Abstract

We propose a novel methodology for analysing change propagation in software using the domain-level behavioural model of a system. We hypothesize that change propagation analysis is feasible based purely on the information visible and understandable to domain experts, trading some accuracy for productivity. Such a method is independent of formal architectural representations and may be practical for applications with heterogeneous subsystems, or missing or undocumented source code. In this paper we introduce the first phase of the methodology: creating and evaluating a connection graph of conceptual relationships between user interface components. We provide results of case studies on two web-based systems which illustrate how our methodology can be applied, and how discovered conceptual relationships match the architectural dependencies.

1 Introduction

Change propagation is the phenomenon whereby a change in one software element may lead to changes to others. Analyzing the effects of this phenomenon plays an important role in the larger picture of software evolution [10]. In our research we ignore changes and change dependencies for low-level elements such as statements and expressions in programs in-the-small. Instead we focus on software components in component-based architectures [13].

Software maintenance includes three stages: initial development, evolution and servicing [2], and each stage requires the introduction, update and deletion of software components [6]. Changing an individual software component has an impact on its host system since it often necessitates changes to other components, both to avoid bugs arising from inconsistencies and to maintain software quality [1]. The resulting extra development and testing can significantly increase the cost of maintenance and push software projects beyond deadlines. Therefore, change propagation is considered a risk in software maintenance, and it increases the difficulty of making decisions about change plans.

What is needed is a pragmatic methodology for change propagation analysis which fulfills the following criteria:

- **Simplicity and usability**: Change propagation analysis is required as part of maintenance management. For typical enterprise applications, domain experts are the primary requesters for software changes. Domain experts are rarely software architects, but they have understanding of system functionalities provided to end users. Simplicity and usability of change analysis for domain expert will reduce the time spent by software architects for change propagation analysis, and in some cases decrease the cost of maintenance.

- **Practicality**: This approach should be applicable to typical enterprise software by considering software environmental difficulties including outdated design artifacts, missing source codes or heterogeneous systems.

- **Time efficiency**: The methodology should be applicable in an acceptable time frame based on the scale and complexity of the system.

In the literature, we identify three major approaches for change propagation analysis. Firstly there are document-based approaches which rely on studying the design artifacts to analyse inter-system dependencies [4–6, 15]. These
approaches are not practical for cases where the design artifacts are not accessible or reliable, such as legacy systems whose documents do not reflect many enhancements and ad hoc developments.

Secondly there are approaches based on source code analysis which extract the dependencies between system components by analysing their source codes [3, 9]. These approaches can be automated and can be fairly accurate when there are some predefined patterns in the source code. However, they are complex and are rarely usable by domain experts, as they require software architecture knowledge and understanding of the source code. Also, they are not practical for systems where all or some of the source code is missing, is not accessible or is multi-lingual (e.g. parts of the system are in C++ and parts in Perl).

Finally there are heuristic approaches, which use maintenance history records to find dependencies between system components. If two components are frequently modified at the same time or by the same programmer, there is some relationship between them, and changing one of them usually requires alteration of the other one [7, 11, 14]. This approach is reasonably time efficient and simpler than source code analysis; however, it is not practical for systems which are at the initial development stage or where maintenance history is not available.

Hence as described above, there are some drawbacks for existing approaches. The state of the art is primarily focused on increasing the accuracy of change propagation analysis in order to reduce the risk of software failure and increase the reliability of the system. However, the provided approaches usually require tools specified for different architectures or high level expertise in software engineering and implementation knowledge. Such factors make these approaches difficult to use for typical enterprise applications.

We propose a new approach for estimating the scope of change propagation. The proposed approach distinguishes between relationships resulting from domain functionalities, and those resulting from architectural dependencies. It trades off accuracy in favor of gaining productivity.

This paper is structured as follows. We begin (section 2) by presenting our domain-based approach for system analysis. Then we present two case studies (sections 3 and 4) which compare the results of the domain-based system analysis with actual architecture dependencies derived from source code analysis. In section 5 we discuss how and where this approach will be applicable, and the paper concludes (section 6) with a discussion of further work.

2 Domain-based System Analysis

A system functional specification is based on the system’s behaviour from the domain user’s point of view. This view includes functions provided by the system, their required inputs and expected outputs. This specification is concerned only with the abstract interface of component software, not with its concrete implementation. We hypothesize that information provided at this level of abstraction is adequate for estimating the scope of change propagation between system functionalities understandable to domain users.

In our work we use the following terminology. A domain variable is a unit of data which has a clear identity at the domain level. For example, a username or a date. A domain functionality is a specific functionality which the system provides to its users. For example, being able to authenticate, or book a room. A user interface component (UIC) is a part of the user interface with one or more domain functionalities. For example, in a web-based system, we might have a web page that provides a sitemap, or the ability to authenticate. Whereas the previous three concepts are at the domain level, i.e., they are things that a domain expert will be familiar with, the next concepts are at the architectural level. An architecture component is a unit of source code such as a class or a library. An architecture dependency is a reference between source codes of two architecture component. For example, a call to a method of a class.

Knowledge of intersystem connections and dependencies is vital for change propagation analysis. Dependencies between architectural components can cause bugs and functionality failures when changes are made to the system. In addition, the logical relationships between system functionalities need to be considered to maintain the coherence in the system functionalities. Such relationships may or may not be wired directly in the source code. For example, there are heterogeneous systems which include related functionalities implemented using different technologies or still communicating to legacy components. In such cases, the source code analysis may not show dependencies between these components as there are no direct reference between their source codes. However, reviewing the behavioral model of the system can reveal logical relationships between these components.

In principle, the purpose of an information system can be defined as providing a set of functionalities to the domain users. These functionalities are implemented in the architecture as source code, such as subsystems, collections of classes and libraries. We call them architectural components to distinguish them from finer-grain software elements such as functions and statements. Such an architectural component should exist individually or in collaboration with others. It provides a subset of functionalities provided to the domain user and it may require functionalities from other components. However, a system may include functional elements and nonfunctional properties invisible to its domain users. Security is an example of a nonfunc-
tional property realised at levels of abstraction not necessarily visible to domain users. In general, we therefore need to restrict the discourse to components and relationships between them, that are visible to the domain user.

We utilize information visible to domain users to analyze logical relationships, and to estimate the existence of respective architectural dependencies. The outcome will be a set of conceptual connections and a subset of architectural dependencies that are the direct result of logical relationships between system functionalities. The adequacy of the achieved results for change propagation analysis depends on the architecture characteristics and type of the change (See section 5).

In order to demonstrate how our hypothesis might work, we looked for an example of a software system where we could apply our approach. We wanted an example which typifies enterprise applications, but is smaller and hence more easily understandable and explainable. We chose a simple web application designed to promote a health club, and allow members to join and book activities online. In this paper we address this application as the example website, and it help us to explain our model and steps for analysing conceptual connections. In section 3 we take this website as a case study to show how our methodology might work.

2.1 Basic Concepts and Notation

For the remainder of the paper we use standard notation for binary relations. For \( R, Q \subseteq A \times A \), we denote by \( R.Q \) their composition, i.e., \( x.R.Qy \iff \exists z : x.Rz \land z.Qy \). \( R^{-1} \) denotes the inverse of \( R \) and \( ID \) the identity relation. Moreover we abbreviate \( x.R = \{y|xRy\} \).

We use graph theory in our work, denoting by \( G = (V, E, l) \) the graph \( G \) with vertices \( V \), edges \( E \subseteq V \times V \) and labeling \( l : E \rightarrow L \) for some label set \( L \). Weighted graphs are labelled graphs where the labeling function assigns numeric weights to the edges.

The edge set in a graph obviously define a binary relation. It is folklore in computer science that any finite set \( X \) of pairwise disjoint relations \( R : A \times A \) on some set \( A \) can be equivalently represented by a graph with vertices \( A \) and directed edges \( E = \bigcup_{R \in X} R \), the union of the given relations. This is achieved by naming relations and labeling the edges of the graph with corresponding relation names. More formally, let \( L \) be a finite set of relation labels and \( l_R \in L \) the name of \( R \) for any \( R \in X \). Then we define \( REL(A, X) \) as the labelled directed graph \( REL(A, X) = (V, E, l) \) with \( V = A, E = \bigcup_{R \in X} R \) such that

\[
(v, v') \in E \text{ and } l(v, v') = l_R \text{ iff } vRv' \text{ for some } R \in X.
\]

We also call \( REL(A, X) \) the relation graph of \( X \) over \( A \). Note that relation application (dot notation in \( x.R \) applying \( R \) to an object \( a \in A \)) and composition (dot notation \( R.Q \)) corresponds to path chasing in the relation graph.

Now the three basic concepts of our paper are modelled by binary relations on sets as follows.

1. **Domain variables** are simply modeled by a finite set \( V \), called variable symbols.

2. **Domain functionalities** are modeled by a finite set \( F \), so-called function symbols. The binary relations \( REL ; USE \subseteq F \times V \) represent elementary dependencies between functions and variables. For convenience, we define \( REF \subseteq USE \) and interpret \( REF \) as input variables (read set) and \( USE \) as the input-output variables (read and write set). Because we are only interested in domain functionalities (interacting with an external user or external software) we assume moreover that \( f.REF \neq \emptyset \) for all \( f \in F \), i.e., a domain function uses one or more variables.

3. **User Interface Components (UICs)** are modeled by a finite set \( C \) called the component symbols. The associated function symbols are represented by the relation \( USE \subseteq C \times F \). We require that \( c.REF \neq \emptyset \) for all \( c \in C \), i.e., a component has one or more functions.

**Example 1.** In the example website the UIC LoginPage has the following functions: Login_page.HAS = \{authenticate, accept, reject\}. Informally, authenticate reads the name and password combination and excludes pathological cases such as empty strings entered by the user. The system then reads the combination, determines whether to accept or reject and produces the status AuthenticationStatus message. This variable use is expressed formally as follows: authenticate. USE = authenticate.REF = accept.REF = reject. REF = \{UserName, Password\}, and accept. USE = reject. USE = accept. REF \cup \{AuthenticationStatus\}.

**Convention 1.** For the rest of the paper, and without loss of generality, we assume the system under analysis (SUA) is fixed, that is, \( V, F \) and \( C \) are fixed and so are their \( REF, USE \) and \( HAS \) relations. We also call the graph \( REL(V \cup F \cup C, \{REF, USE\} \setminus \{REF, HAS\}) \) the behavioural model for the given SUA.

2.2 Dependency Analysis

Domain functionalities and components are related when they share symbols. The relationship \( CNC \) derived from the behavioural model of the SUA captures this as follows.

**Definition 1.** We define the conceptual connection relation \( CNC \subseteq C \times C \) by

\[
CNC = HAS. USE. USE^{-1} HAS^{-1}
\]
We also call $REL(C, \{CNC\setminus ID\})$ the conceptual connection graph of the SUA.

Clearly, $CNC$ is reflexive. Since functions have a nonempty variable set, the following corollary follows by definition.

**Corollary 1.** Two UICs $c, c' \in C$ are conceptually connected iff they are adjacent in the conceptual connection graph.

**Example 2.** In the example website, two UICs ContactPage and LoginPage are conceptually connected. ContactPage.HAS.USE = \{UName, Age, Query, Email\} and LoginPage.HAS.USE = \{UName, Password\}. ContactPage.HAS.USE $\cap$ LoginPage.HAS.USE = \{UName\} $\neq$ \{\}. For a large SUA, the number of variables and functions may be large and so the number of dependencies may not only be large but also dependencies between large-scale components of similar size (number of function points) may vary significantly in the number of domain variables shared. It turns out that it is practically very useful to weight dependencies by their level of sharing. A threshold $t$ can then be used to select relevant dependencies by their weight $w > t$ only. The following definition introduces a suitable notion of relevance.

**Definition 2.** The Weighted Connection Graph of a SUA is the weighted graph $G = (C, CNC\setminus ID, w)$ where the weight $w : E \to [0..1]$ assigns probabilities to edges by

$$w(c, c') = \frac{|c.HAS.USE \cap c'.HAS.USE|}{|c.HAS.USE \cup c'.HAS.USE|}$$

**Example 3.** In the example website, the weight of the link between ContactPage and LoginPage is calculated as follows, where, as noted earlier, the intersection of ContactPage.HAS.USE and LoginPage.HAS.USE is just \{UserName\}, and the union is \{UserName, Password, Age, Query, Email\}. Hence the weight is:

$$w(\text{ContactPage}, \text{LoginPage}) = \frac{1}{5} = 0.2$$

Note that by definition, $(c, c')$ is an edge in this graph iff $cCNCc'$ and $c \neq c'$. In this paper we use this graph to visualize the conceptual connections between UICs. Both the CNC and the Weighted Connection Graph are symmetrical, i.e., $(c, c') \in E \iff (c', c) \in E$ and $w(c, c') = w(c', c)$.

3 Case study of applying the methodology on the example website

The aim of this case study is to demonstrate how domain users can derive a weighted connection graph, and how such a graph can help to estimate the existence of architecture dependencies in the source code.

In order to assess this we had three people (the first three authors) independently analyse the web system based only on domain information, primarily the user interface, i.e., the website architecture (figure 1) was not made available to them. Two people had the role of typical domain users with a superficial understanding of system functionalities, and one had the role of a domain expert. The domain expert studied the system specification document which describes system features and functionalities (excluding implementation data).

The domain expert and both domain users created a list of UICs (web pages), then listed domain variables related to UICs by HAS.USE relationships. The results suggest that the accuracy of the derived relationship model is highly dependent on knowledge about system functionality. Both domain users did not record pages which are not available from the main menu. Also, domain variables which are not directly visible on the webpage were only captured by the domain expert using the website functional specification.

Table 1 shows the number of UICs recognized as part of each analysis, the number of captured conceptual connections and the number of architecture dependencies matching the conceptual connections.
Figure 2. Weighted connection graph for the example website

Figure 2 illustrate the weighted connection graph created from a list of UICs and related domain variables found by the domain expert. Each node is labelled with the name of a UIC and with the number of related domain variables by the HAS.USE relationship. Each edge shows a CNC relationship between two UICs (c and c') and is labelled with the weight $w$.

In the next step we analysed the source code to find how UICs are connected through architecture dependencies. The website is developed using PHP including webpages and libraries. Both domain users and the domain expert recorded only webpages which read/write some domain variables, this excludes Home, Links and AboutUs webpages from the UICs list. Reviewing the source code showed that the excluded pages only contain HTML contents with no dependency or connections to any other file. We concluded changing these static pages are highly unlikely to affect other webpages in the website.

The complete transitive connections in this website connects all the UICs. In order to compare the weighted graph with the website architecture, we searched for only architectural dependencies between UICs result of HAS.USE.USE$^{-1}$.HAS$^{-1}$ (common domain variables) and HAS.HAS$^{-1}$ (common domain functionalities) relationships. Therefore we omitted arbitrary transitive connections between UICs such as Contact and Delete Account. The result set is presented in table 2 as eleven pairs of UICs which use a common library or a shared data file, and no two UICs were found with a direct reference between their source code.

In table 2 we present a comparison of results obtained by analysis of the source code (figure 1) with respect to results obtained via our methodology by the domain expert (figure 2). The third column indicates the architectural dependencies, if any, between the two UICs and the last column shows the weight of the edge between them in the weighted connection graph.

The connection graph shows eleven conceptual connections between UICs, but only nine of them match the architecture connections, i.e., there were two false positives and two false negatives. In the false positive cases, web pages share domain variables which have the same name on the screen but are implemented as different data fields. This is a discrepancy between the system specification and the actual implementation in the source code. In both false negative cases, the web site pages are connected by referencing a utility library, although there is no common functionality or domain variable visible to the domain user. Both false positive results are located on the light edges ($w < 16\%$) which suggests a rough correlation between the weight of the graph edge and whether there is an architecture connection in the source code. In addition there are three edges with ($w > 80\%$) between UICs which are related in the source code as they read and write all fields of the same data source (User.XML).

This case study has shown how architectural dependencies in a typical information system can be derived from domain-based system analysis, and how the weighted connection graph can be used to accomplish such a task.
4 Case study of applying the methodology on an enterprise system

We then applied our methodology to a subsystem of a large scale enterprise application called BEIMS. The selected subsystem is a web-based application called Building Condition Assessment (BCA). In this case study, we evaluated efforts and difficulties for creating a weighted connection graph from BCA using only non-technical information visible to domain users. Also we evaluated to what extent the weighted connection graph can help to find architecture dependencies in the system.

4.1 BCA Weighted Connection Graph

In the first step, the first author who had expertise in the functionality of BCA analysed the behaviour of the system using functional specification. The following explains the steps taken to create the weighted connection graph:

1. Created a list of UICs. Each webpage with a unique address in the site map was considered as a UIC. This resulted in 18 UICs.

2. Extracted domain variables related to each UIC by reviewing domain functionalities and input/output domain variables for each functionality. The result is a set of triples (UIC, Domain Functionality, Domain Variable). This resulted in 231 triples.

3. Create a weighted connection graph using the triples based on definition 2.

The time required for the analysis was slightly more than two hours including reviewing the functional specification. We found that reviewing functionalities of UICs and finding related domain variables are reasonably easy tasks for a domain user who has basic understanding of the system functionalities; however, there were cases where a more detailed understanding of domain functionalities was necessary to clarify ambiguities in the system behaviours. The following are the primary challenges which the domain expert faced during domain-level analysis of the system:

I. There are dynamic behaviours in the system affected by the user profile, or system settings. Such behaviours make some domain variables hidden for specific users.

II. Some domain variables have very generic names on the screens such as Status and Description. These generic names for domain variables make them difficult to be uniquely identified in the weighted connection graph. We changed the name of these domain variables to a unique identifier by adding associative prefix such as WorkOrderStatus instead of Status.

III. Some domain variables have different names in different components. This is a problem result of inconsistency in naming conventions, and we referred to the functional specification to clarified such ambiguities.

![Weighted connection graph with threshold \( w \geq 30\% \)](image_url)

Figure 3. BCA Weighted Connection Graph

The resulting weighted graph was very dense (unreadable), we changed the threshold to \( w \geq 0.3 \) where the graph is more readable (see figure 3). We found the density of the graph is a potential problem for complex systems. Figure 4 is an alternative presentation for the weighted connection graph. It is a cross table which shows all connections in the weighted graph.

---

2 BEIMS is a well-known Facility Management software package, developed by Mercury Computer System, which is widely used in Australia and New Zealand by universities, hospitals and hotels in order to manage assets and building facilities. Its initial version was developed 20 years ago, and it has more than 5 generations in its life cycle. Its core system, add-ons and custom developed subsystems are more than 100 applications.

3 The first author is employed by Mercury Computer Systems.

4 The functional specification used in this case study describes the system behaviour with no implementation or inner working information.
4.2 Compare the graph with architectural dependencies

In the second step, we analyzed the actual software architecture to determine a “ground truth” for comparison against the weighted connection graphs.

BCA is developed based on a three-tier architecture (presentation layer, business layer, and data layer) using .Net technologies including ASP.Net with AJAX extension for presentation layer, Component-based Scalable Logical Architecture (CSLA) for business layer, and Microsoft Enterprise Library (Data Application Block) for the data layer. BCA has an object-oriented design, and consists of classes in different namespaces. In the presentation layer, each UIC is composed of a webpage (ASPX file), and a class which contains the source code of the page.

We used .Net Code Mode to extract the list of all classes and references between them. The result shows that the BCA architecture consists of 154 classes, including 18 classes for UICs. Comparison between these results and the domain-analysis results shows that UIC’s classes in the architecture match the UICs found by the domain expert.

When a member of a class calls another class member then there is a reference between them. If there is at least one reference between members of two classes then they are architecturally dependent. In BCA, we found 1270 individual references between different classes yielding 403 architecture dependencies between pairs of classes.

In the analysis of the example website (section 3), we took a simplistic approach and only searched for direct relationships between UICs, or when two UICs reference another PHP library or XML file. However, this approach is not sufficient for analyzing transitive dependencies in BCA because in a multitier architecture system, functionalities are fragmented to more granular reusable functions encapsulated in different layers. Therefore, we had to extend the discovery method to find related components to domain variables in deeper layers of the software. For example, data tables are sitting in the third layer of the system which required transitive dependencies through the business layer to access them from UICs.

The other challenge which we faced was a lot of generic classes and source codes not related to domain functionalities. BCA, like many other enterprise applications, was built on top of existing reusable libraries. These libraries in BCA create a complex set of dependencies. However, the complexity of this architecture is not visible to domain users. For example, we found 127 references between data layer utility codes from/to other utility and non-utility codes in BCA. These reusable libraries provide functionalities such as reading and writing data to different database platforms like SQL Server and Oracle, and taking care of concurrent transactions and optimizing SQL statements. They are necessary to deliver domain functionalities, but their role is invisible to domain users as they do not change or create values for domain variables.

We found because of these utility libraries, that any two UICs are connected in the source code by at least one transitive connection; however, in the majority of cases changing a domain functionality does not affect these utility libraries. Therefore, we compared the weighted graph with components where their functionalities have a visible output for the domain user. Examples of excluded classes are: classes related to database configuration and exception handling.

We found 45 classes related to domain functionalities, covering majority of classes in the presentation layer and non-utility classes in the business and data layers. These classes have 169 (far less than 1270) individual source code references between them, yielding 121 pairs of dependent classes. In order to compare the result with the weighted graph, we searched for transitive architecture dependencies between classes related to UICs (18 classes out of 45). For example, two UICs RoomAssess and FloorAssess are both architecturally dependent on AssessmentEdit class in the business layer, so there is a transitive dependency between RoomAssess and FloorAssess.

The result shows 107 pairs of connected UICs, each pair connected by at least one transitive architecture dependency. Comparing these pairs with the weighted connection graph shows only 101 pairs match edges in the graph (graph has 116 edges), i.e., 15 false positive results where the edges did not match any transitive dependency in the source code. Also, we found 6 false negative results where there is a transitive dependency between two UICs in the source code but the graph does not include an edge between them. The reason for these false negatives is a specific functionality in BCA which is not visible to domain users. It is a function which protects the data integration, and as part of each edit/delete checks related data tables to avoid deleting a parent record while there is a child record in another table. All false negative results are related to this functionality. Figure 4 shows these false results in the weighted graph.

Figure 5 illustrates the distribution of graph edges based on their weight. The majority of edges in the graph have less than 0.33 weight and few edges have weight higher than 0.9. Also all the false positive results are in low strength edges, which suggests for stronger weighted edges we can be more certain in concluding that there is an architecture dependency between the two UICs. Comparing the architecture dependencies with these edges shows all the UIC pairs connected by an strong weighted edge (\(w > 0.8\)) are reading and writing to the same data table. To avoid false positives in the graph, we changed the threshold to \(w > 0.3\) (see figure 3). Also it turns out to be practically useful to change the threshold where the UICs are grouped by strong edges in the weighted graph (a spring graph).
### Figure 4. Cross table presentation of BCA Weighted Connection Graph

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildingAssess</td>
<td>63%</td>
<td>13%</td>
<td>13%</td>
<td>13%</td>
<td>25%</td>
<td>52%</td>
<td>13%</td>
<td>5%</td>
<td>5%</td>
<td>68%</td>
<td>94%</td>
<td>23%</td>
<td>25%</td>
<td>5%</td>
<td>5%</td>
<td>25%</td>
<td>17%</td>
<td>4%</td>
</tr>
<tr>
<td>BuildingGroup</td>
<td>25%</td>
<td>27%</td>
<td>20%</td>
<td>30%</td>
<td>10%</td>
<td>50%</td>
<td>5%</td>
<td>5%</td>
<td>9%</td>
<td>9%</td>
<td>13%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>BuildingIndividual</td>
<td>52%</td>
<td>40%</td>
<td>41%</td>
<td>14%</td>
<td>13%</td>
<td>13%</td>
<td>13%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>9%</td>
<td>9%</td>
<td>9%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>BuildingSummary</td>
<td>13%</td>
<td>27%</td>
<td>14%</td>
<td>27%</td>
<td>13%</td>
<td>13%</td>
<td>5%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>BuildingEvaluation</td>
<td>9%</td>
<td>9%</td>
<td>20%</td>
<td>20%</td>
<td>9%</td>
<td>9%</td>
<td>20%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>BuildingEvaluationList</td>
<td>9%</td>
<td>9%</td>
<td>20%</td>
<td>20%</td>
<td>9%</td>
<td>9%</td>
<td>20%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>ElementList</td>
<td>5%</td>
<td>5%</td>
<td>25%</td>
<td>17%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>FloorAssess</td>
<td>68%</td>
<td>94%</td>
<td>23%</td>
<td>44%</td>
<td>13%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>FundSourceEdit</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>FundSourceList</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>PDFAssessmentListReport</td>
<td>31%</td>
<td>31%</td>
<td>26%</td>
<td>17%</td>
<td>17%</td>
<td>17%</td>
<td>17%</td>
<td>9%</td>
<td>9%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
</tr>
<tr>
<td>PDFAssessmentSurveySummaryReport</td>
<td>25%</td>
<td>25%</td>
<td>17%</td>
<td>26%</td>
<td>17%</td>
<td>17%</td>
<td>17%</td>
<td>9%</td>
<td>9%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
<td>7%</td>
</tr>
<tr>
<td>RoomAssess</td>
<td>72%</td>
<td>90%</td>
<td>22%</td>
<td>42%</td>
<td>12%</td>
<td>8%</td>
<td>8%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>SurveyEdit</td>
<td>4%</td>
<td>8%</td>
<td>7%</td>
<td>4%</td>
<td>7%</td>
<td>9%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>SurveyList</td>
<td>4%</td>
<td>9%</td>
<td>8%</td>
<td>4%</td>
<td>8%</td>
<td>9%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>8%</td>
<td>8%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>TargetScheduleEdit</td>
<td>4%</td>
<td>4%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>TargetSchedulesList</td>
<td>4%</td>
<td>4%</td>
<td>9%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
<td>4%</td>
</tr>
</tbody>
</table>

#% : Edges with weight w>0  
%: False positive edges  
* : False negative

### Figure 5. Distribution of edges in BCA Weighted Connection Graph
In summary this case study provides an example of how this methodology can be applied to a real life enterprise application and how a Weighted Connection Graph can be derived from only domain-level information. Also source code analysis showed that the resulting graph could be used to find some dependencies in the source code related to domain functionalities. These dependencies can reveal the architecture elements which can be affected by introducing a change to a domain functionality in one or more UICs.

5 Applicability of this approach

Our approach for estimating the scope of change propagation is highly dependent on visibility of the change for domain users. Swanson [12] classified software evolution as being perfective (“perfecting” the software by adding or changing in response to user needs), adaptive (adapting to a changed environment), or corrective (correcting errors). Subsequently a further category, preventive (changing to enhance maintainability or prevent errors), was introduced [8]. Perfective and corrective changes are typically requested by domain users and related to domain functionalities, whereas adaptive and preventive changes are usually introduced by software architects and they can be invisible to domain users. If domain users can relate a change to a domain functionality of a UIC then they will be able to use the weighted connection graph to find other related UICs and domain functionalities.

This approach is applicable when a change introduced to a functionality which is visible to domain users. The intention is to ask a domain user to find UICs which may be affected by this change. Therefore software changes resulting from technical reforms, such as improving the performance of algorithms or improving cryptography, are excluded from the scope of this approach.

The system behaviour characteristics are the other factors for applicability of this approach. From case studies we learned that the level of complexity of system functionalities has an impact on the usability of this approach for domain users. In addition, the visibility of the data scheme in the system presentation layer is essential for creating the weighted connection graph from domain-level information.

The other characteristic of the system to be considered is the distribution of domain variables related to domain functionalities and UICs. For example in an extreme case if all domain functionalities of a system use the same domain variables (perform different functions on the same data) then the resulting weighted connection graph will be a complete graph with all nodes connected to each other, and a weight of 1 on all edges. In this case if there is only a subset of UICs which will be affected by a change then they can not be derived from the resulting weighted graph. Also the efficiency of this approach is highly related to number of UICs and distribution of domain functionalities among them. Therefore this approach is not usable for analysing dependencies in architecture of systems with limited UICs such as operating systems.

We acknowledge that as we are in the early stage of this research, we have not examined all related factors which can impact the usability of this approach. More investigation in this area is needed and creating a metric for evaluating the applicability of this approach is a subject for further work in this research.

6 Conclusions and Further Work

Domain-base system analysis is a novel approach for analyzing conceptual connections and similarities in a system. Such approach can be used for analyzing the scope of change propagation using only information visible to domain users. Simplicity and independence from implementations are two characteristics which make it practical for typical enterprise applications, heterogeneous systems, and legacy applications with missing source codes. In this paper we introduced the first phase of this research: creating a weighted connection graph using information visible to domain users. Such a graph expresses conceptual connections between user interface components (UIC).

We provided the results of two case studies which show how to apply this methodology to web-based systems. The case studies describe potential challenges for using this approach. Also they show how the resulting weighted graph can provide information about architecture dependencies. The results from the two case studies provided encouraging preliminary evidence that our approach gives reasonably accurate information: a significant majority of architectural dependencies were identified with a reasonably low number of false negatives and false positives. The accuracy of such a prediction is subject to the characteristics of the system behavioural model and the type of introduced change. In addition, we showed how changing the threshold value in the weighted graph can remove false positives and visually group the strongly connected UICs; however, the feasibility of selecting the threshold value systematically is yet to be investigated.

As part of our further work, we will provide a methodology for estimating the scope of change propagation using the conceptual connections between system components. The main gain of this methodology is in giving domain experts the ability to analyse the potential change propagations for an enhancement or alteration without referring the change request to software architects. This approach is a trade off between accuracy and practicality in favour of practicality.

In this paper we briefly discussed some of the system characteristics (architectural or behavioural) which affect
the usability and applicability of this approach. We will study these efficiency factors, and provide a metric for evaluating the applicability of this approach for different software architectures.

This approach relies on analyzing UICs and discovering domain functionalities related to each component and domain variables related to each functionality. The required effort for performing such an analysis will have a linear relationship to the number of UICs and the complexity of domain functionalities. Supporting tools can help domain users to record and manage the data easier, and improve the speed of data collection. We envisage that this approach can be automated by using existing data such as system functional test cases, or by extracting the domain variables from HTML pages of a running web-based system. In the third phase of this research we will review the possibilities of such automated solutions, considering simplicity and practicality criteria.

Overall, our conclusion is positive: domain-based derivation of weighted connection graphs is fairly easy, and the results provide good accuracy. However, we can not draw too strong a conclusion from an individual experiment as it is related to the system behaviour model, and more experiments are needed to confirm such a conclusion for different application types.

Acknowledgement

The authors would like to thank Mercury Computer Systems for their support in allowing the use of BEIMS as a case study.

References


In Ninth International Workshop on Program Compre-