

# Feature-Driven Program Understanding Using Concept Analysis of Execution Traces

Thomas Eisenbarth, Rainer Koschke, Daniel Simon

University of Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Germany

{eisenbts, koschke, simon}@informatik.uni-stuttgart.de

## Abstract

*The first task of a programmer who wants to understand how a certain feature is implemented is to localize the implementation of the feature in the code. If the implementations of a set of related features are to be understood, a programmer is interested in their commonalities and variabilities. For large and badly documented programs, localizing features in code and identifying commonalities and variabilities of components and features can be difficult and time-consuming. It is useful to derive this information automatically.*

*The feature-component correspondence describes which components are needed to implement a set of features and what are the respective commonalities and variabilities of features and components. This paper describes a new technique to derive the feature-component correspondence utilizing dynamic information and concept analysis. The method is simple to apply, cost-effective, largely language-independent, and can yield results quickly.*

## 1. Introduction

Let us assume you are programmer who has just recently been assigned to maintain a legacy system, say, a graphical editor for drawing simple pictures consisting of lines, boxes, ellipses, etc. Your manager wants you to add a new feature that allows a user to draw segments of ellipses. The editor currently lets a user draw whole ellipses only, but in different ways: by specifying radius or diameter. The market analyst has identified a need for the same alternative ways to specify segments of ellipses. Unfortunately, you are unfamiliar with the implementation of the system. The former maintainer was fired because he did not document the system at all. Hence, the only reliable source of information is the code itself.

Very likely, you are at first interested in *where the similar features are implemented*, i.e., which components contribute to the existing similar features (the components for drawing ellipses in our example). The primary components you really want to look at are those that are specific to these features. In other words, in the beginning, you are less interested in general-purpose components that con-

tribute to all kinds of features, like components to draw a single point. Then you want to *understand the commonalities and variabilities of the components that specifically contribute to the set of similar features* in order to get design ideas for your own implementation and to investigate ways to integrate your solution. The system consists of hundreds of files. Where do you start?

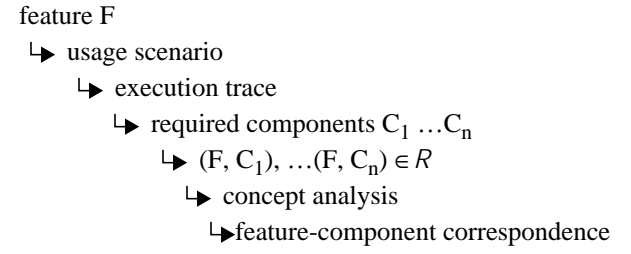
One important piece of information for a maintainer who needs to understand how a set of features is implemented is the so-called **feature-component correspondence**. The feature-component correspondence is a documentation artifact that describes which components are needed to implement a particular feature or set of features. In case of related features, it also describes the jointly and separately used components for those features. For components, it identifies the features to which these components jointly or separately contribute. A **feature** is a realized (functional as well as non-functional) requirement (the term *feature* is intentionally weakly defined because its exact meaning depends on the specific context). **Components** are computational units of a software architecture (see Section 3.1). The simplest example of a component is a **subprogram**, i.e., a function or a procedure.

Generally, to change a program, only partial knowledge is required. Consequently, a complete and hence time-consuming reverse engineering of the system is very likely not appropriate. Instead, the analysis should focus on the given problem. The feature-component correspondence can be used to identify the primary pieces of the software that need to be looked at, gives insights into relations between related features and components, and thus allows to aim further analyses at selected components cost-effectively.

This paper describes a quickly realizable technique to ascertain the feature-component correspondence based on dynamic information (gained from execution traces) and concept analysis. Concept analysis is a mathematical technique to investigate binary relations (see Section 2). The technique is automatic to a great extent.

**Overview.** The technique described here is based on the execution traces generated by a profiler for different usage scenarios (see Figure 1). One scenario represents the invocation of one single feature or a set of features and yields

all subprograms executed for these features. These subprograms identify the components (or are themselves considered components) required for certain features. The required components for all scenarios and the set of features are then subject to concept analysis. Concept analysis gives information on relationships between features and required components.



**Figure 1. Overview.**

We want to point out that not all non-functional requirements, e.g., time constraints, can be easily mapped to components; i.e., our technique primarily aims at functional features. However, in some cases, it is possible to isolate non-functional aspects, like security, in code and map them to specific components. For instance, one could concentrate all network accesses in one single component to enable controlled secure connections.

Moreover, the technique is not suited for features that are only internally visible, like whether a compiler uses a certain intermediate representation. Internal features can only be detected by looking at the source, because it is not clear how to invoke them from outside and how to derive from an execution trace whether these features are present or not.

The remainder of this article is organized as follows. Section 2 introduces concept analysis. Section 3 explains how concept analysis can be used to derive the feature-component correspondence and Section 4 describes our experience with this technique in a case study. Section 5 discusses related research.

## 2. Concept Analysis

Concept analysis is a mathematical technique that provides insights into binary relations. The mathematical foundation of concept analysis was laid by Birkhoff in 1940. Primarily Snelting has recently introduced concept analysis to software engineering. Since then it has been used to evaluate class hierarchies [13], explore configuration structures of preprocessor statements [8, 12], understand type relationships [9], and to recover components [2, 5, 10, 11, 14].

The binary relation in our specific application of concept analysis to derive the feature-component correspondence states which subprograms are required when a feature is invoked (without any knowledge of a system’s

architecture, only the most primitive components are known – which are just subprograms). This section describes concept analysis in more detail.

Concept analysis is based on a relation  $R$  between a set of objects  $O$  and a set of attributes  $A$ , hence  $R \subseteq O \times A$ .

The tuple  $C = (O, A, R)$  is called **formal context**. For a set of objects,  $O \subseteq \bar{O}$ , the set of **common attributes**,  $\sigma$ , is defined as:

$$\sigma(O) = \{a \in A \mid \forall (o \in O)(o, a) \in R\}$$

Analogously, the set of **common objects**,  $\tau$ , for a set of attributes,  $A \subseteq \bar{A}$ , is defined as:

$$\tau(A) = \{o \in O \mid \forall (a \in A)(o, a) \in R\}$$

In Section 3.1, the formal context for applying concept analysis to derive the feature-subprogram relationships will be laid down as follows;

- subprograms will be considered objects,
- features will be considered attributes,
- a pair (*subprogram*  $s$ , *feature*  $f$ ) is in relation  $R$  if  $s$  is executed when  $f$  is invoked.

However, here – for the time being – we will use as an abstract example the binary relation between arbitrary objects and attributes shown in Table 1. An object  $o_i$  has attribute  $a_j$  if row  $i$  and column  $j$  is marked with an **X** in Table 1 (the example stems from Lindig and Snelting [5]). For instance, the following equations hold for this table, also known as **relation table**:

$$\sigma(\{o_1\}) = \{a_1, a_2\} \quad \text{and} \quad \tau(\{a_7, a_8\}) = \{o_3, o_4\}$$

	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	a <sub>6</sub>	a <sub>7</sub>	a <sub>8</sub>
o <sub>1</sub>	X	X						
o <sub>2</sub>			X	X	X			
o <sub>3</sub>			X	X		X	X	X
o <sub>4</sub>			X	X	X	X	X	X

**Table 1: Example relation.**

A pair  $(O, A)$  is called **concept** if  $A = \sigma(O) \wedge O = \tau(A)$  holds, i.e., all objects share all attributes. For a concept  $c = (O, A)$ ,  $O$  is the **extent** of  $c$ , denoted by  $extent(c)$ , and  $A$  is the **intent** of  $c$ , denoted by  $intent(c)$ .

Informally, a concept corresponds to a maximal rectangle of filled table cells modulo row and column permutations. For example, Table 2 contains the concepts for the relation in Table 1.

The set of all concepts of a given formal context forms a partial order via:

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2 \quad \text{or equivalently with}$$

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2.$$

If  $c_1 \leq c_2$  holds, then  $c_1$  is called a **subconcept** of  $c_2$  and  $c_2$  is called **superconcept** of  $c_1$ . For instance,

C <sub>1</sub>	({o <sub>1</sub> , o <sub>2</sub> , o <sub>3</sub> , o <sub>4</sub> }, ∅)
C <sub>2</sub>	({o <sub>2</sub> , o <sub>3</sub> , o <sub>4</sub> }, {a <sub>3</sub> , a <sub>4</sub> })
C <sub>3</sub>	({o <sub>1</sub> }, {a <sub>1</sub> , a <sub>2</sub> })
C <sub>4</sub>	({o <sub>2</sub> , o <sub>4</sub> }, {a <sub>3</sub> , a <sub>4</sub> , a <sub>5</sub> })
C <sub>5</sub>	({o <sub>3</sub> , o <sub>4</sub> }, {a <sub>3</sub> , a <sub>4</sub> , a <sub>6</sub> , a <sub>7</sub> , a <sub>8</sub> })
C <sub>6</sub>	({o <sub>4</sub> }, {a <sub>3</sub> , a <sub>4</sub> , a <sub>5</sub> , a <sub>6</sub> , a <sub>7</sub> , a <sub>8</sub> })
C <sub>7</sub>	(∅, {a <sub>1</sub> , a <sub>2</sub> , a <sub>3</sub> , a <sub>4</sub> , a <sub>5</sub> , a <sub>6</sub> , a <sub>7</sub> , a <sub>8</sub> })

**Table 2: Concepts for Table 1.**

({o<sub>2</sub>, o<sub>4</sub>}, {a<sub>3</sub>, a<sub>4</sub>, a<sub>5</sub>}) ≤ ({o<sub>2</sub>, o<sub>3</sub>, o<sub>4</sub>}, {a<sub>3</sub>, a<sub>4</sub>}) is true in Table 2.

The set,  $L$ , of all concepts of a given formal context and the partial order ≤ form a complete lattice, called **concept lattice**:

$$L(C) = \{(O, A) \in 2^O \times 2^A \mid A = \sigma(O) \wedge O = \tau(A)\}$$

The **infimum** of two concepts in this lattice is computed by intersecting their extents as follows:

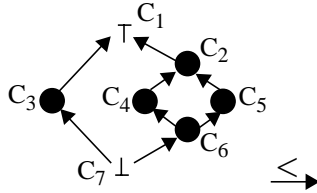
$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

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

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

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

Graphically, the concept lattice for the example relation in Table 1 can be represented as a directed acyclic graph whose nodes represent concepts and whose edges denote the superconcept/subconcept relation < as shown in Figure 2. The most general concept is called the **top element** and is denoted by  $\top$ . The most special concept is called the **bottom element** and is denoted by  $\perp$ .



**Figure 2. Concept lattice for Table 1.**

The combination of the graphical representation in Figure 2 and the contents of the concepts in Table 2 together form the concept lattice. The complete information can be visualized in a more readable equivalent way by marking only the graph node with an attribute  $a \in A$  whose represented concept is the most general concept that has  $a$  in its intent. Analogously, a node will be marked with an object  $o \in O$  if it represents the most special concept that has  $o$  in its extent. The unique element  $\mu$  in the

concept lattice marked with  $a$  is therefore:

$$\mu(a) = \bigvee \{c \in L(C) \mid a \in \text{intent}(c)\} \quad (1)$$

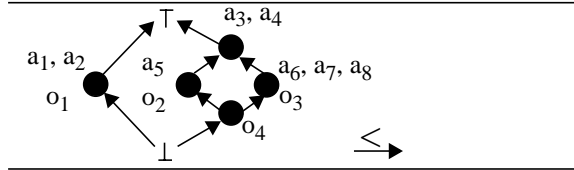
The unique element  $\gamma$  marked with object  $o$  is:

$$\gamma(o) = \bigwedge \{c \in L(C) \mid o \in \text{extent}(c)\} \quad (2)$$

We will call a graph representing a concept lattice using this marking strategy a **sparse representation**. The equivalent sparse representation for Figure 2 is shown in Figure 3. The content of a node  $N$  in this representation can be derived as follows:

- the objects of  $N$  are all objects at and below  $N$ ,
- the attributes of  $N$  are all attributes at and above  $N$ .

For instance, the node in Figure 3 marked with  $o_2$  and  $a_5$  is the concept ({o<sub>2</sub>, o<sub>4</sub>}, {a<sub>3</sub>, a<sub>4</sub>, a<sub>5</sub>}).



**Figure 3. Sparse representation of Figure 2.**

### 3. Feature-Component Correspondence

In order to derive the feature-component correspondence via concept analysis, one has to define the formal context (objects, attributes, relation) and to interpret the resulting concept lattice accordingly.

#### 3.1. Context for Feature and Components

Components will be considered objects of the formal context, whereas features will be considered attributes. Note that in the reverse case, the concept lattice is simply inverted but the derived information will be the same.

The set of relevant features will be determined by the maintainer. For components, we can consider the following alternatives depending on how much knowledge on the system architecture is already available:

1. *cohesive modules* and *subsystems* as defined and documented by the system's architects or re-gained by re-engineers; modules and subsystems will be considered **composite components** in the following;
2. *physical modules*, i.e., modules as defined by means of the underlying programming language or simply directly available as existing files (the distinction to cohesive modules is that one does not know a priori whether physical modules really group cohesive declarations; physical modules are the unscrutinized result of a programmer's way of grouping declarations whether it makes sense or not);

3. *subprograms*, i.e., functions and procedures, and *global variables* of the system; subprograms and global variables will be called **low-level components** in the following.

Ideally, one will use alternative (1) when reliable and complete documentation exists. However, if cohesive modules and subsystems are not known in advance, one would hardly make the effort to analyze a large system to obtain these in order to apply concept analysis to get the feature-component correspondence because it not yet clear which components are relevant at all and reverse engineering of the complete system first will likely not be cost-effective. Only later, if the retrieved feature-component correspondence (using simpler definitions of components, like those in (2) or (3)) clearly shows which lower-level components should be investigated further to obtain composite components, reverse engineering may generally pay off (in order to detect cohesive modules, we have developed a semi-automatic method integrating many automatic state-of-the-art techniques [7]).

Alternative (2) can be chosen if suitable documentation is not available but there is reason to trust the programmers of the system to a great extent. In all other cases, one will fall back on alternative (3). However, for alternative (3), concept analysis may additionally yield hints on sets of related subprograms forming composite components.

The relation for the formal context necessary for concept analysis is defined as follows:

$(C, F) \in R$  if and only if component  $C$  is required when feature  $F$  is invoked; a subprogram is required when it needs to be executed; a global variable is required when it is accessed (used or changed or its address is taken); a composite component is required when one of its parts is required.

In order to obtain the relation, a set of usage scenarios needs to be prepared where each scenario exploits preferably only one relevant feature. Then the system is used according to the set of usage scenarios, one at a time, and the execution traces are recorded. An execution trace contains all required low-level components for a usage scenario or an invoked feature, respectively. If composite components are used for concept analysis, the execution trace containing the required low-level components induces an execution trace for composite components by replacing each low-level component with the composite component to which it belongs. Hence, each system run yields all required components for a single scenario that exploits one feature. Thus, a single column in the relation table can be obtained per system run. Applying all usage scenarios provides the relation table.

An execution trace can be recorded by a profiler. However, most profilers only record subprogram calls but not

accesses to variables. Instead of using a symbolic debugger, for example, that allows to set watchpoints on variable accesses, or even to instrument the code if no sophisticated profiler is available, one can also use a simple static dependency analysis: One considers all variables directly and statically accessed for each executed subprogram also to be dynamically accessed (all transitively accessed variables will automatically be considered because all executed subprograms are examined). In practice, this analysis may be a sufficient approximation. But one should be aware that it may overestimate references because variable accesses may be included that are on paths not executed at runtime, and it will also ignore references to variables by means of aliases if the simple static dependency analysis does not take aliasing into account. For a first analysis to obtain a simplified feature-component correspondence, one can also ignore variables and come back to these in a later phase using more sophisticated dynamic or static analyses.

### 3.2. Interpretation of the Concept Lattice

Concept analysis applied to the formal context described in the last section gives a lattice, from which interesting relationships can be derived. These relationships can be fully automatically derived and presented to the analyst such that the more complicated theoretical background can be hidden. The only thing an analyst has to know is how to interpret the derived relationships. This section explains how interesting relationships can be automatically derived.

As already abstractly described in Section 2, the following base relationships can be derived from the sparse representation of the lattice (note the duality in the interpretation):

- A component,  $c$ , is required for all features at and above  $\gamma(c)$  – as defined by (1) on page 3 – in the lattice.
- A feature,  $f$ , requires all components at and below  $\mu(f)$  – as defined by (2) on page 3 – in the lattice.
- A component,  $c$ , is specific to exactly one feature,  $f$ , if  $f$  is the only feature on all paths from  $\gamma(c)$  to the top element.
- A feature,  $f$ , is specific to exactly one component,  $c$ , if  $c$  is the only component on all paths from  $\mu(f)$  to the bottom element (i.e.,  $c$  is the only component required to implement feature  $f$ ).
- Features, to which two components,  $c_1$  and  $c_2$ , jointly contribute, can be identified by  $\gamma(c_1) \vee \gamma(c_2)$  (the infimum); graphically depicted, one ascertains in the lattice the closest common node toward the top element starting at the nodes to which  $c_1$  and  $c_2$ , respectively,

are attached; all features at and above this common node are those jointly implemented by  $c_1$  and  $c_2$ .

- Components jointly required for two features,  $f_1$  and  $f_2$ , are described by  $\mu(f_1) \wedge \mu(f_2)$  (the supremum); graphically depicted, one ascertains in the lattice the closest common node toward the bottom element starting at the nodes to which  $f_1$  and  $f_2$ , respectively, are attached; all components at and below this common node are those jointly required for these features.
- Components required for all features can be found at the bottom element.
- Features that require all components can be found at the top element.
- If the top element does not contain features, then all components in the top element are superfluous (such components will not exist when the set of objects for concept analysis contains only components executed at least once, which is the case if a filter ignores all subprograms for which the profiler reports an execution count of 0).
- If the bottom element does not contain any component, all features in the bottom element are not implemented by the system (this constellation will not exist, if there is a usage scenario for each feature and every usage scenario is appropriate and relevant to the system).

The information described above can be derived by a tool and fed back to the maintainer, freeing a maintainer not familiar with concept analysis from having to understand the theory of concept lattice.

### 3.3. Implementation

The implementation of the described approach is surprisingly simple (if one already has a tool for concept analysis). Our prototype for a Unix environment is an opportunistic integration of the following parts:

- Gnu C compiler *gcc* to compile the system using a command line switch for generating profiling information,
- Gnu object code viewer *nm*,
- Gnu profiler *prof*,
- concept analysis tool *concepts* [6],
- graph editor *Graphlet* [1] to visualize the concept lattice,
- and a short Perl script to ascertain the executed functions in the execution trace and to convert the file formats of *concepts* and *Graphlet* (the script has just 225 LOC).

The fact that the subprograms are extracted from the object code makes the implementation independent from the programming language to a great extent (as long as the language is compiled to object code) and has the advan-

tage that no additional compiler front end is necessary. On the other hand, because a compiler may replace source names by link names in the object code (for instance, C++ compilers use name mangling to resolve overloading) there is not always a direct mapping from the subprograms in the execution trace back to the original source. Fortunately, tools exist to demangle names and some profilers even demangle names automatically. Because we dealt in our case study with C code, object code names were identical to source names.

## 4. Case Study

As a case study, we analyzed the Xfig system [16] (version 3.2.1) consisting of about 76 KLOCs written in the programming language C. In this section, we will firstly present a general overview of the results and secondly go into further details for particular interesting observations.

Xfig is a menu-driven tool that allows the user to draw and manipulate objects interactively under the X Window System. Objects can be lines, polygons, circles, rectangles, splines, text, and imported pictures. An interesting first task in our case study was to define what constitutes a feature. Clearly, the capability to draw specific objects, like lines, splines, rectangles, etc., can be considered a feature of Xfig. Moreover, one can manipulate drawn objects in different edit modes (rotate, move, copy, scale, etc.) with Xfig. Hence, we considered as main features the following two capabilities:

1. ability to draw different shapes (lines, curves, rectangles, etc.)
2. ability to modify shapes in different editing modes (rotate, move, copy, scale, etc.)

We conducted two experiments. In the first one, we investigated the ability to draw different shapes only. In the second one, we analyzed the ability to modify shapes. The second experiment exemplifies combined features composed by basic features. For the second experiment, a shape is drawn and then modified. Both *draw* and *modify* constitute a basic feature. Combined features add to the effort needed to derive the feature-component correspondence as there are many possible combinations.

In both experiments, we considered subprograms as components. However, in our simple implementation, we do not handle variable accesses. Hence, not all required low-level components are detected.

The resulting concepts contain subprograms grouped together according to their usage for features. Note that the more general subprograms can be found at the lower concepts in the lattice since they are used for many features, while specific components are in the upper region of the lattice. Hence, the concept lattice also reflects the level of

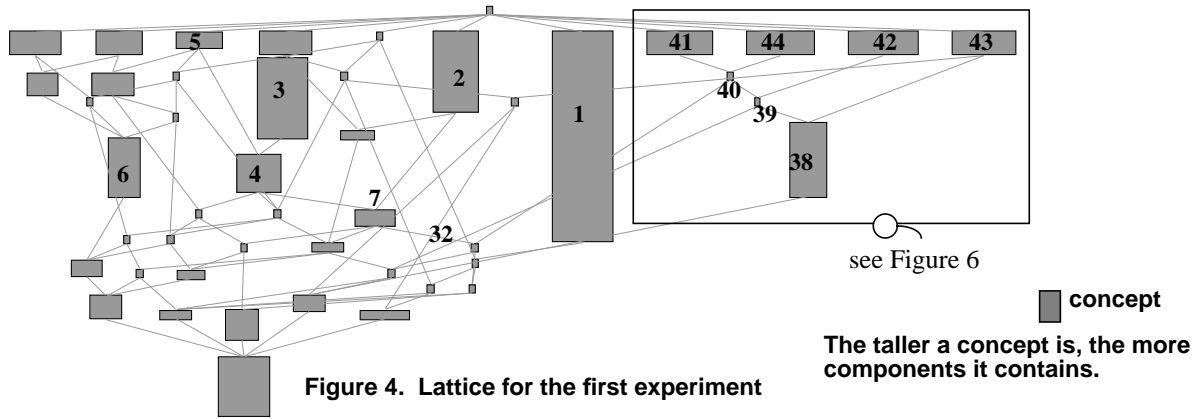


Figure 4. Lattice for the first experiment

abstraction of these subprograms within the given set of scenarios. To identify all subprograms required for a single feature or a set of features, one can then analyze the concept lattice as described in Section 3.2.

**First experiment.** In our first experiment, we prepared 15 scenarios. Each scenario invokes Xfig, performs the drawing of one of the objects Xfig provides, and then terminates Xfig, i.e., the aspects above were not combined and no other functionality of Xfig was used. We used all shapes of Xfig’s drawing panel shown in Figure 5 except *picture objects* and *library objects*.

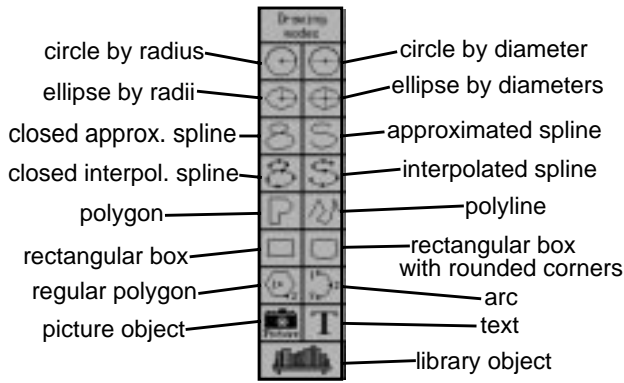


Figure 5. Xfig’s object shapes.

The resulting lattice for this experiment is shown in Figure 4. The contents of the concepts in the lattice are omitted for readability reasons. However, their size in this picture is a linear function of their number of components (except for the bottom element that contains 136 components, mostly initialization and GUI code and very basic functions, and was too large to be drawn accordingly; as a comparison point: the text drawing concept, marked as node #1, has 29 subprograms). As Figure 4 shows, there are a few concepts containing most of the executed subprograms of the system. The lattice contains 47 concepts. 26 of them introduce at least one new component, i.e., to these nodes, a component is attached (more precisely, a

concept  $C$  introduces a component if there exists a component  $c$  for which  $\gamma(c) = C$  holds). 21 of the concepts do not introduce any new component and merely merge functionality needed by several superconcepts.

The first interesting observation is – if one excludes the bottom element – that concepts with many components can be found in the upper region, while in the lower region, the number of components decreases and the number of interferences increases (a lattice is said to be horizontally decomposable if it can be decomposed into independent sublattices that are connected via the top and bottom elements only; an **interference** is an overlap of sublattices that prevents horizontal decomposition). That is to say that there are many specific operations and few shared operations – which may be a sign of little re-use – and also that shared operations are really used for many features. A maintainer would find many application-specific subprograms in the upper region, which he or she would analyze first.

We inspected the concept lattice manually and looked at the source code of the attached subprograms in order to assign meaning to the concepts. Assigning meaning to the concepts was generally simple for concepts in the upper region as they are specific to a subset of all features. Understanding the concepts became increasingly difficult when concepts toward the bottom element were inspected. Moreover, since the lattice does not really reflect the dependencies between components (i.e.,  $\gamma(s_1) > \gamma(s_2)$  does not imply that  $s_1$  calls  $s_2$ ), an additional static analysis of the source code is required. In our experiment, we sometimes needed to browse the call graph.

Concept #1 in Figure 4 is the largest concept (excluding the bottom element). It exploits a single feature “draw text object”. According to the lattice, the feature is largely independent from other features and shares only a few components with other features.

Concept #5 represents the two features “draw polyline” and “draw polygon”. The only difference between these

two features is that an additional line is drawn that closes a polygon. This difference is not visible in the concept lattice since the two features are attached to the same concept. The distinction is made in the body of the function that is called to draw either a polygon or a polyline. Execution traces that only contain called subprograms and not single statements cannot further separate such interleaved subprograms and concept analysis remains uninformed of the difference.

Concept #3 denotes the feature “*draw spline*”. Concept #4 has no feature attached to and represents the components shared for drawing polygons, polylines, and splines. These components are no real drawing operations but operations to keep a log of the points set by the user and to draw lines between set points while the user is still setting points (a spline first appears as polygon and is only reshaped when the user has set all points).

Concept #2 stands for the feature “*draw arc*” and concept #7 is again a concept that represents shared components for drawing elastic lines while the user is setting points. The difference between concept #7 and concept #4 is that the former only contains the components to draw the elastic line, while the latter adds the capability to set an arbitrary number of points. Splines do not need this capability because they are defined by exactly three points.

Concept #6 represents the feature “*draw lines*” and is used for drawing rectangles, polygons, and polylines, as one would expect. The generality of this feature becomes immediately obvious in the concept lattice as it is located in the middle level of the lattice.

The framed area in Figure 4 has a simpler structure than the rest of the lattice. This part deals with circles and ellipses and its details are shown in Figure 6. Each node,  $N$ , in Figure 6 contains two sets: The upper set contains all components attached to the node, i.e., those components,  $c$ , for which  $\gamma(c) = N$ ; the lower set contains all features of  $N$ , including those inherited from other concepts. The names of the features correspond to the objects drawn via the panel in Figure 5; e.g., *draw-ellipse-radius* means that an ellipse was drawn where the radius was specified (as opposed to the diameter).

Nodes #41, #42, #43, and #44 represent the features to draw circles and ellipses using either diameter or radius. They all contain three specific components to draw the object, to plot an elastic bend while the user is drawing, and to resize the object. Note the similarity of the component names. The specific commonalities among circles and ellipses are represented by node #38, which introduces the shared components to draw circles and ellipses (both specified by diameter and radius).

Nodes #32 and #39 connect the circles and ellipses to the other objects. No components are attached to nodes #32 and #39, they only merge components from different

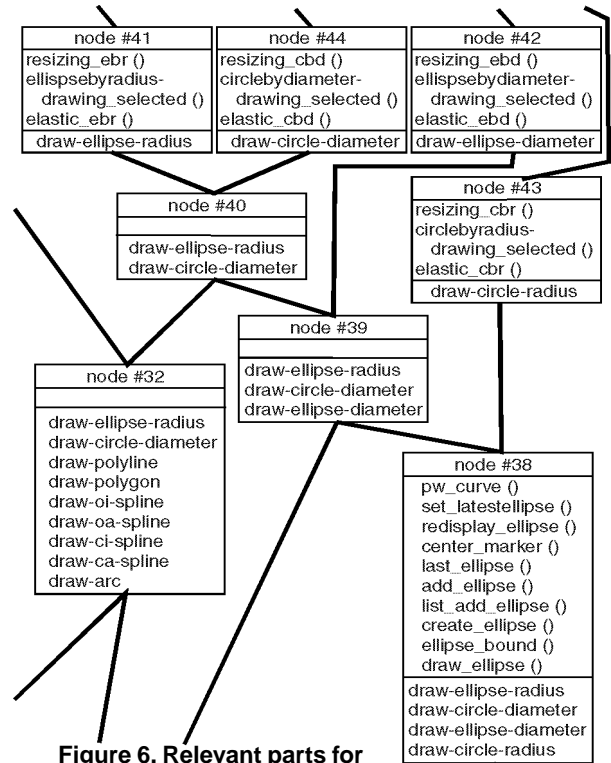


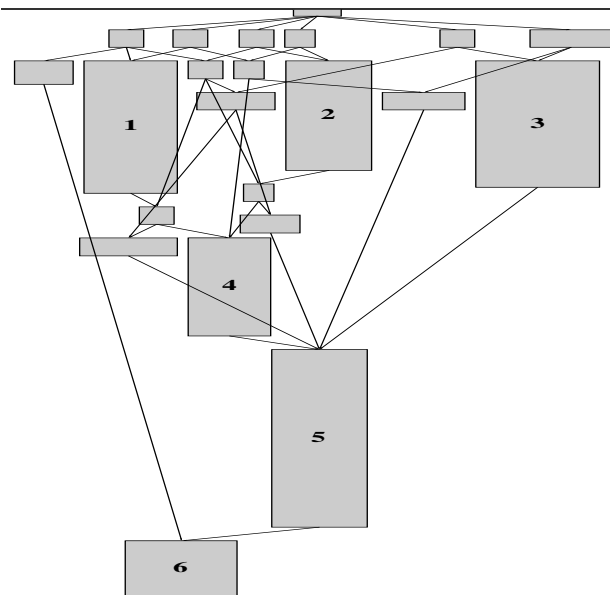
Figure 6. Relevant parts for circles and ellipses

concepts. The two nodes have a direct infimum (not shown in Figure 6) and add the same components to the circle and ellipse features. The components inherited via these two nodes are very basic components of the lowest regions of the lattice, which indicates that ellipses and circles are widely separate from all other objects.

For the problem stated in Section 1, which required us to implement features for drawing segments of ellipses by specifying either radius or diameter, we would now know where to start. Figure 6 identifies the respective subprograms. The distribution of these subprograms into connected concepts according to the invoked features highlights their respective distinctiveness. These subprograms would now be closer investigated by a static analysis related to the code. Figure 6 also identifies the existing components on which the new implementation could be built. One could either integrate the new functionality into the subprograms specific to concepts #41, #42, #43, and #44 (which in this example would probably be the best solution because the new functionality is so close to an existing one) or one could create new components that are based on the components that are already used by #41, #42, #43, and #44 (because the new solution requires the same infrastructure) and analyse #41, #42, #43, and #44 to find out how this infrastructure is properly used.

**Second experiment.** Once objects are drawn, they can be edited. For instance, objects can be rotated. As a matter of

fact, this functionality needs to be provided for new segmented ellipses, too. In a second experiment, we thus analyzed the edit mode *rotate* which comes in two variants: clockwise and counterclockwise. The first ten shapes in Figure 5 were drawn and rotated once clockwise and once counterclockwise, which resulted in 20 scenarios. The resulting lattice contained 55 concepts, most of them introduce no new component. We observed that the related shapes, i.e., the variants of splines, circles, ellipses, etc., were merged at the top of the lattice since they use almost the same components. In order to reduce the size of the lattice, we selected one representative among the related shapes and re-run the experiment with three shapes (ellipse, polygon, and open approximated spline). The resulting lattice is shown in Figure 7.



**Figure 7. Concept lattice for second experiment.**

This lattice consists of 22 concepts, three of them provide the specific functionality for the respective shapes. Concept #1 (21 functions) depicts the functionality for splines and concept #2 (17 functions) represents the one for lines (used for polygons). Both are dependent on concept #4 (29 functions) that groups functions related to points. Concept #3 (20 functions) denotes the ellipse feature, concept #5 (29 functions) the general drawing support functionality and concept #6 (123 functions) the start-up and initialization code of the system.

Analyzing concepts #1, #2, and #3, we found that the shapes provide individual rotate functions. In other words, the rotate feature is implemented specific to each shape, i.e., there is no generic component that draws all different shapes, which would have been an interesting finding in terms of re-use. That means that we would have to add a new rotate operation for our new segmented ellipses or to

modify the rotate operation for ellipses, which can be identified in the concept lattice.

**General observations.** We made the experience that applying our method is easy in principle. However, running all scenarios by hand is time consuming. It may be facilitated by the presence of test cases that allow an automated replay of various scenarios.

Because Xfig has a GUI, running a single scenario by hand is an easy task. However, one has to pay attention not to cause interferences by invoking irrelevant features. For instance, Xfig uses a balloon help facility that pops up a little window when the cursor stays some time on a sensitive area of the GUI (e.g., over the button selecting the circle drawing mode). Sometimes the balloon help mechanism triggers, introducing interferences between features. Such effects affect the analysis because they introduce spurious connections between features. Fortunately, this problem can be partly fixed by providing a specific scenario in which only the accidentally invoked irrelevant feature is invoked, which leads to a refactored concept lattice that contains a new concept that isolates the irrelevant feature and its components. In our example, interferences due to an accidentally invoked irrelevant feature appeared only at the two layers directly on top of the bottom element of the lattice, and could be more or less ignored.

## 5. Related Research

For feature localization, Chen and Rajlich [3] propose a semi-automatic method, in which an analyst browses the statically derived dependency graph; navigation on that graph is computer-aided. Since the analyst more or less takes on all the search, this method is less suited to quickly and cheaply derive the feature-component correspondence. Moreover, the method relies on the quality of the static dependency graph. If this graph, for example, does not contain information on potential values of function pointers, the human analyst may miss functions only called via function pointers. At the other extreme, if the too conservative assumption is made that every function whose address is taken is called at each function pointer call site, the search space increases extremely. Generally, it is statically undecidable which paths are taken at runtime, so that every static analysis will yield an overestimated search space, whereas dynamic analyses exactly tell which parts are really used at runtime (though for a particular run only). However, Chen and Rajlich's technique could be helpful in a later phase, in which the system needs to be more rigorously analyzed. The purpose of our technique is to derive the feature-component correspondence. It handles the system as a black box and, hence, does not give insights in internal aspects with respect to

dependencies and quality.

Wilde and Scully [15] also use dynamic analysis to localize features as follows:

1. The *invoking input set*  $I$  (i.e., a set of test cases or – in our terminology – a set of usage 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 traces, the components can be identified that implement the feature.

Wilde and Scully focus on localizing required components for one specific feature rather than analyzing commonalities and variabilities of related features: For localizing all required components, the execution trace for the including input set is sufficient. By subtracting all components in the execution trace for the excluding input set from those in the execution trace for the invoking input set, only those components remain that specifically deal with the feature.

Note that our technique achieves the same effect if one adds a usage scenario that only starts and ends the system immediately. By considering several execution traces for different features at a time, components not specific to a feature will “sink” in the concept lattice, i.e., will be closer to the bottom element. More precisely, recall from Section 3.2 that a component,  $c$ , is specific to exactly one feature,  $f$ , if  $f$  is the only feature on all paths from  $\gamma(c)$  to the top element. Start-up and finalization subprograms will hence gather in the bottom element.

Our technique goes beyond Wilde and Scully’s technique in that it also allows to derive commonalities and variabilities between components and related features by means of concept analysis, whereas Wilde and Scully’s technique only localizes a single feature. The derived commonalities and variabilities are an import information to a maintainer who needs to understand the system.

## 6. Conclusions

The feature-component correspondence describes which components are required to implement a set of related features and what are the commonalities and variabilities among these set of related features and components. The feature-component correspondence is useful

1. to identify all components that contribute to a certain feature,
2. to identify all features to which a component contributes,

3. to find out which components are jointly needed to implement only a subset of all features,

4. and to understand to which features several components jointly contribute.

The technique presented in this paper yields the feature-component correspondence automatically using execution traces for different usage scenarios each invoking a feature. Fact (1) and (2) above could also be derived by the approach of Wilde and Scully [15] in which one analyses the execution trace for each feature separately. Fact (3) and (4) – which represent the commonalities and variabilities of features and components, respectively – are additionally revealed by applying concept analysis, a sound mathematical technique to analyze binary relations. Moreover, the resulting concept lattice reflects the level of feature specificity of the required components.

The technique is primarily suited for functional features that may be mapped to components. In particular non-functional features do not easily map to components. For example, for applications for which timing is critical (because it may result in diverging behavior), the features would also have to take time into account.

Note also that the technique is not suited for features that are only internally visible, like whether a compiler uses a certain intermediate representation. Internal features can only be detected by looking at the source, because it is not clear how to invoke them from outside and how to derive from an execution trace whether these features are present or not.

The invocation for externally visible features is comparatively simple when a graphical user interface is available (as it was the case in our case study). Then, usually only a menu selection or a similar interaction is necessary. In the case of a batch system, one may vary command line switches and may have to provide different sets of test data to invoke a feature. However, in order to find suitable test data, one might need some knowledge on internal details of a system.

Furthermore, the success of the described approach heavily depends on the clever choice of usage scenarios and the combination of them. Scenarios that cover too much functionality in one step or the clumsy combination of scenarios will result in huge and complex lattices that are unreadable for humans. Moreover, the number of usage scenarios increases tremendously when features are combined.

The implementation of this technique was surprisingly simple. In one day, we opportunistically put together a set of publicly available tools and wrote a single Perl script (225 LOC in total) for interoperability. A drawback of our simple implementation is that one has to run the system for each usage scenario from the beginning to get an exe-

cution trace for each feature. A more sophisticated environment would allow to start and end recording traces at any time.

Our implementation only counts subprogram calls and ignores accesses to global variables and single statements or expressions. It might be useful to analyze at a finer granularity, in particular when subprograms are interleaved, i.e., different strands of control with different functionality are united in a single subprogram, possibly for efficiency reasons. For instance, we have found a subprogram in our case study that draws different kinds of objects. The function contained a large switch statement whose branches drew the specific kinds of objects. In the execution trace, this subprogram showed up for all objects where in fact only specific parts of it were actually executed.

In our case study, the method provided us with valuable insights. The lattice revealed dependencies among features for the Xfig implementation and the absence of such dependencies, respectively; e.g., the abilities to draw text and circles/ellipses are widely independent from other shapes. Related features were grouped together in the concept lattice, which allowed us to compare our mental model of a drawing tool to the actual implementation of Xfig. The lattice also classified components according to their abstraction level, which is a useful information for re-use; general components can be found at the lower level, specific components at the upper level.

Whether we can generalize our results to other systems remains to be further investigated. In particular, even though we found this information very useful in our case study, it is to be shown by additional case studies and controlled experiments whether this information really helps a maintainer to understand a system and whether repeatable results can be obtained with different people.

As future work, beyond further case studies and controlled experiments, we want to explore how results obtained by the method described in this paper may be combined with results of additional static analyses. For example, we want to investigate the relation between the concept lattice based on dynamic information and static software architecture recovery techniques.

## References

- [1] Brandenburg, F.J., 'Graphlet', Universität Passau, <http://www.infosun.fmi.uni-passau.de/Graphlet/>.
- [2] Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G.A., 'A Case Study of Applying an Eclectic Approach to Identify Objects in Code', *Workshop on Program Comprehension*, pp. 136-143, Pittsburgh, 1999, IEEE Computer Society Press.
- [3] Chen, K. und Rajlich, V., 'Case Study of Feature Location Using Dependence Graph', *Proc. of the 8th Int. Workshop on Program Comprehension*, pp. 241-249, June 10-11, 2000, Limerick, Ireland, IEEE Computer Society Press.
- [4] Graudejus, H., *Implementing a Concept Analysis Tool for Identifying Abstract Data Types in C Code*, master thesis, University of Kaiserslautern, Germany, 1998.
- [5] Lindig, C. and Snelting, G., 'Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis', *Proc. of the Int. Conference on Software Engineering*, pp. 349-359, Boston, 1997.
- [6] Lindig, C., Concepts, <ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/misc>.
- [7] Koschke, R., 'Atomic Architectural Component Recovery for Program Understanding and Evolution', Dissertation, Institut für Informatik, Universität Stuttgart, 2000, <http://www.informatik.uni-stuttgart.de/ifi/ps/rainer/thesis>.
- [8] Krone, M. and Snelting, G., 'On the Inference of Configuration Structures From Source Code', *Proc. of the Int. Conference on Software Engineering*, pp. 49-57, May 1994, IEEE Computer Society Press.
- [9] Kuipers and Moonen, L., 'Types and Concept Analysis for Legacy Systems', *Proc. the International Workshop on Program Comprehension, IWPC*, IEEE Computer Society Press, 2000.
- [10] Sahraoui, H., Melo, W, Lounis, H., and Dumont, F. (1997), 'Applying Concept Formation Methods to Object Identification in Procedural Code', *Proc. of the Conference on Automated Software Engineering*, Nevada, pp. 210-218, November, IEEE Computer Society.
- [11] Siff, M. and Reps, T., 'Identifying Modules via Concept Analysis', *Proc. of the Int. Conference on Software Maintenance*, Bari, pp. 170-179, October, 1997, IEEE Computer Society.
- [12] Snelting, G., 'Reengineering of Configurations Based on Mathematical Concept Analysis', *ACM Transactions on Software Engineering and Methodology* 5, 2, pp. 146-189, April, 1997.
- [13] Snelting, G. and Tip, F., 'Reengineering Class Hierarchies Using Concept Analysis', *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 99-110, November, 1994.
- [14] Van Deursen, A. and Kuipers, 'Identifying Objects Using Cluster and Concept Analysis', *Proc. of the International Conference on Software Engineering*, IEEE Computer Society Press, 1999.
- [15] Wilde, N. and Scully, M.C., 'Software Reconnaissance: Mapping Program Features to Code', *Software Maintenance: Research and Practice*, vol. 7, pp. 49-62, 1995.
- [16] Xfig system, <http://www.xfig.org>.