Splitting a Large Software Archive for Easing Future Software Evolution — An Industrial Experience Report using Formal Concept Analysis —

Marco Glorie[1], Andy Zaidman*[2], Lennart Hofland[1], and Arie van Deursen[3]

[1]Philips Medical Systems – marco.glorie@gmail.com, lennart.hofland@philips.com
[2]Delft University of Technology – a.e.zaidman@tudelft.nl
[3]Delft University of Technology & CWI – arie.vandeursen@tudelft.nl

Abstract

Philips Medical Systems produces medical diagnostic imaging products, such as MR, X-ray and CT scanners. The software of these devices is complex, has been evolving for several decades and is currently a multi-MLOC monolithic software archive. In this paper we report on splitting a single software archive into multiple smaller archives so that these can be developed independently, easing the software's evolution. To determine how to split such a single software archive we use formal concept analysis. Because of the sheer size of the monolithic software archive, we also propose to use a 'leveled approach'. This leveled approach 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. We conclude this paper with an evaluation of the used analysis method in this industrial context.

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) scanners. The software for these products is very complex and has been evolving for decades. The systems are a combination of hardware and software and contains (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), at which more than 100 developers are currently working on the software.

In this study we will focus on the software archive of one of the aforementioned medical diagnostic imaging products[1] has a multi-MLOC software archive consisting of approximately 30,000 source code files that is being developed (and maintained) using branching. Merging the multiple development branches causes integration problems and the many dependencies among the components make that the feature that has the longest development time determines the release time of the entire project.

This way of developing is the result of many years of software evolution 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[2] software archives that can be developed independently.

In order to obtain these independent archives the current software archive is analyzed and subsequently modules can be extracted from the single software archive into seven smaller software archives. 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 archives.

To detect and subsequently map the dependencies that exist in the monolithic archive we employed *formal concept analysis* (FCA). This analysis method has previously been used for purposes similar to ours, albeit on a smaller scale [4, 16, 2, 13, 15]. As such, a major contribution of this paper is the description of our experiences with apply-

[1]Due to a non-disclosure agreement, we are not at liberty to divulge on which product we applied our analysis, nor can we state an exact amount of LOC. In the remainder of this text we will refer to the case as the PMS case.

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

ing FCA in an industrial setting on a large-scale legacy application. This leads us to our principle research question for this study: *does formal concept analysis allow for the splitting of a large-scale monolithic software archive?*

The structure of this paper is as follows: the next section provides insight into the context at Philips Medical Systems. In Section 3 we provide a primer to formal concept analysis and we discuss how we apply formal concept analysis to the software archive at hand. Section 4 introduces the leveled approach, while Section 5 presents the results. Section 6 discusses the approach and results. Section 7 points to related work, while Section 8 concludes.

2 The application from Philips Medical Systems

The archive 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 code archive 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 code archive 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 MLOC. A commercial tool called 'Sotograph' is available at PMS to extract static relations from the archive [14]. 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 abstraction levels — such as the file or building block level — are obtained by accumulating the relations to and from the lower level abstraction levels.

Detailed documentation about the project is available, e.g., in the form of UML class diagrams. Another form of documentation 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, we do not have this type of documentation for all building blocks. A fictional example of this project-documentation can be found in Table 1.

Affected building blocks	Why and what
ExportImage	Include new type printer for export
PrinterSelect	Add new printer type for selection
PrinterConfiguration	Add parameters for new printer

Table 1. The scope of fictive project 'newImagePrinter'

3 Formal concept analysis

Formal concept analysis (FCA) is a branch of lattice theory that has been introduced by Wille [17]. It allows to identify sensible groupings of objects that have common attributes [7].

To illustrate FCA, let us consider a toy example about musical preferences [3]. The objects are a group of people Marco, Anne, Arthur, John, Thomas, and Michael; and the properties are Rock, Pop, Jazz, Folk, and Tango. Table 2 shows which people prefer which kind of music, called the *incidence table*.

prefers	Rock	Pop	Jazz	Folk	Tango
Marco	√	√		√	
Anne	√	√			√
Arthur			√	√	
Catherine			√		
Thomas			√		
Arthur			√	√	

Table 2. Incidence table of the music example

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. In Table 2, a concept is a maximal rectangle we can obtain with relations between people and musical preferences. For example, ({Marco, Anne}, {Rock, Pop}) is a concept, whereas ({Catherine}, {Jazz}) is not, since $\sigma(\{Catherine\}) = \{Jazz\}$, but $\tau(\{Jazz\}) = \{Arthur, Catherine, Thomas, Michael\}$. The extent and intent of each concept is shown in Table 3.

More formally, a triple (O, A, R) is called a formal context when O and A are finite sets (the objects and the attributes respectively) and R is a binary relation between O and A that is:

top	({all objects}, $\emptyset$)
c_7	({Arthur, Catherine, Thomas, Michael}, {Jazz})
c_6	({Marco, Arthur, Michael}, {Folk})
c_5	({Marco, Anne}, {Pop})
c_4	({Arthur, Michael}, {Jazz, Folk})
c_3	({Marco}, {Rock, Pop, Folk})
c_2	({Anne}, {Rock, Pop, Tango})
bottom	($\emptyset$, {all attributes})

Table 3. The set of concepts of the example of Table 2.

$R \subseteq O \times A$. $(o,a) \in I$ is read: object o has attribute a.

In order to apply FCA to the PMS case we discuss the work of Siff and Reps [12]. They used FCA in a process to identify modules in legacy code. In their paper they present these three steps in the process:

1. Build the context where objects are functions defined in the input program and attributes are properties of those functions.
2. Construct the concept lattice from the context with a lattice construction algorithm.
3. Identify concept partitions-collections of concepts of which the extents partition the set of objects. Each concept partition corresponds to a possible modularization of the input program.

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). More formally this means that $CP = \{(X_0,Y_0)\ldots(X_n,Y_n)\}$ is a concept partition iff the extents of the concepts blanket the object set and are pair wise disjoint [16, 12]:

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

Tonella found concept partitions to introduce an overly restrictive constraint on concept extents by requiring that their union covers all the objects [16]. More generally, when concepts are disregarded because they cannot be combined with other concepts to cover all attributes, 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 attributes 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* iff [16]:

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

Where CPs can be directly mapped to object partitions — that is partitions of the object set — CSPs have to be extended to the object set using the so-called partition subtraction.

The *partition subtraction* of an object subpartition SP from an object partition P gives the subpartition complementary to SP with respect to P. It can be obtained by subtracting the union of the sets in SP from each set in P.

$$P \text{ sub } SP = \{M_k = M_i - \bigcup_{M_j \in SP} M_j | M_i \in P\}$$

P sub SP is itself a subpartition because sets in P are disjoint and remain such after the subtraction. The partition subtraction is used in the *subpartition extension* to obtain the object set partitioning [16].

An object subpartition *SP* can be extended to an object partition Q, with reference to an original partition P, by the union of *SP* and the subtraction of *SP* from P. The empty set is not considered an element of Q.

$$Q = SP \bigcup (P \text{ sub } SP) - \emptyset$$

We should remark however, that in the case that there is no overlap between the object sets of a set of concepts, the number of CSPs resulting from this set of concepts will grow enormously. The number of CSPs then equals the number of ways to partition a set of n elements into k nonempty subsets. (This number is given by the Stirling numbers of the second kind.)

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 [12, 16].

3.1 Application of FCA to the PMS archive

Now that we have defined the process to use, we now define the objects and attributes to use in our specific context. As *objects* we choose the set of *building blocks* in the PMS archive, 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 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 indicate that a building block:

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

We now 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 code-archive, 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 of dependency between the building blocks [14]. 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.

PMS context		attributes: from code archive			from project documentation		
		coupledBB(1)	...	coupledBB(n)	murc	ecap	gvsa
objects	BB(1)	√					√
	BB(2)	√					√
	BB(3)			√	√		
	...						
	BB(n-1)			√		√	
	BB(n)	√		√	√		

Table 4. Example context using project documentation

Feature attribute. The second type of attribute is extracted from: (1) existing architecture overviews and project documentation at PMS and (2) domain experts. As such, these attributes pertain to the *'as-built'* architecture. The particular properties that we use for this type of attribute are: (1) specificity to the PMS application, (2) layering and (3) 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 3.2.

The reasons as to why we combine two sets of attributes are:

1. The first set of attributes assumes that building blocks which are highly dependent on each other should reside in the same archive.
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 archive.

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 code archive and features extracted from the 'as-built' architecture, according to the documentation and the domain experts. Table 4 shows an example of this combination in the context, using existing documentation.

3.2 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[3].

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

In Sections 3.2.1 and 3.2.2 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.

3.2.1 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 archive, but because projects are often used to implement certain functionality, there typically is an established relation between the building blocks in the scope. An example of a scope of a fictional project can be found in Table 1.

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 archive 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 code archive 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 should be kept in mind when analyzing the results.

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.

3.2.2 PMS-specificity and layering

The other approach is taking into account the PMS-specificity and layering of the entities in the archive. PMS-specificity can be explained as: 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.

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.

By assigning these features to each building block, building blocks that share the same characteristics of PMS-specificity and layering are grouped by deducing concepts from a context which include these features as attributes. 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.

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*$\}$. For the layering a similar scale holds starting from application/UI level to the service level. Of importance to note is that 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 25 possible combinations with respect to the PMS-specificity and layering.

Building blocks that have certain common features — such as a group of building blocks that are 'very PMS-specific' and are on the 'application/UI level' — are grouped based on these features. 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.

4 The leveled approach

The process that we propose to obtain a splitting of the archive 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 archive as the set of objects in the context. The archive consists of around 360 building blocks, which results in a big context with the attributes defined, resulting in a large corresponding concept lattice. Because we expect that identifying CSPs from the concept lattice will result in enormous amounts of CSP-collections, which have to be manually evaluated, we are undertaking steps to cope with this volume of CSP-collections.

To cope with the large amount of results we propose to use a *leveled approach*. The approach makes use of the hierarchical structuring of the PMS archive: the archives are 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 archive can be analyzed in detail, while keeping the other parts at a high level. The results from such 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

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*
	...	

Table 5. Example context, shown in the building block hierarchy

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

Table 6. Accumulated features for the platform subsystem from Table 5

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.

We will now give a short example to explain how this accumulation works. When a part of the hierarchy of the archive is not examined in detail the attributes are accumulated to the entity that is examined globally. Table 5 shows part of an example hierarchy and the assigned attributes. We now decide that the 'platform-subsystem' in the hierarchy of the archive is analyzed globally and the others in detail. Table 6 shows that all the features in the lower levels in the top Table 5 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 archive and relations between building blocks. Further, separate input files are given 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' [5]. ConExp creates given a context the corresponding concept lattice. 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.

Merging concepts. Considering the number of CSPs resulting from the number of concepts, the amount 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. We will call this *merging a concept* from now on.

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. Expected is that the context is now reduced in size for a successive analysis round and a smaller amount of 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.

Merging 'some' concepts raises the question which concepts to choose when merging. Picking concepts from a set of, for example, 300 concepts resulting from an analysis step is difficult. To aid in the choosing of the concepts a quality function of a concept is developed, which we will call the *concept quality*.

This quality function is developed to ease the choice of concepts to group and is not part of FCA. The quality function takes in account how 'strong' a concept is, that is, based on how many attributes. Also relatively small groups of objects in a concept with relatively large groups of attributes could be ranked as a 'strong' concept. The following section elaborates the notion of 'concept quality' for this purpose.

4.1 Concept quality

The *concept quality* is developed as an extension for FCA to ease choosing concepts that should be merged for a next analysis round as described in Section 4. In order to give each concept a value indicating the quality of the concept a quality function is needed. This function indicates how strong the grouping of the concept is.

Basically the concept quality is based on two measures:

- The relative number of attributes of a concept
- The relative number of objects of a concept

A grouping of a concept is based on the maximal set of attributes that a group of objects share. Intuitively when a small number of objects is grouped by a large number of attributes, this indicates a strong grouping of this objects and therefore is assigned a high concept quality value. Conversely, when a small number of objects share a single attribute, this intuitively can be seen as a less strong grouping than with a large number of attributes, 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 a small number of objects share this set of attributes, this can intuitively be seen as a strong grouping, on the other hand a large number of objects sharing this set as less strong grouping. So the quality function is also based on the degree of objects in a concept sharing a set of attributes. The smaller the number of objects, sharing a set of attributes, the lower the resulting quality function value is of the specific concept.

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. The maximum number of objects in a set of concepts is called the 'MaxObjects' and the maximum number of attributes in a set of attributes is called the 'MaxAttributes'.

Now we define the *concept quality* for a concept is as follows, the values are in the range [0,100]:

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

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 Results

The analysis has been performed for each of the two approaches: the approach with the project documentation features and the approach with the PMS specificity and layering features. Both approaches make use of the hierarchy of the archive. Because the two approaches give two different results, we discuss them separately.

5.1 Project documentation features

Our analysis has been performed 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 1 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 it. Figure 1 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 resulted in more than 10 million CSPs.

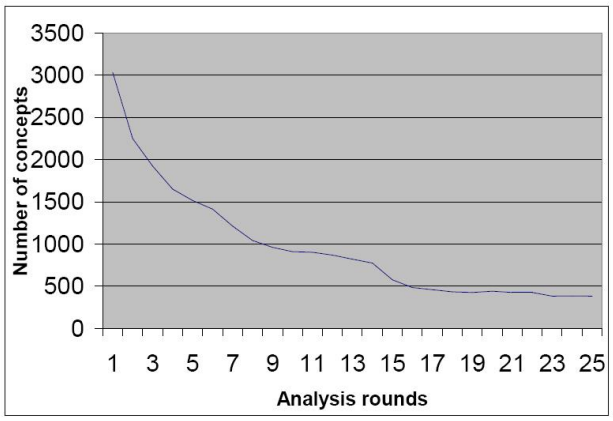

Figure 1. Results of analysis with features extracted from project documentation

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.

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 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 PMS-specificity and layering features

This section presents the results from analysis on the context with the PMS-specificity and layering features.

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, and used in the context (Figure 2).

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 2 shows that the number of concepts decreases over the analysis rounds. After 27 rounds the number of concepts

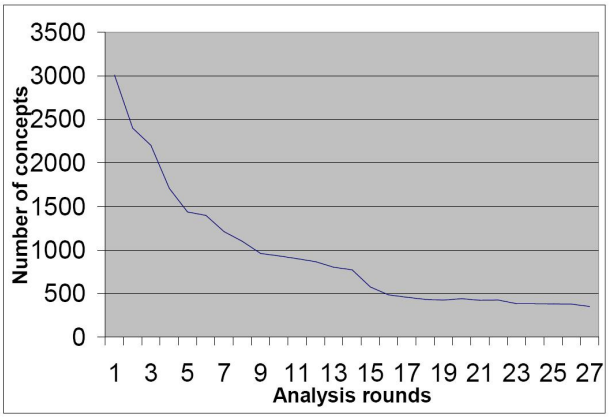

Figure 2. Results of the analysis with PMS-specificity and layering features

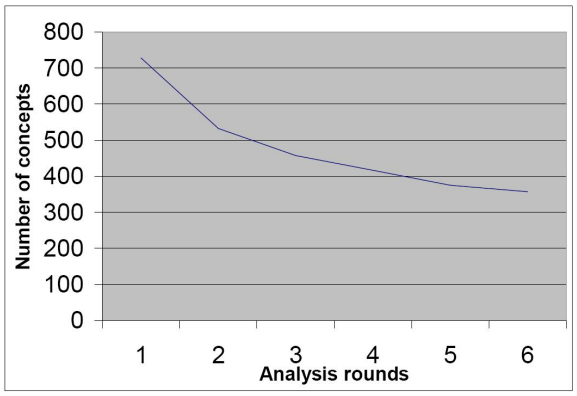

Figure 3. Results of the analysis with PMS-specificity and layering features, with imposed threshold of 25

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 3 shows a decrease of the number 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 4 shows the results of this analysis on one subsystem, for which we used the PMS-specificity and layering

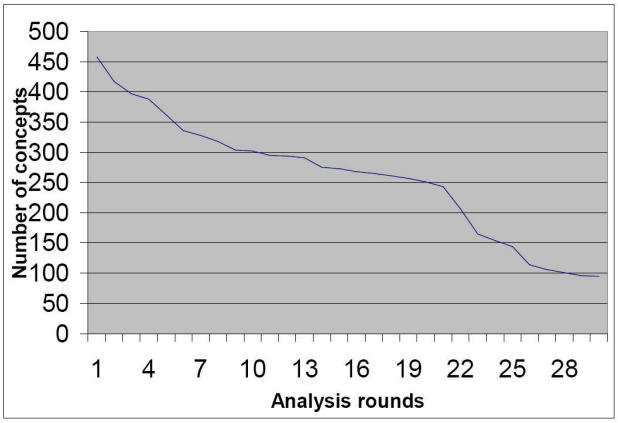

Figure 4. Results of the analysis with PMS-specificity and layering features on one subsystem

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

Table 7. Resulting concept subpartitions from the set of concepts of the analysis on one subsystem with no threshold

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 7 shows the number of resulting CSPs from the last five analysis rounds.

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 5, with the left figure showing the results with a threshold of 25 and the right figure showing the results with a threshold of 50.

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.

6 Discussion

In this experiment we used FCA as an analysis method to obtain a splitting of the archive for the PMS case. The pro-

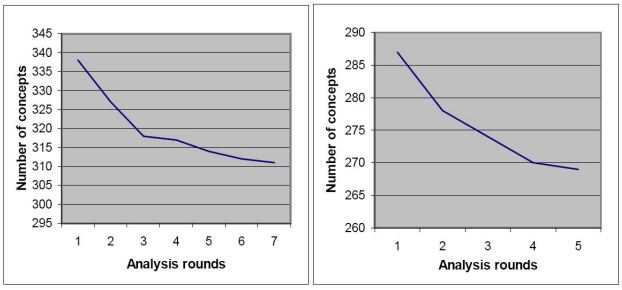

Figure 5. Results of analysis with PMS-specificity and layering features on one subsystem, with imposed threshold of 25 (left) and 50 (right)

cess of analysis works with a leveled approach, i.e., some parts of the building block hierarchy in the archive 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 newly defined concept quality.

Indeed, when we look at the results of both analyses (with the two types of features) on the complete building block structure, we can indeed see that the number of concepts decreases by applying the leveled approach and the concept quality function. However, we 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 a lot of concepts in the sets with a small number of objects. Concepts with a small number of building blocks as the 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 an amount of around 100 concepts and a resulting number of around 500,000 CSPs, we do not expect to get a significantly smaller amount of 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 an enormous amount of CSPs.

In essence, we found that the leveled approach works, as evidenced by the decreasing number of concepts. However, when deducing CSPs from a set of concepts — in order to obtain partitionings of the set of building blocks — we are faced with scalability issues, which have become evident through the size of the PMS case.

7 Related work

When considering related work, we found that many papers report on restructuring software system. However, most of the experiments described in literature report on smaller scale software systems, whereas this paper describes our experiences with FCA on a large scale software system. We now provide an overview of related research.

In this context, Snelting provides a comprehensive overview of applications of FCA to software (re)engineering problems [13], while Siff and Reps present a method to use FCA to identify modules in legacy code [12].

Hutchens and Basili identify potential *modules* by clustering on data-bindings between procedures [8]. Schwanke also identifies potential modules but clusters call dependencies between procedures and shared features of procedures to come to this abstraction [11].

Another level of abstraction with respect to the entities to be clustered is presented by van Deursen and Kuipers. They identify potential *objects* by clustering highly dependent data records fields in Cobol. They choose to apply cluster analysis to the usage of record fields, because they assume that record fields that are related in the implementation are also related in the application domain and therefore should reside in a object [6].

'High-level system organizations' are identified by Mancoridis et al. by clustering modules and the dependencies between the modules using a clustering tool called Bunch [10, 9]. Anquetil and Lethbridge also identify abstractions of the architecture but cluster modules based on file names [1].

8 Conclusion

In this paper we have presented our experiences with using formal concept analysis to come to a reclustering of a large software archive 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 archive. In literature we could not find cases with comparable goals and scale.

- We presented the *leveled approach*, which makes use of the existing hierarchy in the software archive, in order to cope with scalability issues that arise when applying FCA on a large-scale software archive.
- 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.

FCA provides ways to identify sensible groupings of objects that have common attributes. For the context of FCA we defined building blocks as objects and for the attributes two sets are combined: a set of attributes indicating that building blocks are highly dependent on each other — using a threshold — and a set of attributes that represent certain features of the building blocks. These features are derived from existing project documentation and domain experts.

From this context we derived concepts, which are used to generate concept subpartitions (CSPs). Each CSP represents a possible partitioning of the object set of building blocks, which can serve as a basis for splitting the archive.

The process of analysis works with a leveled approach, i.e., some parts of the building block hierarchy in the archive are analyzed in detail, while other parts are analyzed 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 looking at the results with the domain experts at PMS, the idea of applying the leveled approach in combination with the concept quality works well: the number of concepts decreases significantly. However, the resulting number of CSPs is still enormous. Therefore, we think that within our industrial setting applying FCA — i.e., obtaining CSPs from the set of concepts, given a context of reasonable size — is not feasible in practice.

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

Acknowledgements. This work could not have been carried out without the support of many colleagues at Philips Medical Systems. Our thanks also go to Bas Cornelissen, for proofreading this document. This work is sponsored by the NWO Jacquard Reconstructor research project.

References

[1] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, 1999.

[2] G. Antoniol, M. Di Penta, G. Casazza, and E. Merlo. A method to re-organize legacy systems via concept analysis. In *Proc. of the Int'l Workshop on Program Comprehension (IWPC)*, pages 281–292. IEEE, 2001.

[3] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proc. of the Conf. on Software Maintenance and Reengineering (CSMR)*, pages 62–71. IEEE, 2005.

[4] G. Arévalo, S. Ducasse, and O. Nierstrasz. Lessons learned in applying formal concept analysis to reverse engineering. volume 3403 of *LNCS*, pages 95–112. Springer, 2005.

[5] ConExp. http://sourceforge.net/projects/conexp, 2007.

[6] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. of the Int'l Conference on Soft. Engineering (ICSE)*, pages 246–255. IEEE, 1999.

[7] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1997.

[8] D. H. Hutchens and V. R. Basili. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, 1985.

[9] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc. of the Int'l Conference on Software Maintenance (ICSM)*, pages 50–59. IEEE, 1999.

[10] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. of the Int'l Workshop on Program Comprehension (IWPC)*, pages 45–52. IEEE, 1998.

[11] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. of the International Conference on Software engineering (ICSE)*, pages 83–92. IEEE, 1991.

[12] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 170–179. IEEE, 1997.

[13] G. Snelting. Software reengineering based on concept lattices. In *Proc. of the Conf. Software Maintenance and Reengineering*, pages 3–10. IEEE, 2000.

[14] Sotograph. http://www.software-tomography.com/html/sotograph.html, 2007.

[15] T. Tilley, R. Cole, P. Becker, and P. Eklund. A survey of formal concept analysis support for software engineering activities. In B. Ganter, G. Stumme, and R. Wille, editors, *Formal Concept Analysis*, volume 3626 of *LNCS*. Springer, 2005.

[16] P. Tonella. Concept analysis for module restructuring. *IEEE Trans. on Software Engineering*, 27(4):351–363, 2001.

[17] R. Wille. *Restructuring lattice theory: an approach based on hierarchies of concepts*, pages 445–470. Reidel, 1982.