Locating features in source code. IEEE Trans Softw Eng

Article *in* IEEE Transactions on Software Engineering · April 2003

DOI: 10.1109/TSE.2003.1183929 · Source: IEEE Xplore

3 authors, including:

Rainer Koschke
Universität Bremen
196 PUBLICATIONS **6,625** CITATIONS

Locating Features in Source Code

Thomas Eisenbarth, Rainer Koschke, and Daniel Simon

Abstract— **Understanding the implementation of a certain feature of a system requires to identify the computational units of the system that contribute to this feature. In many cases, the mapping of features to the source code is poorly documented. In this paper, we present a semi-automatic technique that reconstructs the mapping for features that are triggered by the user and exhibit an observable behavior.**

The mapping is in general not injective; that is, a computational unit may contribute to several features. Our technique allows to distinguish between general and specific computational units with respect to a given set of features. For a set of features, it also identifies jointly and distinctly required computational units.

The presented technique combines dynamic and static analyses to rapidly focus on the system's parts that relate to a specific set of features. Dynamic information is gathered based on a set of scenarios invoking the features. Rather than assuming a one-to-one correspondence between features and scenarios as in earlier work, we can now handle scenarios that invoke many features.

Furthermore, we show how our method allows incremental exploration of features while preserving the "mental map" the analyst has gained through the analysis.

Keywords— **program comprehension, formal concept analysis, feature location, program analysis, software architecture recovery**

I. Introduction

UNDERSTANDING how a certain feature is implemented is a major problem of program understanding. Before real understanding starts, one has to locate the implementation of the feature in the code. Systems often appear as a large number of modules each containing hundreds of lines of code. It is in general not obvious which parts of the source code implement a given feature. Typically existing documentation is outdated (if it exists at all), the system's original architects are no longer available, or their view is outdated due to changes made by others. So maintenance introduces incoherent changes which cause the system's overall structure to degrade [1]. Understanding the system in turn becomes harder any time a change is made to it.

One option, when trying to escape this vicious circle, is to completely reverse engineer the system in order to exhaustively identify its components and to assign features to components. We integrated published automatic techniques for component retrieval in an incremental semi-automatic process, in which the results of selected automatic techniques are validated by the user [2].

However, exhaustive methods are not cost-effective. Fortunately, knowledge of components implementing a specific set of features suffices in many cases. Consequently, a feature-oriented search focusing on the components of interest is needed.

T. Eisenbarth, R. Koschke, and D. Simon are with the Institute of Computer Science at the University of Stuttgart, Breitwiesenstraße 20–22, D-70565 Stuttgart, Germany. E-mail: {eisenbarth,simon,koschke}@informatik.uni-stuttgart.de.

This article describes a process and its supporting techniques to identify those parts of the source code which implement a specific set of related features. The process is automated to a large extent. It combines static and dynamic analyses and uses concept analysis—a mathematical technique to investigate binary relations—to derive correspondences between features and computational units. Concept analysis additionally yields the computational units jointly and distinctly required for a set of features.

An advantage of starting with features is that domain knowledge from the user's perspective may be exploited, which is especially useful for external change requests and error reports expressed in the terminology of a program's problem domain.

The remainder of this article is organized as follows. Sect. II gives an overview of our technique and introduces the basic concepts. Sect. III introduces concept analysis. Sect. IV describes the process for locating and analyzing features in more detail. In Sect. V, we report on two case studies conducted to validate our approach. The related research in the area is summarized in Sect. VI.

II. Overview

The goal of our technique is to identify the computational units that specifically implement a feature as well as the set of jointly or distinctly required computational units for a set of features. To this end, the technique combines *static* and *dynamic* analyses.

This section gives an overview on our technique, describes the relationships among features, scenarios, and computational units (summarized in Fig. 1) and explains what kind of dynamic information is used as input to our technique. The section also introduces a simple example that we will use throughout the description of the method in the following sections. The example is inspired by a previous case study [3] in which we analyzed the drawing tool XFIG [4].

Computational unit. A **computational unit** is an executable part of a system. Examples for computational units are instructions (like accesses to global variables), basic blocks, routines, classes, compilation units, components, modules, or subsystems. The exact specification of a computational unit is a generic parameter of our method.

Feature. A **feature** is a realized functional requirement of a system (the term feature is intentionally defined weakly because its exact meaning depends on the specific context). Generally, the term *feature* also subsumes non-functional requirements. In the context of this paper, only *functional* features are relevant; that is, we consider a feature an ob-

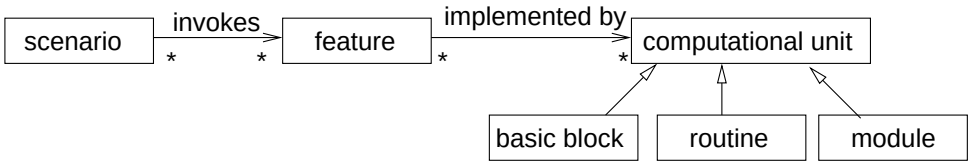


Fig. 1. Conceptual model in UML notation.

servable behavior of the system that can be triggered by the user.

EXAMPLE. Our fictitious drawing tool FIG (which resembles XFIG [4]) allows a user to draw, move, and color different objects, such as rectangles, circles, ellipses, and so forth. From the viewpoint of an analyst who is interested in the implementation of circle operations in FIG, the ability to draw, to move, and to color a circle are three relevant features. □

Every computational unit (excluding dead code) contributes to the purpose of the system and thus corresponds to at least one feature—be it a very basic feature, such as the ability of the system to start or terminate. Yet, only few features may actually be of interest to the analyst for her task at hand. In the following, we assume that only a subset of features is relevant. Consequently, only the computational units required for these features are of interest, too. The **feature-unit map**—as one result of our technique— describes which computational units implement a given set of relevant features.

Scenario. Features are abstract descriptions of a system's expected behavior. If a user wants to invoke a feature of a system, he needs to provide the system with adequate input to trigger the feature. For instance, to draw a circle, the user of FIG needs to press a certain button on the control panel for selecting the circle drawing operation, then to position the cursor on the drawing area for specifying the center of the circle, to specify the diameter by moving the mouse, and eventually to press the left mouse button for finalizing the circle. Such sequences of user inputs that trigger actions of a system with observable result [5] are called **scenarios**.

Our technique requires a set of scenarios that invoke the features the analyst is interested in. A scenario s **invokes** a feature f if f*'s* result can be observed by the user when the system is used as described by scenario s. A scenario may invoke multiple features and features may be invoked by multiple scenarios. For instance, a scenario for moving a circle requires to draw the circle first, so this scenario also invokes feature "circle drawing". There may be even different scenarios all invoking the same set of features. Each scenario, then, represents an alternative way of invoking the features. For instance, FIG allows a user to push a button or to use a keyboard shortcut to begin a circle drawing operation. A set of scenarios each representing options and choices for the same feature resembles a use case.

Scenarios are used in our technique to gather the computational units for the relevant features through dynamic analysis, similarly to Wilde and Scully's technique [6]. If the system is used as described by the scenario, the **execution trace** lists the sequence of all performed calls for this scenario. Since our technique aims at only identifying the computational units rather than at the order of the computational units' execution, we need only the execution profile. The **execution profile** of a given program run is the set of computational units called during the run without information about the order of execution. From the execution profile, we gather the fact that a computational unit has been executed at least once. We ignore the duration of the computational unit's execution because computation time hardly gives hints for feature-specific computational units. Once the specific computational units have been identified through our technique, other techniques, such as static or dynamic slicing [7], [8], can be used to obtain the order of execution if required. These techniques can then be applied more goal-oriented by focusing on the most feature-specific computational units yielded by our technique.

Feature-unit map. Our technique derives the feature-unit map through *concept analysis*, a mathematically sound technique. In our application of concept analysis, concept analysis—simply stated—mutually intersects the execution profiles for all scenarios and all resulting intersections to obtain the specific computational units for a feature and the jointly and distinctly required computational units for a set of features.

EXAMPLE. FIG allows to draw a circle either by diameter or by radius. The analyst who is interested in the differences of these two circle operations and their differences to other circle operations, such as moving and coloring, will set up the scenarios listed in Fig. 2. Figure 3 lists the computational units executed for the scenarios in Fig. 2. Intersecting the execution profiles shows that *setRadius* is specific to feature *Draw-circle-radius*, *move* to *Move-circle*, and *color* to *Color-circle*. □

scenario name	*actions performed*
Draw-circle-diameter	draw a circle by diameter
Draw-circle-radius	draw a circle by radius
Move-circle	draw a circle by diameter and move it
Color-circle	draw a circle by diameter and color it

Fig. 2. Example scenarios for FIG.

Beyond simply identifying the computational units

scenario	*executed computational units*
Draw-circle-diameter	draw, setDiameter
Draw-circle-radius	draw, setRadius
Move-circle	draw, setDiameter, move
Color-circle	draw, setDiameter, color

Fig. 3. Execution profiles for Fig. 2.

specifically required for a feature, concept analysis additionally allows to derive detailed relationships between features and computational units. These relationships identify computational units jointly required by any subset of features and classify computational units as low-level or high-level with respect to the given set of features.

EXAMPLE. Intersecting the execution profiles in Fig. 3 additionally shows that the computational units jointly required for *Draw-circle-diameter*, *Move-circle*, and *Color-circle* are *draw* and *setDiameter*, where *draw* is required for all scenarios. □

The information gained by concept analysis is used to guide a subsequent static analysis along the static dependency graph in order to narrow the computational units to those that form self-contained and understandable feature-specific computational units. Computational units that are only very basic computational units used as building blocks for other computational units but not containing any application-specific logic are sorted out. Additional static analyses, like strongly connected component identification, dominance analysis, and program slicing [7] support the search for the units of interest.

For large and complex systems, our approach can be applied incrementally as described in this paper.

Applicability

The retrieval of the feature-unit map is based on dynamic information where all computational units that are executed for a scenario are collected. The scenario describes how to invoke a feature. This section describes the assumptions on features, scenarios, and computational units we make.

Features. Our technique is primarily suited for functional features that may be mapped onto computational units. In particular, non-functional features, such as robustness, reliability, or maintainability, do not easily map to computational units.

The technique is suited only for features that can be invoked from outside; internal implementation features, such as the use of a garbage collector, may not necessarily be deterministically and easily triggered from outside.

Scenarios. Scenarios are designed (or selected from existing test cases) to invoke a known set of relevant features; that is, we assume that the analyst knows in advance which features are invoked by a scenario.

Because suitable scenarios are essential to our technique, a domain expert is needed to set up scenarios. In many cases, the domain expert can reuse existing test cases as scenarios to locate features. However, the purpose of test cases is to reveal errors, and hence test cases tend to be complex and to cover many features. Contrarily, scenarios for our feature location technique should be simpler and invoke fewer features to differentiate the computational units more clearly.

In order to explore variations of a feature, the domain expert provides several scenarios, each triggering a feature variation with a different set of input. To obtain effective and efficient coverage, he builds equivalence classes of relevant input data. Identifying equivalence classes may require knowledge on internal details of a system.

Computational units. The exact notion of computational unit is a generic parameter to our technique and depends on the task and system at hand. In principle, there is no limit to the granularity of computational units: One could use basic blocks, routines, classes, modules, or subsystems. Subsystems as computational units are suitable to obtain an overview for very large systems. Considering routines, methods, subprograms, etc. as computational units gives an overview at the global declaration level, whereas classes and modules lie in between subsystem and global declaration level. Basic blocks as computational units are only adequate for smaller systems or parts of a system where more detail is needed due to the likely information overload to the analyst.

For practical reasons, for this paper we decided to use routines as the computational unit of choice, where a *routine* is a function, procedure, subprogram, or method according to the programming language. For the case studies presented later on in this paper, routines were appropriate.

Static and dynamic dependencies. The results from concept analysis based on dynamic information are used to guide the analyst in her static analysis, that is, her inspection of the static dependency graph. We use dynamic information only as a guide and not as a definite answer because dynamic information depends upon suitable input data and the test environment in which the scenarios are executed.

The static dependency graph can be extracted from procedural, functional, as well as object-oriented programming languages. Because execution profiles can be recorded for these languages, too, our technique is applicable to all these languages. However, the precision of the static extraction influences the ease of the analyst's inspection of the static dependencies, and static analysis is inherently more difficult for object-oriented languages (and for functional languages with higher-order functions) than for procedural languages.

Static analyses need to make conservative assumptions in the presence of pointers and dynamic binding, which weaken the precision of the dependency graph. Fortunately, research in pointer analysis has made considerable progress. There is a large body of work on pointer analysis for procedural languages [9], [10], [11], [12], [13], [14], [15], [16] and object-oriented languages [17], [18] that resolves general pointers, function pointers, and dynamic binding. These techniques vary in precision and costs. Interestingly enough, Milanova and others have recently

presented empirical data indicating that less expensive and—theoretically—less precise techniques to resolve function pointers reach the precision of more expensive and—theoretically—more precise techniques [19] due to the common way of using function pointers (as opposed to pointers to stack and heap objects).

III. Formal Concept Analysis

This section presents the necessary background information on formal concept analysis. Readers already familiar with concept analysis can skip to the next section.

Formal concept analysis is a mathematical technique for analyzing binary relations. The mathematical foundation of concept analysis was laid by Birkhoff [20] in 1940. For more detailed information on formal concept analysis we refer to [21], where the mathematical foundation is explored.

Concept analysis deals with a relation $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{A}$ between a set of objects $\mathcal{O}$ and a set of attributes $\mathcal{A}$. The tuple $C = (\mathcal{O}, \mathcal{A}, \mathcal{I})$ is called a **formal context**. For a set of objects $O \subseteq \mathcal{O}$, the set of common attributes $\sigma(O)$ is defined as:

$$\sigma(O) = \{a \in A \mid (o, a) \in \mathcal{I} \text{ for all } o \in O\} \tag{1}$$

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

$$\tau(A) = \{o \in O \mid (o, a) \in \mathcal{I} \text{ for all } a \in A\} \tag{2}$$

A formal context can be represented by a relation table, where the columns hold the objects and the rows hold the attributes. An object o_i and attribute a_j are in the relation $\mathcal{I}$ iff the cell at column i and row j is marked by "×". As an example, a binary relation between arbitrary objects and attributes is shown in Fig. 4(a). For that formal context, we have:

$$\begin{aligned} \sigma(\{o_1\}) &= \{a_1, a_4, a_6, a_7\} \\ \tau(\{a_6, a_7\}) &= \{o_1, o_3\} \end{aligned}$$

A tuple $c = (O, A)$ is called a **concept** iff $A = \sigma(O)$ and $O = \tau(A)$, that is, all objects in c share all attributes in c. For a concept $c = (O, A)$, O is called the **extent** of c, denoted by $\mathit{extent}(c)$, and A is called the **intent** of c, denoted by $\mathit{intent}(c)$. Informally speaking, a concept corresponds to a maximal rectangle of filled table cells modulo row and column permutations. In Fig. 4(b), all concepts for the relation in Fig. 4(a) are listed.

The set of all concepts of a given formal context forms a partial order via the superconcept-subconcept ordering $\leq$:

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2 \tag{3}$$

or, dually, with

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2 \tag{4}$$

Note that (3) and (4) imply each other by definition. If we have $c_1 \leq c_2$, then c_1 is called a **subconcept** of c_2 and c_2 is called **superconcept** of c_1. For instance, in Fig. 4(b) we have $c_4 \leq c_2$.

The set $\mathcal{L}$ of all concepts of a given formal context and the partial order $\leq$ form a complete lattice, called **concept lattice**:

$$\mathcal{L}(C) = \{(O, A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}} \mid A = \sigma(O) \text{ and } O = \tau(A)\} \tag{5}$$

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

$$(O_1, A_1) \sqcap (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2)) \tag{6}$$

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

$$(O_1, A_1) \sqcup (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2) \tag{7}$$

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

The concept lattice for the formal context in Fig. 4(a) can be depicted as a directed acyclic graph whose nodes represent the concepts and whose edges denote the superconcept-subconcept relation $\leq$ as shown in Fig. 5(a). 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 $\bot$.

The concept lattice can be visualized in a more readable equivalent way by marking only the 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 $o \in \mathcal{O}$ iff it represents the most special concept that has o in its extent. The unique element in the concept lattice marked with a is therefore:

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

The unique element marked with object o is:

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

We will call a graph representing a concept lattice using this marking strategy a **sparse representation** of the lattice. The equivalent sparse representation of the lattice in Fig. 5(a) is shown in Fig. 5(b). 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 Fig. 5(b) marked with o_1 and a_1 is the concept $c_4 = (\{o_1\}, \{a_1, a_4, a_6, a_7\})$.

For practical reasons, it is sometimes useful to apply only one of (8) or (9). For example if we have a large number of attributes but just a small number of objects, we eliminate the redundant appearance of attributes and keep the full list of objects in the concepts.

IV. Analysis Process

Our process to locate features is depicted in Fig. 6 using the IDEF0 notation [22]. It consists of five major activities:

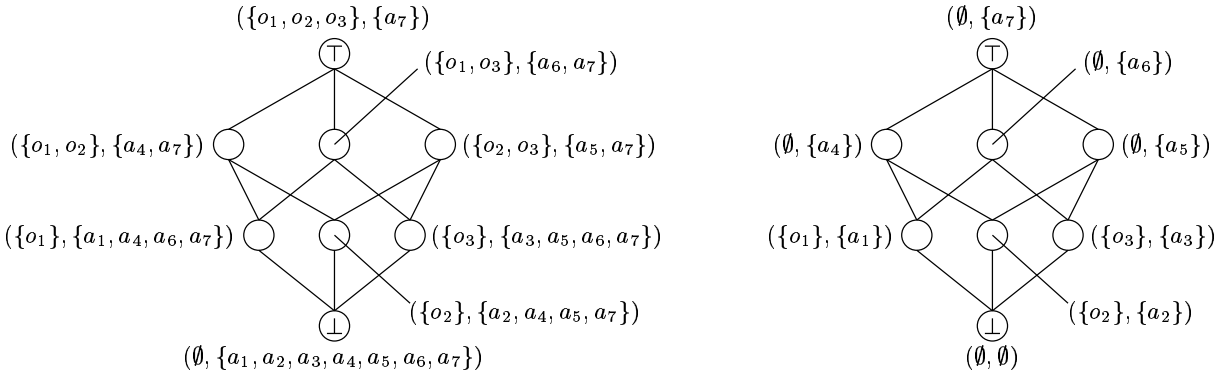
	a_1	a_2	a_3	a_4	a_5	a_6	a_7
o_1	×			×		×	×
o_2		×		×	×		×
o_3			×		×	×	×

(a) A formal context.

$\top$	$(\{o_1, o_2, o_3\}, \{a_7\})$
c_1	$(\{o_1, o_2\}, \{a_4, a_7\})$
c_2	$(\{o_1, o_3\}, \{a_6, a_7\})$
c_3	$(\{o_2, o_3\}, \{a_5, a_7\})$
c_4	$(\{o_1\}, \{a_1, a_4, a_6, a_7\})$
c_5	$(\{o_2\}, \{a_2, a_4, a_5, a_7\})$
c_6	$(\{o_3\}, \{a_3, a_5, a_6, a_7\})$
$\bot$	$(\emptyset, \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\})$

(b) Concepts for the formal context.

Fig. 4. An example relation between objects and attributes. The corresponding concepts that can be derived from the formal context are listed on the right.



(a) Full concept lattice.

(b) Sparse representation.

Fig. 5. The concept lattices for the example context in Fig. 4.

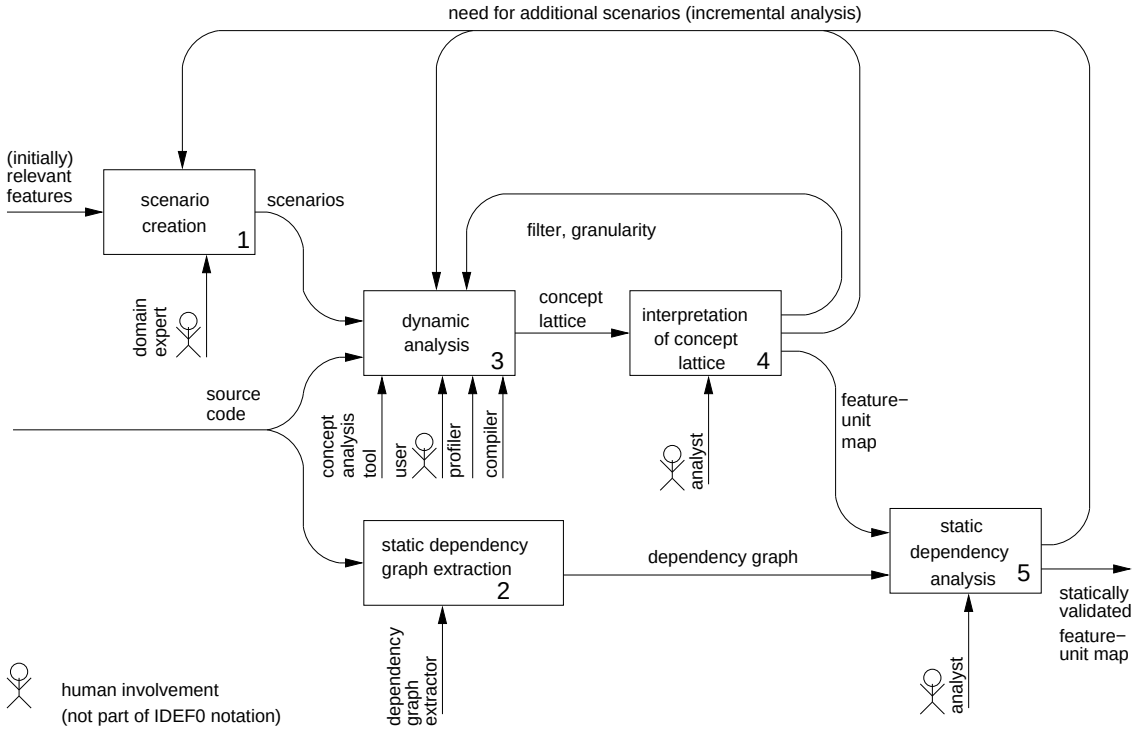


Fig. 6. Process for feature location in IDEF0 notation.

1. Scenario creation: Based on features (either known initially or discovered during incremental analysis), the domain expert creates scenarios.
2. Static dependency-graph extraction: The static dependency graph of the system under analysis is extracted.
3. Dynamic analysis: The system is used according to selected scenarios.
4. Interpretation of concept lattice: The data yielded by the dynamic analysis is presented to and interpreted by the analyst. Relevant computational units are identified.
5. Static dependency analysis: The analyst searches the system for additional computational units that are relevant to selected features.

The different roles of human resources for these activities are (human resources are highlighted in the process diagrams by a UML actor icon):
- The **analyst** is the person interested in how features map onto source code. She interprets the concept lattice and performs the static analysis.
- The **domain expert** designs the scenarios and lists the invoked features for each scenario.
- The **user** is the person who uses the system according to the selected scenarios.

All activities except the static dependency graph extraction (which is done only once) benefit from the knowledge that is gained in previous iterations and can be applied repeatedly until sufficient knowledge about the system has been gained. The order of the activities is specified by the IDEF0 diagram in Fig. 6: An activity may start once its input is available. The activities are explained in the following sections.

A. Static Dependency Graph Extraction

The static dependency graph should subsume all types of entities and dependencies present in the dynamic dependency graph: It is unnecessary to extract dynamic information that is not used in the subsequent static analysis. Yet, the static dependency graph may provide additional types of entities and dependencies and also more fine-grained information if a static extraction tool is used that exceeds the capabilities of the available dynamic extraction tool. In this case, the static analysis can leverage less dynamic information but is still conservative. In our case studies, for instance, we extracted many detailed static dependencies among global declarations (routines, global variables, and user-defined types) but the profiler we used let us only extract the dynamic call relationship among routines. This way, we had to analyze static variable accesses that might have never been executed in any of our scenarios.

B. Scenario Creation

A domain expert is needed for creating the scenarios. Any available information on the system's behavior (e.g., documentation, existing test cases, domain models, etc.) is useful as input to him. Existing test cases may be useful but not necessarily directly applicable, because the focus during testing is to cover the code completely and to combine features in many ways. Scenarios in our sense are very distinctive; that is, they should invoke all relevant features but as few other features as possible to ease the mappings from scenarios to features and from features to computational units (often it is unavoidable to invoke features that are not of interest for the task at hand).

The scenarios are documented for future use similarly to test cases. Additionally, the documentation includes the features invoked by the scenarios. If the domain experts also specifies the expected result of the scenario, the scenario may also be used as simple test case.

C. Dynamic Analysis

The goal of the dynamic analysis is to find out which computational units contribute to a given set of features. Each feature is invoked by at least one of the prepared scenarios.

The process that deals with the dynamic analysis is shown in more detail in Fig. 8. The inputs to the process are source code and a set of scenarios created by process step 1 in Fig. 6. We proceed as follows:
3.1 Compile for recording: The source code is compiled with profiling options or is instrumented to obtain the execution profile.
3.2 Scenario execution: The system is executed by a user according to the scenarios and execution profiles are recorded.

If suitable tool support is available, a scenario's execution may be recorded at wish to exclude parts of the execution that are not relevant, such as start-up and shutdown of the system [23], [24], [25]. Certain debuggers, for instance, allow to start and end trace recording. Instrumenting the source code so that only relevant parts are recorded is generally not an option because this requires that the feature-unit map is at least partially known already.

An alternative solution is to specify a special "start-end" scenario containing the actions to be filtered out. For instance, in order to mask out initialization and finalization code, the domain expert may prepare a "start-end" scenario in which the system is started and immediately shut down.

Since each scenario is a precise description of the sequence of user inputs that trigger actions of the system, every execution of a scenario yields the same execution profile unless the system is nondeterministic. In case of nondeterminism, one could either unite the profiles of all executions of the same scenario or differentiate each scenario execution. The latter is useful to identify differences due to nondeterminism.

D. Interpretation of Concept Lattice

In this process step, a concept lattice for the relation table created by process step 3 is built. The goals of interpreting the resulting concept lattices are:
1. Identification of the relationships between scenarios and computational units (process steps 4.1–4.3)
2. Identification of the relationships between scenarios and features and thus between features and computational units (process step 4.4)

Sect. III		main part	
object	o	u	computational unit
set of objects	O	U	set of computational units
all objects	$\mathcal{O}$	$\mathcal{U}$	all computational units
attribute	a	s	scenario
set of attributes	A	S	set of scenarios
all attributes	$\mathcal{A}$	$\mathcal{S}$	all scenarios
incidence relation	$\mathcal{I}$	$\mathcal{I}$	invocation table

Fig. 7. Translation from the identifiers of Sect. III and the identifiers used from here on, which instantiate formal concept analysis.

The following subsections describe how to achieve these goals. The basic process of lattice interpretation is depicted in Fig. 9.

D.1 Scenario Selection

A number of execution profiles is selected in order to set up the context. Execution profiles may be recombined to analyze various aspects of a system, where execution profiles and scenarios can be reused.

EXAMPLE. The analyst of FIG may first be interested in the two different ways to draw a circle. She would therefore select the two scenarios *Draw-circle-diameter* and *Draw-circle-radius*. When she understands the differences between these two features, she would investigate other circle operations and additionally select *Move-circle* and *Color-circle*. □

D.2 Concept Analysis

This process embodies a completely automated step that creates a concept lattice from the invocation table.

In order to derive the feature-unit map by means of concept analysis, we have to define the formal context (i.e., the objects, the attributes, and the relation) and to interpret the resulting concept lattice accordingly.

The formal context for applying concept analysis to derive the relationships between scenarios and computational units will be laid down as follows:

- Computational units will be considered objects.
- Scenarios will be considered attributes.
- A pair (computational unit u, scenario s) is in relation $\mathcal{I}$ if u is executed when s is performed.

Figure 7 shows how to map the identifiers used in the general description of concept analysis in Sect. III to the identifiers used in the specific instantiation of concept analysis within our method.

The system is used according to the set of scenarios, one at a time, and the execution profiles are recorded. Each system run yields all executed computational units for a single scenario; that is, one column of the relation table can be filled per system run. Applying all scenarios that have been selected during the process of scenario selection provides the relation table for formal concept analysis.

EXAMPLE. Figure 10 shows the concept lattice for the invocation table in Fig. 3, where all scenarios have been selected. □

D.3 Basic Interpretation

Concept analysis applied to the formal context described in the last section yields a lattice from which interesting relationships can be derived. These relationships can be fully automatically derived and presented to the analyst. Thus, the analyst has to know how to interpret the derived relationships, but does not need to be familiar with the theoretical background of lattices.

The following base relationships can be derived from the sparse representation of the lattice (note the duality):

- A computational unit u is required for all scenarios at and above $\gamma(u)$ in the lattice; for instance, *SetDiameter* is required for *Draw-circle-diameter*, *Move-circle*, and *Color-circle* according to Fig. 10.
- A scenario s requires all computational units at and below $\mu(s)$ in the lattice; for instance, *Color-circle* requires *color*, *setDiameter*, and *draw* according to Fig. 10.
- A computational unit u is specific to exactly one scenario s if s is the only scenario on all paths from $\gamma(u)$ to the top element; for instance, *color* is specific to *Color-circle* according to Fig. 10.
- Scenarios to which two computational units u_1 and u_2 jointly contribute can be identified by the supremum $\gamma(u_1) \sqcup \gamma(u_2)$. In the lattice, the supremum is the closest common node toward the top element starting at the nodes to which u_1 and u_2 are attached. All scenarios at and above this common node are those jointly implemented by u_1 and u_2. For instance, *setDiameter* and *color* jointly contribute to *Color-circle* according to Fig. 10.
- Computational units jointly required for two scenarios s_1 and s_2 are described by the infimum $\mu(s_1) \sqcap \mu(s_2)$. In the lattice, the infimum is the closest common node toward the bottom element starting at the nodes to which s_1 and s_2 are attached. All computational units at and below this common node are those jointly required for s_1 and s_2. For instance, *setDiameter* and *draw* are jointly required for *Move-circle* and *Color-circle* according to Fig. 10.
- Computational units required for all scenarios can be found at the bottom element; for instance, *draw* is required for all scenarios according to Fig. 10.
- Scenarios that require all computational units can be found at the top element. In Fig. 10, there is no such scenario.

Beyond these relationships between computational units and scenarios, further useful aspects between scenarios on one hand and between computational units on the other hand may be derived:

- If $\gamma(u_1) < \gamma(u_2)$ holds for two computational units u_1 and u_2, then computational unit u_2 is more specific with respect to the given scenarios than computational unit u_1 because u_1 contributes not just to the features for which u_2 contributes, but also to other features. For instance, *color* is more specific to *Color-circle* than *setDiameter* and *setDiameter* is more specific than *draw* according to Fig. 10.
- If $\mu(s_1) < \mu(s_2)$ holds for two scenarios s_1 and s_2, then scenario s_2 is based on scenario s_1 because if s_2 is executed, all computational units in the extent of $\mu(s_1)$ need also to be executed. For instance, *Move-circle* and *Color-circle*

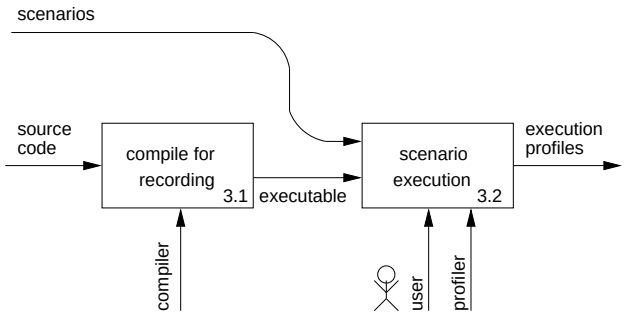


Fig. 8. The process for the dynamic analysis in Fig. 6.

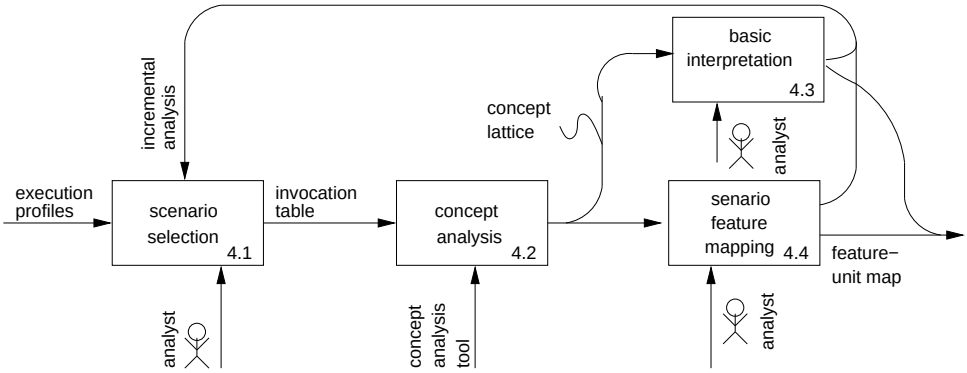


Fig. 9. The process for interpretation of concept lattice in Fig. 6.

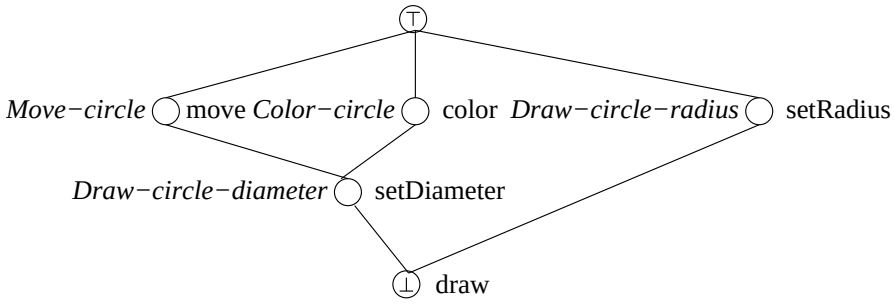


Fig. 10. Sparse concept lattice for Fig. 3.

are based on *Draw-circle-diameter* according to Fig. 10.

Thus the lattice also reflects the level of application specificity of computational units. The information described above can be derived by a tool and fed back to the analyst. Inspecting the relationships derived from the concept lattice, a decision may be made to analyze only a subset of the original features in depth due to the additional dependencies that concept analysis reveals. All computational units required for these features (easily derived from the concept lattice) form a starting point for further static analyses to validate the identified computational units and to identify further computational units that were possibly not executed during dynamic analysis because of limitations in the design of the scenarios.

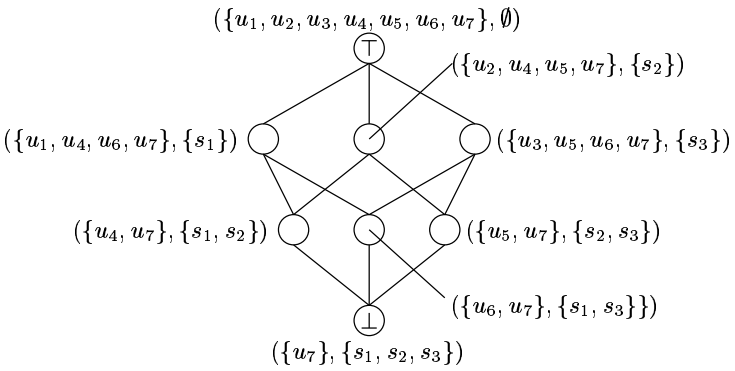
D.4 Scenario Feature Mapping

The interpretation of the concept lattice as described above gives insights into the relationship between scenarios S and computational units U. However, the analyst is primarily interested in the relationship between features F and computational units U. This section describes how to identify this relationship in the concept lattice if there is no one-to-one correspondence between scenarios and features.

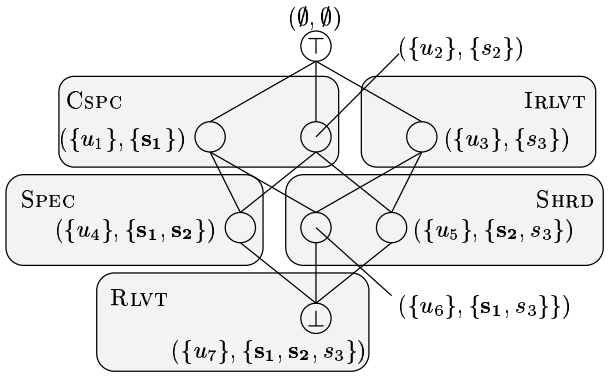
Because one feature can be invoked by many scenarios and one scenario can invoke several features, there is not always a strict correspondence between features and scenarios. For instance, as discussed above, the scenarios *Move-circle* and *Color-circle* of FIG are based on *Draw-circle-diameter* according to Fig. 10 because in order to move or color a shape, one has to draw it first. The scenario for moving or coloring a shape will thus necessarily invoke the feature which draws a shape. Fortunately, there

	f_1	f_2	f_3	u_1	u_2	u_3	u_4	u_5	u_6	u_7
s_1	×		×	×			×		×	×
s_2	×	×			×		×	×		×
s_3		×	×			×		×	×	×

(a) Invocation relation $\mathcal{I}$.



(b) Concept lattice for context in Fig. 11(a)



(c) Sparse concept lattice of Fig. 11(b) categorized with respect to feature f_1 that has been exposed in scenarios s_1 and s_2.

Fig. 11. Categorizing concept lattices.

is still a simple way to identify computational units relevant to the actual features in the concept lattice, although an unambiguous identification may require additional discriminating scenarios. The basic idea is to isolate features in the concept lattice through combinations of overlapping scenarios.

If a scenario invokes several features, one can formally model a scenario as a set of features $s = \{f_1, f_2, \ldots, f_m\}$, where $f_n \in F$ for $1 \leq n \leq m$ (F is the set of all relevant features). This modeling is simplifying because it abstracts from the exact order and frequency of feature invocations in a scenario. On the other hand, if the order or frequency of feature invocations do count, the scenarios may indeed be considered complex features in their own right. If these scenarios yield different execution profiles, they will appear in different concepts in the lattice and their commonalities and differences are revealed and may be analyzed.

With the domain expert's additional knowledge of which features are invoked by a scenario we can identify the computational units relevant to a certain feature. Let us consider the invocation relation $\mathcal{I}$ in Fig. 11(a) (for better legibility, scenarios are listed as rows and computational units as listed as columns). The table contains the called computational units $u_1, \ldots, u_7$ per scenario, and furthermore the invoked features per scenario: $s_1 = \{f_1, f_3\}$, $s_2 = \{f_1, f_2\}$, and $s_3 = \{f_2, f_3\}$. The corresponding concept lattice for the invocation relation in Fig. 11(a) is shown in Fig. 11(b). The feature part of the table is ignored while constructing this lattice.

Computational units specific to feature f_1 can be found in the intersection of the executed computational units of the two scenarios s_1 and s_2 because f_1 is invoked for s_1 and s_2. The intersection of the computational units executed for s_1 and s_2 can be identified as the extent of the infimum of the concepts associated with s_1 and s_2: $\mu(s_1) \sqcap \mu(s_2) = (\{s_1, s_2\}, \{u_4, u_7\})$. Since s_1 and s_2 do not share any other feature, the computational units particularly relevant to f_1 are u_4 and u_7.

We notice that u_7 is also used in all other scenarios, so that one cannot consider u_7 a specific computational unit for any of f_1, f_2, or f_3. Computational unit u_4, in contrast, is only used in scenarios executing f_1. We therefore state the hypothesis that u_4 is specific to f_1 whereas u_7 is not. Because there is no other scenario containing f_1 other than s_1 and s_2, computational unit u_4 is the only computational unit specific to f_1.

Note that this is just a hypothesis because other features might be involved to which u_4 is truly specific and that are not explicitly listed in the scenarios. Another explanation could be that, by accident, u_4 is executed both for f_2 (in s_2) and f_3 (in s_1); then, it appears in both scenarios but nevertheless is not specific to f_1. However, chances are high that u_4 is specific to f_1 because u_4 is not executed when f_2 and f_3 are jointly invoked in s_3, which suggests that u_4 at least comes into play only when f_1 interacts with f_2 or f_3. At any rate, the categorization is hypothetic and needs to be validated by the analyst.

Computational units that are somehow related to but not specific for f_1 are such computational units that are executed for scenarios invoking f_1 amongst other features. In our example, both s_1 and s_2 invoke f_1. Computational units in extents of concepts which contain s_1 or s_2 are there-

fore potentially relevant to f_1. In our example, u_1, u_2, u_5, and u_6 are potentially relevant in addition to u_4 and u_7. Computational unit u_3 is only executed for scenario s_3, which does not contain f_1.

Altogether, we can identify five categories for computational units with regard to feature f_1 (see Fig. 11(c)):
SPEC: u_4 is *specific* to f_1 because it is used in all scenarios invoking f_1 but not in other scenarios.
RLVT: u_7 is *relevant* to f_1 because u_7 is used in all scenarios invoking f_1; but it is also more general than u_4 because u_7 is also used in scenarios not invoking f_1 at all.
CSPC: u_1 and u_2 are only executed in scenarios invoking f_1. They are less specific than u_4 because they are not used in all scenarios that invoke f_1; that is, these computational units are only *conditionally specific*. Whether u_1 and u_2 are more or less specific than u_7 is not decidable based on the concept lattice. On one hand, they are used in all scenarios invoking f_1 and other scenarios, whereas u_7 is also executed in scenarios that do not require f_1. On the other hand, u_7 is executed whenever f_1 is required, whereas u_1 and u_2 are not executed in some scenarios that do require f_1.
SHRD: u_5 and u_6 are executed in scenarios *invoking* f_1 but they are also executed in scenarios *not invoking* f_1; that is, they are *shared* with other features. These computational units are presumably less relevant than u_1 and u_2, which are executed only when f_1 is invoked, and also less relevant than u_7, which is executed in all scenarios invoking f_1.
IRLVT: u_3 is *irrelevant* to f_1 because u_3 is only executed in scenarios not containing f_1.

These facts are more obvious in the sparse representation of the lattice. Using this representation, given a feature f, one identifies the concept, c_f, for which the following condition holds:

$$c_f = (U, S) \text{ and } \bigcap_{s_j \in S} s_j = \{f\} \qquad (10)$$

Concept c_f is called a **feature-specific concept** for f. Based on the feature-specific concept, one can categorize the computational units as follows:
SPEC: all computational units u for which $\gamma(u) = c$ holds.
RLVT: all computational units u for which $\gamma(u) = c'$ and $c' < c$ holds.
CSPC: all computational units u for which $\gamma(u) = c'$ and $c < c'$ holds.
SHRD: all computational units u for which u is in the intent of concept c' where $c < c'$ holds and c and $\gamma(u)$ are incomparable.
IRLVT: all other computational units not categorized by other categories.

When the distance between c and c' is considered, there are additional nuances within categories RLVT, CSPC, and SHRD possible. The distance measures the size of the set of features a computational unit is potentially relevant for. The larger the set, the less specific the computational unit is.

EXAMPLE. The scenario *Move-circle* in Fig. 2 invokes two features: the ability of FIG to draw a circle by diameter and the ability to move this circle. The scenario *Color-circle* also uses the ability to draw a circle; yet, it colors the circle instead of moving it. Hence, the computational units responsible for drawing a circle are attached to the concept in Fig. 10 that represents the intersection of the features invoked by *Move-circle* and *Color-circle*. The scenario *Draw-circle-diameter* would not necessarily have been required to identify the computational units for drawing a circle by diameter: The sparse lattice reveals these computational units as the direct infimum of *Move-circle* and *color-circle* even if *Draw-circle-diameter* is not considered. However, *Draw-circle-diameter* is useful to separate *draw* from *setDiameter*. □

As a matter of fact, there could be several concepts for which condition (10) holds when different computational units are executed for the given feature, depending on the scenario contexts in which the feature is embedded. For instance, let us assume we are analyzing FIG's undo capabilities. Three scenarios can be provided to explore this feature:

- Draw a circle: {draw-circle}
- Undo circle drawing: {draw-circle, undo}
- Undo without preceding drawing operation: {undo}

For the overlapping scenarios {draw-circle, undo} and {undo}, we may assume that different computational units will be executed beyond those that are specific to command *draw-circle*: Quite likely, additional computational units will be executed to handle the erroneous attempt to call *undo* without previous operation. Consequently, the lattice will contain an own concept for {draw-circle, undo} and another one for {undo}, where the latter is not a subconcept of the former. The infimum of these two scenarios will contain the computational units of the undo operation executed for normal as well as exceptional execution, whereas the concept representing {undo} contains the computational units for error handling.

In case of multiple concepts for which condition (10) holds, we can unite the computational units that are in SPEC with respect to these concepts. If the identified concepts are in a subconcept relation to each other, the superconcept represents a strict extension of the behavior of the feature. If the concepts are incomparable, these concepts represent varying context-dependent behavior of the feature.

If there is no concept for which condition (10) holds, one needs additional scenarios that factor out feature f. For instance, in order to isolate feature f_1 in scenario $s_1 = \{f_1, f_3\}$, one can simply add a new scenario $s_2 = \{f_1, f_2\}$. The computational units specific to f_1 will be in $\mu(s_1) \sqcap \mu(s_2)$.

It is not necessary to consider all possible feature combinations in order to isolate features in the lattice. Intersecting all currently available scenarios exactly tells which features are not yet isolated (the intersection could be done by concept analysis applied to the formal context consisting of scenarios and features, where the incidence relation describes which feature is invoked by which scenario). Slightly modified variants of scenarios invoking the feature can be added to isolate the feature specifically.

The addition of new scenarios in order to discriminate features in the lattice will lead us to an incremental construction of the concept lattice described in Sect. IV-F. Before we come to that, we describe the static dependency analysis.

E. Static Dependency Analysis

From the concept lattice, we can easily derive all computational units executed for any set of relevant features. However, this gives us only a set of computational units, but it is not clear which of these computational units are truly feature-specific and which of them are rather general-purpose computational units used as building blocks for other computational units. Given a feature f of interest, this question can be answered as follows:

- As a first approximation, all computational units in the extents of all feature-specific concepts for f jointly contribute to f.
- The analyst refines this approximation by adding and removing computational units: By inspecting the static dependency graph and the source code of the computational units, she sorts out irrelevant computational units; she may also add feature-relevant computational units that were not executed due to an incomplete input coverage of the scenarios. The concept lattice is an important guidance for the analyst's inspection of the dependency graph.

EXAMPLE. For FIG's ability to color a circle, the analyst will need to validate the set of computational units $\{color, setDiameter, draw\}$ according to the concept lattice in Fig. 10. The lattice shows that the analyst should start with inspecting *color* because this appears as the most specific computational unit for coloring a circle. □

E.1 Building the Starting Set

All computational units in the extent of a concept jointly contribute to all features in the intent of the concept, which immediately follows from the definition of a concept. However, there may also be computational units in the extent that contribute to other features as well, so that they are not specific to the given feature. There may be computational units in the extent that do not contain any feature-specific code at all. Thus, computational units in the extent of the concept need to be inspected manually. Because there are no reliable criteria known that automatically distinguish feature-specific code from general-purpose code, this analysis cannot be automated and human expertise is necessary. However, the concept lattice may narrow the candidates for manual inspection.

The concept lattice and the dependency graph can help to decide in which order the computational units are to be inspected such that the effort for manual inspection can be reduced to a minimum. Since we are interested in computational units most specific to a feature f, we start at those computational units u_i that are attached to a feature-specific concept of f, that is, for which $c_f = \gamma(u_i)$ holds, where c_f is a feature-specific concept for f. If there are no such computational units, we collect all computational units below any of the feature-specific concepts c_f of f with minimal distance to c_f in the sparse representation. There can be more than one concept c_f, so we unite all computational units that are attached to one of these concepts. The subset of computational units identified in this step that is accepted after manual inspection is called the starting set $S_{start}(f)$.

EXAMPLE. The starting set for FIG's ability to color a circle, $S_{start}(\textit{color-circle})$, is $\{color\}$. □

E.2 Inspection of the Static Dependency Graph

Next, we inspect the executable static dependency graph (as one specific subset of the static dependency graph) that contains all transitive control-flow successors and predecessors of computational units in $S_{start}(f)$. We concentrate on computational units here because they are the active constituents and because they were subject to the dynamic analysis. The executable static dependency graph can be annotated with the features and scenarios for which the computational units were executed. If a computational unit is not annotated with any scenario, the computational unit was not executed. Non-executable parts of the system, namely, declarative parts, may be added once all relevant computational units have been identified. A static points-to analysis is needed to resolve dynamic binding and calls via routine pointers if present. The static points-to analysis may take advantage of the knowledge about actually executed computational units yielded by the dynamic analysis.

We primarily consider only those computational units u_i for which $u_i \in \textit{extent}(c_f)$ holds because only those computational units are actually executed when f is invoked according to the dynamic analysis. Hence, we combine static and dynamic information to eliminate conditional static computational units executions in order to reduce the search space. Nevertheless, one should check for the reasons why certain computational units have not been executed.

Any kind of traversal of the executable static dependency graph is possible, but a depth-first search along the control-flow is most suited because a computational unit can only be understood if all its executed computational units are understood. In a breadth-first search, a human would have to cope with continuous context switches. The goal of the inspection is to sort out computational units that do not belong to the feature in a narrow sense because they do not contain feature-specific code.

The executable static dependency graph rather than the concept lattice is traversed for inspection because the lattice does not really reflect the control-flow dependencies: $\gamma(u_1) > \gamma(u_2)$ does not imply that u_1 is a control-flow predecessor of u_2. However, the concept lattice may still provide useful information for the inspection. In Section IV-D, we made the observation that the lower a concept $\gamma(u)$ is in the lattice, the more general computational unit u is because it serves more features—and vice versa. Thus, the concept lattice gives us insight into the level of abstraction of a computational unit and, therefore, contributes to the degree of confidence that a specific computational unit

contains feature-specific code.

EXAMPLE. The analyst would first validate the starting set for FIG's ability to color a circle $S_{start}(\textit{color-circle}) = \{\textit{color}\}$. Then she would inspect the control-flow predecessors and successors of *color*. Some of them might not be executed, yet a brief check is still necessary to make sure that they are indeed irrelevant. Then, she would continue with *setDiameter* and eventually inspect *draw*. □

Two additional analyses gather further information useful while navigating on the dependency graph:

- Strongly connected component analysis is used to identify cycles in the dependency graph: If there is one computational unit in a cycle that contains feature-specific code, all computational units of the cycle are related to the feature because of the cyclic dependency.
- Dominance analysis is used to identify computational units that are local to other computational units. A computational unit u_1 **dominates** another computational unit u_2 if every path in the dependency graph from its root to u_2 contains u_1. In other words, u_2 can only be reached by way of u_1. If a computational unit u is found to be feature-specific, then all its dominators are also relevant to the feature, because they need to be executed in order for u to be executed. If none of a dominator's dominatees contains feature-specific code and the dominator itself is not feature-specific, then the dominator is a clear cutting point as all its dominatees are local to it. Consequently, the dominator and all its dominatees can be omitted while understanding the system.

If more than one feature is relevant, one simply unites the starting sets for each feature and then follows the same approach. For more than one feature, the concept lattice identifies computational units jointly and distinctly used by those features.

Once all relevant computational units have been identified, other static (e.g., program slicing) as well as dynamic analyses (e.g., trace recording to obtain the order of execution) can be applied to obtain further information. These analyses can be performed more goal-oriented by leveraging the retrieved feature-unit map.

F. Incremental Analysis

There are at least two reasons why an incremental consideration of scenarios is desirable. First, one might not get the suite of scenarios sufficiently discriminating the first time. New scenarios become necessary to further differentiate scenarios into features. Second, new scenarios are useful when trying to understand an unfamiliar system incrementally. One starts with a small set of relevant scenarios to locate and understand a fundamental set of features by providing a small and manageable overview lattice. Then, one successively increments the set of considered scenarios to widen the understanding.

Adding scenarios means adding attributes to the formal context; but there are also situations in which objects are added incrementally: in cases where computational units need to be refined. For instance, computational units with low cohesion—that is, computational units with multiple, yet different functions—will "sink" in the concept lattice if they contribute to many features. A routine containing a very large switch statement where only one branch is actually executed for each feature is a typical example. If the analyst encounters such a routine during static analysis, she could lower the level of granularity for computational units specifically for this routine to basic blocks. Basic blocks as computational units disentangle the interleaved code: For the example routine with the large switch statement, the individual switch branches would be more clearly assigned to the respective feature in the concept lattice.

In this section, we describe an incremental consideration of attributes, namely, scenarios. Incremental consideration of objects—that is, refinement of computational units—is analogous.

As soon as one understands the basics of a system, one adds new scenarios for further detailed investigation and exploration of the unknown portions of the system. If one tries to capture all features of a software at once, the resulting lattice may become too large, too detailed, and thus unmanageable. If one starts with a smaller set of scenarios and further increases this set, all accumulated knowledge an analyst gained while working with the smaller lattice has to be preserved. The lattice—the mental map for the analyst's understanding—changes when new scenarios are added. Fortunately, the smaller lattice can be mapped to the larger one (the smaller lattice is the result of a so-called *subcontext*).

DEFINITION. Let $C = (O, A, \mathcal{I})$ a context, $O' \subseteq O$, and $A' \subseteq A$. Then $C' = (O', A', \mathcal{I} \cap (O' \times A'))$ is called a **subcontext** of C and C is called a **supercontext** of C'. □

In our application of concept analysis, we only add new rows (one for each new scenario, assuming that scenarios occur in rows of the relation table) but never new columns to the relation table (because we statically know all computational units in advance). Adding new rows leads to a new formal context $(U, S', \mathcal{I}')$ in which relation $\mathcal{I}'$ extends relation $\mathcal{I}$.

PROPOSITION. Let $C = (O, A, \mathcal{I})$ and $C' = (O, A', \mathcal{I}')$, where $A' \subseteq A$ and $\mathcal{I}' = (\mathcal{I} \cap (O \times A'))$. Then every extent of C' is an extent of C. □

PROOF. See [21]. □

According to this proposition, each extent within the subcontext will show up in the supercontext. This can be made plausible with the relation table: Added rows will never change existing rows, so the maximal rectangles forming concepts will only extend in vertical direction (if scenarios are listed in rows).

This proposition on the invariability of extents of subcontexts that only differ in the set of objects results in a simple mapping of concepts from the subcontext to the supercontext (for a formal proof see [21]):

$$(U, S) \mapsto (U, \sigma(U))$$

The mapping is a $\sqcap$-preserving embedding, meaning that

the partial order relationship is completely preserved. Consequently, the supercontext is basically a refinement of the subcontext. By this mapping all concepts of the subcontext can be found in the supercontext.

The supercontext may include new concepts not found in the subcontext. The consequence for the visualization of the supercontext is that the newly introduced concepts can be highlighted easily in the visualized lattice of the supercontext and that concepts in the subcontext can be mapped onto concepts in the superconcept along with possible user annotations. Additionally, an incremental automatic graph layout can be chosen: Only additional nodes and edges may be introduced in the supercontext, nodes and edges of the subcontext are kept. Thus, the position of concepts relatively to each other will be preserved.

EXAMPLE. Let us assume the analyst of FIG is now interested whether invoking the feature "circle drawing" twice makes a difference and what the differences between drawing a circle and drawing a dot ("Draw-dot") on one hand and between moving a circle and undoing a circle move operation ("Move-circle-undo") on the other hand are. The domain expert will design the appropriate scenarios. The resulting invocation table for these and all previous scenarios may be as in Fig. 12(a). The lattice for this new supercontext is shown in Fig. 12(b). The new scenario *Draw-circle-diameter-twice* is subsumed by the existing scenario *Draw-circle-diameter*, showing that using the feature twice does not lead to additional relevant computational units. The new scenario *Draw-dot* is subsumed by the bottom concept; thus, *Draw-dot* shares only the computational unit *draw* with the feature "circle drawing". Both scenarios *Draw-circle-diameter-twice* and *Draw-dot* do not change the general structure of the lattice. Only the concept highlighted in Fig. 12(b) is new. This concept shows the difference between *Move-circle* and *Move-circle-undo*, which is the additionally executed computational unit *undo*. □

V. CASE STUDIES

This section describes two case studies evaluating our method. The first case study on web browsers shows the benefit from combining static and dynamic information. The second case study focuses on dynamic information and exemplifies the incremental analysis for a very large commercial system.

In both case studies, the computational units of choice are routines. The Bauhaus [26] tools were used to extract the static dependency graph. The extracted static dependency graph contains all global declarations (routines, global variables, and user-defined types) and many dependencies such as calls between routines, references of global variables by routines, type information for variables, dependencies between user-defined types, occurrences of types in routine signatures, and so on [27].

For the dynamic analysis, we used a standard profiler to gather execution profiles. The profiler has the limitation that it does not record accesses to variables. We therefore analyzed variable accesses statically.

system	version	KLOC(wc)	#subprograms
Mosaic	2.6	51,440	701
Chimera	2.0a19	38,208	928

Fig. 13. Analyzed web browsers.

A. Web Browsers

In this section, we discuss the usefulness of static and dynamic informations as introduced in Sect. IV-E.

We analyzed two web browsers (both written in C; see Fig. 13) using the same set of relevant related features. The concept lattice for each of these systems was derived as described in Sect. IV. The required routines as identified by dynamic analysis and the relationships derived by concept analysis formed a starting point for the static dependency analysis.

A.1 Case Study Setup

In two experiments, we tried to understand how two specific sets of related features are implemented in both browsers using the process described above. The goal of this analysis was to recover the feature-specific computational units and the way they interact—that is, to reverse engineer a partial description of the software architecture. The partial software architecture, for instance, allows one to decide whether feature-specific computational units can be extracted from one system and integrated into another system with only minor changes. Chimera does not implement all features that Mosaic provides and we wanted to find out whether the respective feature-specific computational units of Mosaic can be reused for Chimera.

- Experiment "History" (H): Chimera allows going back in the history of already visited URLs, but Chimera does not have a forward button that allows a user to move forward in the history again after the back button was used. Mosaic has both a back and a forward button. In this experiment, going back and going forward were considered related features.
- Experiment "Bookmark" (B): Both Mosaic and Chimera offer bookmarks for visited URLs. URLs may be bookmarked, and bookmarked URLs may be loaded and removed. We considered the following related features: addition of a new bookmark for a currently viewed URL, removal of a bookmark, and navigation to a bookmarked URL.

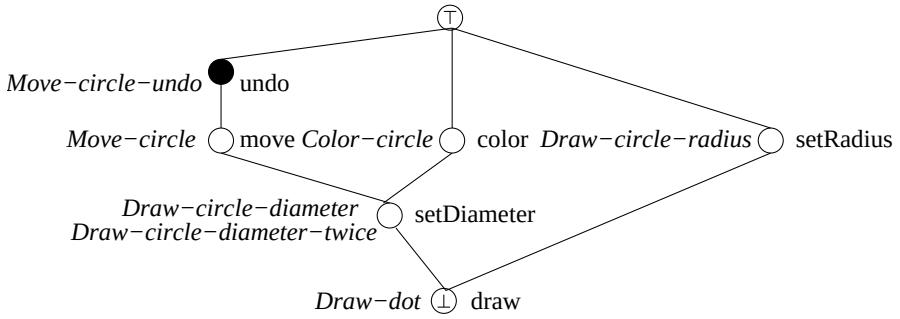
A.2 Objectives

The questions we wanted to answer in our case study are as follows:

- Identification and extraction: How are the history and the bookmark features implemented in Mosaic (Chimera)? What are the interfaces between the specific computational units that implement these features and the rest of Mosaic (Chimera)? In both cases, a partial description of the software architecture was recovered.
- Integration: How can the identified portion of the code of one browser be integrated into the other browser?

	draw	setDiameter	setRadius	move	color	undo
Draw-circle-diameter	×	×				
Draw-circle-radius	×		×			
Move-circle	×	×		×		
Color-circle	×	×			×	
Draw-circle-diameter-twice	×	×				
Move-circle-undo	×	×		×		×
Draw-dot	×					

(a) Supercontext of Fig.3.



(b) Lattice for the (super)context in Fig. 12(a)

Fig. 12. The lattice for the supercontext of Fig. 10.

The whole experiment (from initial setup of scenarios and compiling with profiler options up to the architectural sketches) took two people half a day of work altogether for Mosaic and Chimera.

A.3 Scenarios for Dynamic Analysis

For each experiment and each browser, we ran the browser in a start-end scenario in which the browser was started and immediately quit in order to separate start-up and shutdown code. The following additional scenarios were defined specifically to the two experiments. Experiment "History" was covered by the following three scenarios:

(H1) Basic scenario doing nothing but browsing
(H2) Scenario using the back button
(H3) Scenario using the back and forward buttons

For Chimera, the last scenario was not performed (because Chimera possesses no forward button).

Experiment "Bookmark" was covered by the following four scenarios:

(B1) Basic scenario: simply opening and closing the bookmark window
(B2) Scenario: adding a new bookmark for the currently displayed URL
(B3) Scenario: removing a bookmark
(B4) Scenario: selecting a bookmark and visiting the associated URL

Each scenario was immediately ended by quitting the respective system. We provided scenarios that invoke one feature only except for one scenario: One cannot use the forward button without using the back button. Consequently, the concept containing routines executed for scenario (H2) is a subconcept of the concept related to (H3). Likewise, a bookmark can only be deleted when a URL has been added before. To circumvent this problem, we started the browser with a non-empty bookmark file in all scenarios. Thus, we did not consider the case of insertion into an empty bookmark list.

	(1)	(2)	(3)	(2) ∩ (3)	relevant
Mosaic/(B)	701	359	99	74	16
Mosaic/(H)		348	74	65	6
Chimera/(B)	928	431	89	55	3
Chimera/(H)		419	123	55	24

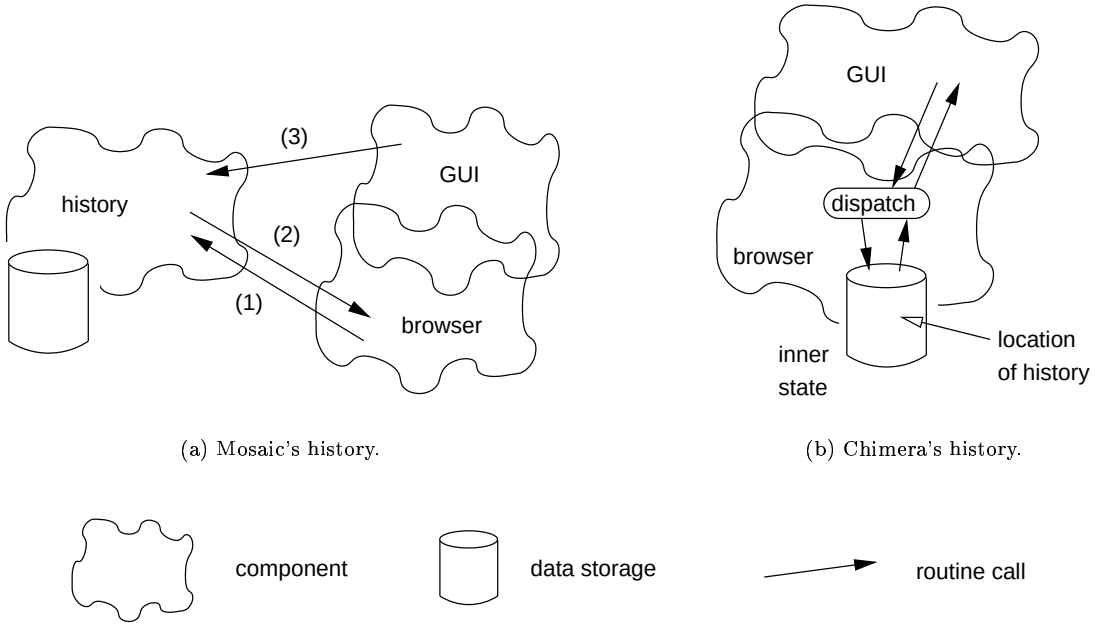
Fig. 14. Subprogram counts for Mosaic and Chimera.

A.4 Static Dependency Analysis

In the dependency graph for the browsers, visualized using the Bauhaus extension to Rigi [28], we derived all statically transitively called routines (using Rigi's basic selection facilities [28]) and intersected the static information with the actually executed routines manually. We additionally filtered out all routines specific to HTML and the X-window-based graphical user interface guided by the browser's proper naming conventions. These routines were all in the bottom element of the concept lattice.

A.5 Results

Figure 14 provides a summary of the numbers of routines that needed to be further considered in each step and shows how the search space could be reduced in each step.



(a) Mosaic's history.

(b) Chimera's history.

Fig. 16. Mosaic's and Chimera's history architecture.

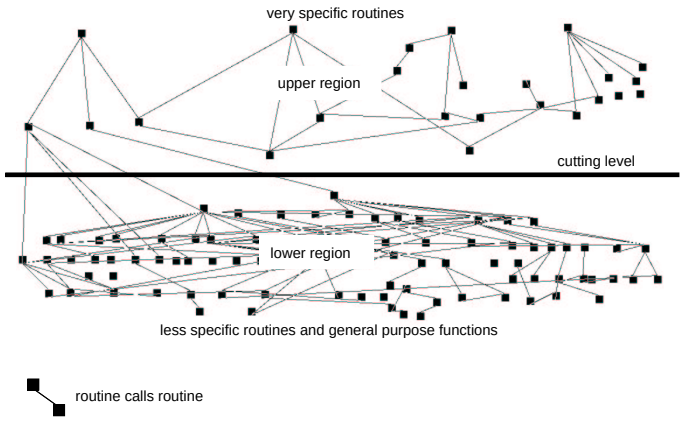


Fig. 15. Relevant parts of Chimera for history.

The history experiment is denoted by (H) and the bookmark experiment is denoted by (B). The total number of all routines of the kernels (not including libraries such as html, jpeg, zlib) is in column (1), the number of actually executed routines for any of the scenarios is shown in column (2). All routines *statically* called by routines selected from the set of dynamically executed routines in upper concepts of the lattice (i.e., called from routines in the starting set) are in column (3). The intersection of column (2) and (3) contains all routines *dynamically* called by routines selected from the set of dynamically executed routines in upper concepts of the lattice; their number is reported in column "(2) ∩ (3)". Column *relevant* reports all routines in column (2) ∩ (3) that are specific to the selected features according to our manual inspection. All other routines are used for other purposes than bookmarks and histories.

Eventually, only a small number of routines needed to be inspected more thoroughly due to the top-down inspection process. As an example, Fig. 15 shows the remaining routines of Chimera (omitting their names) relevant to the history experiment. This picture clearly shows the possible cutting points in the dependency graph (consisting of routines, global variables, and user-defined types and their dependencies) of routines specific to the history features (upper region) and non-specific routines (lower region): Only two entities need to be removed to isolate feature-specific from non-specific entities.

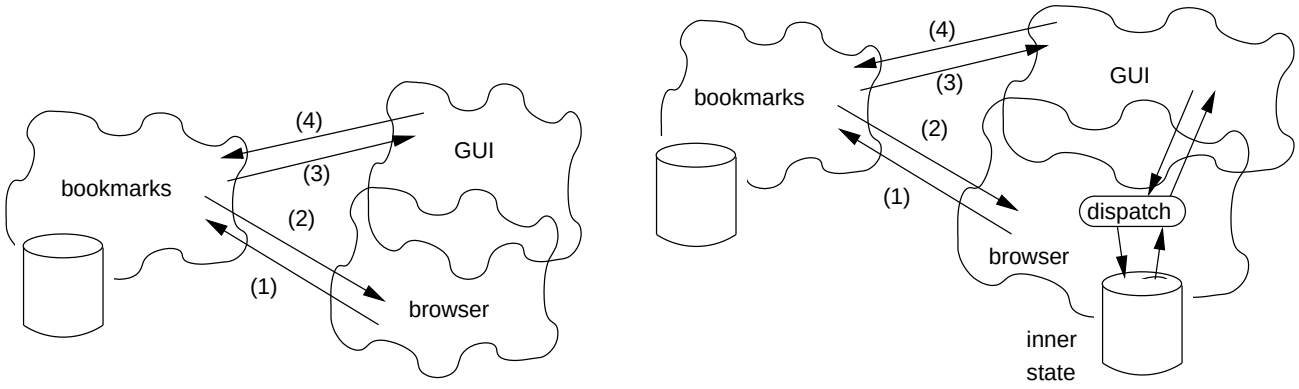
We recovered the parts of the architecture of Mosaic and Chimera relevant to the two experiments.

A.6 Results for History

The interface between Mosaic's browser kernel and the history component (see Fig. 16(a)) is formed by three routines to (1) get the current URL, (2) set the current URL, and (3) communicate the action and event (changed URL).

The history component can be easily extracted from Mosaic's source code because it is a separate component—whereas the history is an integral part of Chimera's kernel (cf. Fig. 16(b)). There is no set of routines of Chimera that could be reasonably addressed as "history manager component" as in Mosaic. Chimera uses a layer of wrappers calling a dispatching routine around a list of actions where the displayed URLs are part of that list.

The recovered partial architecture shows that Chimera's browser kernel is built around a list of visited URLs whereas Mosaic's browser kernel does not know the history of visited URLs at all. As the analysis of the partial architectural architectures reveals, re-using Mosaic's history components in Chimera would be very difficult due to the architectural mismatch [29].



(a) Mosaic's bookmarks.

(b) Chimera's bookmarks.

Fig. 17. Mosaic's and Chimera's bookmark architecture.

A.7 Results for Bookmarks

The partial architectures of the two systems are similar to each other with respect to bookmarks. Both architectures include an encapsulated bookmark component, which communicates via a narrow interface with the basic browser kernel (see Fig. 17).

The basic actions that have to be performed are: (1) get currently shown URL, (2) set currently shown URL, (3) display the bookmarks, and (4) communicate the bookmark selection back.

Exchanging the two implementations between Mosaic and Chimera would be reasonably easy.

B. Case Study Agilent

This section reports on a case study conducted to investigate the usefulness of the approach in a realistic full-scale industrial setting. The case study stresses the importance of incremental understanding of very large concept lattices as described in Section IV-F and the modeling of scenarios as set of features as explained in Section IV-D.4.

The system analyzed is part of the software of the Agilent 93000 SOC Series, a semi-conductor test equipment produced by Agilent Technologies.

B.1 Agilent 93000 SOC Series

The Agilent 93000 SOC Series is a single scalable tester platform used in the manufacturing process of integrated circuits. It provides test capabilities for digital, analog, and radio frequency circuits as well as for embedded memories. The *SmarTest* software controls the complex tester hardware. It is an interactive environment for developing and running test programs.

SmarTest consists of numerous tools supporting test engineering tasks. At the center of the software lies the *firmware*, an interpreter for IEEE-488-like commands. The firmware is responsible for programming the hardware. The input to the firmware are the test cases, which are sequences of firmware commands. The firmware parses and interprets each command, drives the Agilent 93000 device, and returns the result. It is the firmware that was analyzed in our case study.

The software of the Agilent 93000 SOC series is maintained by several geographically distributed groups. Two of them are situated in the USA, one in Japan, and one in Germany. The group in which the case study was conducted is the *SOC Test Platform Division* at Böblingen, Germany.

The firmware of the Agilent 93000 has evolved over 15 years. Today, it consists of 1.2 million commented lines of C code—counted with the Unix program *wc*—or about 500.000 non-empty lines of declarative or executable C code, respectively. The static call graph of the part of the firmware that was analyzed for this case study had 9.988 routines and 17.353 call edges excluding standard C routines and operating system routines.

Figure 18 depicts the software architecture of the firmware as described by one of the software architects at Agilent. The firmware is used simultaneously by different tools running as separate processes. Interaction between these tools and the firmware is through shared memory and message queues as part of the firmware. A semaphore is used to synchronize interaction between firmware and other tools.

The firmware is basically an interpreter for test programs. When a test program is filed into the shared memory, the firmware parses and runs each command. In order to run a command, the firmware dispatches the corresponding C routine that acts as an entry point to the implementation of the command. There is one such C routine—also referred to as *executor*—for each command. When the executor has finished, its result is written back to the shared memory and the waiting process is informed through the message queue. As Fig. 18 suggests, the executors share a set of re-usable utility routines—routines offering more general services. Which utility routines are actually shared by which executors is, however, not shown in the architectural sketch. As a matter of fact, the software architect currently does not exactly know what the precise relation

between executors and utility routines is due to the size of the system and the lack of documentation.

Many commands interpreted by the firmware come in pairs: the actual command and an additional command to fetch the result of its execution. The latter is called the query command. The commands are named by four-letter acronyms. Query commands are additionally annotated with a question mark. For instance, `CNTR?` is the query command of `CNTR`.

The firmware understands about 250 different actual commands; most of them have a corresponding query command. Altogether, there are about 450 different commands.

For this case study, we focused on the digital part of the firmware, namely on *Configuration Setup*, *Relay Control*, *Level Setup*, *Timing Setup*, and *Vector Setup* commands (other classes of commands are *Analog Setup*, *AC Test Function*, *DC Measurement*, *Test Result*, *Utility Line*, and *Calibration and Attributes* commands):

Configuration Setup Commands: Configuring pins is the first step one must take when preparing a test. Commands of this class allow assigning pin names to a test or power supply channel, configuring pin type and operation modes, specifying the series resistor, and other things.

Routing Setup Commands: The Routing Setup commands specify the signal mode and connection for each pin, and the order of connections.

Level Setup Commands: The Level Setup commands specify the required driver amplifier and receiver comparator voltage levels, as well as set termination via the active load or set the clamp voltage.

Timing Setup Commands: The Timing Setup commands define the length of the device cycle, the shape of the waveforms making up a device cycle, and the position of the timing edges in a tester cycle for all configured pins.

Vector Setup Commands: The Vector Setup commands are required to set up and sequence test vectors.

Relays Control Commands: The Relay Control commands are used to set relay positions and the tester state.

B.2 Objectives

This case study had three goals:

1. The architectural sketch in Fig. 18 had to be mapped to the source code so that the parts of the system that contribute to the blocks "executors" and "utility functions" are identified. It had to be clarified which routines are executors.
2. The utility routines were to be assigned to the executors they support. This mapping clarifies the fine structure of the "utility functions" block in Fig. 18.
3. Some commands of the Agilent 93000 firmware we investigated were not assigned to one of the classes of *Configuration Setup*, *Relay Control*, *Level Setup*, *Timing Setup*, or *Vector Setup* commands, neither by the architect nor by the user manual. These were to be classified according to the resulting concept lattice to see whether the lattice provides useful information to classify features.

The overall goal of our case study was to map the architecture sketch in Fig. 18 to the source and to show which utility routines are really shared. Given the above mentioned classes of commands, our hypothesis was that the executors for commands of the same class share many utility routines. On the other hand, for commands of different classes, we expected less commonalities, in other words, one would expect that only more general utility routines are shared.

B.3 Scenarios for the firmware of Agilent 93000

The software architect at Agilent selected the commands for digital tests that were to be investigated. Three students of the University of Stuttgart created the test cases—advised by the expert. For each relevant firmware command, a test case was provided that executes the command.

The execution of some commands is bound to certain preconditions that need to be fulfilled by calling other commands first, which requires to add these commands to the test cases. Hence, a test case is generally not a single command but a sequence of firmware commands, of which one is the relevant command and the others are required preparing steps. The order of preparing commands was the same for all test cases that had these commands as preconditions, and there were no two test cases executing the same set of routines. As already described in Section IV-D.4, we can thus model a test case (scenario) as set of commands (features) $s = \{command_1, command_2, \ldots, command_m\}$.

In order to identify the routines specific to the relevant command only, one can factor out preparing steps by additional test cases, which execute the preparing commands but not the relevant command. For instance, in order to call command `UDPS`, one needs to execute `DFPS` first. Thus, the test case for `UDPS` is {`DFPS`, `UDPS`} where only `UDPS` is relevant. In order to identify the routines for `UDPS` specifically, one can simply add another test case executing `DFPS` only. The routines specific to `UDPS` can then be identified in the concept lattice as described in Section IV-D.4.

If a command has a query command, two test cases were created: one for the actual command and one for the query command. The former contains only the actual command but not the query command and the latter only the query command but not the actual command (in all cases where the query command can be called without calling the actual command before).

If a command has different options, the test case executes the command with several different combinations of options. The combination is aimed at covering equivalence classes of option settings.

For one pair of an actual and a query command, namely, the command `SDSC`, four scenarios were created: two for the actual and two for the query command. The difference of the two scenarios for both the actual and the query command is the setting of the specification parameter, that either relates to Timing or Level Setup. The distinction was made to see whether the command requires routines from different parts of the system, that is, the timing setup and level setup parts.

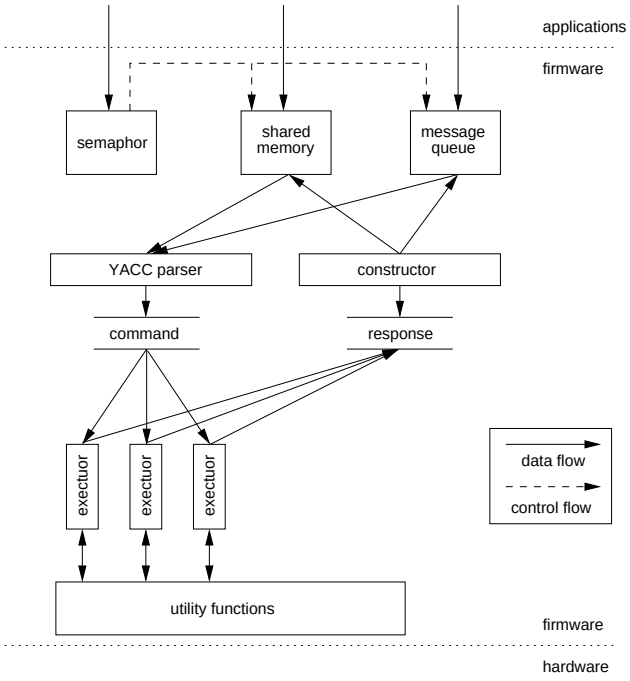


Fig. 18. Software architecture of Agilent 93000 firmware

real	76	scenarios for relevant commands
	1	scenario for `NOP` command
additional	2	additional parameter combinations
factoring	1	start-end
	13	scenarios for preparing steps
total	93	scenarios

Fig. 19. Test cases / scenarios.

Each test case represents a scenario. In total, 93 scenarios were provided (cf. Fig. 19). Among these, 76 scenarios correspond to one relevant firmware command for digital tests. One additional scenario contained just the *no-operation* (`NOP`) command, which has no effect on the tester. Two additional scenarios were added to call command `SDCS` and its query command with the alternative parameter setting. The remaining scenarios were used to refactor scenarios: The *start-end* scenario was used to remove start-up and shutdown code by simply starting the system, executing a reset command, and shutting down the system, and 13 *factoring* scenarios were provided to factor out preparing steps in real scenarios.

Agilent's own large test suite for testing the firmware could not be used since we needed scenarios that explore preferably one command (or feature, respectively) at a time. Agilent's test cases use combinations of commands. Moreover, the existing test driver of the test suite executes all tests in one run so that the result would have been a single profile for all test cases instead of an individual profile for each test case.

B.4 Resulting Concept Lattice

The resulting concept lattice is shown in Fig. 20. It consists of 165 concepts and 326 non-transitive subconcept relations. Out of the 9.988 statically declared routines, only 1.463 were actually executed by at least one of the 92 considered scenarios (the start-end scenario is used to remove those routines from the profiles of the other scenarios that are executed for initialization, reset, and shutdown of the system only).

Although, the worst case execution time to compute a concept lattice is exponential in the number of objects and attributes, our computation of the concept lattice for the firmware took less than 2 minutes on an Intel Pentium III 800 MHz machine running Linux.

Another developer at Agilent (different from the software architect who sketched the firmware architecture) was asked to validate the resulting concept lattice. To make a clear distinction between this validating expert and the expert who sketched the firmware architecture, the former will be called **developer** and the latter **software architect** in the following.

The developer was familiar with the firmware but was not involved in the preparation of the test cases. We explained the test cases that were selected and the interpretation of the concept lattice as described in this paper. We did not show the architecture sketch from the software architect. We asked the developer to explain the general structure of the system with the concept lattice and whether there are any surprises in the lattice.

The developer immediately spotted in the 65 direct subconcepts of the top element—that is, concepts in the first row below the top element of the lattice—the individual executors for 65 commands (including the executor for `NOP`).

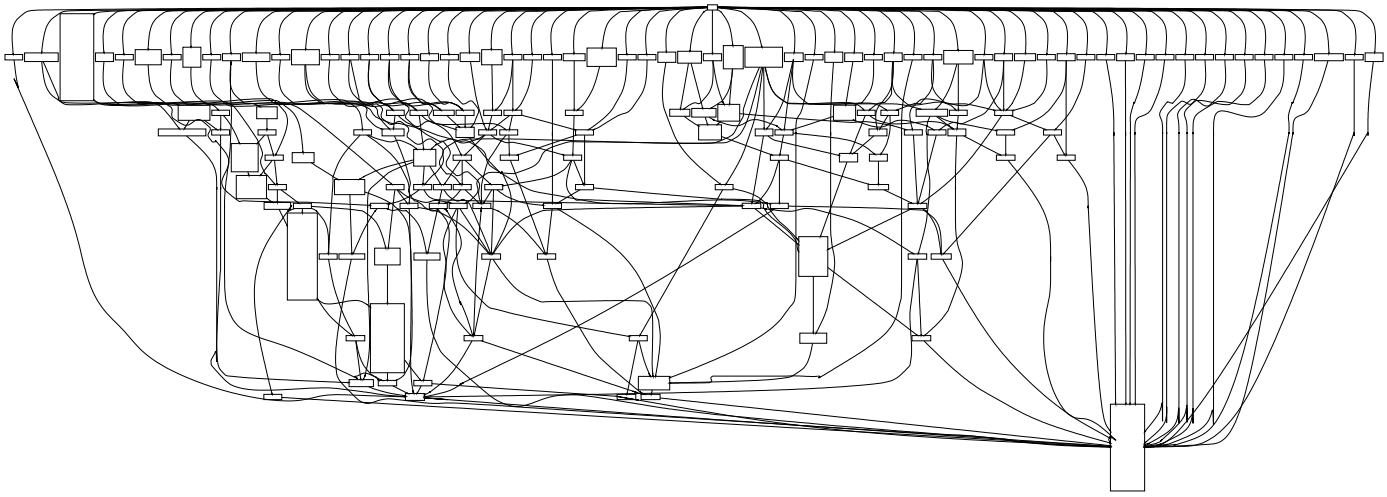


Fig. 20. The lattice for all commands. The boxes' height corresponds to the number of routines in the concepts.

(The top element itself does not contain any scenario.) Among these 65 concepts, 63 contain a single scenario and two contain two scenarios. The ones with two scenarios are the two different parameter settings for the SDSC command and the corresponding query command (cf. Sect. V-B.3). Consequently, the implementation of the SDSC command executes the same routines independently from the parameter that refers to timing or level setup, respectively. Thus, 65 executors could immediately be detected in the lattice. Based on these observations, we could easily map the concept lattice in Fig. 24 to the architecture sketch of Fig. 18.

The other 12 real scenarios can be found in subconcepts of the above mentioned 65 concepts. The reason why these scenarios cannot be found directly below the top element is that they represent commands that are also needed as preparing steps for other commands. For instance, before the commands PSLV and UDPS can be called, one must call DFPS. The scenarios for PLSV and UDPS are consequently {DFPS, PLSV} and {DFPS,UDPS}, respectively. The scenario that contains DFPS only will therefore be part of the concept that is the common infimum of the scenarios for PLSV and UDPS since {DFPS} = {DFPS,PLSV} ∩ {DFPS,UDPS}. By representing test cases (scenarios) as sets of commands (features) and isolating commands through intersecting test cases as described in Section IV-D.4, we could easily identify the executors for the remaining 12 commands whose test case is not directly located below the top element.

As described above, the firmware commands can be categorized in different classes (Configuration Setup, Relay Control, Level Setup, Timing Setup, and Vector Setup command). In order to visualize the jointly used routines by executors for commands of the same class, we colored the concept lattice as follows:

1. Each concept representing an executor in the lattice gets the color of the executor's class; the colored concept is the starting node for the traversal in the next step.
2. By top-down traversal starting at the colored concept, the color of the respective executor is propagated to all subconcepts of the executor's concept (until a different executor is reached).

The colored concept lattice for Agilent's firmware gives interesting insights. All concepts directly below the top element in Fig. 24 have just one color because these concepts actually represent just one executor of a given command. If a concept, c, has more than one color, the routines, u_i, for which $\gamma(u_i) = c$ holds contribute to commands of different classes. As a matter of fact, there were only few concepts above the bottom element with different colors showing that there is substantial sharing of routines among executors of the same class of commands. The utility routines in concepts having only one color seem to be specific to just a single class of commands. In other words, either a routine is specific to a class of commands or it is used for all command classes in general.

The dynamic analysis in conjunction with concept analysis thus has given important insight into the internal structure of the black box labeled "utility routines" in Fig. 18: 534 routines (out of 1.463 routines executed for at least one test case and 9.988 statically declared routines, respectively) could be related to the executors, that is, are not specifically attached to the bottom element.

There are also executors for commands of the same class that share only the most general routines in the bottom element, that is, those routines executed for all executors. The most remarkable example are the executors for the configuration setup of single pins on one hand and those for the configuration setup of whole pin groups. While the executors for single pins share many routines specific to their class, the executors for pin groups (which also belong to the same class *Configuration Setup*) do not share any routine beyond those in the bottom element, neither with executors for single pins nor with other executors for pin groups. Our hypothesis was that there are many routines jointly used by configuration setup commands for pin groups similarly to commands for single pins. The developer reviewing the concept lattice explained that macros are heavily used for routine inlining in the subsystem implementing pin group configuration. According to the developer, this subsystem

is an older part of the system. Apparently, at its initial development, no compiler with automatic routine inlining was available. The use of macros undermines our way to collect dynamic information. The profiler we used records only routine calls and, hence, cannot reveal code sharing among these pin group commands.

Generally, the concepts just below the top element contain only one routine. Some contain more than one routines but less than five. In these cases, a programmer apparently has split a large executor into smaller pieces for better modularization. There is one concept just below the top element that contains a very large number of routines. This concept represents the test execution. The developer explained that the routines specifically attached to this concept are strongly related but could have been further grouped if more scenarios for test execution would have been provided.

The developer also looked at another very large concept located in the middle of the concept lattice. By looking at the routines specifically attached to this concept, he told us that about 70% of these routines deal with memory management. Hence, this concept collected a large number of semantically related routines.

There are 929 routines specifically attached to the bottom element, that is, routines that are used for all scenarios. For these routines, either the selection of test cases failed to further structure this set of routines or the routines are necessarily required for all possible usage scenarios, in which case other techniques are needed to group these routines semantically. Since our goal was to identify the executors and the routines shared by the executors, we did not further investigate the routines in the bottom element.

B.5 Inferring Categorization from Concept Lattice

Prior to our analysis, the software architect selected firmware commands that were to be investigated. He also categorized the commands as described in Section V-B.3. As it turned out during our analysis of the concept lattice, the categorization was incomplete. The software architect categorized only the commands listed in Fig. 21. Additionally, he prepared scenarios that explored the commands listed in Fig. 22. The incomplete categorization gave us the opportunity to check whether it would be possible to categorize commands into the above classes just on the basis of the concept lattice without any knowledge of the system and the application domain.

One of the authors of this article guessed the categories based on the concept lattice only—more precisely, based on the sharing of utility routines with other already classified commands. The assumption was that a command belongs to the class of commands with which it shares most utility routines. Altogether 7 out of 12 commands were actually assigned to one of these classes based on this assumption. For the remaining commands, the lattice did not provide unambiguous information.

We used two oracles to validate these guesses. Firstly we asked the developer to classify these commands and

Configuration Setup
CNTR, CNTR?, CONF, CONF?
UDEF, UDPS, UDGP
DPFN, DFPN?, DFPS, DFPS?
DFGP, DFGP?, DFGE, DFGE?
PALS, PALS?, PSTE, PSTE?
PSFC, PSFC?, PQFC, PQFC?
PACT, PACT?
Relay Control (Test Execution)
RLYC, RLYC?
Level Setup Commands
LSUS, LSUS?, DRLV, DRLV?
RCLV, RCLV?, TERM, TERM?
Timing Setup Commands
PCLK, PCLK?, DCDF, DCDF?
WFDF, WFDF?, WAVE, WAVE?
ETIM, ETIM?, BWDF, BWDF?
Vector Setup Commands
SQLA, SQLB, SQLB?, SQPG, SQPG?
SPRM, SPRM?, SQSL, SQSL?

Fig. 21. Categorization of commands as found by the software architect.

Uncategorized
FTST, VBMP, PSLV, CLMP
WSDM, DCDT, CLKR, VECC
SDSC, SREC, DMAS, STML

Fig. 22. Commands not categorized by the software architect.

Guess	Developer	Manual
Relay Control (Test Execution)		
	FTST	
Level Setup Commands		
PSLV	PSLV	PSLV
CLMP	CLMP	
FTST		
	VBMP	VBMP
Timing Setup Commands		
DCDT		DCDT
CLKR	CLKR	CLKR
WSDM		
Vector Setup Commands		
VECC	VECC	VECC
	DMAS	DMAS
		SREC
Others/Multiple		
SDSC	SDSC	SDSC
DMAS		
STML	STML	STML
SREC	SREC	
VBMP		
	DCDT	
	WSDM	WSDM
		FTST
		CLMP

Fig. 23. Comparison with oracles.

secondly we checked the user manual for the firmware. The comparison of the guesses with the two oracles is shown in Fig. 23.

Interestingly enough, the classification given in the manual is also incomplete. Two of the used commands, namely, `CLMP` and `STML`, are not described in the manual. Moreover, the command `FTST` does not really belong to the targeted classes of commands according to the manual; it was added by the software architect because it is the starting command for the actual test execution. `SDSC` and `WSDM` are commands that cannot be assigned to one class of command only but rather contain aspects of different classes.

As can be seen in Fig. 23, the classification of the developer is also incomplete since he did not know all firmware commands. There are more than 250 commands, not counting the corresponding query commands. The classification of the developer is in accordance with the user manual except for `CLMP`, which is not described in the manual.

If we compare the lattice-based guesses with the oracle, we find that the author was truly wrong only once, namely, for command `FTST`. In case of command `WSDM`, he assigned a command to one class of two equally possible classes.

It was interesting to see that many commands could be assigned correctly simply based on the lattice without any knowledge of the application domain and implementation of the system.

B.6 Lessons Learnt

In the beginning of our case study, we explained the basic interpretation of the concept lattice to the developer without going into the formal mathematical details. The developer learnt how to read the concept lattice surprisingly quickly in less than 10 minutes, which suggests that the technique can easily be adopted by practitioners.

The developer confirmed that the technique could be useful for maintenance programmers who are less familiar with the system in order to quickly identify the executors. Since there was a naming convention for executors in place, locating the executors could have been done with textual search tools, such as *grep*, more easily, he noted. The developer also confirmed the general approach for the static analysis once the executors have been located: If he is to modify a command, he also traverses the dependency graph. For lack of more sophisticated tools, he is using simple tools, such as the Unix tool *ctags*, to get the necessary cross-reference information. However, the developer agreed that it would have been very difficult for him—using such simple tools—to identify the firmware commands to which a given routine contributes. Such kind of information would help him in the impact analysis of changes. Moreover, it would also have been very difficult for him to identify the sharing of utility routines among executors.

This case study also revealed some difficulties with the proposed technique. For instance, due to the use of inlining of routines by way of macros, the profiler could not identify the code sharing of commands for pin groups. For such inlining, a static analysis is necessary. In order to identify this kind of code sharing, one could try to identify joint uses of macros in the non-preprocessed code or duplicated code in the preprocessed code by way of clone detection techniques.

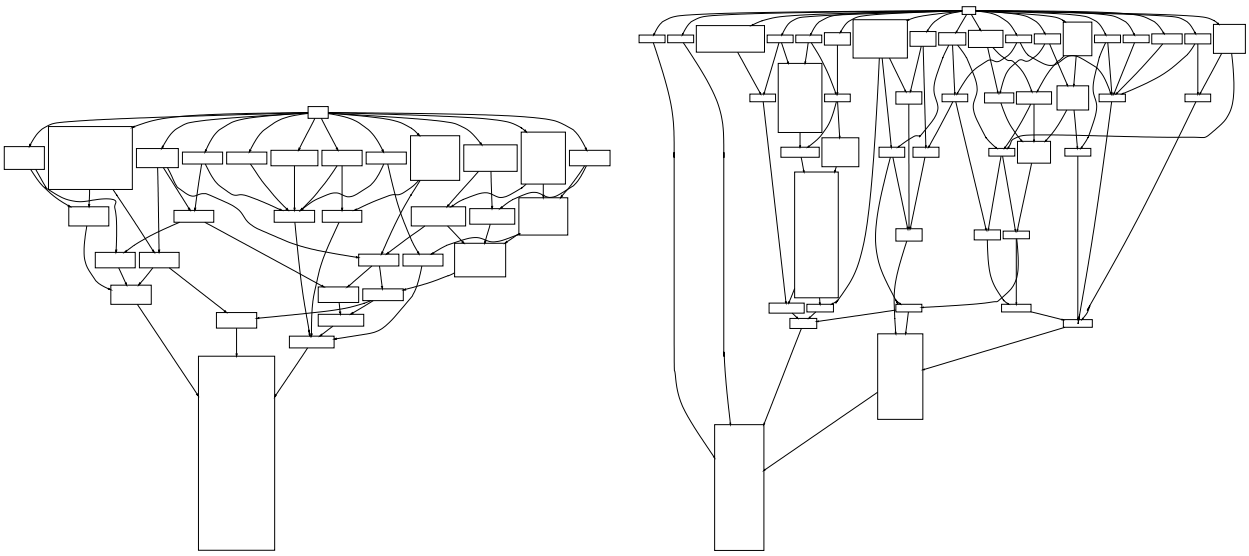
Another difficulty that had to be tackled in this case study is the problem of handling parameterized scenarios, that is, scenarios that are alike except for values of certain parameters. For instance, most commands of the firmware have options. The options, of course, influence the behavior of the system. The same command may execute different routines for different options. This problem is equivalent to the input coverage problem of testing software in general. Analogously, the test cases for the Agilent case study were defined so as to cover equivalence classes of possible parameter values. The firmware commands were then called with different combinations of representative values of equivalent parameter settings. However, full coverage of all possible combinations would exceed all available resources, and there is no guarantee that the software actually behaves equivalently for all apparently equivalent input values.

Due to the dynamic analysis, only about 15% of the almost 10,000 routines were present in the formal context for concept lattice. Likewise, the number of scenarios was realistic, yet trimmed to only the digital part of the system. Nevertheless, the concept lattice for the firmware of the Agilent 93000 chip tester—containing 165 concepts—was relatively large and complex. Such large concept lattices are a challenge for visualization. Not so much with regard to the time to produce a visualization but with the reading and understanding of such a large graph. We used GraphViz by AT&T [30] to layout the graph automatically in virtually no time. Also, the resulting layout was acceptable—at any rate, much better than we could have drawn the graph. However, we would have liked to group the nodes of the graph semantically in terms of the classes to which the associated commands belong beyond the aesthetic criterion of minimizing edge crossings. Moreover, the lattice was too large to be presented on a 21" screen. For this reason, we used a print-out of the lattice with 19 pages (DIN A4 format) for the discussion with the developer, and even on this print-out, the names of routines and scenarios were hard to read.

The experiences with size and complexity of the final lattice in the Agilent case study lead us to develop support for incremental construction and understanding of the concept lattice as described in Section IV-F. The visual difference for considering scenarios incrementally is illustrated by Fig. 24. Figure 24(a) contains the concept lattice for all Timing Setup commands. For the lattice in Fig. 24(b), all scenarios for Vector Setup have been added. When all scenarios for all classes of commands are added, the lattice in Fig. 20 is obtained.

VI. Related Research

This section discusses research related to our work. We discuss work on several aspects that are of interest. First, we take a look at papers most closely related to our own approach. Next, we summarize work that visualizes dynamic



(a) Timing commands. (b) Timing and vector commands.

Fig. 24. Concept lattice for digital part of Agilent 93000 firmware.

and static information in different ways.

Feature Location

Wilde et al. [6], [31] pioneered in locating features taking a fully dynamic approach. The goal of their *Software Reconnaissance* is the support of maintenance programmers when they modify or extend the functionality of a legacy system.

Based on the execution of test cases for a particular feature f, several sets of computational units are identified:
- computational units *commonly involved* (code executed in all test cases, regardless of f),
- computational units *potentially involved* in f (code executed in at least one test case that invokes f),
- computational units *indispensably involved* in f (code that is executed in all test cases that invoke f, and
- computational units *uniquely involved* in f (code executed exactly in cases where f is invoked)

Since the primary goal is the location of starting points for further investigations, Wilde and Scully focus on locating specific computational units rather than all required computational units. The approach deals with one feature at a time and gives little insight into connections between sets of related features. If a set of related features is to be considered rather than a single feature, one could repeat the analysis invoking each feature separately and then unite the specifically required computational units. Even then the relationships among groups of features cannot be recognized.

Another approach based on dynamic information is taken by Wong and colleagues [32]. They analyze execution slices (corresponds to our execution profiles) 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 computational units can be identified that implement the feature.

For deriving all required computational units, the execution slice for the including input set is sufficient. By subtracting all computational units in the execution slice for the excluding input set from those in the execution slice for the invoking input set, only those computational units remain that specifically deal with the feature. This information alone is not sufficient to identify the interface and the constituents of a component in the source code, but those computational units are at least a starting point for a more detailed static analysis. Again, interdependencies between features are not revealed easily.

In [33], Wong et al. present a way for quantification of features. Metrics are provided to compute the *dedication* of computational units to features, the *concentration* of features in computational units, and the *disparity* between features. This work complements their earlier research and can be used as a refinement for Wilde's technique.

Chen and Rajlich [34] propose a semi-automatic method for feature location, in which the programmer browses the statically derived abstract system dependency graph (ASDG). The ASDG describes detailed dependencies among routines, types, and variables at the level of global declarations. The navigation on the ASDG is computer-aided and the programmer takes on all the search for a feature's implementation. The method takes advantage of

the programmer's experience with the analyzed software. It is less suited to locate features if programmers without any pre-knowledge do not know where to start the search.

The ASDG's quality is essential for the method. If the ASDG includes overoptimistic assumptions on function pointers, the programmer may miss routines called via function pointers. If it reflects too conservative assumptions, the search space increases drastically. It is statically undecidable which control flow paths are taken at runtime, so that every conservative static analysis will yield an overestimated search space. In contrast, dynamic analyses exactly reveal which parts are actually used at runtime—although only for a particular run. Insights from dynamic analyses are only valid for the input data used and the environment in which the system was run.

Recently, Wilde and Rajlich compared their approaches [35]. In the presented case study, both techniques were effective in locating features. The Software Reconnaissance showed to be more suited to large infrequently changed programs, whereas Rajlich's method is more effective if further changes are likely and require deep and more complete understanding.

Visualization of Object-Oriented Systems

De Pauw and colleagues [36], [37], [38] provide a general model for the visualization of the execution of object-oriented systems. Their language and platform independent approach visualizes dynamic information about the runtime behavior by means of message sequence charts and chart-like views for summary information.

Program Explorer [24], [25] by Lange and Nakamura is a tool for understanding C++ programs by means of visualization. Both static and dynamic information is extracted and combined for the presentation of an object-oriented system. The static information derived from the source (like class hierarchy and structural data) is stored in a program database. The dynamic information comprises method invocation, object longevity, and variable accesses and is gained off-line from execution traces. Program Explorer offers selective instrumentation of the source, requiring the user to have a certain knowledge about the system. To cope with the amount of information, the user can further merge, prune, or slice results of analyses to remove undesired information. The dynamic information is coupled with the static information yielding class-to-object and object-to-class clarification. Program Explorer is not useful for global understanding, the user must have knowledge about the system and then focus on relevant parts. The approach is class and object centered and does not offer other levels of abstraction.

Koskimies and Mössenböck developed Scene [23], a tool for visualizing object-oriented systems written in the programming language Oberon. Scene uses scenario diagrams for visualizing the message flow between objects in terms of method invocations. The scenario diagrams are generated from event traces and linked to other sources of information.

Jerding and colleagues [39], [40] focus on the interactions between program components at runtime. They observed that recurring interaction pattern can be used in the abstraction process for program understanding. The authors developed a pattern identification algorithm and structure the dynamic information by using identified patterns. The work primarily aims at object-oriented systems but also seems applicable for procedural programming paradigms. Jerding and Rugaber present the tool ISVis [40] to support architectural localization and extraction. They use both static and dynamic information to extract components and connectors. The components are specified by the analyst (using traditional static analyses) whereas the connectors are recognized from actual execution traces. These execution traces are then analyzed with the aforementioned methods. The dynamic information is visualized as a variant of message sequence charts; the user has the ability to restrict the instrumentation to specific files of the system.

Systä [41] focuses on reverse engineering Java legacy systems. She discusses the combination of static and dynamic information when reengineering a Java environment. Rigi [28] is used to extract the static information from class files and to connect the dynamic information (represented as state diagrams) gained through program runs.

Visualization and Abstraction

Another effort to combine dynamic and static information about object-oriented systems is taken by Richner and Ducasse [42]. They offer a query-based approach where the facts about the legacy system are modeled in terms of logical facts. The queries produce different views of the software (at different levels of abstraction) and help to restrict the amount of data generated. There is no information exchange between the views.

Sefika and colleagues [43] visualize statics and dynamics of an object-oriented system in terms of its architectural abstractions. The code instrumentation is light-weight and architecture-aware. It provides efficient on-line instrumentation to support architecture-guided queries. The architectural abstraction are taken as a basis for the visualization. Similarly, Walker and colleagues [44] aim at visualization of dynamic information on a higher level of abstraction. They use program animation techniques for program understanding.

Most recently, Robbillard and Murphy [45] address the problem of crosscutting concerns in object-oriented systems. They propose the usage of *Concern Graphs* that abstract implementation details of concerns and explicitly show relationships between parts of the concerns. The extraction of concern graphs from a given legacy system could benefit from dynamic feature-location techniques.

Concept Analysis

Primarily Snelting has recently introduced concept analysis to software engineering. Since then it has been used to evaluate class hierarchies [46], explore configuration structures of preprocessor statements [47], [48], for redocumentation [49], and to recover components [50]–[56].

All of that research utilizes static information derived from source code.

A technique similar to ours is taken by Ball [57]. He describes how to use concept analysis for the dynamic analysis of test sets. The source code is instrumented and profile information is gathered. The results of concept analysis on the data are used to provide an intermediate point between entity-based and path-based coverage criteria.

Summary

All the researchers using program traces face the same problem: the huge amount of data that is produced by the execution. The problem is tackled by removing undesired information—either by instrumenting only parts of the system or by providing filtering mechanisms (patterns or static information) on the stored traces.

The amount of information gained by profiling rather than tracing is much smaller (and less precise), and can therefore be handled more efficiently. Even profiling on a more fine grained level than routines or methods (e.g., basic blocks) leads to comprehensible results. For our primary goals, the sequences of operations was not crucial and can at least in parts be regained from static information. The frequency of invocations does not play a major role by now, but we believe that such information could be exploited in future research.

VII. Conclusions

The technique presented in this paper identifies computational units specific to a set of related features using execution profiles for different usage scenarios. At first, concept analysis—a mathematically sound technique to analyze binary relations—allows locating the most feature-specific computational units among all executed computational units. Then, a static analysis uses these feature-specific computational units to identify additional feature-specific computational units along the dependency graph. The combination of dynamic and static information reduces the search space drastically.

The value of our technique has been demonstrated by several case studies. In one case study, analyzing two web browsers, we could recover a partial description of the software architecture with respect to a specific set of related features. Commonalities and variabilities between these partial architectures could be recovered quickly. Altogether, we found in two experiments with two systems 16 and 6, respectively, feature-specific routines out of 701 routines for Mosaic and 3 and 24, respectively, out of 928 for Chimera. Only very few routines needed to be inspected manually.

The second case study was performed on a 1.2 million LOC production system. The experiences we made during that case study showed two problems of our approach: the growing complexity of concept lattices for large systems with many features and the need for handling compositions of features.

In this paper, we extended our technique to solve these problems. We showed how the method allows incrementally exploring features while preserving the "mental map" the analyst has gained through the analysis.

The second improvement described in this paper is a detailed look at composing features into more complex scenarios. Rather than assuming a one-to-one correspondence between features and scenarios as in earlier work, we can now handle scenarios that invoke many features.

Further, the implementation of our approach is simple. For concept analysis we used the tool concepts [58]. For visualization we used our graphical Bauhaus front end [26]. Layouts are generated by GraphViz [30]. The glue code is written in Perl, for compiling and profiling we used gcc and gprof.

Acknowledgments

We would like to thank Gerd Bleher and Jens Elmenthaler (both at Agilent Technologies) for their support in the Agilent case study. We also like to thank Tahir Karaca, Markus Knauss, and Stefan Opferkuch (all students at the University of Stuttgart) for preparing the test cases in the Agilent case study.

References

[1] Meir M. Lehman, "Programs, Life Cycles and the Laws of Software Evolution," *Proceedings of the IEEE, Special Issue on Software Evolution*, vol. 68, no. 9, pp. 1060–1076, Sept. 1980.

[2] Rainer Koschke, *Atomic Architectural Component Recovery for Program Understanding and Evolution*, Dissertation, Universität Stuttgart, Germany, 2000.

[3] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon, "Derivation of Feature-Component Maps by Means of Concept Analysis," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, Mar. 2001, pp. 176–179, IEEE Computer Society Press.

[4] "The XFIG drawing tool, Version 3.2.3d," Available at `http://www.xfig.org/`, 2001.

[5] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[6] Norman Wilde and Michael C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49–62, Jan. 1995.

[7] Susan Horwitz, Thomas Reps, and David Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[8] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy, "Dynamic slicing method for maintenance of large C programs," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*. Mar. 2001, pp. 105–113, IEEE Computer Society Press.

[9] Lars Ole Andersen, *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, University of Copenhagen, Danmark, 1994.

[10] Guiliano Antoniol, F. Calzolari, and Paolo Tonella, "Impact of Function Pointers on the Call Graph," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, Amsterdam, Netherlands, Mar. 1999, pp. 51–59.

[11] Ben-Chung Cheng and Wen-Mei W. Hwu, "Modular interprocedural pointer analysis using access paths," in *Proceedings of the Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000, pp. 57–69.

[12] Manuvir Das, "Unification-based Pointer Analysis with Directional Assignments," in *Proceedings of the Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000, pp. 35–46.

[13] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the

Presence of Function Pointers," in *Proceedings of the Conference on Programming Language Design and Implementation*, Orlando, FL, USA, 1994, pp. 242–257.

[14] Robert P. Wilson and Monica S. Lam, "Efficient context-sensitive pointer analysis for c programs," in *Proceedings of the Conference on Programming Language Design and Implementation*, La Jolla, CA, USA, 1995, pp. 1–12.

[15] Sean Zhang, Barbara G. Ryder, and William Landi, "Program decompositon for pointer aliasing: A step towards pratical analyses," in *Symposium on the Foundations of Software Engineering*, 1996, pp. 81–92.

[16] Bjarne Steensgaard, "Points-To Analysis in almost linear time," in *Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, USA, Jan. 1996, pp. 32–41.

[17] Amer Diwan, Kathryn McKinley, and Eliot Moss, "Using types to analyze and optimize object-oriented programs," *Programming Languages and Systems*, vol. 23, no. 1, pp. 30–72, 2001.

[18] Atanas Rountev, Ana Milanova, and Barbara G. Ryder, "Points-To Analysis for Java using Annotated Constraints," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, Tampa, FL, USA, Oct. 2001, pp. 43–55.

[19] Ana Milanova, Atanas Rountev, and Barbara G. Ryder, "Precise Call Graph Construction in the Presence of Function Pointers," in *Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation*, Montreal, Canada, Oct. 2002, IEEE Computer Society Press.

[20] Garret Birkhoff, *Lattice Theory*, American Mathematical Society Colloquium Publications 25, Providence, RI, USA, first edition, 1940.

[21] Bernhard Ganter and Rudolf Wille, *Formal Concept Analysis—Mathematical Foundations*, Springer, 1999.

[22] "IDEF0," Available at `http://www.idef.com/idef0.html`, Dec. 1993.

[23] Kai Koskimies and Hanspeter Mössenböck, "Scenario-Based Browsing of Object-Oriented Systems with Scene," Report 4, Johannes Kepler Universität Linz, Austria, Aug. 1995.

[24] Danny B. Lange and Yuichi Nakamura, "Program Explorer: A Program Visualizer for C++," in *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, CA, USA, June 1995.

[25] Danny B. Lange and Yuichi Nakamura, "Object-Oriented Program Tracing and Visualization," *Computer*, vol. 30, no. 5, pp. 63–70, May 1997.

[26] "The New Bauhaus Stuttgart," Available at `http://www.bauhaus-stuttgart.de/`, 2002.

[27] Rainer Koschke, Jean-François Girard, and Martin Würthner, "An Intermediate Representation for Reverse Engineering Analyses," in *Proceedings of the 5th Working Conference on Reverse Engineering*, Honolulu, HI, USA, Oct. 1998, pp. 241–250, IEEE Computer Society Press.

[28] "Rigi—a visual tool for understanding legacy systems," Available at `http://www.rigi.csc.uvic.ca/`, 2002.

[29] David Garlan, Robert Allen, and John Ockerbloom, "Architectural Mismatch or Why It's Hard to Build Systems Out Of Existing Parts," in *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, USA, Apr. 1995, pp. 179–185, ACM Press.

[30] AT&T Labs-Research, "GraphViz — Open Source Graph Drawing Software," Available at `http://www.research.att.com/sw/tools/graphviz/`, 2002.

[31] Norman Wilde, Juan A. Gomez, Thomas Gust, and Douglas Strasburg, "Locating User Functionality in Old Code," in *Proceedings of the International Conference on Software Maintenance*, Orlando, FL, USA, Nov. 1992, pp. 200–205, IEEE Computer Society Press.

[32] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi, "Locating Program Features using Execution Slices," in *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, Richardson, TX, USA, Mar. 1999, pp. 194–203, IEEE Computer Society Press.

[33] W. Eric Wong, Swapna S. Gokhale, and Joseph R. Hogan, "Quantifying the Closeness between Program Components and Features," *The Journal of Systems and Software*, vol. 54, no. 2, pp. 87–98, Oct. 2000.

[34] Kunrong Chen and Václav Rajlich, "Case Study of Feature Location Using Dependence Graph," in *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000, pp. 241–249, IEEE Computer Society Press.

[35] Norman Wilde, Michelle Buckellew, Henry Page, and Václav Rajlich, "A Case Study of Feature Location in Unstructured Legacy Fortran Code," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, Mar. 2001, pp. 68–75, IEEE Computer Society Press.

[36] Wim de Pauw, Richard Helm, Doug Kimelman, and John Vlissisdes, "Visualizing the Behavior of Object-Orient Systems," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, Washington, DC, USA, Sept. 1993, pp. 326–337, ACM Press.

[37] Wim de Pauw, Doug Kimelman, and John Vlissides, "Modeling Object-Oriented Program Execution," in *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, July 1994, vol. 821 of *Lecture Notes in Computer Science*, pp. 163–182, Springer.

[38] Wim de Pauw, David Lorenz, John Vlissides, and Mark Wegman, "Execution Patterns in Object-Oriented Visualization," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems*, Santa Fe, NM, USA, 1998, pp. 219–234.

[39] Dean F. Jerding, John T. Stasko, and Thomas Ball, "Visualizing Interactions in Program Executions," in *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, May 1997, pp. 360–370, ACM Press.

[40] Dean F. Jerding and Spencer Rugaber, "Using Visualization for Architectural Localization and Extraction," *Science of Computer Programming*, vol. 36, no. 2–3, pp. 267–284, Mar. 2000.

[41] Tarja Systä, "On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software," in *Proceedings of the 6th Working Conference on Reverse Engineering*, Atlanta, GA, USA, Oct. 1999, pp. 304–313.

[42] Tamar Richner and Stéphane Ducasse, "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information," in *Proceedings of the International Conference on Software Maintenance*, Oxford, England, UK, Aug. 1999, pp. 13–22, IEEE Computer Society Press.

[43] Hohlalefi Sefika, Aamod Sane, and Roy Campbell, "Architecture-Oriented Visualization," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, San Jose, CA, USA, Oct. 1995, pp. 389–405, ACM Press.

[44] Robert J. Walker, Gail C. Murphy, Bjørn N. Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak, "Visualizing Dynamic Software System Information Through High-Level Models," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, Canada, 1998, pp. 271–283.

[45] Martin P. Robillard and Gail C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," in *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, USA, May 2002.

[46] Gregor Snelting and Frank Tip, "Reengineering Class Hierarchies using Concept Analysis," in *Proceedings of the 6th SIGSOFT Symposium on Foundations of Software Engineering*, Orlando, FL, USA, Nov. 1998, pp. 99–110, ACM Press.

[47] Maren Krone and Gregor Snelting, "On The Inference of Configuration Structures from Source Code," in *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 49–58, IEEE Computer Society Press.

[48] Gregor Snelting, "Reengineering of Configurations Based on Mathematical Concept Analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 146–189, Apr. 1996.

[49] Tobias Kuipers and Leon Moonen, "Types and Concept Analysis for Legacy Systems," in *Proceedings of the 8th International Workshop on Program Comprehension*. June 2000, pp. 221–230, IEEE Computer Society Press.

[50] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Guiseppe A. Di Lucca, "A Case Study of Applying an Eclectic Approach to Identify Objects in Code," in *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, PA, USA, May 1999, pp. 136–143, IEEE Computer Society Press.

[51] Holger Graudejus, "Implementing a Concept Analysis Tool for Identifying Abstract Data Types in C Code," Diplomarbeit, Universität Kaiserslautern, Germany, 1998.

[52] Christian Lindig and Gregor Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis," in *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, May 1997, pp. 349–359, IEEE Computer Society Press and ACM Press.

[53] Hourai Sahraoui, Walcélio Melo, Hakim Lounis, and François Dumont, "Applying Concept Formation Methods to Object Identification in Procedural Code," in *Proceedings of the International Conference on Automated Software Engineering*, Lake Tahoe, CA, USA, Nov. 1997, pp. 210–218, IEEE Computer Society Press.

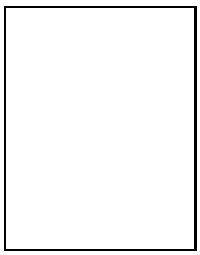
[54] Michael Siff and Thomas Reps, "Identifying Modules via Concept Analysis," in *Proceedings of the International Conference on Software Maintenance*, Bari, Italy, Oct. 1997, pp. 170–179, IEEE Computer Society Press.

[55] Arie van Deursen and Tobias Kuipers, "Identifying Objects using Cluster and Concept Analysis," in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, 1999, pp. 246–255, IEEE Computer Society Press.

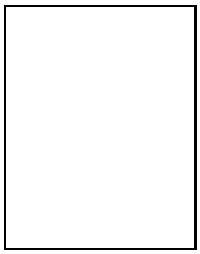
[56] Paolo Tonella, "Concept Analysis for Module Restructuring," *IEEE Computer Society Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, Apr. 2001.

[57] Thomas Ball, "The Concept of Dynamic Analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, Nov. 1999.

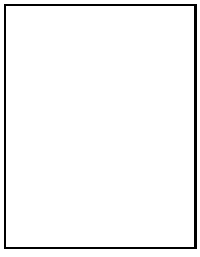
[58] Christian Lindig, "Concepts 0.3e," Available at `http://www.gaertner.de/~lindig/software/`, 1999.



Thomas Eisenbarth received his Diploma in computer science from the University of Stuttgart, Germany in 1998. Since then, he is working for his dissertation at the University of Stuttgart in the field of reverse engineering as a member of the Bauhaus [26] project. His research interest is in reengineering, reverse engineering, program understanding, and software architecture. He focuses recovery methods for connectors from the source code.



Rainer Koschke is a post-doctoral researcher at the computer science department at the University of Stuttgart. His research interests are primarily in the fields of software engineering and program analyses. His current research includes architecture recovery, feature location, program analyses, and reverse engineering. He teaches reengineering, compilers, and programming language concepts. He holds a doctoral degree in computer science from the University of Stuttgart, Germany.



Daniel Simon received his Diploma in computer science from the Saarland University at Saarbrücken, Germany in 2000. Since then, he is working for his dissertation at the University of Stuttgart in the field of reverse engineering as a member of the Bauhaus [26] project. His research interest are in the field of reverse engineering, program analysis, and program understanding. He co-authored several papers on feature location and software product lines, which is his current research focus.