Detection of Logical Coupling Based on Product Release History

Harald Gall, Karin Hajek, and Mehdi Jazayeri

Technical University of Vienna, Distributed Systems Group
Argentinierstrasse 8/184-1, A-1040 Wien, Austria, Europe
{gall,hajek,jazayeri}@infosys.tuwien.ac.at

Abstract

Code-based metrics such as coupling and cohesion are used to measure a system's structural complexity. But dealing with large systems—those consisting of several millions of lines— at the code level faces many problems. An alternative approach is to concentrate on the system's building blocks such as programs or modules as the unit of examination. We present an approach that uses information in a release history of a system to uncover logical dependencies and change patterns among modules. We have developed the approach by working with 20 releases of a large Telecommunications Switching System. We use release information such as version numbers of programs, modules, and subsystems together with change reports to discover common change behavior (i.e. change patterns) of modules. Our approach identifies logical coupling among modules in such a way that potential structural shortcomings can be identified and further examined, pointing to restructuring or reengineering opportunities.

1 Introduction

Large software systems are continuously modified and increase in size and complexity. After many enhancements and other maintenance activities, modifications become hard to do. Therefore, methods and techniques are needed to restructure or even reengineer a system into a more maintainable form.

To evaluate the impact of changes, we need to understand the relationships, that is, dependencies among modules that compose the system. Current methods of identifying dependencies are based on metrics such as coupling and cohesion measures [6,17]. These measures identify dependencies among modules by the existence of such relationships as procedure calls or "include" directives. There are two basic issues with these measures:

1. These measures are based on source code which is usually very large. In our case study the source code consists of 10 million lines of code (MLOC) per system release.
2. Such measures do not reveal all dependencies (e.g. dynamic relations). In fact, some dependencies are not written down either in documentation or in the code. The software engineer just "knows" that to make a change of a certain type, he or she has to change a certain set of modules.

We may say that such code-based measures reveal *syntactic* dependencies and what we are really interested in is *logical* dependencies among modules. The purpose of this paper is to present an approach to uncover such logical dependencies by analyzing the release history of a system. Release histories contain a wealth of information about the software structure. The task is just to analyze them and uncover the information.

In particular, we can analyze release histories to look for patterns of change: are there some modules that are always changed together in a release? Are there sequential dependencies such as if module A is changed in one release, module B is changed in the next release? And so on.

We have developed a technique called CAESAR for detecting such patterns. We have applied the technique to a large system with a 20-release history and identified potential dependencies among modules. To validate the accuracy of these dependencies identified by our technique, we examined change reports that contain specific change information for a release. The results have shown that this approach is promising in identifying "logical" couplings among modules across several releases.

Our technique reveals hidden dependencies not evident in the source code and identifies modules that are candidates for restructuring. The technique requires very little data to be kept for each release of a system. Rather than dealing with millions of lines of code, it works with structural information about programs, modules, and subsystems, together with their version numbers and change reports for a release. Such release data is both easy to compute and usually available in a company.

The CAESAR approach is distinguished from metrics-based approaches by being based on empirical observation of structural modifications of a system through its evolution. The technique leads naturally to an evaluation of a system’s architecture and points out potential structural and architectural enhancements.

2 Related work

We describe the evolution of a large Telecommunication Switching System (TSS) based on information about its structure stored in a database. The investigation involves 20 system releases that were delivered over a period of about two years. These releases were triggered by product improvement and new customer demands. The data on which our analysis is based consists of information about the names and version numbers of programs, modules, and subsystems of the 20 releases. Based on the findings in [14,15,23] and our quantitative analysis of the TSS described in [7] we again use the modules as our unit of investigations, rather than the source code.

Our goal is to identify logical coupling of modules that is otherwise hidden in the source code in terms of change patterns. If programs change together across module or subsystem boundaries, the decomposition structure of the application should be reconsidered and possibly restructured. Restructuring or reengineering methods are well-developed, for example in [1] or [4] or [9].

Related approaches differ from our work in that they mainly focus on a micro-level to analyze the evolution of a software system: the source code is analyzed and source code metrics are used as indicators of the system’s quality and complexity [19]. Other approaches identify fault-prone modules using statistical techniques based on design metrics [18] and discriminant analysis [12,13]. Fault and defect metrics are used for in-process project control and for process improvement over time in [4].

Coupling and cohesion measures were defined by Yourdon and Constantine [24] as a way to measure structural cohesiveness of a design. The main purpose of such measures is to evaluate how maintainable a design and resulting implementation are, and to guide improvement efforts. The basic idea is that the more dependencies that exist among modules, the less maintainable the system is because a change in one module will necessitate changes in dependent modules. Approaches to measuring module dependencies fall into two categories according to the information on which it is based:

- *code-level* approaches measure coupling based on analysis of source code; naturally, such measures can only be made after the code has been written.
- *predictive* measures try to measure coupling based on design information; such approaches attempt to evaluate the complexity of the system *before* the code has been written.

Our approach attempts to measure coupling based on empirical analysis of multiple releases of a system. This approach is based on observed change behavior of modules in a system and may be categorized as *retrospective*. Our measures may be used not only as coupling measures to guide restructuring efforts but also to validate the effectiveness of predictive and code-level coupling measures.

Other related work analyzes the structure and the architecture of software systems. Methods for architectural reasoning and assessment as described in [20] or [22] could be used for restructuring the architecture.

Visualization approaches such as SAAM [11], SeeSys [2], or SeeSoft [5] deal with the visualization of software in different ways by comparing architectures or architectural styles, visualizing statistics associated with the code, or visualizing source code information. We focus on a macro-level of software evolution by tracking the release history of a system. We thereby investigate only structural information about each release (such as version numbers of system modules) but no source code metrics.

The paper is organized as follows: In Section 3 we describe the case study to the extent needed to understand the evolution observations. Section 4 describes our approach for identifying logical coupling among modules based on release histories. We report on our results in Section 5 and draw some conclusions in Section 6.

3 The case study

The software examined in this case study is a Telecommunication Switching System (TSS). Telecommunication Switches are used to connect telephone lines and consist of both hardware and software. Our evaluation only concerns the software. The TSS covers a wide range of utilization: for example, it can be used as a switch in a fixed network, as a large international switch and as a switch for mobile telephones. The source code of TSS consists of over 10 million lines of code and several thousand files.

The TSS was first shipped in the early 1980s. The implementation of the software of the initial release was done in a machine-specific low-level language. After a few years this language was gradually replaced. So far, many different languages, such as Assembler, C and Basic, have been used to code new parts of the system. Presently, the system is being developed using SDL a high level language popular in telecommunication systems. SDL programs are translated into C and then compiled with a standard C-compiler.

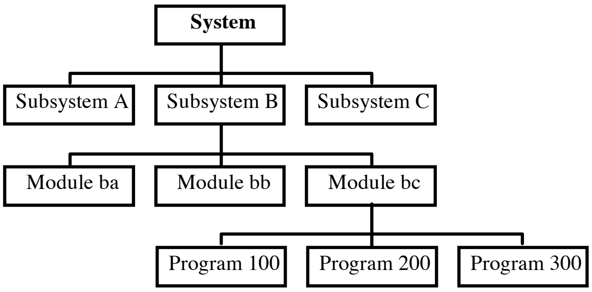

Figure 1. The software structure of the TSS

3.1 The structure of the case study

The software structure of the TSS is a tree hierarchy with four levels: the system, subsystem, module, and program level. Each level consists of one or more elements and each element of a certain level is connected to one element of the higher level. The system level contains only one element representing the root of the tree. The elements in each level are named corresponding to the names of the levels.

Figure 1**SEQARABIC**SEQARABIC shows the generic software structure of the TSS. The tree hierarchy limits the visibility of the code contained in the program level. For instance, an algorithm of a specific program can only be seen by another program of the same module. The tree hierarchy, however, does not restrict the use of the algorithms at the implementation level. Note that this logical structure has been defined after the implementation of many releases of the system and that it represents the organizational structure rather than the structure of the actual implementation.

Telecommunication switches are products that require extensive customization for different markets and applications. Currently, this customization affects large parts of the system, mainly because the customization is performed by making changes directly to the code. Each customer receives a specially adapted program. This kind of customization makes the system expensive to develop, test, and maintain.

3.2 The Product Release Database (PRDB)

A system of 10 MLOC is difficult to manage. To help simplify the management and to enable the study of the structure, a database stores structural information of the whole system. The information required to populate the product release database (PRDB) is derived directly from the source code: during compile time preprocessors extract the required information and store it.

The PRDB contains 20 different releases (representing releases over 21 months). The requirements for new releases vary from functionality enhancements (both customer and environment driven) to bug fixes.

For each release stored, the database contains entries for elements at the system, subsystem, module, and program level. Systems and programs are characterized by a version number. Each system has the version number of the specific release. Program version numbers are independent from the version number of the system to which they are connected. Programs which have been changed from one release to the following are identified by an incremented version number in the newer release. Furthermore, relations between various elements of the system are stored in the PRDB (e.g. Module bc consists of Programs 100, 200, and 300). Properties are used to attach additional information to elements or relations, such as textual descriptions of an element or the name of the developer. Each system release stored in the database consists of eight subsystems, 47 to 49 modules, and about 1500 to 2300 programs.

4 The CAESAR approach

In this section, we describe our approach to identifying change patterns among modules and revealing hidden dependencies among them. To do that, we define two processes that use the Product Release Database (PRDB). We give an overview of the two processes here and define them in detail later in the paper.

1. The *Change Sequence Analysis (CSA)* identifies patterns of change. Each change of a module (reflected in a change of its version number) is related to the system level—with system releases—as shown in Table 1. All changes of a module can then be viewed on the system level and put together to form a change sequence. A change sequence for a module shows the releases in which the module has been changed. Such change sequences allow to compare different modules in terms of their change history and identify common "change patterns." The output of the CSA process is a set of change patterns that define a so-called "logical" coupling among specific modules.

2. *Change Report Analysis (CRA):* To verify the logical coupling identified in the CSA process, it is necessary to examine change reports as a further source of release information. A change report describes the

reasons, error class, amount and type of a change of a single program with regard to a particular version number. The Change Report Analysis looks at the change reports for programs with the same change sequence. If the change reports identify the same reason for the change, such as the same bug report in programs with the same change sequence, then the logical coupling identified in the CSA process is verified.

4.1 Definitions

We start by giving some definitions that allow us to represent programs and their change patterns abstractly as numbers and sequences of numbers.

Table 1 shows the program P_i which occurs from system release 3 through 8. The first row shows the version number of P_i as it appears in the PRDB. For our analysis, the important point about the program is the release number(s) in which it is changed. In this case, from the version numbers in the first row, we can see that the program is changed in releases 3 and 5. This information is shown in the second row.

System Release	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
P_i version no.			1.1	1.1	1.3	1.3	1.3	1.3	
P_i change sequence			**3**		**5**				

Table 1. Program P_i’s version numbers and representation as change sequence

- A *change sequence* is defined on the program level and is an n-tuple <1 2..n> of those release numbers in which the program changes its version number. A program change sequence contains all system release numbers in which the program changes (e.g. in Table 1 <3 5> denotes one change of P_i in release 5).
- A *subsequence* (SUB) is a contiguous part of a sequence.
- *Changes* are represented by a sequence or subsequence.
- A *Change Report* (CR) is a report of a version number change of a program. It contains all descriptions of changes of the programs involved. There exists different types of changes and different types of error classes.

4.2 The Change Sequence Analysis (CSA)

The Change Sequence Analysis allows to reason about “logical coupling” among different elements (i.e. programs or modules). Logical coupling refers to observed identical change behavior of different elements during system evolution. The main principle of CSA is to represent each change of a version number of an element on the system level as a system release change. This abstract way of representation is chosen to be independent of the level on which the CSA is performed. Therefore it is possible to compare the behavior of different decomposition levels during system evolution.

This paper focuses on the level of programs because of the following TSS-specific reason: If a program changes its version number, the containing module must also change its version number. A version of a module defines the version of a program belonging to it. The module level is not representative since the modules contain all the changes performed on the level of the programs. This results in a high number of changes for each of the modules. Therefore, we focus on the level of the programs viewed on the level of subsystems. Since this level consists of only 8 different subsystems it is a good way to represent the changes of the different programs.

Two kinds of coupling are considered in the CSA process:

- *System coupling* represents relationships among different subsystems via sequences
- *Sequence coupling* represents relationships among different sequences via subsystems

Both of them use subsequences to represent a specific behavior in a certain part of the system evolution. In the following, we discuss the two different couplings in terms of the TSS.

4.2.1 Coupling among subsystems

This coupling represents subsystems related via different sequences. Since the evaluations are done on the version numbers of the programs, each subsystem represents a specific program to which it is related. The subsystem level has a compact representation since there are a fixed number of subsystems (for TSS: 8). Subsystems are coupled if they are related to the same sequence and contain the same defined subsequence. Intuitively, if two subsystems are related to the same subsequence, it means that the subsystems were changed in the same releases.

After listing the change sequences for all programs, different subsequences are compared against all sequences. If two or more sequences contain the same subsequence, we postulate a logical coupling among those sequences and the associated programs and subsystems. Figure 3 is a simplified model of the TSS with respect to a particular change sequence. We have changed the subsystem names to hide the identity of the real system. Each subsystem is represented by a circle. The different styles of lines represent the different amount of changes belonging to the sequences. The sequence name S_k=<c_1 c_2 .. c_n> represents the changes c_i in which this sequence occurs. Figure 3 shows that, except for subsystem F, each of the subsystems includes the subsequence SUB_1=<1 2 4>. It further depicts that subsystem G has no coupling to any other subsystem; its only coupling is internal to its own programs with respect to the change sequence SUB_1.

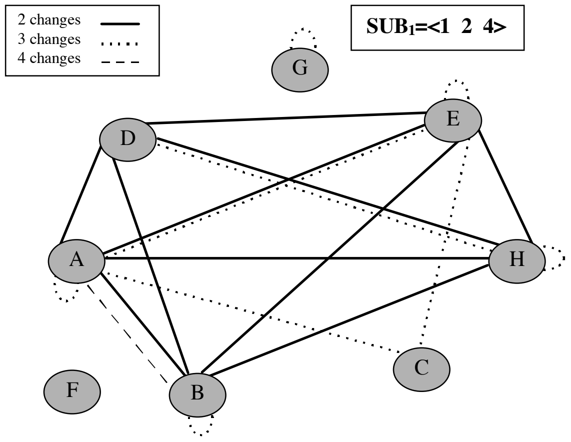

Figure 3. Coupling among subsystems

Subsystem A shows existing coupling inside the same block which is depicted as a self reference. Therefore this coupling refers to at least two different programs belonging to the same subsystem as shown in Table 2 (E.g., ‘A.ac.144’ denotes program 144 in module ac of subsystem A).

SUB_1=<1 2 4>					
A.ac.144	**1**	**2**	**4**	6	19	20
A.ad.200	**1**	**2**	**4**	6	19	20
A.ad.201	**1**	**2**	**4**	6	19	20

Table 2. Coupling among modules (ac, ad) within subsystem A for SUB_1

In Figure 3, subsystems A, C, and E are seen to be coupled via a sequence of 3 changes with every sequence including SUB_1 as subsequence. All three subsystems include programs related to the sequence S40=<1 2 4 7> with 3 changes (S40 includes the subsequence SUB_1=<1 2 4>).

In general, there could exist many sequences that include SUB_1 and represent 3 changes. Each line in Figure 3 represents the fact that there is one or more sequences shared by the related subsystems. To see how many shared sequences are represented by each line, we have to check the sequence coupling (see Section 4.2.2).

Figure 3 further shows that subsystems D and H are also coupled via a sequence of 3 changes: sequence S49=<1 2 4 6> is related to several programs of the two subsystems D and H; this fact is represented in a line that “logically” couples D and H.

Subsystems are coupled via sequences that can represent different amounts of changes. Let us consider the example shown in Table 3.

SUB_2=<1 2 3 4 6 7 9 10 14>											
A.aa.111	**1**	**2**	**3**	**4**	**6**	**7**	**9**	**10**	**14**	17	19
B.ba.222	**1**	**2**	**3**	**4**	**6**	**7**	**9**	**10**	**14**	16	18

Table 3. Coupling among subsystems A and B via SUB_2

Subsequence SUB_2 represents 8 changes. There are some programs with change sequences larger than 8 which include SUB_2 as a subsequence. An example is given in Table 3, in which program 111 of subsystem A and program 222 of subsystem B include SUB_2. As a result of this, subsystem A and B are “logically” coupled via subsequence SUB_2. Intuitively, this means that in eight different releases, these programs were both changed.

Subsequences are used to compare different blocks. Short subsequence may be shared by programs coincidentally. But the longer the subsequence that is shared among programs, the higher is the probability of coupling among the programs. If there exists a long subsequence included in many different change sequences, it can be assumed that the programs are dependent on each other. We, therefore, look for long sequences in the CSA process.

If commonalities via subsequences are detected, then logical coupling among different programs and, as a consequence, among different subsystems exists. This is a first step in identifying the same change behavior during system evolution based on different levels and blocks. To determine whether or not this logical coupling represents real dependencies, it is necessary to inspect the change reports via the change report analysis (CRA).

4.2.2 Coupling among sequences

This coupling represents sequences which connect different subsystems. The sequences are subdivided into groups of different amount of changes which belong to 2, 3, .., 9, and >9 changes. The commonality among sequences is always a subsequence that is included in each of the sequences such as SUB_1=<1 2 4> in Figure 4.

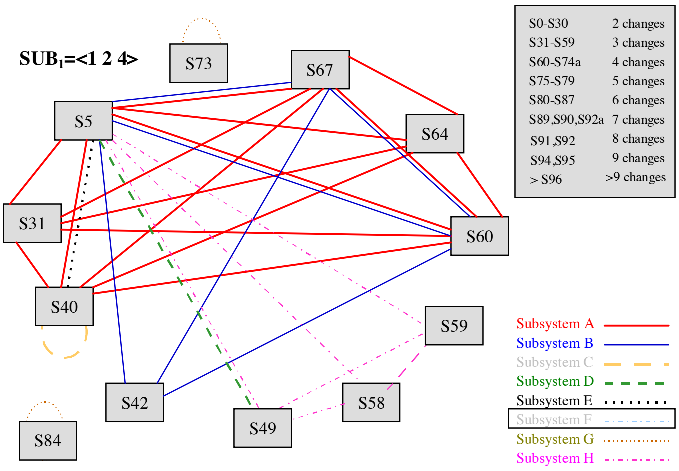

Figure 4. Coupling among sequences

Figure 4 depicts coupling among sequences. The boxes represent sequences and the different styles of lines represent the subsystems of TSS. In Figure 4, sequence S5 belongs to the group of 2 changes. The dotted line in S84, for example, indicates that this sequence is only referred to by change sequences of programs in subsystem G.

Let us examine the sequences S5=<1 2 4> and S40=<1 2 4 7>. S5 is connected by 5 different lines which indicate 5 different subsystems (A, B, D, E, and H, see also Table 4). Both S5 and S40 are connected to subsystem E represented as a dotted line. Sequence S84, which belongs to the sequence group of 6 changes and represents coupling within its sequence, is not related to any other sequence but points to subsystem G.

Table 4 shows sequence S_5=SUB_1=<1 2 4> and the coupled programs and subsystems in more detail.

S_5=<1 2 4> = SUB_1			
A.aa.005	1	2	4
B.ba.098	1	2	4
D.da.307	1	2	4
D.da.309	1	2	4
E.ec.330	1	2	4
H.ha.377	1	2	4
H.hb.390	1	2	4

Table 4. Coupling among sequence S5

Analyzing the behavior of sequences reveals which subsystems are related to which sequence while observing a specific subsequence. This supports the search for a specific behavior—such as SUB_1=<1 2 4>—where changes in different subsystems were done in system releases 2 and 4.

The coupling among sequences adds more detail to the logical coupling among subsystems and identifies the specific programs and the specific releases in which the subsystems exhibited exactly the same change pattern.

4.3 Change Report Analysis (CRA)

During the maintenance phases, once a failure is reported and its cause determined, the problem is fixed by one or more changes. These changes may include modifications to one or more of the development products, including the specification, design, code, test plans, test data or documentation. Change reports are used to record the changes and track the products affected by them. A typical change report may look as follows [6]:

Change report	
Location:	identifier of document or module change
Timing:	when change was made
Symptom:	type of change
End result:	success for change, as evidenced by regression or other testing
Mechanism:	how and by whom change was performed
Cause:	corrective, adaptive, preventive, or perfective
Severity:	impact on rest of system, sometimes as indicated by an ordinal scale
Cost:	time and effort for change implementation and test

We use the change reports for programs to verify whether the logical coupling among different programs represents real dependencies among those programs. A change report contains a report of the type, error class, kind and number of a change done in a program. Logical coupling represents a real dependency if the change reports for the different programs include significant similarities, for example, they reference the same bug report.

4.3.1 Description of a change report

In general, a program has several change reports since each belongs to a change from one version number to another. We consider the following change report of program 111 (with comments included):

Ver 2.4 — 96/03/12 10:10:07
TSS---PROGRAM CHANGE DESCRIPTION
ELEMENT NAME: Program 111 2.3 --> 2.4
CHANGED BY: John DOE
CHANGES as follows:

CHANGE NR: 1
CHANGE TYPE: B // bug fix
REFERENCE: BR 4711 // reference to a bug report number
ERROR CLASS: A // i.e. operation in working state
DESCRIPTION: hanging of the circuits in environment xy.

The change reports of the TSS case study refer to 3 main types: “Further Development (FD)” which is divided into several parts such as FD based on system specification, development of technology (software and hardware) and so on. The second is a general “Change” such as optimizing or improvement. The third is a correction of an error which refers to a specific “Bug Report (BR).” For our case study it was only important to differentiate between these three types although some more types exist for the TSS.

For the TSS case study we use the terms with numbers, such as FD1 or C5 referring to a specific subtype of change. To completely identify whether two changes in two different change reports refer to the same change, we also have to inspect the change-reports' comments which usually are not categorized in great detail.

4.3.2 Analysis steps

To analyze the change reports we have to inspect the version numbers of the programs together with the change sequence as shown in Table 5.

		system releases						
		1	2	3	**4**	5	**6**	**7**
S28	A.aa.111				**2.3**	2.3	**2.4**	**2.6**
					4		**6**	**7**

Table 5. Analyzed program for a change report

For every change of a program (reflected in its change sequence), the corresponding change report(s) are identified and analyzed. The change reports that describe a new version of a program are then listed as shown in Figure 5.

	4		**6**		**7**
Program 111	2.3	BR 4711	2.4	FD 1	2.6

Figure 5. Change history of Program 111

In Figure 5, Program 111 occurred in system release 4 with version number 2.3. The first change was done in system release 6 resulting in version number 2.4. The change was a bug fix according to the specific bug report BR 4711.

The second change occurred from system release 6 to 7 as a change from version number 2.4 to 2.6. This change is of type “FD 1” and refers to a specific “further development” change.

Sometimes a program has several changes until its version is included in a specific system release as shown in Figure 6.

	10		**11**
Program 222	6.1	BR 1234 BR 1235 BR 1239	6.4

Figure 6. Change history of Program 222

Figure 6 depicts that 3 changes occurred between system releases 10 and 11. Three bugs were fixed which refer to the bug reports BR 1234 (from version 6.1 to 6.2), BR 1235 (from version 6.2 to 6.3), and BR 1239 (from version 6.3 to 6.4). After fixing these bugs, the final version number of Program 222 (i.e. 6.4) was included in system release 11.

Change reports are necessary to determine whether the logical coupling identified in the CSA process indicates real dependencies or just coincidental changes. The next section examines some programs which have important commonalities in those change reports.

4.3.3 An example

Let us assume that after analyzing the system there was logical coupling found with the subsequence SUB_1=<2 4 6 7>. Four different programs including this subsequence caused the logical coupling. These four programs are related to two different subsystems A and B as shown in Figure 7. By examining the change reports of those 4 programs we see that the logical coupling via the subsequence SUB_1 results in the following behavior:

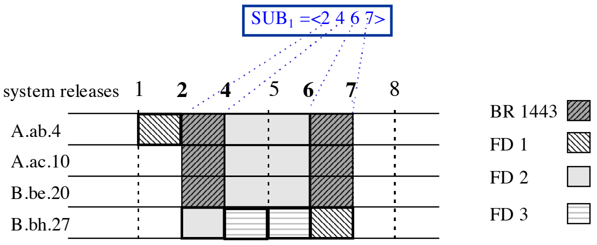

Figure 7. Programs, change reports, and commonalities

If two different programs have the same FD-number they are dependent on each other in some way. But if two or several programs refer to the same bug report then they have a strong dependency because the same bug had to be fixed.

In Figure 7, FD1 was done in program 4 from system releases 1 to 2 and in program 27 from system release 6 to 7. This shows that in program 27 the same change was done 5 releases later than the first development of program 4 happened.

Inspecting the programs 4, 10, and 20 revealed a strong dependency: All three programs have the same changes from system release 2 to 4 and from 6 to 7. All of them refer to the same bug report number BR1443. In all of the three programs the same error occurred which had to be fixed. From system release 6 to 7, the same bug report number occurred again for the examined programs. Since one bug report refers to a specific error, one possible answer for this is that the bug was not fixed properly in all affected parts of the system from system release 2 to 4. Therefore, in system release 6 the same bug occurred again and had to be fixed.

4.4 Résumé

The CSA process locates potential couplings and then analyzes in more detail the changes done in a program. With CRA we can verify logical coupling through descriptions in change reports. If the change was a bug fix and several programs refer to a specific bug report number, this change verifies a real logical coupling. Other types of changes (e.g. further development etc.) can be analyzed for commonalities accordingly. For more details about the specific examples see [10].

5 Results

In our case study, we examined different subsequences to identify change patterns in different programs and subsystems. The length of a subsequence is important when analyzing the coupling. As a consequence, our technique detects a stronger logical coupling of subsystems (or programs), if they are coupled via a long subsequence.

In general, we discovered that in accordance to our findings in [7], many changes were performed in system releases 1, 2, 4, 5, 7, and 11. This corresponds to the logical coupling among the different subsystems. For example, 31 programs refer to the same change sequence.

Considering the size of each subsystem, subsystem C was continuously growing over the 20 releases (for details refer to [7]), but this fact is not reflected in a high logical coupling with other subsystems. The potential for restructuring is, therefore, within subsystem C and its modules. The structural shortcoming is local in terms of subsystems but with a high interrelationship among modules of subsystem C. From the CSA and the CRA we have identified those modules and programs that should undergo restructuring or even reengineering.

6 Conclusions and future work

We have presented a new way to analyze large software systems with millions of lines of source code: building on the quantitative analysis of a Telecommunication Switching System performed in [7], this paper considers logical attributes of the TSS. Such large systems often reach a high level of complexity and any extension or adaptation causes a large maintenance effort. Therefore, it is necessary to examine the structure of the system concerning its architecture and the dependencies of the different modules and subsystems. Based on these results, further maintenance activities can be estimated more accurately in terms of time needed and software parts affected.

We developed a technique called CAESAR for detecting change patterns and applied it to a large Telecommunication Switching System with a 20-release history. We identified potential dependencies among modules, and validated these potential dependencies by examining change reports that contain specific change information for a release. The results have shown that this approach is promising in identifying such "logical" couplings among modules across several releases.

Our technique reveals hidden dependencies not evident in the source code, identifies modules that should undergo restructuring, and is based on minimal amount of data that must be kept about each release. Rather than dealing with millions of lines of code we use structural information about programs, modules, and subsystems, together with their version numbers and change reports for a release. Such release data is both easy to compute and usually available in a company.

Although our results are preliminary, the use of CAESAR was able to uncover phenomena such as bugs being fixed in one system release and "re-emerging" a couple of releases later to be fixed in other parts of the system, and we were able to verify couplings identified by CSA using CRA. Our results indicate that such retrospective analysis is a valuable complement to code-based and predictive analyses that are commonly practiced today.

The insights gained from the case study allow us to define those data that are required to detect logical coupling in very large systems: change sequences, subsystem and sequence couplings together with those change reports that are referred to in the version changes of modules and programs.

So far, we have used only simple tools in our study. In the future, we plan to add a visualization capability to the release database to enable a maintenance engineer to view the identified relationships with 3-dimensional graphs (as presented in [21]) and to navigate across the releases and the discovered module and subsystem dependencies.

7 Acknowledgments

We thank our industrial partner that provided all the information and helped us with the interpretation of the results. We are also grateful to Georg Trausmuth for his critical remarks.

This work was supported by the European Commission within the ESPRIT Framework IV project no. 20477 ARES (Architectural Reasoning for Embedded Systems).

8 References

[1] Arnold R. S., "Software Reengineering," Proceedings, IEEE Computer Society Press, Los Alamitos, CA, 1993.

[2] Baker M.J. and Eick S.G., "Visualizing Software Systems," AT&T Bell Laboratories, 1994.

[3] Choi S.C. and Scacchi W., "Extracting and Restructuring the Design of Large Systems," IEEE Software, pp. 66-71, January 1990.

[4] Daskalantonakis M.K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola," IEEE Transactions on Software Engineering, Vol. 18, No. 11, pp. 998-1010, November 1992.

[5] Eick S. G., Steffen J. L., and Summer E. E. Jr., "Seesoft-A Tool For Visualizing Line Oriented Software Statistics," *IEEE Transaction on Software Engineering*, Vol. 18, No. 11, November 1992.

[6] Fenton N.E., Pfleeger S.L., Software Metrics—A Rigorous & Practical Approach, International Thomson Computer Press, Second Edition, 1996.

[7] Gall H., Jazayeri M., Klösch R., and Trausmuth G., "Software evolution observations based on product release history," International Conference on Software Maintenance (ICSM '97), Bari, Italy, pp.160-166, October 1997.

[8] Gefen D. and Schneberger S.L. "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications," International Conference on Software Maintenance, pp. 134-141, November 1996.

[9] Griswold W.G. and Notkin D., "Automated Assistance for Program Restructuring," ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 3, pp. 228-269, July 1993.

[10] Hajek K., "Detection of Logical Coupling Based on Product Release History," Master's Thesis, Technical University of Vienna, Austria, March 1998.

[11] Kazman R., Bass L., Abowd G., and Webb M., "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proceedings of ICSE 16*, Sorento, Italy, pp. 81-90, May 1994.

[12] Khoshgoftaar T., Allen E.B., Kalaichelvan K.S., and Goel N., "Early Quality Prediction: A Case Study in Telecommunications," IEEE Software, Vol. 13, No. 1, pp. 65-71, January 1996.

[13] Khoshgoftaar T.M. and Halstead R., "Detection of Fault-Prone Software Modules During a Spiral Life-Cycle," International Conference on Software Maintenance, pp. 69-76, November 1996.

[14] Lehman M.M., "Programs, life cycles and laws of software evolution," Proceedings of the IEEE, pp. 1060-1076, September 1980.

[15] Lehman M.M. and Belady L. A., Program evolution, Academic Press, London and New York, 1985.

[16] Lientz B.P. and Swanson E.B., Software Maintenance Management, Addison-Wesley, 1980.

[17] Offen R.J., "Establishing Software Measurement Programs," IEEE Software, pp. 45-53, March/April 1997.

[18] Ohlsson N. and Alberg H., "Predicting Fault-Prone Software Modules in Telephone Switches," IEEE Transactions on Software Engineering, Vol. 22, No. 12, pp. 886-894, December 1996.

[19] Pearse T. and Oman P., "Maintainability Measurements on Industrial Source Code Maintenance Activities," International Conference on Software Maintenance, pp. 295-313, October 1995.

[20] Perry A. E., and Wolf A. L., "Foundations for the Study of Software Architecture," Software Engineering Notes, ACM SIGSOFT, Vol. 17, No. 4, pp. 40-52, October 1992.

[21] Riva C., "Visualizing Software Release Histories: The Use of Color and Third Dimension," Master's Thesis, Politecnico di Milano, June 1998.

[22] Shaw M., and Garlan D., Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.

[23] Turski W.M., "Reference Model for Smooth Growth of Software Systems," IEEE Transactions on Software

Engineering, Vol. 22, No. 8, pp. 599-600, August 1996.

[24] Yourdon E., and Constantine L., Structured design: Fundamentals of a discipline of computer program and systems design, Prentice-Hall, 1979.