Delft University of Technology
Software Engineering Research Group
Technical Report Series

Splitting a large software repository for easing future software evolution – an industrial experience report

Marco Glorie, Andy Zaidman, Arie van Deursen, Lennart Hofland

Report TUD-SERG-2009-002

TU Delft

SERG

TUD-SERG-2009-002

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Journal on Software Maintenance and Evolution — Research and Practice, Wiley, 2009.

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Experience

Splitting a Large Software Repository for Easing Future Software Evolution — An Industrial Experience Report‡

Marco Glorie[1], Andy Zaidman[2,*,†], Arie van Deursen[2], Lennart Hofland[1]

[1] *Philips Medical Systems, The Netherlands*
[2] *Delft University of Technology, The Netherlands*

SUMMARY

Philips Medical Systems produces medical diagnostic imaging products, such as MR, X-ray and CT systems. The software of these devices is complex, has been evolving for several decades and is currently a multi-MLOC monolithic software repository. In this paper we report on splitting a single software repository into multiple smaller repositories so that these can be developed independently, easing the software's evolution. For splitting the single software repository, we set up two experiments that involve well-known analysis techniques, namely formal concept analysis and clustering. Because of the sheer size of the monolithic software repository, we also propose to use a 'leveled approach', which implies that the analysis technique is applied in several iterations, whereby in some iterations only part of the application is subjected to the analysis technique. Unfortunately, both analysis techniques failed to produce an acceptable partitioning of the monolithic software repository, even if they are combined with our newly proposed leveled approach. We provide a number of valuable lessons learned, which might prevent others from falling into the same pitfalls.

1. Introduction

Philips Medical Systems (PMS) develops and produces complex systems to aid the medical world with monitoring, diagnostic and other activities. Among these systems are the MR (magnetic resonance), the X-ray and the CT (computed tomography) systems. The software for these products is very complex and has been evolving for decades. The systems are a combination of hardware and software,

*Correspondence to: Andy Zaidman, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
‡This extends our previous work which can be found in [1].
†E-mail: a.e.zaidman@tudelft.nl

Received 27th June 2008
Revised 21st November 2008
Accepted 18th December 2008

containing (real-time) embedded software modules. Many software technologies (C, C++, C#, Perl, ...) are used and third party off-the-shelf modules are integrated in the software. The software is developed at multiple sites (Netherlands, USA and India) and more than 100 developers are currently working on the software.

In this study we focus on the software of one of the aforementioned medical diagnostic imaging products[†]. This medical imaging product has a multi-MLOC software repository, called an *archive* in Philips Medical System's terminology [1]. The repository contains approximately 30,000 source code files that are being developed (and maintained) using branching. Merging the multiple development branches causes significant overhead due to the many dependencies. These dependencies make that the feature that has the longest development time determines the release time of the entire project. This approach to development has resulted from many years of evolving the system and the software department of PMS realizes that the current development process needs to be improved in order to speed up and ease future evolution. The *software architecture team* (SWAT) is currently investigating how to improve the development process by evolving the current architecture into a new architecture that allows for easier maintenance. The vision of this team is a new architecture consisting of about seven[‡] software components that can be developed independently.

In order to obtain these independent components the current software repository is analyzed and subsequently modules can be extracted from the single software repository into seven smaller software components. Although out of scope for this particular paper, in order to complete the migration process, clear and stable interfaces should be defined for each of these seven newly formed software components. These stable interfaces should ensure backward and ideally also (partial) forward compatibility. To detect and subsequently map the dependencies that exist in the monolithic software repository we set up two experiments that let us investigate whether two well-known analysis techniques, namely *formal concept analysis* (FCA) and *cluster analysis* (CA), are able to come to an acceptable splitting of the monolithic software repository. Both FCA [2, 3, 4, 5, 6] and CA [7, 8, 9, 10, 11] have previously been used for purposes similar to ours, albeit on a smaller scale. As such, the contributions of this paper are:

- the description of our experiences with applying FCA and CA in an industrial setting on a large-scale legacy application,
- the introduction of a *leveled approach*, to address scalability issues when working with large-scale applications. This approach allows one to apply FCA or CA in several iterations, whereby in some iterations only part of the software repository is subjected to the analysis,
- the introduction of the *concept quality* measure, which can help decide which parts to analyze in detail when using the leveled approach.

This brings us to our main research question for this study: *do formal concept analysis or cluster analysis allow for the splitting of a large-scale monolithic software repository?*

[†]Due to a non-disclosure agreement, we are not at liberty to divulge certain details, amongst others on which specific product we applied our analysis, the exact size, and certain diagrams of the software product under study. In the remainder of this text we will refer to the case as the PMS case.

[‡]This number is based on the experiences of the members of SWAT with (1) the current structuring of the software repository and (2) their own development activities.

The structure of this paper is as follows: the next section provides insight into the context at Philips Medical Systems. In Section 3 we introduce formal concept analysis and explain how we can apply it to the software repository at hand. Section 4 introduces the leveled approach for concept analysis, while Section 5 presents the results we obtained from applying FCA. Cluster analysis is introduced in Section 6, while Section 7 presents and discusses the results that we have obtained with cluster analysis. Section 8 covers related work, after which we conclude with a summary of contributions and suggestions for future work.

2. The Philips Medical Systems Repository

The software repository that we consider contains roughly 30,000 source code files totaling several million lines of code. In turn, these source code files are grouped in nearly 600 *building blocks*; many dependencies exist between these building blocks. Furthermore, the source code repository is organized by structuring the nearly 600 building blocks into a tree-structure. At the highest level of this building block hierarchy we find the *subsystems*, which in turn contain multiple lower-level building blocks. The tree-structure of building blocks, however, does not map directly onto the high-level architecture of the system, as a number of building blocks are part of multiple high-level components.

In this article we narrow the scope to the parts of the source code that are written in C and C++. This means that the scope of the analysis for our experiment in this paper is limited to around 15,000 files and 360 building blocks, still totaling several million lines of source code. A commercial tool called *Sotograph* is available at PMS to extract static relations from the repository [12]. These relations include the following reference kinds: *call*, *read*, *write*, *throw*, *friend declaration*, *inheritance*, *aggregation*, *type access*, *throws*, *polymorphic call*, *component interface call*, *component call*, *component inheritance* and *catch*. The relations are analyzed at the method / function level. Relations on higher levels of abstraction — such as the file or building block level — are obtained by accumulating the relations to and from the lower level levels of abstraction.

We had access to detailed documentation in the form of UML class diagrams. Another form of documentation we had access to is the so-called *project-documentation*, which specifies on a per-project basis (1) the purpose of the project and (2) which building blocks are expected to be within the scope of this particular project. We used the project-documentation of the last two years, which currently means that we have around 50 documents available; as such, unfortunately, the content of these documents does not cover all building blocks.

3. Repository Splitting using Formal concept analysis

3.1. A Primer in Formal Concept Analysis

Formal concept analysis (FCA) is a branch of lattice theory that has been introduced by Wille [13]. It is an automated technique that aims to identify sensible groupings of objects (also called elements) that have common attributes (also called properties) [14].

To illustrate FCA, let us consider a toy example about musical preferences [15]. The objects are a group of people Alice, Bob, Carol, David, Emily, and Frank; and the properties are Rock, Pop,

Table I. Incidence table of the music example

prefers	Rock	Pop	Jazz	Folk	Tango
Alice	√	√		√	
Bob	√	√			√
Carol			√	√	
David			√		
Emily			√		
Frank			√	√	

Table II. The set of concepts of the example of Table I

top	({all objects}, ∅)
c_7	({Carol, David, Emily, Frank}, {Jazz})
c_6	({Alice, Carol, Frank}, {Folk})
c_5	({Alice, Bob}, {Rock, Pop})
c_4	({Carol, Frank}, {Jazz, Folk})
c_3	({Alice}, {Rock, Pop, Folk})
c_2	({Bob}, {Rock, Pop, Tango})
bottom	(∅, {all attributes})

Jazz, Folk, and Tango. Table I shows which people prefer which kind of music, called the *incidence table*. Formal concept analysis helps to find maximal groups of people sharing maximal sets of music preferences.

More formally, a *context* is a triple (O,A,R), consisting of a set of *objects* O, a set of *attributes* A, and an *incidence* relation $R \subseteq O \times A$ containing elements $(o,a) \in R$ indicating that object o has attribute a.

Let $X \subseteq O$ and $Y \subseteq A$. Then we can define $\sigma(X)$ as the set of common attributes for a set of objects X, and $\tau(Y)$ as a the set of common objects for attributes Y. Then a concept is a pair of sets — a set of elements (the *extent*) and a set of properties (the *intent*) *(X, Y)* — such that $Y = \sigma(X)$ and $X = \tau(Y)$. In other words, a concept is a maximal collection of elements sharing common properties.

With these definitions, we can obtain maximal rectangles from Table I with relations between people and musical preferences. For example, ({Alice, Bob}, {Rock, Pop}) is a concept, whereas ({David}, {Jazz}) is not, since $\sigma(\{David\}) = \{Jazz\}$, but $\tau(\{Jazz\}) = \{Carol, David, Emily, Frank\}$. The extent and intent of each concept is shown in Table II.

The set of all concepts consisting of sets objects O_1, O_2 and sets of attributes A_1, A_2 that can be derived from a context forms a partial order via

$$(O_1,A_1) \leq (O_2,A_2) \Longleftrightarrow O_1 \subseteq O_2 \Longleftrightarrow A_1 \supseteq A_2$$

This partial order allows us to organize the concepts in a lattice with meet $\wedge$ and join $\vee$ defined as

$$\begin{aligned} (O_1,A_1) \wedge (O_2,A_2) &= (O_1 \cap O_2, \sigma(O_1 \cap O_2)) \\ (O_1,A_1) \vee (O_2,A_2) &= (\tau(A_1 \cap A_2), A_1 \cap A_2) \end{aligned}$$

Once the context has been set up, efficient algorithms exist for computing the lattice [16]. The concept lattice shows the different concepts identified and the relations between them.

After the lattice has been constructed for a given context, concept *partitions* can be identified, which are collections of concepts of which the extents partition the set of objects. In our setting, each concepts partition corresponds to a possible modularization of the system analyzed. More formally, a *concept partition* is a set of concepts of which the extents are non-empty and form a partition of the set of objects O, given a context (O,A,R). This means that a set of concepts $CP = \{(X_0,Y_0) \ldots (X_n,Y_n)\}$ is a concept partition if and only if the extents of the concepts cover the object set and are pair wise disjoint [3, 16]:

$$\bigcup_{i=1}^{n} X_i = O \text{ and } \forall i \neq j,\ X_i \bigcap X_j = \emptyset$$

Tonella found concept partitions to introduce an overly restrictive constraint on concept extents by requiring that their union covers all the objects [3]. He argues that, when concepts are disregarded because they cannot be combined with other concepts to cover all objects, important information that was identified by concept analysis is lost without reason. As such, Tonella found that identifying meaningful organizations should not be limited by the unnecessary requirement that all objects are covered. Therefore, he proposes the idea of *concept subpartitions*. He defines that $CSP = \{(X_0, Y_0) \ldots (X_n, Y_n)\}$ is a *concept subpartition* if and only if [3]:

$$\forall i \neq j,\ X_i \bigcap X_j = \emptyset$$

Where CPs can be directly mapped to object partitions — that is partitions of the full object set — CSPs have to be extended to the object set by subtracting the subpartition from the full set, a process that is described by [3]. In our application of FCA we employ the algorithms proposed by Siff and Reps [16], using the concept partitions from Tonella [3]. Full details are available in [17].

We apply FCA by using the process presented by Siff and Reps, but instead of using the concept partition we use the concept subpartition as proposed by Tonella [16, 3]. More details on the process that we have followed can be found in [1].

3.2. Setting up FCA for the PMS repository

Having defined the process to use, we can define the objects and attributes to use in our specific context. As *objects* we choose the set of *building blocks* in the PMS repository, a set of size 360. The reason for this choice is twofold: (1) the building block level of abstraction is instigated by the domain experts from PMS, as they indicated that building blocks are designed to encapsulate particular functionality and (2) we expect to be able to cope with the size of the building block set for our analysis.

To complete the context, the set of attributes has to be defined. The set of attributes has to be chosen in such a way that building blocks that are highly related to each other appear in concepts of the context. In order to make sure that highly related building blocks appear in the same concept, we explicitly choose a combination of attributes that we consider to be good indicators of a building block:

1. whether it is *highly dependent* on another building block;
2. whether it has particular *features* associated to it.

We next discuss these attributes in some more detail.

High dependency attribute. The first type of attribute is extracted from the source code. We consider a building block *A* to be dependent on a building block *B* if *A* uses a function or data structure in *B*. The term 'highly dependent' is used to discriminate between the heavy use and occasional of a building block. As this first kind of attribute is collected from the source code, we can say that it is representative for the *'as-is'* architecture. We used the commercial tool Sotograph to extract the interdependencies of the building blocks in the architecture and subsequently determine the degree

Table III. Example context using project documentation

PMS context		Attributes from source code			Attributes from project documentation		
		coupledBB$_1$	...	coupledBB$_n$	murc	ecap	gvsa
Objects	BB$_1$	√					√
	BB$_2$	√					√
	BB$_3$			√	√		
	...						
	BB$_{n-1}$			√		√	
	BB$_n$	√		√	√		

of dependency between the building blocks [12]. Sotograph determines the degree of dependency by summing up static dependencies up to the desired abstraction level. A (lower-bound) threshold is used to filter relations on the degree of dependency.

Feature attribute. The second type of attribute is extracted from: existing architecture overviews and project documentation at PMS as well as from domain experts. As such, these attributes pertain to the *'as-built'* architecture. The particular properties that we use for this type of attribute are: specificity to the PMS application, layering and historical information about what building blocks were affected during prior software (maintenance) tasks. The features associated with the building blocks are discussed in more detail in Section 3.3.

The reasons to combine two sets of attributes are:

1. The first set of attributes assumes that building blocks that are highly dependent on each other should reside in the same repository.
2. The second set of attributes assumes that building blocks that share the same features, such as building blocks that are all very specific to the PMS application, should be grouped in the same repository.

As such, the two sets of attributes that form the attributes of the context are a combination of the 'as-is' architecture extracted from the source code and features extracted from the 'as-built' architecture, according to the documentation and the domain experts. Note that while the former is typically available in most circumstances, the latter might not always be available due to a lack of documentation or domain experts. Table III shows an example of this combination in the context, using existing documentation.

3.3. Feature attributes

As mentioned in the previous section we use two types of attributes. The first type of attribute indicates whether building blocks are highly dependent on other building blocks and is extracted from

source code. The second type of attribute takes into account several features of building blocks, more specifically:

- Information about which building blocks are affected during specific software maintenance operations.
- To which architectural layer a building block belongs and how application-specific the building block is[§].

In Sections 3.3.1 and 3.3.2 we take a closer look at how exactly the information on which building blocks are affected during specific maintenance operations and the information on the architectural layering come into play. Furthermore, because we expect that applying FCA using the two attribute-variants will provide different results, we will evaluate them individually in cooperation with the system architects.

3.3.1. Information extracted from project documentation

The first approach relies on the software's documentation. The specific type of documentation describes for each (sub)project which building blocks are in its scope, implying that the buildings blocks mentioned in the documentation are expected to change when a maintenance operations is carried out on that particular (sub)project. This scope is determined by the system architects prior to the start of the project. The scope can consist of building blocks that are scattered through the entire repository, but because projects are often used to implement certain functionality, there typically is an established relation between the building blocks in the scope.

This particular relation is used to group the building blocks together in the form of concepts after the construction of the context. The fact that this grouping possibly crosscuts the repository makes this feature interesting to use for FCA in combination with the high dependency relations between building blocks.

> *Example*: given a project that implements a certain feature, named 'projectA-feature1', there is documentation at PMS that describes that 'buildingblockA', 'buildingblockB' and 'buildingblockC' are within the scope of this project, which crosscuts the source code with respect to the building block hierarchy. Now the feature 'projectA-feature1' is assigned to each of the three building blocks in the scope.

When carrying out the experiment however, it became clear that not all building blocks were documented with the features they are implementing. As such, the features do not cover the complete object set of building blocks in the context. This has consequences for deducing concepts from a context with these features. The building block that has no features assigned to it, will be grouped based on the attributes that indicate high dependency on other building blocks. This high-dependency attribute however could also be missing, either because there are no dependencies from this building block to other building blocks or because the number of dependencies to another building block is below a chosen threshold. This is a factor that we should keep in mind when analyzing the results.

[§]Building blocks may be application-specific or can be shared with other medical equipment, such as echo-equipment.

While extracting information from the documentation we noticed differences in the level of detail of the documentation, that is, some project-scopes were defined in great detail with respect to the building blocks in the hierarchy, while others were only defined at a very high level of abstraction. For example, we encountered a scope in the documentation that was defined as a complete subsystem, without specifying specific building blocks. If we encountered such an instance, we substituted the subsystem with all the building blocks that are underlying to that subsystem. For example, when the project documentation states that the scope of 'projectA-feature1' is 'platform', all underlying building blocks in the building block structure of 'platform' are given the feature 'projectA-feature1', including 'platform' itself.

The basic idea of this approach is that building blocks will be grouped together based on whether they are related through certain features of the software that they implement. This grouping can be different from a grouping based on high dependencies between the building blocks and as such, we think it is interesting to use both types of features in the context for analysis, as a combination of the 'as-is' architecture and the 'as-built' architecture.

3.3.2. PMS-specificity and layering

The other approach is taking into account the PMS-specificity and layering of the entities in the repository.

PMS-specificity refers to the notion that some building blocks are only to be found in PMS software, while others are common in all medical scanner applications or even in other applications, such as database management entities or logging functionality. Domain experts at PMS assigned the features to the building blocks. This was done using a rough scale for the PMS-specificity: $\{$*very specific*, *specific*, *neutral*, *non-specific*, *very non-specific*$\}$.

With regard to the layering attribute, we use a designated scale for the building blocks that states whether a building block is at the 'service level' or at the 'application/UI level'. For example, a 'process dispatcher' is most likely to belong to the service level, while 'scan-define UI' is likely to be found at the application/UI level. For the layering a similar scale holds starting from application/UI level to the service level.

In our analysis, the complete object set of building blocks is covered, that is, each entity has a feature indicating the PMS-specificity and a feature indicating the layering. As such, for each building block there are $5 * 5 = 25$ possible combinations with respect to the PMS-specificity and layering.

We have chosen these specific features — PMS-specificity and layering — because of the wish of Philips Medical Systems to evolve to a more homogeneous organization in terms of software applications. As such, an interesting opportunity arises to consider reusing building blocks that are common in medical scanner software in other departments or develop maybe start developing building blocks together with other departments and use them as reusable building blocks.

Concept analysis using these feature attributes will find combinations such as a group of building blocks that are 'very PMS-specific' and are on the 'application/UI level'. We expect the resulting grouping to be different from the grouping based on the high dependencies between building blocks, and, as such, it is to contrast the obtained solution as we are looking at the results of the groupings obtained from the 'as-built' architecture versus the 'as-is' architecture.

Table IV. Example context, shown in the building block hierarchy

subsystem	**building block**	**attributes**
platform		*PMS-neutral*
	basicsw	*PMS-neutral, acqcontrol*
	computeros	*PMS-non-specific*
	configuration	*PMS-specific*
acquisition		*PMS-specific*
	acqcontrol	*PMS-specific, patientsupport*
	...	

4. A leveled approach to concept analysis

The process that we propose to obtain a splitting of the repository generates CSP-collections from the specified context. Considering the size of the application at hand, we expect that scalability issues come into play, because we use the set of building blocks in the repository as the set of objects in the context. The repository consists of around 360 building blocks, which results in a big context with the attributes defined, which in turn yields a large corresponding concept lattice, and many CSP-collections (which have to be processed by hand; to give an indication of size: some of our results generated millions of CSPs).

To cope with the large number of results we introduce a *leveled* approach, which we designed to make use of the hierarchical structuring of the PMS repository; the repository is modularized in high level 'subsystems', which consist of multiple 'building blocks', which again are structured in a hierarchy.

By analyzing parts of the hierarchy in detail, resulting concepts from that analysis are *merged* for the next analysis. This will make the context and concept lattice of the next analysis round smaller and we expect the resulting number of CSPs to also decrease. Through the use of the leveled approach some parts of the repository can be analyzed in detail, while keeping the other parts at a high level. The results from this analysis, such as groupings of 'lower level' building blocks, can be accumulated to a next analysis round where another part is analyzed in detail. These groupings are accumulated by merging the building blocks into a single fictive building block to make the context and resulting concept lattice smaller. This is repeated until all building blocks are analyzed in detail and the results are accumulated.

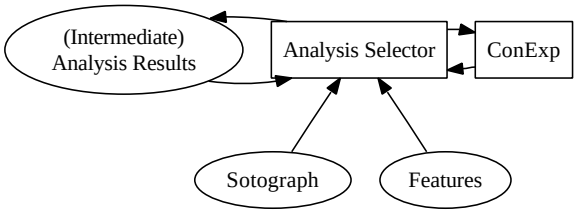
As an example of this accumulation, when a part of the hierarchy of the repository is not examined in detail, the attributes are accumulated to the entity that is examined globally. Table IV shows part of an example hierarchy and the assigned attributes. We then can decide that the 'platform-subsystem' in the hierarchy of the repository is analyzed globally and the others in detail. This results in Table V showing that all the features in the lower levels in the top Table IV of the 'platform-subsystem' are accumulated to 'platform'.

The analysis itself was performed using a newly developed tool — named *'Analysis Selector'* — that uses the output of the analysis performed by Sotograph, which recognizes the hierarchy of the repository and relations between building blocks (see Section 2). Further, separate input files are given

Table V. Accumulated features for the platform subsystem from Table IV

subsystem	building block	attributes
platform		*PMS-neutral.* *PMS-non-specific,* *PMS-specific, acqcontrol*
acquisition		*PMS-specific*
	acqcontrol	*PMS-specific, patientsupport*
	...	

Figure 1. Overview of the FCA process when using the leveled approach.



to the tool for the features, either extracted from the project planning or from PMS-specificity and layering documentation.

The tool enables the selection of parts of the hierarchy to analyze in detail and enables viewing the resulting concept subpartitions and merging resulting groupings in the concept subpartitions for further analysis. After selecting what part should be analyzed in detail and what parts should not be analyzed in detail the context is created using the accumulated attributes of the context.

This context can be exported to a format that an existing tool can use as input. For this study, we used 'ConExp' [18]; ConExp creates the concept lattice corresponding to a given context. This concept lattice can be exported again to serve as input for our Analysis Selector tool, which can deduce concept subpartitions from the concept lattice (also see Figure 1).

Merging concepts Considering the large number of CSPs that might result from concepts, the number of concepts taken into consideration should be kept small when calculating the concept subpartitions. This can be accomplished by *merging* the extents of the concepts (the object sets of the concepts) resulting from the context of an analysis (a process we call *concept merging*).

When a concept is merged, the objects of that concept will be grouped into a so-called 'merge' which is a fictive object with the same attributes as the original concept. It is expected that the context is now reduced in size for a successive analysis round and a fewer concepts will result from the next analysis round. This process of merging and recalculating the context and concepts can be continued until a small number of concepts result from the defined context. From these concepts then the CSPs can be calculated.

Concept quality In order to select concepts that should be merged, we propose a function to measure the *concept quality*. This function indicates how strong the grouping of the concept is, and is based on the relative number of attributes of a concept on the one hand, and on the relative number of objects on the other.

With respect to the number of attributes, recall that a concept includes a the maximal set of attributes that a group of objects share. Intuitively when few objects are grouped by many attributes, this indicates a strong grouping of these objects and therefore is assigned a high concept quality value. Conversely, when a small number of objects shares just a single attribute, this can be seen as weaker grouping, and therefore is assigned a lower quality value.

A similar degree of quality is devised for the number of objects. Given a set of attributes, when few objects share this set of attributes, this can intuitively be seen as a strong grouping, while a large number of objects sharing this set can be seen as a weaker grouping. Therefore, the quality function is also based on the degree of objects in a concept sharing a set of attributes.

Because a degree of attributes in a concept and a degree of objects in a concept is measured for each concept, a *ceiling value* is defined. As ceiling value the maximum number of objects and attributes respectively are taken given a set of concepts (called 'MaxObjects' and 'MaxAttributes', repsectively).

Thus, we define the *concept quality* ranging from 0 to 100 for a concept c as follows:

$$Quality(c) = \frac{MaxObjects - \#Objects}{MaxObjects} \times \frac{\#Attributes}{MaxAttributes} \times 100$$

Given this quality function all resulting concepts from a context are evaluated and based on the values, concepts are merged into single entities. These merges of building blocks are taken as one entity for the next round of analysis, with the purpose of decreasing the number of concepts.

5. Formal Concept Analysis Results

5.1. Project documentation features

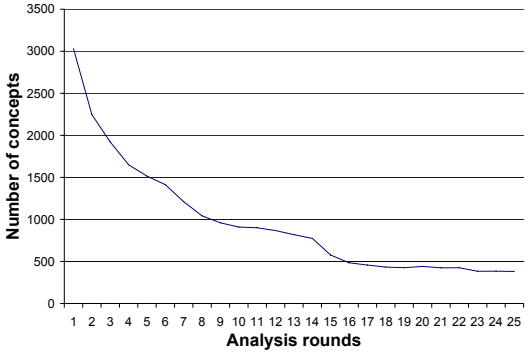
We first discuss how we use the information extracted from the project documentation (see Section 3.3.1) in combination with the dependency attribute that we extract from the source code. We perform our analysis on the complete building block structure, with no threshold imposed on the relations. This means that every static relation between building blocks is considered to be a high dependency attribute in the context. This is combined with the information extracted from the project documentation. Figure 2 shows the number of concepts plotted against the number of analysis rounds.

The full context results in 3,031 concepts. This number is too large for us to derive CSPs from. Figure 2 shows that the number of concepts decreases over the successive analysis rounds.

After 25 rounds the number of concepts has decreased to 379. However, calculating CSPs from this set of concepts still results in more than 10 million CSPs.

When discussing the resulting (intermediate) concepts with the domain experts, we noticed that the objects of the concepts were mainly grouped on the high dependency attributes in the context. This can be explained by the fact that the features extracted from existing project documentation covered around 30% of the complete set of around 360 building blocks.

Figure 2. Results of analysis with features extracted from project documentation



Thus, when grouping objects in the defined context by concepts the main factor of grouping is determined by the dependencies. The degree of influence of the high dependency attributes can be decreased by using a threshold on the strength of the static relations. This also means that there will be more objects in the context that have no attributes assigned to them. This in turn implies that no information is available on how to group these objects, resulting in objects that will not be grouped into meaningful concepts.

For this reason we have chosen not to continue the analysis with features extracted from project documentation and an imposed threshold. We also decided not to analyze the same setup using our leveled approach.

5.2. Analysis Based on PMS-specificity and layering features

Next we present the results from analysis on the context with the PMS-specificity and layering features (see Section 3.3.2) in combination with the dependencies extracted from source code.

Full building block structure. The first results of the analysis with the PMS-specificity and layering features are the results of the analysis of the complete hierarchy of building blocks, with no threshold imposed on the static dependencies (Figure 3).

The full context results in 3,011 concepts. Similar to the analysis on the project documentation features, this number of concepts is too large to derive CSPs from.

Thus, concepts were chosen to be merged each round. Figure 3 shows that the number of concepts decreases over the analysis rounds. After 27 rounds the number of concepts has decreased to 351. Calculating CSPs from this set of concepts resulted in more than 10 million CSPs.

As a next step we imposed a threshold of 25 on the static dependencies and proceeded with analyzing the complete hierarchy of building blocks with the PMS-specificity and layering features. This resulted in a full context containing 719 concepts, which is still too large a number to create CSPs from. Therefore, concepts were chosen to be merged each round. Figure 4 shows the decrease of the number

Figure 3. Results of the analysis with PMS-specificity and layering features

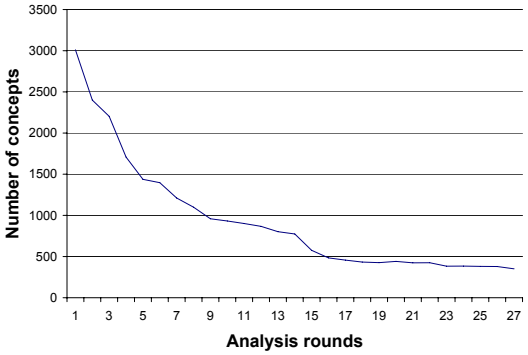
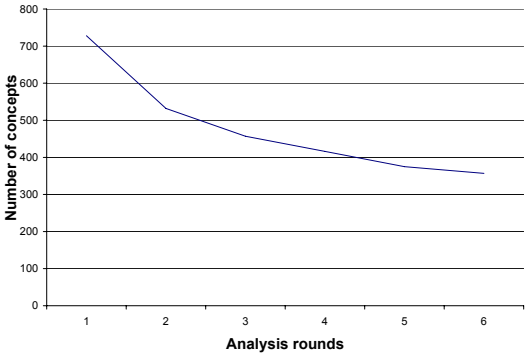


Figure 4. Results of the analysis with PMS-specificity and layering features, with imposed threshold of 25



of concepts over the successive analysis rounds. After 6 analysis rounds the number of concepts has decreased to 378. Calculating CSPs from this set of concepts still resulted in more than 10 million CSPs, making it difficult to define a splitting.

One subsystem in detail. Because of the inherent scalability issues that we encountered when analyzing the complete application, we decided to focus on a single subsystem. Figure 5 shows the results of this analysis on one subsystem, for which we used the PMS-specificity and layering features; the analysis has no threshold imposed on the static dependencies.

The aforementioned analysis results in 458 concepts in the context. By applying our leveled approach that merges concepts using the concept quality measure, we obtain 95 concepts after 30 analysis, which in turn yields 490,000 CSPs. Table VI shows the number of resulting CSPs from the last five analysis rounds.

Figure 5. Results of the analysis with PMS-specificity and layering features on one subsystem

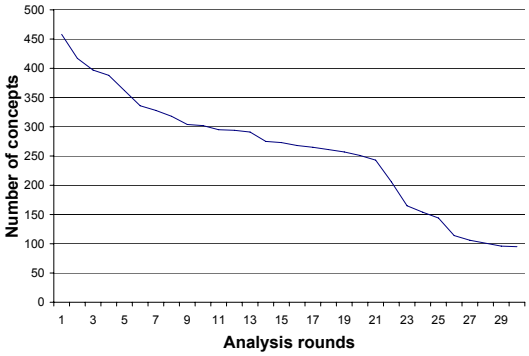


Table VI. Resulting concept subpartitions from the set of concepts of the analysis on one subsystem with no threshold

Round	Concepts	CSPs ($\times$1000)
26	114	600
27	106	500
28	101	555
29	96	500
30	95	490

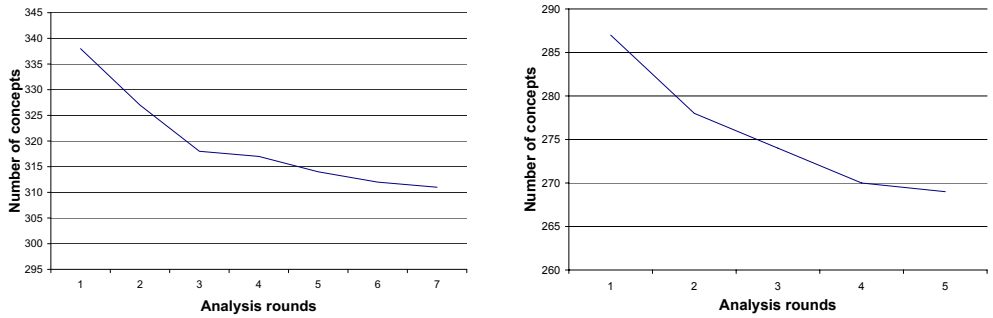
We also carried out the same basic analysis, but this time with an imposed threshold on the static relations. More specifically, we performed the analysis twice, once with a threshold of 25 and once with a threshold of 50. The results of this analysis are shown in Figure 6.

The analysis with an imposed threshold of 25 starts with a context that results in 338 concepts, whereas the analysis with a threshold of 50 starts with a context that results in 287 concepts. When performing the leveled approach, we see that in the case of the threshold of 25, we obtain 311 concepts after 7 analysis rounds. Similarly, for the threshold of 50, we obtain 269 concepts after 7 analysis rounds.

5.3. Discussion

In this experiment we used FCA as an analysis method to obtain a splitting of the repository for the PMS case. The process of analysis works with a leveled approach, i.e., some parts of the building block hierarchy in the repository are analyzed in detail, while other parts are analyzed at a higher level. Results from a previous analysis are used for a next analysis where more parts are analyzed in detail. Selecting which parts are used for successive analysis rounds is steered by the concept quality function.

Figure 6. Results of analysis with PMS-specificity and layering features on one subsystem, with imposed threshold of 25 (left) and 50 (right)



When we look at the results of both analyses (with the two types of features) on the complete building block structure, we can see that the number of concepts decreases by applying the leveled approach and the concept quality function. However, we had expected that by decreasing the number of concepts over the several analysis rounds, the number of concept subpartitions (CSPs) would also decrease.

The number of resulting CSPs actually decreases, e.g., when we consider both the analyses on the complete building block hierarchy. However, as each CSP represents a possible partitioning of the set of building blocks, the number of CSPs has to remain under control as each CSP has to be evaluated by domain experts. During our experiment, we have not been able to keep the number of CSPs at such a level that it would have been manageable for domain experts to evaluate each of the CSPs.

The huge number of CSPs can be explained by the degree of overlap of the objects in a resulting concept set. For example, when each combination of two concepts in this set of concepts has an intersection of their objects set that is empty, the number of CSPs from this set grows quickly.

Furthermore, we observed many concepts in the sets with a small number of objects. Concepts with a small number of building blocks as their object set are likely to be combined with other concepts, as the chance of an overlap with building blocks in the other concepts is small. More combinations of concepts result in more CSPs.

If we consider around 100 concepts resulting in around 500,000 CSPs, we do not expect to get significantly fewer concept subpartitions if we choose to use different attributes in the starting context, for example other features, or a more detailed scale for the PMS-specificity and layering features. This is inherent to the mathematical definition of the CSP, which enables small sets of concepts to result in many of CSPs.

In essence, we found that the leveled approach works, as evidenced by the decreasing number of concepts. However, we also found that obtaining partitions of the set of building blocks is still not possible. This is due to the large number of CSPs that need to be considered for a system of the size of our PMS case.

6. Cluster analysis

Our second series of experiments involves the use of *cluster analysis* to regroup building blocks contained in the PMS repository. The goal of software clustering is to partition the structure of a software system using the entities and relations that are specified in the source code into a set of cohesive clusters, which (will) typically form the subsystems [19].

6.1. Search-Based Software Modularization

A range of different clustering algorithms have been proposed in literature. For our experiments, we decided to use the search-based algorithm as implemented in the Bunch tool [10, 20], which has been specifically designed for remodularization purposes, and which has been used in several other remodularization cases [21, 20]. Furthermore, performance has been a key driver in the design of Bunch, and Mitchell and Mancoridis claim that relatively large systems can be clustered in about 30 seconds [20].

Bunch works independently of the programming language and relies on source code analysis tools to build up a graph representation of the source code, the so-called *Module Dependency Graph* (MDG) [20]). Like in the concept analysis case, in our setting the modules are the PMS building blocks; the dependencies between them reflect usage of functions or datastructures as obtained by Sotograph (see Section 3.2).

Addressing the modularization problem using search techniques has been described as heuristically "finding a good partitioning of the MDG", where "good" refers to a partition where highly interdependent modules (nodes) are grouped in the same subsystems (clusters), and, conversely, independent modules are assigned to separate subsystems [20]. This can be formalized by means of an objective function, called the *Modularization Quality* (MQ) function, which determines the quality of a partition as the trade-off between interconnectivity and intraconnectivity. Various definitions of MQ can be used, which are described by Mitchell and Mancoridis [20], and which are implemented in the Bunch tool.

With a way of measuring the quality of resulting modularizations, we adopt a hill climbing algorithm to search for partitions that are of good quality. Bunch starts with a random partitioning of the MDG. Nodes in this partition are then systematically rearranged in an attempt to find an improved with a higher MQ-value. Furthermore, Bunch adopts simulated annealing to mitigate the risk that local optima prohibit searching for further alternatives [20].

6.2. Leveraging Domain Knowledge

While Bunch can operate in a fully automatic way, it also allows to perform "user-directed clustering". With this feature enabled, the user is able to cluster some modules manually, using domain knowledge, while still taking advantage of the automatic clustering capabilities of Bunch to organize the remaining modules [11].

We propose to use this mechanism to iteratively improve the starting point of the genetic algorithm by using the suboptimal solution of the algorithm and by reviewing the solution with domain experts, such as architects and developers of the PMS department. The process can be described as follows:

1. Obtain the Module Dependency Graph (MDG) of the source code using the commercial tool Sotograph.
2. Use the algorithm provided by the tool Bunch to obtain a clustering of the MDG.
3. Review the results with domain experts by visualizing the results (see Section 6.4).
4. Use the new knowledge thus obtained as a starting solution for the genetic algorithm provided by Bunch.
5. Repeat steps 2 until 4 until satisfying results are obtained, validated by the domain experts and the value of the MQ function.

6.3. Scalability issues and the leveled approach

Initial experiments when applying the process proposed in the previous section have shown that there are scalability issues with this approach. To be more precise, the search space for the analysis is extremely large, as the number of possible clusterings grows exponentially with the number of building blocks. As such, a first run clustering of the building blocks will probably result in a clustering from which domain experts cannot extract usable information.

Therefore, we again propose to use a *leveled approach*, in similar fashion to what we did for FCA (see Section 4). The leveled approach makes use of the hierarchical structure of the source code, which is composed of multiple 'subsystems', which in turn consist of multiple 'building blocks'.

With the leveled approach, certain parts of the repository can be analyzed in detail while other parts of the repository are analyzed at a higher level. Results from the analysis of one part of the repository can be used for a next iteration of the analysis. For example, one subsystem can be analyzed in detail, while other subsystems are analyzed globally. When a subsystem is analyzed globally, the relations from all underlying building blocks in the hierarchy are accumulated to be relations from the higher level subsystem. Also relations from one building block to the underlying building blocks in the hierarchy are adjusted to the higher level subsystem.

When after an analysis round some building blocks are clustered together, these building blocks can be *merged* into one entity, called a 'merge'. This merge is in fact a fictive building block. The relations are accumulated to the merge as mentioned above.

An example of a merge is shown in Table VII. Here the building blocks 'platform', 'basicsw', 'computeros' and 'configuration' are merged into one new merge, named 'merge'. Table VII also shows that the relations of the named building blocks are accumulated and designated to the new building block 'merge'. Also the relations *to* the building blocks are changed to relations to the merge.

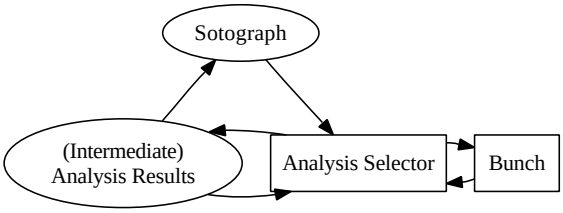
The analysis itself is performed using the newly developed tool 'Analysis Selector', which we also used to enable the leveled approach for FCA. The tool uses the output of the static dependency analysis of Sotograph, from which we can also extract the hierarchy of the building blocks of the PMS application.

The tool enables the selection of parts of the repository to be analyzed in detail, while other parts are kept at a higher level. When the user has made his selection the necessary relation accumulation that is a consequence of the merge is performed and the Module Dependency Graph (MDG) is adjusted accordingly. This new MDG is then used as input for Bunch, which performs the actual clustering.

Table VII. Before/after example of a merge of a group of building blocks.

Subsystem	Building block	Relation (strength)
platform		
	basicsw	acquisition (5)
	computeros	acquisition (5)
	configuration	acqcontrol (2)
acquisition		
	acqcontrol	basicsw (5)
	...	
merge		acquisition (10) , acqcontrol (2)
acquisition		
	acqcontrol	*merge* (5)
	...	

Figure 7. Overview of the process for the leveled approach.



The results of this clustering can be imported into the Sotograph tool using a plug-in that we have written. Sotograph then visualizes the results and provides metrics on the clustering result. More information about this can be found in section 6.4. Using Sotograph, domain experts can now select certain parts of the clustering to be merged for a next analysis step. The parts can be selected in the Analysis Selector, after which a next round of analysis can start. Figure 7 shows how this works.

6.4. Visualization and validation

Reviewing the result of the clustering with domain experts can be aided by visualizing the results. Early results were visualized with a tool called ‘dotty’, which displays the clustering of the modules and the dependencies between the modules [22]. However, we have chosen to use Sotograph [12] as a visualization tool, because we found the visualization capabilities to be superior and because Sotograph offers several useful metrics for studying the clusterings, such as LOC of clusters and dependency metrics. Therefore, we developed a plug-in for Sotograph that allows to load in our (intermediate) analysis results.

Table VIII. Results of analysis on complete hierarchy

Result	Number of clusters	Objective Function Value (MQ)
1	84	4.034784274
2	39	2.574595769
3	40	3.970811187

Sotograph maintains a database of information extracted from the source code using static analysis. This information includes the dependencies between source code entities, size of entities and many other metrics. The tool enables the user to visualize the clustering, get more detailed information on the dependencies between the clusters and within the clusters, see what entities are placed in a particular cluster and retrieve the sizes of the clusters.

Results of the cluster analysis are evaluated by the domain experts at PMS in Sotograph and building blocks are selected as merges for a next analysis round, as discussed in the previous Section 6.3.

7. Results of the cluster analysis approach

This section presents the results of several variations of cluster analysis that we performed on the building block hierarchy of the PMS software. We start with a naive setup for the experiment, where we just provide the nodes and edges as Sotograph extracted them. We then apply our leveled approach to cluster one subsystem in detail, after which we start working with improved start solutions for the clustering process, based on prior knowledge. In total we investigate 5 solutions in Sections 7.1 through 7.5. For each analysis that we perform, we discuss the scope of the analysis and present the evaluation of the domain experts. In Section 7.6 we evaluate our approach.

7.1. Complete building block hierarchy

Cluster analysis was performed on the complete building block hierarchy. The input file given to Bunch consisted of 369 nodes (building blocks) and 2232 edges (weighted dependencies). We applied Bunch three times, because the genetic algorithm does not guarantee the same results for each run. Table VIII shows the results for this analysis.

As can be seen in Table VIII the MQ values do not differ significantly, indicating that each clustering is of almost equal quality. We therefore visualized all three results in Sotograph and subsequently, all three solutions were evaluated by domain experts at PMS. The domain experts however could not extract clusters of building blocks from the three clusterings to merge for a next analysis round. The reasons for that include:

- The three executions of Bunch result in three completely different clusterings, which made them hard to compare.
- The resulting clusters differed significantly in size (LOC), up to a factor 20,000.
- On average around 25% of the clusters contained one (low-level) building block.

Table IX. Results of analysis on one subsystem only

Result	Number of clusters	Objective Function Value (MQ)
1	12	3.906895931
2	13	5.343310963
3	12	4.470282759

Table X. Results of analysis with one subsystem in detail

Result	Number of clusters	Objective Function Value (MQ)
1	13	3.782851141
2	19	4.331546729
3	15	3.921600527

- For some of the resulting clusters dependencies between building blocks within that cluster were not evident or were not there at all.

7.2. One subsystem in detail only

This analysis was performed on one subsystem only, namely `platform`, of which all the building blocks were analyzed in detail. The input file given to Bunch consisted of 121 building blocks as nodes and 409 weighted dependencies as edges. Table IX shows the results of three executions of the genetic algorithm in Bunch.

All the three results were visualized in Sotograph. Domain experts have evaluated all three clusterings. Domain experts at PMS could not extract groupings of building blocks from the three clusterings for a next analysis round. The reasons for that include:

- The three executions of Bunch result in three completely different clusterings, which made them hard to compare.
- A big difference in size of the clusters with respect to LOC, up to a factor 100.
- For some clusters dependencies between building blocks placed in the clusters were not evident or were not there at all.

7.3. One subsystem in detail – others at higher level

This analysis was performed on the hierarchy with one subsystem in detail (platform), while keeping the others at the higher level. The input file given to Bunch consisted of 138 nodes (building blocks) and 655 edges (weighted dependencies). Table X shows the results of three executions of the genetic algorithm in Bunch.

All three results were visualized using Sotograph, after which domain experts have evaluated the proposed clusterings. However, the domain experts at PMS could not extract groupings of building blocks from the three clusterings for a next analysis round. The reasons for that include:

Table XI. Results of analysis with one subsystem in detail - improved start solution

Result	Number of clusters	Objective Function Value (MQ)
1	31	3.865839028
2	29	3.493834319
3	27	2.750577642

- The domain experts had difficulty understanding the presence of multiple subsystems in one particular cluster in the results. This cluster contained about 40% of the total size in lines of code of the entire repository.
- A big difference in size of the clusters with respect to LOC, up to a factor 200.
- On average (over the three runs) 20% of the clusters contained one (low-level) building block.
- For some clusters the dependencies between building blocks placed in these clusters were not evident or were not there at all.

7.4. One subsystem in detail – improved start solution

This analysis was performed on the building block hierarchy with one subsystem in detail, namely the *platform* subsystem, while the other subsystems were analyzed at a higher level. Each of these higher level subsystems were placed in a cluster to start with, which Bunch enables through the so called 'user-directed' clustering input file. The basic idea behind this setup is that all the building blocks of the platform subsystem will be divided among the other subsystems.

The input file for Bunch consisted of 138 building blocks as nodes and 655 weighted dependencies as edges; a configuration-file assigned all higher level subsystems to a cluster. Table XI shows the results of three executions of the genetic algorithm in Bunch.

Domain experts have evaluated the three results. The following points were observed:

- The three executions of Bunch result in three completely different clusterings, which made them hard to compare.
- On average (over the three runs) 20% of the clusters contained one (low-level) building block.
- For some clusters dependencies between building blocks placed in the clusters were not evident or were not there at all.
- The third clustering has a nice distribution of the subsystems over the clusters; there are no multiple subsystems present in a cluster.

Having evaluated the three results, the domain experts recommended to take the results from the third clustering for a next analysis round. In agreement with the domain experts, we decided to take all clusters containing more than 3 building blocks as a merge for the next analysis. The remaining clusters were seen by the domain experts as not logical and therefore the building blocks contained in these clusters are used as single entities for the second analysis round. Table XII shows the results of

Table XII. Results of analysis with one subsystem in detail - improved start solution, round 2

Result	Number of clusters	Objective Function Value (MQ)
1	12	2.132841135

Table XIII. Results of analysis with two predefined large clusters

Result	Number of clusters	Objective Function Value (MQ)
1	29	2.022904392

one execution of the analysis. We have chosen to perform only one execution, because of lack of time of the domain experts to review three new executions for this setup.

The input file for Bunch consisted of 42 nodes and 226 edges. When visualizing the results of the analysis in Sotograph the domain experts observed that there was one huge cluster containing around 40% of the lines of code of the entire repository. This is not desirable for PMS and therefore we decided not to continue with the analysis of one subsystem, while keeping the other subsystems at the highest level.

7.5. Two predefined large clusters – improved start solution

We performed another cluster analysis on the building block hierarchy, but this time we had two predefined large clusters as start solution. We prepare this start solution with the two predefined clusters through the merging of the building blocks in the clusters before executing the analysis. The two clusters were defined by domain experts: one cluster contained mainly image handling software and the other contained mainly scanning software. The first cluster contains 2 subsystems from the building block hierarchy and the latter cluster contains 11 subsystems from the building block hierarchy. The remaining 2 subsystems were analyzed in detail.

As a start solution there is already a ‘bipartitioning’ of the source code repository. The idea behind the analysis is to see how the remaining building blocks are divided among the two partitions, or see how the remaining building blocks are grouped in new partitions. Table XIII shows the results of one execution of the genetic algorithm.

The input file given to Bunch consisted of 129 building blocks as nodes and 485 weighted dependencies as edges. Domain experts observed the following items:

- A few building blocks were divided among the two predefined clusters.
- The majority of the building blocks were clustered in 27 clusters.
- For some of the 27 clusters dependencies between building blocks placed in the clusters were not evident or were not there at all.

Together with the domain experts we decided to continue with the analysis, by merging the building blocks that were clustered with the two predefined clusters. The remaining building blocks (in the 27

Table XIV. Results of analysis with two predefined large clusters, round 2

Result	Number of clusters	Objective Function Value (MQ)
1	20	3.069774546

other clusters) were not merged and are used as single entity for the next analysis round. Table XIV shows the results of one analysis with the genetic algorithm in Bunch.

The input file given to Bunch consisted of 109 building blocks as nodes and 385 weighted dependencies as edges. Domain experts observed the following items:

- A few building blocks were divided among the two predefined clusters.
- The majority of the building blocks were clustered in 18 clusters.
- For some of the 18 clusters dependencies between building blocks placed in the clusters were not evident or were not there at all.

Because the building blocks that were clustered with the two predefined clusters had very low dependencies or even no dependencies on the other building blocks in the clusters, we decided that continuing with this analysis — by merging the added building blocks to the two predefined clusters — was not useful. Therefore we decided not to continue with the analysis on the two predefined clusters.

7.6. Discussion

This section investigated whether cluster analysis allows us to obtain multiple independent components from the original source code repository of the PMS case. We use building blocks as entities for analysis, static dependencies as similarity measure and a genetic algorithm to execute the actual analysis. We performed the analysis using the leveled approach on several parts of the source code repository. This means that some parts of the repository are analyzed in detail, while other parts are analyzed at a higher level.

Domain experts from Philips Medical Systems evaluated the (intermediate) results that we obtained by visualizing these results in Sotograph. However, the domain experts could not identify groupings of building blocks from clusterings that could serve as ‘merges’ for a next analysis because of the following reasons:

- Different executions of the algorithm result in different clusterings, which makes it hard to compare the results. This is similar to the stability criterion that Wu et al. introduced for clusterings [23].
- Multiple clusters contain one building block, i.e., the extremity of the cluster distributions is high [23].
- Multiple clusters contain building blocks that are not related according to the domain experts, i.e., the authoritativeness of the result is low [23].

The intermediate results of the analysis were often found to be 'non-acceptable' for the domain experts. A direct consequence then is that the leveled approach does not work, as this approach relies on previous analysis rounds which need to be acceptable to some degree to continue with the analysis.

The fact that the intermediate results could — in most cases — not be used by the domain experts is also caused by the genetic algorithm of Bunch. The algorithm produces results with clusters that highly differ in size (LOC). The system's architects found that this is not desirable for the PMS case. 'Cluster size equality', or in other words a small extremity [23], is a requirement that was not made explicit in the beginning of this research, but during the evaluations with the domain experts, it became clear that this requirement is very important to the domain experts. In some cases the genetic algorithm also produced clustering results with one cluster containing up to 40% of the original size of the repository, which was deemed undesirable by the domain experts.

Also of importance to note for the practical applicability of the cluster analysis approach is the fact that it takes a long time to perform the actual clustering for a system of the size of PMS. Some of the clusterings that we have performed have taken 10 hours or more, with some clusterings taking more than 24 hours. Another factor to take into account is that having domain experts evaluate a potential remodularization can be costly. In our case, evaluating the usefulness of a remodularization done by cluster analysis takes at least 1/2 day.

Using Bunch also caused some problems. More specifically, providing a pre-defined starting solution to Bunch proved problematic, because Bunch sometimes tears apart pre-defined clusters and reclusters the elements contained in these clusters. Such a situation is particularly undesirable when domain experts are able to provide an initial clustering.

8. Related work

The remodularization of (legacy) applications is an active area of research, in which both clustering and concept analysis are in use. However, most of the experiments described in literature report on systems that are (significantly) smaller than our PMS setting.

Snelting provides a comprehensive overview of applications of FCA to software (re)engineering problems in general, and remodularization in particular [5]. Siff and Reps report on a method that uses FCA to identify modules in legacy C code to improve the structure of the source code. Their eventual aim is to migrate procedural code to object-oriented code by turning modules into classes [16].

Tilley et al. present an overview of academic papers that report the application of formal concept analysis to support software engineering activities [6]. They conclude that the majority of work has been in the areas of detailed design and software maintenance.

Hutchens and Basili identify potential *modules* by clustering on data-bindings between procedures [7]. Schwanke also identifies potential modules but clusters call dependencies between procedures and shared features of procedures to come to this abstraction [8].

In a Cobol context, Van Deursen and Kuipers identify potential objects by clustering highly dependent data records fields in Cobol. They apply cluster analysis to the usage of record fields, assuming that record fields that are related in the implementation are also related in the application domain and therefore should reside in an object [9].

Mancoridis et al., creators of the Bunch clustering tool we used during our experiments, identify high-level system organizations by clustering modules using the dependencies between these

modules [10, 11, 24]. Anquetil and Lethbridge on the other hand use file names for creating an abstraction of the architecture using clustering [25]. They also provide an overview of a range of issues to tackle when adopting cluster analysis, and illustrate the effects on several open source systems as well as a system comprising two million lines of code [26].

Wierda et al. use clustering to group classes of an existing object-oriented system of significant size into subsystems, based on structural relations between the classes. They show with a case study on a system consisting of around 2,700 classes that cluster analysis was applicable in practice for the given problem size using a hill-climbing algorithm combined with simulated annealing [21].

Schwanke's tool Arch implements a heuristic semi-automatic approach to software clustering [8]. Similar to Bunch, Schwanke's heuristics try to maximize the cohesion of procedures placed in the same module, while minimizing coupling between procedures that reside in different modules.

Wu et al. offer a comparison of six clustering algorithms, including the Bunch tool we used [23]. Their three primary criteria are stability (which indicates whether a small change in the system results in a small change in the clustering), authoritativeness (how close is the clustering result to an authoritative result) and extremity of the cluster distribution (does the clustering algorithm avoid many large and many small clusters). Bunch did not score well on the stability criterion, but it scored the best in the extremity and authoritativeness test.

Maqbool and Babri provide an assessment of the behavior of various similarity and distance measures that may be employed during software clustering, which they applied to legacy systems comprising thirty to seventy thousand lines of code [27].

Andreopoulos et al. propose a (layered) clustering approach that besides static properties makes use of dynamic information, such as the number of function invocations during run time [28]. They applied their approach to the Mozilla browser, which comprises four million lines of code.

Recently, Adnan et al. used clustering to semi-automatically regroup the interface definitions of a large scale industrial software system [29]. The aim of their regrouping process is to increase the coherence within an interface and to reduce the build time when a modification to an interface is made. While their dependencies are obtained from a multi-million system written in C, their regrouping is done within particular subsystems, which is how they deal with the scalability problem. They also aim at around seven clusters (interfaces) per subsystem. Their approach is to automatically cluster until around 20 clusters appear, after which the remaining clusters are merged by hand. In this way they have successfully approached remodularizations proposed by system experts entirely by hand.

From a broader perspective, remodularization can be considered as a software architecture reconstruction activity [30]: remodularization searches for a view on the current module structure, in which the actual (as implemented) dependencies are reflected. Besides concept and cluster analysis, other techniques have been proposed, involving the analysis of design decisions [31], the identification of change coupling in different versions included in a repository [32], or the use of information retrieval methods [33]. It remains future work to see whether and how such techniques can be applied to systems of similar size as our PMS case.

9. Conclusion

In this paper we have presented our experiences with using formal concept analysis (FCA) and cluster analysis to remodularize a large software repository from a medical diagnostic imaging product from Philips Medical Systems. In that context, we have made the following contributions:

- We discussed how FCA can be applied to a large-scale, non-trivial industrial software repository. In literature we could not find cases with comparable goals and scale.
- We discussed how cluster analysis can be applied to that same large scale software repository. Again, we did not find any papers describing cases with comparable goals and scale.
- We presented the *leveled approach*, which makes use of the existing hierarchy in the software repository, in order to cope with scalability issues that arise.
- We presented the *concept quality*, a measure that aids in choosing the optimal concepts of a set of concepts to consider in a next analysis round when applying the leveled approach in combination with formal concept analysis.

FCA provides ways to identify sensible groupings of objects that have common attributes. In the context of our case study, we used building blocks as objects and for the attributes we combined two sets: a set of attributes indicating that building blocks are highly dependent on each other (information extracted from source code) and a set of attributes that represent certain features of the building blocks (information obtained from documentation and domain experts).

We furthermore used a leveled approach that allows to analyze some parts of the software at a higher level. Results from previous analyses are used for successive analysis rounds, where more parts are analyzed in detail. The leveled approach is supported by the concept quality measure, which selects what parts should ideally be analyzed in detail in a successive analysis round.

When evaluating the results together with the domain experts at PMS, we see that the idea of applying the leveled approach in combination with the concept quality works reasonably well, as we are able to decrease the number of concepts significantly. However, the resulting number of concept subpartitions (CSPs) and by extension the possible number of remodularizations remains enormous. Therefore, we have to conclude that applying FCA for remodularizing within our industrial is not feasible in practice.

Based on our findings, it is our our opinion that FCA is very appropriate for recognizing patterns in or groupings of objects based on attributes, but FCA is not very well suited for an analysis that should result in a precise non-overlapping partitioning of the object set.

We have also investigated an alternative approach for remodularizing the PMS software repository, namely through cluster analysis. For this analysis, we chose building blocks as the entities to cluster, the strength of static dependencies as similarity measure and a genetic algorithm to evaluate the clustering quality. We again used a leveled approach, similar to the one we used for the formal concept analysis approach. As a basis for our approach, we used the Bunch tool.

During the evaluation of (intermediate) results with the domain experts however, a number of shortcomings to the clustering approach were identified. More specifically, the fact that the proposed clusters differed significantly in terms of size was seen as a stumbling point, i.e., the so-called

extremity of the proposed clustering. The domain experts were also not convinced by some of the proposed clusterings as the proposed clusters contained building blocks that are unrelated in their experience. This is directly related to the authoritativeness of the clustering approach. Finally, applying the clustering algorithm multiple times resulted in different clustering results, which was due to the genetic algorithm that we used. Unfortunately, this also contributed to the fact that the domain experts found it hard to start using our cluster analysis approach to start remodularizing the current software repository.

When we come back to our initial research question of whether formal concept analysis and cluster analysis are suited for splitting a large-scale software repository, we have to conclude that for the approach and the experimental setup that we chose, neither formal concept analysis nor cluster analysis seems to be able to cope with the scale of the system under analysis or the constraints of the domain experts. Nevertheless, we are still convinced that both techniques have their merit for solving similar, yet smaller-scale challenges.

Lessons learned. Our study has given us insight into a number of issues when trying to remodularize a large-scale software system with the help of formal concept analysis or cluster analysis. The most important issues are discussed below.

- When domain knowledge is available and you want to use that domain knowledge to define a rough initial clustering, make sure that your clustering tool (1) supports a pre-defined starting solution and (2) does not tear this initial clustering apart. Unfortunately, Bunch failed for the second criterion.
- When considering the use of formal concept analysis, an initial assessment of the feasibility of the approach can be made, by evaluating the degree of overlap of the objects in the concept set. If this overlap is minimal, the number of CSPs — and possible modularizations — grows quickly.

Future work. Currently, the Software Architecture Team (SWAT) at PMS is undertaking a remodularization of the PMS application using both structural information from the source code and domain knowledge from the software architects, without any specific tool support for proposing clusterings. However, they do see the fact that domain knowledge of some parts of the system is not complete as a major hurdle and, as such, automated tools could still play an important role in the future as well. In this light, we have established a number of future research directions that we would like to investigate:

- Using the results of formal concept analysis as a starting solution for the clustering analysis.
- Switch clustering tools, because Bunch has difficulties to deal with pre-defined starting solutions. More specifically, we have experienced that clusters that are part of the pre-defined starting solution are subdivided and reclustered, a situation that is often undesirable.
- Using other approaches such as information retrieval methods, the identification of change coupling over different versions and the analysis of previous design decisions.

ACKNOWLEDGEMENTS

This work could not have been carried out without the support of many colleagues at Philips Medical Systems. This work is sponsored by the NWO Jacquard Reconstructor research project.

REFERENCES

1. Glorie M, Zaidman A, Hofland L, van Deursen A. Splitting a large software archive for easing future software evolution an industrial experience report using formal concept analysis. *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2008; 153–162.
2. Arévalo G, Ducasse S, Nierstrasz O. *Formal Concept Analysis*, *LNCS*, vol. 3403/2005, chap. Lessons Learned in Applying Formal Concept Analysis to Reverse Engineering. Springer: Springer Berlin / Heidelberg, 2005; 95–112.
3. Tonella P. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering* 2001; **27**(4):351–363.
4. Antoniol G, Di Penta M, Casazza G, Merlo E. A method to re-organize legacy systems via concept analysis. *Proceedings of the International Workshop on Program Comprehension (IWPC)*, IEEE Computer Society: Washington, DC, USA, 2001; 281–292.
5. Snelting G. Software reengineering based on concept lattices. *Proceedings of the Conference Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Washington, DC, USA, 2000; 3–10.
6. Tilley T, Cole R, Becker P, Eklund P. A survey of formal concept analysis support for software engineering activities. *Formal Concept Analysis*, *LNCS*, vol. 3626/2005, Ganter B, Stumme G, Wille R (eds.), Springer: Springer Berlin / Heidelberg, 2005; 250–271.
7. Hutchens DH, Basili VR. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering* 1985; **11**(8):749–757.
8. Schwanke RW. An intelligent tool for re-engineering software modularity. *Proceedings of the International Conference on Software engineering (ICSE)*, IEEE Computer Society: Los Alamitos, CA, USA, 1991; 83–92.
9. van Deursen A, Kuipers T. Identifying objects using cluster and concept analysis. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society: Los Alamitos, CA, USA, 1999; 246–255.
10. Mancoridis S, Mitchell B, Rorres C, Chen Y, Gansner E. Using automatic clustering to produce high-level system organizations of source code. *Proceedings of the International Workshop on Program Comprehension (IWPC)*, IEEE Computer Society: Washington, DC, USA, 1998; 45–52.
11. Mancoridis S, Mitchell B, Chen Y, Gansner E. Bunch: A clustering tool for the recovery and maintenance of software system structures. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE: Washington, DC, USA, 1999; 50–59.
12. Sotograph. http://www.hello2morrow.com/products/sotograph. Last visisted on: January 14th, 2009.
13. Wille R. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered sets*, Rival I (ed.), Reidel: Dordrecht, The Netherlands, 1982; 445–470.
14. Ganter B, Wille R. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag: Berlin, 1997.
15. Arévalo G, Ducasse S, Nierstrasz O. Discovering unanticipated dependency schemas in class hierarchies. *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Washington, DC, USA, 2005; 62–71.
16. Siff M, Reps T. Identifying modules via concept analysis. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society: Washington, DC, USA, 1997; 170–179.
17. Glorie M. Philips medical archive splitting. Master's Thesis, Software Engineering Research Group, Delft University of Technology 2007.
18. ConExp. http://sourceforge.net/projects/conexp. Last visited on: January 14th, 2009.
19. Mitchell BS, Mancoridis S. On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing - A Fusion of Foundations, Methodologies and Applications* 2008; **12**(1):77–93.
20. Mitchell BS, Mancoridis S. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.* 2006; **32**(3):193–208, doi:http://dx.doi.org/10.1109/TSE.2006.31.
21. Wierda A, Dortmans E, Somers LL. Using version information in architectural clustering - a case study. *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Washington, DC, USA, 2006; 214–228.
22. North SC, Koutsofios E. Application of graph visualization. *Proceedings of Graphics Interface '94*, Canadian Information Processing Society: Banff, Alberta, Canada, 1994; 235–245.
23. Wu J, Hassan AE, Holt RC. Comparison of clustering algorithms in the context of software evolution. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society: Washington, DC, USA, 2005; 525–535.

24. Mitchell BS, Mancoridis S. Comparing the decompositions produced by software clustering algorithms using similarity measurements. *International Conference on Software Maintenance (ICSM)*, IEEE Computer Society: Washington, DC, USA, 2001; 744–753.
25. Anquetil N, Lethbridge TC. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice* 1999; **11**(3):201–221.
26. Anquetil N, Lethbridge TC. Experiments with clustering as a software remodularization method. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society: Washington, DC, USA, 1999; 235–255.
27. Maqbool O, Babri HA. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering* November 2007; **33**(11):759–780.
28. Andreopoulos B, An A, Tzerpos V, Wang X. Clustering large software systems at multiple layers. *Inf. Softw. Technol.* 2007; **49**(3):244–254.
29. Adnan R, Graaf B, van Deursen A, Zonneveld J. Using cluster analysis to improve the design of component interfaces. *Proceedings International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society: Washington, DC, USA, 2008; 383–386.
30. van Deursen A, Hofmeister C, Koschke R, Moonen L, Riva C. Symphony: View-driven software architecture reconstruction. *Proceedings of the IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE Computer Society: Washington, DC, USA, 2004; 122–132.
31. Jansen A, Bosch J, Avgeriou P. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software* 2008; **81**(4):536–557.
32. Zhou Y, Würsch M, Giger E, Gall H, Lü J. A bayesian network based approach for change coupling prediction. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society: Washington, DC, USA, 2008; 27–36.
33. Kuhn A, Ducasse S, Gîrba T. Semantic clustering: Identifying topics in source code. *Information & Software Technology* 2007; **49**(3):230–243.

TUD-SERG-2009-002
ISSN 1872-5392