Visualizing and Measuring Enterprise Application Architecture: An Exploratory Telecom Case

HARVARD | BUSINESS | SCHOOL

Visualizing and Measuring Enterprise Application Architecture: An Exploratory Telecom Case

Robert Lagerström
Carliss Y. Baldwin
Alan MacCormack
Stephan Aier

Working Paper

13-103

June 21, 2013

Visualizing and Measuring Enterprise Application Architecture:

An Exploratory Telecom Case

Robert Lagerström, Carliss Y. Baldwin, Alan MacCormack and Stephan Aier

Abstract

We test a method for visualizing and measuring enterprise application architectures. The method was designed and previously used to reveal the hidden internal architectural structure of software applications. The focus of this paper is to test if it can also uncover new facts about the applications and their relationships in an enterprise architecture, i.e., if the method can reveal the hidden external structure between software applications. Our test uses data from a large international telecom company. In total, we analyzed 103 applications and 243 dependencies. Results show that the enterprise application structure can be classified as a core-periphery architecture with a propagation cost of 25%, core size of 34%, and architecture flow through of 64%. These findings suggest that the method could be effective in uncovering the hidden structure of an enterprise application architecture.

1. Introduction

Contemporary business environments are constantly evolving, requiring continual changes to the software applications that support those businesses. Moreover, during the past decades the sheer number of those applications has steadily grown, and they have become increasingly interdependent. As a result, the management of software applications has become a very complex task, and many companies have found that implementing changes to their applications architecture is increasingly difficult and expensive. What would help tremendously is a tool that would enable them to visualize and analyze the modularity of their enterprise architecture and the degree of coupling between the applications.

In [1], Baldwin et al. present a method based on Design Structure Matrices (DSMs) and classic coupling measures to visualize the hidden structure of software architectures. This method has been tested on numerous software releases for large applications (such as Linux, Mozilla, Apache, and GnuCash) but not on enterprise architectures with a potentially large number of interdependent applications. This paper performs such a test using

data from a business unit of a large telecom company. The data consisted of a total of 103 applications and 243 directed dependencies.

We find that the telecom application architecture can be classified as core-periphery. This means that 1) there is one cyclic group (the "Core") of software applications that is substantially larger than the second biggest cyclic group, and 2) the Core also makes up a large portion of the entire architecture. The analysis also shows a propagation cost of 25%, meaning that one-fourth of the architecture may be affected when a change is made to a randomly selected software application in the architecture. In addition, we find that the Core contains 35 applications, which embody 34% of the architecture. And lastly, the analysis uncovers that the architecture flow through accounts for as much as 64% of the architecture, meaning that more than half of the applications are either in, depend on, or are dependent on the Core.

The remainder of this paper is structured as follows: Section 2 presents related work; Section 3 describes the hidden structure method; Section 4 presents the telecom case used for the analysis; Section 5 discusses the approach and outlines future work; and Section 6 concludes the paper.

2. Related work

In this section, we first describe the most common metrics used to assess complexity in software engineering. These metrics help analyze a single software application so that, for example, managers can estimate development efforts or programmers can find troublesome code passages. Next we describe recent work on modularity visualization for complex software architectures. These network approaches have emerged because many software applications have grown into large systems containing thousands of interdependent components, making it difficult for a designer to grasp the full complexity of the architecture. Last, we present related work on the complexity of enterprise application architectures. Enterprise Architecture (EA), which has gained much recent attention, deals with the complex networks of hundreds (or thousands) of interdependent applications in a company. Interestingly, many of the problems encountered by software architects dealing with a single software system are similar to those that occur for enterprise architects on a system-of-systems level.

2.1 Software engineering

In software engineering, metrics like Lines of Code (LOC) and Function Points (FP) have existed for many years.

We present the most common measures that are specifically relevant to software complexity. According to [2], software complexity "*is the degree to which a system or component has a design or implementation that is difficult to understand and verify.*"

One of the first complexity metrics proposed and one of the most used today is McCabe's Cyclomatic Complexity (MCC), which is based on the control structure of a software component. The control structure can be expressed as a control graph in which the cyclomatic complexity value of a software component can be calculated [3]. Only a year later, another well-known metric was introduced, namely, Halstead's complexity metric [4], which is based on the number of operators (e.g., "and," "or," or "while") and operands (e.g., variables and constants) in a software component. A few years after McCabe and Halstead, the Information Flow Complexity (IFC) metric was introduced [5]. IFC is based on the idea that a large amount of information flows is caused by low cohesion, which in turn results in high complexity.

Another important type of metric is the coupling measure. [2] defines coupling as "*the manner and degree of interdependence between software modules. Types include common-environment coupling, content coupling, control coupling, data coupling, hybrid coupling, and pathological coupling.*" Fenton and Melton [6] have defined a coupling measure based on the different levels of coupling, including the following: content coupling (if *x* refers to the internals of *y*, i.e., it branches into, changes data, or alters a statement in *y*), common coupling (if *x* and *y* refer to the same global variable), control coupling (if *x* passes a parameter to *y* that controls its behavior), stamp coupling (if *x* passes a variable of a record type as a parameter to *y*, and y uses only a subset of that record), data coupling (if *x* and *y* communicate by parameters, each one being either a single data item or a homogeneous set of data items that does not incorporate any control element), and no coupling (if *x* and *y* have no communication, i.e. are totally independent). The Fenton and Melton coupling metric *C* is pairwise calculated between components, where *n* = number of dependencies between two components and *i* = level of highest (worst) coupling type found between these two components, such that

$$C = i + \frac{n}{n+1}$$

All these complexity metrics have been tested and are used widely for assessing the complexity of software components.

2.2 Software architecture

To characterize the architecture of a complex system (instead of a single component), studies often employ network representations [7]. Specifically, they focus on identifying the linkages that exist between the different elements (nodes) in the system [8,9]. A key concept here is modularity, which refers to the way in which a system's architecture can be decomposed into different parts. Although there are many definitions of "modularity," authors tend to agree on some fundamental features: interdependence of decisions within modules and independence between modules, and hierarchical dependence of modules on components that embody standards and design rules [10,11].

Studies that use network methods to measure modularity typically focus on capturing the level of coupling that exists between different parts of a system. In this respect, one of the most widely adopted techniques is the so-called Design Structure Matrix (DSM), which illustrates the network structure of a complex system in terms of a square matrix [12-14], where rows and columns represent components (nodes in the network) and off-diagonal elements represent dependencies (links) between the components. Metrics that capture the level of coupling for each component can be calculated from a DSM and used to analyze and understand system structure. For example, [15] and [16] use DSMs and the metric "propagation cost" to compare software system architectures. DSMs have been used to visualize architectures and measure the coupling of the internal design of single software systems.

2.3 Enterprise architecture

Although DSMs have proven valuable for architecture representation, we have yet to see them deployed in enterprise architecture modeling. Instead, the following approaches have been used:

- [17] and [18] present a tool based on a metamodel that specifies the classes, attributes, and relationships needed to analyze the modifiability of an enterprise architecture. The tool includes classes (such as systems, components, documentation, change-management processes, tools, infrastructure, and change organizations) and attributes (such as the component size, system coupling, change-management process maturity, and team expertise). The metamodel was designed based on metrics used, for example, in COCOMO II.2000 [19], COBIT [20], and the Definition and Taxonomy for Software Maintainability [21]. Thus far, use of the tool has focused on estimating the development costs by looking at a number of software change projects.

- [22] present a modeling approach for virtual decoupling for IT/Business alignment. The approach is based on a metamodel that contains business processes, software systems, and the relationships between them. In this approach, the instantiated model is transformed into a graph and a clustering algorithm is applied to that graph in order to suggest architecture changes for improving the IT/business alignment.
- [23] study the relationship between an organization's software portfolio architecture and its ability to make changes to it. They conclude that both the architecture and component complexities affect the flexibility of the software portfolio.
- [24] rely on measures from disciplines like economics and anti-monopoly legislation. They propose a definition of heterogeneity in an IT landscape as a statistical property, and their generic approach quantifies heterogeneity in IT landscapes.

These enterprise architecture approaches all rely on coupling and complexity measures to analyze architectures. None, however, uses DSMs to visualize the hidden structure of the architecture or to account for the indirect dependencies among software systems when measuring coupling.

3. Method description

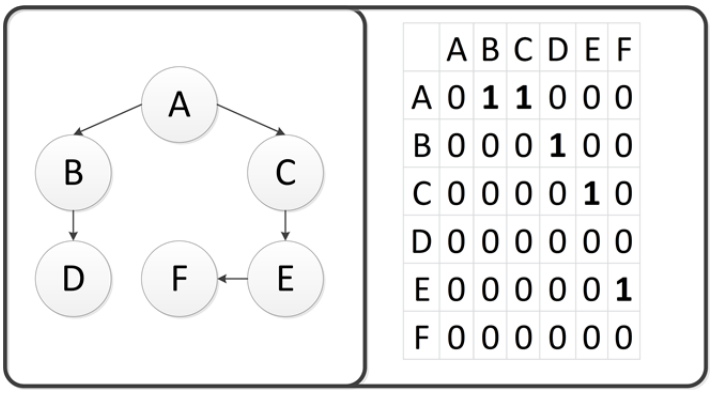
The method used for architecture network representation is based on and extends the classic notion of coupling. Specifically, after identifying the coupling (dependencies) between the elements in a complex architecture, the method analyzes the architecture in terms of hierarchical ordering and cycles, enabling elements to be classified in terms of their position in the resulting network.

In a Design Structure Matrix (DSM), each diagonal cell represents an element (node), and the off-diagonal cells record the dependencies between the elements (links): If element *i* depends on element *j*, a mark is placed in the row of *i* and the column of *j*. The content of the matrix does not depend on the ordering of the rows and columns, but if the elements in the DSM are rearranged in a way that minimizes the number of dependencies above the main diagonal, then dependencies that remain there will show the presence of cyclic interdependencies (A depends on B, and B depends on A) which cannot be reduced to a hierarchical ordering. The rearranged DSM would then reveal significant facts about the underlying structure of the architecture that cannot be inferred from standard measures of coupling or from the architect's view alone. In the following subsections, a method that makes this "hidden structure" visible is presented and metrics that can be used to compare architectures and track changes in architecture

structures over time are described. (Note: A more detailed method description can be found in "Hidden Structure: Using Network Methods to Map System Architecture" by Baldwin et al. [1].)

3.1 Identify the direct dependencies between elements

The architecture of a complex system can be represented as a directed network composed of elements (nodes) and directed dependencies (links) between them. Figure 1 contains an example (taken from [15]) of an architecture that is shown both as a directed graph and a DSM. This DSM is called the "first-order" matrix to distinguish it from a visibility matrix (defined below).



	A	B	C	D	E	F
A	0	**1**	**1**	0	0	0
B	0	0	0	**1**	0	0
C	0	0	0	0	**1**	0
D	0	0	0	0	0	0
E	0	0	0	0	0	**1**
F	0	0	0	0	0	0

Figure 1. A directed graph and Design Structure Matrix (DSM) example.

3.2 Compute the visibility matrix

If the first-order matrix is raised to successive powers, the result will show the direct and indirect dependencies that exist for successive path lengths. Summing these matrices yields the visibility matrix *V* (Figure 2), which denotes the dependencies that exist for all possible path lengths. The values in the visibility matrix are binary, capturing only whether a dependency exists and not the number of possible paths that the dependency can take [15]. The matrix for N=0 (i.e., a path length of zero) is included when calculating the visibility matrix, implying that a change to an element will always affect itself.

$V=\sum M^n$; n=[0,4]						
	A	B	C	D	E	F
A	**1**	**1**	**1**	**1**	**1**	**1**
B	0	**1**	0	**1**	0	0
C	0	0	**1**	0	**1**	**1**
D	0	0	0	**1**	0	0
E	0	0	0	0	**1**	**1**
F	0	0	0	0	0	**1**

Figure 2. Visibility matrix for example in Figure 1.

3.3 Construct measures from the visibility matrix

Several measures are constructed based on the visibility matrix *V*. First, for each element *i* in the architecture, the following are defined:

- VFI_i (Visibility Fan-In) is the number of elements that directly or indirectly depend on *i*. This number can be found by summing the entries in the i^{th} column of *V*.
- VFO_i (Visibility Fan-Out) is the number of elements that *i* directly or indirectly depends on. This number can be found by summing the entries in the i^{th} row of *V*.

In Figure 2, element *A* has *VFI* equal to 1, meaning that no other elements depend on it, and *VFO* equal to 6, meaning that it depends on all other elements in the architecture.

To measure visibility at the architecture level, Propagation Cost (PC) is defined as the density of the visibility matrix. Intuitively, propagation cost equals the fraction of the architecture affected when a change is made to a randomly selected element. It can be computed from Visibility Fan-In (*VFI*) or Visibility Fan-Out (*VFO*):

$$\text{Propagation Cost} = \frac{\sum_{i=1}^{N} VFI_i}{N^2} = \frac{\sum_{i=1}^{N} VFO_i}{N^2}.$$

3.4 Identify and rank cyclic groups

The next step is to find the cyclic groups in the architecture. By definition, each element within a cyclic group depends directly or indirectly on every other member of the group. So the elements are sorted, first by *VFI* descending then by *VFO* ascending. Next one proceeds through the sorted list, comparing the *VFI*s and *VFO*s of

adjacent elements. If the *VFI* and *VFO* for two successive elements are the same, they might be members of the same cyclic group. Elements that have different *VFI*s or *VFO*s cannot be members of the same cyclic group, and elements for which n_i=1 cannot be part of a cyclic group at all. However elements with the same *VFI* and *VFO* could be members of different cyclic groups. In other words, disjoint cyclic groups may, by coincidence, have the same visibility measures. To determine whether a group of elements with the same *VFI* and *VFO* is one cyclic group (and not several), simply inspect the subset of the visibility matrix that includes the rows and columns of the group in question and no others. If this submatrix does not contain any zeros, then the group is indeed one cyclic group.

The cyclic groups found via this algorithm are referred to as the "cores" of the system. The largest cyclic group (the "Core") plays a special role in the architectural classification scheme, described next.

3.5 Classification of architectures

The method of classifying architectures is motivated in [1] and was discovered empirically. Specifically, Baldwin et al. found that a large percentage of the architectures they analyzed contained four distinct types of elements: 1) one large cyclic group, called the "Core," 2) "Control" elements that depend on other elements but are not themselves used by many, 3) "Shared" elements that are used by other elements but do not depend on that many other, and 4) "Periphery" elements that are not used by or depend on a large group of other elements.

From those empirical results, a core-periphery architecture was defined as one containing a single cyclic group of elements that is dominant in two senses: it is large relative to the architecture as a whole, and it is substantially larger than any other cyclic group. The empirical work also showed that not all architectures fit into the category of core-periphery. Some architectures (called "multi-core") have several similarly sized cyclic groups rather than one dominant one. Others (called "hierarchical") have only a few extremely small cyclic groups.

Based on the large dataset of software architectures analyzed in [1], the first classification boundary is set empirically to assess whether the largest cyclic group contains at least 5% of the total elements. Architectures that do not meet this test are labeled "hierarchical." Next, within the set of large-core architectures, a second classification boundary is applied to assess whether the largest cyclic group contains at least 50% more elements than the second largest cyclic group. Architectures that meet the second test are labeled "core-periphery"; those that do not (but have passed the first test) are labeled "multi-core." Figure 3 summarizes the classification scheme.

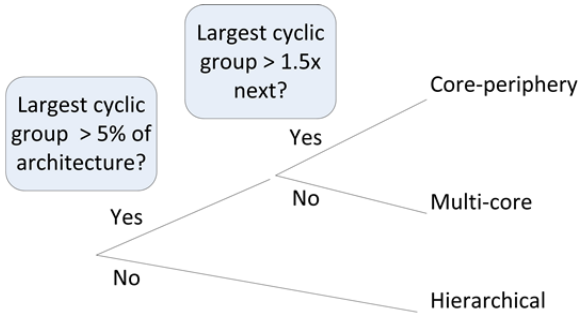


Figure 3. Architectural classification scheme.

3.6 Classification of elements

The elements of a core-periphery architecture can be divided into four basic groups:

- "Core" elements are members of the largest cyclic group and have the same *VFI* and *VFO*, denoted by VFI_C and VFO_C, respectively.
- "Control" elements have $VFI < VFI_C$ and $VFO \geq VFO_C$.
- "Shared" elements have $VFI \geq VFI_C$ and $VFO < VFO_C$.
- "Periphery" elements have $VFI < VFI_C$ and $VFO < VFO_C$.

Together the Core, Control, and Shared elements define the flow through of the architecture. (Note: For the classification of elements in hierarchical and multi-core architectures, see [1].)

3.7 Visualizing the architecture

Using the above classification scheme, a reorganized DSM can be constructed that reveals the "hidden structure" of the architecture by placing elements in the order Shared, Core, Periphery, and Control down the main diagonal of the DSM, and then sorting within each group by *VFI* descending then *VFO* ascending.

4. Telecom case

We now apply the described method to a real-world example using data from a business unit of a U.S. telecommunications supplier with global operations. The company (from here on referred to as "Telecom") has multibillion-dollar revenues and belongs to the Fortune 500. The business unit produces, configures, and sells professional radio systems to corporate and public-sector clients worldwide. A subset of the data was used

previously in a study on virtual decoupling for IT/business alignment [22].

4.1 Identifying the direct dependencies between the software applications

The Telecom dataset contains 103 software applications and 243 direct dependencies. We can represent that architecture as a directed network, with the applications as nodes and directed dependencies as links, and then convert that network into a DSM. Figure 4 contains what we call the “architect’s view," with dependencies indicated by dots. We also placed dots along the main diagonal, implying that each software application is dependent on itself.

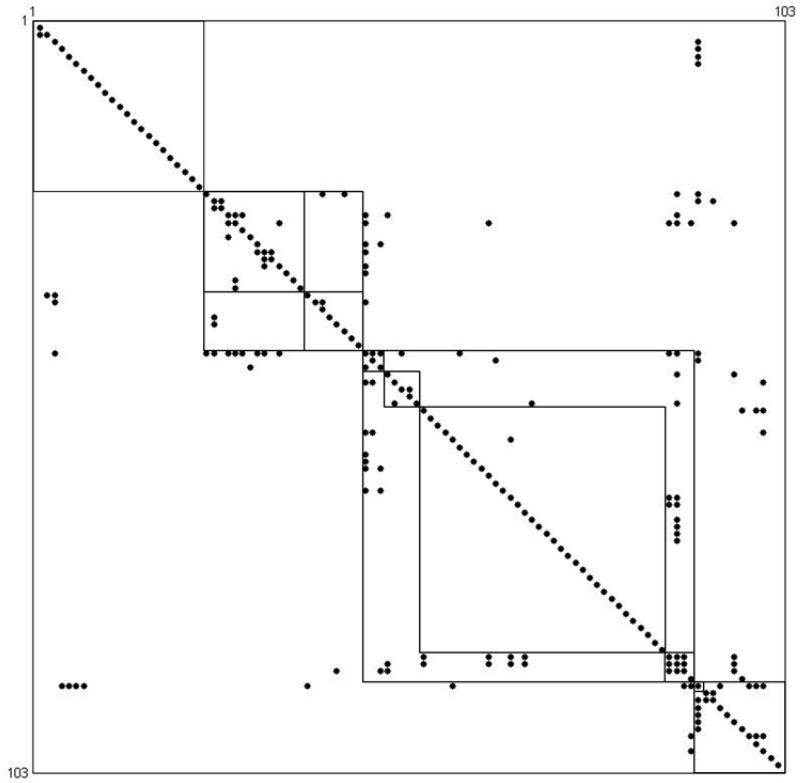


Figure 4. The Telecom DSM - architect's view.

The squares in Figure 4 represent business layers found in the company’s architecture descriptions. In order, from top left, we find “Bid & Quote,” “Finance,” “Operation,” and “Rollout.” Within these larger squares, smaller squares highlight different types of applications within each layer.

From the DSM, we calculate the Direct Fan-In (*DFI*) and Direct Fan-Out (*DFO*) measures by summing the rows and columns for each software application respectively. Table 1 shows, for example, that Software Application 1 (SA1) has a *DFI* of 2, indicating that one other application depends on it, and a *DFO* of 1, indicating that it depends

only on itself.

4.2 Computing the visibility matrix and constructing the coupling measures

The next step is to derive the visibility matrix by raising the first-order matrix (the architect's view) to successive powers, such that both the direct and all the indirect dependencies appear. The Visibility Fan-In (*VFI*) and Visibility Fan-Out (*VFO*) measures can then be calculated by summing the rows and columns in the visibility matrix for each respective software application. Table 1 shows that Software Application 1 (SA1) has a *VFI* of 55, indicating that 55 other applications directly or indirectly depend on it, and a *VFO* of 1, again indicating that it depends only on itself.

Table 1. A sample of Telecom Fan-In and Fan-Outs.

Software application	***DFI***	***DFO***	***VFI***	***VFO***
SA1	2	1	55	1
SA2	2	2	54	2
SA3	2	3	53	48
SA4	14	13	53	48
SA5	4	2	53	48
SA6	1	4	1	49
SA7	15	17	53	48
…	…	…	…	…
SA103	1	1	1	1

Using the *VFI* and *VFO* measures, we can calculate the propagation cost of the Telecom architecture:

$$\text{Propagation Cost} = \frac{\sum_{i=1}^{103} VFI_i}{103^2} = \frac{\sum_{i=1}^{103} VFO_i}{103^2} = 25\%$$

A propagation cost of 25% means that one-fourth of the architecture may be affected when a change is made to a randomly selected software application.

4.3 Identifying cyclic groups and classifying the architecture

To identify cyclic groups, we first ordered the list of software applications based on *VFI* descending and *VFO* ascending. We could then identify three possible cyclic groups: two groups each containing two software applications (*VFI*=54/*VFO*=2 and *VFI*=2/*VFO*=49) and one large group containing 35 applications (*VFI*=53/*VFO*=48). When inspecting the visibility submatrices of these possible clusters, we found that the groups containing two applications were not cyclic. In other words, these applications had ended up with the same *VFI* and

VFO by coincidence. The possible cluster with 35 software applications, however, proved to be a cyclic group, which we labeled as "Core." In Table 1, software applications 3, 4, 5, and 7 are all part of the Core. Because the Core makes up 34% of the architecture and because there are no other clusters, the architecture is classified as core-periphery, according to the classification scheme discussed earlier (Figure 5).

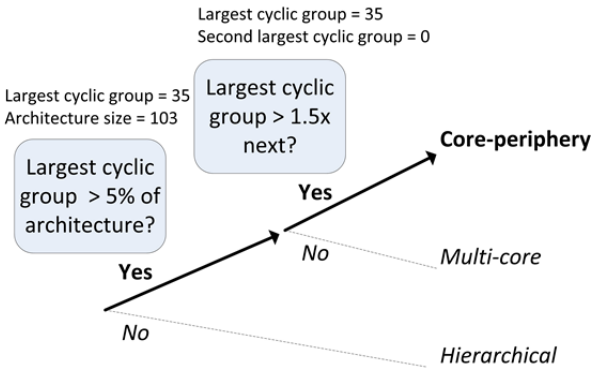


Figure 5. Telecom architecture classification.

4.4 Classifying the software applications and visualizing the architecture

After identifying applications that belong to the Core, the next step is to classify the remainder of the software as Shared, Periphery, or Control. To do so, we compare the *VFI* and *VFO* of each application with the VFI_C and VFO_C of the Core applications. Thirteen applications have a *VFI* that is equal to or larger than the VFI_C and a *VFO* that is smaller than the VFO_C, classifying them as Shared. Thirty-seven software applications have *VFI* and *VFO* numbers that are smaller than the Core, classifying them as Periphery. And 18 applications have a *VFI* that is smaller than the VFI_C and a *VFO* that is equal to or larger than the VFO_C, classifying them as Control. Table 2 summarizes those results.

Table 2. Telecom software applications classification.

Classification	No. of	% of total
Shared	13	12.6%
Core	35	34.0%
Periphery	37	35.9%
Control	18	17.5%

By sorting the original DSM using the different classifications, we can uncover the hidden structure of the architecture. First, the applications are sorted in the order of Shared, Core, Periphery, and Control. Then, within each

group the applications are ordered by *VFI* descending and *VFO* ascending.

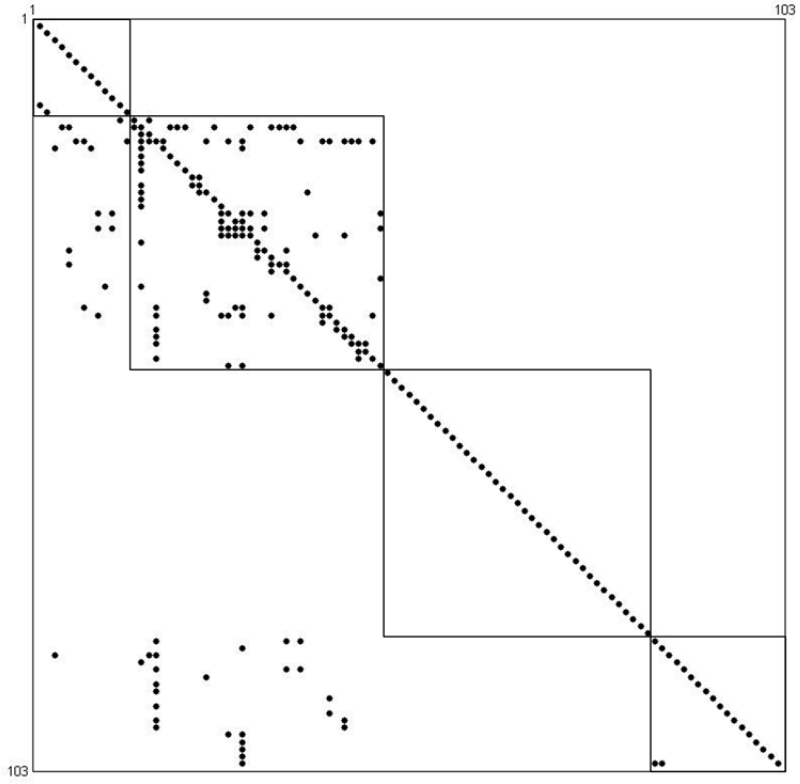


Figure 6. Telecom rearranged DSM.

From Figure 6, which shows the rearranged DSM, we see a large cyclic group of software applications, which appear in the second block down the main diagonal. Each element in this group both depends on and is depended on every other member of the group. These "Core" applications account for 34% of the elements. Furthermore, the Core, the applications depending on it ("Control"), and those it depends on ("Shared"), account for 64% of the architecture. The remaining applications are "Periphery," in that they have few relationships with other applications.

5. Discussion and research outlook

As presented in [1], the hidden structure method was designed based on empirical regularity from cases investigating large complex software systems. All those cases were focused on one software system at a time, independent of its surrounding environment, analyzing the dependencies between its source files. In other words, that work considered the internal coupling of a system. In this paper, the same method is tested on the dependencies between software applications; i.e., the current work considers the external coupling between applications.

For the Telecom case, the method revealed a hidden structure (thus presenting new facts) similar to those cases on software architecture investigated in previous studies. And the method also helped classify the architecture as core-periphery using the same rules and boundaries as in the previous cases. However, because this is only one set of data from one business unit, additional studies are needed.

Compared to other complexity, coupling, and modularity measures, the hidden structure method considers not only the direct network structure of an architecture but also takes into consideration the indirect dependencies between applications. Both these features provide important input for management decisions. For instance, applications that are classified as Periphery or Control are probably easier (and less costly) to modify because of the lower probability of a change spreading and affecting other applications. In contrast, applications that are classified as Shared or Core are more difficult to modify because of the higher probability of changes spreading to other applications. This information can be used in change management, project planning, risk analysis, and so on.

From Table 1, we see that software applications 1, 2, 3, 5, and 6 all have low Direct Fan-In (*DFI*) and Direct Fan-Out (*DFO*) numbers. As such, those applications might be considered as low risk when implementing changes (compared to applications 4 and 7, which have high *DFI* and *DFO* values). But if we also look at the Visibility Fan-In (*VFI*) and Visibility Fan-Out (*VFO*) numbers, which measure indirect dependencies, we see that applications 3 and 5 both belong to the Core of the architecture. Thus any change to one of those might spread to many other applications (even though they have few direct dependencies). The same goes for applications 1 and 2, which are classified as Shared. Therefore, we argue that the hidden structure method, which considers indirect dependencies, provides more valuable information for decision-making.

In our experience, we have found that many companies working with enterprise modeling have architecture blueprints that describe their application portfolio. Often, these are described using entity-relationship diagrams with boxes and arrows. When the entire application architecture is visualized using this type of model, the result is often a chaotic, messy picture that is difficult to interpret. Typically these models depict somewhat of a “spaghetti” architecture, with many applications and dependencies. This representation can be directly translated to the architect’s view DSM (cf. Figure 4). But this visualization does not really provide that much information either other than that applications are depending on each other in a complex network. With this representation (and the entity-relationship model), we can trace a dependency between two applications, which then can be used for decision-making (compare with the discussion above on *DFI*/*DFO* versus *VFI*/*VFO* measures). However, if we instead use

the hidden structure method and rearrange the DSM, as in Figure 6, we can actually see what applications are considered to be Core, Shared, Control, and Periphery. This gives us more insight about the structure of the architecture. Core applications are spread out across the business processes and they vary between small, very specific tools to large, central ERP systems and data warehouses. Without the hidden structure method, an architect would have difficulty uncovering this type of complex architecture.

Measures such as the propagation cost, the architecture flow through, and the size of the core can be useful when trying to improve an architecture. Future scenarios can be compared in terms of these metrics.

A first step in future research is to test the hidden structure method with more enterprise application architectures. This will provide valuable input either supporting the method as currently constructed or with improvement suggestions for future versions. Another step would be to extend the application area. Future research could involve tests with a "complete" enterprise architecture model, considering many different types of elements such as business processes and roles, software applications and services, and databases and servers. One hypothesis is that business layer elements typically are classified as Control, infrastructure elements as Shared, and software elements as Core. This, however, remains to be tested. If the hidden structure method does enable the useful visualization and classification of complete enterprise architecture descriptions, then it could be deployed to analyze the quality of a particular architecture and possibly help improve that quality in terms of the removal or addition of elements and dependencies.

Both in the previous work by Baldwin et al. and in this case, it can be seen that many architectures have a single large Core. A limitation of the hidden structure method is that it only shows which elements (in this case, software applications) belong to the Core but does not help in describing the structure of that Core. Thus, future research might extend the hidden structure method with a sub-method for that purpose. That sub-method could help identify the elements within the Core that are most important in terms of dependencies and cluster growth. The hypothesis is that there are some elements in a Core that bind the group together or that make the group grow faster. As such, removing these elements or reducing their dependencies (either to or from them) may decrease the size of the Core and thus the complexity of the architecture. Identifying these elements also helps pinpoint where the Core is most sensitive to change.

We have also seen in previous work with enterprise application architectures that these often contain non-directed dependencies, thus forming symmetric matrices that have special properties and behave differently than

those matrices containing directed dependencies. This could, for instance, be due to the nature of the link itself (as in social networks) or, as in most cases we have seen, due to imprecision in data (often because of the high costs of data collection). For companies, the primary concern is whether two applications are connected. The direction of the dependency is secondary. In one of our cases, the company had more than a thousand software applications but did not have an architecture model or application portfolio describing those applications. For that firm, collecting information about what applications it had and what those applications did was of primary importance. That process was costly enough, and consequently the directions of the dependencies between the applications were not a priority.

A lack of tool support is one reason for the high costs associated with data collection. In prior the work of Baldwin et al. [1], the analysis of internal coupling in a software system was supported by a tool that explored the source files and created a dependency graph automatically. In the enterprise architecture domain, however, such useful practical tools generally do not exist. Consequently, data collection requires considerable time. The most common methods are interviews and surveys of people (often managers) with already busy schedules. As such, future work needs to be directed towards data collection support in the enterprise architecture domain. Some work has already been done but is limited in either scope or application, as described in [25,26].

For the hidden structure method to be useful in practice, it needs to be incorporated into existing or future enterprise architecture tools. Most companies today already use modeling tools like Rational System Architect [27], BiZZdesign Architect [28], TrouxView [29], ARIS 9 [30], and MooD Business Architect [31] to describe their enterprise architecture. Thus, having a stand-alone tool that supports the hidden structure method is not feasible or very cost efficient. Moreover, if the method is integrated with current tools, companies can then perform a hidden structure analysis by re-using their existing architecture descriptions. The modeling software Enterprise Architecture Analysis Tool (EAAT) [32,33] is currently implementing the hidden structure method, and future studies will use it.

Last, but not least, the most important future work is to test the *VFI*/*VFO* metrics and the element classification (Shared, Core, Periphery, and Core) with performance outcome metrics. Doing so will help prove that the method is actually useful in architectural work. Currently, we can argue its benefits only with respect to other existing methods.

6. Conclusions

Although our method is used only in one case, the results suggests that it can reveal new facts about the architecture structure on an enterprise application level, equal to past results in the initial cases of single software

system. The analysis reveals that the hidden external structure of the software applications at the Telecom business unit can be classified as core-periphery with a propagation cost of 25%, architecture flow through of 64%, and core size of 34%. For the Telecom company, the architectural visualization and the computed coupling metrics can provide valuable input when planning architectural change projects (in terms of, for example, risk analysis and resource planning). In future work, we plan to provide more evidence supporting the usefulness of the hidden structure method, both by testing it with more software application architectures and by extending the scope to include additional complete enterprise architecture models consisting of elements such as business processes and infrastructure. Also, we plan to test the method outcome (architecture classification and fan-in/fan-out metrics) with performance outcome metrics such as change cost.

10. References

[1] C. Baldwin, A. MacCormack, and J. Rusnak, "Hidden Structure: Using Network Methods to Map System Architecture", Harvard Business School Working Paper, No. 13–093, May 2013.

[2] IEEE Standards Board, "IEEE Standard Glossary of Software Engineering Technology", Technical report, the Institute of Electrical and Electronics Engineers, Sep. 1990.

[3] T. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308–320, 1976.

[4] M. Halstead, "Elements of Software Science", Operating and Programming Systems Series, Elsevier Science Inc, 1977.

[5] S. Henry, D. Kafura, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, vol. 7, no. 5, pp. 510–518, 1981.

[6] N. Fenton and A. Melton, "Deriving Structurally Based Software Measures", Journal of Systems and Software, vol. 12, no. 3, pp. 177–187, 1990.

[7] A. L. Barabási, "Scale-Free Networks: A Decade and Beyond", Science, vol. 325, no. 5939, pp. 412-413, 2009.

[8] H. A. Simon, "The Architecture of Complexity", in Proc. of the American Philosophical Society, vol. 106, no. 6, pp. 467-482, 1962.

[9] C. Alexander, "Notes on the Synthesis of Form", Harvard University Press, 1964.

[10] C. Mead and L. Conway, "Introduction to VLSI Systems", Addison-Wesley Publishing Co., 1980.

[11] C. Baldwin and Kim Clark, "Design Rules, Volume 1: The Power of Modularity", MIT Press, 2000.

[12] D. Steward, "The Design Structure System: A Method for Managing the Design of Complex Systems", IEEE Transactions on Engineering Management, vol. 3, pp. 71-74, 1981.

[13] S. D. Eppinger, D.E. Whitney, R.P. Smith, and D.A. Gebala, "A Model-Based Method for Organizing Tasks in Product

Development", Research in Engineering Design, vol. 6, no. 1, pp. 1-13, 1994.

[14] M. Sosa, S. Eppinger, and C. Rowles, "A Network Approach to Define Modularity of Components in Complex Products", Transactions of the ASME, vol 129, pp. 1118-1129, 2007.

[15] A. MacCormack, C. Baldwin, and J. Rusnak, "Exploring the Duality Between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis", Research Policy, vol. 41, no. 8, pp. 1309-1324, 2006.

[16] M. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak, "Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases", in Proc. of the 7th Working IEEE/IFIP Conference on Software Architectures (WICSA7), 2008.

[17] R. Lagerström, P. Johnson, and M. Ekstedt, "Architecture Analysis of Enterprise Systems Modifiability – A Metamodel for Software Change Cost Estimation", Software Quality Journal, vol. 18, no. 4, pp. 437–468, 2010.

[18] M. Österlind, R. Lagerström, and P. Rosell, "Assessing Modifiability in Application Services Using Enterprise Architecture Models – A Case Study", in Proc. of the Trends in Enterprise Architecture Research (TEAR) and Practice-Driven Research on Enterprise Transformation (PRET) workshop, Springer Berlin Heidelberg, pp. 162-181, 2012.

[19] B. Boehm, R. Madachy, and B. Steece, "Software Cost Estimation with COCOMO II", Prentice Hall PTR, 2000.

[20] The IT Governance Institute, "Control Objectives for Information and Related Technology (COBIT) ", vol. 4.1, Technical Report, 2007.

[21] P. Oman, J. Hagemeister, and D. Ash, "A Definition and Taxonomy for Software Maintainability", Technical Report, Software Engineering Lab, 1992.

[22] S. Aier and R. Winter, "Virtual Decoupling for IT/Business Alignment – Conceptual Foundations, Architecture Design and Implementation Example", Business & Information Systems Engineering, vol. 1, no. 2, pp. 150-163, 2009.

[23] D. Dreyfus and G. Wyner, "Digital Cement: Software Portfolio Architecture, Complexity, and Flexibility", in Proc. of Americas Conference on Information Systems (AMCIS), Association for Information Systems, 2011.

[24] T. Widjaja, J. Kaiser, D. Tepel, P. Buxmann, "Heterogeneity in IT Landscapes and Monopoly Power of Firms: A Model to Quantify Heterogeneity", in Proc. of International Conference on Information Systems (ICIS 2012), Orlando, USA, 2012.

[25] H. Holm, M. Buschle, R. Lagerström, and M. Ekstedt, "Automatic Data Collection for Enterprise Architecture Models", Software & Systems Modeling, Online first, 2012.

[26] M. Buschle, S. Grunow, F. Matthes, M. Ekstedt, M. Hauder, and S. Roth, "Automating Enterprise Architecture Documentation using an Enterprise Service Bus", in Proc. of the 18th Americas Conference on Information Systems (AMCIS), 2012.

[27] IBM, Rational System Architect, www.ibm.com/software/products/us/en/ratisystarch, accessed May 2013.

[28] BiZZdesign, BiZZdesign Architect, www.bizzdesign.com/tools/bizzdesign-architect, accessed May 2013.

[29] Troux solutions, TrouxView™ Enterprise Portfolio Management, www.troux.com, accessed May 2013.

[30] Software AG, ARIS 9, www.softwareag.com/corporate/products/aris_platform, accessed May 2013.

[31] MooD International, MooD Business Architect, www.moodinternational.com/moodplatform, accessed May 2013.

[32] Industrial Information and Control Systems - the Royal Institute of Technology, The Enterprise Architecture Analysis Tool, www.ics.kth.se/eaat, accessed May 2013.

[33] M. Buschle, J. Ullberg, U. Franke, R. Lagerström, and T. Sommestad, "A Tool for Enterprise Architecture Analysis using the PRM Formalism", in Information Systems Evolution, pp. 108-121, 2011.