Detecting Communities of Methods using Dynamic Analysis Data

Boyd Duffee and Peter Andras
School of Computing and Mathematics
Keele University
Keele, United Kingdom, ST5 5BG

***Abstract*—Maintaining large-scale software is difficult due to the size and variable nature of such software. Network analysis is a promising approach to extract useful knowledge from network representations of large and complex systems. Community detection is a network analysis method that aims to detect communities of nodes that share some common feature that is relevant for the whole system. We aim in this paper to investigate the usefulness of community detection for software maintenance considering networks of methods and method calls that represent execution traces of the analysed software. Our results show that the method communities that we extract are relatively persistent over multiple execution traces and that they are associated with functional features of the software. Our results also show that method communities are not associated with method level design features, but each method community has a specific distribution over method stereotypes.**

I. Introduction

Large-scale software is developed by many programmers over considerable time and includes many integrated components, which may be written by different teams at different times and in response to changing requirements [1], [2]. Such software is typically over 100 KLOC and often over 1MLOC in size — for example the Google Chrome browser has over 6 MLOC [3].

Maintaining such huge software systems is difficult because of the many interdependencies and the limited understanding by any developer of the whole software. This triggers the need for automated tools and computational techniques that can support the analysis large-scale software and improve the understanding of it by software developers [4]–[7].

Software systems developed in object oriented languages can be considered as networks of interactions where the interacting nodes can be classes or objects and the interactions the method calls between these [8], [9]. In a finer grained view the network nodes themselves are considered as being the methods of classes and the calls between the methods the interactions between the nodes [10], [11]. These networks can be considered in static sense [12], when the network is built by analysing the code of the software, or in a dynamic sense [12], when the network is built by considering the behaviour of the software at runtime (e.g. through some dynamic analysis instrumentation). Furthermore, another more social-network aspect of the large software can be captured by considering the joint work of developers in various teams [13], [14]. Given the success of network analysis methods applied to biological and social networks [15], [16], it may be expected that analysing the network representation of large-scale software may lead to useful insight that can help the understanding of these huge systems.

In general network analysis methods rely on the assumption that structurally important parts of the network representation of a system represent functionally important parts of the system. So network analysis applied to software systems is expected to reduce the complex search for functionally meaningful parts in a large-scale software in some functional context to the search for the structurally important parts of network representation of the software. Network analysis has been applied to develop novel metrics for large-scale software [3], [11]–[14], [17]–[19] and these metrics may be used to assess software quality, error proneness, or for functionality localisation [11]–[14], which are all useful for the understanding and maintenance of large-scale software systems.

While the success of network analysis methods in uncovering the nature of complex systems is celebrated in many fields [15], [16], often these methods are developed and tested using artificially generated surrogate data because there are relatively few real world large network data sets that are available widely. Thus in order to confirm the functional validity of these methods in any particular application area they should be tested with relevant real world data sets. In particular, in the context of engineering and maintenance of large-scale software these methods have to be applied to real world large-scale software data to establish to what extent they provide functionally useful analysis results.

An interesting area of network analysis methods is the detection of communities [20]–[22]. In the context of networks, communities are clusters of structurally related nodes that are associated also in some functional sense such that this functional aspect of the community contributes to the overall functionality of the system. For example, in the context of social networks communities may represent circles of friendship or groups of specialists. In the case of biological networks communities may represent proteins involved in functional cycles in cells or sets of cells that form key components of tissues. Naturally arises the question whether such community detection algorithms can help in any sense the functional understanding of large-scale software systems.

In this paper we investigate the usefulness of community detection in networks in the context of dynamic analysis data

gathered from large-scale software systems. We use a hard limit community detection algorithm that allocates each node to one community. We assess the extent to which communities detected in the network of methods and method calls have any associated functional or design related meaning (e.g. is it the case that a community has some well defined function — for example exporting data in a range of file formats, or is it the case that belonging to a community means the sharing of some design features — for example in terms of method stereotypes [7], [23]). Our analysis shows that the network communities that we find have associated functional meaning, but the methods belonging to a community do not share particular design features. We also find that some, but not all method communities that we detect are preserved across execution traces. The communities that are more preserved across traces are likely to deliver core functionalities of the software, while less preserved communities are likely to contribute to the delivery of trace-specific software features.

The rest of the paper is structured as follows. First we review briefly the related works, including the interpretation of software as a network, the application of network analysis to software systems, and the community detection algorithms in networks. Next we present the conceptual framework for the application of network community detection algorithms to software systems. Then we present our results and their interpretation. Next we discuss the implications of our analysis and results. Then we address the validity limitations of our approach and results. Finally the paper is closed by the conclusion section.

II. Related works

A. Software as network

Software has been considered as a graph or network since the beginning of research on computer programs. In the context of software developed in object oriented programming languages the network nature of software is obvious. One may consider such software as a network of classes, where the method calls between classes constitute the directed arcs (or undirected edges) of the network, the latter may also be labelled by the called method [9], [10]. Another approach considering the runtime instantiation of classes as objects views the software as a network of objects linked by method calls between objects, and again the calls may be labelled by the called method [9], [10]. A further approach is to consider the methods themselves as the nodes of the network and the arcs (or edges) of the network as the calls from one method to another [11], [12] — note that this approach does not emphasise the class-based grouping of the methods.

The network representation of a software system may be constructed on the basis of the code of the software [24], in which case the network is static network representation of the software. In static networks all possible links between classes or methods are considered according to the code of the software. The alternative is to consider the network representation of the software during runtime and to take into consideration only classes or objects that get instantiated and methods and method calls that get actually executed [8], [11], [12], [24]. For this purpose the software has to be instrumented such that the trace of execution in terms of method-to-method calls can be extracted [11], [12], [24]. The resulting network is a dynamic network representation of the software corresponding to an execution trace that may represent the delivery of some functionality of functional feature of the software. We note that dynamic analysis has been used recently to support program understanding, for example by the identification of concept locations [25]–[27].

A further take on the network aspect of large-scale software is to consider the collaboration network of software developers who contributed to the development of the software [13], [14], [17], [18]. This approach represents the human development environment of the software, assigning parts of the software to parts of this network, depending on the involved developers.

In general the network representation of the software or of its human development environment is expected to allow the use of structural analysis of the network representation in order to discover functionally important parts or features of the software. This expectation is based on the assumption that structurally important components of the network representation of a system indicate functionally important parts of the system that is represented by the network [11]. This assumption has been tested widely in biological and social networks and has been confirmed in many instances [15], [16].

B. Software network analysis

The earliest network metrics for software are based on graph-theoretic analysis of the graph representing the software, such as coupling measures [28], [29]. More recently through the consideration of network representation of large-scale software a number of other network-based metrics and measures have been considered [3], [11]–[14], [17]–[19]. These metrics and measures in general are based on structural analysis of the network and are associated with the functional parts of the software system which is represented by the analysed structural component.

For example, such network metrics are proposed to assess the error proneness of the software using the analysis of the software developer network and estimating the error proneness of software developed through the collaboration of different teams of software developers [13], [14], [18]. A similar approach can provide a classification of developers indicating likely developer roles which may be useful for setting up developer teams in the future [17]. Another approach uses network analysis to assess the level of vulnerability of methods in the context of delivery of functional features of the software [11].

Complexity metrics play an important role in the context of security and exposure to vulnerability [14]. Network analysis is can be used to assess the complexity of the software leading to network metrics of software complexity [3], [14]. One approach is to calculate a fractal dimension metric of the network representing the software to estimate the complexity of the software [30]. Researchers have shown that in many cases

the network complexity of the software grows as it undergoes further development [19]. Similar network complexity metrics may also be applied to software developer networks as well in order to assess the likelihood of developing security related problems [14].

Network analysis of dynamic networks can be used to develop metrics that indicate functional importance of methods for the delivery of a given functional feature [11]. This metric can used for feature localisation in the code. This can reduce very much the maintenance effort in the context of adapting large-scale software to changing functional requirements. Similarly network based metrics calculated using dynamic networks can be used to assess the match between the design and implementation of the software system and indicate problems with the quality of the software (i.e. mismatch between design and implementation) [12].

The above mentioned network metrics for large-scale software work reasonably well as reflected by the published papers that propose and analyse them. However, in principle, these network analysis methods can be expected to work to the extent to which software networks share their structural features with networks representing social and biological systems for which network analysis methods were proposed in the first place. Comparative analysis of static software networks and biological networks shows that although there is some match between these networks there are also considerable differences, which are attributed to the designed and engineered nature of the software system [31]. On the other side, dynamic networks representing software at runtime appears to be more similar in general structural terms to biological and social networks than static software networks [11]. Naturally, the application of network metrics to software developer networks is expected to work well, given that these are instances of social networks.

C. Network community analysis

Early work in social networks identified the relevance of communities to the information flow along the network through personal contacts [32]. Although no single definition of community is accepted in all areas of research [33], for simplicity it is defined in network studies in terms of its topological features as a subgraph with higher link density within the subgraph than external to it [34]. The assumption is that members of the community have properties in common. In a social context, the structural meaning of a community might signify the physical or organisational proximity, such as being neighbours, while the functional meaning would imply strong interpersonal connections, such as friendship. Protein interaction networks (PIN), with proteins as nodes and chemical interactions as links, can be also analysed in a meaningful way by using community detection algorithms [16]. Communities in a PIN are interpreted as *modules* corresponding to cellular functions that fulfil differing biological roles. It has been found that topological properties can correlate with functional homogeneity [35] and the network structure of these PINs allow cells to adapt to a changing environment.

Using the concept of link density to define community, many different algorithms have been developed to find clusters in the topology of the network. Community finding algorithms place nodes into either hard clusters where membership is of a binary nature or soft clusters where membership can belong to more than one cluster and is characterised by a degree of association. It has been shown that finding an exact solution for the best network partition based on modularity, a measure which evaluates the strength of clustering, is NP-complete (believed to be NP-hard) [36] and all such algorithms make accommodations in order to reduce the computational difficulty.

The Louvain algorithm performs hard clustering based on the Fast Greedy optimisation of modularity increases the value of modularity, Q, by combining small communities into larger ones, making it a very fast algorithm and has been shown to exceed other community finding algorithms in efficiency while maintaining good quality community detection [37]. An example of soft clustering is the Infomap algorithm. Rather than optimising modularity, it uses a random walker on the network to explore the flow between components. The "map equation" minimises the description length of the path of the random walker and each node is assigned a strength to which it is associated to each community [38]. A review that provides an extensive list of community finding algorithms with detailed descriptions and commentary was published by Fortunato [39].

While community finding and clustering are used almost interchangeably, if a difference is to be made, clustering is associated with the structural properties of the network down to the microscopic level whereas community tends to refer to the functional groupings that bind the members together.

Recent work in finding communities has focused on how close the clusters are to the "ground truth" of the actual communities in the network [40]. Benchmarks for algorithms are small, real networks e.g. Zachary's karate club [41] or large synthetic networks featuring clusters intentionally constructed as a part of the network synthesis. Large annotated networks have only recently become available and no community finding algorithm has been found to perform particularly well in extracting the annotated groups from the topological structure of large social networks such as Flickr or Orkut [40].

III. Network community algorithms for software analysis

Communities in networks are clusters of nodes that share their connectivity patterns, for example they are connected to similar sets of other nodes or they form relatively densely connected sub-networks [20]–[22]. Such communities can be identified in networks representing large-scale software systems. In the context of social or biological systems communities within network representations of the system often have an associated specific functionality in the context of the overall system, e.g. ethnic, professional and interest-driven communities within social systems. In principle, we may expect that communities in network representations of large-sale software also have some associated meaning in terms of

shared function or shared design, for example. However, it is not obvious to what extent this may hold actually true. The research question that we address in this paper is the following:

RQ: To what extent can we associate a functional or design meaning to communities of methods that can be determined from the network representation of large-scale software having as nodes the methods and edges the calls between methods?

From the perspective of software maintenance the actual running part of the code is more relevant than the parts of the code that are not practically used in the delivery of commonly used features and functionalities. For example, if requirements related to a feature or functionality are revised the code that may require maintenance intervention is the part of the code that delivers the relevant feature or functionality. Consequently, analysing the dynamic network representation of the running software with the aim to identify method communities is likely to be practically more useful than the analysis of the static network representation of the full code of the software. Given that the execution trace that delivers the chosen feature or functionality involves methods that are required for this purpose, it is more likely that the method communities that can be determined have some associated functional nature.

Considering that we analyse dynamic network representation of the software system that corresponds to an execution trace that delivers a feature or functionality (or some combination of these) in the first instance it is expected that if method communities have an associated meaning this is likely to be related to their functionality. For example, methods belonging to a community may belong to the same class, or may deal with the same kind of data, or may share functionality in some other sense. However, it is also possible that method communities get established according to shared design features [7], [12], [23]. For example, GET methods may share the pattern of connectivity to other methods on which basis they may form a community within the network. It is also possible that network communities may lift out some other common aspect of the methods belonging to a community, e.g. mapping on aspects of non-functional requirements.

Assuming that network communities of methods have some associated meaning that makes sense in the context of the software (e.g. design, functionality, etc.) it is expected that communities that are associated with functionalities or design features of central importance within the software are preserved across execution traces. Naturally, execution traces will vary in terms of the list of methods invoked within the trace, but there will be many methods shared between some traces. The expectation of preservation of communities with core importance means that the corresponding best matching communities determined for different traces will share a large number of methods (the majority of the methods within the community) across traces. The best match may be calculated using the Jaccard index [42], [43] or by using a comparison of the communities through some indicators of functionality or design features — for example by using the cosine similarity [42], [43] of vectors of values indicating the presence of functionality or design features among methods belonging to a community. If the level of preservation is low for a given community across two execution traces this may indicate that the respective communities in these traces relate to functionally orthogonal features of the software.

If there is a valid association of these communities with functional or design features (or some combination of these or other aspects of the software) — i.e. the meaning of the community in the context of the software — then the determination of network communities can be used for localisation of methods for which the meaning of the community is relevant. Depending on this meaning this may help the localisation of features or functionalities within the software, or the dominant design features, and so on. This in turn may support software maintenance (e.g. localisation of methods that need revision in response to changing requirements), design and implementation quality assessment (e.g. are the intended design features indeed the dominant ones for the methods), or other revision, assessment or validation of the software (e.g. the extent of matching non-functional requirements).

Hard membership algorithms have a very clear expression for belonging to a community. All boundary cases are placed in the group in which they share the most links, much like all methods belonging to only one class or a maximum likelihood of the method belonging within a group of methods. By restricting membership, the community labels gain improved clarity with their leading features. It makes it easier to interpret the common features of the method community and is conceptually simpler.

Soft community boundaries recognise that some members interact with other groups to a greater or lesser extent and ascribe partial membership accordingly. Each method has an associated distribution over the communities in which it participates. Delving deeper into its full role within the program, a complete picture is obtained at the expense of conceptual complexity which can obscure the interpretation of the community's purpose. This can become computationally intensive for large networks.

To summarise, we investigate the extent to which network community algorithms applied to dynamic networks of methods and method calls can reveal useful information about large-scale software systems that can help the maintenance of these software. We expect that method communities that are associated with functional or design features of core importance are preserved across execution traces. We prefer the use of hard membership algorithms for the community detection as these reduce the ambiguity of interpretation of the association of the meaning to communities and the interpretation of anomalies that we may detect through the analysis of method communities across a set of execution traces of the software.

IV. Results

We used three open source software development projects to explore the usefulness of the determination of network communities in networks of methods and method calls, these

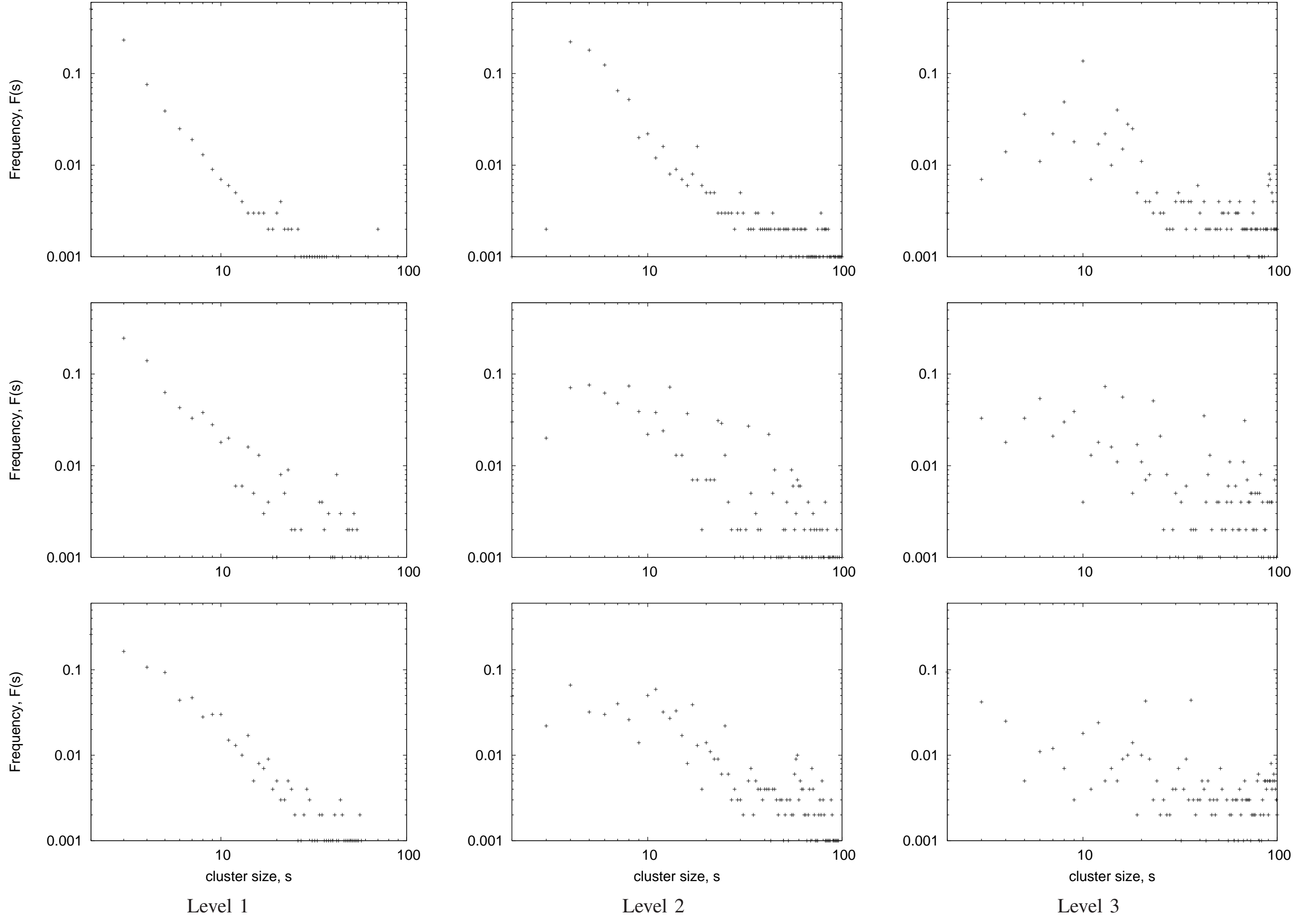

Fig. 1: Community size v normalized frequency for ArgoUML (top), JabRef (middle) and muCommander (bottom) for Levels 1–3 of the hierarchy found using the Louvain algorithm

software projects are ArgoUML (version 0.22; 924 KLOC) a UML modelling tool, JabRef (version 2.6; 148 KLOC) a bibliographic reference manager, and muCommander (version 0.8.5; 85 KLOC) a file manager tool. These software projects have been used previously in software maintenance research for benchmarks and they have publicly available execution traces for many usage scenarios. The definitions of the scenarios that generated the execution traces are described in [44]. The traces are available as XML files generated using the Eclipse Test & Performance Tools Platform (TPTP) [45]. The trace files were parsed to construct the network of methods with edges representing calls between methods and weights corresponding to the frequency of these calls in individual execution traces. The methods were assigned one or two stereotypes using the JStereoCode tool for Java [46] and the results of this were reported previously in [12]. Each network corresponding to an execution trace was analysed separately to determine the the communities of methods in the network.

The links (edges) in the method call networks were weighted by the number of times a method-to-method call was observed by the number of times the caller method calls the callee method during the execution trace. Note that we ignore the direction of the calls between methods. The Louvain algorithm was then used to find communities in the network. In the context of the software network, methods are the nodes i and j of the network, the method calls are the links between nodes and the method call frequencies are used as the link weights, represented in Equation 1 by the matrix elements A_{ij}.

The modularity function is the sum of weights of links between communities, c, greater than that expected in a randomly-wired network. The modularity, Q, (derived in [47]) is given by

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \quad (1)$$

where A_{ij} represents the weight of the link between nodes

TABLE I: Average number of communities found at each level of the hierarchy with level 0 being a community size of 1

level	*ArgoUML*		*JabRef*		*muCommander*	
0	2,682	± 319	1,030	± 227	2,695	± 243
1	498	± 36	108	± 25	211	± 17
2	88	± 6	35	± 6	46	± 19
3	25	± 2	21	± 3	24	± 23

i and j, $k_i = \sum_j A_{ij}$, $k_j = \sum_i A_{ij}$, $m = \frac{1}{2}\sum_{ij} A_{ij}$ and δ is a Kronecker delta function selecting only nodes in the same community (1 when $c_i = c_j$, 0 otherwise where node i belongs to community c_i). Given that $k_i k_j/2m$ is the probability of a link existing between i and j, a non-zero value for Q represents a departure from the random linkages of an equilibrium network. The Louvain algorithm iterates over 2 phases to find the maximum value for Q. It starts with each node assigned its own community and then calculates, for each neighbour j of node i, the gain in Q by moving i into a community with j. Blondel *et al.* [37] took advantage of the fact the expression of the difference in modularity is quick to calculate, such that very large networks ($> 10^6$ nodes) are only limited by the storage space required by the network, not the computation involved. The difference is given by

$$\Delta Q = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right], \quad (2)$$

where $\sum_{in}$ is the sum of the weights inside the community, $\sum_{tot}$ is the sum of the weights of links to the community, k_i is the sum of the weights of the links to node i and $k_{i,in}$ is the sum of the weights from i to nodes inside the community.

The Louvain algorithm is a hierarchical community finder. It creates large communities by combining small groups that are interlinked. It has been observed that the distribution of community sizes displays a power law form, but unlike the preferential attachment mechanism for the degree distribution of a complex network, no reason for this behaviour has yet been proposed [48]. This is shown in Fig. 1 at the lowest level of the hierarchy. As communities are combined at higher levels the plot trends to higher values on the right as the numbers of small communities are depleted.

The algorithm found 4 levels of hierarchy in the dynamic method networks that we analysed. The lowest level of the hierarchy with the greatest number of communities was selected for analysis, yielding between 47 and 582 communities found in a network. A label was automatically generated for each community by finding the term frequencies of the classname or stereotype of each method in the community and normalising them with the document frequencies to avoid uncommon terms being undervalued in the label ordering.

To compare the communities found across method traces, the Jaccard index was used to match the closest communities between traces based on the list of method names. The weighted descriptors in the labels were treated as values of vector components in descriptor-space and the cosine similarity between the two labels was calculated using the dot product of the label vectors [49]. The similarity was then plotted against the Jaccard index of the communities grouped according to size with large groups having more than 10 members, as shown in Fig. 2. On the whole, the cosine similarity of the labels is higher than the Jaccard index of the communities indicated by the weight of data points lying above the diagonal line, more so for large groups. This means that if a community exists across method traces, indicated by a high Jaccard index, the cosine similarity of the labels of the two communities is also high, implying that the distribution of class or method names is stable even if the community membership varies to some extent.

In order to associate functionality to method communities we analysed the class labels and method names of the methods in each community. This analysis indicates that methods belonging to a community perform together a functionality or intended action of the software system. Usually some of the class names represents closely related classes, but also usually there are methods in the community, which belong to classes further away in the class hierarchy. Overall, the communities that we were able to determine represent functionalities of the software that can be expected to map on functionalities implied by requirements that apply to the software system. We note that the analysis that leads to the association of functionality to method communities involves subjective assessment of the meaning of class and method names, however, this analysis can be performed in a semi-automated manner by cataloguing the class belonging of the involved methods (see above the generation of class labels for communities).

Analysing the method stereotype labels associated with the communities we found that in most cases it is not possible to associate a dominant stereotype to the communities. In the cases when this was possible for communities with more than 10 methods, it was always the case that the Constructor stereotype dominated the stereotype label of the community. In general we found that communities that are preserved across traces (i.e. indicated by high Jaccard index for the matched communities) have a stable distribution over stereotypes even if the list of methods belonging to the communities changes to some extent from one execution trace to another execution trace. The lack of dominant method stereotype association to most communities indicates that the delivery of the functionality that can be associated to method communities requires in most cases a mix of method stereotypes. Exceptions from this are communities for which the associated functionality is the creation of functionally related objects in which case the community may have the Constructor stereotype as the associated dominant method stereotype.

To measure cross-trace community preservation, communities were matched across method traces using the highest Jaccard Index found. If the Jaccard Index exceeded a threshold of 0.5, a link in a preservation chain was recorded. For those method traces falling below that threshold, up to 5 other

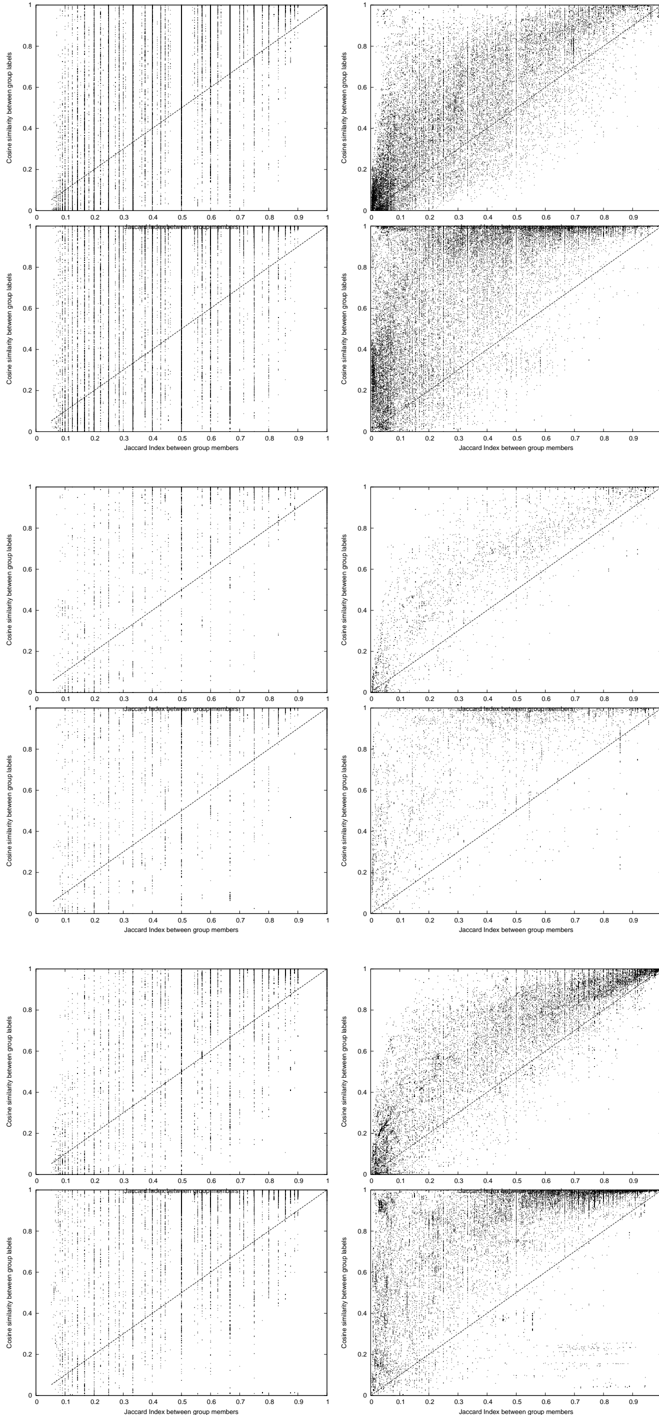

Fig. 2: A plot of label similarity against the Jaccard Index of the community membership for small groups on the left and large groups on the right, classname labels above, stereotype labels below, ArgoUML(top), JabRef (middle), muCommander (bottom)

trace comparisons were examined in order to keep the chain from too early ending. Fig. 3 shows the joint distribution of chain length and community size for each considered software project. The cross-trace community preservation analysis shows that most communities have either short or long preservation chains across execution traces, and relatively few communities have a mid-size preservation chain. The results (see also Table II) show that the preservation pattern of method communities differs across the three software projects. In JabRef, a large part of large communities are preserved, in ArgoUML, the larger part of large communities are not preserved and in muCommander, more large communities are preserved than not preserved across traces. In terms of small communities, the three software projects behave similarly having almost half of the small method communities preserved and almost half not preserved across all execution traces.

TABLE II: Statistics on persistence for each software project where group sizes are divided into Large(> 10 methods) and Small, the top block refers to the proportion of those groups that have a chain length above the threshold and their mean chain length divided by the maximum number possible and the bottom block refers to the proportion of groups with a chain length below the threshold and their normalised mean chain length.

project	group size	threshold	proportion	norm. length
ArgoUML	Large	70	0.23	0.98
ArgoUML	Small	70	0.46	0.97
JabRef	Large	30	0.89	0.94
JabRef	Small	30	0.47	0.92
muCommander	Large	70	0.44	0.97
muCommander	Small	70	0.47	0.96
ArgoUML	Large	30	0.77	0.09
ArgoUML	Small	30	0.46	0.07
JabRef	Large	12	0.11	0.21
JabRef	Small	12	0.45	0.11
muCommander	Large	30	0.38	0.09
muCommander	Small	30	0.40	0.07

V. Discussion

The community analysis of dynamic networks of methods and method calls identifies methods that contribute to the delivery of a certain functionality or functional feature of the analysed software. As we already suggested, this analysis can support the identification of methods that need changing in response to changing functional requirements, or methods that are involved in the delivery of undesired functional features or bugs. We note that the methods belonging to a community do not necessarily belong to the same class, but they all are likely to contribute to the delivery of the same functional feature of the software.

The method communities that we found do not associate with a particular design feature of the methods (i.e. in terms of method prototypes [7], [12], [23]). Thus the considered approach cannot be used to detect design commonalities across methods belonging to network communities. However, the distribution over method stereotypes of method communities is stable across execution traces, even as the list of methods belonging to matching communities changes to some extent. Thus it appears that the delivery of software functionality by communities of methods is associated with a particular

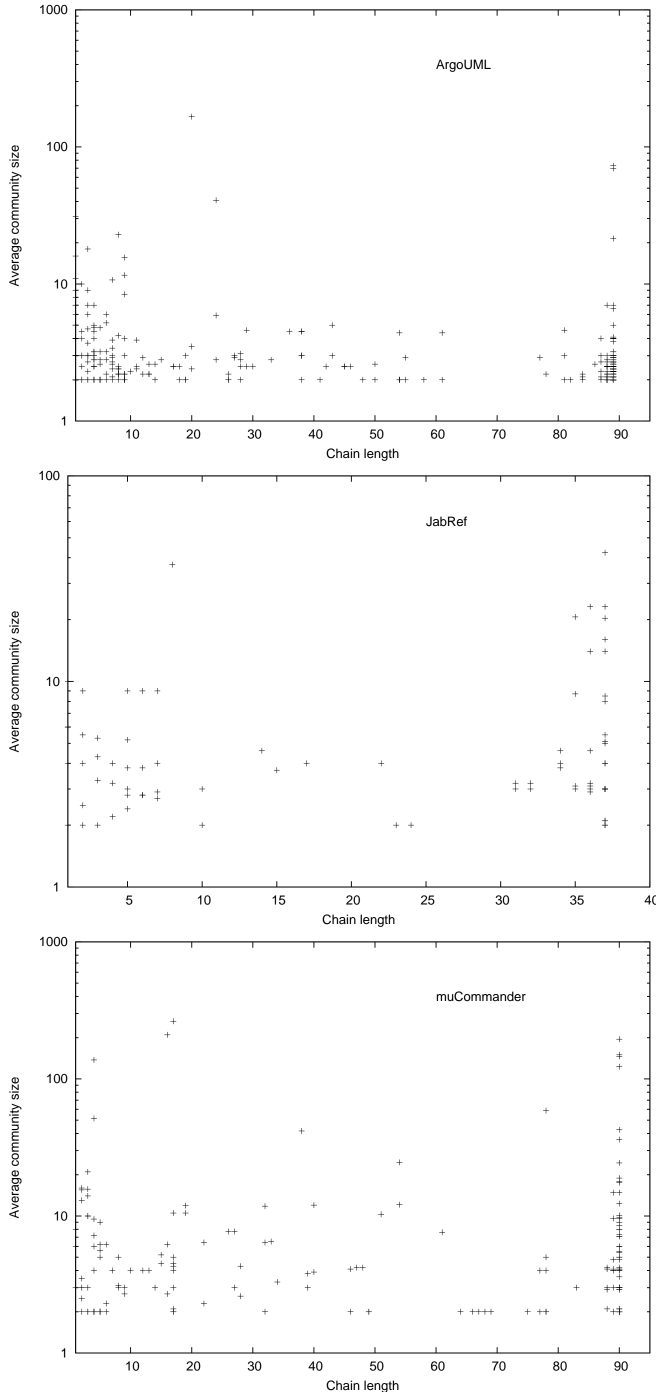

Fig. 3: Average community size versus persistence chain lengths for ArgoUML, JabRef and muCommander

distribution of design features. This property of network communities of methods can be used to detect undesired deviations of the delivery of software functionalities from the well established and valid way of delivering it, following a revision of the software. Alternatively it may also be used to quantify the change from one version to another version of the software with respect to the delivery of a given software functionality (e.g. in response to modified requirements).

Some but not all method communities are preserved across execution traces. The communities with high preservation across traces are likely to deliver core functional features of the software, while those which are less preserved are likely to deliver more specialist features. This implies that revisions of methods that belong to highly preserved method communities might impact the delivery of core functionalities of the software while revisions of methods involved in less preserved method communities are likely to have less wide ranging potential impact. This may help in prioritising checking and testing efforts during extensive revisions of the software system.

Naturally, our results invite the extension of this work to the use of soft membership based community detection algorithms. As we noted the disadvantage of this approach is the ambiguity of the results, however on the other side this approach may provide a finer grained picture of how methods collaborate to deliver functional features of the software. The use of soft membership methods is also much more computationally intensive, which makes the analysis of large volumes of network data slow. The overlapping communities may make the interpretation of functional meaning associated with communities less clear. At the same time the measurement of community preservation across traces may work as a more sensitive measure of design and implementation problems than the same measure applied to non-overlapping communities.

VI. Limitations of validity

The software systems that we analysed are modestly large (only one of them is around 1 MLOC, the other two are smaller). All three software systems that we analysed are open source developed by a community of developers. These mean that software developed in a stricter industrial context may behave to some extent differently in terms of method community analysis, although in principle we expect that our results will hold in the case of such software as well. In the case of smaller scale software the results of method community analysis may be different from those presented here, given that small software may be designed and implemented in a more controlled and more optimal way than large-scale software developed over long time by multiple teams.

As we already noted the use of hard membership based community detection algorithms means that methods that could have been allocated to multiple communities get allocated to the community to which they mostly belong. On one side, this make the interpretation of the community detection more clear, but on the other side ignores the multiple community affiliations of methods. The choice of the community detection algorithm also has an impact on the detected communities. Here we used a well established algorithm which gives very consistent community detection. Other algorithms may lead to detection of different method communities with potentially different associated meaning.

To build the dynamic networks of methods and method calls it is required to do dynamic analysis instrumentation of the

software. This may slow down the software although with current fast machines this is likely to be not very significant. At the same time, the instrumentation in genera requires access to the code, so this kind of analysis can be applied to software for which the code is accessible.

The association of functionality to network communities is based on interpretation of the results. This may input some subjective judgement in the case of large communities with methods that belong to a number of classes and deliver a number of elementary functional features. In the case of smaller communities this is usually not a problem.

VII. Conclusions

In this paper we investigated the usefulness of network community detection algorithms in the context of software maintenance and the use of dynamic networks representing methods as nodes and calls between methods as edges. We considered hard community detection which assigns each method to only one community. We applied this analysis to three open source software projects (ArgoUML, JabRef, and muCommander) for which dynamic analysis data is publicly available. The results show that the communities that we can detect have an associated function that can be determined by analysing the class names, method names, and method contents. We also found that many communities that we detect persist over many execution traces indicating that these communities of methods deliver some core functionality within the software. At the same time we found no clear association of method stereotypes to communities, however, the method stereotype signature (i.e. distribution over method stereotypes) of the persistent communities remains unchanged even as the list of methods changes to some extent across the traces.

Our results show that the community detection applied to dynamic networks of methods with edges representing calls between methods, is able to group together methods belonging to multiple classes such that the methods in the community have a functional association. This means that this analysis can identify in automated manner functionally related methods across classes and these method communities then can be used to map onto functionalities of the software that correspond to requirements. Turning this around this approach can help to map requirements onto functional features of the software that are realised by communities of methods. This can be useful in the context of design and implementation of changes of the software in response to changes in the requirements for the software.

The determination of method communities and the analysis of the preservation of these across execution traces may also help the assessment of the software quality. The possible change of the method stereotype signature of method communities as the software evolves through versions may indicate stricter or more relaxed design guidelines and may also indicate the extent of revision of methods delivering the community associated functionality between versions. The reduction in the extent of preservation of method communities across traces may indicate functional fragmentation of the software and potentially also functional overlap between method communities, which might be undesirable features for the software that require quality improving intervention.

We intend to extend in the future this research by considering a range of hard and soft community detection algorithms comparing the communities that they detect and the interpretation of these in functional or design or other terms relevant for the understanding and maintenance of the software. We find exciting the challenge to look for or develop community detection algorithms that are able to pick our communities with certain functional or design features. We find also very intriguing the comparison between the interpretation of the results of hard and soft community detection algorithms and the assessment of the benefits of these approaches and trade-offs between the benefits and costs of the application of them to method networks. We expect that this research will lead to interesting new results and potentially also to new tools that can support effectively the understanding of large-scale software and the maintenance of such software systems.

References

[1] G. Goth, "Ultralarge systems: Redefining software engineering?" *Software, IEEE*, vol. 25, no. 3, pp. 91–94, 2008.

[2] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and maintenance of large-scale multi-language software applications," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 497–500.

[3] D. Li, "Complexity measurement of large-scale software system based on complex network," *Journal of Networks*, vol. 9, no. 5, pp. 1317–1324, 2014.

[4] "The chromium projects," Jan 2015. [Online]. Available: http://www.chromium.org

[5] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Using metrics to identify design patterns in object-oriented software," in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*. IEEE, 1998, pp. 23–34.

[6] J. K. Ng, Y. Guéhéneuc, and G. Antoniol, "Identification of behavioural and creational design motifs through dynamic analysis," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 8, pp. 597–627, 2010.

[7] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[8] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, 2009.

[9] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Physical Review E*, vol. 68, no. 4, p. 046116, 2003.

[10] L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye, "Linux kernels as complex networks: A novel method to study evolution," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 41–50.

[11] A. Pakhira and P. Andras, "Using network analysis metrics to discover functionally important methods in large-scale software systems," in *Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on*. IEEE, 2012, pp. 70–76.

[12] P. Andras, A. Pakhira, L. Moreno, and A. Marcus, "A measure to assess the behavior of method stereotypes in object-oriented software," in *Emerging Trends in Software Metrics (WETSoM), 2013 4th International Workshop on*. IEEE, 2013, pp. 7–13.

[13] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 531–540.

[14] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 772–787, 2011.

[15] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, pp. 509 –512, Oct 1999.

[16] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási, “The large-scale organization of metabolic networks,” *Nature*, vol. 407, no. 6804, pp. 651–654, 2000.

[17] M. Pohl and S. Diehl, “What dynamic network metrics can tell us about developer roles,” in *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. ACM, 2008, pp. 81–84.

[18] R. Premraj and K. Herzig, “Network versus code metrics to predict defects: A replication study,” in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 215–224.

[19] Y. Qu, Q. Zheng, T. Liu, J. Li, and X. Guan, “In-depth measurement and analysis on densification power law of software execution,” in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 55–58.

[20] M. E. J. Newman, “Fast algorithm for detecting community structure in networks,” *Physical Review E*, vol. 69, no. 6, p. 066133, Jun 2004, arXiv:cond-mat/0309508.

[21] J. Leskovec, K. J. Lang, and M. Mahoney, “Empirical comparison of algorithms for network community detection,” in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 631–640.

[22] M. Girvan and M. E. J. Newman, “Community structure in social and biological networks,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 12, pp. 7821 –7826, Jun 2002.

[23] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse engineering method stereotypes,” in *Software Maintenance, 2006. ICSM’06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 24–34.

[24] M. D. Ernst, “Static and dynamic analysis: Synergy and duality,” in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[25] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, “A heuristic-based approach to identify concepts in execution traces,” in *2010 14th European Conference on Software Maintenance and Reengineering (CSMR)*, Mar 2010, pp. 31–40.

[26] S. Medini, P. Galinier, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “A fast algorithm to locate concepts in execution traces,” in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, M. B. Cohen and M. . Cinnide, Eds. Springer Berlin Heidelberg, 2011, no. 6956, pp. 252–266.

[27] L. Alawneh and A. Hamou-Lhadj, “Identifying computational phases from inter-process communication traces of HPC applications,” in *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, Jun 2012, pp. 133–142.

[28] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[29] E. Arisholm, L. C. Briand, and A. Foyen, “Dynamic coupling measurement for object-oriented software,” *Software Engineering, IEEE Transactions on*, vol. 30, no. 8, pp. 491–506, 2004.

[30] I. Turnu, G. Concas, M. Marchesi, and R. Tonelli, “The fractal dimension metric and its use to assess object-oriented software quality,” in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*. ACM, 2011, pp. 69–74.

[31] K.-K. Yan, G. Fang, N. Bhardwaj, R. P. Alexander, and M. Gerstein, “Comparing genomes to computer operating systems in terms of the topology and evolution of their regulatory control networks,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 20, pp. 9186–9191, 2010.

[32] M. S. Granovetter, “The strength of weak ties,” *American Journal of Sociology*, vol. 78, no. 6, pp. 1360–1380, May 1973.

[33] L. Danon, J. Duch, A. Arenas, and A. Díaz-Guilera, “Community structure identification,” in *Large scale structure and dynamics of complex networks: from information technology to finance and natural science*, ser. Complex Systems and Interdisciplinary Science, G. Caldarelli and A. Vespignani, Eds. World Scientific, 2007, vol. 2.

[34] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, “Uncovering the overlapping community structure of complex networks in nature and society,” *Nature*, vol. 435, no. 7043, pp. 814–818, Jun 2005.

[35] A. C. Lewis, N. S. Jones, M. A. Porter, and C. M. Deane, “The function of communities in protein interaction networks at multiple scales,” *BMC Systems Biology*, vol. 4, no. 1, p. 100, Jul 2010.

[36] U. Brandes, D. Delling, M. Gaertler, R. Goerke, M. Hoefer, Z. Nikoloski, and D. Wagner, “Maximizing modularity is hard,” *arXiv:physics/0608255*, Aug 2006, arXiv: physics/0608255.

[37] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct 2008.

[38] M. Rosvall and C. T. Bergstrom, “Maps of random walks on complex networks reveal community structure,” *Proceedings of the National Academy of Sciences*, vol. 105, no. 4, pp. 1118–1123, Jan 2008.

[39] S. Fortunato, “Community detection in graphs,” *Physics Reports*, vol. 486, no. 35, pp. 75–174, Feb 2010.

[40] D. Hric, R. K. Darst, and S. Fortunato, “Community detection in networks: structural clusters versus ground truth,” *arXiv:1406.0146 [physics, q-bio]*, Jun 2014.

[41] W. Zachary, “An information flow model for conflict and fission in small groups,” *Journal of anthropological research*, vol. 33, no. 4, pp. 452–473, 1977.

[42] L. Leydesdorff, “On the normalization and visualization of author cocitation data: Salton’s cosine versus the jaccard index,” *Journal of the American Society for Information Science and Technology*, vol. 59, no. 1, pp. 77–85, 2008.

[43] L. Hamers, Y. Hemeryck, G. Herweyers, M. Janssen, H. Keters, R. Rousseau, and A. Vanhoutte, “Similarity measures in scientometric research: the jaccard index versus salton’s cosine formula,” *Information Processing & Management*, vol. 25, no. 3, pp. 315–318, 1989.

[44] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, Jan 2013.

[45] “Eclipse test & performance tools platform project.” [Online]. Available: http://eclipse.org/tptp/

[46] L. Moreno and A. Marcus, “JStereoCode: automatically identifying method and class stereotypes in java code,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 358–361.

[47] M. E. J. Newman, “Analysis of weighted networks,” *Physical Review E*, vol. 70, no. 5, p. 056131, Nov 2004.

[48] S. Battiston, M. Catanzaro, and G. Caldarelli, “Social and financial networks,” in *Large scale structure and dynamics of complex networks: from information technology to finance and natural science*, ser. Complex Systems and Interdisciplinary Science, G. Caldarelli and A. Vespignani, Eds. World Scientific, 2007, vol. 2.

[49] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.