Master's Thesis

Network-theoretic Analysis of Socio-technical Relations in Software Projects

Gabriele Zorn-Pauli, Bakk.rer.soc.oec.
g.zorn-pauli@student.tugraz.at
Matriculation Number: 0330146

Institute for Knowledge Management
Graz University of Technology
Head: Univ.-Prof. Dr.rer.nat. Klaus Tochtermann

Advisor: Dipl.-Ing. Dr. techn. Markus Strohmaier

Graz, May 2010

For my significant other

Abstract

The software development process is a complex and dynamic task performed by humans. The reduction of source code and maintenance dependencies among software artifacts, and the improvement of coordination between developers, teams, and stakeholders, are essential issues regarding software engineering. Conway's Law was one of the first formulated assumption about the relations between the technical structure of a product and the organizational structure producing them, expecting that the technical architecture reflects the organizational. Since the emergence of web-supported distributed software projects, access to a large range of software repositories and corresponding traces of collaboration has now become available. The interest of software engineers in socio-technical congruence, motivated by Conway's Law, consists of developing methods to calculate congruence of coordination requirements, arisen through technical dependencies, and the corresponding actual coordination activities, because an association with development productivity and effectivity was acknowledged. This thesis deals with investigations about socio-technical congruence of two real world software projects, Eclispe SDK and IBM Jazz Rational Team Concert, for investigating two issues. On the one hand, the extent to which congruence measures are useful for solving practical problems is studied. On the other hand, theoretical implications of Conway's law are explored. In an empirical study, software quality and success factors are identified which are supposed to affect socio-technical congruence of software projects, such as number of bugs per software component.

Keywords: Socio-technical Congruence, Conway's Law, Network-theoretic Analysis, Repository Mining, Coordination Requirements

Zusammenfassung

Der Software Entwicklungsprozess ist eine sehr komplexe und dynamische, von Menschen ausgeführte, Tätigkeit. Die Reduzierung von Source Code und Maintenance Abhängigkeiten zwischen Softwarekomponenten und die Verbesserung der Koordination zwischen Entwicklern, Entwicklerteams und anderen Interessengruppen, gehört zu den grundlegenden Fragestellungen in der Softwareentwicklung. Conway's Gesetz war eine der ersten formulierten Annahmen über den Zusammenhang von technischen Strukturen von Produkten und der organisationalen Struktur, welche die Produkte fertigt. Seit dem Aufkommen von web-basierten verteilten Software Projekten ist eine große Anzahl an Software Repositories und die zugehörigen aufgezeichneten Kollaborationsspuren von Entwicklern zugänglich geworden. Das Interesse von Software Ingenieuren an sozio-technischer Übereinstimmung, motiviert durch Conway's Gesetz, besteht aus der Entwicklung von Methoden zur Berechnung von Kongruenz zwischen Koordinationsanforderungen, welche durch technischen Abhängigkeiten entstehen, und dazugehörigen tatsächlichen Koordinationsaktivitäten, weil ein Zusammenhang mit Entwicklungsproduktivität und -effektivität eingeräumt wird. Diese Arbeit beschäftigt sich mit Untersuchungen von sozio-technischen Übereinstimmungen von zwei realen Software Projekten, Eclipse SDK und IBM Jazz Rational Team Concert, um zwei Fragestellungen zu untersuchen. Zum einen den Umfang in dem Kongruenz Metriken nützlich sind, um praktische Probleme zu lösen und zum anderen,welche theoretischen Implikationen, beruhend auf Conway's Gesetz, erforscht werden können. In einer empirischen Studie werden verschiedene Qualitäts- und Erfolgsfaktoren identifiziert, die vermeintlich die sozio-technische Kongruenz von Software beeinflussen, wie beispielsweise die Anzahl an Softwarefehlern pro Software Komponente.

Schlagworte: Sozio-technische Kongruenz, Conway's Gesetz, Netzwerk-theoretische Analyse, Datengewinnung, Koordinationsanforderungen

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
(date)	(signature)

Acknowledgements

I would like to express my gratitude to my advisor Dr.Markus Strohmaier, who inspired me to social network analysis and improved my understanding for scientific writing, supporting and guiding me during the course of this thesis.

Thanks to my parents, who harbored me during the time of writing my thesis and to my family and friends who always encouraged me, especially with my constant commuting between Austria and Germany.

Ela, thank you for your trust, understanding and being there for me.

Gabriele Zorn-Pauli, May 2010

Contents

1 Introduction 3

1.1 The Complexity of Socio-technical Structures . . . 4
1.2 Research Questions . . . 5
1.3 Contribution . . . 5
1.4 Structure of this Document . . . 5

2 Related Work 7

2.1 Social Network Analysis . . . 7
2.1.1 Concepts and Definitions . . . 7
2.1.2 Network Types . . . 8
2.1.3 Topological Network Properties . . . 11
2.2 Socio-technical Congruence . . . 16
2.2.1 Understanding Congruence . . . 16
2.2.2 Measuring Congruence . . . 18
2.2.3 Achieving Congruence . . . 19
2.3 Software Development Quality and Success Metrics . . . 19
2.4 Mining Software Repositories . . . 21
2.4.1 Mining Source Code . . . 22
2.4.2 Mining Issue-tracking Systems . . . 26
2.4.3 Mining Cross-Media Resources . . . 29

3 Data Sets and Network Construction 32

3.1 Eclipse SDK Data Set . . . 32
3.1.1 Social-inferred Eclipse Dependency Network . . . 33
3.1.2 Code-inferred Eclipse Dependency Network . . . 36
3.2 IBM Jazz Data Set . . . 36
3.2.1 Social-inferred Jazz RTC Dependency Network . . . 37
3.2.2 Code-inferred Jazz RTC Dependency Network . . . 38
3.3 Social Network Analysis of the Generated Networks . . . 39

4 Selected Congruence Measures 42

4.1 Arc Mirroring . . . 42
4.2 Weighted Coordination Requirements . . . 44
4.3 Edge Weight Ranking . . . 47
4.4 Similarities, Distinctions and Characteristics . . . 48

5 Results and Discussion **49**
5.1 Impact on Congruence by Reducing Data Volume 50
5.2 Evolution of Congruence over Time 51
5.3 Segmentation of Static and Dynamic Code Dependencies 54
5.4 Correlation of Congruence Algorithm Measures 55
5.5 Identification of Quality- and Success Factors Effecting Congruence . 56

6 Conclusion **60**
6.1 Contribution 60
6.2 Outlook 61
6.2.1 Future Work 61
6.2.2 Improvements 61

A Implementation **68**
A.1 Matlab/Python Framework 68
A.2 Additional Tools 69

B Resources **71**
B.1 Excerpt of the Bugzilla Data Set 71
B.2 Eclipse SDK Component List 72
B.3 Example plugin.xml Metadata File IBM Jazz RTC Project 72
B.4 Example MANIFEST.MF Metadata File IBM Jazz RTC Project . . 73
B.5 IBM Jazz RTC Project Matched Software Component Tag List . . . 73
B.6 Screenshot IBM Jazz Work Item GUI 74

Chapter 1

Introduction

Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

Melvin E. Conway

The software development process is a dynamic and complex activity performed by humans. The reduction of code and maintenance dependencies among software artifacts and the improvement of coordination between developers, teams, and stakeholders are essential issues regarding software engineering. [Conway, 1968] was one of the first formulating an assumption about the relation between the technical structure of a product and the organizational structure producing them, expecting that the technical architecture reflects the organizational. Conway's Law, illustrated in figure 1.1, implies two aspects. On the one hand, that there may exist a *similarity* between the technical and organizational or socially-inferred structures of products, measurable through congruence. On the other hand, that there may exist a *causal* relation, where the socially-inferred structure causes the technical structure of a product.

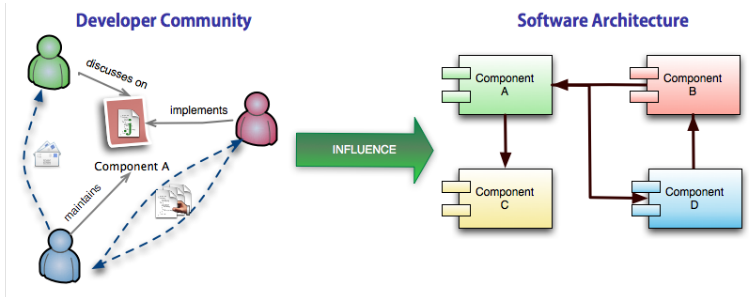


Figure 1.1: Conway's Law

This thesis deals with investigations about socio-technical congruence of real world software projects, examining whether the technical or code-inferred software architecture correlate with the socially-inferred software architecture using socio-technical congruence measures. The interest of software engineers in socio-technical congruence, motivated by Conway's Law [Conway, 1968], consists of developing methods to calculate congruence of coordination requirements, arisen through technical dependencies, and the corresponding actual coordination activities. Hence, socio-technical congruence underlines the importance of detecting relationships among social and technical dependencies, because an association with development productivity and effectivity was acknowledged [Cataldo et al., 2008].

1.1 The Complexity of Socio-technical Structures

An enormous problem in software development is the growing complexity of software systems. Therefore, an established approach in software development is to divide-and-conquer problems into smaller units, that means decomposing software into several subsystems or components where developers could work independently [Fonseca et al., 2006]. This leads on to the problem of reassembling the different software components as well as the coordination effort between developers working on two components which depend on each other. [Grinter, 2003] refers to this as a recomposition process, *'the work necessary to ensure that a software product can be assembled from its component pieces'*, where dependencies between components play an important role. In this case, a very important question is the definition of dependencies and for example, [Wermelinger and Yu, 2008] define software or code-inferred dependencies of software components in the following way:

Software component X depends on component Y if the compilation of X requires Y

On the other side, socially-inferred dependencies could be identified, if for example two developers work on the same component, traced through occurring in the same bug report or feature request. Hence, they got socially linked according to [Crowston and Howison, 2005]. Therefore, code-inferred dependency networks comprise dependencies between software artifacts acquired through code dependency analysis and socially-inferred dependency networks reflect dependencies among software components, gained through organizational coordination activities.

Furthermore, a software development process requires many tasks of coordination as well as communication between people who are engaged within the development process. Concerning this process there are two dimensions which have to be aligned. The first dimensions focuses on the technical perspective, because especially software development contains technical dependencies between software components based on code dependencies as well as for example maintenance dependencies. Otherwise, the second dimension relates to the organizational structure and the behavior of developers or people who are associated with the development process.

For being successful in software development it would be necessary to focus on both dimensions and align the technical and the social aspects [Cataldo et al., 2008]. [Valetto et al., 2007] state that the knowledge about socio-technical congruence of software projects and the understanding of how involved people, not only developers, communicate and interact with each other is important to improve the quality and probability of success. [Herbsleb and Mockus, 2003] argue that a bad alignment would result in a longer development time and also cause a higher amount of bugs and therefore higher costs.

1.2 Research Questions

This thesis concentrates on studying the relationships of socially- and technically inferred dependency networks acquired from software projects, using new approaches of socio-technical congruence and methods of social network analysis answering the following questions:

- To what extent is Conway's Law amenable for qualitative analysis?
- Are the measures of socio-technical congruence algorithms useful and practically applicable?
- Does congruence evolve over time? Growth or stable equilibrium?
- Is congruence correlated with some quality and success criteria, for example number of bugs?

1.3 Contribution

This thesis evaluates different approaches for measuring congruence of dependency networks by assessing three socio-technical congruence algorithms applied to two different real world data sets extracted from web-mediated software repositories.

The resulting insights about relationships between socially-inferred networks and software code-inferred networks through network-theoretic analysis could have implications on the design of an efficient team organization in software development projects. Therefore, the results and conclusions could support solving problems in coordination of software development projects by providing a better understanding of network-theoretic properties and characteristics of social and technical networks.

Additionally, the research on socio-technical congruence is a rather young discipline, and proof of concept and comparison of different approaches and metrics through an empirical study have not taken place on real world data sets yet.

1.4 Structure of this Document

The introduction chapter is followed by a synopsis of related works in chapter 2, which includes foundations of social network analysis, socio-technical congruence, software quality measures and different practices of mining of software repositories.

Figure 1.2 illustrates the thesis process chart of necessary steps conducting the empirical study on socio-technical congruence measures.

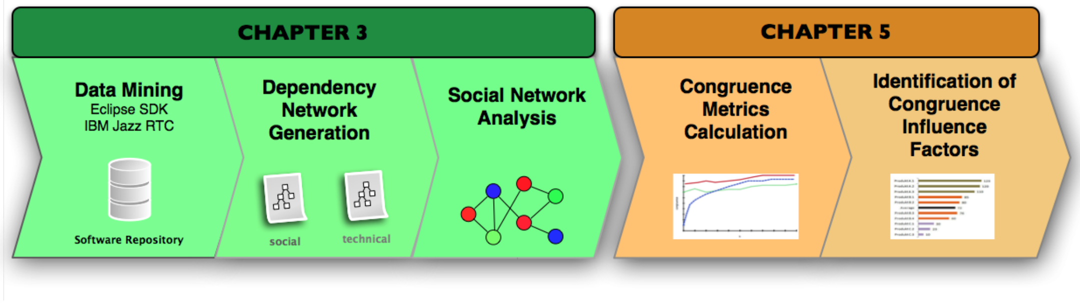


Figure 1.2: Thesis process chart

Chapter 3, contains descriptive properties of the two real world data sets, mined from software repositories and describes the generation of the dependency networks and results of a social network analysis. The next chapter 4 characterizes the three selected socio-technical congruence measures and the corresponding algorithms to consider similarities and differences. The following chapter 5 illustrates the results of the empirical study used to answer the questions in chapter 1.2 evaluating the results of the social network analysis, congruence measure calculations, and the identification of congruence influence factors. Chapter 6 discusses the results and insights and gives an outlook on future work and improvements.

The appendix A provides details on the implemented MATLAB/Python framework and the additionally used tools. Further, appendix B illustrates some resource examples including excerpts of the data and file formats for mining the software repositories and the dependency networks construction.

Chapter 2

Related Work

This chapter provides an overview of the most related literature which discusses the issues and approaches of socio-technical architectures and according research areas such as data mining or social network analysis. The first section provides an introduction and foundations of social network analysis metrics and methodologies. The second section deals with related works from the socio-technical congruence domain and the third section deals with different software quality measures, including both software code quality and software community measures. In the last section, different practices on mining software repositories using source code, bug reports or email archives for producing social networks reflecting collaboration or coordination structures of software developers are discussed.

2.1 Social Network Analysis

Various studies in the past have given an advice on the pervasion of social network analysis concerning software development processes and the regarding alignment of socio-technical aspects. The presence of network structures in technical and also in business domains make network research useful. Since Web 2.0 and the rise of web-mediated social software tools and services, the importance of social network analysis has increased. This section provides an introduction to key concepts and foundations of social network analysis, network theory and measurement of network topologies.

2.1.1 Concepts and Definitions

One of the key concepts in social network analysis are the social entities and their links among each other, denoted as **actors**, where this description is used as a metaphor and does not limit actors to entities which are able to *act*. These actors could represent different characteristics of entities such as a real person, person groups up to technical units or modules. For constructing a social network, the concept of relating actors by *relation ties*, or so called edges, is used. Thereby, the decision of linking two actors depends on the specific context and ties could represent different relations such as friendship, affiliation or technical dependencies.

In literature, many approaches of network analysis concern the issue of linking node pairs, where this relation concept is denoted as *dyad.* This was discussed by [Wasserman and Faust, 1994] and represents the basic unit of network analysis.

Dyad A dyad is an unordered pair of actors and the arcs that exist between the two actors in the pair. The dyad consisting of actors i and j will be denoted by $D_{i,j} = (X_{i,j},\ X_{j,i})$, for $i{\neq}j$. There are exactly $\frac{n(n-1)}{2}$ dyads for n nodes represented in a adjacency matrix X which is also called *sociomatrix* in literature.

Sociomatrix Define $\mathbf{X_{i,j}}$ as the value of the tie from the ith actor to the jth actor on the single relation. Since there are n actors, the matrix is of size $n{\times}n$ with n rows and n columns. The value of the tie from n_{i} to n_{j} is placed into the (i, j)th element of X.

[Wasserman and Faust, 1994] also defined three dyadic isomorphism states of dyads concerning different arc directions illustrated in figure 2.1 and arc weights are represented by different values in the adjacency matrix.

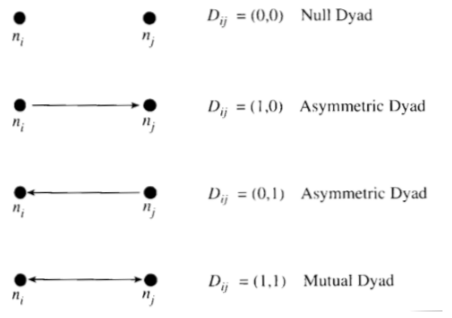

n_i
n_j
$D_{ij} = (0,0)$ Null Dyad
$D_{ij} = (1,0)$ Asymmetric Dyad
$D_{ij} = (0,1)$ Asymmetric Dyad
$D_{ij} = (1,1)$ Mutual Dyad

Figure 2.1: Dyadic isomorphism states [Wasserman and Faust, 1994]

The next greater unit which is considered in many network analysis approaches is the **triad** which represents a subgroup containing three actors and their relationships. An interesting issue concerning triads is the analysis of **transitively** relations discussed in [Wasserman and Faust, 1994] and [Granovetter, 1973] who define the model of the *forbidden triad*, shown in figure 2.2, specifying that within a triad with two existing edges, the absence of the third is forbidden.

Transitivity The *triad* involving actors i, j and k will become a fully connected triad if i is connected to j and j is connected to k then i is connected to k.

2.1.2 Network Types

On studying social networks, a division of different network types or network modes categorizing networks by the number of disjoint node types takes place. Therefore,

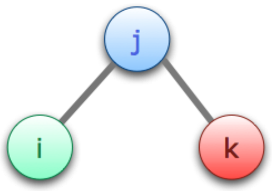


Figure 2.2: Forbidden Triad by [Granovetter, 1973]

this subsection describes the different network types of social networks.

One-mode Networks
A traditional network or *unipartite* graph $G = (V, E)$ includes only one particular set of node types H, for example actors, and consists of a number of nodes or vertexes V and a set of edges $E \subseteq V \times V$ capturing the relations between nodes. (see also definition of sociomatrix in section 2.1.1) [Latapy et al., 2008]

Two-mode Networks
Considering a two-mode network or *bipartite* graph, the network represents relations only between two different node types. [Wasserman and Faust, 1994] relate to them also as dyadic two-mode networks because of the references to dyad performance clarifying that the two vertices of the dyad are assigned to different node types. [Latapy et al., 2008] define two-mode networks as follows:

> Let $G = (\top, \bot, E)$ denote a two-mode network, where the $\bot$-projection of G is the graph $G_\bot = (\bot, \mathrm{E}_\bot)$ in which two nodes (of $\bot$) are linked together if they have at least one neighbor in common in G: $\mathrm{E}_\bot = (u, v)$, $\exists$ x $\in \top$: $(u, x) \in E$ and $(v, x) \in$ E. The $\top$-projection $\mathrm{G}_\top$ is denoted in the same way.

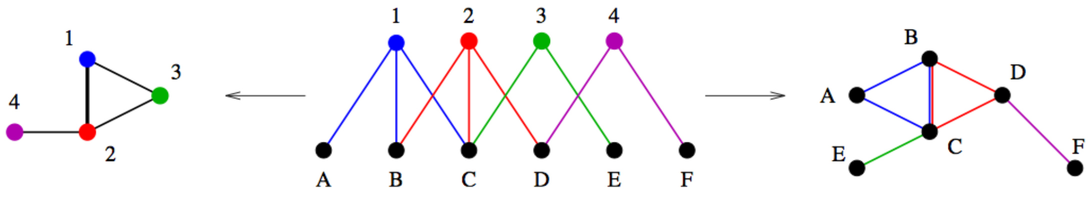


Figure 2.3: A two-mode network (middle) with according $\top$ (left) and $\bot$-projections (right) [Latapy et al., 2008]

Therefore, figure 2.3 illustrates the top and bottom projections of an exemplary two-mode network, depicting the disjoint relations between the two node types and the according one-mode networks derived from the projections using graph representation. Assuming that one node type describes a set of actors and the other a set of events in which an actor could be affiliated with, then in literature such networks are defined as **affiliation networks** [Wasserman and Faust, 1994]. Compared to one-mode networks, affiliation networks provide a focus on two perspectives of actors and events depicted in one network structure, because information about affiliations is concretely mapped in the affiliation network.

$$G = \begin{pmatrix} & A & B & C & D & E & F \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \qquad G_{\top} = \begin{pmatrix} & A & B & C & D & E & F \\ A & \mathit{1} & 1 & 1 & 0 & 0 & 0 \\ B & 1 & \mathit{2} & \mathbf{2} & 1 & 0 & 0 \\ C & 1 & \mathbf{2} & \mathit{3} & 1 & 1 & 0 \\ D & 0 & 1 & 1 & \mathit{2} & 0 & 1 \\ E & 0 & 0 & 1 & 0 & \mathit{1} & 0 \\ F & 0 & 0 & 0 & 1 & 0 & \mathit{1} \end{pmatrix}$$

Table 2.1: Adjacency matrix representation of the affiliation network shown in figure 2.3 and according $\top$-projection

[Datta et al., 2010] examine the dynamics of software development teams by constructing affiliation networks relating developers to roles, seniority and location. Further, most of the approaches discussed in the section 2.4.1, use affiliation networks for representing mined network data and relations between developers, bugs or software artifacts.

Hence, one approach of analyzing two-mode networks is to transform or reduce the network into two one-mode networks which is also denoted in literature as *projection* by using matrix manipulations [Latapy et al., 2008]. Therefore, it has to be observed that data is lost through transformation, because the one-mode network projection only provides information about one perspective of actors or events. To get the different projections, for example the $\perp$-projection involving the actor information and $\top$-projection the events, two matrix manipulations are necessary as denoted in formula 2.1 and 2.2. Referring to figure 2.3 and the transformed $\top$-projection, containing nodes from A to F, node pairs were linked only if they share the same neighbors in the affiliation network. An adjacency matrix representation would reveal the results of the transformation function shown in table 2.1. The different entries in the transformed adjacency matrix $G_{\top}$, denote the number of shared neighbors, for example node B and C share the same two neighbors (node 1 and 2) in the affiliation network and were linked with a edge weight of 2. Further, the diagonal provides information about the node degree in the affiliation network, for example node C possesses a degree of 3 due to three neighbors in the two-mode network.

$$G_{\perp} = G \times G^{\mathrm{T}} \tag{2.1}$$

$$G_{\top} = G^{\mathrm{T}} \times G \tag{2.2}$$

Most methodologies for analyzing social networks are based on and optimized for one-mode networks. [Latapy et al., 2008] provide an overview of basic notations and methods of transforming networks with multi-level node types into one-mode network projections.

k-mode Networks

An extension of two-mode networks are networks which include k different node types, where edges only exist between nodes of different types. Specially in terms of Web 2.0 and according to social web mechanisms like tagging, structures of so called *tripartite* or three-mode networks could be generated, including affiliations between user, resources such as videos, photos or bookmarks and between tags. Such three-mode structures could be indeed shown in graph representation but the network structure could only be stored in three two-mode sociomatrices comprising affiliations for example between users and objects, users and tags and relations between objects and tags.

[Neubauer and Obermayer, 2009] deal with k-mode networks for community detection, identifying groups which are denoted through a close connectedness in large and complex networks.

2.1.3 Topological Network Properties

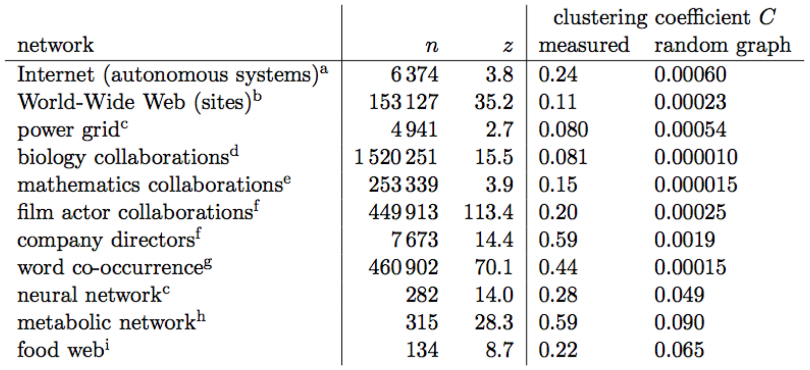
The field of social network analysis provides a number of methods and metrics for measuring social network properties. This section gives an overview of selected topological network properties used for investigations in this thesis.

Over the last decades, social network analysis was conducted using randomly generated network models like Erdös-Renyi networks [Erdös and Rényi, 1960] for simulating real-world networks. The construction of random networks according to the Erdös-Renyi model is accomplished by connecting every pair of N nodes with a probability of p. This produces a network with randomly distributed edges of around $\frac{pN(N-1)}{2}$ edges.

However, real-world networks like a software project developer community do not share the same attributes like a randomly generated network, because they are more complex and reflect some organizational structures or principles such as scale-free or small-world attributes in their network topology [Albert and Barabási, 2002], [Watts and Strogatz, 1998] and [Newman, 2002].

2.1.3.1 Small-world Property

Several studies of social networks are focused on the connectedness of networks. The concept of small-world networks is defined as *'the principle that we are all linked by short chains of acquaintances'* by [Kleinberg, 2000], which means that for example the developer of a large development team is related approximately to any other developer within the group and therefore the paths for reaching any other developer are relatively small. [Milgram, 1967] was the first pointing out the small-world phenomenon investigating network average path lengths of social networks. In an experimental study, [Travers and Milgram, 1969] examine *'the probability that any two people, selected arbitrarily from a large population, such as that of the United States, will know each other'* where they denote that two persons A and B may not know each other directly, but they share the same acquaintances. Further research on the small-network phenomenon such as that by [Watts and Strogatz, 1998] or [Kleinberg, 2000] was carried out and the small-world phenomenon is referred to as the *six degrees of separation* in literature. However, some criticism arises for example by [Kleinfeld, 2002] who argues that [Milgram, 1967] did not use an adequate amount of data and that the mathematical models of [Watts and Strogatz, 1998] will not replace empirical evidence.



			clustering coefficient C	
network	n	z	measured	random graph
Internet (autonomous systems)[a]	6 374	3.8	0.24	0.00060
World-Wide Web (sites)[b]	153 127	35.2	0.11	0.00023
power grid[c]	4 941	2.7	0.080	0.00054
biology collaborations[d]	1 520 251	15.5	0.081	0.000010
mathematics collaborations[e]	253 339	3.9	0.15	0.000015
film actor collaborations[f]	449 913	113.4	0.20	0.00025
company directors[f]	7 673	14.4	0.59	0.0019
word co-occurrence[g]	460 902	70.1	0.44	0.00015
neural network[c]	282	14.0	0.28	0.049
metabolic network[h]	315	28.3	0.59	0.090
food web[i]	134	8.7	0.22	0.065

Figure 2.4: Comparison of real world networks and random networks concerning small-world characteristics by [Newman, 2002]

Quantified indicators for a *small-world* property are a relatively low *diameter* and a high *clustering coefficient*, which is discussed by [Watts and Strogatz, 1998] and [Newman, 2002], who examine the dynamics and properties of small-world networks and show that Erdös-Renyi networks do not share these properties. Table 2.4 shows the results of the investigations of [Newman, 2002], comparing clustering coefficients of real-world networks with randomly generated networks, where n denotes the number of nodes and z the mean degree.

Distance and Diameter The distance $dist(x, y)$ in G of two nodes x, y is the length of the shortest $x - y$ path in G; if no such path exists, $dist(x, y) := \infty$. Concerning distances between every possible node pair in G, the longest shortest (geodesic) path of G is denoted as *diameter*. [Diestel, 2005]

$$diameter = max(dist(G)) \tag{2.3}$$

Clustering Coefficient Ratio value between actual and possible edges among a specific node v and its neighbors k, where i represents the number of existing edges between the neighbor nodes [Watts and Strogatz, 1998].

$$cc. = \frac{2i}{k(k-1)} \tag{2.4}$$

2.1.3.2 Scale-free Property

[Albert and Barabási, 2002] and [Barabasi and Bonabeau, 2003] examine the different degree distributions of real-world networks and random networks and observe that many real world networks present a power law degree distribution which significantly differs from Poisson distributions of random networks. This feature of *power law* distribution is one property denoting so called scale-free networks and defined as follows by [Barabasi and Bonabeau, 2003] in function 2.6. A further feature of scale-free networks is that network evolution is denoted by preferential attachment. [Albert and Barabási, 2002] describe this conjecture by an increasing probability of receiving new edges depending on the node degree.

Degree Distribution The degree $\mathrm{d_G(v)}$ of a vertex v is the number of $|E(v)|$ edges at v or the number of neighbors of v. A vertex of degree 0 is isolated [Diestel, 2005]. The degree distribution $P(k)$, formulated in formula 2.5, represents the fraction of vertices in the network of degree k for each value of k [Newman, 2002].

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k} \tag{2.5}$$

Power Law Distribution Denotes the probability $P(k)$ that a vertex in the network is interacting with k other vertices, decays as a power law. [Barabasi and Bonabeau, 2003]

$$P(k) \sim k^{\text{-}\gamma} \tag{2.6}$$

Preferential Attachment defined by [Albert and Barabási, 2002] When choosing the nodes to which the new node connects, assuming that the probability $\prod$ that a new node will be connected to node i depends on the degree k_{i} of node i, so that

$$\prod(k_{\mathrm{i}}) = \frac{k_{\mathrm{i}}}{\sum_{\mathrm{j}} k_{\mathrm{j}}} \tag{2.7}$$

The results and conclusions of [Albert and Barabási, 2002] were extended with further research on scale-free network properties such as the study of [Newman, 2002] or [Barabasi and Bonabeau, 2003], who examines among others degree distributions of real world networks, shown in figure 2.5, which appear with a power law degree distribution.

Related to the tenor of this thesis, for example [Hein et al., 2006] investigated in a case study the impact of network topologies influencing communication and diffusion

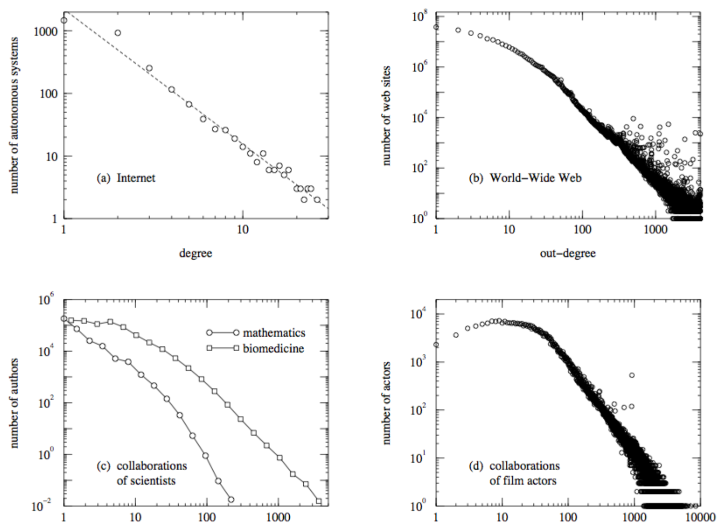


Figure 2.5: Power Law denoted degree distributions of real world networks [Newman, 2002]

processes in socio-technical structures. Concerning the nature of social networks appearing with small-world and scale-free networks, a corresponding performance of Open Source development communities was recognized and examined. For example [Xu et al., 2006] determined Open Source projects concerning network topology properties and observe that such projects feature characteristics of small-world and scale-free networks. Table 2.6 illustrates the results of [Xu et al., 2006] and implies that specially the nature of Open Source software projects, where new members join in depending on their individual interests and the evolution of the communities tends to highly connected networks, represents a successful policy of software development network structures which is verified and quantified in the amount of successful Open source projects.

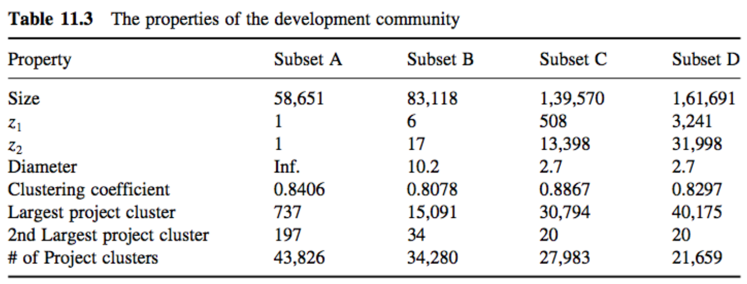


Table 11.3 The properties of the development community

Property	Subset A	Subset B	Subset C	Subset D
Size	58,651	83,118	1,39,570	1,61,691
z_1	1	6	508	3,241
z_2	1	17	13,398	31,998
Diameter	Inf.	10.2	2.7	2.7
Clustering coefficient	0.8406	0.8078	0.8867	0.8297
Largest project cluster	737	15,091	30,794	40,175
2nd Largest project cluster	197	34	20	20
# of Project clusters	43,826	34,280	27,983	21,659

Figure 2.6: Descriptive data of development community properties [Xu et al., 2006]

During the course of this thesis, further topology measures were used for a social network analysis of the generated dependency networks, described in chapter 3. The definitions of this metrics are as follows, formulated by [Wasserman and Faust, 1994].

Subnetwork A network G_S is a subnetwork of G if the set of nodes of G_S is a subset of the set of nodes of G, and the set of lines in G_S is a subset of the lines in the graph G. Denoting the nodes in G_{S} as N_{S} and the lines in G_{S} as L_{S}, where L_{S} is a subset of L, then G_{S} is a subnetwork of G if $N_{\mathrm{S}} \subseteq N$ and $L_{\mathrm{S}} \subseteq L$. [Wasserman and Faust, 1994]

Network Component A network is *connected* if there exist a path between every pair of nodes in the network. If there exist nodes that were not reachable, than the network is *disconnected*. The nodes in a disconnected network may be partitioned into two or more subnetworks where no paths between the nodes in different subnetworks exist. The connected subnetworks in the network are called *components*. [Wasserman and Faust, 1994]

Density Is defined for a network as the ratio of the number of undirected lines L or directed arcs A present to to maximum possible number lines or arcs that could arise. The density Δ fraction goes from 0, if no arcs or lines present, to a maximum of 1, if all arcs or lines present. [Wasserman and Faust, 1994]

$$\Delta_{\text{undirected}} = \frac{L}{n(n-1)/2} \tag{2.8}$$

$$\Delta_{\text{directed}} = \frac{A}{n(n-1)} \tag{2.9}$$

Closeness Centrality Measures how close for example an actor is to all the other actors in the set of actors. The idea is that an actor is central if it can quickly interact with all other. Let dist($\mathrm{n_i}$,$\mathrm{n_j}$) be the number of lines in the geodesic linking of actor i and j. The total distance that i is from all other actors is $\sum_{j=1}^{g} dist(n_{\mathrm{i}}, n_{\mathrm{j}})$, where the sum is taken over all $j{\neq}i$. [Wasserman and Faust, 1994]

$$c_{\mathrm{c}} = \left[\sum_{j=1}^{g} dist(n_{\mathrm{i}}, n_{\mathrm{j}}) \right]^{-1} \tag{2.10}$$

Betweenness Centrality Suppose that in order for actor i to contact actor j, actor k must be used as an intermediate station. Actor k in such a network has a certain *responsibility* to actors i and j. If we count all of the minimum paths which pass through actor k, then we have a measure of the *stress* which actor k must undergo during the activity of the network. Let g_{jk} be the number of geodesics linking the two actors and let $\mathrm{g_{jk}(n_i)}$ be the number of geodesics linking two actors that contain actor i. The actor betweenness index $c_{\mathrm{b}}(n_{\mathrm{i}})$ is the sum of estimated probabilities over all pairs of actors not including the ith actor and standardize on g, $c'_{\mathrm{b}}(n_{\mathrm{i}})$ takes on values between 0, if $\mathrm{n_i}$ falls on no geodesics and 1, if n_{i} is present in all geodesics of all node pairs not including n_{i} which is defined as $(g-1)(g-2)/2$. [Wasserman and Faust, 1994]

$$c_{\mathrm{b}}(n_{\mathrm{i}}) = \sum_{j<k} g_{\mathrm{jk}}(n_{\mathrm{i}})/g_{\mathrm{jk}} \tag{2.11}$$

$$c'_{\mathrm{b}}(n_{\mathrm{i}}) = c_{\mathrm{b}}(n_{\mathrm{i}})/[(g-1)(g-2)/2] \tag{2.12}$$

Clique A *clique*, illustrated in figure 2.7, in a network is a maximal complete subnetwork of three or more nodes. It consists of a subset of nodes, all of which are adjacent to each other, and there are no other nodes that are also adjacent to all of the members of the clique.

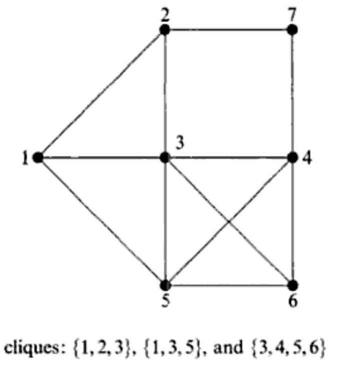


Figure 2.7: An example network and including cliques [Wasserman and Faust, 1994]

2.2 Socio-technical Congruence

Socio-technical congruence highlights the importance of identifying and tracking the dynamic relationships between social and technical dependencies.

[Cataldo et al., 2009]

Software engineering is inherently a socio-technical endeavor and the work of [Conway, 1968] was one of the earliest acknowledgements of that fact, arguing that any organization that creates a product, certainly produces a copy of its organizational structure. The interest in socio-technical congruence research, establish issues of socio-technical congruence in software development, aligning coordination needs and coordination requirements within the development process. The coordination of development teams, specially in large, complex and more often geographically distributed development teams is a well known challenge which claims development cost, time and software quality to improve.

2.2.1 Understanding Congruence

The work of [Sarma et al., 2008] provides an extensive introduction to the different challenges of understanding, measuring and achieving congruence because of the nature of comprising social and technical aspects in software projects.

> *Congruence, the state in which a software development organization harbors sufficient coordination capabilities to meet the coordination demands of the technical products under development [...]* [Sarma et al., 2008]

Therefore, congruence could be interpreted as a state of alignment between coordination requirements, gained through technical dependencies and actual coordination conducted through the organizational structures. This state is dynamical, only a snapshot, because specially in software development, code dependencies or the evolution of development teams change continuously [Sarma et al., 2008].

Issues of socio-technical congruence comprise the measurement of socio-technical congruence and policies for improving technical and organizational structures.

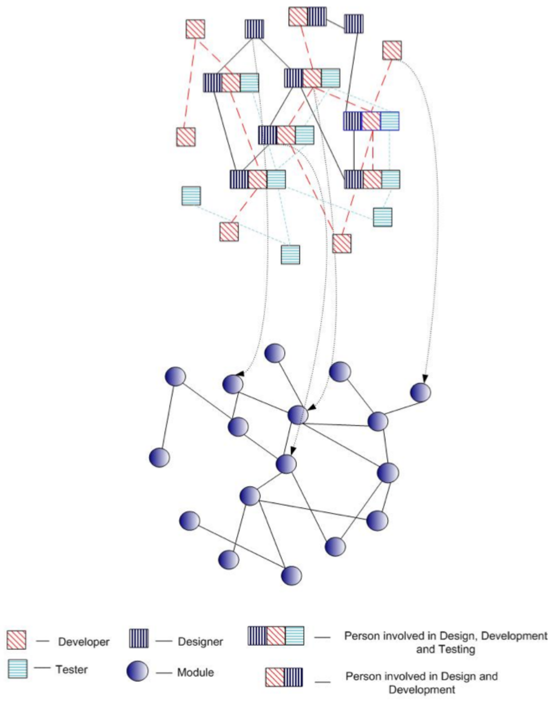

Developer
Designer
Person involved in Design, Development and Testing
Tester
Module
Person involved in Design and Development

Figure 2.8: Mapping between a social developer network (top) and the according network of software modules (bottom) [Amrit et al., 2004]

Further research extends the methods of representing the socio-technical structures by generating socially- and code-inferred networks, reflecting the technical dependencies of software artifacts on the one side and dependencies gained from organizational coordination activities on the other.

One of the first works capturing the idea of matching similar networks derived from different structures was done by [Bowman and Holt, 1998] who argue that their results validate Conway's Law through an empirical study using code *ownership* architectures as predictors for the *concrete* software architectures. They conclude that the organization structure of development teams mirrors the architecture of the software. [Amrit et al., 2004] map developer networks and software components networks, shown in figure 2.8, where they consider different developer roles and relate them in an affiliation network of developers and software modules. Therewith they show the possibility of relating different tasks concerning a specific software component to the appropriate developer from the developer network.

[Cataldo et al., 2006] use a concept of coordination requirements gained through task dependencies and task assignments for representing the socio-technical architectures, and they see a chance in identifying task dependencies' evolution and changes over time for a better possibility of designing information interchange with task dependencies. Therefore, they examine that congruence have an effect on task performance and observe congruence evolution over time.

Further works about the relationships of socio-technical structures were carried out in several studies. For example [Curtis et al., 1988] show that

> *higher connectivity among components required more communication among developers to maintain agreed upon interface definitions. Occasionally, the partitioning was based not only on the logical connectivity among components, but also on the social connectivity among the staff.*

[Herbsleb and Grinter, 1999] describes the different problems of coordination, specially in geographic distributed software development teams and the importance of informal ad hoc communications.

2.2.2 Measuring Congruence

A further issue concerning socio-technical congruence are the questions about useful sources containing dependency data. Related to software engineering, an established practice is the mining of software repositories and related communication and collaboration tools like mailing lists or newsgroups. The following list below, related to [Sarma et al., 2008], provides an overview of data which could be used for constructing social and technical dependencies.

- Team or community structures considering multiple team membership
- Direct and indirect communication between people
- Formal and informal communication paths
- Constrained prescribed processes or work practices
- Actual coordination actions traced by issue-tracking systems
- Tacit knowledge that individuals possess, implicitly reflected in expert hierarchies or code ownership
- Technical artifact dependencies captured for example in code structures or code dependencies

- Locality of people and artifacts
- Task assignment over time considering priorities or task dependencies
- Changes in work procedures, patterns or work practices, which indicates coordination problems

Different approaches and practices of data extraction or mining were discussed in chapter 2.4.1. The next issue concerning socio-technical congruence are the approaches measuring the achievement of congruence.

2.2.3 Achieving Congruence

Research on socio-technical congruence provides several approaches and algorithms for calculating a global congruence measure, denoting the aligning state between technical coordination requirements and actual coordination of development teams. In the course of this thesis, three selected approaches of [Valetto et al., 2007], [Strohmaier et al., 2009] and [Kwan et al., 2009] were discussed in chapter 4 determining their characteristics and similarities.

> *Congruence is achieved when coordination capabilities match or exceed coordination required. [Cataldo et al., 2006]*

These measures provide both, global and local focused congruence measurement and prioritization of coordination requirements and also offer monitoring capabilities of the congruence state to intervene if a trend of incongruence emerges. For achieving congruence, [Sarma et al., 2008] denote two basic strategies *'lowering demand for coordination and increasing coordination capacity'*, and indicate that these two strategies are certainly dependent on each other.

- *Lowering demand* by reducing code dependencies and therefore dependencies among software artifacts by restructuring the code structure
- *Increasing coordination capacity* by identifying coordination gaps between members and support them for example with collaboration or awareness tools

2.3 Software Development Quality and Success Metrics

For improving software development processes and therefore the software quality, a wide range of software quality metrics was established in the research area of software engineering. Until several years ago, traditional metrics were used for predicting failure-proneness in software projects such as code churn [Graves et al., 2000] or code dependencies [Schröter et al., 2006], which do not capture the influence of team dynamics in software development teams relating software quality with the organizational structures.

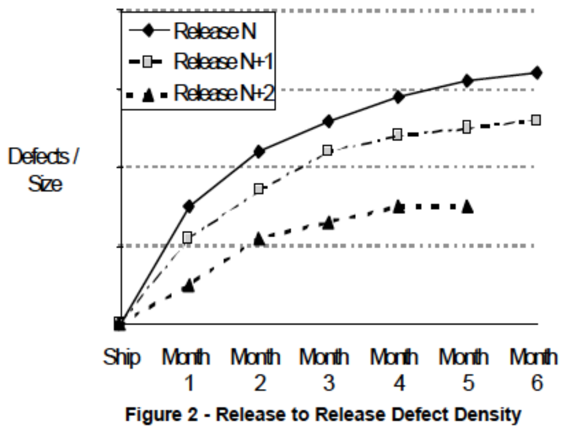


Figure 2.9: Release defect density evolution [Westfall, 2008]

An important established measure for software quality is the *defect density* which provides the possibility of comparing different software projects through normalizing. It is further used for the identification of faulty software components by comparing the number of defects per component with the relative number of failures. Figure 2.9 shows the evolution of defect density over time from release to release.

> **Defect Density** is the measure of totally known defects (bugs) divided by the size of the software entity (components) being measured. The number of known defects is the count of total defects identified against a particular software entity during a particular time period [Westfall, 2008].

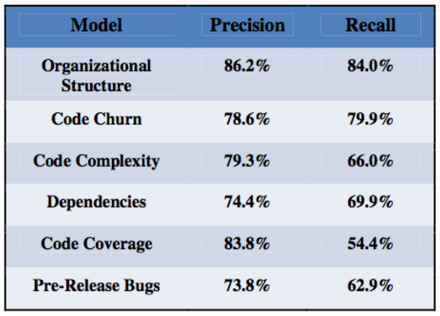
$$\frac{\text{number of known defects}}{\text{size}} \tag{2.13}$$

[Brooks, 1995] argues that there exists a correlation between product quality and the organizational structure which extends the conjecture of [Conway, 1968], who has not stated a qualifying causal relationship between socio-technical architectures.

Two influential examples of relating software quality and organizational dynamics are the work of [Nagappan et al., 2008] that created a metric scheme to quantify the impact of the organizational structures on software quality by observing correlations with failure-proneness, and the work of [Crowston et al., 2004] measuring the success of Open Source Software projects.

[Nagappan et al., 2008] define eight measures, listed below, and evaluate them by predicting failure-prone binaries in Windows Vista. Figure 2.10 illustrates the results of comparing different previous prediction models with the organizational metric prediction model, which denotes a rather high precision and recall level compared to code-based failure-proneness models.

1. number of engineers
2. number of ex-engineers
3. edit frequency
4. depth of master ownership
5. percentage of organizational contribution to development
6. level of organizational code ownership
7. overall organizational ownership
8. organizational intersection factor



Model	Precision	Recall
Organizational Structure	86.2%	84.0%
Code Churn	78.6%	79.9%
Code Complexity	79.3%	66.0%
Dependencies	74.4%	69.9%
Code Coverage	83.8%	54.4%
Pre-Release Bugs	73.8%	62.9%

Figure 2.10: Results of the organizational failure prediction model by [Nagappan et al., 2008] compared to code-based failure prediction models

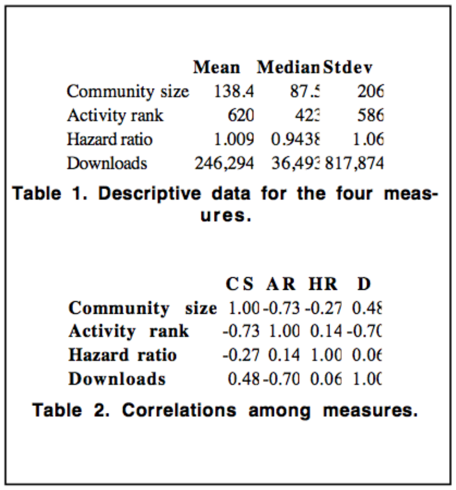
[Crowston et al., 2004] deal with metrics, measuring the process of system development of Open Source projects analyzing

- development team size,
- bug fixing time (hazard ratio),
- activity rank and
- number of downloads.

Therefore, table 2.11 illustrates an overview of extracted data concerning the quality measures and their correlation values.

2.4 Mining Software Repositories

Socio-technical congruence benefits from Open Source Software, but needs to deal with the challenges and difficulties of extracting and mining data according to [Crowston and Howison, 2005] who describe issues of data extraction and also data interpretation for generating social networks derived from software repositories. Specially since the increasing appearance of web-mediated accessible Open Source projects like



	Mean	Median	Stdev
Community size	138.4	87.5	206
Activity rank	620	423	586
Hazard ratio	1.009	0.9438	1.06
Downloads	246,294	36,493	817,874

Table 1. Descriptive data for the four measures.

	CS	AR	HR	D
Community size	1.00	-0.73	-0.27	0.48
Activity rank	-0.73	1.00	0.14	-0.70
Hazard ratio	-0.27	0.14	1.00	0.06
Downloads	0.48	-0.70	0.06	1.00

Table 2. Correlations among measures.

Figure 2.11: Descriptive data and correlations among quality measures [Crowston et al., 2004]

SourceForge[1] or Eclipse[2], data mining of software repositories and according bug tracking systems has become easier and a huge amount of data has been available.

2.4.1 Mining Source Code

Since the last decades, a huge problem in software engineering has been the lack of system documentation, because an accurate documentation enhances software quality and reduces development costs. Software reverse engineering provides practices on analyzing source code to reconstruct the software architecture by identifying the system components and their interdependencies [Byrne, 1991]. An approach for extracting a so called *ownership architecture* derived from the developer organization structure to predict the *concrete architecture* of the software system is described by [Bowman and Holt, 1998].

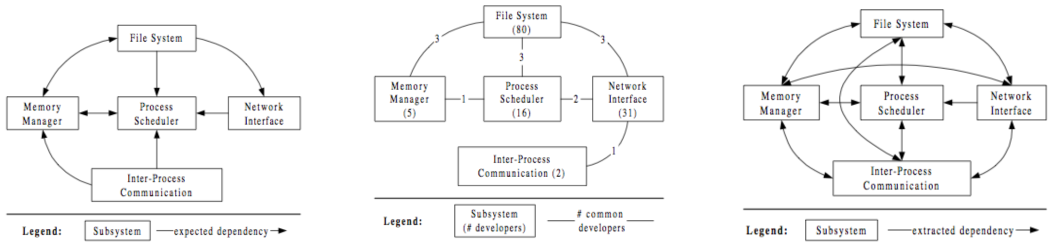

File System
Memory Manager
Process Scheduler
Network Interface
Inter-Process Communication
Legend: Subsystem —expected dependency→
File System (80)
Memory Manager (5)
Process Scheduler (16)
Network Interface (31)
Inter-Process Communication (2)
3
1
2
Legend: Subsystem (# developers) # common developers
File System
Memory Manager
Process Scheduler
Network Interface
Inter-Process Communication
Legend: Subsystem —extracted dependency→

Table 2.2: Conceptual (left), ownership (middle) and concrete (right) system architectures of a Linux Kernel [Bowman and Holt, 1998]

They capture the idea of Conway's Law [Conway, 1968] which states that the organizational structure of an organization influences the design of a product or the

[1]`http://sourceforge.net`
[2]`http://eclipse.org`

system's architecture. To validate this assumption of predicting the concrete system architecture using the *ownership architecture*, [Bowman and Holt, 1998] determine three software systems. Table 2.2 represents the three constructed architecture dependency networks of the Linux Kernel (800KLOC) containing about 1.600 source files, where the conceptual architecture is derived manually from the existing system's documentation, the ownership architecture represents dependencies extracted from revision control logs and the concrete architecture is derived from the actual system implementation. The challenge of creating the system dependency networks is the decision of connecting subsystems. In the ownership architecture, the interconnection of subsystems depends on code ownership, meaning that a subsystem A is connected with a subsystem B if there exists a developer that was working on both subsystems. Further, dependencies in the concrete system architecture network arise via function call relations and variable reference relations, which means that if a source file X calls a function or variable by reference from file Y, then file X depends on file Y - where the files were clustered into different subsystems by using, for example, the source code file directory structure.

	Linux	Mozilla	Aleph
Conceptual			
Edges (E)	9	20	9
Correct (K)	9	19	9
False Negative (M)	10	15	3
False Positive (V)	0	1	0
Ownership			
Edges (E)	12	30	18
Correct (K)	11	24	9
False Negative (M)	8	9	2
False Positive (V)	1	6	9

Figure 2.12: Using conceptual and ownership architecture as predictor of concrete software architecture [Bowman and Holt, 1998]

[Bowman and Holt, 1998] argue that if Conway's Law is valid, the ownership architecture network will be an adequate predictor for the concrete architecture network. The following metrics according to the derived conceptual and ownership architecture networks were calculated and compared:

- *predicted edges* (E) representing the number of predicted edges
- *correct edges* (K) representing the number of concrete edges which were correctly predicted
- *false negative* (M) representing the difference of existing concrete edges and not predicted edges
- *false positive* (V) representing the number of predicted edges which do not exist in the concrete architecture

The results summarized in figure 2.12 indicate that the ownership architecture network is even proper for predicting dependency edges of the concrete architecture network as the conceptional architecture network.

However, [Souza et al., 2005] extract developer networks from software repositories, observing source code dependencies using call graph analysis and developer co-occurrence. Therefore, the authors create Augur, a visualization tool investigating the question *'how the relationships between software modules expose one view of the underlying social structure'* [Souza et al., 2005]. Figure 2.13 shows a screenshot of the *Augur* tool providing software developers with an overview of dependencies among software artifacts and the according software development process activities. By contrast to [Bowman and Holt, 1998], the conclusion of [Souza et al., 2005] is that regarding to Conway's Law both socially-inferred and technically-inferred system architectures reflect and constrain bilaterally, and that the relations between software artifacts also include relations between developers.

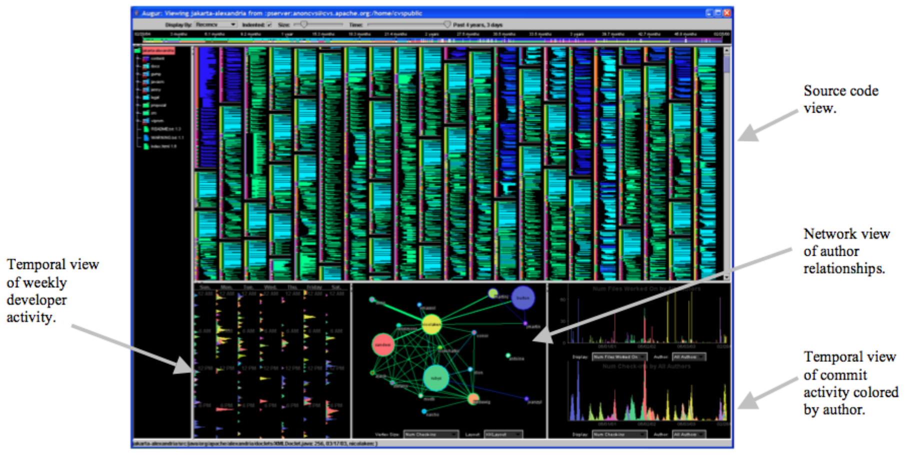

Source code view.
Network view of author relationships.
Temporal view of weekly developer activity.
Temporal view of commit activity colored by author.

Figure 2.13: Augur visualization tool [Souza et al., 2005]

The works of [Strohmaier et al., 2009] and [Wermelinger and Yu, 2008] have influenced this thesis by adopting the approach of data mining code dependencies using software repository metadata files. [Strohmaier et al., 2009] apply a code-inferred dependency network generated with a provided tool of [Wermelinger and Yu, 2008] for analyzing Eclipse plugins' evolution.

This approach differs from previous approaches of mining source code in generating software artifact dependency networks. The major difference consists in extracting code dependencies without touching the code. This is realized by using metadata files which provide information about the technical architecture of the software systems such as the Eclipse project which is organized in plugins. These plugins represent the different subsystems of the software system which were also denoted as software components.

A further difference of this approach is the distinction of static and dynamic code dependencies. The authors provide the possibility of generating three different types of Eclipse plugin-based code-inferred dependency networks extracted from metadata files.

1. *static* dependencies: containing only static (compile-time) dependencies

2. *dynamic* dependencies: containing only dynamic (run-time) dependencies

3. *combined* dependencies: containing static and dynamic dependencies combined in one network

A metamodel for architectural evolution shown in figure 2.14, illustrates the different relations and dependencies. For extracting the statical code dependencies, using the information provided in the $MANIFEST.MF$ file, the relation *depends*, shown in figure 2.14, will be verified. Further, the dynamical code dependencies extractable from the *plugin.xml* file are represented in the relations *use* and *provide* between plugins and extension points. [Wermelinger and Yu, 2008] define these code dependencies as follows:

> A plugin X statically **depends** on plugin Y, if the compilation of X requires Y - if the compilation of X requires Y, and we say that X dynamically depends on Y if X **uses** an extension point that Y **provides**.

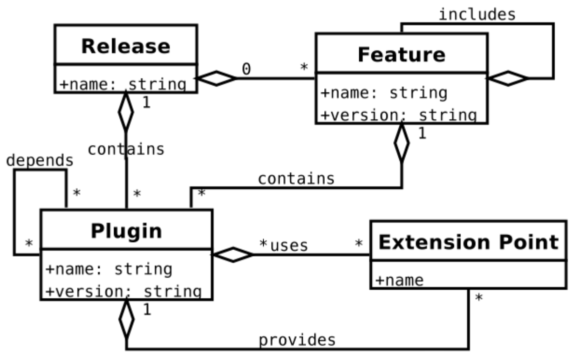


Figure 2.14: Metamodel for plugin-based software architectures [Wermelinger and Yu, 2008]

The discussed publications provide an insight into different practices of generating dependency networks representing relations or interconnections between software artifacts and developers, mining source code architecture. While [Bowman and Holt, 1998] use code ownership attributes mined from revision logs, [Souza et al., 2005] apply call graph analysis and [Wermelinger and Yu, 2008] analyze code dependencies without touching code by using metadata files containing information about the technical architecture stored in software repositories.

Whereas [Bowman and Holt, 1998] and [Souza et al., 2005] also deal with socio-technical aspects related to Conway's Law and possible conclusions, [Wermelinger and Yu, 2008] only focus on the technical aspects, investigating structural design principles by analyzing the evolution of eclipse plugin architectures.

2.4.2 Mining Issue-tracking Systems

Systems such as task management or bug tracking frameworks represent an important element in the software development process and expose revealing information about the evolution of developer activity, incidence of bugs or feature request and also information about developer community or collaboration structures.

[Anvik et al., 2006] devote themselves to the question of bug report assignment through a semi-automated approach of analyzing the former assignments of bug reports archived in the Eclipse bug tracking system Bugzilla. In this case, specially the Eclipse bug report life-cycle, shown in figure 2.15, plays a major role because the status of a bug report changes repeatedly over lifetime and has to be considered. Further, this paper describes the anatomy of a bug report and the different features such as activity logs which were tracked by the system.

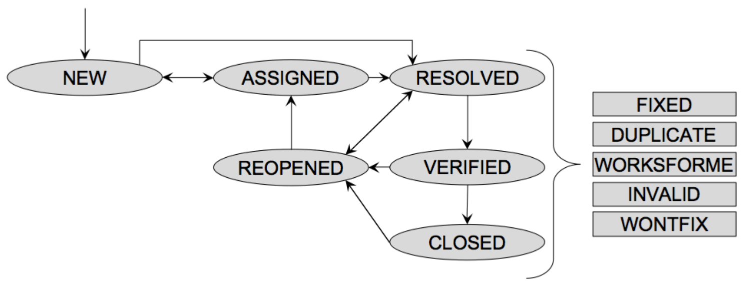


Figure 2.15: Eclipse bug report life cycle [Anvik et al., 2006]

For improving the bug report assignment, [Anvik et al., 2006] use a software component-based classifier for the report assignment and therefore they extracted information about the number of components which were handled by developers mined from the Bugzilla bug tracking system. The extracted information, shown in figure 2.3, provides an insight into the multi-tasking level of software development teams and concludes that the component-based approach is more accurate for software projects comprising developers that concentrate on a small amount of software components.

	Project Components	Min	Mean	Max
Eclipse	17	1	1.8	9
Firefox	30	1	2.9	29
gcc	30	1	3.2	28

Table 2.3: Number of components for which a developer has resolved a bug report [Anvik et al., 2006]

[Schröter et al., 2006] have also mined the Eclipse bug tracking system and Eclipse version database for dedicating bugs to Eclipse components by calculating the defect density of all Eclipse software components. Further they demonstrated how to associate code or developers to failures, and provided their results online hosted by the project[3] website.

The next level of mining issue-tracking systems constitutes the extraction of social structures by relating developers with software artifacts. This would provide a better insight into the organization or structure of development teams, the monitoring of developer contribution or level of activity and also the possibility of locating experts. The extraction of affiliation or participation networks was done by several authors such as [Linstead et al., 2007] who mined the Eclipse bug data obtaining developer contribution lists and modified source files per developer data through author-topic modeling. This approach provides the possibility of an automated framework for extracting author-document relations. Therefore, they have produced an author-document matrix m, representing the participation network, where

$$m[i,j] = \begin{cases} 1, & \text{If author } i \text{ contributes to document } j \\ 0, & \text{otherwise} \end{cases}$$

The work of [Linstead et al., 2007] deals with the extraction of social structures using bug tracking data from the Eclipse bug tracking system, which is usually accessible in XML files comprising a set of bug report entries. Other works such as [Crowston and Howison, 2005] or [Xu et al., 2006] describe the extraction of data from SourceForge[4], a web-mediated Open Source management software repository.

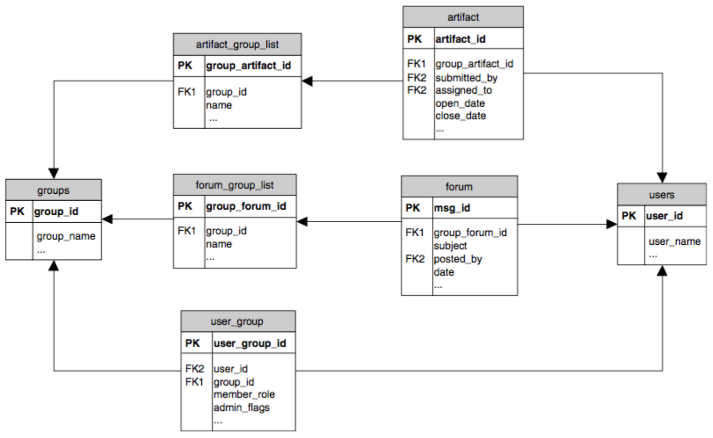


Figure 2.16: Excerpt of SourceForge database schema [Xu et al., 2006]

The difference to the previous discussed Eclipse bug tracking system is the provision of data via database access. The database schema, illustrated in figure 2.16, is still the same for all hosted Open Source projects. The mining of data from a SourceForge data dump is described by [Xu et al., 2006] who used the relations stored in the database to identify member activities and participations in different projects.

[3]`http://www.st.cs.uni-sb.de/softevo/`
[4]`http://sourceforge.net/`

The database provides therefore information about projects, developers and occurring communities, and the different roles of developers are extractable using the seven tables which contain the following information:

- **Table** *groups***:** list of all projects
- **Table** *artifact_group_list***,** *artifact***:** stores information about bug tracking and feature requests or document writing
- **Table** *forum_group_list***,** *forum***:** contains developer participation activities in discussion forums
- **Table** *users***:** includes all user data like user id or joining date
- **Table** *user_group***:** illustrates relationships among projects and project leaders and core developers

Using these tables for data processing, [Xu et al., 2006] were able to extract relations between developers and projects, including specifying different member roles such as project leader, core developer and passive or active users.

Further, [Crowston and Howison, 2005] use bug tracking data of SourceForge projects for extracting information about developer interaction networks, as shown in figure 2.17, by identifying cooperations among developers through bug related discussions traced in bug reports for examining the social structure of open source software projects.

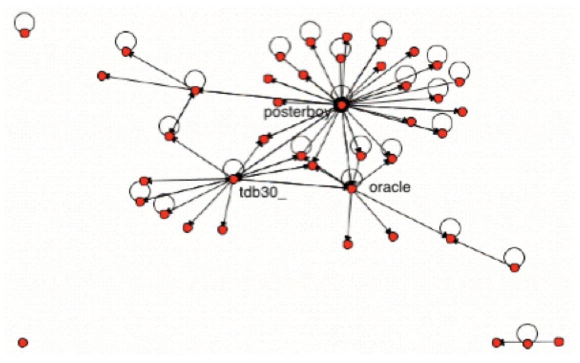


Figure 2.17: Interaction network mined from a SourceForge project [Crowston and Howison, 2005]

The influence of technologies like tagging mechanisms find more and more practical application in many disciplines of collaborative work. In software development tagging-based collaborative tools such as the IBM Jazz work item management platform were used for work item organization. [Treude and Storey, 2009] examine the impact and influence of using tagging-mechanism supporting software development for improving the alignment of socio-technical aspects and the identification of informal processes in software engineering. The authors argue that *'collaborative tagging* *implies an underlying social structure'* and therefore they mine work items including information about developers and tags.

2.4.3 Mining Cross-Media Resources

The previous approaches have focused on the features that an issue-tracking systems provide for improving the usefulness of such systems or gathering latent socio-technical structures in software repositories. However, nowadays software development-supporting systems comprise also different Web 2.0 applications for collaboration or social networking like wikis or rather new technologies like Twitter or Google Wave. Therefore, another direction in mining social activities, not only in software development processes, includes cross-media analysis for extracting social networks, where the preliminary perspective was so far limited only to the activities traced in software project management systems or so called software repositories.

For depicting more accurate latent social structures or community patterns within software projects, the step into mining cross media resources would be incidental. [Cao et al., 2008], for example, describe methodologies like cross-media analysis and social pattern-based analysis introduced by [Degenne and Forsé, 2004] and were concerned with the mining of social network data extracted from varying sources segmented in

- **non-public access** (private SMS or chats)
- **semi-public access** (mailing lists, group chats, forums or protected wikis)
- **public access** (blogs, public wikis, public image or video sharing)

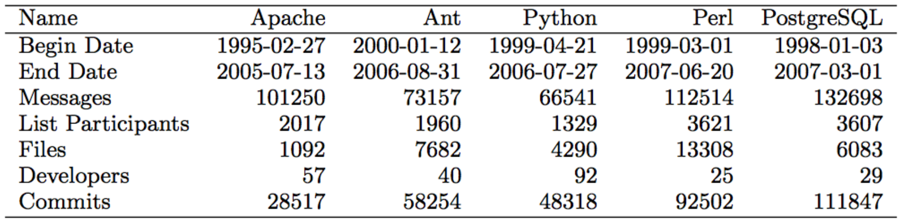
A similar direction follows the approach of [Agrawal et al., 2003], which mines newsgroups for generating social networks specially for identifying user categories of proponents and opponents of newsgroup topics through analyzing the arisen social network structures. In comparison, [Bird et al., 2008] mine social structures from mailing lists to examine the social organization structures of large, complex and stable software projects such as the Apache Webserver, Python or Perl project. The mining approach is based on text parsing of all email messages for reconstruction of conversation threads regarding the following email message entries:

- date
- body
- sender name
- sender email address
- message id header
- in-reply-to header

The decision of linking two specific messages to sender or replier is defined by [Bird et al., 2008] as follows:

> If the message id of message A appears in the in-reply-to header of message B, then B was sent in response to A which indicates that the sender of B found message A *interesting.*

Table 2.18 gives an overview of gathered data mined from mailing lists by [Bird et al., 2008].



Name	Apache	Ant	Python	Perl	PostgreSQL
Begin Date	1995-02-27	2000-01-12	1999-04-21	1999-03-01	1998-01-03
End Date	2005-07-13	2006-08-31	2006-07-27	2007-06-20	2007-03-01
Messages	101250	73157	66541	112514	132698
List Participants	2017	1960	1329	3621	3607
Files	1092	7682	4290	13308	6083
Developers	57	40	92	25	29
Commits	28517	58254	48318	92502	111847

Figure 2.18: Data overview extracted from mailing lists [Bird et al., 2008]

The discussed publications in this chapter cover topics closely related to socio-technical congruence, such as social network analysis, software quality measures and data mining. An overview of social network analysis is given by discussing fundamentals like different network types or network topology properties in section 2.1.

Moreover, characterizing properties of real world social networks such as the small-world and scale-free properties were discussed. These insights into network characteristics benefit the understanding of social- and code-inferred dependency networks, which represent a software architecture gathered from two different perspectives for improving congruence achievement. Concerning the nature of social networks appearing as small-world and scale-free networks, a corresponding study of Open Source development communities was recognized and examined [Xu et al., 2006].

For the generation of code-inferred dependency networks which reflect the technical coordination requirements, data mining of software repositories is necessary. An insight into different practices on mining and analyzing source code was provided by [Bowman and Holt, 1998], [Souza et al., 2005] and [Wermelinger and Yu, 2008]. Further, provided network data comprised in issue-tracking systems, discussed by [Crowston and Howison, 2005], [Linstead et al., 2007] and [Xu et al., 2006], have exposed revealing information about the evolution of developer activity, incidence of bugs and also information about developer collaboration structures. These social network data were used for the generation of socially-inferred dependency networks representing coordination capacity.

Socio-technical congruence deals with issues of aligning the technical coordination requirements with the social coordination capabilities [Cataldo et al., 2009]. In section 2.2, publications concerning the understanding [Conway, 1968], [Cataldo et al., 2006], measuring [Sarma et al., 2008], and achievement [Valetto et al., 2007], [Strohmaier et al., 2009] [Kwan et al., 2009] of congruence were discussed. Section 5.5 has covered different works about code quality [Westfall, 2008] and software quality and success metrics to quantify the impact of the organizational structures on software quality [Nagappan et al., 2008] and [Crowston et al., 2004].

Hence, socio-technical congruence underlines the importance of detecting relationships among social and technical dependencies, because an association with development productivity and effectivity was acknowledged [Cataldo et al., 2008].

Chapter 3

Data Sets and Network Construction

Since the emergence of web-supported distributed software projects, especially in the area of Open Source projects, access to a large range of software repositories has become available. This chapter describes the data sets of the Eclipse[1] SDK and IBM Jazz[2] collaboration tool *Rational Team Concert*[3] which were used in this work, and which include useful information for generating dependency networks derived from collaboration tracking between developers or source code analysis.

However, this information is implicitly available and could be mined from web-based software repositories using practices as introduced in chapter 2.4.1. Therefore, this chapter depicts the characteristics and structures of the two real-world data sets and the essential steps of mining and realized transformations using the mined data.

3.1 Eclipse SDK Data Set

The Eclipse SDK was selected because of the idea of [Strohmaier et al., 2009] to use this Open Source software for their socio-techncial congruence measure and the fact that the Eclipse project provided a stable environment relating to technical architecture and developer community. The data set of the bug tracking system contains about 200.000 bug reports and feature requests from the Eclipse Bugzilla[4] data base exported in XML format for the MSR Challenge[5] 2008, including feature requests from the year 2001 to 2007. This data was used to generate the socially-inferred dependency network described in the next section. The creation of the code-inferred dependency network is illustrated in chapter 3.1.2.

[1]`http://www.eclipse.org/`

[2]`http://jazz.net/`

[3]`http://jazz.net/projects/rational-team-concert/`

[4]`https://bugs.eclipse.org/bugs/`

[5]`http://msr.uwaterloo.ca/msr2008/challenge/`

3.1.1 Social-inferred Eclipse Dependency Network

The available Bugzilla data set provides a huge amount of bug reports and feature requests, and every bug report entry contains several attributes. An example bug report item using the Bugzilla schema[6] is illustrated in appendix B.1. For the purpose of generating a socially-inferred dependency network, the following bug report attributes shown in table 3.1 were extracted and stored in CSV format shown in figure 3.2. This information was used as starting point for the socially-inferred dependency network construction.

Attribute	Description
bug id	The bug ID.
creation_ts	The times of the bug's creation.
product	Eclipse software product.
component	The product component.
reporter	The user who reported this.
assigned_to	The current owner of the bug.
who	Developers involved through discussion.

Table 3.1: Mined Eclipse bug report attributes

time	bug_id	product/component	developer
2004-08-07	67877	PlatformUserAssistance	konradk@ca.ibm.com
2004-08-07	67877	PlatformUserAssistance	birsan@ca.ibm.com
2004-09-21	67879	PDEBuild	pascal_rapicault@ca.ibm.com
2003-09-24	67882	PlatformUI	michaelvanmeekeren@yahoo.ca
2003-10-03	67882	PlatformUI	erich_gamma@ch.ibm.com
2003-10-03	67883	PlatformSWT	kim_horne@ca.ibm.com

Table 3.2: CSV format of mined bug report entries

One of the difficulties of generating the socially-inferred dependency networks is the decision of linking nodes. For example, [Crowston and Howison, 2005] generated an edge between a component A and a developer X, if messages of X were tracked in bug report B.

The same approach was used by [Strohmaier et al., 2009], linking a component A with developer X if they are transitively related via a bug report. For the Eclipse bug tracking data this would generate a three-mode or tripartite structure as shown

[6]`http://tinyurl.com/bugzilla-schema`

in figure 3.3 of three different node types, software components C with 16 nodes, developers D with 14.741 nodes and bugs B counting 99.089 nodes and an amount of 439.324 edges. For the creation of the socially-inferred dependency network, the relations between components and developers are interesting and useful.

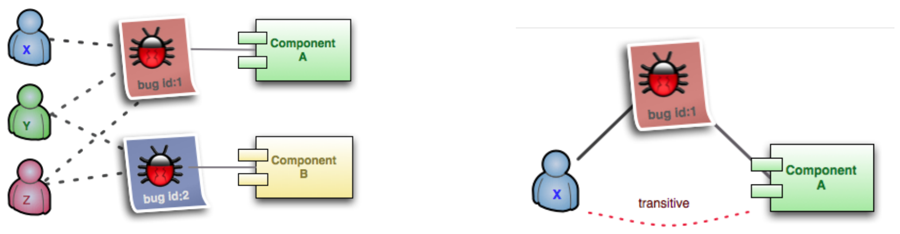


Table 3.3: Three-Mode network structure (left) and transitive relation (right)

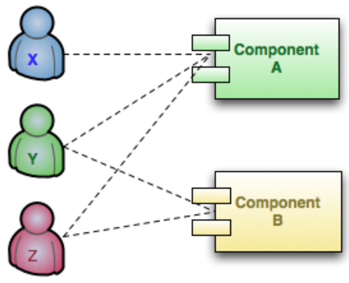
This information is given implicitly or transitively, see [Wasserman and Faust, 1994], who denote that if, using the example from figure 3.3, developer X is tied with bug id 1 and component A is also tied with bug id 1, a relation between developer X and component A can be inferred. In this case it makes sense, because if a developer co-occurs with a software component or another developer within a bug report, a relation obviously exists since an activity of collaboration takes place. Using the information of the three-mode structure, three different two-mode networks can be generated:

- *developer* $\times$ *bug* network ($D{\times}B$)
- *component* $\times$ *bug* network ($C{\times}B$)
- *component* $\times$ *developer* network ($D{\times}C$)

For the generation of the dependency network the $D{\times}C$ two-mode network is useful, because it includes information about the relations of developers tied with different components and vice versa. To get the relations between the software components derived from social collaboration activities of the developers, the generated two-mode $D{\times}C$ network was transformed into a one-mode network containing only software components and their relations.

The calculation was realized by transforming the $D{\times}C$ two-mode network into its $\perp$ (bottom) projection, see chapter 2.1.2 or [Latapy et al., 2008]. Thus, the transformed $C{\times}C$ network answered the linking question and revealed in satisfying the condition if two components share the same neighbor or in this case developer, then they got linked. The edge weights of the $C{\times}C$ dependency network result through the transformation step, representing the number of shared neighbors. Using the example of the three-mode network in figure 3.3, the two-mode adjacency matrix $D{\times}C$ would be defined as shown in figure 3.4. After the transformation step,

$$D{\times}C = \left(\begin{array}{ccc} & A & B \\ X & 1 & 0 \\ Y & 1 & 1 \\ Z & 1 & 1 \end{array}\right)$$



$$C{\times}C = \left(\begin{array}{ccc} & A & B \\ A & 3 & 2 \\ B & 2 & 2 \end{array}\right)$$

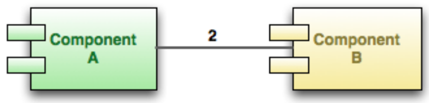


Table 3.4: Two-mode (top) and according one-mode $\top$-projection network (bottom) with corresponding adjacency matrices

$C{\times}C = D{\times}C^{\mathrm{T}} \times D{\times}C$, the $\top$-projection adjacency matrix $C{\times}C$ would look like shown in figure 3.4.

The information that is now provided, is included in the diagonal of the matrix denoting the degree of the components within the two-mode network, that depicts how many developers were connected to a specific component and also different edge weights result in the amount of developers that are shared through two components. For example, component A and B are related with the same two developers Y and Z. Thus, they are connected with an edge weight of 2. Therefore, the dependency of two software components is higher if the number of shared developers is higher, because it indicates a raised level of relation importance between the two software components.

The generated socially-inferred dependency network S_{E} extracted from the Eclipse Bugzilla data set is illustrated in figure 3.1, where the strength of the edge lines indicate the edge weights. The thicker and darker an edge, the more weight is assigned to the edge. Additionally, it has to be observed that during the course of this thesis, conducting the empirical study in chapter 5, the generated socially-inferred dependency network was also split up into monthly time stamps for observing the network evolution. The creation of the code-inferred Eclipse dependency network is described in the next chapter.

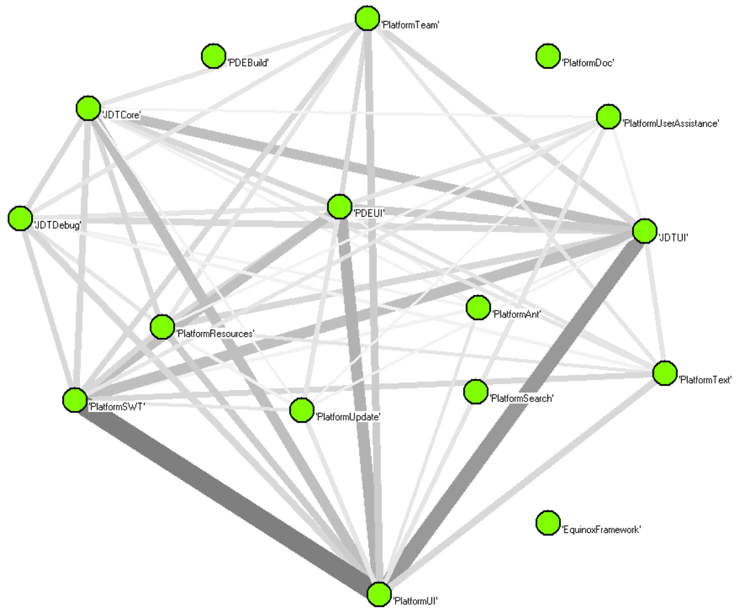


Figure 3.1: Generated Eclipse socially-inferred dependency network S_{E}

3.1.2 Code-inferred Eclipse Dependency Network

The generated code-inferred dependency networks provided by [Wermelinger and Yu, 2008] comprise static and dynamic code dependencies among software components or plugins of the Eclipse SDK Release 3.3.1.1, which is the according release from the Bugzilla data set, described in section 3.1.1.

The first $C_{\mathrm{E\_stat}}$ represents the Eclipse SDK code-inferred dependency network including only static dependencies, the next network $C_{\mathrm{E\_dyn}}$ included only dynamic dependencies and $C_{\mathrm{E\_all}}$ combined both of them. In figure 3.2, the code-inferred dependency network is illustrated including both dynamic and static code dependencies, and the same 16 software components as in the socially-inferred dependency network described in the previous chapter. The representation of the edges is the same as described in chapter 3.1.1 and a list of the Eclipse component labels is provided in appendix B.2.

3.2 IBM Jazz Data Set

The second data set consists of an IBM Jazz work item collection tracking software development activities from 2006 to 2008 and the according release 1.0 of the Rational Team Concert[7] (RTC) software project. [Treude and Storey, 2009] consider in a case study the use of tagging mechanisms to support collaborative software development by *'bridging the gap between technical and social aspects of managing work items'*. Therefore, [Treude and Storey, 2009] use a data set containing information about tags, work items and developers that could be extracted from the web-based

[7]`http://jazz.net/projects/rational-team-concert/`

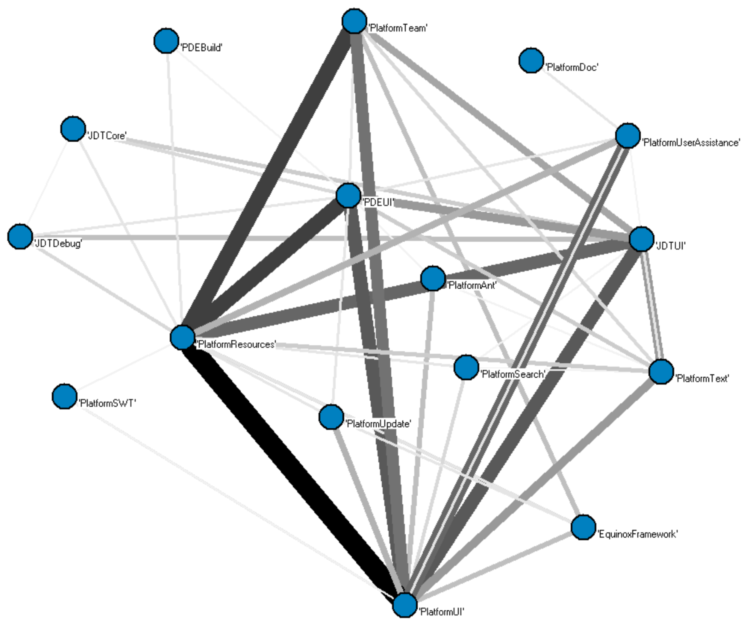


Figure 3.2: Generated Eclipse code-inferred dependency network C_{E}

IBM Jazz work item system. An excerpt of a work item example is illustrated in appendix B.6. Compared to the Eclipse data set, a minor difference lies in the varying development phases. While the Eclipse data set reflects activities in the maintenance phase, the IBM Jazz data were generated in the implementation phase developing the first release.

3.2.1 Social-inferred Jazz RTC Dependency Network

The challenge for generating a socially-inferred dependency network derived from the IBM Jazz data set was to identify or match the used tags in the work item collection with the relevant software plugins or components. Table 3.5 depicts an excerpt of the used data set containing information about the work item id, the used tag and the related developer working on or discussing a specific work item. Please note that the developers were alienated because of protecting privacy.

Creation time	**workitemID**	**Tag**	**Developer**
15.11.2006	4	ux	[UUID _wNof8PYJEdqU64Cr2VV0dQ]
13.12.2006	10	ux	[UUID _wNof8PYJEdqU64Cr2VV0dQ]
15.12.2006	10	mwhot	[UUID _wNof8PYJEdqU64Cr2VV0dQ]
07.06.2006	12	collaboration	[UUID _KGRY4CFWEdq-WY5y7lROQw]
15.11.2006	12	ux	[UUID _wNof8PYJEdqU64Cr2VV0dQ]
07.06.2006	13	workitem	[UUID _KGRY4CFWEdq-WY5y7lROQw]
15.11.2006	13	ux	[UUID _wNof8PYJEdqU64Cr2VV0dQ]
07.06.2006	17	workitem	[UUID _KGRY4CFWEdq-WY5y7lROQw]

Table 3.5: Excerpt of the extracted IBM Jazz RTC data set

Using this data, a three-mode network structure could be generated which includes work items W including 18.592 nodes, tags T with 1.169 nodes and developers D counting 360 nodes and an amount of 52.963 edges. The approach of using the three-mode network to produce two-mode networks using transitively relations is still the same as applied in chapter 3.1.1.

However, an intermediate step was necessary to match the tags T with existing software components. Therefore a plugin list was extracted from the software repository and matched against the tag list. Known issues such as individual tag using, internal specific idioms or the use of singular or plural forms had to be resolved with NLP[8] methods and results in a list of 20 tags which matched a specific software plugin or component.

After that, the new three-mode network structure contains a reduced amount of tags and the generation of the dependency network transforming the $T \times D$ network into a one-mode network $T \times T$ projection resulted in a socially-inferred dependency network S_{J}, containing 20 software components related to the IBM Jazz Rational Team Concert software derived from the tag matching and developer collaboration activities. Figure 3.3 illustrates the generated socially-inferred dependency network, and a list of component labels is shown in appendix B.

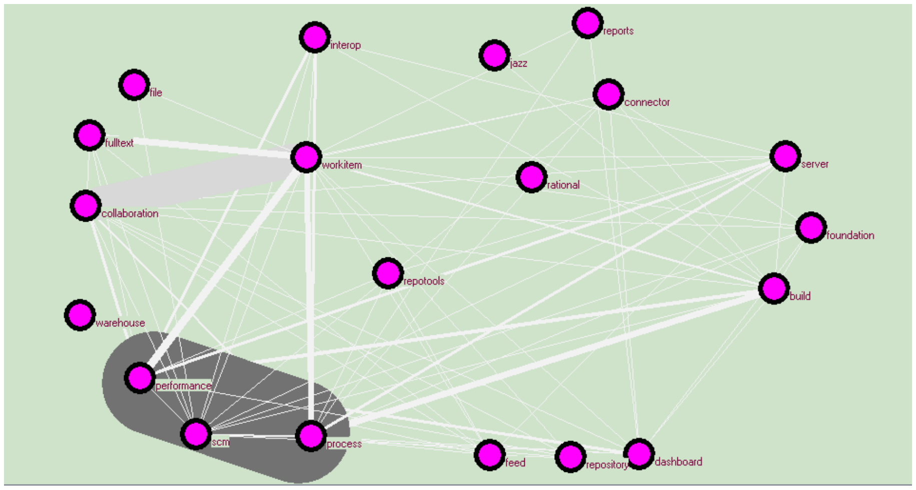


Figure 3.3: Generated Jazz socially-inferred dependency network S_{J}

3.2.2 Code-inferred Jazz RTC Dependency Network

For the code analysis of the IBM Jazz Rational Team Concert project, to generate the code-inferred dependency network, the provided tool by [Wermelinger and Yu, 2008] was used to extract the compile-time (static) and run-time (dynamic) dependencies between the software plugins or components. This approach for generating code dependency is based on analyzing only the meta files of the specific plugins which include static and dynamic code dependencies. The IBM Jazz RTC project

[8]Natural Language Processing

shares the same architectural structures as an Eclipse project and is also organized with plugins, features and releases as described in the metamodel of software architecture in figure 2.14. Therefore, the architectural structure of the software is stored in metadata files which provide any necessary information without concerning the source code. Examples of the *plugin.xml* and *MANIFEST.MF* files of the RTC project are illustrated in appendix B.3. Further, an overview of the used tool set provided by [Wermelinger and Yu, 2008] and described in more detail in [Yu and Wermelinger, 2008], is shown in appendix A.2.

The generated code-inferred dependency network C_J, using the tool set of [Wermelinger and Yu, 2008] is shown in figure 3.4. The network also denotes all dependencies, means combining static and dynamic dependencies and contains the same 20 components as represented in S_J. The representation of the edges is related also to their weights, the higher the edge weight, the thicker and darker the edges are illustrated.

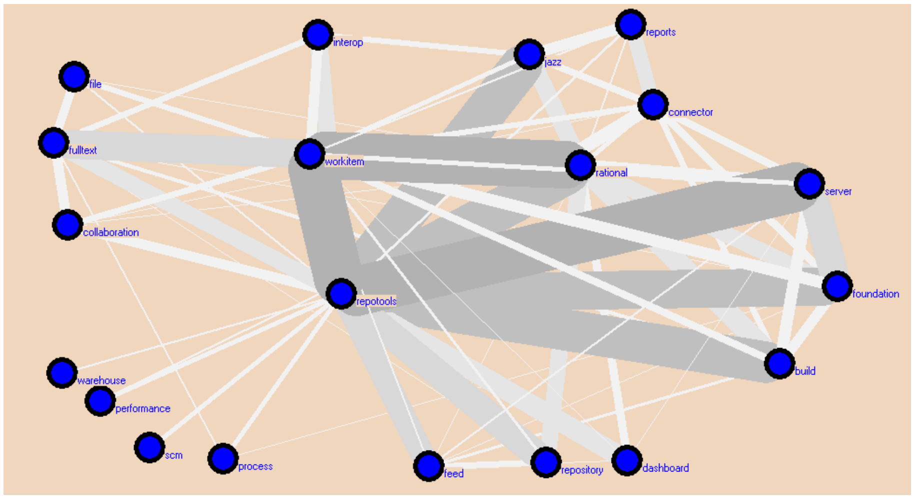


Figure 3.4: Generated Jazz code-inferred dependency network C_J

3.3 Social Network Analysis of the Generated Networks

The generated socio-technical dependency networks shall provide an insight into the different relation characteristics between software artifacts derived from code dependencies as well as organizational structures through the collaboration activities of the software developer community. Therefore, an interesting point was to gain knowledge about the generated networks from a social network analysis perspective. For this purpose, several social analysis metrics were calculated to get attributes and characteristics, which were helpful for interpreting the results of the socio-technical congruence study in chapter 5.

Table 3.6 provides an overview of the calculated social network analysis measures that were theoretically discussed in chapter 2.1. Interesting at this point were the measures of diameter and the clustering coefficient. These two measures are indicators of the small-world phenomenon, if the networks possess a low diameter and therefore a high clustering coefficient, determined by [Watts and Strogatz, 1998] and [Newman, 2002].

Comparing the results of the two data sets of Eclipse and Jazz, the Eclipse derived networks met the requirements of the small-world phenomenon much better. A high clustering coefficient indicated a lower risk of redundancies through breakdowns of network nodes and therefore provided a more stable system. A relatively small network diameter indicated short and fast communication paths. Besides, the evolution of the degree distribution, shown in table 3.7, denotes the attributes of scale-free networks. Therefore, it was interesting to observe that the generated dependency networks share the same characteristics as small-world networks which represent real world networks better than randomly generated networks.

ECLIPSE	$\mathbf{C_{E\_all}}$	$\mathbf{C_{E\_stat}}$	$\mathbf{C_{E\_dyn}}$	$\mathbf{S_E}$
n	16	16	16	16
possible edges	120	120	120	120
actual edges	40	31	32	98
max edge weight	24	23	12	2547
$cc.$	0.68	0.66	0.64	0.96
Δ	0.33	0.26	0.27	0.82
D	4	6	4	7
c_c	0.71	0.66	0.66	0.96
c_b	0.33	0.43	0.43	0.51
$cliques$	7	7	6	8

JAZZ	$\mathbf{C_{J\_all}}$	$\mathbf{C_{J\_stat}}$	$\mathbf{C_{J\_dyn}}$	$\mathbf{S_J}$
n	20	20	20	20
possible edges	190	190	190	190
actual edges	82	76	44	81
max edge weight	165	66	99	12180
$cc.$	0.55	0.57	0.26	0.66
Δ	0.43	0.40	0.23	0.43
D	18	17	27	31
c_c	0.63	0.38	0.31	0.26
c_b	0.59	0.52	0.82	0.44
$cliques$	7	7	4	8

Table 3.6: Social network analysis measure results, where the columns comprise the Eclipse C_E and Jazz C_J code-inferred dependency networks, divided into static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies, where S_E and S_J denote the socially-inferred dependency networks. The rows contain topology measures, where n denotes the number of network nodes and *cc.* the clustering coefficient, Δ the density, D the network diameter, c_c closeness centrality and c_b betweenness centrality (for definitions see chapter 2.1.3)

However, table 3.7 denotes no indication of a power law distribution which is an indicator for scale-free networks, and the use of preferential attachment as network evolution or construction principle could not be observed using only this data. Scale-free network properties provide advantages compared to other network types, contributed through their prevalence in social networks, because they are *'robust against accidental failures but vulnerable to coordination attacks* according to [Barabasi and Bonabeau, 2003]. Therefore, for example the degree distribution of S_{E} shows a skewed distribution, whereas the degree distributions of the remaining networks are rather similarly characterized and differ significantly from S_{E}. Moreover, these measures were also supportive in finding influence factors concerning socio-technical congruence.

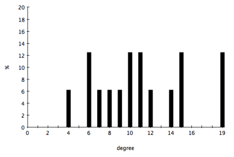
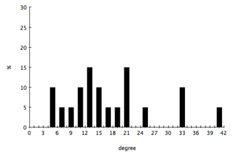
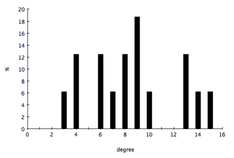
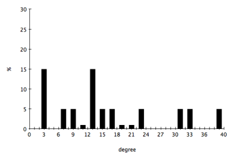
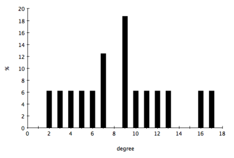
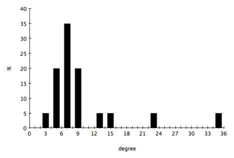
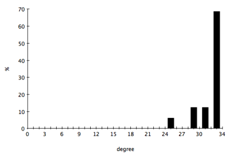
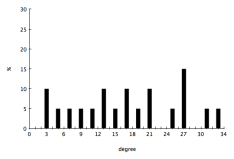
Eclipse	*Degree Distribution*	*Jazz*	*Degree Distribution*
$\mathrm{C}_{\mathrm{E\_all}}$		$\mathrm{C}_{\mathrm{J\_all}}$	
$\mathrm{C}_{\mathrm{E\_stat}}$		$\mathrm{C}_{\mathrm{J\_stat}}$	
$\mathrm{C}_{\mathrm{E\_dyn}}$		$\mathrm{C}_{\mathrm{J\_dyn}}$	
S_{E}		S_{J}	

Table 3.7: Overview of degree distributions for Eclipse C_{E} and Jazz C_{J} code-inferred dependency networks, divided into static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies, and socially-inferred dependency networks S_{E} and S_{J}.

In this particular case, the socially-inferred dependency networks are derived from social networks and referencing to Conway's Law [Conway, 1968], who argues that the social or organizational structure influence the technical and would imply that socio-technical congruence may also be affected by social network characteristics. An interesting question that arises is whether code-inferred dependency networks share the same network properties as socially-inferred dependency networks, thus being comparable.

Chapter 4

Selected Congruence Measures

This thesis includes the evaluation of socio-technical congruence measures to determine if existing algorithms are useful and practically applicable. Therefore, this chapter discusses three selected approaches of measuring socio-technical congruence which were applied later on to the two real world data sets described in chapter 3. This chapter provides an overview of the different approaches, characteristics and differences of the congruence measures illustrated on small examples, and also describes the modifications for using the congruence algorithms with the generated data sets.

4.1 Arc Mirroring

The first selected approach for calculating congruence of social and technical structures is an extension of the approach of [Cataldo et al., 2006]. [Valetto et al., 2007] enhance the concepts of unweighted coordination requirements using two approaches, *ArcMirroring*, which is further discussed in this chapter, and node ties.

The basic information for the *ArcMirroring* algorithm are three networks containing information about the different relationships between people, between software artifacts, and also between both of them, which is defined as *work relationship*, illustrated in figure 4.1. Further, [Valetto et al., 2007] formalize the dependencies and relationships and also define an algorithm for calculating the congruence value.

Let $G_{\mathrm{P}} = (P, E_{\mathrm{P}})$ formalize an undirected network of people with a node set P and an edge set E_{P} including relationships between pairs of persons, probably developers, $(i, j) \in P$.

Further, let $G_{\mathrm{S}} = (S, A_{\mathrm{S}})$ formalize a directed network of software artifacts with a node set S and an arc set A_{S} including interconnections between pairs of software artifacts $(v, u) \in S$.

Additionally, let J denote a set of *joins* connecting node $i \in P$ with node $v \in S$. The calculation of *ArcMirroring* based congruence $\cong$ is defined as follows in formula

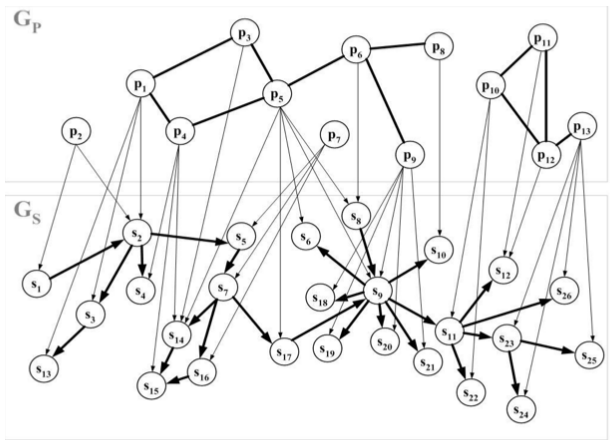


Figure 4.1: A socio-technical software network [Valetto et al., 2007] where G_P represents a people network and G_S a network of software artifacts

4.1, where κ means the number of corresponding or mirrored arcs and γ specifies the number of edges in E_P.

$$Congruence(G_\mathrm{P}, G_\mathrm{S}, J) = \kappa/\gamma \tag{4.1}$$

During the course of this thesis, the *ArcMirroring* algorithm is used in a modified way, because of different available input networks which characterize still similar relationships as described in [Valetto et al., 2007]. For the evaluation of the algorithm two dependency networks including relations between software artifacts or components are given, where one network is generated code-inferred and the other socially-inferred. Details on the generation of the dependency networks can be found in chapter 3.

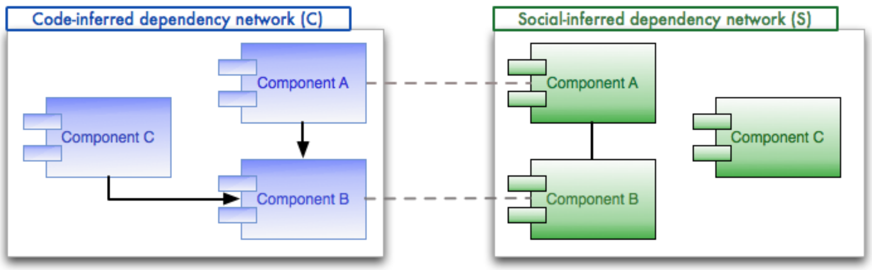

Code-inferred dependency network (C)
Component A
Component C
Component B
Social-inferred dependency network (S)
Component A
Component C
Component B

Figure 4.2: ArcMirroring of code and socially-inferred dependency networks

The results of this algorithm reflect the ratio of founded mirrored edges and existing edges which only match the network structure of both dependency networks where the information about existing edge weights gets lost. [Valetto et al., 2007] and [Cataldo et al., 2006] suggest the possibility of taking weighted coordination requirements into account to refine the results of using unweighted coordination requirements. For the specific example in figure 4.2, the input networks C for code-inferred and S for socially-inferred are specified as adjacency matrices as follows:

$$C = \begin{pmatrix} & A & B & C \\ A & 0 & 1 & 0 \\ B & 0 & 0 & 0 \\ C & 0 & 1 & 0 \end{pmatrix} \qquad S = \begin{pmatrix} & A & B & C \\ A & 0 & 1 & 0 \\ B & 1 & 0 & 0 \\ C & 0 & 0 & 0 \end{pmatrix}$$

Referring to equation 4.1 and the matrices C and S, κ are the number of mirrored edges which would only be the edge between component A and B in matrix C and S and γ the number of edges in C are 2 (A-B, C-A) which results in a congruence measure of $(C \cong S) = 0.5$, which means that there exists a congruence between the two dependency networks of about 50%, where the congruence value would range between 0 and 1.

Moreover, [Valetto et al., 2007] argue that this concept of socio-technical congruence could be used in a global and local focus, providing a better insight into the alignment of technical and social software architectures of the whole network or specific subgroups or project teams.

4.2 Weighted Coordination Requirements

[Kwan et al., 2009] capture the limitations of existing socio-technical congruence measures according to [Valetto et al., 2007] and develop an approach where calculations of congruence would not be based on dichotomized edges. They develop a weighted congruence measure to identify important gaps of coordination. These gaps arise if coordination needs gained through the technical structures will not be met through the organizational activities of the developers and these coordination requirements could be defined as more or less important for the software project.

The model of weighted coordination described by [Kwan et al., 2009] would be more properly by using weighted task dependency and weighted task assignment matrices for calculating coordination requirements, where the measurements of edge weights could range from 0 to 1.

The weighted task assignment and task dependency matrices are defined by [Kwan et al., 2009] as follows:

Weighted Task Assignment (TA) matrix is a $m \times n$ matrix where m is the number of selected people and n denotes the number of selected tasks. Each entry in the matrix defines the strength of the connection between a person i and a task j.

Weighted Task Dependency (TD) matrix is a $n \times n$ matrix where n denotes the number of selected tasks. Each entry in the matrix describes the strength of the relation between two tasks.

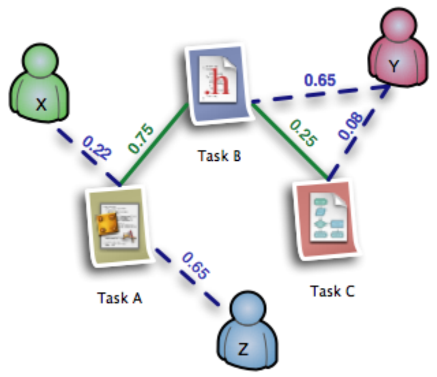


Figure 4.3: Weighted task assignments and dependencies

Figure 4.3 illustrates an example of a weighted task assignment and dependency network which could be represented in the two adjacency matrices TA for task assignment and TD for task dependency illustrated in table 4.1. The edges between tasks and people could be weighted for example by proportion of working hours or the amount of expertise a person has about a specific task. In figure 4.3, person Y spends 0.65 percent of her work time on task B and 0.08 percent on task C. The dependencies between the different tasks could be defined by the degree of coupling between tasks, which means that for example task A depends to 0.75% on task B where task B only depends to 0.25% on task C.

$$TA = \begin{pmatrix} & A & B & C \\ X & 0.22 & 0.00 & 0.00 \\ Y & 0.00 & 0.65 & 0.08 \\ Z & 0.65 & 0.00 & 0.00 \end{pmatrix} \qquad TD = \begin{pmatrix} & A & B & C \\ A & 0.00 & 0.75 & 0.00 \\ B & 0.75 & 0.00 & 0.25 \\ C & 0.00 & 0.25 & 0.00 \end{pmatrix}$$

Table 4.1: Task Assignment and Task Dependency matrices

These two adjacency matrices in table 4.1, are used to calculate the technical coordination requirements CR_{C} and are related to [Cataldo et al., 2006] which have denoted the formula 4.2.

$$CR_{\mathrm{C}} = TA \times TD \times (TA)^{\mathrm{T}} \tag{4.2}$$

The result of the calculation using formula 4.2 implicates the coordination requirements derived from the defined weighted task dependencies and task assignments and represents technical or code-inferred dependencies. The next step for calculating congruence is a subtraction of the actual coordination requirement matrix CR_{S}, which represents the socially-inferred dependencies mined from software repositories, from the calculated coordination requirement matrix CR_{C}.

For the calculations in this thesis, the coordination requirements are represented through the code-inferred dependency networks and the actual coordination requirements are represented through the socially-inferred dependency networks, see an example illustrated in figure 4.4. After subtraction a so called *lack-of-coordination* CR_{L} matrix would be calculated and suggests information about local lacks of coordination between specific, in this case software components or artifacts. To get a global socio-technical congruence value, first change all values less than zero in the *lack-of-coordiantion* matrix to zero and sum the edge weights of the matrix. Next also sum the edge values of the coordination requirement matrix, in this case the code-inferred dependency network, and calculate the ratio between these two sums.

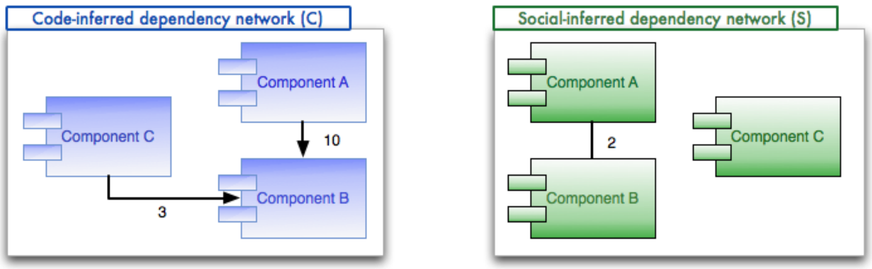

Code-inferred dependency network (C)
Component A
Component C
10
Component B
3
Social-inferred dependency network (S)
Component A
2
Component C
Component B

Figure 4.4: Weighted code and socially-inferred dependency networks

Summarizing the algorithm is defined as follows:

1. Create the *coordination requirement* matrix CR_{C} which is represented by the code-inferred dependency network (see figure 4.4).
2. Create the *actual coordination requirement* matrix CR_{S} which is represented by the socially-inferred dependency network (see figure 4.4).
3. Calculate the lack-of-coordination matrix: $CR_{\text{L}} = CR_{\text{C}} - CR_{\text{S}}$
4. Set all edges in CR_{L} lower than zero to zero.
5. $(CR_{\text{C}} \cong CR_{\text{S}}) = 1 - \left(\frac{\sum_{v \in CR_{\text{L}}} = weight(v)}{\sum_{u \in CR_{\text{C}}} = weight(u)} \right)$

For the specific example in figure 4.4, the network matrices would look like this:

$$CR_{\text{C}} \begin{pmatrix} & A & B & C \\ A & 0 & 10 & 0 \\ B & 0 & 0 & 0 \\ C & 0 & 3 & 0 \end{pmatrix} \text{ - } CR_{\text{S}} \begin{pmatrix} & A & B & C \\ A & 0 & 2 & 0 \\ B & 2 & 0 & 0 \\ C & 0 & 0 & 0 \end{pmatrix} = CR_{\text{L}} \begin{pmatrix} & A & B & C \\ A & 0 & 8 & 0 \\ B & 0 & 0 & 0 \\ C & 0 & 3 & 0 \end{pmatrix}$$

This results in a congruence measure of $(CR_\mathrm{C} \cong CR_\mathrm{S}) = 1-\left(\frac{11}{13}\right) = 0.15$ and provides a congruence range from 0 to 1. This approach differs from other practices which calculate congruence only with ratio values because of providing the possibility of identifying local discrepancies and priorities using the lack-of-coordination matrix.

4.3 Edge Weight Ranking

In comparison with the two congruence algorithms in previous sections, the approach of [Strohmaier et al., 2009] is different. The basis of the algorithm is focused on edge weights which denote the importance of relations between software components or artifacts. The core approach lies in calculating edge ranking correlations of two dependency networks using *Spearman* correlation and the congruence measure is defined as the result of this correlation.

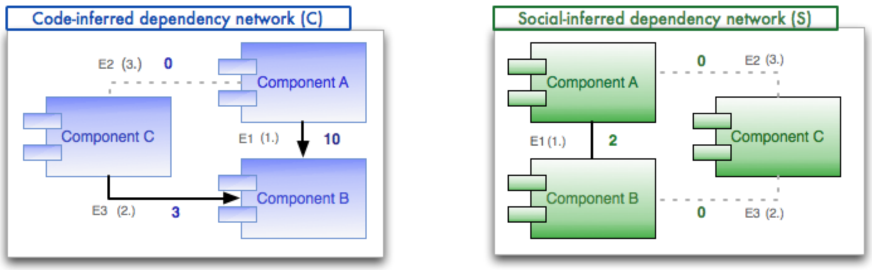


Figure 4.5: Edge weight ranked socially- and code-inferred dependency networks

Using figure 4.5 as reference example, the first step is to number or index all existing edges of both dependency networks. Next generate the edge ranking vectors of each dependency network depending on edge weights where the highest weight ranks the highest. For the small example in figure 4.5, the two edge ranking vectors would be defined as shown in table 4.2. This generates a spearman correlation with the two rank vectors of 0.87 between [-1..+1] which indicates the congruence of the two dependency networks assuming that a high correlation of the edge weight ranking provides a high socio-technical congruence and vice versa.

code-inferred			***socially-inferred***		
Rank	*Edge Index*	*Weight*	***Rank***	*Edge Index*	*Weight*
1	E1	10	**1**	E1	2
3	E2	0	**2.5**	E2	0
2	E3	3	**2.5**	E3	0

Table 4.2: Edge weight ranking vectors of code- and socially-inferred software dependency networks

To make this congruence measure of edge weight ranking comparable with the two other congruence measures discussed in previous sections, a transformation of the values into the same range of [0..1] is necessary. With an inverse Z-transformation according to [Kähler, 2004], the congruence value z could be transformed from $z \rightarrow z'$ taking values in [-1..+1] to taking values in [0..1], using formula 4.3 where s constitutes the standard variance and m represents the mean.

$$z \rightarrow z' = s * z + m \tag{4.3}$$

The transformation of the calculated congruence value of 0.87 using $s = 0.5$ and $m = 0.5$ transforms the congruence value to 0.93 which makes this congruence measure comparable with the other congruence values in the same range of [0..1].

4.4 Similarities, Distinctions and Characteristics

The three selected and described approaches for calculating socio-technical congruence provide different levels of congruence for the small example used in the previous sections. Therefore, it could be noticed that every congruence algorithm has a specific focus and needs a different interpretation of the congruence results. It is difficult to assert which congruence measure fits best, independent of a particular problem domain, but the possibility of having local coordination information or knowledge about coordination priorities benefits congruence achievement.

However, the trend of weighting communication and coordination activities or the importance of relations could be observed. Comparing the *ArcMirroring* and the *WeightedCoordinationRequirements* algorithms, it could be recognized that the approaches are still the same except that one time dichotomized or unweighted edges were used and one time weighted edges to represent the coordination requirements. *ArcMirroring* provides a structural similarity measure where different priorities or importance of relations are ignored.

An interesting observation of the *ArcMirroring* and *WeightedCoordinationRequirements* approach was that both concepts are still motivated by Conway's Law [Conway, 1968] which argues that the technical structure or design of products will be influenced by the organizational structure. [Valetto et al., 2007] using the *ArcMirroring* approach and [Kwan et al., 2009] applying the *WeightedCoordinationRequirements* approach assume the opposite way. This implies that both the *ArcMirroring* and the *WeightedCoordinationRequirements* algorithm proceed on the assumption that the technical structure of a product, for example a software, influences the organizational structure by measuring how the social or organizational structure fits the technical coordination requirements and not inversely. The concept of [Strohmaier et al., 2009] using the *EdgeWeightRanking* algorithm provides an approach for looking in both directions. By calculating edge ranking correlations, it could be considered that the organizational structure could predict or mirror the technical structure and vice versa. Compared to the *ArcMirroring* and *WeightedCoordinationRequirements*, the *EdgeWeightRanking* approach does not directly measure structural similarity, but considers if the strength of component dependencies correlates with the amount of developers dealing with two components.

Chapter 5

Results and Discussion

This chapter provides the results of an empirical study investigating the practical usefulness of the selected socio-technical congruence measures applied to the two real world data sets described in chapter 3.

The socio-technical congruence measures and corresponding algorithms were implemented using a Python and MATLAB framework, see appendix A.1. This framework was used for (1) generating the code- and socially-inferred dependency networks and (2) applying the algorithms to the generated socio-technical network pairs. The first step was to apply them to different network pair set-ups.

		Eclipse			
		$C_{\text{E\_all}} \cong S_{\text{E}}$	$C_{\text{E\_all}} \cong N_{\text{rand}}$	$C_{\text{E\_all}} \cong C_{\text{E\_all}}$	$C_{\text{E\_all}} \cong N_{\text{zero}}$
Eclipse	**Arc Mirroring:**	0.93	0.51	1.00	0
	Edge Weight Ranking:	0.76	0.08	1.00	Inf
	Weighted Coord. Requ.:	0.90	0.12	1.00	0
		Jazz			
		$C_{\text{J\_all}} \cong S_{\text{J}}$	$C_{\text{J\_all}} \cong N_{\text{rand}}$	$C_{\text{J\_all}} \cong C_{\text{J\_all}}$	$C_{\text{J\_all}} \cong N_{\text{zero}}$
Jazz	**Arc Mirroring:**	0.35	0.43	1.00	0
	Edge Weight Ranking:	0.48	0.01	1.00	Inf
	Weighted Coord. Requ.:	0.12	0.03	1.00	0

Table 5.1: Congruence output values on different network pair set-ups, where $C_{\text{E\_all}}$ and $C_{\text{J\_all}}$ denote a code-inferred dependency network extracted from Eclipse (E) or Jazz (J), comprising combined static and dynamic code dependencies (*all*) and S_{E} and S_{J} denote a socially-inferred dependency network mined from Eclipse (E) or Jazz (J). N_{rand} and N_{zero} denote a random and a disconnected zero network.

Table 5.1 shows the different characteristics of the algorithms, applying them with code-inferred and socially-inferred *Eclipse* ($C_{\text{E\_all}}|S_{\text{E}}$) or *Jazz* ($C_{\text{J\_all}}|S_{\text{J}}$) network pairs, where *all* means that in code-inferred networks both dynamic and static dependencies were considered. Additionally, an interesting point was to observe the characteristics of the measures applying untypical network pairs like a random generated network N_{rand}, a disconnected zero network N_{zero} or committing the same code-inferred network for example $C_{\text{E\_all}}$ twice.

The results shown in table 5.1 underline therefore the assumed performances of the algorithms that two identical networks produced a congruence value of 1.00, randomly applied networks generated congruence values in the range of 0 and 0.5 and a comparison with a zero network produced $Null$ or $Infinite$ values, because a level of congruence was not measurable.

5.1 Impact on Congruence by Reducing Data Volume

The amount and accessibility of data extracted from web-mediated distributed software projects play an important role in measuring congruence of socio-technical software architectures, because a development process should be monitored during all phases of development and not only be reactive if the process has been more or less successfully finished.

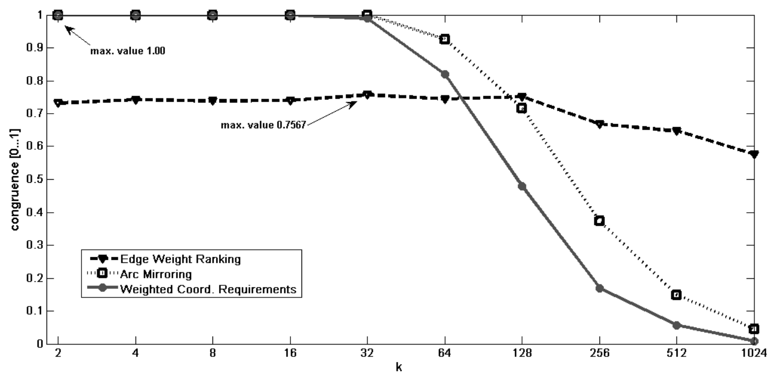


Figure 5.1: Congruence values for Eclipse code-inferred and socially-inferred dependency network $C_{E\_all} \cong S_{E\_k}$, where *all* denote the combination of static and dynamic code dependencies and k denotes a threshold ($2 \leq k \geq 1024$) for discarding network edges of weight $< k$ in the Jazz developer×component network. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1] (see chapter 4.3)

Therefore an approach was to study the impact of reducing the data volume of the extracted two-mode developer×component networks before transforming into a one-mode projection, see chapter 3, which means that edges with a lower weight as defined with threshold k were eliminated in every step. This was conducted as simulation for a decrementing access of network data. To get a better insight, figure 5.1 shows the evolution of congruence values with reduced data volume through threshold k with $k = 2^1$ to 2^{10}. The generated code-inferred network $C_{\text{E\_all}}$ and the socially-inferred network S_{E} of the Eclipse data set were applied to the three algorithms through different $k's$. Figure 5.1 points out the different trends of the algorithm's response. The *EdgeWeightRanking* algorithm coped with less diversifications and the congruence values remained relatively stable over k. Interesting in this case are the different moments of reaching congruence maxima. While *ArcMirroring* and *WeightedCoordRequirements* reach their maxima still at $k = 2^1$, the *EdgeWeightRanking* obtain the maxima value at $k = 2^5$.

Figure 5.2 depicts that the congruence values of the *Jazz* networks, ranging from 0.12 to 0.48, are lower compared to the Eclipse networks in figure 5.1, which range from 0.73 to 1. This results in the different software development phases of the software architectures. While the Eclipse networks are located in the maintenance phase, the Jazz dependency networks are build during the implementation phase of the first release. However, figure 5.2 also shows that the *EdgeWeightRanking* algorithm is relatively stable over different k's, which implied that *EdgeWeightRanking* would be tough on data reducing and could also be applied to smaller development projects with a less amount of network data.

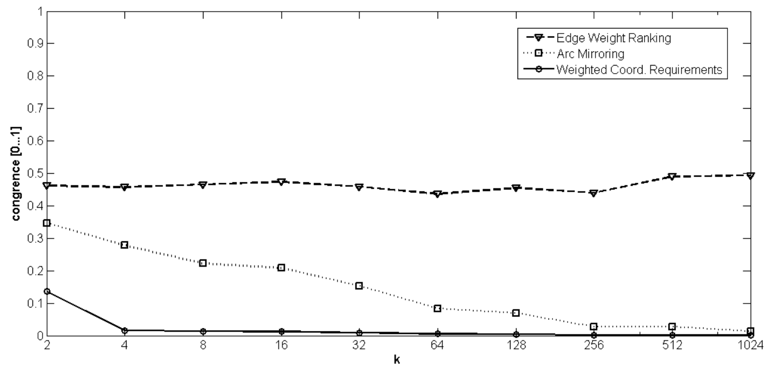


Figure 5.2: Congruence values for Jazz code-inferred and socially-inferred dependency network $C_{J\_all} \cong S_{J\_k}$, where *all* denote the combination of static and dynamic code dependencies and k denotes a threshold ($2 \leq k \geq 1024$) for discarding network edges of weight $< k$ in the Jazz developer$\times$component network. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1] (see chapter 4.3)

5.2 Evolution of Congruence over Time

Software development is a very dynamic and complex process and this assumes that socio-technical congruence would also be dynamic regarded over time. Depending on the maturity of the software architecture and the organizational structure influenced by the communication and collaboration behavior of development teams, the level of congruence would grow over time.

To investigate the trend of congruence over time, the available data sets were split up in monthly time stamps observing the trends within the given time period. Figure 5.3 displays the congruence evolution of the Eclipse code- and socially-inferred dependency networks from October 2002 to August 2007. The different times needed for adjusting to the maxima congruence values was interesting to note. While *EdgeWeightRanking* needed less time to level at the maxima congruence value, *ArcMirroring* and *WeightedCoordinationRequirements* required a longer period and the congruence value ranges, against expectations, different. The congruence value range over time of the *ArcMirroring* and *WeightedCoordinationRequirements* con-

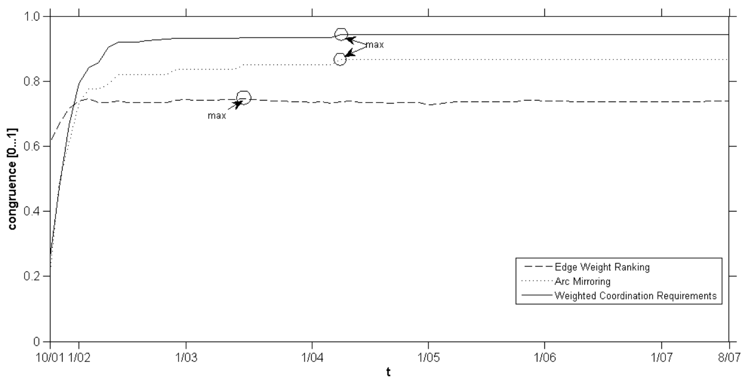


Figure 5.3: Progress of congruence over time for Eclipse code- and socially-inferred dependency network $C_{\mathrm{E\_all}} \cong S_{\mathrm{E}}$, where *all* denote the combination of static and dynamic code dependencies. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1], see chapter 4.3

gruence was from 0.25 to 0.94 which is significantly greater than the congruence value range of *EdgeWeightRanking* from 0.63 to 0.74. After reaching the maxima values, all congruence measures were relatively stable till the end of the period.

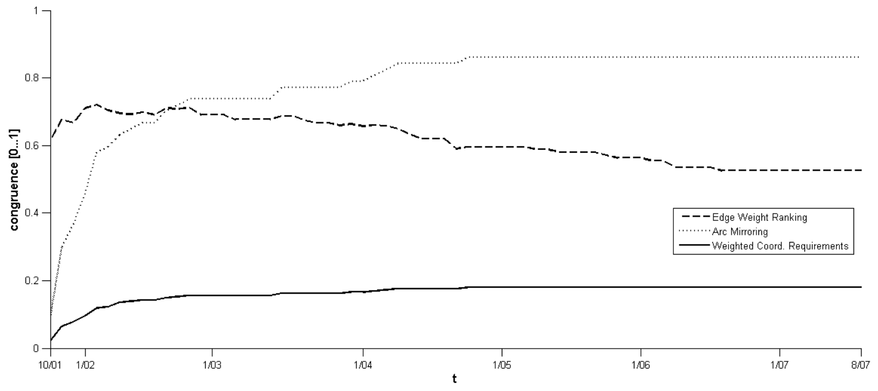


Figure 5.4: Impact on congruence evolution by edge destruction for Eclipse code- and socially-inferred dependency network $C_{\mathrm{E\_all}} \cong S_{\mathrm{E\_t=20}}$, where *all* denotes the combination of static and dynamic code dependencies and t denotes a threshold for discarding edges with an edge weight $<$ 20 in the socially-inferred dependency network.

A further experiment was observing the characteristics of congruence manipulating the derived one-mode component×component socially-inferred dependency networks by eliminating low weighted edges by threshold t. Therefore, figure 5.4 shows the impact by eliminating edges with edge weights lower $t = 20$. This approach depicted that only the *WeightedCoordinationRequirements* algorithm was strongly influenced, which was expected because this algorithm is based on edge weights. Interesting was also the fact that both *EdgeWeightRanking* and *ArcMirroring* were not significantly affected, which was verified by calculating the correlation of congruence measures with and without edge destruction shown in figure 5.5.

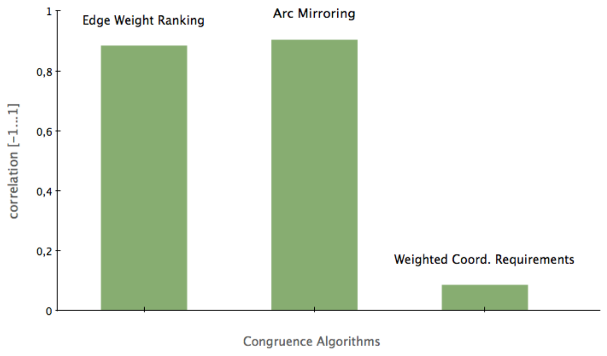


Figure 5.5: Correlation of the three congruence measures, applied to Eclipse code- and socially-inferred dependency networks $C_{\mathrm{E\_all}} \cong S_{\mathrm{E}}$ with Eclipse code- and socially-inferred dependency networks $C_{\mathrm{E\_all}} \cong S_{\mathrm{E\_t=20}}$, where *all* denotes the combination of static and dynamic code dependencies and t denotes a threshold for discarding edges with an edge weight < 20 in the socially-inferred dependency network.

In figure 5.6, the evolution of the Jazz dependency networks is shown in a monthly period from November 2006 to March 2008. The characteristics of the *EdgeWeightRanking* algorithm was similar to them in figure 5.3 and *ArcMirroring* and *WeightedCoordinationRequirements* depicted a linear growth. Another interesting issue was the observation that the congruence values of *ArcMirroring* and *WeightedCoordinationRequirements* still increased over time, but *EdgeWeightRanking* indicated a continuous, but gently drop of the congruence level during the time period.

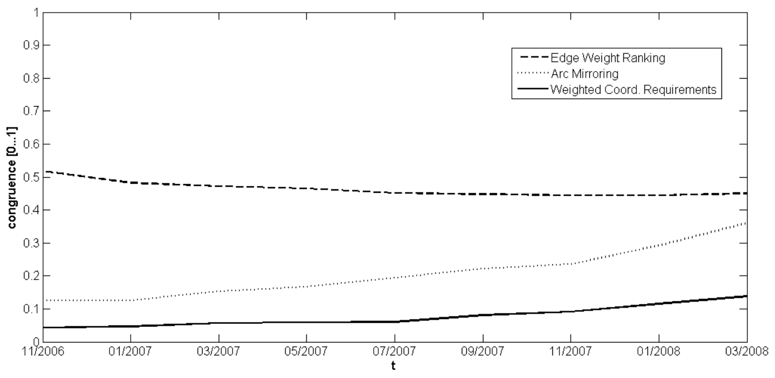


Figure 5.6: Progress of congruence over time for Eclipse code- and socially-inferred dependency network $C_{\mathrm{J\_all}} \cong S_{\mathrm{J}}$, where *all* denote the combination of static and dynamic code dependencies. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1] (see chapter 4.3)

5.3 Segmentation of Static and Dynamic Code Dependencies

The technical architecture of software could be modeled as a network of software components related through code dependencies. These code dependencies are dividable into static and dynamic dependencies. So far, in previous sections both static and dynamic dependencies were considered.

Related to [Wermelinger and Yu, 2008], static dependency means that two software components depend statically if the compilation of the first component requires the second, and dynamic dependency states the addiction of two software components if the first component provided an extension point at run-time which is used by the second component. Concerned with socio-technical congruence, an interesting step was to split up the code-inferred networks in dependency networks with only static or dynamic dependencies.

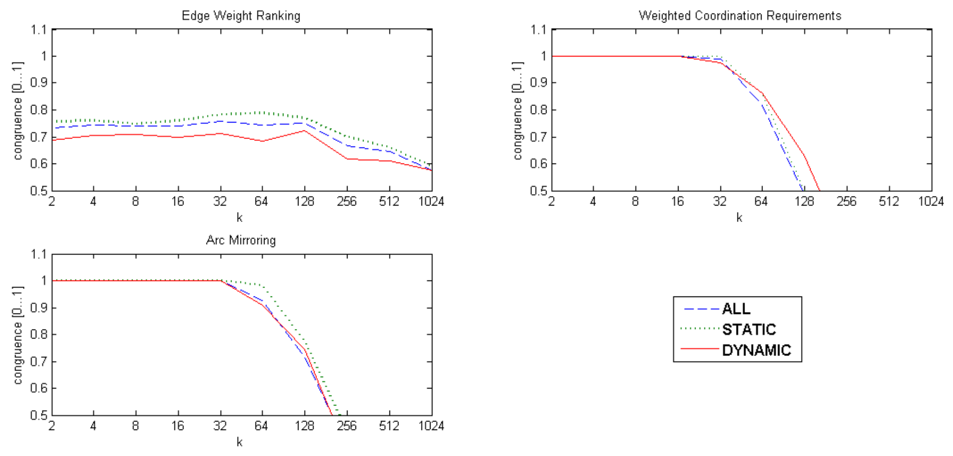


Figure 5.7: Division of static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies for congruence calculations applied to the Eclipse code- and socially-inferred dependency network $C_{\mathrm{E\_all|dyn|stat}} \cong S_{\mathrm{E}}$, where k denotes a threshold ($2 \leq k \geq 1024$) for discarding network edges of weight $< k$ in the Jazz developer$\times$component network.

In figure 5.7, the same approach was used as illustrated in figure 5.1, but the code-inferred Eclipse networks were divided into dynamic and static dependencies and each congruence algorithm was considered separately. The same was applied to the code-inferred Jazz dependency networks which is shown in figure 5.8. An interesting observation was that different congruence measures, specially within the Jazz networks, acted divergently whereas the congruence performance was rather convergent within the Eclipse networks.

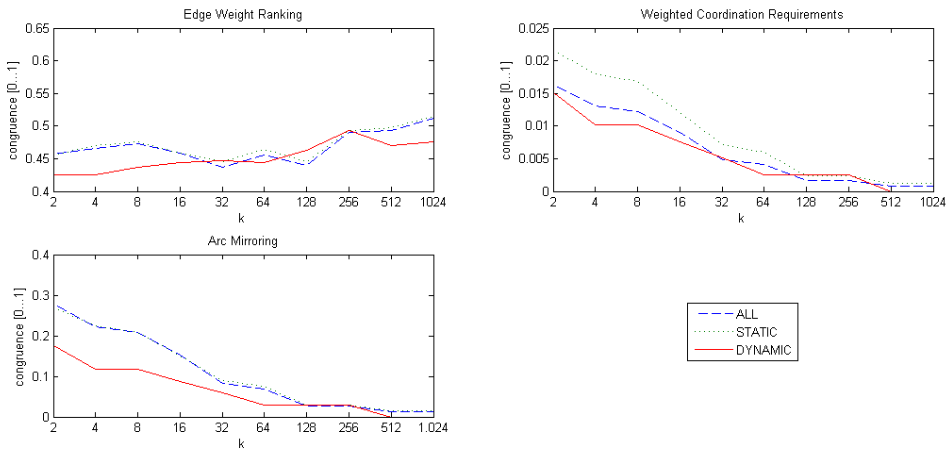
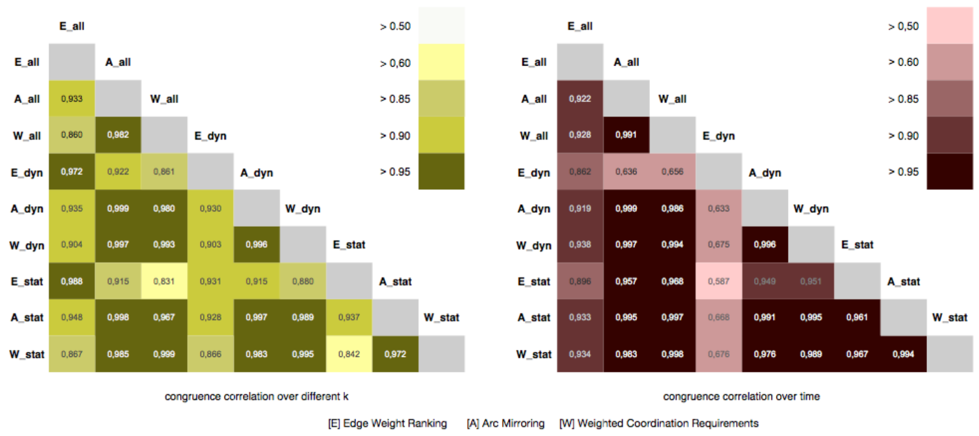


Figure 5.8: Division of static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies for congruence calculations applied to the Jazz code- and socially-inferred dependency network $C_{\mathrm{J\_all|dyn|stat}} \cong S_{\mathrm{J}}$, where k denotes a threshold ($2 \leq k \geq 1024$) for discarding network edges of weight $< k$ in the Jazz developer$\times$component network.

5.4 Correlation of Congruence Algorithm Measures

In the previous sections, the different characteristics of the congruence algorithms were determined. In the next step, the correlations between the three algorithms distinguishing also static and dynamic code dependencies were calculated. The heat map in figure 5.9 shows both the congruence value correlations over different k and the correlations between congruence values calculated over time for the Eclipse social- and code-inferred dependency network congruence.



congruence correlation over different k

	E_all	A_all	W_all	E_dyn	A_dyn	W_dyn	E_stat	A_stat	W_stat
E_all									
A_all	0,933								
W_all	0,860	0,982							
E_dyn	0,972	0,922	0,861						
A_dyn	0,935	0,999	0,980	0,930					
W_dyn	0,904	0,997	0,993	0,903	0,996				
E_stat	0,988	0,915	0,831	0,931	0,915	0,880			
A_stat	0,948	0,998	0,967	0,928	0,997	0,989	0,937		
W_stat	0,867	0,985	0,999	0,866	0,983	0,995	0,842	0,972	

> 0.50	⬜
> 0,60	🟨
> 0.85	🟨
> 0.90	🟨
> 0.95	🟫

congruence correlation over time

	E_all	A_all	W_all	E_dyn	A_dyn	W_dyn	E_stat	A_stat	W_stat
E_all									
A_all	0,922								
W_all	0,928	0,991							
E_dyn	0,862	0,636	0,656						
A_dyn	0,919	0,999	0,986	0,633					
W_dyn	0,938	0,997	0,994	0,675	0,996				
E_stat	0,896	0,957	0,968	0,587	0,949	0,951			
A_stat	0,933	0,995	0,997	0,668	0,991	0,995	0,961		
W_stat	0,934	0,983	0,998	0,676	0,976	0,989	0,967	0,994	

> 0,50	🟥
> 0.60	🟥
> 0.85	🟫
> 0.90	🟫
> 0.95	⬛

[E] Edge Weight Ranking [A] Arc Mirroring [W] Weighted Coordination Requirements

Figure 5.9: Correlations between congruence measures for Eclipse socio-technical dependency networks over different k, see chapter 5.1 and correlations between congruence measure for Eclipse socio-technical dependency networks over time, see chapter 5.2, where E denotes *EdgeWeightRanking* and A denotes *ArcMirroring*, W denotes *WeightedCoordinationRequirements* and *stat*, *dyn*, *all* denote static, dynamic and combined code dependencies.

An interesting observation was that in both correlation heat maps, the congruence values using the *EdgeWeightRanking* algorithm with dynamic code dependencies ($E_{\_dyn}$) correlated the least, which indicates that dynamic dependencies are less correlated.

The fact that the correlations between the different congruence measures are at a rather high correlation level, indicates that all three congruence approaches measure a similar socio-technical relation within a software project.

5.5 Identification of Quality- and Success Factors Effecting Congruence

One of the major goals of this thesis is the identification of quality and success factors that correlate with the congruence values of social and technical structures covered in web-based software project repositories. In literature a huge range of software quality and quantity measures exist, see chapter 5.5. For this purpose, the focus is not set only on code quality but on a combination with social network topology measures such as in [Crowston et al., 2004].

Factors	**Description**
(1) bugs	Incidence of bugs or feature requests reported in the back tracking or work item system within a defined period.
(2) developers	Amount of developers traced in the bug tracking or work item system as bug reporter, assigned as bug fixer or participant in discussions.
(3) clustering coef. (*cc.*)	Network average clustering coefficient, describes the local average connectivity of neighborhood nodes. [Watts and Strogatz, 1998]
(4) comp. size	Average size of network components, where size means the number of nodes contained within a particular network component and a component is defined as subnetwork, where any node is connected to any other node in the subnetwork via a path. [Anvik et al., 2006]
(5) largest comp. size	Number of nodes within the largest network component. [Anvik et al., 2006]
(6) num components	Amount of network components within the entire network.
(7) developers/component	Indicates the ratio of developers working on a specific software component (team size).
(8) bugs/component	Indicates the amount of change concerning incidence of bugs per component (defect density) [Westfall, 2008].
(9) components/developer	Ratio number of components handled by a specific developer (multitasking).
(10) bugs/developer	The ratio amount of resolved or worked on bugs or feature requests per developer (level of activity) [Crowston et al., 2004].
(11) density (Δ)	Implies the degree of connection within the network by calculating the ratio of actually edges and possible edges. [Wasserman and Faust, 1994]
(12) diameter (D)	Describes the extensiveness of the network by calculating the geodesic path (longest shortest path) between all network nodes. [Wasserman and Faust, 1994]
(13) closeness (c_c)	Closeness Centrality states the average amount of steps to hit every other node in the network from a specific node. [Wasserman and Faust, 1994]
(14) betweenness (c_b)	Betweenness Centrality indicates the average frequency of node presence in all geodesic paths of the network. This reflects the importance of node v within a network by calculating the number of nodes node v is related indirectly. [Wasserman and Faust, 1994]
(15) *cliques*	Implies the number of cliques (fully connected sub-networks) within the network. [Wasserman and Faust, 1994]

Table 5.2: Overview software quality and success factors

To this end, a set of software project quality and success factors was defined, shown in table 5.2. These factors involve both software code quality measures like defect density and network-theoretical measures like clustering coefficient. For calculating correlations, the factor values for the different time stamps were correlated with the according congruence measures. The results of the correlations regarding the Eclipse congruence measures are displayed in figure 5.10. It illustrates the correlations of the three different congruence values with potential influence factors and gives an overview of which factors provide a highly positive or negative correlation and which were uncorrelated.

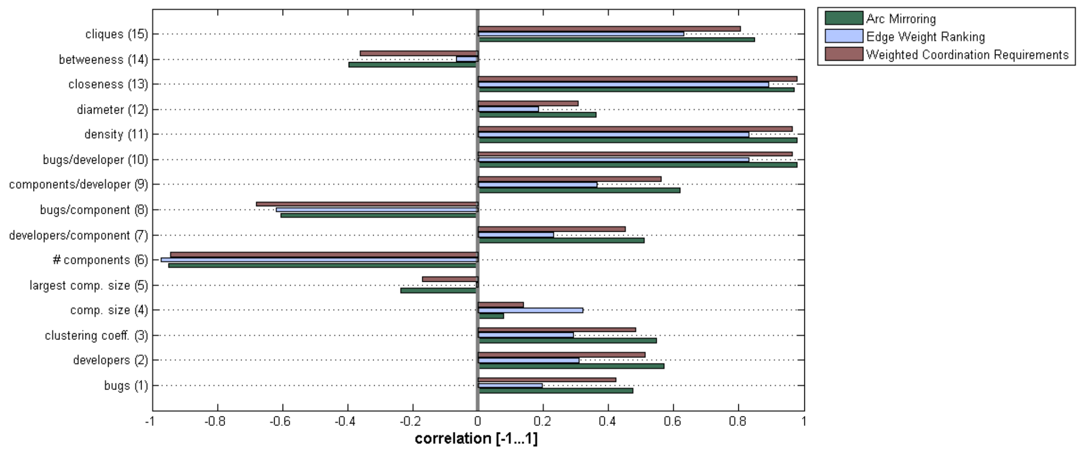


Figure 5.10: Eclipse: Correlations between quality factors and congruence values

Considering factors 11, 13 and 15 in figure 5.10 that show a rather high positive correlation, it appealed that especially factors describing the network topology were intensely related with the congruence measures. Specially factor 10 which represents the *level of activity* correlates significantly with the socio-technical congruence measures, because a higher activity level of developers may result in higher coordination and collaboration activities, that may benefit congruence. The correlation results concerning the Jazz networks are shown in figure 5.12.

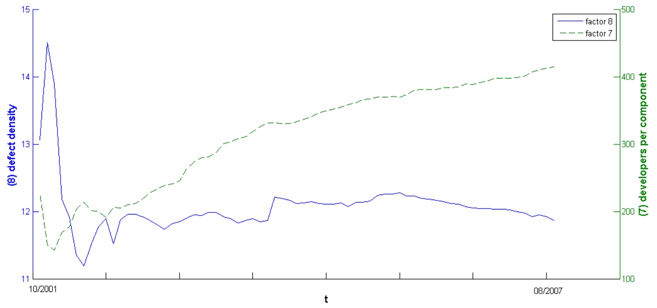


Figure 5.11: Divergence between the evolution of defect density (factor 8) and software component developer team size (factor 7) over time

Further, an interesting observation is the divergence between factor 7 and 8 which indicates that the lesser the defect density of software projects, the more developers are working on a specific component. [Brooks, 1995] states that a higher amount of developers working on a binary or a component requires a higher coordination effort and therefore the frequency of modification would increase. This implies that a higher divergence between defect density and the number of developers indicates a high level of socio-technical congruence. [Brooks, 1995] argues that for N developers, $N \times (N-1)/2$ possible diffusion paths persist for interchanging. Figure 5.11 illustrates this divergence for the Eclipse data set.

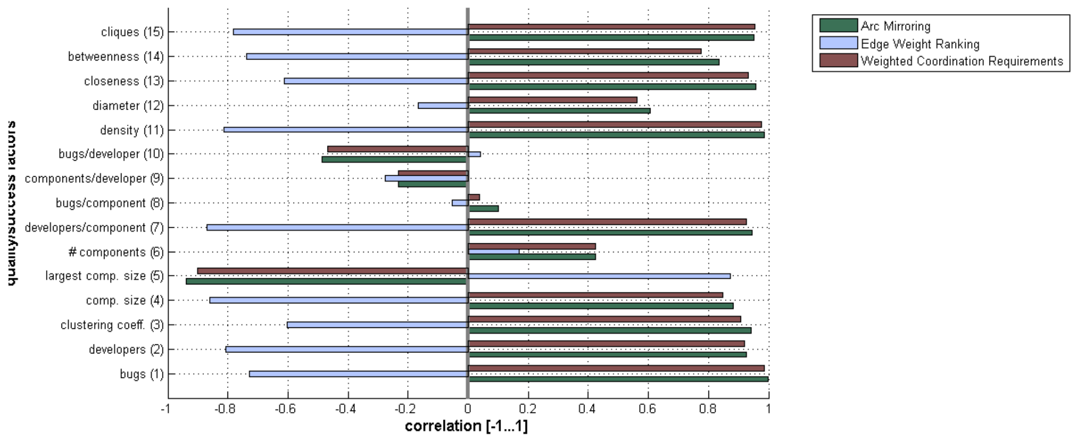
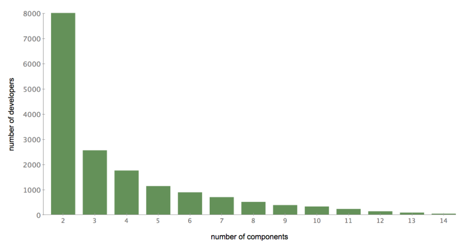


Figure 5.12: Jazz: Correlations between quality factors and congruence values

Moreover, the evolution of multi-tasking developers, factor 9, also indicates a relation with socio-technical congruence and would intensify [Brooks, 1995] expectations because of increasing coordination efforts. Figure 5.3 depicts the distribution of multi-tasking developers, developers working on more than one software component. An interesting observation was the detection of scale-free network characteristics, denoted by the power law distribution derived from the two-mode developer×component network.



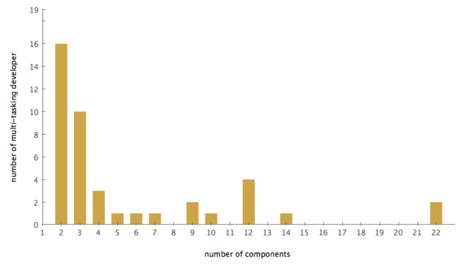


Table 5.3: Multi-tasking developer distribution of socially-inferred dependency networks mined from Eclipse (left) and Jazz (right)

The results further indicate that also the level of community maturity plays an important role, comparing for example the different clustering coefficient values of socially-inferred dependency networks derived from Eclipse and Jazz, see table 3.6, it also indicates that a higher level of connectivity of neighborhood nodes and higher amount of developer team size also promotes a lower defect density.

Chapter 6

Conclusion

This thesis deals with the network-theoretic analysis of socio-technical relations in software projects mined from web-supported software repositories. This chapter discusses the results of the empirical study in chapter 5, providing contributions and insights in trying to answer the research questions, motivated by Conway's Law [Conway, 1968], in chapter 1.2.

6.1 Contribution

The first examined issue for gathering a basis for congruence evaluation was the construction of socially- and code-inferred software component dependency networks derived from two real world data sets. The comparison of the generated socio-technical dependency networks showed that the dependency network pairs share similar network-theoretic topology attributes, which were evaluated through a social network analysis in chapter 2.1. The analysis also exposes that the Eclipse networks comprised properties of small-world networks, indicated through a high clustering coefficient and a relatively small network diameter. Additionally, a power law distribution of multi-tasking developers in both software projects indicated properties of a scale-free network.

The results of the empirical study applying different socio-technical congruence measures to two real world data sets were documented in chapter 5 and provide insights into the characteristics of the congruence measure levels and congruence evolution over time. The study has shown that the three selected congruence measures of [Strohmaier et al., 2009], [Valetto et al., 2007] and [Kwan et al., 2009] were practically useful and practicable due to their rather simple implementation and adequate resistance against data reduction.

However, it was observed that the results of the three congruence measure algorithms have to be interpreted differently because of varying basics and perspectives applying the congruence measure calculation. Resuming the correlations between the different congruence measures, a rather high correlation indicated that all three congruence approaches measure similar socio-technical relations within socially- and code-inferred dependency networks of software architectures.

Due to the dynamical nature of software development projects, the assumption was set up that socio-technical congruence will have a dynamical evolution of congruence over time. The results of the empirical study underline this conjecture and show a rather consistently increasing evolution of congruence over time, illustrated in chapter 5.2.

A further goal of this thesis was the identification of software quality or success factors, which influence the socio-technical congruence such as defect density. The insights into studying influence factors showed that both, software quality metrics and social network topology metrics such as closeness centrality, correlate with socio-technical congruence of the two software projects, see chapter 5.5.

An interesting observation was a divergence property of *defect density* and the average *team size* per software component, which indicates that a larger development team benefits a lower defect density. Another high correlated factor is given by the *level of activity* represented by the average number of bug reports worked on by a specific developer.

Concluding, it could be said that referring to Conway's Law conjecture of socio-technical relation, it could be found some evidence that social structures correlate with technical structures. Nevertheless, the causal relationship between those needs to be investigated in future work.

6.2 Outlook

6.2.1 Future Work

In the future, an empirical study of socio-technical congruence of different commercial software projects will be interesting, comparing the results with those of this thesis, which deals with Open Source software projects.

Additionally, the influence of different software development methodologies such as agile software development could be investigated, observing whether different software development methodology benefits socio-technical congruence.

6.2.2 Improvements

The generation of the socially-inferred dependency networks succeeds an approach of relating developers to software components through co-occurrence in bug reports. The possibility of a finer grained level of distinguishing between different developer roles would potentially improve the generation of more accurate dependency networks.

Additionally, the existing socio-technical congruence measures provide partially the possibility of local and priority focused congruence measures. A new approach regarding measuring congruence could be the calculation of *gap compensators* noticed by [Crowston and Howison, 2005], developers whose communication behavior conceals coordination gaps and decreases the probability of project success if the specific developer leaves the development team. Therefore, also the concerning of so called weak ties or informal communication or collaboration process paths could improve the usefulness of socio-technical congruence measures.

List of Figures

1.1 Conway's Law 3
1.2 Thesis process chart 6

2.1 Dyadic isomorphism states [Wasserman and Faust, 1994] 8
2.2 Forbidden Triad by [Granovetter, 1973] 9
2.3 A two-mode network (middle) with according $\top$ (left) and $\bot$-projections (right) [Latapy et al., 2008] 9
2.4 Comparison of real world networks and random networks concerning small-world characteristics by [Newman, 2002] 12
2.5 Power Law denoted degree distributions of real world networks [Newman, 2002] 14
2.6 Descriptive data of development community properties [Xu et al., 2006] 14
2.7 An example network and including cliques [Wasserman and Faust, 1994] 16
2.8 Mapping between a social developer network (top) and the according network of software modules (bottom) [Amrit et al., 2004] 17
2.9 Release defect density evolution [Westfall, 2008] 20
2.10 Results of the organizational failure prediction model by [Nagappan et al., 2008] compared to code-based failure prediction models 21
2.11 Descriptive data and correlations among quality measures [Crowston et al., 2004] 22
2.12 Using conceptual and ownership architecture as predictor of concrete software architecture [Bowman and Holt, 1998] 23
2.13 Augur visualization tool [Souza et al., 2005] 24
2.14 Metamodel for plugin-based software architectures [Wermelinger and Yu, 2008] 25
2.15 Eclipse bug report life cycle [Anvik et al., 2006] 26
2.16 Excerpt of SourceForge database schema [Xu et al., 2006] 27
2.17 Interaction network mined from a SourceForge project [Crowston and Howison, 2005] 28
2.18 Data overview extracted from mailing lists [Bird et al., 2008] 30

3.1 Generated Eclipse socially-inferred dependency network S_{E} 36
3.2 Generated Eclipse code-inferred dependency network C_{E} 37
3.3 Generated Jazz socially-inferred dependency network S_{J} 38
3.4 Generated Jazz code-inferred dependency network C_{J} 39

4.1 A socio-technical software network [Valetto et al., 2007] where G_P represents a people network and G_S a network of software artifacts 43

4.2 ArcMirroring of code and socially-inferred dependency networks . . . 43

4.3 Weighted task assignments and dependencies 45

4.4 Weighted code and socially-inferred dependency networks 46

4.5 Edge weight ranked socially- and code-inferred dependency networks 47

5.1 Congruence values for Eclipse code-inferred and socially-inferred dependency network $C_{E\_all} \cong S_{E\_k}$, where *all* denote the combination of static and dynamic code dependencies and k denotes a threshold $(2 \leq k \geq 1024)$ for discarding network edges of weight $< k$ in the Jazz developer×component network. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1] (see chapter 4.3) 50

5.2 Congruence values for Jazz code-inferred and socially-inferred dependency network $C_{J\_all} \cong S_{J\_k}$, where *all* denote the combination of static and dynamic code dependencies and k denotes a threshold $(2 \leq k \geq 1024)$ for discarding network edges of weight $< k$ in the Jazz developer×component network. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1] (see chapter 4.3) 51

5.3 Progress of congruence over time for Eclipse code- and socially-inferred dependency network $C_\mathrm{E\_all} \cong S_\mathrm{E}$, where *all* denote the combination of static and dynamic code dependencies. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1], see chapter 4.3 52

5.4 Impact on congruence evolution by edge destruction for Eclipse code- and socially-inferred dependency network $C_\mathrm{E\_all} \cong S_\mathrm{E\_t=20}$, where *all* denotes the combination of static and dynamic code dependencies and t denotes a threshold for discarding edges with an edge weight < 20 in the socially-inferred dependency network. 52

5.5 Correlation of the three congruence measures, applied to Eclipse code- and socially-inferred dependency networks $C_\mathrm{E\_all} \cong S_\mathrm{E}$ with Eclipse code- and socially-inferred dependency networks $C_\mathrm{E\_all} \cong S_\mathrm{E\_t=20}$, where *all* denotes the combination of static and dynamic code dependencies and t denotes a threshold for discarding edges with an edge weight < 20 in the socially-inferred dependency network. 53

5.6 Progress of congruence over time for Eclipse code- and socially-inferred dependency network $C_\mathrm{J\_all} \cong S_\mathrm{J}$, where *all* denote the combination of static and dynamic code dependencies. Please note, that the values of *EdgeWeightRanking* were transformed into a value range of [0...1] (see chapter 4.3) 53

5.7 Division of static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies for congruence calculations applied to the Eclipse code- and socially-inferred dependency network $C_{\mathrm{E\_all|dyn|stat}} \cong S_{\mathrm{E}}$, where k denotes a threshold ($2 \leq k \geq 1024$) for discarding network edges of weight $< k$ in the Jazz developer$\times$component network. 54
5.8 Division of static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies for congruence calculations applied to the Jazz code- and socially-inferred dependency network $C_{\mathrm{J\_all|dyn|stat}} \cong S_{\mathrm{J}}$, where k denotes a threshold ($2 \leq k \geq 1024$) for discarding network edges of weight $< k$ in the Jazz developer$\times$component network. 55
5.9 Correlations between congruence measures for Eclipse socio-technical dependency networks over different k, see chapter 5.1 and correlations between congruence measure for Eclipse socio-technical dependency networks over time, see chapter 5.2, where E denotes *EdgeWeightRanking* and A denotes *ArcMirroring*, W denotes *WeightedCoordinationRequirements* and *stat*, *dyn*, *all* denote static, dynamic and combined code dependencies. 55
5.10 Eclipse: Correlations between quality factors and congruence values 57
5.11 Divergence between the evolution of defect density (factor 8) and software component developer team size (factor 7) over time 57
5.12 Jazz: Correlations between quality factors and congruence values . 58

A.1 Python STC project class diagram 68
A.2 MATLAB class diagram 69
A.3 Graphic-centric code analyzing script collection 70

List of Tables

2.1 Adjacency matrix representation of the affiliation network shown in figure 2.3 and according $\top$-projection 10
2.2 Conceptual (left), ownership (middle) and concrete (right) system architectures of a Linux Kernel [Bowman and Holt, 1998] 22
2.3 Number of components for which a developer has resolved a bug report [Anvik et al., 2006] 26

3.1 Mined Eclipse bug report attributes 33
3.2 CSV format of mined bug report entries 33
3.3 Three-Mode network structure (left) and transitive relation (right) . 34
3.4 Two-mode (top) and according one-mode $\top$-projection network (bottom) with corresponding adjacency matrices 35
3.5 Excerpt of the extracted IBM Jazz RTC data set 37
3.6 Social network analysis measure results, where the columns comprise the Eclipse C_{E} and Jazz $\mathrm{C_J}$ code-inferred dependency networks, divided into static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies, where $\mathrm{S_E}$ and $\mathrm{S_J}$ denote the socially-inferred dependency networks. The rows contain topology measures, where n denotes the number of network nodes and *cc.* the clustering coefficient, Δ the density, D the network diameter, $\mathrm{c_c}$ closeness centrality and $\mathrm{c_b}$ betweenness centrality (for definitions see chapter 2.1.3) 40
3.7 Overview of degree distributions for Eclipse C_{E} and Jazz $\mathrm{C_J}$ code-inferred dependency networks, divided into static (*stat*), dynamic (*dyn*) and combined (*all*) code dependencies, and socially-inferred dependency networks $\mathrm{S_E}$ and $\mathrm{S_J}$. 41

4.1 Task Assignment and Task Dependency matrices 45
4.2 Edge weight ranking vectors of code- and socially-inferred software dependency networks 47

5.1 Congruence output values on different network pair set-ups, where $C_{\mathrm{E\_all}}$ and $C_{\mathrm{J\_all}}$ denote a code-inferred dependency network extracted from Eclipse (E) or Jazz (J), comprising combined static and dynamic code dependencies (*all*) and S_{E} and S_{J} denote a socially-inferred dependency network mined from Eclipse (E) or Jazz (J). N_{rand} and N_{zero} denote a random and a disconnected zero network. 49

5.2 Overview software quality and success factors 56
5.3 Multi-tasking developer distribution of socially-inferred dependency networks mined from Eclipse (left) and Jazz (right) 58

Appendix A

Implementation

This appendix gives a short overview of the developed frameworks including the implementation of the congruence measure algorithms and used resources and further detailed results of the social network analysis.

A.1 Matlab/Python Framework

Python was used in combination with the networkX[1] library package for creating an manipulating network structures. The class diagram illustrated in figure A.1 gives an overview of implemented classes and test classes. Basically the Python framework was used to parse XML files which were mostly not stored in a valid XML format and therefore an adapted own XML parser, class MyParser, was implemented. Moreover, the framework also handles data using CSV format for network construction and network outputs represented as adjacency matrices.

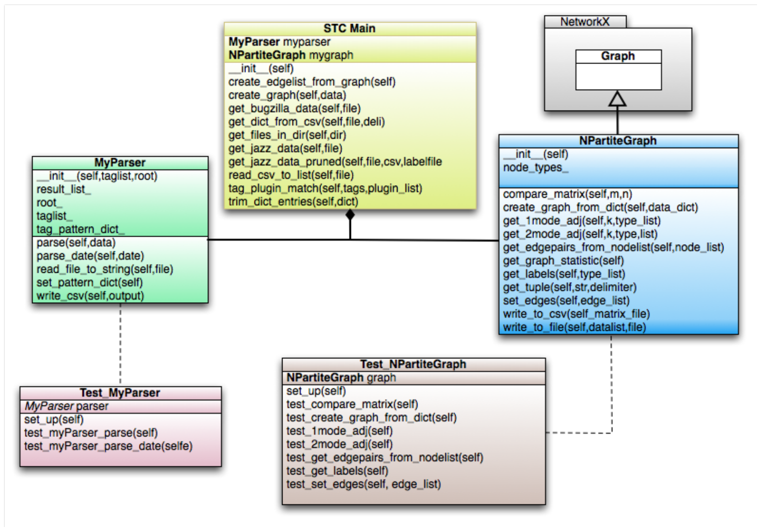

STC Main
MyParser myparser
NPartiteGraph mygraph
__init__(self)
create_edgelist_from_graph(self)
create_graph(self,data)
get_bugzilla_data(self,file)
get_dict_from_csv(self,file,deli)
get_files_in_dir(self,dir)
get_jazz_data(self,file)
get_jazz_data_pruned(self,file,csv,labelfile
read_csv_to_list(self,file)
tag_plugin_match(self,tags,plugin_list)
trim_dict_entries(self,dict)
NetworkX
Graph
MyParser
__init__(self,taglist,root)
result_list_
root_
taglist_
tag_pattern_dict_
parse(self,data)
parse_date(self,date)
read_file_to_string(self,file)
set_pattern_dict(self)
write_csv(self,output)
NPartiteGraph
__init__(self)
node_types_
compare_matrix(self,m,n)
create_graph_from_dict(self,data_dict)
get_1mode_adj(self,k,type_list)
get_2mode_adj(self,k,type,list)
get_edgepairs_from_nodelist(self,node_list)
get_graph_statistic(self)
get_labels(self,type_list)
get_tuple(self,str,delimiter)
set_edges(self,edge_list)
write_to_csv(self_matrix_file)
write_to_file(self,datalist,file)
Test_MyParser
MyParser parser
set_up(self)
test_myParser_parse(self)
test_myParser_parse_date(selfe)
Test_NPartiteGraph
NPartiteGraph graph
set_up(self)
test_compare_matrix(self)
test_create_graph_from_dict(self)
test_1mode_adj(self)
test_2mode_adj(self)
test_get_edgepairs_from_nodelist(self)
test_get_labels(self)
test_set_edges(self, edge_list)

Figure A.1: Python STC project class diagram

[1]http://networkx.lanl.gov/

The generated network matrices were used as input data for the MATLAB framework. MATLAB also provides object-oriented development and therefore a class *stcNetwork* was implemented comprising attributes and functions for calculating network topology measures. Further the three selected socio-technical congruence measure algorithms were implemented in class *congruence*. Using the MATLAB xUnit[2] test framework, the classes were tested using unit tests.

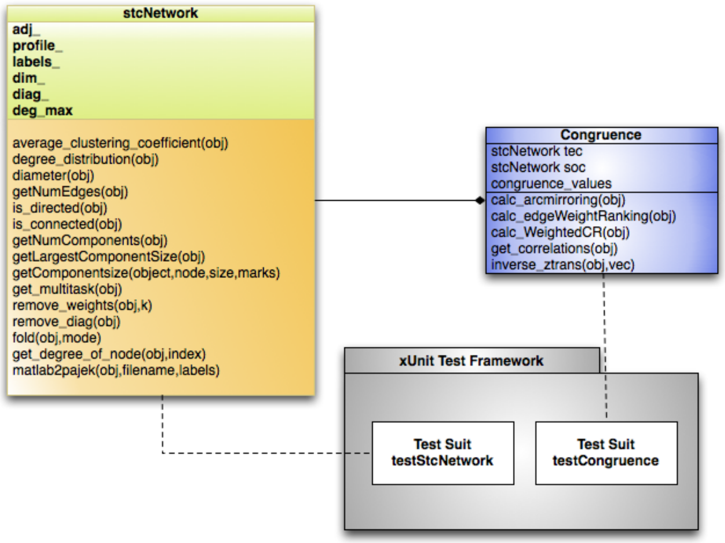


Figure A.2: MATLAB class diagram

A.2 Additional Tools

Pajek

An often used tool for representing and manipulating large scaled networks is Pajek[3]. In course of this thesis, Pajek was basically used for network visualization of the constructed dependency networks. Further it would be used for calculating social network topology metrics such as density, cliques and so on, which were not covered in the MATLAB framework [Nooy et al., 2005].

Tool Set for Code Analysis

Figure A.3 gives an overview of a graph-centric tool set for code analysis developed by [Wermelinger and Yu, 2008] for investigations on evolution and relations of software architectures.

[2]`http://www.mathworks.com/matlabcentral/fileexchange/22846`
[3]`http://vlado.fmf.uni-lj.si/pub/networks/pajek/`

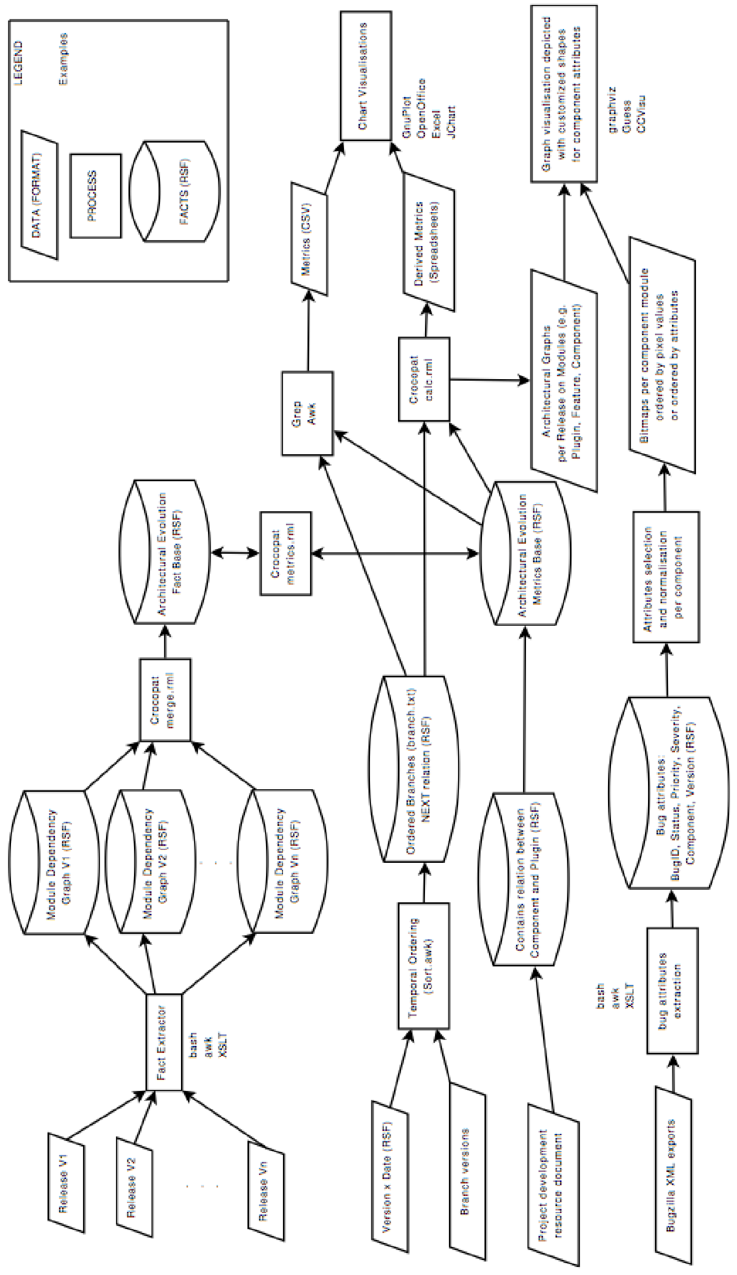

LEGEND
Examples
DATA (FORMAT)
PROCESS
FACTS (RSF)
Release V1
Release V2
. . .
Release Vn
Fact Extractor
bash
awk
XSLT
Module Dependency Graph V1 (RSF)
Module Dependency Graph V2 (RSF)
. . .
Module Dependency Graph Vn (RSF)
Crocopat merge.rml
Architectural Evolution Fact Base (RSF)
Crocopat metrics.rml
Version x Date (RSF)
Branch versions
Temporal Ordering (Sort,awk)
Ordered Branches (branch.txt) NEXT relation (RSF)
Project development resource document
Contains relation between Component and Plugin (RSF)
Architectural Evolution Metrics Base (RSF)
Grep Awk
Metrics (CSV)
Chart Visualisations
GnuPlot
OpenOffice
Excel
JChart
Crocopat calc.rml
Derived Metrics (Spreadsheets)
Architectural Graphs per Release on Modules (e.g. Plugin, Feature, Component)
Graph visualisation depicted with customized shapes for component attributes
graphviz
Guess
CCVisu
Bugzilla XML exports
bug attributes extraction
bash
awk
XSLT
Bug attributes: BugID, Status, Priority, Severity, Component, Version (RSF)
Attributes selection and normalisation per component
Bitmaps per component module ordered by pixel values or ordered by attributes

Figure A.3: Graphic-centric code analyzing script collection

Appendix B

Resources

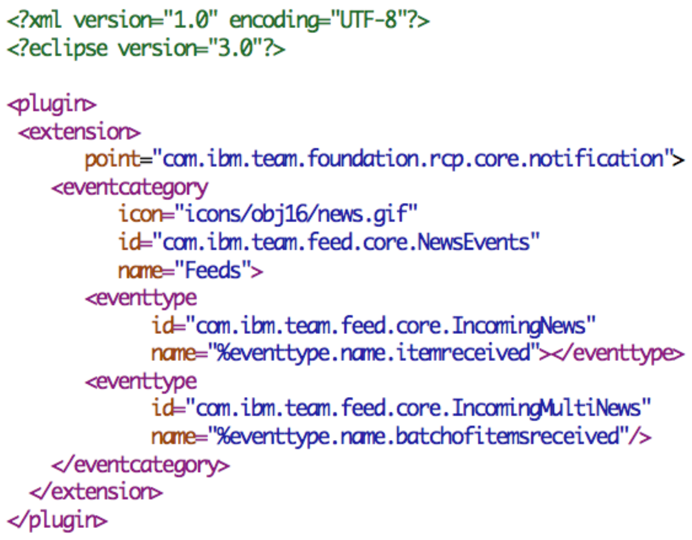
B.1 Excerpt of the Bugzilla Data Set

```xml
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE bugzilla SYSTEM "https://bugs.eclipse.org/bugs/bugzilla.dtd">
<bugzilla version="3.0.1" urlbase="https://bugs.eclipse.org/bugs/" maintainer="webmaster@eclipse.org">
    <bug>
          <bug_id>44901</bug_id>
          <creation_ts>2003-10-15 10:01 -0400</creation_ts>
          <short_desc>Memory leak on actions icons</short_desc>
          <delta_ts>2005-01-18 16:44:52 -0400</delta_ts>
          <reporter_accessible>1</reporter_accessible>
          <cclist_accessible>1</cclist_accessible>
          <classification_id>4</classification_id>
          <classification>Tools</classification>
          <product>Hyades</product>
          <component>UI</component>
          <version>1.0.2</version>
          <rep_platform>PC</rep_platform>
          <op_sys>Windows 2000</op_sys>
          <bug_status>CLOSED</bug_status>
          <resolution>FIXED</resolution>
          <priority>P3</priority>
          <bug_severity>normal</bug_severity>
          <target_milestone>3.0 M10</target_milestone>
          <everconfirmed>1</everconfirmed>
          <reporter name="Unknown User">unknown@eclipse.org</reporter>
          <assigned_to name="Valentina Popescu">popescu@ca.ibm.com</assigned_to>
          <long_desc isprivate="0">
            <who name="Unknown User">unknown6798@eclipse.org</who>
            <bug_when>2003-10-15 10:01:34 -0400</bug_when>
                <thetext>Memory leak on actions icons Driver used: win.daily.ad-5.11-20030916.zip
                         Using Sleak, icons used in toolbar and popup actions are not disposed probably
                         are shown.
                </thetext>
          </long_desc>
          <long_desc isprivate="0">
            <who name="Eugene Chan">ewchan@ca.ibm.com</who>
            <bug_when>2003-11-12 17:40:39 -0400</bug_when>
                <thetext>code dropped to cvs on Nov 12, 2003</thetext>
          </long_desc>
          <long_desc isprivate="0">
            <who name="Eugene Chan">ewchan@ca.ibm.com</who>
            <bug_when>2003-11-12 17:41:08 -0400</bug_when>
            <thetext>Valentina/Zahra, please mark this as fixed. thx</thetext>
          </long_desc>
    </bug>
```

B.2 Eclipse SDK Component List

EquinoxFramework
PlatformAnt
JDTCore
JDTDebug
JDTUI
PDEBuild
PDEUI
PlatformDoc
PlatformResources
PlatformSWT
PlatformSearch
PlatformTeam
PlatformText
PlatformUI
PlatformUpdate
PlatformUserAssistance

B.3 Example plugin.xml Metadata File IBM Jazz RTC Project



```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>

<plugin>
 <extension>
       point="com.ibm.team.foundation.rcp.core.notification">
    <eventcategory
          icon="icons/obj16/news.gif"
          id="com.ibm.team.feed.core.NewsEvents"
          name="Feeds">
       <eventtype
             id="com.ibm.team.feed.core.IncomingNews"
             name="%eventtype.name.itemreceived"></eventtype>
       <eventtype
             id="com.ibm.team.feed.core.IncomingMultiNews"
             name="%eventtype.name.batchofitemsreceived"/>
    </eventcategory>
  </extension>
</plugin>
```

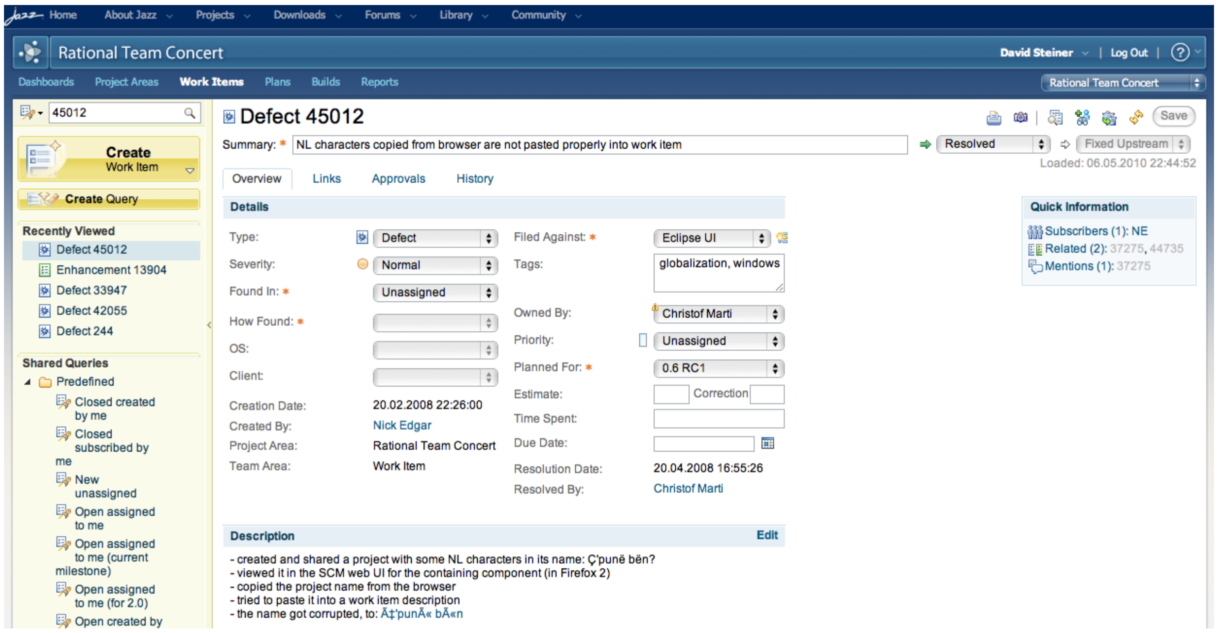
B.4 Example MANIFEST.MF Metadata File IBM Jazz RTC Project

```
Manifest-Version: 1.0
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Import-Package: com.ibm.team.repository.jdbcdriver.internal.db2,org.ap
ache.commons.logging
Bundle-SymbolicName: com.ibm.team.repository.jdbcdriver.db2z;singleton:=true
Bundle-ManifestVersion: 2
Bundle-Name: Repository component - DB2z JDBC driver
Bundle-Version: 0.6.0.I200805291754
Bundle-ClassPath: library.jar,external:$DB2Z_JDBC$/db2jcc_license_cisuz.jar,db2jcc.jar
Bundle-ActivationPolicy: lazy
Bundle-Vendor: IBM
Require-Bundle: com.ibm.team.repository.service,com.ibm.team.repository.common,
com.ibm.team.repository.jdbcdriver.db2
```

B.5 IBM Jazz RTC Project Matched Software Component Tag List

reports
scm
process
workitem
fulltext
feed
foundation
datawarehouse
connector
rational
performance
server
filesystem
build
dashboard
interop
repository
repotools
jazz
collaboration

B.6 Screenshot IBM Jazz Work Item GUI



45012

Recently Viewed
- Defect 45012
- Enhancement 13904
- Defect 33947
- Defect 42055
- Defect 244

Shared Queries
- Predefined
  - Closed created by me
  - Closed subscribed by me
  - New unassigned
  - Open assigned to me
  - Open assigned to me (current milestone)
  - Open assigned to me (for 2.0)
  - Open created by

Defect 45012

Summary: * NL characters copied from browser are not pasted properly into work item

Resolved | Fixed Upstream

Loaded: 06.05.2010 22:44:52

Details

Type:	Defect	Filed Against: *	Eclipse UI
Severity:	Normal	Tags:	globalization, windows
Found In: *	Unassigned		
How Found: *		Owned By:	Christof Marti
OS:		Priority:	Unassigned
Client:		Planned For: *	0.6 RC1
		Estimate:	Correction
Creation Date:	20.02.2008 22:26:00	Time Spent:	
Created By:	Nick Edgar	Due Date:	
Project Area:	Rational Team Concert		
Team Area:	Work Item	Resolution Date:	20.04.2008 16:55:26
		Resolved By:	Christof Marti

Quick Information
- Subscribers (1): NE
- Related (2): 37275, 44735
- Mentions (1): 37275

Description Edit

- created and shared a project with some NL characters in its name: Ç'punë bën?
- viewed it in the SCM web UI for the containing component (in Firefox 2)
- copied the project name from the browser
- tried to paste it into a work item description
- the name got corrupted, to: Ã‡'punÃ« bÃ«n

Bibliography

[Agrawal et al., 2003] Agrawal, R., Rajagopalan, S., and Srikant, R. (2003). Mining newsgroups using networks arising from social behavior. *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 529–535. Available from: `http://portal.acm.org/citation.cfm?id=775227`.

[Albert and Barabási, 2002] Albert, R. and Barabási, A. (2002). Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97.

[Amrit et al., 2004] Amrit, C., Hillegersberg, J., and Kumar, K. (2004). A social network perspective of conway's law. *Proceedings of the CSCW Workshop on Social Networks.*

[Anvik et al., 2006] Anvik, J., Hiew, L., and Murphy, G. (2006). Who should fix this bug? *Proceedings of the 28th International Conference on Software Engineering (ICSE '06).* Available from: `http://portal.acm.org/citation.cfm?id=1134285.1134336`.

[Barabasi and Bonabeau, 2003] Barabasi, A. and Bonabeau, E. (2003). Scale-free networks. *Scientific American*, 289(5):60.

[Bird et al., 2008] Bird, C., Pattison, D., D'Souza, R., and Filkov, V. (2008). Latent social structure in open source projects. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* Available from: `http://portal.acm.org/citation.cfm?id=1453101.1453107`.

[Bowman and Holt, 1998] Bowman, I. and Holt, R. (1998). Software architecture recovery using conway's law. *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '98).* Available from: `http://portal.acm.org/citation.cfm?id=783160.783166`.

[Brooks, 1995] Brooks, F. (1995). *The Mythical Man Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley.

[Byrne, 1991] Byrne, E. (1991). Software reverse engineering: a case study. *Software-Practice and Experience*, 21(12):1349–1364.

[Cao et al., 2008] Cao, Y., Petrushyna, Z., Anna, G., and Ralf, K. (2008). Getting to "know" people on the web 2.0. *Proceedings of International Conference on Knowledge Management and Knowledge Technologies (I-Know '08).*

[Cataldo et al., 2009] Cataldo, M., Easterbrook, S., Damian, D., Herbsleb, J., Devanbu, P., and Mockus, A. (2009). 2nd international workshop on socio-technical congruence. *Proceedings of the 31st International Conference on Software Engineering (ICSE '09 COMPANION)*. Available from: `http://portal.acm.org/citation.cfm?id=1585694.1586623`.

[Cataldo et al., 2008] Cataldo, M., Herbsleb, J., and Carley, K. (2008). Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. *Proceedings of the 2nd ACM-IEEE international symposium on Empirical Software Engineering and Measurement (ESEM '08)*. Available from: `http://portal.acm.org/citation.cfm?id=1414004.1414008`.

[Cataldo et al., 2006] Cataldo, M., Wagstrom, P., Herbsleb, J., and Carley, K. (2006). Identification of coordination requirements: implications for the design of collaboration and awareness tools. *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW '06)*. Available from: `http://portal.acm.org/citation.cfm?id=1180875.1180929`.

[Conway, 1968] Conway, M. (1968). How do committees invent? *Datamation*, 14(4):28–31. Available from: `http://www.melconway.com/research/committees.html`.

[Crowston et al., 2004] Crowston, K., Annabi, H., Howison, J., and Masango, C. (2004). Towards a portfolio of floss project success measures. *Collaboration, Conflict and Control: the 4th Workhop on Open Source Software Engineering. International Conference on Software Engineering (ICSE 2004)*, page 29.

[Crowston and Howison, 2005] Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. *First Monday*, 10(2):1–100.

[Curtis et al., 1988] Curtis, B., Krasner, H., and Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31(11). Available from: `http://portal.acm.org/citation.cfm?id=50087.50089`.

[Datta et al., 2010] Datta, S., Kaulgud, V., Sharma, V., and Kumar, N. (2010). A social network based study of software team dynamics. *Proceedings of the 3rd India Software Engineering Conference*, pages 33–42.

[Degenne and Forsé, 2004] Degenne, A. and Forsé, M. (2004). *Introducing Social Networks*. Sage Publications Ltd, 1 edition.

[Diestel, 2005] Diestel, R. (2005). *Graph Theory*. Springer-Verlag Heidelberg, 3 edition.

[Erdös and Rényi, 1960] Erdös, P. and Rényi, A. (1960). On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*.

[Fonseca et al., 2006] Fonseca, S., de Souza, C., and Redmiles, D. (2006). Exploring the relationship between dependencies and coordination to support global software development projects. *Proceedings of the International Conference on Global Software Engineerring (ICGSE '06)*, pages 243 – 243. Available from: `http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4031768&isnumber=4031726&punumber=4031725&k2dockey=4031768@ieeecnfs`.

[Granovetter, 1973] Granovetter, M. (1973). The strength of weak ties. *The American Journal of Sociology*, 78(6):1360.

[Graves et al., 2000] Graves, T., Karr, A., Marron, J., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software*, 27(7):653–661.

[Grinter, 2003] Grinter, R. (2003). Recomposition: Coordinating a web of software dependencies. *Computer Supported Cooperative Work*, 12(3):297–327. Available from: `http://www.springerlink.com/index/l486j81k188kk458.pdf`.

[Hein et al., 2006] Hein, O., Schwind, M., and König, W. (2006). Scale-free networks - the impact of fat tailed degree distribution on diffusion and communication processes. *Wirtschaftsinformatik*, 48(4):267–275.

[Herbsleb and Mockus, 2003] Herbsleb, J. and Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):1–14.

[Herbsleb and Grinter, 1999] Herbsleb, J. D. and Grinter, R. E. (1999). Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 16(5):63–70. Available from: `http://www2.computer.org/portal/web/csdl/doi/10.1109/52.795103`.

[Kähler, 2004] Kähler, W. (2004). *Statistische Datenanalyse: Verfahren verstehen und mit SPSS gekonnt einsetzen.* Vieweg Verlag, 3 edition.

[Kleinberg, 2000] Kleinberg, J. (2000). The small-world phenomenon: an algorithm perspective. *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, page 170.

[Kleinfeld, 2002] Kleinfeld, J. (2002). Could it be a big world after all? the 'six degrees of separation' myth. *Society, April*, 39(2):61–66.

[Kwan et al., 2009] Kwan, I., Schröter, A., and Damian, D. (2009). A weighted congruence measure. *Proceedings of the 31st International Conference on Software Engineering (ICSE'09) STC Workshop.*

[Latapy et al., 2008] Latapy, M., Magnien, C., and Vecchio, N. (2008). Basic notions for the analysis of large two-mode networks. *Social Networks*, 30(1):31–48.

[Linstead et al., 2007] Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., and Baldi, P. (2007). Mining eclipse developer contributions via author-topic models. *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR '07)*, page 30. Available from: `http://portal.acm.org/citation.cfm?id=1268983.1269044`.

[Milgram, 1967] Milgram, S. (1967). The small world problem. *Psychology Today*, 2:60 – 67.

[Nagappan et al., 2008] Nagappan, N., Murphy, B., and Basili, V. (2008). The influence of organizational structure on software quality: an empirical case study. *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 521–530. Notizen. Available from: `http://portal.acm.org/citation.cfm?id=1368088.1368160`.

[Neubauer and Obermayer, 2009] Neubauer, N. and Obermayer, K. (2009). Towards community detection in k-partite k-uniform hypergraphs. *Proceedings of th NIPS 2009 Workshop on Analyzing Networks and Learning with Graphs*. Available from: `http://snap.stanford.edu/nipsgraphs2009/papers/neubauer-paper.pdf`.

[Newman, 2002] Newman, M. (2002). Random graphs as models of networks. *Working Paper*. Available from: `http://arxiv.org/abs/cond--mat/0202208`.

[Nooy et al., 2005] Nooy, W. D., Mrvar, A., and Batagelj, V. (2005). Exploratory social network analysis with pajek. *Cambridge Press*. Available from: `http://en.scientificcommons.org/8505289`.

[Sarma et al., 2008] Sarma, A., Herbsleb, J., and Hoek, A. V. D. (2008). Challenges in measuring, understanding, and achieving social-technical congruence. *Workshop on Socio-Technical Congruence, Collocated with ICSE 2008,*, (CMU-ISR-08-105).

[Schröter et al., 2006] Schröter, A., Zimmermann, T., and Premraj, R. (2006). If your bug database could talk. *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18–20.

[Souza et al., 2005] Souza, C., Froehlich, J., and Dourish, P. (2005). Seeking the source: software source code as a social and technical artifact. *Proceedings of the 2005 international ACM SIGGROUP Conference on Supporting Group Work (GROUP '05)*, pages 197–206. Available from: `http://portal.acm.org/citation.cfm?id=1099203.1099239`.

[Strohmaier et al., 2009] Strohmaier, M., Wermelinger, M., and Yu, Y. (2009). Using network properties to study congruence of software dependencies and maintenance activities in eclipse. *Proceedings of the 31st International Conference on Software Engineering (ICSE'09) STC Workshop.*

[Travers and Milgram, 1969] Travers, J. and Milgram, S. (1969). An experimental study of the small world problem. *Sociometry*, 32(4):425–443.

[Treude and Storey, 2009] Treude, C. and Storey, M.-A. (2009). How tagging helps bridge the gap between social and technical aspects in software development. *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 12–22. Available from: `http://portal.acm.org/citation.cfm?id=1555001.1555018`.

[Valetto et al., 2007] Valetto, G., Helander, M., Ehrlich, K., Chulani, S., Wegman, M., and Williams, C. (2007). Using software repositories to investigate socio-technical congruence in development projects. *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR '07)*, page 25. Available from: `http://portal.acm.org/citation.cfm?id=1268983.1269039`.

[Wasserman and Faust, 1994] Wasserman, S. and Faust, K. (1994). *Social Network Analysis: Methods and Applications.* Cambridge University Press.

[Watts and Strogatz, 1998] Watts, D. and Strogatz, S. (1998). Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440–442.

[Wermelinger and Yu, 2008] Wermelinger, M. and Yu, Y. (2008). Analyzing the evolution of eclipse plugins. *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR'08)*, pages 133–136.

[Westfall, 2008] Westfall, L. (2008). *The Certified Software Quality Engineer Handbook.* Quality Press.

[Xu et al., 2006] Xu, J., Christley, S., and Madey, G. (2006). Application of social network analysis to the study of open source software. *The Economics of Open Source Software Development*, pages 205–224.

[Yu and Wermelinger, 2008] Yu, Y. and Wermelinger, M. (2008). Graph-centric tools for understanding the evolution and relationships of software structures. *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 329–330. Available from: `http://portal.acm.org/citation.cfm?id=1448010`.