the 5th CSMR* (Lisbon, Portugal, Mar. 2001), IEEE Press, pp. 68–75.

[13] Wilde, N., and Scully, M. Software reconnaissance: Mapping from features to code. *Journal on Software Maintenance and Evolution 7* (Jan. 1995), 49–62.

[14] Wong, W. E., Gokhale, S. S., Horgan, J. R., and Trivedi, K. S. Locating Program Features using Execution Slices. In *Proc. of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology* (Richardson, TX, USA, Mar. 1999), IEEE Press, pp. 194–203.

[15] Zhao, W., Zhang, L., Hao, D., Mei, H., and Sun, J. Alternative scalable algorithms for lattice-based feature location. In *ICSM* (2004), IEEE Press, p. 528.

[16] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F. Sniafl: Towards a static non-interactive approach to feature location. In *ICSE* (2004), IEEE Press, pp. 293–303.