Incremental Location of Combined Features for Large-Scale Programs

Thomas Eisenbarth, Rainer Koschke, and Daniel Simon
Universität Stuttgart
Breitwiesenstraße 20–22
70565 Stuttgart, Germany

E-mail: {eisenbarth,koschke,simon}@informatik.uni-stuttgart.de

Abstract

The need for changing a program frequently confronts maintainers with the reality that no valid architectural description is at hand. To solve that problem, we presented at ICSM 2001 a language-independent and easy to use technique for opportunistic and demand driven location of features in source code based on static and dynamic analysis and concept analysis.

In order to further validate the technique, we recently performed an industrial case study 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.

This paper extends our technique to solve these problems. We show how this method allows incremental exploration of features while preserving the "mental map" the maintainer has gained through the analysis. The second improvement 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.

1. Introduction

Our technique for feature location presented at ICSM 2001 [4] identifies all components specific to a set of related features using execution profiles for different usage scenarios. At first, concept analysis allows us to locate the most feature-specific routines among all executed routines. Then, a static analysis uses these feature-specific routines to identify additional feature-specific routines along the dependency graph. The combination of dynamic and static information reduces the search space drastically.

We have conducted a large scale industrial case study and extended the technique presented in [4] by two important aspects: if scenarios execute several features, we can still identify the features in the lattice. Further, we now can handle large and complex systems, because our technique can be applied incrementally.

We assume that the reader is familiar with the basics of formal concept analysis. A brief introduction can be found in [4] whereas the mathematical background is discussed in [5]. For the end-user of the technique, the mathematical background can be hidden completely.

1.1 Terminology

A *feature* f is a realized functional requirement (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, i.e., we consider a feature an observable result of value to the user.

A *scenario* s is a sequence of user inputs triggering actions of a system that yields an observable result. A scenario s *executes* a feature f if f's result can be observed by the user when the system is used as described by the scenario. A scenario may execute multiple features.

A *component* is a computational unit of a system. In our case study, we consider routines as components. A *routine* r is a function, procedure or subprogram according to the programming language.

The *execution trace* lists the sequence of all performed calls. The *execution profile* of a given program is the set of all routines in the execution trace.

A *feature-component map* describes which components implement a given set of relevant features.

2. Analysis Process

The process steps relevant here are the following:

1. Perform dynamic analysis.
2. Interpret the lattice as a result of the dynamic analysis.
3. Iterate through steps 1 to 2 until sufficient knowledge about the system has been gained.

The basic interpretation of the concept lattice is described in previous work [4]. The refinement of the interpretation and the ideas required for the iteration are explained in the following sections.

2.1. Dynamic Analysis

The dynamic analyses we perform make use of *formal concept analysis*. The mathematical background that is required to understand the following considerations is provided in [4, 5]. The inputs to the process are the source code and a set of initially relevant features. We proceed as follows:

1. The source code is compiled with profiling information.
2. Based on previously identified features, the domain expert creates scenarios.
3. The system is executed according to the scenarios, and execution profiles are recorded.
4. A number of execution profiles is selected in order to set up the context for concept analysis.
5. Concept analysis is performed.

In order to derive the feature-component map by means of concept analysis, we have to define the formal context $C = (O, A, \mathcal{R})$, i.e., the objects, the attributes, and the relation, and to interpret the resulting concept lattice accordingly.

The goal of the dynamic analysis is to find out which routines contribute to a given set of features. For each feature, a scenario is prepared that exploits this feature. The formal context for applying concept analysis to derive the feature-component map will be laid down as follows:

- $O = R$ (the set of all routines),
- $A = S$ (the set of all scenarios),
- a pair ($r \in R$, $s \in S$) is in relation $\mathcal{R}$ if r is executed when s is performed.

	f_1	f_2	f_3	r_1	r_2	r_3	r_4	r_5	r_6	r_7
s_1	×		×	×			×		×	×
s_2	×	×			×		×	×		×
s_3		×	×			×		×	×	×

Figure 1. Invocation relation $\mathcal{R}$.

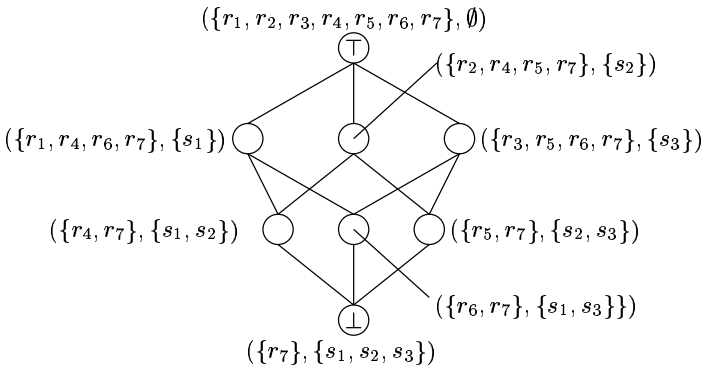

Figure 2. Concept lattice for context in Fig. 1.

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 routines for a single scenario, i.e., one column of the relation table. Applying all scenarios provides the complete relation table.

2.2. Interpretation of Concept Lattice

How to identify relationships between scenarios and routines based on the concept lattices is described in [4]. There, we assumed a one-to-one correspondence between scenarios and features. With real systems, we rather have an m-to-n mapping. Therefore, we now describe how to identify relationships between scenarios and features and thus between features and routines.

Despite an m-to-n relationship between features and scenarios, there is a simple way to identify routines 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 executes 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 two scenarios executing the same features differ only in the order or frequency of feature invoca-

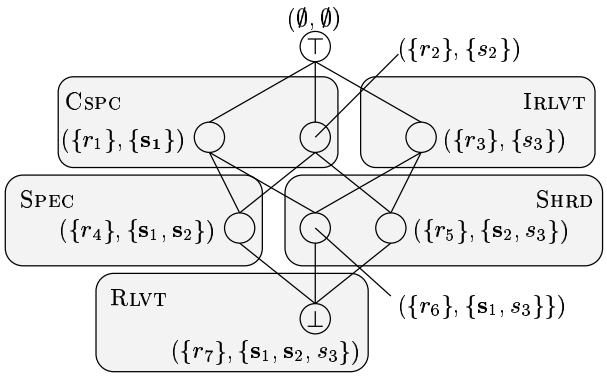

Figure 3. Sparse concept lattice for Fig. 2 with categorization.

tion, the scenarios may indeed be considered complex features in their own right.

With the maintenance engineer's additional knowledge of which features are invoked by a scenario we can identify the routines relevant to a certain feature. Let us consider the invocation relation $\mathcal{R}$ in Fig. 1 (for better legibility, scenarios are listed as rows and routines are listed as columns). The table contains the called routines $r_1, \ldots, r_7$ per scenario, and furthermore the executed 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. 1 is shown in Fig. 2. The feature part of the table is ignored while constructing this lattice.

Assume we are interested in feature f_1. Routines specific to feature f_1 can be found in the intersection of the executed routines of the two scenarios s_1 and s_2 because f_1 is used for s_1 and s_2. The intersection of the routines 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: $\mu(s_1) \sqcap \mu(s_2) = (\{s_1, s_2\}, \{r_4, r_7\})^\dagger$. Since s_1 and s_2 do not share any other feature, the routines particularly relevant to f_1 are r_4 and r_7.

We notice that r_7 is also used in all other scenarios, so that one cannot consider r_7 a specific routine for either one of f_1, f_2, or f_3. Routine r_4, in contrast, is only used in scenarios executing f_1. We therefore conclude that r_4 is specific to f_1 whereas r_7 is not. Because there is no other scenario containing f_1 other than s_1 and s_2, routine r_4 is the only routine specific to f_1.

Note that this is just an hypothesis because other features might be involved to which r_4 is truly specific and that are not explicitly listed in the scenarios. Another explanation could be that, by accident, r_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 r_4 is specific to f_1 because r_4 is not executed when f_2 and f_3 are jointly invoked in s_3, which suggests that r_4 at least comes only into play 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.

Routines that are somehow related to but not specific for f_1 are such routines that are executed for scenarios containing f_1 amongst other features. Consider all routines in our example executed for s_1 or s_2. Routines in extents of concepts which contain s_1 or s_2 are therefore potentially relevant to f_1. In our example, this yields r_1, r_2, r_5 and r_6 in addition to r_4 and r_7. Routine r_3 is only executed for scenario s_3, which does not contain f_1.

Altogether, we can identify five categories for routines with regard to feature f_1 (see Fig. 3):
SPEC: r_4 is specific to f_1 because it is used in all scenarios using f_1 but not in other scenarios.
RLVT: r_7 is relevant to f_1 because r_7 is used in all scenarios using f_1; but it is also more general than r_4 because r_7 is also used in scenarios not using f_1 at all.
CSPC: r_1 and r_2 are only called in scenarios using f_1. They are less specific than r_4 because they are not used in all scenarios that use f_1. Whether r_1 and r_2 are more or less specific than r_7 is not decidable based on the concept lattice. On one hand, they are used in all scenarios using f_1 and other scenarios, whereas r_7 is also called in scenarios that do not require f_1. On the other hand, r_7 is called whenever f_1 is required, whilst r_1 and r_2 are not called in some scenarios that do require f_1.
SHRD: r_5 and r_6 are called in scenarios using f_1 but they are also called in scenarios not using f_1. These routines are presumably less relevant than r_1 and r_2, which are called only when f_1 is used, and also less relevant than r_7, which is called in all scenarios using f_1.
IRLVT: r_3 is irrelevant to f_1 because r_3 is only called 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 for which the following condition holds:

$$c = (R, S) \text{ and } \bigcap_{s_j \in S} s_j = \{f\} \qquad (*)$$

Based on the identified concept, one can categorize the routines as follows:
SPEC: all routines r for which $\gamma(r) = c$ holds.

†The *common attributes* (σ), *attribute concept* (μ), *object concept* (γ), and *infimum* ($\sqcap$) are defined as usual.

RLVT: all routines r for which $\gamma(r) = c'$ and $c' < c$ holds.
CSPC: all routines r for which $\gamma(r) = c'$ and $c < c'$ holds.
SHRD: all routines r for which r is in the intent of concept c' where $c < c'$ holds and c and $\gamma(r)$ are incomparable.
IRLVT: all other routines not categorized by other categories.

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

As a matter of fact, there could be several concepts for which condition (∗) holds when different routines 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 a symbolic debugger and we are interested in its features to set and delete breakpoints. Three scenarios can be provided to explore these two features: "set a new breakpoint ({`break`})", "set and delete a breakpoint ({`break`, `delete`})", and "try to delete a breakpoint never set ({`delete`})".

For the overlapping scenarios {`break`, `delete`} and {`delete`}, we may assume that different routines will be called beyond those that are specific to command `break`: Quite likely, additional routines will be called to handle the erroneous attempt to delete a breakpoint that was never set in the latter scenario.

In case of multiple concepts for which condition (∗) holds, we can unite the routines that are in category 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 (∗) holds, one needs additional scenarios that factor out feature f. For instance, in order to isolate feature f_a in scenario $s_1 = \{f_a, f_c\}$, one can simply add a new scenario $s_2 = \{f_a, f_b\}$. The routines specific to f_a 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. The lattice exactly tells which features are not yet isolated and which scenarios invoke these features. Slightly modified variants of these scenarios can be added to isolate these features specifically.

	r_1	r_2	r_3	r_4
s_1			×	
s_2		×	×	
s_3	×	×		
s_4	×			

Figure 4. Subcontext.

2.3. 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 localize 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, routines with low cohesion, i.e., routines 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 is analogous. If one starts with a smaller set of scenarios and further increases this set, all accumulated knowledge a maintenance engineer gained while working with the smaller lattice has to be preserved. The lattice—the mental map for the engineer'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*).

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

In our application of concept analysis, we only add

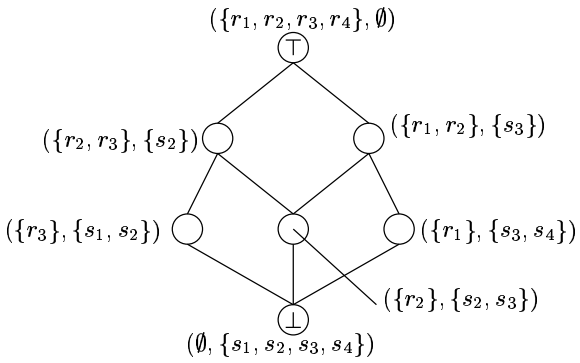

Figure 5. Lattice for the context in Fig. 4.

new attributes but never new objects to the relation table. Adding new attributes leads to a new formal context $(O, A, \mathcal{R})$ in which relation $\mathcal{R}$ extends relation $\mathcal{R}'$.

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

Proof. See [5].

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 [5]):

$$(O, A) \mapsto (O, \sigma(O))$$

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 (e.g., the shaded concept in Fig. 7 does not show up in Fig. 5). 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. Names and numbering schemes for existing contexts can be reused. Additionally, an incremental automatic graph layout can be chosen: Only additional

	r_1	r_2	r_3	r_4
s_1			×	
s_2		×	×	
s_3	×	×		
s_4	×			
s_5	×		×	×
s_6	×	×		

Figure 6. Supercontext.

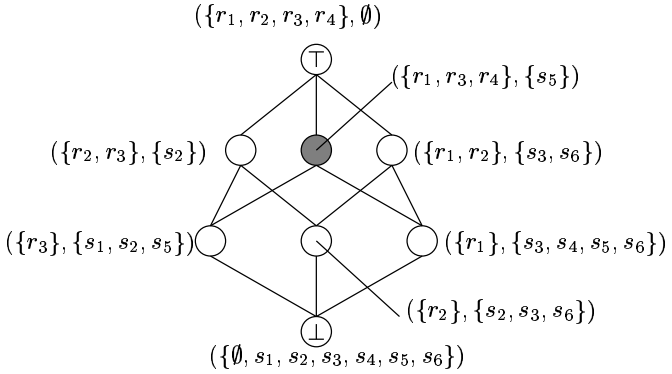

Figure 7. Lattice for the context in Fig. 6.

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 may be preserved.

For instance, in Fig. 6, the new scenarios s_5 and s_6 are added to the context of Fig. 4. The lattices in Fig. 5 and the extended lattice in Fig. 7 show the effects of the transition from the sub- to the supercontext.

3. Case Study

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 stressed the importance of incremental understanding of very large concept lattices as described in Sect. 2.3 and the modeling of scenarios as set of features as explained in Sect. 2.1.

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

3.1. Agilent 93000 SOC Series

The *SmarTest* software controls Agilent 93000 devices. It lets a chip test engineer write test cases used for chip testing. The software comprises several tools

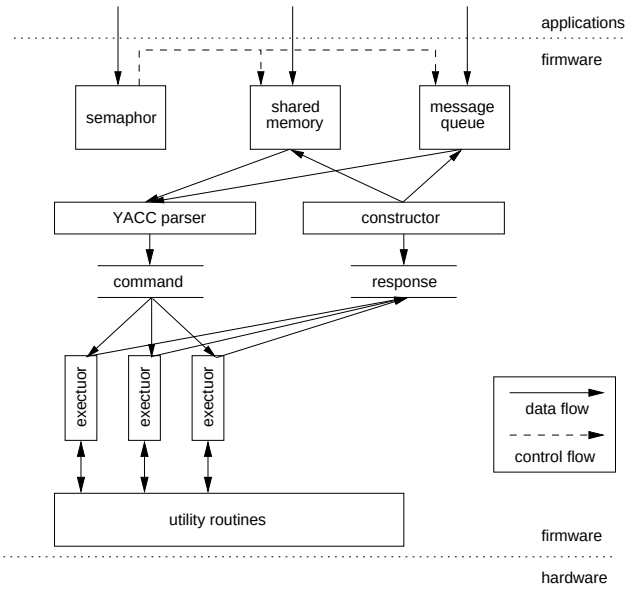

Figure 8. Software architecture of Agilent 93000 firmware.

for test setup and result visualization. The interface between the test software and the hardware is called *firmware*. In this case study, we analyzed the firmware.

The firmware has evolved over 15 years and is written in C. Today, it consists of 1.2 MLOC with comments or about 500 KLOC non-commented. The static call graph of the part of the firmware that was analyzed for this case study had 9.988 routines.

Figure 8 depicts the software architecture of the firmware as described by one of the software's architects. The input to the firmware are the test cases consisting of firmware commands. The firmware parses and interprets each command, drives the Agilent 93000 device, and returns the result. In order to run a command, the firmware dispatches the corresponding C routine (the *executor*) that acts as an entry point to the implementation of the command.

As Fig. 8 suggests, the executors share a set of reusable utility routines. Which utility routines are actually shared by which executors is not shown in the architectural sketch. As a matter of fact, the software architect does currently not exactly know the relation between executors and utility routines.

For this case study, we focused on the commands of the digital part of the firmware:

- *Configuration Setup Commands* assign 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* specify the signal mode and connection for each pin, and the order of connections.
- *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* 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* are required to set up and sequence test vectors.
- *Relay Control Commands* are used to set relay positions and the tester state.

The goal of our case study was to validate the architecture sketch in Fig. 8 and to show which utility routines are really shared. Given these 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.

3.2. 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. 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. As already described in Sect. 2.2, we can thus model a test case (scenario) as set of commands (features) $s = \{cmd_1, cmd_2, \ldots, cmd_m\}$.

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 Sect. 2.2.

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

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 all possible 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 varying in 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, i.e., the timing setup and level setup parts.

Each test case represents a scenario. In total, 93 scenarios were provided (cf. Fig. 9). 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.

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

Figure 9. Test cases / scenarios.

3.3. Resulting Concept Lattice

The resulting concept lattice is shown in Fig. 10. 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).

Another developer at Agilent (different from the software architect who prepared the test cases) was asked to validate the resulting concept lattice. To make a clear distinction between this validating expert and the expert who prepared the scenarios, 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 the lattice surprises him.

According to the developer, the concept lattice in Fig. 10 maps well to the architecture sketch of Fig. 8. He immediately spotted in the 65 direct subconcepts of the top element—i.e., concepts in the first row below the top element of the lattice—the individual executors for 65 commands (including the executor for NOP). (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. 3.2). 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.

The other twelve real scenarios can be found in subconcepts of the above mentioned 65 concepts. The

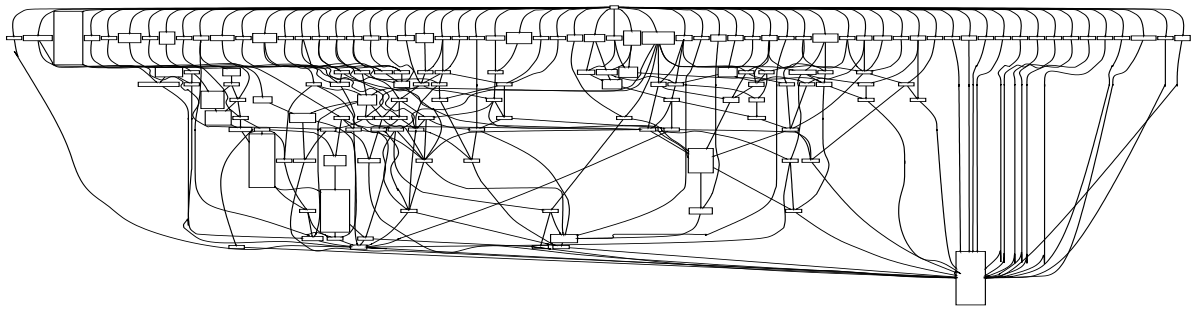

Figure 10. Sparse lattice for all commands. The size of the boxes corresponds to the number of routines.

twelve real scenarios cannot be found directly below the top element because 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 Sect. 2.2, we could easily identify the executors for the remaining twelve commands whose test case is not directly located below the top element.

As described above, the firmware commands can be categorized in different classes. In order to visualize the jointly used routines by executors for commands of the same class, we colored the concept lattice as follows:

- 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;
- 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. 10 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 r_i for which $\gamma(r_i) = c$ holds contribute to commands of different classes.

As a matter of fact, there were only few concepts above the bottom element with multiple colors showing that there is substantial sharing of routines among executors for the same class and that these utility routines seem to be specific to just one 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. 8: 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, i.e., 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. 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. Whereas 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 function inlining in the subsystem implementing pin group configuration.

In general, the concepts just below the top element contain only one routine, some of them contain more than one but less than five. In these cases, a programmer has split a large executor into smaller pieces. There is only 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 fur-

ther 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, i.e., 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.

3.4. 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 ten 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. However, since there was a naming convention for executors in place, localizing the executors could have been done with textual search tools, such as *grep*, more easily, he noted. 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 at 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. 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.

Another difficulty is the problem of handling parameterized scenarios, i.e., scenarios that are alike except for values of certain parameters. For instance, most commands of the firmware have options. The same command may execute different routines for different options.

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 was relatively large and complex. Such large concept lattices are a challenge for visualization.

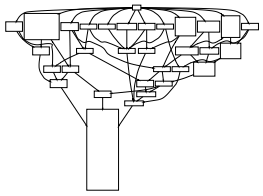

Figure 11. Timing commands.

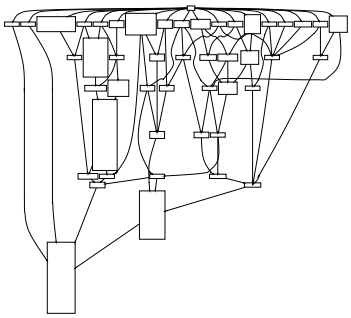

Figure 12. Timing and vector commands.

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 Sect. 2.3. The visual difference for considering scenarios incrementally is illustrated by Fig. 11 and Fig. 12. Figure 11 contains the concept lattice for all Timing Setup commands. For the lattice in Fig. 12, all scenarios for Vector Setup have been added. When all scenarios for all classes of commands are added, the lattice in Fig. 10 is obtained.

4. Related Research

Primarily Snelting has recently introduced concept analysis to software engineering. Since then it has been used to evaluate class hierarchies [11], explore configuration structures of preprocessor statements [7, 10], for re-documentation [8], and to recover components, e.g., [2, 13, 12]. All of that research utilizes static information derived from source code.

Wilde et al. [15] pioneered in localizing features taking a fully dynamic approach. The goal of their *Software Reconnaissance* is the support of maintenance programmers when modifying or extending the func-

tionality of a legacy system. Another approach based on dynamic information is taken by Wong et al. [16]. They analyze execution profiles of test cases implementing a particular functionality. Chen and Rajlich [3] propose a semi-automatic method for feature localization, in which the programmer browses the statically derived abstract system dependency graph. Recently, Wilde and Rajlich compared their approaches [14]. 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.

5. Conclusions

Our technique presented at last year's conference identifies all components specific to a set of related features using execution profiles for different usage scenarios. We validated the technique in a case study on a 1.2 MLOC 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 us to incrementally explore features preserving the "mental map" the maintainer 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 [9]. For visualization we used our graphical Bauhaus front end [1]. The lattice layouts are generated by GraphViz [6]. The glue code is written in Perl, for compiling and profiling we used gcc and gprof.

Acknowledgments The authors would like to thank Gerd Bleher and Jens Elmenthaler (both at Agilent Technologies SOC Test Platform Division at Böblingen, Germany) for their support in the Agilent case study. We also want to thank Tahir Karaca, Markus Knauss, and Stefan Opferkuch (all students at the Universität Stuttgart) for preparing the test cases in the Agilent case study.

References

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

[2] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code. In *Proc. of the 7th IWPC*, pages 136–143, Pittsburgh, PA, USA, May 1999. IEEE Press.

[3] K. Chen and V. Rajlich. Case Study of Feature Location Using Dependence Graph. In *Proc. of the 8th IWPC*, pages 241–249, Limerick, Ireland, June 2000. IEEE Press.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proc. of the ICSM*, pages 602–611, Florence, Italy, Nov. 2001. IEEE Press.

[5] B. Ganter and R. Wille. *Formal Concept Analysis—Mathematical Foundations*. Springer, 1999.

[6] GraphViz. Available at `http://www.research.att.com/sw/tools/graphviz/`, 2002. AT&T Labs-Research.

[7] M. Krone and G. Snelting. On The Inference of Configuration Structures from Source Code. In *Proc. of the 16th ICSE*, pages 49–58, Sorrento, Italy, May 1994. IEEE Press.

[8] T. Kuipers and L. Moonen. Types and Concept Analysis for Legacy Systems. In *Proc. of the 8th IWPC*, pages 221–230. IEEE Press, June 2000.

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

[10] G. Snelting. Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM TOSEM*, 5(2):146–189, Apr. 1996.

[11] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *Proc. of the 6th SIGSOFT Symposium on Foundations of Software Engineering*, pages 99–110, Orlando, FL, USA, Nov. 1998. ACM Press.

[12] P. Tonella. Concept Analysis for Module Restructuring. *IEEE TSE*, 27(4):351–363, Apr. 2001.

[13] A. van Deursen and T. Kuipers. Identifying Objects using Cluster and Concept Analysis. In *Proc. of the 21st ICSE*, pages 246–255, Los Angeles, CA, USA, 1999. IEEE Press.

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

[15] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, Jan. 1995.

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