Visualizing and Measuring Enterprise Architecture: An Exploratory BioPharma Case

Robert Lagerström, Carliss Baldwin, Alan Maccormack, David Dreyfus

Visualizing and Measuring Enterprise Architecture: An Exploratory BioPharma Case

Robert Lagerström[1,2], Carliss Baldwin[1], Alan MacCormack[1] and David Dreyfus[3]

[1] Harvard Business School, Soldiers Field Park, Boston, MA 02163, United States
[2] The Royal Institute of Technology, Osquldas väg 10, 10044, Stockholm, Sweden
[3] Boston University, 595 Commonwealth Avenue, Boston, MA 02215, United States
[1] {cbaldwin, amaccormack}@hbs.edu
[2] robertl@ics.kth.se
[3] ddreyfus@bu.edu

Abstract. We test a method that was designed and used previously to reveal the hidden internal architectural structure of software systems. The focus of this paper is to test if it can also uncover new facts about the components and their relationships in an enterprise architecture, i.e., if the method can reveal the hidden external structure between architectural components. Our test uses data from a biopharmaceutical company. In total, we analyzed 407 components and 1,157 dependencies. Results show that the enterprise structure can be classified as a core-periphery architecture with a propagation cost of 23%, core size of 32%, and architecture flow through of 67%. We also found that business components can be classified as control elements, infrastructure components as shared, and software applications as belonging to the core. These findings suggest that the method could be effective in uncovering the hidden structure of an enterprise architecture.

Key words: Enterprise Architecture, Design Structure Matrices, Enterprise Modeling, Architecture Visualization

1 Introduction

Managing software applications has become a complex undertaking. Today, achieving effective and efficient management of the software application landscape requires the ability to visualize and measure the current status of the enterprise architecture. To a large extent, that huge challenge can be addressed by introducing tools such as enterprise architecture modeling as a means of abstraction.

In recent years, Enterprise Architecture (EA) has become an established discipline for business and software application management [1]. EA describes the fundamental artifacts of business and IT as well as their interrelationships [1-4]. Architecture models constitute the core of the approach and serve the purpose of making the complexities of the real world understandable and manageable [3]. Ideally, EA aids the stakeholders of the enterprise to effectively plan, design, document, and communicate IT and business related issues; i.e. they provide decision support for the stakeholders [5].

In relation to supporting decisions, a key underlying assumption of EA models is that they should provide aggregated knowledge beyond what was put into the model in the first place. For instance, the discipline of software architecture does more than just keep track of the set of source files in an application; it also provides information about the dependencies between those files. More broadly, an EA covers the dependencies between the business and the software applications so that, for example, conclusions can be drawn about the consequences in the enterprise should a specific application be removed or changed.

Enabling this type of analysis is extremely important for EA to provide value to stakeholders. Unfortunately, though, EA frameworks rarely explicitly state the kinds of analyses that can be performed given a certain model, nor do they provide details on how the analysis should be performed [6].

In [7], Baldwin et al. present a method based on Design Structure Matrices (DSMs) and classic coupling measures to visualize the hidden structure of software system architectures. This method has been tested on numerous software releases for large systems (such as Linux, Mozilla, Apache, and GnuCash) but not on enterprise architectures with a potentially large number of interdependent components. This paper performs such a test using data from a biopharmaceutical company (referred to as BioPharma). The data consisted of a total of 407 architecture components and 1,157 dependencies.

We find that the BioPharma enterprise architecture can be classified as core-periphery, meaning that 1) there is one cyclic group (the "Core") of architecture components 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 23%, meaning that almost one-fourth of the architecture may be affected when a change is made to a randomly selected component in the architecture. In addition, we find that the Core contains 132 architecture components, which embody 32% of the architecture. And lastly, the analysis uncovers that the architecture flow through accounts for as much as 67% of the architecture, meaning that more than half of the components 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 biopharmaceutical 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 argue that the EA frameworks available today do not provide support for architecture analysis. Then we present system architecture approaches that aim to solve these problems.

2.1 Enterprise Architecture Analysis

As stated in the introduction, EA frameworks rarely supply the exact procedure or algorithm for performing a certain analysis given an architecture model. But most do recognize the need to provide special-purpose models as well as different viewpoints intended for different stakeholders. Unfortunately, however, most viewpoints are designed from a model-entity point of view rather than from an analysis-concern point of view. Thus, they cannot perform the visualizing and measuring of the modularity or coupling of an architecture in a straightforward manner. The Department of Defense Architecture Framework (DoDAF) [8], for instance, provides products (i.e., viewpoints) such as "systems communications description," "systems data exchange matrix," and "operational activity model." These are all viewpoints based on a delimitation of elements of a complete metamodel. The Zachman framework presented in [2, 9] does connect model types describing different aspects (Data, Function, Network, People, Time, and Motivation) with abstractly described stakeholders (Strategists, Executive Leaders, Architects, Engineers, and Technicians), but it does not provide any deeper insights as to how different models should be used for analysis. The Open Group Architecture Framework (TOGAF) [4] explicitly states the concerns for each suggested viewpoint, but it does not describe the exact mechanism for analyzing the stated concerns. With respect to modularity, the most appropriate viewpoints provided would, according to TOGAF, arguably be the "software engineering view," "systems engineering view," "communications engineering view," and "enterprise manageability view." The descriptions of these views contain statements such as, "the use of standard and self-describing languages, e.g. XML, is good in order to achieve easy to maintain interface descriptions." What is not included, however, is the exact interpretation of such statements when it comes to architectural models or how they relate to the analysis of, for example, the flexibility of a system as a whole. Moreover, these kinds of "micro theories" are only exemplary and do not claim to provide a complete theory for modularity or similar concerns.

Other analysis frameworks focus on the assessment of non-functionality qualities such as availability [10], interoperability [11], modifiability [12], and security [13]. These frameworks use Bayesian analysis or probabilistic versions of the Object Constraint Language for enterprise modeling. They do not, however, provide any analysis capabilities when it comes to revealing the hidden structure of an enterprise architecture. Also, the visualization capabilities of these frameworks are limited because they all use entity-relationship modeling without any proper views dealing with large complex models.

2.2 System Architecture Visualization

If we instead turn to the discipline of system architecture, we find work that aims to solve the issue of architecture analysis and visualization. Studies that attempt to characterize the architecture of complex systems often employ network representations [14]. Specifically, they focus on identifying the linkages that exist between the different elements (nodes) in a system [15, 16]. 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 [17, 18].

Studies that use network methods to measure modularity have typically focused 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 [19-21], 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, [22] uses DSMs and the metric "propagation cost" to compare software system architectures. DSMs have been used to visualize architectures and to measure the coupling of the internal design of single software systems.

3 Method Description

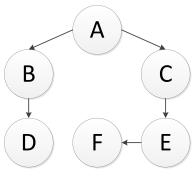
The method used for architecture network representation is based on and extends the classic notion of coupling. Specifically, after identifying the coupling (dependencies) between the elements in a complex architecture, the method analyzes the architecture in terms of hierarchical ordering and cycles, 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. The following subsections present a method that makes this "hidden structure" visible and describe metrics that can be used to compare architectures and track changes in architecture structures over time. (Note: A more detailed method description can be found in "Hidden Structure: Using Network Methods to Map System Architecture" by Baldwin et al. [7].)

3.1 Identify the Direct Dependencies and Compute the Visibility Matrix

The architecture of a complex system can be represented as a directed network composed of *N* elements (nodes) and the directed dependencies (links) between them. Fig. 1 contains an example (taken from [22]) 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 Directed Graph	Design Structure Matrix	Visibility Matrix $V=\sum M^n$; n=[0,4]
		

	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

	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**

Fig. 1. A directed graph, Design Structure Matrix (DSM), and Visibility matrix example.

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 (Fig. 1), 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 [22]. 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.

3.2 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 the visibility matrix (Fig. 1), 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, the Propagation Cost (PC) is defined as the density of the visibility matrix. Intuitively, it 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*) as described in Eq. 1.

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

3.3 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 we sort the elements, first by *VFI* descending then by *VFO* ascending. Next we proceed 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. But 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), we 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.

3.4 Classification of Architectures

The method of classifying architectures is motivated in [7] 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 others, 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 [7], 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." Fig. 2 summarizes the classification scheme.

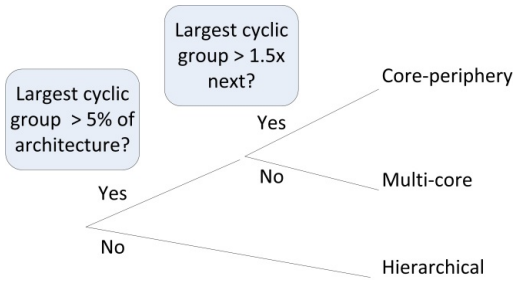


Fig. 2. Architectural classification scheme.

3.5 Classification of Elements and Visualizing the Architecture

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 [7].)

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 of 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 BioPharma Case

We now apply the described method to a real-world example of a U.S. biopharmaceutical company (BioPharma). Data were collected at the research division by examining strategy documents, entering architectural information into a repository, using automated system scanning techniques, and conducting a survey. A subset of the data employed for the analysis presented in this paper was previously used in the study "Digital Cement: Software Portfolio Architecture, Complexity, and Flexibility," by Dreyfus and Wyner [23], with a more extensive exploration in [24].

4.1 Identifying the Direct Dependencies between the Architecture Components

The BioPharma dataset contains 407 architecture components and 1,157 dependencies. The architectural components are divided as follows: eight "business groups," 191 "software applications," 92 "schemas," 49 "application servers," 47 "database instances," and 20 "database hosts" (cf. Table 1).

Table 1. Component and dependency types in the BioPharma case.

Component type	No. of
Business Group	8
Software Application	191
Schema	92
Application Server	49
Database Instance	47
Database Host	20

Dependency type	No. of
Communicates With	742
Runs On	165
Is Instantiated By	92
Uses	158

The dependencies between the architecture components belong to the following types (cf. Table 1): 742 "communicates with" (bidirectional), 165 "runs on" (unidirectional), 92 "is instantiated by" (unidirectional), and 158 "uses" (unidirectional).

We can represent this architecture as a directed network, with the architecture components as nodes and dependencies as links, and then convert that network into a DSM. Fig. 3 contains the "architect's view," with dependencies indicated by dots. (Note: We placed dots along the main diagonal, implying that each architecture component is dependent on itself.) The squares in Fig. 3 represent the architecture components, which have been ordered in terms of typical enterprise architecture layers (from top left to bottom right): business groups, software applications, schemas, applications servers, database instances, and database hosts.

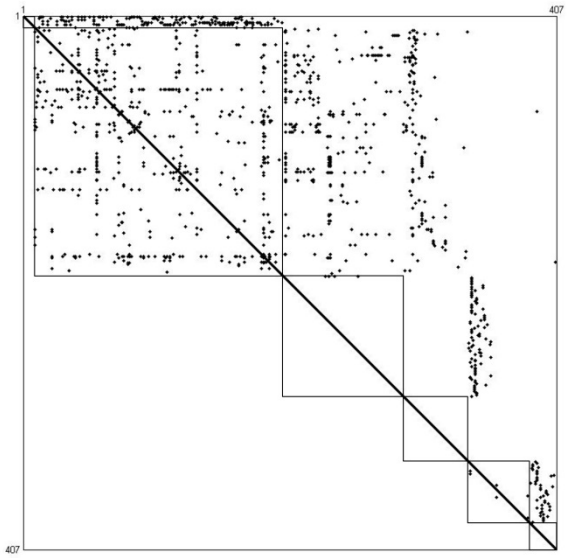


Fig. 3. The BioPharma DSM – architect's view.

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 2 shows, for example, that Architecture Component 324 (AC324) has a *DFI* of four, indicating that three other components depend on it, and a *DFO* of 2, indicating that it depends on only one component other than 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 architecture component. Table 2 shows that Architecture Component 403 (AC403), for example, has a *VFI* of 173, indicating that 172 other components directly or indirectly depend on it, and a *VFO* of 2, indicating that it directly or indirectly depends on only one component other than itself.

Table 2. A sample of Biopharma Fan-In and Fan-Outs.

Architecture component	***DFI***	***DFO***	***VFI***	***VFO***
AC324	4	2	140	3
AC333	2	3	139	265
AC347	2	2	140	3
AC378	8	23	139	265
AC403	29	2	173	2
AC769	1	6	1	267
AC1030	3	2	3	2

Using the *VFI* and *VFO* measures, we can calculate the propagation cost of the BioPharma architecture, as described in Eq. 2.

$$\text{Propagation Cost} = \frac{\sum_{i=1}^{407} VFI_i}{407^2} = \frac{\sum_{i=1}^{407} VFO_i}{407^2} = 23\% \qquad (\mathbf{2})$$

A propagation cost of 23% means that almost one-fourth of the architecture may be affected when a change is made to a randomly selected architecture component.

4.3 Identifying Cyclic Groups and Classifying the Architecture

To identify cyclic groups, we first ordered the list of architecture components based on *VFI* descending and *VFO* ascending. We could then identify 15 possible cyclic groups. When inspecting the visibility submatrices of these possible clusters, we found that most groups were not cyclic. In other words, these applications had ended up with the same *VFI* and *VFO* by coincidence. But one possible cluster had 132 architecture components and proved to be the largest cyclic group, which we labeled as "Core." In Table 2, Architecture Components 333 and 378 are part of the Core. Because the Core makes up 32% of the architecture and because the second largest cluster contains only four components, the architecture is classified as core-periphery, according to the classification scheme discussed earlier (cf. Fig. 2).

4.4 Classifying the Components and Visualizing the Architecture

After identifying components that belong to the Core, the next step is to classify the remainder of the architecture components as Shared, Periphery, or Control. To do so, we compare the *VFI* and *VFO* of each component with the VFI_C and VFO_C of the Core components. A total of 133 components 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. A total of 135 architecture components have *VFI* and *VFO* numbers that are smaller than the Core, classifying them as Periphery. And seven components 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 3 summarizes those results.

Table 3. BioPharma architecture component classification.

Classification	No. of	% of total
Shared	133	33%
Core	132	32%
Periphery	135	33%
Control	7	2%

By sorting the original DSM using the different classifications, we can uncover the hidden structure of the architecture. First, the components are sorted in the order of Shared, Core, Periphery, and Control. Then, within each group the components are ordered by *VFI* descending and *VFO* ascending.

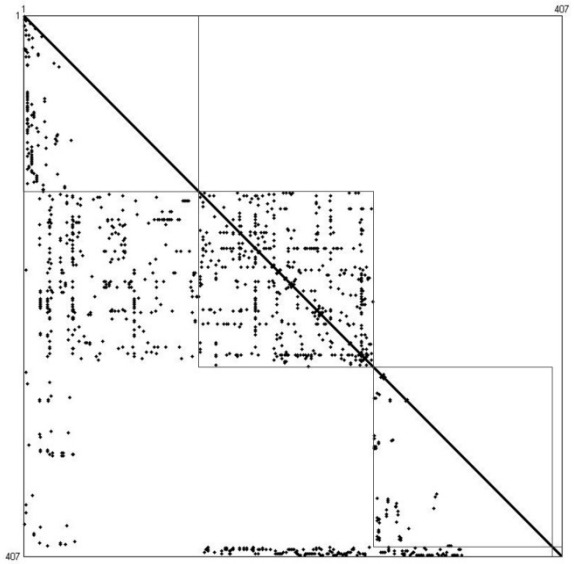


Fig. 4. BioPharma rearranged DSM.

From Fig. 4, which shows the rearranged DSM, we see a large cyclic group of architecture components that appear in the second block down the main diagonal. Each element in this group both depends on and is dependent on every other member of the group. These "Core" components account for 32% of the elements.

Furthermore, the Core, the components depending on it ("Control"), and those it depends on ("Shared"), account for 67% of the architecture. The remaining components are "Periphery," in that they have few relationships with other components.

If we examine where the different types of components in the architecture end up after the classification and rearrangement, we find the following: The Shared category contains only infrastructure components (schema, application server, database instance, and database host); the Core consists of only software-application elements; the Periphery contains a mix; and the Control category consists of just business-group components (see Table 4).

Table 4. Distribution of architecture components between classification categories.

	Business group	**Software application**	**Schema**	**Application server**	**Database instance**	**Database host**
Shared	0	0	83	27	15	8
Core	0	132	0	0	0	0
Periphery	1	59	9	22	32	12
Control	7	0	0	0	0	0

5 Discussion and Research Outlook

As presented in [7], the hidden structure method was designed based on the 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 architecture components; i.e., the current work considers the external coupling between not only software applications but also other enterprise architecture components.

For the BioPharma case, the method revealed a hidden structure (thus presenting new facts) similar to those cases on software systems 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 company, additional studies are needed. We present one such study using enterprise application architecture data from a Telecom company in [25].

Compared to many other complexity, coupling, and modularity measures, the hidden structure method considers not only the direct network structure of an architecture but also takes into account the indirect dependencies between components (not unlike some measures used in social networks). Both these features provide important input for management decisions. For instance, components 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 components. In contrast, components that are classified as Shared or Core are more difficult to modify because of the higher probability of changes having an impact elsewhere. This information can be used in change management, project planning, risk analysis, and so on.

From just the architect's view (cf. Fig. 3), we see some of the benefits of using Design Structure Matrices for enterprise architecture visualization. If the matrix elements are arranged in an order that comes naturally for most companies, with the business layer at the top, infrastructure at the bottom, and software in between, we see that 1) the business groups depend on the software applications, 2) the software applications communicate with each other in what looks like a clustered network of dependencies, 3) the software applications depend on the schemas and application servers, 4) the schemas depend on the database instances, and 5) the database instances depend on the database hosts. Although these observations are neither new nor surprising, they do help validate that the components in the investigated architecture do interact as expected.

From Table 2, we see that architecture components 324, 333, 347 769, and 1030 all have rather low Direct Fan-In (*DFI*) and Direct Fan-Out (*DFO*) numbers. As such, those components might be considered as low risk when implementing changes. But if we also look at the Visibility Fan-In (*VFI*) and Visibility Fan-Out (*VFO*) numbers, which measure indirect dependencies, we see that application 333 belongs to the Core of the architecture. Thus any change to it might spread to many other components (even though it has few direct dependencies). The same goes for components 324, 347, and 403, 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 organization, often with entity-relationship diagrams containing boxes and arrows. When the entire architecture is visualized using this type of model, however, the result is typically a "spaghetti" tangle of many components and dependencies that are difficult to interpret. But this representation can be translated directly to the architect's view DSM (cf. Fig. 3), which, along with the entity-relationship model, can be used to trace a dependency between two components, thus enabling better decision-making (compare with the discussion above on *DFI*/*DFO* versus *VFI*/*VFO* measures). Moreover, if we instead use the hidden structure method and rearrange the DSM, as in Fig. 4, we can actually see what components are considered to be Core, Shared, Control, and Periphery, which gives us much more insight about the structure of the architecture. Lastly, 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 because future scenarios can be compared in terms of these metrics.

In the explored BioPharma case, we found that the Control category contains only business groups; the Core consists of only software applications; the Shared elements are all infrastructure-related components (schemas, application servers, database instances, and database hosts); and the Periphery category contains a mix of all types. These results provide support for the method, as we would expect that the business controls the underlying components in the architecture (e.g. a business group depends on the software it uses but not the other way around). Also, infrastructure components such as databases are supposed to be shared among the applications in a sound architecture.

A first step in future research is to test the hidden structure method with additional enterprise architectures, like the one in [25]. This will provide valuable input either

supporting the method as currently constructed or with suggested improvements for future versions.

Both in the previous work by Baldwin et al. [7], Lagerström et al. [25], and in this case, the architectures studied have a single large Core. A limitation of the hidden structure method is that it only shows which elements belong to the Core but does not help in describing the inner structure of that Core. Thus, future research might extend the hidden structure method with a sub-method that 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 might also help pinpoint where the Core is most sensitive to change.

We have also seen in previous work that enterprise application architectures often contain non-directed dependencies, thus forming symmetric matrices that have special properties and behave differently from matrices with 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, it could be due to imprecision in the data (often because of the high costs of data collection). For companies, the primary concern is whether two applications are connected, and 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 them. 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 direction of the dependencies between the applications was not a priority.

Effective tools could help lower the high costs associated with data collection. In the prior work of Baldwin et al. [7], 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, 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 [26, 27].

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 [28] and BiZZdesign Architect [29] to describe their enterprise architecture. Thus, having a stand-alone tool that supports the hidden structure method would not be 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) [30] 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, Control, Periphery, and Core) with performance outcome metrics such as change cost. 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 in only one case, the results suggests that it can reveal new facts about the architecture structure on an enterprise level, equal to past results in the initial cases of single software systems. The analysis reveals that the hidden external structure of the architecture components at BioPharma can be classified as core-periphery with a propagation cost of 23%, architecture flow through of 67%, and core size of 32%. For BioPharma, 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). Also the analysis shows that business components are Control elements, infrastructure components are Shared elements, and software applications are in the Core, thus providing verification that the architecture is sound.

References

1. Ross, J.W., Weill, P., Robertson, D.: Enterprise Architecture As Strategy: Creating a Foundation for Business Execution. Harvard Business School Press (2006)
2. Zachman, J. A.: A Framework for Information Systems Architecture. IBM Systems Journal 26.3, 276--292 (1987)
3. Winter, R., Fischer, R.: Essential Layers, Artifacts, and Dependencies of Enterprise Architecture. Journal of Enterprise Architecture 3.2, 7--18 (2007)
4. The Open Group: The Open Group Architecture Framework (TOGAF). Version 9, The Open Group (2009)
5. Kurpjuweit, S., Winter, R.: Viewpoint-based Meta Model Engineering. In: the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures: Concepts and Applications, pp. 143--161 (2007)
6. Johnson, P., Lagerström, R., Närman, P., Simonsson, M.: Enterprise Architecture Analysis with Extended Influence Diagrams. Information Systems Frontiers 9.2-3, 163--180 (2007)
7. Baldwin, C., MacCormack, A., Rusnack, J.: Hidden Structure: Using Network Methods to Map System Architecture. Harvard Business School Working Paper, no. 13-093, May (2013)
8. Department of Defense Architecture Framework Working Group: DoD Architecture Framework. Version 1.5, Technical report, Department of Defense, USA (2007)
9. Zachman International, http://www.zachmaninternational.com
10. Franke, U., Johnson, P., König, J., Marcks von Würtemberg, L.: Availability of Enterprise IT Systems: An Expert-based Bayesian Framework. Software Quality Journal 20.2, 369--394 (2012)
11. Ullberg, J., Johnson, P., Buschle, M.: A Language for Interoperability Modeling and Prediction. Computers in Industry 63.8, 766--774 (2012)
12. Lagerström, R., Johnson, P., Höök, D.: Architecture Analysis of Enterprise Systems Modifiability: Models, Analysis, and Validation. Journal of Systems and Software 83.8, 1387--1403 (2010)

13. Sommestad, T., Ekstedt, M., Holm, H.: The Cyber Security Modeling Language: A Tool for Assessing the Vulnerability of Enterprise System Architectures. IEEE Systems Journal, Online-first (2013)
14. Barabási, A.: Scale-Free Networks: A Decade and Beyond. Science 325.5939, 412--413 (2009)
15. Simon, H. A.: The Architecture of Complexity. In: the American Philosophical Society 106.6, pp. 467--482 (1962)
16. Alexander, C.: Notes on the Synthesis of Form. Harvard University Press (1964)
17. Mead, C., Conway, L.: Introduction to VLSI Systems. Addison-Wesley Publishing Co. (1980)
18. Baldwin, C., Clark, K.: Design Rules, Volume 1: The Power of Modularity. MIT Press (2000)
19. Steward, D.: The Design Structure System: A Method for Managing the Design of Complex Systems. IEEE Transactions on Engineering Management 3, 71--74 (1981)
20. Eppinger, S. D., Whitney, D.E., Smith, R.P., Gebala, D. A.: A Model-Based Method for Organizing Tasks in Product Development. Research in Engineering Design 6.1, 1--13 (1994)
21. Sosa, M., Eppinger, S., Rowles, C.: A Network Approach to Define Modularity of Components in Complex Products. Transactions of the ASME 129, 1118--1129 (2007)
22. MacCormack, A., Baldwin, C., Rusnak, J.: Exploring the Duality Between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis. Research Policy 41.8, 1309--1324 (2006)
23. Dreyfus, D., Wyner, G.: Digital Cement: Software Portfolio Architecture, Complexity, and Flexibility. In: the Americas Conference on Information Systems (AMCIS), Association for Information Systems (2011)
24. Dreyfus, D.: Digital Cement: Information System Architecture, Complexity, and Flexibility. PhD Thesis. Boston University Boston, MA, USA, ISBN: 978-1-109-15107-7 (2009)
25. Lagerstrom, R., Baldwin, C. Y., MacCormack, A., Aier, S.: Visualizing and Measuring Enterprise Application Architecture: An Exploratory Telecom Case. Harvard Business School Working Paper, no. 13–103, June (2013)
26. Holm, H., Buschle, M., Lagerström, R., Ekstedt, M.: Automatic Data Collection for Enterprise Architecture Models. Software & Systems Modeling, Online first (2012)
27. Buschle, M., Grunow, S., Matthes, F., Ekstedt, M., Hauder, M., Roth, S.: Automating Enterprise Architecture Documentation using an Enterprise Service Bus. In: the 18th Americas Conference on Information Systems (AMCIS) (2012)
28. IBM Rational System Architect, www.ibm.com/software/products/us/en/ratisystarch
29. BiZZdesign Architect, www.bizzdesign.com/tools/bizzdesign-architect
30. The Enterprise Architecture Analysis Tool, www.ics.kth.se/eaat