International Journal of Cooperative Information Systems
Vol. 21, No. 4 (2012) 297–341

DOI: 10.1142/S0218843012500050

DISCOVERING AND MANAGING SOCIAL COMPOSITIONS IN COLLABORATIVE ENTERPRISE CROWDSOURCING SYSTEMS

FLORIAN SKOPIK
AIT Austrian Institute of Technology
Safety and Security Department
2444 Seibersdorf, Austria
florian.skopik@ait.ac.at

DANIEL SCHALL
Siemens CT T CEE
Siemensstrasse 90, 1211 Vienna, Austria
daniel.schall@siemens.com

SCHAHRAM DUSTDAR
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8, 1040 Vienna, Austria
dustdar@infosys.tuwien.ac.at

Received 18 July 2012
Accepted 10 December 2012
Published 5 February 2013

Crowdsourcing is an increasingly used model to outsource certain tasks to be carried out by external experts on the Web. Especially when lacking experience or expertise with certain task types, crowdsourcing offers a convenient way to receive instant support. In this paper, we introduce an in-house enterprise crowdsourcing model, which leverages the crowdsourcing concept and transfers it to traditional organizations. Here, a company's staff is considered a crowd that — besides its regularly assigned tasks — can also receive tasks from colleagues from other departments and across hierarchical structures. The aim is to offer instant support and utilize free capacities throughout a large organization more efficiently. In our work, we describe this concept and supporting mechanisms in context of an agile software development use case. However, in contrast to usually crowdsourced microtasks, complex software architectures usually consist of tens and hundreds of connected modules that can be potentially crowdsourced. These technical dependencies between modules require active coordination and interactions between crowd members that process the single artifacts. Hence, technical dependencies of artifacts result in social dependencies of *collaborating crowd members* that create them. In order to efficiently discover *member compositions* based on artifact dependencies, we introduce an indexing and discovery approach based on subgraph matching. Typically, assigning tasks to well-rehearsed teams results in more reliable task processing, faster results, and higher quality of work. We evaluate our approach in terms of system scalability and overall applicability by mining and analyzing the popular SourceForge community. We show that our approach of member composition discovery is feasibly

in terms of scalability and quality of discovery results. Our findings deliver important input for the design and implementation of supporting information systems for future large-scale collaboration platforms.

Keywords: SOA-based collaboration; interaction mining; enterprise 2.0; crowdsourcing; social networks; social composition discovery; subgraph matching.

1. Introduction

The collaboration landscape has changed dramatically over the last years by enabling users to shape the Web and availability of information. While in the past collaborations were bounded to intra-organizational collaborations using company-specific platforms, and also limited to messaging tools such as e-mail, it is nowadays possible to utilize the knowledge of an immense number of people participating in collaborations on the Web. The shift toward the Web 2.0 allows people to write blogs about their activities, share knowledge in forums, write Wiki pages, and utilize social platforms to stay in touch with other people. Task-based platforms for human computation and crowdsourcing, enable access to the manpower of thousands of people on demand by creating human-tasks that are processed by the crowd. Human-tasks include activities such as designing, creating, and testing products, voting for best results, or organizing information. This paradigm is increasingly utilized by today's companies to enable scalable distributed software development by outsourcing tasks to external experts when lacking particular in-house expertise or development time.

A wide variety of software development processes, models, and managing techniques have been proposed in the last decades.[1] They emerged from the need for structuring and managing work in large-scale teams. Though these models have led to accepted standards for software development approaches, they typically focus on task and artifact dependencies but widely neglect social aspects of software development. We argue that both technical *and* social dependencies are of paramount importance, especially for today's agile paradigms. **Technical dependencies** are mainly induced by artifact dependencies, e.g. software modules, being created and developed. Various methods exist to identify these dependencies, such as *call-graphs* that show links of components during run-time. Once relations between artifacts, e.g. software components and modules of a planned system, have been identified, we use the concept of flexible, collaborative activities to describe work to be performed, i.e. artifacts to be created in order to build the designed system. Thus, activity dependencies reflect artifact dependencies. By dividing work in separate pieces, activities can be distributed across crowd members. However, as a consequence, initially decomposed work eventually needs to be composed and integrated again to obtain the final result. This process requires substantial coordination effort among crowd members, i.e. artifact creators. Due to this collaborative aspect it deems to be more beneficial to pick crowd members that are familiar with each other's working style. Managing these **social dependencies**

is typically not covered by today's crowdsourcing platforms that hardly support interactions between single members. Moreover, crowd members typically need to interact frequently if they work on related artifacts in order to make sure that their work is aligned and will finally fit together. Even rigidly defined interfaces between separately developed software modules usually cannot avoid this inherent need for communication. As described by Conway's Law[2] decades ago, *because software modules interact, this creates a similar need of interaction among software developers*.[3]

1.1. *The principle of collaborative enterprise crowdsourcing*

In this paper, we discuss the foundational model for a collaborative enterprise crowdsourcing environment. Essentially, this model takes the usual concept of crowdsourcing on the Web and applies it to an enterprise collaboration context. The basic properties of this environment are that (i) *preselected experts* (ii) can be *flexibly involved* in *ongoing work* by outsourcing them generally encapsulated tasks, however (iii) still giving them the means to coordinate their work (*interaction tool support*). Especially the last point (active coordination and collaboration between crowd members) is a strong requirement to deal with complex tasks within enterprises compared to crowdsourcing microtasks on the Web.

The notable point here is that crowdsourcing is driven by company staff who are all hired experts and who collaborate with each other in context of the crowdsourced tasks. However, it is not the intention to replace existing working collaboration models but to establish this collaborative crowdsourcing model in parallel to traditional models in order to better utilize company resources. Open Source Software (OSS) development shares some of these properties with the envisioned environment, especially that interactions between team members happen largely online, participation of people is (often) more flexible, work fully distributed, and participants are only loosely coupled to management. We argue that these properties are vital to an enterprise environment in which — besides regular work — instant help and support can be flexibly gathered, even across organizational boundaries and hierarchical structures. Of course, in order not to render existing management structures useless, this model is applied only for a small fraction of the work time, i.e. indeed for instant help and support and tasks being carried out in free cycles only.

1.2. *Research challenges*

The overall research challenge is to design an infrastructure that supports the described environment. Previous research — which is referred to, however not in scope of this work — tackles the following challenges:

- Flexible integration of humans in collaborative environments using service-oriented concepts, such as *Human-Provided Services* (*HPS*)[4–6]

- Dynamic creation of trust networks using collaboration monitoring techniques,[7,8] which allows us to automatically detect social relations between crowd members.

The current paper deals with challenges on top of a flexible trust-based collaboration network, especially dealing with the following concerns:

- Indexing approach for social compositions utilizing emerging trust networks.
- Query and discovery mechanisms using subgraph matching.

1.3. *Contributions*

In this paper, we describe an approach to manage socio-technical dependencies in collaborative crowdsourcing environments. Here we introduce *socio-computational crowds* based on the concept of human computation (e.g. see Ref. 9) that was established in the context of crowdsourcing. Traditional crowdsourcing environments lack to a great extent collaborative aspects. Our approach leverages social networks to support the collaborative processing of crowdsourced tasks. For that purpose we focus on enabling interactions and establishing social relations in crowdsourcing environments, as well as on the management of reliable crowd member compositions. In particular, this paper deals with the following contributions:

- *Socio-computational crowdsourcing fundamentals.* We motivate the need for interactions between crowd members that usually act only isolated on today's platforms. Our approach utilizes service-oriented computing paradigms to build a loosely coupled crowd community of experts within large-scale enterprises.
- *Dependency management approach.* We highlight a conceptual model to enable sophisticated management of social dependencies in aforementioned collaborative environments. In detail, we discuss the implementation of a feature-based discovery model that enables the efficient search for crowd member compositions.
- *Structure and dynamics of the sourceforge community.* We study real community data and extract common properties that build the basis for further evaluations of our approach in terms of scalability and applicability. Furthermore, we discuss the limits of our approach.

The remainder of this paper is organized as follows. Section 2 deals with an enterprise collaboration use case for a private crowd that spans numerous departments and sub-organizations, and discusses foundational concepts of our proposed *socio-computational crowd* approach. Then, the core contribution — a feature-based discovery model to identify common crowd member compositions — is formulated in Sec. 3. We study a real data set, show evaluation results and discuss major key findings in Sec. 4. Section 5 outlines the state of the art in collaborative software research and crowdsourcing applications, and discusses related work applied in our approach, including service-oriented computing, social networks and trust. Finally, the paper is concluded in Sec. 6.

2. Basic Building Blocks of Enterprise Crowdsourcing

We start with a motivating scenario that introduces basic concepts. Furthermore, we discuss foundational building blocks, that enable seamless human participation in service-oriented architectures (SOA) and account for social implications.

2.1. *Agile software development processes*

Let us consider a software development project that is executed in an agile manner.[10] Such projects do not follow a top-down approach where all requirements are gathered upfront. In agile software engineering, typically all engineering cycles are performed iteratively. Each iteration consist of (i) requirements analysis, (ii) design, (iii) implementation, (iv) integration, and (v) testing. Many of today's software companies have adopted this methodology since it allows to consider emerging customer requirements. For example, a customer may request new features during the development of a software product. These new requirements are then evaluated in a new development iteration.

Here we consider a large scale software development project comprising various team members including project managers, requirements engineers, software architects, developers, and testers. We assume that the overall project is structured roughly in three essential phases.

(i) The initial requirements are gathered, basic technologies are evaluated and crucial design decision are made by a small core team. The core team usually consists of senior people (e.g. chief architect, senior developers) that define the most essential features of the software. For example, suppose the team needs to design and implement an extensible framework that offers an API, tools, libraries, etc.

(ii) The next step is the implementation of the core features and the API, which is done by the senior developers. At this stage, the team is still kept small and typically geographically collocated because people need to collaborate and discuss extensively. This phase may already be performed according to the agile process where all before mentioned phases are part of an iteration. Depending on the size, resources, and duration of the project, an iteration may take several weeks. Also, we assume that certain automation in the development process already takes place such as automatic builds, generation of test reports, check of test coverage, etc.

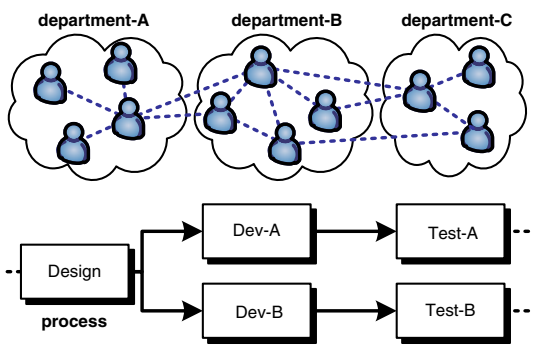
(iii) In the third phase of the project we assume a transition of the centralized team to a distributed team. At this point, the core features of the framework have already been defined, documented, and implemented. Team members located in different departments or company sites have the ability to use and extend the core framework. This can be accomplished by designing and implementing components using the framework's API. Again, certain steps of the process are automated such as executing batch scripts, copying files, deployment of

software components, or testing of individual components. However, we assume that various features are implemented and tested by distributed team members where certain steps require tight collaboration between members. For example, the tester may report a problem to the developers who in turn need to discuss changes in the source code.

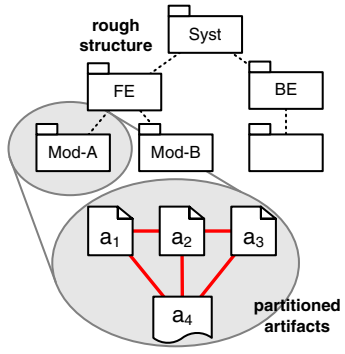
In the following, we discuss a use case scenario with the focus on socio-computational crowds embedded in distributed collaborations.

2.2. *Use case*

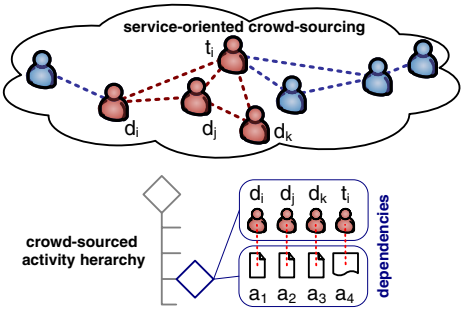
Basic Setting. A motivating scenario, including involved actors and artifacts, is depicted in Fig. 1. Here, in Fig. 1(a), people in geographically distributed departments of a large-scale enterprise collaboratively participate in a software engineering project. In particular, the basic steps and responsibilities are modeled in a



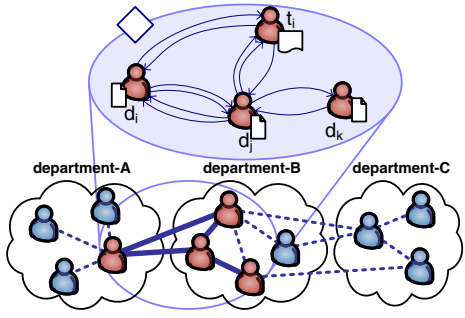
(a) Cross-organizational process.



(b) Technical dependencies.



(c) Social dependencies.



(d) Collaborative crowdsourcing.

Fig. 1. Motivating scenario: (a) geographically distributed departments execute a cross-organizational software development process; (b) a rough package view and detailed artifact partitions capture technical dependencies; (c) technical relations enable the discovery of matching social compositions and creation of crowdsourcing activities; (d) interactions during collaborations approve and update registered social networks.

process that spans numerous organizational units. Using today's modern Web 2.0 approaches and service-oriented architectures[11] people have all tools at hand to flexibly collaborate on the Web. In our scenario, employees of a multi-national organization are connected through a social trust network (reflected by the dotted lines between people).

Single departments take over the responsibility for particular tasks of the global process. We assume that this company applies an agile software development approach, where artifacts are designed, implemented, tested and subsequently refined in short iterative cycles.[10] After creating a rough overview, e.g. a UML package view of the software framework that is going to be developed, each module [e.g. see *Mod-A* in Fig. 1(b)] is partitioned in its (atomic) artifacts that can be processed by individuals. For instance, a software module is decomposed in three classes or submodules a_1, a_2, and a_3. Each of these artifacts is going to be created by a software developer and accompanied by a software tester who creates test cases and a final report a_4.

After technical dependencies have been thoroughly identified [see Fig. 1(b)], artifact structures are mapped onto matching social structures. In other words, tightly coupled technical artifacts typically require a lot of coordination and integration effort, thus, can be best performed by people that know each other's working style from previous collaborations (according to *Conway's Law*[2]).

Several research challenges arise when assigning responsibilities for creating technical artifacts to people:

- What if for given technical artifact compositions, there are no matching social structures, i.e. well-proven team compositions, in the responsible department?
- What if the responsible department has no free capacities while other departments still have?
- What if there are only matching social structures across the borders of numerous organizational units?

Enterprise Crowdsourcing. Today, large-scale enterprises are facing the challenge of effective management and exploitation of the employees' knowledge and resources. Usually, expertise in numerous fields is available but often this knowledge is neither discovered nor captured somewhere. Since decades, researchers invent models and approaches to overcome that issue.[12] Enterprise crowdsourcing[13] follows a different path. Here, employees are encouraged to actively participate in a *private crowd environment*, where they offer their skills and expertise to other departments of the company. On the one side, rare expertise can be discovered and, on the other side, free capacities in one department can be used to tackle peak loads in other departments by **outsourcing especially noncritical activities**. Thus, enterprise crowdsourcing deems to be an elegant new paradigm to harness people's capabilities in a flexible and far more effective manner compared to rather static traditional cross-department collaborations. Service-orientation is the ideal means to realize such private crowds (i.e. not open to the public), because members can

be dynamically discovered, are loosely coupled and thus composed at run-time, and flexibly assigned to activities.

In our use case, a majority of the employees participate in the described *socio-computational crowd environment* as shown in Fig. 1(c). In contrast to the widely used notion of crowdsourcing, we do not use this environment to outsource tasks to single individuals only. We rather outsource compositions of problems, e.g. the creation of technical artifacts having interdependencies, to compositions of crowd members. Therefore, one major challenge is to identify reliable social compositions, i.e. groups of crowd members that have proven their reliable and successful collaboration behavior before. Once identified [as highlighted in Fig. 1(c)], a collaboration activity is created and artifacts a_1 to a_4 (or templates respectively), as well as crowd members d_i, d_j, d_k, and t_i assigned.

Finally, these members create, modify, and extend the required (or given) artifacts. This activity requires an extensive amount of interactions [Fig. 1(d)] to coordinate work, align artifacts, and ensure a smooth integration of software modules later on. Today, a wide range of service-oriented communication, coordination, and collaboration tools are available for crowd members. Furthermore, since interactions are performed through these tools, they can be observed and even analyzed. Thus, valuable information about real collaboration behavior and spirit can be obtained and used to approve and update the social trust network between crowd members. This network is the basis for an effective future discovery of reliable crowd member compositions.

2.3. *Socially-enhanced SOA*

Numerous concepts support the realization of socio-computational crowd communities as described in the use case. We especially focus on Service-oriented Architectures (SOA), since SOA is a natural fit here. It provides exactly the right methodology of being able to flexibly discover resources and bind them to a certain task. However, here we do not aim at discovering services, but human experts who are described by their profiles and social connections they have. Previous work on Human-Provided-Services (HPS)[4] strongly supports this idea. SOA provides a convenient way to support reliable interactions between collaboration partners who remain still loosely coupled, and discovery/binding can be performed using software tools (as evaluated in detail in this paper).

2.3.1. *Flexible service-oriented collaboration*

An activity model[14] attempts to structure loosely coupled collaborations in service-oriented systems. Examples of collaborative activities at various levels of granularity are "sending emails", "reviewing a paper", "organizing a workshop", and "managing a multi-national research project". A single activity provides basic collaborative data. It describes the work to be done and lists the involved people. This data is often sufficient in static collaborative settings where all members are aware of the

overall working environment. In dynamic and distributed collaborative environments, one has to explicitly model the embedding of a single activity in the overall collaboration context. The context contains the structure of activities, dependencies between activities, the temporal flow of activities, and history of activity changes. This provides the core structure of collaborative work. In addition, the collaboration context describes the involvement of members, their roles, required and applied skills, work artifacts, and resources.

Web services play a fundamental role in supporting flexible, cross-enterprise collaboration scenarios. We discuss human interactions in SOA as introduced in our previous work (see HPS approach[4]). HPS enhances the traditional "SOA-triangle" approach by enabling people to provide services using the very same technology as implementations of software-based services (SBS) use. By following the SOA paradigm, three essential steps are performed (see Fig. 2):

(i) *Publish.* Users have the ability to create HPSs and publish the services on the Web using a registry. Publishing a service is as simple as posting a blog entry on the Web. It is the association of the user's profile with an activity template described as a service (WSDL). Interfaces provide the needed metadata support for the discovery of suitable HPSs.

(ii) *Search.* The service requester performs a keyword-based search (reflecting expertise areas) to find HPSs. Ranking is performed to find the most relevant HPS based on, for example, the expertise of the user providing the service. Expertise is determined automatically by the HPS framework through context-sensitive interaction mining techniques.[5]

(iii) *Interact.* The framework supports automatic user interface generation using XML-Forms technology.[a] Thus, personalized interaction interfaces can be generated and rendered for different devices. The HPS framework can be used for interactions between humans and also for interactions between SBS and HPSs.

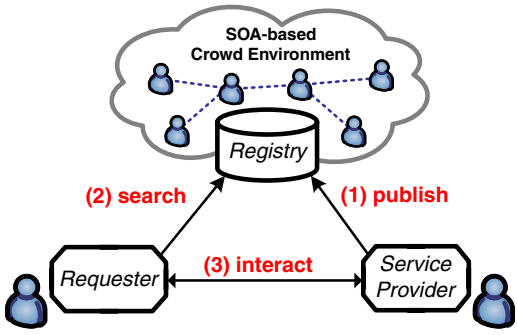


Fig. 2. Enhancing SOA with human capabilities.

[a]XML Forms: `http://www.w3.org/MarkUp/Forms/`

2.3.2. *Emergence of social trust networks*

In contrast to a widely used security perspective on trust, we define *social trust* relying on the interpretation of previous collaboration behavior,[8,15] the similarity of dynamically changing interests,[8,16] and the maturity of social structures. Especially in collaborative environments where users are exposed to higher risks as compared to common social network scenarios[17] and business is at stake, considering social trust is essential to effectively guide interactions.[18] Here, we define trust as follows[8,15,19]: *Trust reflects the expectation one actor has about another's future behavior to perform given activities dependably, securely, and reliably based on experiences collected from previous interactions.*

Interactions and Monitoring. As motivated in the introduced use case, people interact to perform their tasks. Work is modeled as activities, that describe the type and goal of work, temporal constraints, and used resources. As interactions take place in the context of activities, they can be categorized and weighted. SOAP is the standard message format to support interactions between distributed software services. Also human interactions can be supported in a service-oriented manner using technologies such as SOAP (see HPS[4]). This technology including extensions such as addressing and correlation mechanisms is state-of-the-art in service-oriented environments and well supported by a wide variety of software frameworks. This fact enables the adoption of various monitoring and logging tools to observe interactions in service-oriented systems.

Link Metrics. Interaction logs are used to infer metrics that describe the relations of single actors. Various metrics can be calculated by analyzing interaction logs such as behavior in terms of availability and reciprocity. Relation metrics describe the links between actors by accounting for (i) recent interaction behavior, (ii) profile similarities (e.g. interest or skill similarities), (iii) social and/or hierarchical structures (e.g. role models). However, we argue that social trust relations largely depend on personal interactions. We model a community of actors with their social relations as a directed graph, where the nodes denote network members, and edges reflect (social) relations between them. Since interaction behavior is usually not symmetric, actor relations are represented by *directed links*.

Social Trust. Our approach considers the diversity of trust by enabling the flexible aggregation of various interaction metrics (e.g. *success rate* and *responsiveness*) that are determined by observing ongoing collaborations. Finally, available relation metrics are weighted, interpreted, and composed by a rule engine (the detailed mechanisms are described in Ref. 8). The result, e.g. a normalized value in a predefined interval or even a linguistic representation such as `low`, `medium`, or `high`, describes social trust between the actors reflecting maturity of link structures, compatibility of actors' working styles, and potential success of future collaborations. Notice, it is essential for our social network management approach that interactions are captured by the system. Otherwise, if not manually declared, social relations

cannot be managed and are thus not considered in subsequent operations that account for social relation properties.

3. Feature-based Discovery Model

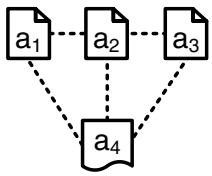
The basic aim of our approach is to discover reliable actors for given activity templates created from artifact structures. Typically, in a collaborative environment, activities are performed by compositions of actors. Such compositions, e.g. teams, consist of actors with potentially different roles having social dependencies. For instance an activity might demand for three software developers specifying and implementing software modules, and a software tester who assesses the created artifacts. The fundamental challenge is to find one (or even more) compositions of actors who can perform this activity. Therefore, this group should have the following properties: (i) the capabilities of humans match the required roles, and (ii) social relations follow artifact relations to enable reliable communication and coordination between dependent actors.

3.1. *Feature-based search*

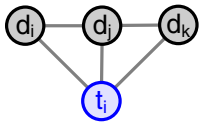
From a scientific point of view, this problem is described by the *concept of induced subgraph isomorphism*.[20] In order to measure the substructure similarity between a target graph (here: distinct parts of a social network) and a query graph (here: a template describing required actor composition properties), different models[21–23] have been proposed; in particular (i) physical property-based, (ii) feature-based, and (iii) structure-based. In this section, we describe a feature-based approach since it allows to introduce some degree of fuzziness in the discovery process and is not as complex to compute as structure-based models; and thus, fit better to large-scale networks. In short, commonly searched elementary features of subgraphs (to be more exact: social compositions[b]) are extracted, for instance, the number of software developers in a team, the degree of cross-links between them, or their field of experience and expertise. Whether a subgraph in the social network matches a query graph is determined by the number of matching features. Given that similarity definition, each frequently active team (i.e. social composition represented by a highly cross-linked subgraph in a social network) is represented by a feature vector, where its single components represent the frequency of common predefined features. The distance between the query graph and a potential match in the social network is measured by the distance between corresponding feature vectors.

Figure 3 depicts an illustrative example. In Fig. 3(a) technical dependencies between three concrete software modules $a_1 - a_3$ and a test case a_4 are identified.

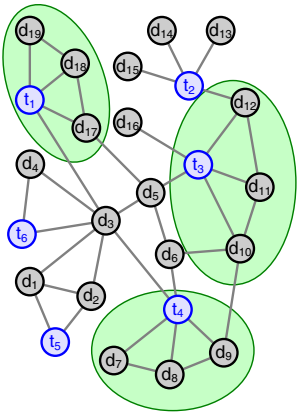
[b]Notice, we use the term *subgraph* if we refer to any arbitrary segment of a graph in mathematical definitions or algorithms. However, we use *(social) composition* if we refer to a concrete set of people in context of the crowd sourcing scenario. Thus, every concrete social composition is also a subgraph in the mathematical sense, but not every arbitrary subgraph is a social composition that emerged from collaborations in joint activities.



(a) Artifact dependencies between three software modules $a_1 - a_3$ and a test report a_4.



(b) Query graph template: roles and social relations between three software **(d)**evelopers and one **(t)**ester.



(c) Discovering query graph structures in a large-scale crowd network (consisting of **(d)**evelopers and **(t)**esters.

Fig. 3. Feature-based search approach: (a) from artifact dependencies (b) a social graph query is inferred and (c) matching social compositions are discovered.

These artifacts and their dependencies are mapped to a social structure consisting of software developers and a tester in Fig. 3(b). Using this generic subgraph template, appropriate instances are discovered in the large-scale social network as highlighted in Fig. 3(c). Notice, in this example, software modules are decomposed in segments so that each of them can be processed by exactly one crowd member.

3.2. *Approach outline*

The feature-based composition model is applied in context of other concepts to keep track of the dynamics in flexible socio-computational crowd environments. The whole approach depicted in Fig. 4 is described as follows:

(i) *Social network definition*: Single crowd members register their personal profiles, consisting of interests, expertise and usual roles; but also their relations to

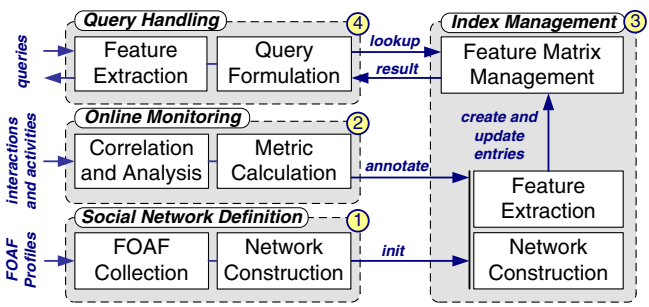


Fig. 4. Approach to feature-based discovery: (1) network definition, (2) network annotation, (3) feature index management, and (4) query handling.

well-known collaboration partners (e.g. FOAF (Friend-of-a-friend)[24] fragments including `knows`-relations[25]). Through semantically reasoning over FOAF snippets a large-scale network model is created and periodically updated.

(ii) *Online monitoring*: Since members communicate over Web services, all SOAP interactions can be logged and analyzed. Metrics, such as interaction frequency and density, describe the strength of predefined social relations between collaborating members. This information is utilized to confirm registered profiles; for instance, to discover defined but not approved relations.

(iii) *Index management*: Periodically analyzing member interactions and performed activities enables the identification of regularly used collaboration patterns (i.e. frequently applied compositions of actors) with commonly requested features.

(iv) *Query handling*: Whenever a query is issued, features of the query graph are extracted (e.g. number of roles, degree of cross-linkage) and an approximate query is created based on features only.

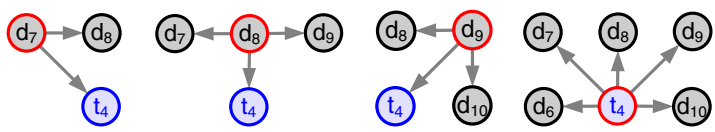
3.3. *Detailed formulation*

We proceed with the definition of an analytical model that supports the discovery of social compositions considering artifact dependencies.

3.3.1. *Social network definition*

Crowd members define their individual FOAF profiles where each one represents a single graph fragment $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ of the social network. In particular crowd members define which other members they `know`, i.e. collaborate with. For instance, members d_7, d_8, d_9, and t_4 define structures as shown in Fig. 5. Of course, they only know their direct neighbors, but by reasoning over these RDF[c] structures, the whole graph can be constructed [being a part of the network depicted in Fig. 3(c)]. We define the whole social network $\mathcal{G}$ to be composed of single fragments as given in Eq. 1. Notice, manually declared relations are the basis for determining which interactions need to be monitored and analyzed to confirm social structures and enrich relations with expressive interaction behavior metrics.

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) = \left(\bigcup_{\forall \mathcal{G}_i} \mathcal{V}_i, \bigcup_{\forall \mathcal{G}_i} \mathcal{E}_i \right). \tag{1}$$



(a) d_7's view (b) d_8's view (c) d_9's view (d) t_4's view

Fig. 5. Aggregating individual FOAF profiles allows the construction of large-scale social networks.

[c]Resource Description Framework: `http://www.w3.org/RDF/`

3.3.2. *Interaction mining and social trust inference*

Since we apply SOAP interceptors and access layers for Web service calls, we are able to capture directed interactions $\mathcal{I}$ that carry some payload p, such as support requests and work delegations, between pairs of crowd members (u, v) in context of an activity a (from a set of activities $\mathcal{A}$). These interactions are realized as standard SOAP calls as explained in detail in our previous work.[8]

$$\mathcal{I} = \{(u, v, a, p) \mid u, v \in \mathcal{V}, a \in \mathcal{A}\}. \tag{2}$$

This enables us to annotate FOAF `knows`-relations with evidence-based interaction metrics $m_i(u, v)$. Common metrics are, for instance, request reciprocity, availability, interaction intensity, frequency and uniformity.[26] These metrics that characterize the behavior of people can be interpreted in terms of trustworthiness and dependability.[8,15] We exemplarily define the following ones, which are also used in the evaluation section:

Reciprocity (recpr). A typical social behavior metric is reciprocity[15] that here reflects the ratio between obtained and provided support in a community. Let $\mathcal{I}_{\text{REQ}}(u, v)$ be the set of u's sent support requests to v, and $\mathcal{I}_{\text{RES}}(u, v)$ the set of u's provided responses to v's requests. Then we define reciprocity in $[-1, 1]$ as in Eq. 3; hence, 0 reflects a balanced relation of mutual give and take.

$$\text{recpr}(u, v) = \frac{|\mathcal{I}_{\text{RES}}(u, v)| - |\mathcal{I}_{\text{REQ}}(u, v)|}{|\mathcal{I}_{\text{RES}}(u, v)| + |\mathcal{I}_{\text{REQ}}(u, v)|}. \tag{3}$$

Availability (avail). This metric describes u's availability for v's requests, i.e. the amount of answered requests. The result of Eq. (4) is a value in $[0, 1]$.

$$\text{avail}(u, v) = 1 - \frac{|\mathcal{I}_{\text{REQ}}(v, u)| - |\mathcal{I}_{\text{RES}}(u, v)|}{|\mathcal{I}_{\text{REQ}}(v, u)|}. \tag{4}$$

Responsiveness (resp). This metric [see e.g. Eq. (5)] describes the response behavior of a crowd member. In particular in today's highly dynamic businesses fast responses on support requests are a key success factor. Here, we calculate the average of response times as a measure for someone's commitment. For that purpose, the average time span t between single interactions $\imath$, i.e. requests and corresponding responses, is determined.

$$\text{resp}(u, v) = \frac{\sum_{\mathcal{I}_{\text{RES}}} t(\imath_{\text{REQ}}(u, v)) - t(\imath_{\text{RES}}(u, v))}{|\mathcal{I}_{\text{RES}}(u, v)|}. \tag{5}$$

Social trust (τ). Finally, these metrics are interpreted and aggregated to reflect the strength of FOAF relations by one score. For that purpose either an arithmetical approach that simply weights normalized metrics, or a rule-based approach that truly interprets metric values (i.e. the trustworthiness of interaction behavior) according to a given rule-base is applied.[8] This operation is represented by $\otimes$ in Eq. (6).

$$\tau(u, v) = \langle(\text{recpr}(u, v), \text{avail}(u, v), \text{resp}(u, v)), \otimes\rangle. \tag{6}$$

3.3.3. *Index management*

The basic aim is to identify frequently occurring actor compositions, e.g. teams, collaborating in context of activities. For that purpose an indexing algorithm uses two data sources: (i) *interaction logs* $\mathcal{I}$ from the monitoring facilities to discover strong social connections and dependencies, and (ii) *activity structures* $\mathcal{A}$ to identify member compositions working in the same context. Since crowd members interact in context of activities, the indexing algorithm traverses a list of finished activities and analyzes interactions between assigned members. Hence, social network structures are identified relying on evidence through interaction mining, rather than manually defined connections from FOAF profiles only.

We harness an index structure, referred to as the feature-graph matrix $\mathcal{M}$,[22] to facilitate the feature-based registration of actor compositions. Each row of the matrix $\mathcal{M}$ corresponds to a registered subgraph $\mathcal{G}_i$ of the social network, while each column corresponds to a feature f_j being indexed. Each entry x_{ij} records the number of the embeddings (values respectively) of a specific feature f_j in the registered subgraph $\mathcal{G}_i$ (Table 1).

Social Composition Features. One of the most challenging parts in our approach is the definition of meaningful and significant features. For our given software engineering scenario, we defined the features given in Table 2 to describe significant properties of social compositions of crowd members. While the first five features deal with structural properties of the social graph only (i.e. number of nodes, particular roles, hubs, and links, as well as the average node degree), the last two features can be interpreted as quality attributes. Here, *avg_trust* is a measure for the link quality between members, while *maturity* is a means to express the stability and

Table 1. Feature-graph matrix index.

	$\mathbf{f_A}$	$\mathbf{f_B}$	$\mathbf{f_C}$	...	$\mathbf{f_j}$
$\mathcal{G}_\mathbf{1}$	x_{1A}	x_{1B}	x_{1C}	...	x_{1j}
$\mathcal{G}_\mathbf{2}$	x_{2A}	x_{2B}	x_{2C}	...	x_{2j}
$\mathcal{G}_\mathbf{3}$	x_{3A}	x_{3B}	x_{3C}	...	x_{3j}
⋮	⋮	⋮	⋮	⋱	⋮
$\mathcal{G}_\mathbf{i}$	x_{iA}	x_{iB}	x_{iC}	...	x_{ij}

Table 2. Social composition features for software development scenarios.

Feature name	Feature description
num_nodes	Number of crowd members in the composition
num_{role}	Number of {role}, e.g. developers, testers
num_links	Number of links in the composition
avg_nodedeg	Average degree of nodes (distribution of links)
num_hubs	Number of nodes linked to all other nodes
avg_trust	Average trust score between members
maturity	Number of activities that approved the composition

reliability of a connection. In particular, maturity captures the number of activities which have been performed by a certain social composition and is therefore an important attribute to express the probability of future collaboration success.

Notice, varying features are useful for different use cases. For instance, if a requester (i.e. someone issuing a query) defines clear requirements on roles s/he can omit num_nodes. However, especially in software development roles are sometimes fuzzy (e.g. differences in activities of designers, implementers, and testers). In such cases, a requester could also simply look for a social composition that is sufficient in terms of manpower (num_nodes) and omit num_{role} instead.

Index Updates. The index requires frequent updates to reflect the real social networks of highly dynamic collaboration environments. Thus, whenever an activity is finished, involved members and their relations are analyzed and either a new subgraph $\mathcal{G}_i$ added to the index, or an existing one (consisting of the same members) is updated in terms of changed features. Indexing algorithms are discussed in the next section in greater detail.

3.3.4. *Approximate discovery and query relaxation*

In order to discover appropriate crowd members to process a set of technical artifacts, queries on the social network, in detail on the feature graph index, are issued. Typically a composition of technical artifacts $\mathcal{G}_\mathcal{A}$, as shown in Fig. 3(a), is given; for instance a class diagram, entity-relationship diagram or higher-level dependency model. This input is mapped to a social dependency graph [Fig. 3(b)] using a scenario-specific bijective mapping between user roles and artifact types, i.e. different types of technical artifacts require distinct roles of crowd members. For instance, a software module requires a software developer, or a test report demands for a tester. Furthermore, the type (implements, uses, calls, etc.) and degree of coupling between technical artifacts determine the required strength of social relations between assigned crowd members; for instance, the development of tightly coupled modules results in higher coordination effort and thus requires distinct social trust τ.

Algorithm 3.1 shows the details. First, for each technical artifact $x \in \mathcal{V}_\mathcal{A}$ of a particular type ($x.Type$) a node template for a crowd member with a corresponding role ($u.Involvement.Role$) is created (Line 3). This template for a crowd member, which is essentially a node in the query graph, is to be "filled" later with a certain crowd member. Then, for all existing dependencies from $x \in \mathcal{V}_\mathcal{A}$ to any $y \in \mathcal{V}_\mathcal{A}$, a social trust value τ between corresponding actors $u, v \in \mathcal{V}_\mathcal{Q}$ of the social query graph $\mathcal{G}_\mathcal{Q}$ is set (Line 7).

Once, the query graph has been constructed, the features that characterize the required social composition need to be extracted. Using the metrics of Table 2 and the query graph given in Fig. 3(b), one would get the features $num\_nodes = 4$, $num\_d = 3$, $num\_t = 1$, $num\_links = 5$, $avg\_nodedeg = 2.5$, $num\_hubs = 1$ (here, *avg_trust* and *maturity* are omitted). A query that is basically an *ordered*

Algorithm 3.1 Build query graph from techn. dependencies.

```
1: Input: artifact dependencies G_A = (V_A, E_A)
2: Output: social dependency query graph G_Q = (V_Q, E_Q)
3: for each x ∈ V_A do
4:    u ← createNodeTemplate(x.Type)
5:    addNode(V_Q, u)
6: end for
7: for each x ∈ V_A do
8:    u ← getCorrespondingNodeTemplate(V_Q,x)
9:    for each y ∈ V_A do
10:      v ← getCorrespondingNodeTemplate(V_Q,y)
11:      τ(u, v) ← mapFromTechnicalDependencyType(x,y)
12:   end for
13: end for
14: return G_Q
```


set of these criteria, is issued by comparing these query graph properties with the feature index. This is an approximate querying mechanism; i.e. because features of the query graph match features of the index graphs does not mean that query and result graphs are structurally identical. However, the more features match the higher is the probability that resulting graphs match the query.

In case no matching subgraph is found, two mechanisms can be applied:

- *Query relaxation*[22]: is a mechanism that subsequently removes less important features from the query until matching results are found; for instance, *num_nodes* must hold in order to have a workforce of appropriate size, but the distribution among roles *num_*{*roles*} is relaxed.
- *Subgraph fusion*: if no single subgraph satisfies the query, a result can be constructed out of two subgraphs, e.g. construct a larger workforce composed of two smaller social compositions. In that case, overlapping nodes between these two subgraphs ensure proper interlinks in the final result. A heuristic is required to realize this feature, which is out of scope of this paper.

Notice, our approach focuses only on the discovery of appropriate and approved social compositions for a given problem. However, further negotiation with actors are required prior to starting a collaboration, e.g. accounting for their free capacities, personal constraints, and rewarding.

3.4. *Activity-based indexing algorithm*

The further presented indexing algorithm (Algorithm 3.2) basically operates on a list of finished activities. First, for each activity the list of involved members is extracted from the activity structure (Line 4). Since members are sometimes officially involved in activities but actually inactive (since other actors take over their responsibilities), we remove all crowd members whose amount of performed actions is less than ϑ_{inv} (Line 6). Then, mutual `knows`-relations from FOAF profiles determine potential social dependencies in context of the finished activity.

Algorithm 3.2 Basic periodic index update.

```
1: Input: list of activities 𝒜, index matrix ℳ
2: /* extract one social graph per finished activity */
3: for each a ∈ 𝒜 do
4:    𝒢_i ← createGraph(a.InvolvedMembers)
5:    /* remove officially involved but actually inactive members */
6:    for each u ∈ 𝒱_i do
7:       if |a.actions(u)| / |a.actions(any)|< ϑ_inv then
8:          removeNode(𝒢_i,u)
9:       end if
10:   end for
11:   /*annotate strength of social relations */
12:   for each u ∈ 𝒱_i do
13:      for each v ∈ 𝒱_i do
14:         if ∃ knows(u, v) ∧ ∃ knows(v, u) then
15:            τ(u, v, ) = updateMetrics(u, v, ℰ_i)
16:            if τ(u, v, ) > ϑ_τ then
17:               ℰ_i = ℰ_i ∪ {e(u, v)}
18:            end if
19:         end if
20:      end for
21:   end for
22:   /* further annotations to 𝒢_i */
23:   addOriginActivity(𝒢_i, a.id)
24:   setUpdateTimestamp(𝒢_i)
25:   /* social graph registration in index ℳ */
26:   ℱ ← extractFeatures(𝒢_i, featurelist)
27:   /*... apply alternative template-based filtering here (see next) */
28:   if 𝒢_i ∈ ℳ then
29:      updateIndexEntry(𝒢_i, ℱ, ℳ)
30:   else
31:      createIndexEntry(𝒢_i, ℱ, ℳ)
32:   end if
33: end for
```


However, if these social relations actually have been relevant, i.e. interactions along these relations have been performed, needs to be verified by metrics gathered from interaction mining. Thus, if social trust $\tau(u, v)$ is above a certain threshold ϑ_τ, this relation is considered important for the success of this collaboration. Otherwise, i.e. no interactions occurred between u and v or trust is low, the relation is not considered for the social composition $\mathcal{G}_i$ (Line 15). Afterwards, beginning with Line 22, $\mathcal{G}_i$ is "tagged" with two further properties: (i) its origin(s) (activity ids), and (ii) a timestamp to capture the up-to-dateness of this composition. Finally values of predefined features ($featurelist$) are extracted that describe this graph (see Table 2) and either a new entry in the index matrix created or an existing entry updated (Line 25). An entry already exists, if it contains exactly the same members (identified by their `uris`) and connections between them. This means that there are no two subgraphs with identical nodes and edges in the index.

Template Mechanism. If features of recognized social compositions largely differ (e.g. in terms of roles and trust values) than numerous unique compositions are registered. In order to avoid that phenomenon (i.e. to be able to provide various alternatives to a given search query), subgraphs can be registered according to predefined common *social graph templates* (Algorithm 3.3). That means the features of a social composition $\mathcal{G}_i$ are tested against predefined templates, i.e. compositions that frequently occur or whose features have been recognized as highly requested (e.g. from query log analysis[27]) while further properties are neglected.

Algorithm 3.3 shows the basic principle. First, all features $\mathcal{F}$ of a social composition $\mathcal{G}_i$ are extracted. Then, these features are tested against a list of predefined temples $\mathcal{T}$ that characterize the desired features (Line 6). The algorithm counts the number of matching features of a composition $\mathcal{G}_i$ and template $t \in \mathcal{T}$. If, finally, most features of a template t are covered by $\mathcal{G}_i$, this subgraph is added to the index (Line 13). Otherwise, the recognized composition is not of interest and skipped.

Pruning. Of course, social compositions of crowd members that do not frequently perform activities need to be removed from the index. For that purpose, we apply a self-pruning index mechanism[28] (Algorithm 3.4) that recognizes whether the last update of an index entry due to a finished activity is older than a predefined threshold (ϑ_{age}). In that case, the outdated $\mathcal{G}_i$ is removed (Line 2). However, *mature* and long-term settled compositions that were involved in an extraordinary amount of activities (ϑ_{act}) may remain in the index (Line 6).

4. Evaluation and Discussion

Since we have not applied our approach and implementation in large-scale environments but like to evaluate its feasibility and applicability (e.g. of the indexing algorithm), we need to utilize either synthetic data or data sets from other domains. In order to set up a realistic setting, we study the properties of free and open

Algorithm 3.3 Template-based filtering for index updates.

```
 1: Input: list of graph templates T, social composition G_i
 2: F ← extractFeatures(G_i, featurelist)
 3: /* check each registered template */
 4: for each t ∈ T do
 5:    featurematch ← 0
 6:    for each f ∈ F do
 7:       /* single feature match */
 8:       if match(t, f) then
 9:          featurematch ← featurematch + 1
10:       end if
11:    end for
12:    /* G_i matches (nearly) all template feature */
13:    if featurematch/numFeatures(t) ≈ 1 then
14:       /* add/update G_i in M here ... */
15:    end if
16: end for
```

Algorithm 3.4 Self-pruning index mechanism.

```
1: Input: index matrix M
2: /* inspect each registered subgraph */
3: for each G_i ∈ M do
4:    /* if composition not used recently (aged out) */
5:    if currentTime - getUpdateTimestamp(G_i) > ϑ_age then
6:       /* and if no long-term settled composition */
7:       if getNumOriginActivities(G_i) < ϑ_act then
8:          deleteIndexEntry(G_i, M)
9:       end if
10:   end if
11: end for
```


source software development communities,[29,30] such as the *SourceForge Research Data Archive* (SRDA)[31] to create a synthetic collaboration network graph consisting of users with certain roles having interrelations and being involved in sets of activities. Although our social composition management and discovery approach was designed to be applied in private crowds, i.e. within enterprise boundaries, we argue that the SRDA is still a valuable basis for evaluation (considering the scale and fundamental nature of captured collaborations), especially with some modifications and corrections as further elaborated in this section.

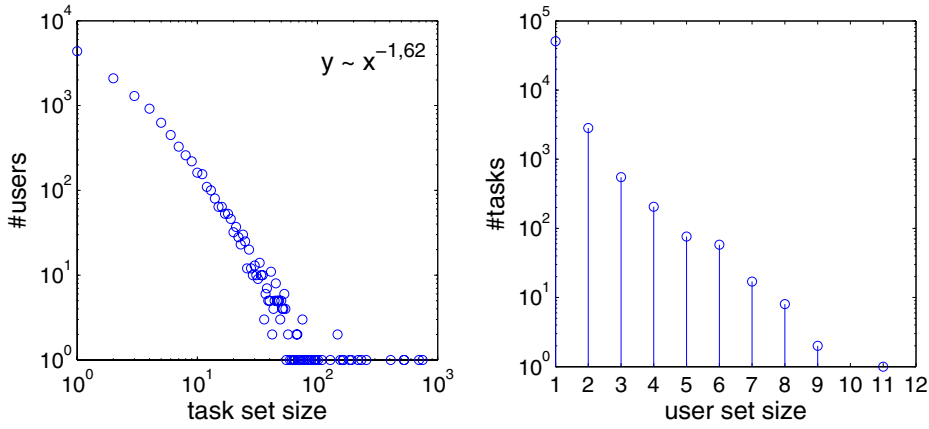
4.1. *SRDA data set properties*

The Sourceforge[d] platform provides a free management and development infrastructure for open source projects, including CVS repositories, mailing lists, discussion forums, and task management and tracking — just to name a few. The SRDA consists of collected data from the SourceForge community. We analyze SRDA tables[e] from January 2011 regarding (i) *project tasks and structures*: `project_assigned_to` and `project_task`, (ii) *discussion forums:* `forum` and `forum_group_list`, (iii) *artifacts:* `artifact`, `artifact_message` and `artifact_category`. Furthermore, we correlate user actions and user roles using prior work from Ref. 32. Here, user actions also include CVS operations. Analyzing and harnessing real data, combined with synthetic data under feasible assumptions allows us to create a realistic scenario of large-scale crowd-sourced software development.

4.1.1. *Activity structures*

Analyzing SRDA enables us to create a realistic setting in terms of activity sizes (i.e. number of involved members[f]), number of activities where one user is involved

[d] `http://sourceforge.net`
[e] online Wiki: `http://srda.cse.nd.edu/mediawiki`
[f] Notice, we removed all anonymous users (with id=100) from the data set, because there is no way to distinguish between different anonymous users and this hinders us to create a realistic social network later.



(a) User effort (b) Task size

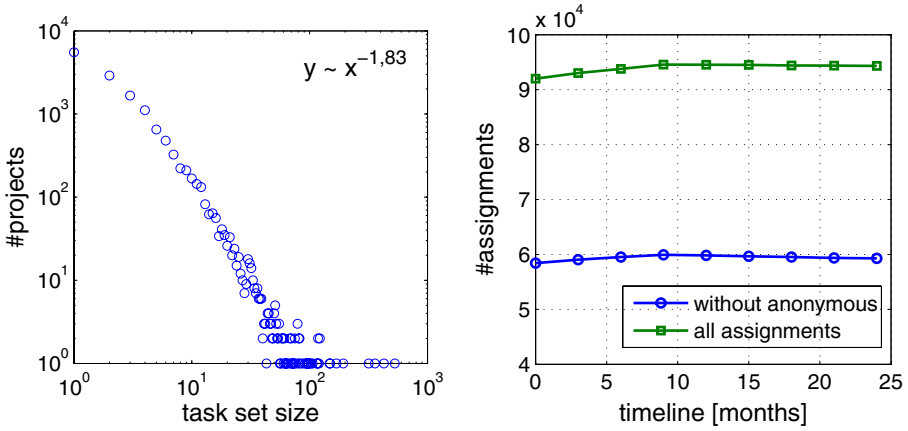
Fig. 6. Community structures on task level: (a) user effort in terms of involved tasks; (b) task size in terms of involved users.

at the same time, as well as the distribution of activities among projects. We create one activity in our system for each single task in the data set. In order to show the feasibility of this approach, we study the properties of tasks and show some characteristics in Fig. 6.

In detail, Fig. 6(a) visualizes the typical number of tasks (task set size) where a single user is involved. From 11,915 users, 4,369 are involved in only one task, 2,098 in two tasks, and 1,297 in three tasks. There are around 45 users who are involved in more than 50 tasks. The rest is distributed as shown in the figure. Figure 6(b) shows a different perspective, in particular the number of tasks having a particular user set size (i.e. team) assigned. Obviously, most tasks (50,758 of 54,500) are performed by single users only, while there is no task where more than 11 users are involved. Due to that reason, we further analyze team structures on project levels by aggregating tasks.

Figure 7(a) shows the distribution of tasks on project level. From 14,285 analyzed projects, there are 5,518 that consist of only one task, 2,901 with two tasks, and 1,669 with three tasks. On the other side, the largest project consists of 527 tasks. Figure 7(b) demonstrates that the number of newly created task assignments over an observed timeframe of 18 months remains nearly constant at around 59,000 and 94,000 (including assignments to anonymous users) respectively.

We conclude that for around 12,000 users, we will create around 4,000 activity entries (corresponding to tasks having more than one user in SRDA) consisting of two to 11 users, and assigned to 500 projects (having more than one task with more than one involved user) with a distribution as discussed before.

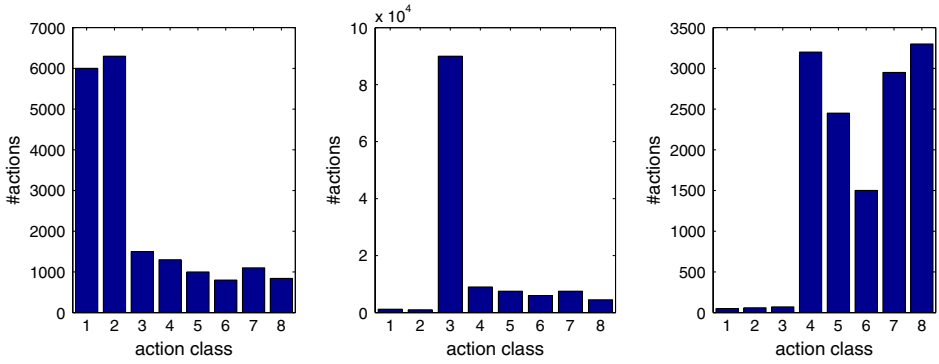


(a) Project size (b) Temporal evolution

Fig. 7. Community structures on project level: (a) project size in terms of number of tasks; (b) temporal evolution of created task assignments in a 18 months timeframe.

4.1.2. *Role mining*

Implicit user roles describe a user's typical collaboration behavior and focus based on performed actions. Roles are an inherent concept of our composition discovery approach. However, since user roles are *not* explicitly set in SourceForge (neither statically in user profiles nor dynamically at project setup), we need mechanisms to identify user roles by analyzing the SRDA data set. For that purpose we study experiments first performed by Ref. 32 (and described in greater detail in Ref. 33). In short, users (of the largest and most active projects) are clustered based on actions they perform on the SourceForge platform. Here, we only study a small subset of available actions, in particular, (i) create a new forum message, (ii) create a follow up forum message, (iii) modify a project (adding/removing members, changing permissions etc.), (iv) check out source code from CVS repository, (v) add source code to CVS repository, (vi) remove code, (vii) modify code, and (viii) update code. There are much more actions available,[32] including tracking bugs, releasing files and submitting patches, feature requests, artifacts, todos, etc. A clustering approach is used to identify similarities of action distributions among users. Finally, around 40 member custers are identified (depending on the thresholds in the clustering algorithm[32]), which describe around seven different roles. The two largest clusters, containing software users — who check out software and discuss bugs and features — and project managers, contain around 75% of all users. However, there are clearly other roles, such as pure software developers, task managers and bug reporters.[33] Figure 8 visualizes action distributions (i.e. the number of performed actions by all users in the corresponding cluster) for the three major roles software user, project administrator, and software developer.



(a) sw user (b) Project admin (c) sw developer

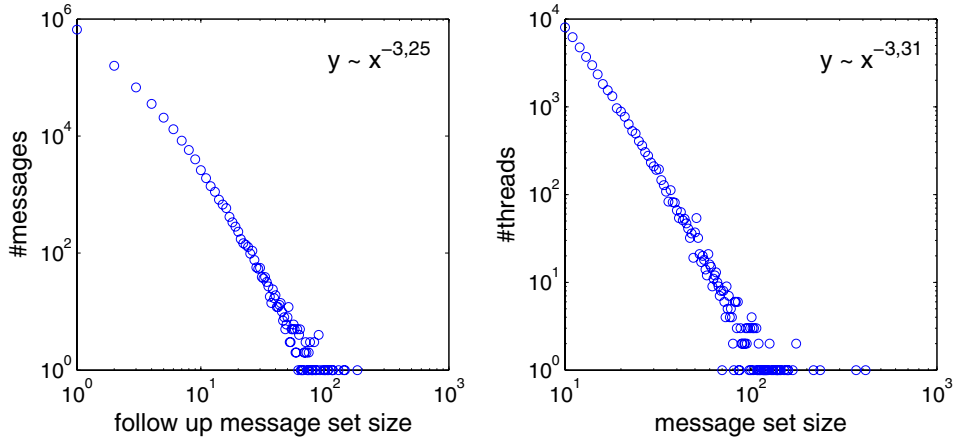
Fig. 8. Action distribution in clusters of various user roles.

We conclude that in our crowdsourcing environment around 41% of members will be software users (i.e. general experts that drive the further development by testing prototpyes and discussing issues in forums), 34% project administrators, and 17% pure developers. The rest (8%) does not fit into these roles.

4.1.3. *Interaction data*

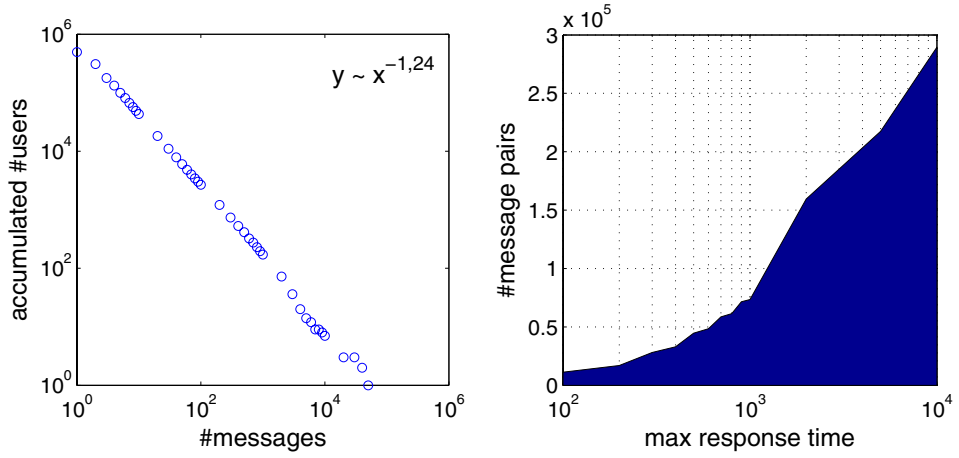
Interactions on SourceForge may take place over various channels. However, not all of them can be easily captured or are included in the SRDA. For instance, while subscriptions to mailing lists are part of the data set, actually sent e-mails are not. Therefore, we utilize two different interaction channels: (i) forum messages, and (ii) artifact messages.

Threaded Forum. The forum allows to discuss missing features, bugs or further development in a hierarchical manner. That means to every message a so-called follow-up message can be posted. The data set consists of 31,67,605 captured messages (and 51,91,629 if including posts by anonymous users respectively). We count 4,94,302 distinct forum posters. Figure 9 shows some basic characteristic of the forum. In particular, Fig. 9(a) visualizes the distribution of the set size of follow-up messages on the same hierarchical layer; in other words, the number of posts attracted by one particular message. Figure 9(b) shows thread sizes by aggregating all messages on all levels beginning with the top message in one message tree. From 9,85,838 messages that have at least one follow-up (i.e. not being an unanswered post), 11,857 have equal or more than 10 follow-ups, from that set 915 messages have more than 25 follow-ups. Regarding the thread size, we investigated thread structures with at least 10 messages in order to capture serious discussion effort. There are 40,830 threads that fulfill this criterion. From this set 3,887 threads consist of more than 25 messages, 509 have more than 50 messages, and 125 more than 75. The largest thread consists of 414 messages (without anonymous posts). We conclude that in SourceForge discussions are typically focused, i.e. a message does



(a) Follow up messages (b) Thread size

Fig. 9. Forum structure: (a) number of follow up posts to one message; (b) typical number of messages in one thread.



(a) User effort (b) Response time (seconds)

Fig. 10. User involvement in forums: (a) number of messages posted by users; (b) typical response times of users (up to 10,000 s).

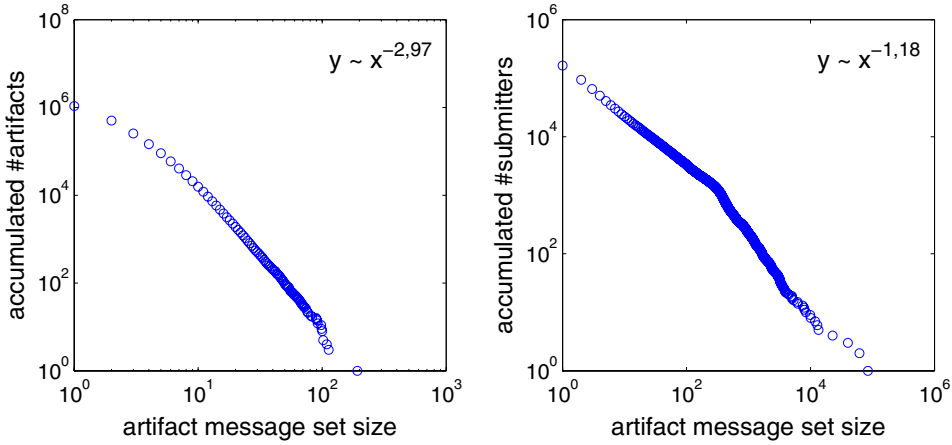
not have dozens of follow-ups, however, deep thread structures emerge, e.g. due to controversy on certain issues.

Figure 10 provides information about general user behavior. In detail, Fig. 10(a) shows the number of posts per user. For better readability — the number of posts per user highly varies from 1 to 90,071 — users are aggregated on the y-axis. That means, the figure shows on the y-axis the number of users who posted more than the given amount of messages on the x-axis. There are 43,302 users who posted more than 10 messages, 6,001 with more than 50 messages, and 2,679 users with more

than 100 messages. Thus, serious analysis is only possible with a small fraction of the whole user base. We furthermore investigated response times in Internet forums. Since we do not know which message is a question and which one a comment, we simply capture the differences of timestamps between each message and its follow-up (iff there is one). Figure 10(b) reflects the responsiveness of forum posters by categorizing message pairs according to their posting times. Response times are given in seconds. Note the logarithmic scale.

Artifact Messages. A second source of interaction data are artifact messages. SourceForge users can "attach" messages and comments to artifacts of various types. Overall, there are 10,76,517 artifacts where at least one message is assigned in the studied time span. In sum, 1,64,429 distinct users submitted 23,05,702 messages (29,91,274 if including artifact messages by anonymous users). Figure 11(a) shows the accumulated number of artifacts for various message set sizes. As shown in this cumulative diagram, there are around 15,654 artifacts with equal to or more than 10 messages attached (thus, really collaboratively processed artifacts), and 96 artifacts with equal to or more than 50 messages. More than 50% of artifacts have only one message assigned.

Figure 11(b) displays a cumulative submitter perspective; i.e. the number of message submitters on the y-axis that submitted more messages than the message set size given on the x-axis. For instance, from the full set of 1,64,429 distinct submitters, 22,214 submitted equal or more than 10 artifact messages, 3,306 users more than 100 messages, and 225 users more than 1,000 messages. Approximately 43% of all users submitted only a single artifact message. Since artifact messages are sparsely distributed over a large set of artifacts, we assume an artifact message as a kind of point-to-multi-point communication, where each single message addresses



(a) Amount of message assignments (b) Submitter effort

Fig. 11. Artifact message and submitter distributions.

all users who submitted messages to the same artifact. This approach, instead of a point-to-point model, enables us to infer still meaningful metrics that quantify users' relations in our proposed socio-computational crowdsourcing model.

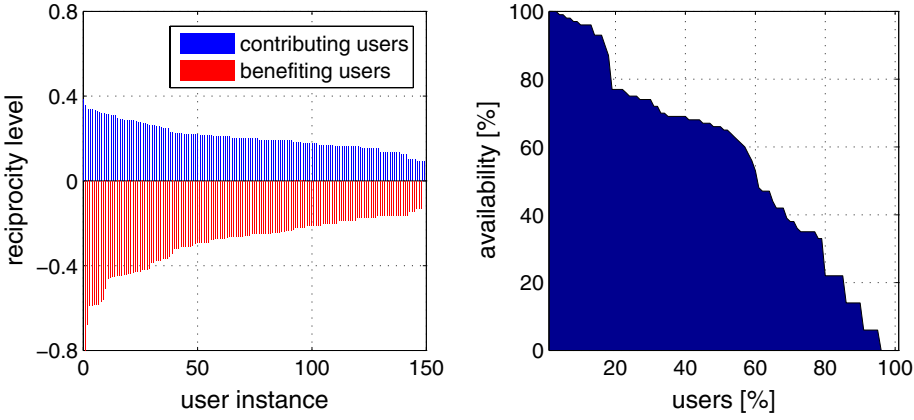
4.2. *Collaborative crowd environment setup*

After extensively analyzing a real Web-based large-scale software development environment, we create a synthetic crowdsourcing environment that reflects attributes from SRDA in terms of (i) *structural properties* such as number and sizes of activities, and (ii) *dynamic interactions* such as exchanged messages and performed task assignments. Furthermore, we assign the roles of software developers, software users (aka testers) and project administrators to the crowd members using exactly the same distribution as given in the SRDA data set. The following section deals with metric calculation as defined in Sec. 3. In particular, we demonstrate the calculation of *reciprocity*, *availability* and *responsiveness* based on SRDA data.

Reciprocity. The SourceForge data set offers two valuable properties to calculate reciprocity, i.e. the amount of obtained support from the community compared to the amount of provided support. First, tasks are created by one person and may be assigned to another one. In total, there are 11,915 users involved in task processing; 10,613 of them assign tasks to others than themselves. In sum, 94,308 task assignments have been captured. By removing assignments from anonymous users and also self-assignments, only 15,207 tasks remain to be analyzed. Finally, there are 5,941 unique user pairs, i.e. a mapping from one assigner to another assignee. From that amount, 3151 distinct users assign tasks to 5335 other users.

A second source of information is the Internet forum. Our assumption is that a top post, i.e. the first post in a thread, is usually a question or support request (except announcements which however are mostly unreplied). All further replies are attempts to address this request. Thus, the poster of the first message obtains some help from the community, while repliers provide some support. Using the forum mining algorithm from Ref. 34,[g] we calculate each user's contribution score and subsequently reciprocity. If a user obtains more support from the community, i.e. posts many top messages but replies less or assigns many tasks but processes only a few, than this value is negative. Figure 12(a) shows the distribution of reciprocity for top-150 contributing and top-150 benefiting users. Notice, project administrators have negative reciprocity by nature (e.g. it is part of their role to assign tasks to others). Thus, we calculate reciprocity values only between members with the same roles, in particular, software users, project administrators, and software developers.

[g]Notice, since there are no frequently discussing distinct user pairs, we calculate reciprocity not for personal relations between two particular users, but for users with respect to the whole community. This further reduces the computational complexity from around $\mathcal{O}(n^2)$ — one potential link between every pair of nodes — to $\mathcal{O}(n)$. However, in our proposed socio-computational crowd environment, interactions would not be performed in a public manner but addressed directly to receivers, thus, metrics would be calculated for personalized links.



(a) Reciprocity distribution (b) Availability distribution

Fig. 12. Metric distribution: (a) *reciprocity*, (b) *availability*.

Overall, there are few users who benefit very much, while on the other side the majority of users contribute a little. Accumulating all reciprocity values of all users results in ≈ 0.

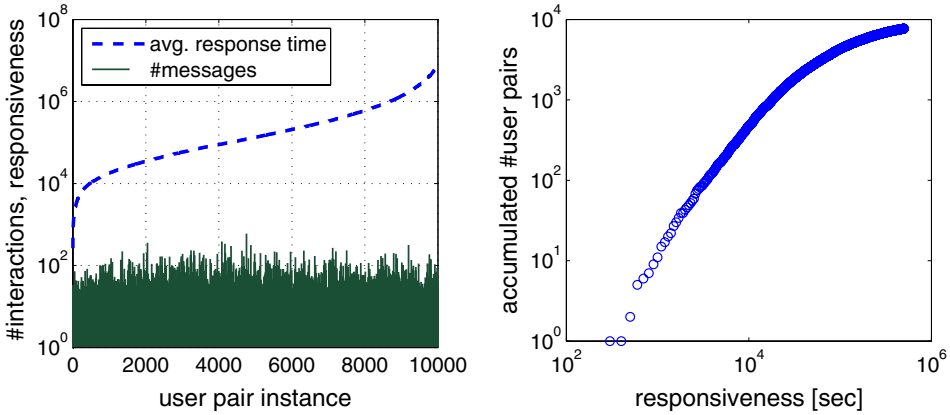
Availability. Users submit artifacts of various types to the SourceForge platform. However, they can be assigned to and finally closed by other users. Thus, an artifacts ownership and relation to users is described by a triple $\langle submitted\_by, assigned\_to, closed\_by \rangle$. In the studied time span, there are 4,22,443 artifacts that have not been submitted by, assigned to, or closed by anonymous users (in sum, there are 22,92,054 artifacts on the platform). From that set, 3,01,095 artifacts are submitted by and assigned to two different users. We assume that artifacts who are submitted and closed by the same user ($\approx 11.9\%$) are processed successfully (at least if the status is set to closed), and thus the assigned user was available to process a given task. In case the submitter and closing user are two different persons ($\approx 70.3\%$), the assigned user was available for collaboration if s/he assigned at least one message to the corresponding artifact. For the rest, we consider artifacts with state closed or pending as success, others as failed.[h] Figure 12(b) depicts the *minimum* availability in percent for a certain amount of users (in percent from the whole population). Note, we only considered a small user base of 250 users, since for larger populations availability could not be calculated seriously. There is simply not enough data to prove availability with smaller amounts of interactions. Basically, in this figure there are two buckles, one where availability drops from around 95% to around 75% and another one where the same happens from around 65% to 40%. This effect seems to be caused by highly varying numbers of captured interactions

[h]Of course, we cannot prove that these assumptions are correct in all cases, however, given the massive amount of data we argue that the trend of handling data this way is feasible.

(i.e. artifact assignments). We conclude that for the availability metric, a larger pool of personal interaction is required to calculate stable and reliable values.

Responsiveness. In order to calculate this metric for SourceForge community members, we utilize once more the Internet forum. In contrast to reciprocity calculation, here we can partly use posts from anonymous users too. In particular, we consider how fast distinct users reply to anonymously posted messages. We proceed as follows: First, all final answers (i.e. unreplied follow-up posts) from anonymous users are removed from the data set. Then, we determine unique[i] user pairs, count how often they replied to each other's posts, and study those, who had at least 10 interactions. Doing so, there remain 10,189 user pairs of which 1,780 had at least 25 interaction, 491 at least 50 interactions, and 131 equal or more than 100 interactions. The number of posts and the average of response times do virtually not correlate (Pearson correlation coefficient of -0.024). Figure 13(a) shows average response time values and corresponding number of exchanged messages. Figure 13(b) deals in more detail with the responsiveness values of the fastest replying users. Notice, here we record user pairs. Thus, there are 11 user pairs who have an average response time below 1,000 s ($\approx$ 17 min), 471 with response times below 10,000 s ($\approx$ 2.8 h), and 4,307 below 1,00,000 s (slightly above one *day*). We do not use other data sources for responsiveness calculation, such as task assignments, since processing times (and thus response times) highly vary according to task complexity.

Social Trust. Various approaches exist to infer social trust values from captured behavior metrics. Here we use normalization and weighted average, i.e. metric values



(a) Responsiveness vs. #messages

(b) Responsiveness distribution

Fig. 13. Metric distribution: (a) *responsiveness vs. number of messages*, (b) *average response time*.

[i]Posts from anonymous users are assigned to *one* virtual user. This method does not distort the measurement since we calculate average values only (and do not sum up contributions such as for reciprocity).

are normalized, e.g. to fit the interval [0,1] and are than combined with predefined weights (all three metrics (recpr, avail, resp) use $\frac{1}{3}$ weight/impact). Alternatively, more sophisticated approaches, including rule-based aggregation and fuzzy set theory[8] can be applied. The final outcome is a scale-free social trust network, as visualized in Fig. 14. The graph's properties regarding degree distribution and connectivity match attributes of common collaborative communities as investigated by Ref. 35. In particular, degree distributions of such networks follow power laws with degree exponents between 2.1 and 2.5; here, for our network created from SourceForge data we calculated an exponent of 2.10, thus matching overall expectations.

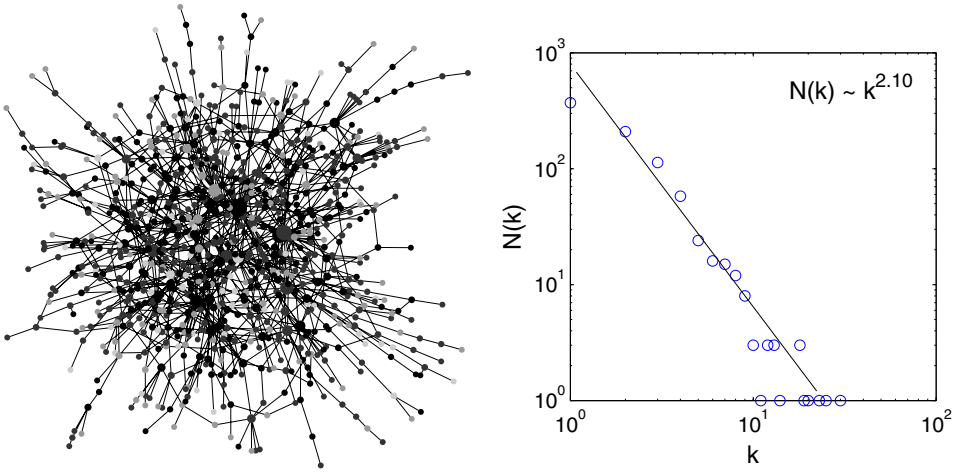
4.3. *Experiments and results*

The basic aims of these experiments are (i) to demonstrate the *feasibility* of our dependency management approach, and (ii) to measure the *scalability* and performance of the prototype implementation.

4.3.1. *Scale of social network management*

The first step in our evaluation approach is to *create a synthetic network that has realistic properties* (in terms of size, node degree, member roles, activity involvements etc.) extracted from real community data as discussed before. For that purpose we use the following model:

(i) Create 12,000 user instances (**nodes**).
(ii) Assign **roles**: software users (41%), project admins (34%), software developers (17%), undefined (8%).



(a) Graph view (b) Degree distribution

Fig. 14. Created social trust network (reduced view for 1,000 nodes).

(iii) Create list of **activities** with 4,000 entries.
(iv) Create **links** between users according to SourceForge data set, i.e. distribution of link strength between users of same roles.
(v) Partition users in **groups** (subgraphs) consisting of 2 to 11 nodes; $\approx \frac{1}{3}$ having 2-3 users, $\approx \frac{1}{3}$ having 4-6 users, $\approx \frac{1}{3}$ having > 7 users.
(vi) **Assign groups** to activities.
(vii) Structure activities in **projects**.
(viii) Create **FOAF profiles** that are processed by our system.

Properties of the resulting graph are summarized in Table 3. Metric definitions follow common standards as further explained by the utilized software tool *Network Analyzer*.[j] This overview shows the complexity[k] of typical networks that our algorithms will have to cope with.

In order to utilize the created graph[l] for the evaluation of our prototype implementation for managing and discovering member compositions, we create a FOAF profile[m] for every single user, containing his/her name (`lastname`), role (`Group`), relations to collaboration partners (`knows`) extended with a trust value, and activity involvement (`currentProject` and `pastProject` respectively). This process is supported by the *Jena Semantic Web Framework*.[n]

4.3.2. *Basic graph construction and indexing*

We discuss performance aspects of our proposed indexing approach as defined by Algorithm 3.2. Notice, this algorithm operated on top of the previously created synthetic collaboration network. The performance of interaction log analysis, metric

Table 3. Complexity of created network.

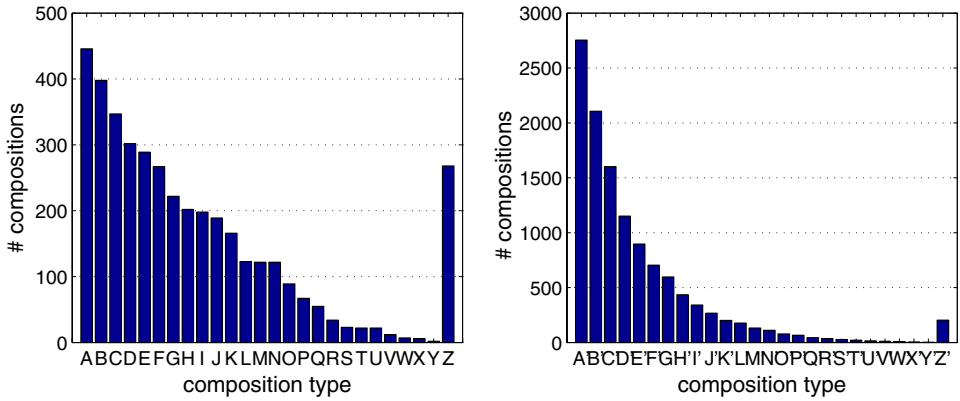
Social network metric	Value
Number of nodes	12,000
Number of edges	23,976
Connected components	1
Clustering coefficient	0.001
Network radius	6
Network diameter	9
Network centralization	0.016
Characteristic path length	5.381
Avg. number of neighbors	3.996

[j] `http://med.bioinf.mpi-inf.mpg.de/netanalyzer/`
[k] Notice, the mentioned software tool required 7,969 seconds to calculate the given properties on a Pentium D with 3.0 GHz.
[l] Here we use the *Java Universal Network/Graph Framework (JUNG)* available at `http://jung.sourceforge.net`.
[m] The related FOAF concept is highlighted with a `typewriter` font.
[n] `http://jena.sourceforge.net`

calculations and trust inference is not in scope of this work but has been extensively studied in Ref. 8. Here we traverse a list of 4,000 activities and index social compositions by determining the predefined features[o] of Table 2.

We run the indexing algorithm two times: (i) *Activity-centric indexing*: applies the algorithm as defined where for each finished activity the features of the corresponding social composition are extracted. (ii) *Node-centric indexing*: refers to the creation of one virtual activity per user and adding all node's neighbors in advance. In fact, here the surrounding social network of each node (and from its individual perspective), potentially emerged from numerous activity involvements, is indexed, rather than social compositions from a third person's view.

For **activity-centric indexing** the result are 4,000 social compositions (one per activity), which are grouped according to equality of features. Figure 15(a) depicts the size of clusters and their distribution. Here, the 25 most common compositions are clustered and labeled (A-Y), while further 268 of 4,000 compositions do not fit into these schemes (lable Z). Table 4 shows the top-5 occurring social compositions



(a) Activity-centric indexing (b) Node-centric indexing

Fig. 15. Composition detection frequency for different indexing approaches.

Table 4. Most frequent compositions (in sum ≈ 44.5% of all compositions).

Type	num_{role}	num_links	avg_nodedeg	avg_trust	Count
A	2 su	1	1	[0.25,0.5[	446
B	1 su, 1 pa	1	1	[0.25,0.5[	398
C	2 su	1	1	[0.5,0.75[	347
D	2 su, 1 pa	2	1.5	[0.25,0.5[	302
E	1 su, 1 sd, 1 pa	2	1.5	[0.5,0.75[	289

[o]Notice, we skip the calculation of the *maturity* feature since social compositions are rarely reapplied in SourceForge and thus in our synthetic model. However, we argue that when using our system from the beginning, this feature will have major impact, especially when querying for well-trained and frequently applied social compositions.

and some relevant features, i.e. num_{role},[p] num_links, avg_nodedeg, avg_trust.[q] Notice, that smaller compositions occur of course more often.

For **node-centric indexing** results are depicted in Fig. 15(b). In this case, a higher amount of activities (i.e. 12,000 matching to the number of nodes) is analyzed. The results show compositions (A'-Z') from each user's point of view (as given, for instance, in Fig. 5). In other words, the results are the decompositions as reflected by the single FOAF profiles of network members. Since social structures from a node's perspective include only direct relations and hence are not as complex[r] as in the previous case (again see e.g. Fig. 5), only 256 (of 12,000) social compositions do not fit into one of the created 25 clusters. This situation would significantly change, if we did not only consider a node's direct relations, but also neighbors of neighbors (a kind of recommendation mechanism.)

We discuss the **performance** of these approaches from an abstract perspective, in terms of number of invocations of core services (infrastructure discussed later), and number of database accesses, because real run-time performance varies depending on numerous impacts, such as network latency, processor speed and load, and memory consumption. Table 5 compares the effort of applying the two discussed indexing mechanisms. In general, activity-centric indexing [see e.g. count(i)] results in less but larger (partly unique) compositions compared to node-centric indexing [see e.g. count(ii)] which produces more, but less complex social compositions.

After performing this set of experiments with basic indexing, we motivate the application of template-based indexing, as discussed earlier in this work, which allows to:

- group social compositions more efficiently, e.g. group actually different but in terms of features quite similar compositions.
- better support the discovery of frequently requested compositions (e.g. matching to often issued queries)

Table 5. Performance in terms of number of calls and resulting index complexity for (i) activity-centric indexing and (ii) node-centric indexing.

	Measurement	Count (i)	Count (ii)
infrastructure	#ActivityServiceAccesses	4 000	12 000
	#UserDBAccesses	59 818	47 592
	#SocialTrustNetworkAccesses	23 976	23 976
	#FOAFProfileAccesses (public)	12 000	12 000
	#RegistryDBAccesses	25 888	51 684
data	#CompositionsRegistered	25 888	51 684
	#IndexNetworkCoverage (percent)	93.3%	98.3%

[p]su=software user, sd=software developer, pa=project admin.
[q]measured in intervals, otherwise compositions could not be grouped.
[r]Notice, here the average degree of a node is 3.996.

- cut the long tail of the distribution in Fig. 15 which provides little value to most users but causes significant management overhead (e.g. size of index)
- recognize sub-subgraphs, e.g. a social composition which performed a set of activities could be split into two frequently requested compositions and thus registered multiple times.

4.3.3. *Template-based composition indexing*

We pick the most recognized social compositions types A-J (see top-five in Table 4) as templates and re-run indexing Algorithm 3.2, but additionally apply Algorithm 3.3. We configure the indexing process to categorize recognized social compositions by comparing the four features $num\_\{role\}$, num_links, $avg\_nodedeg$, and avg_trust as used in Table 4 before. Figure 16 visualizes the results. Given our data set, we can recognize around 11% of social compositions with only one template, around 44.5% with five templates, and already 71.4% with 10 templates. We tested template-based indexing with up to 25 different templates with which we can categorize 93.3% of all occuring social compositions.

Submatching. Until now, we categorized social compositions $\mathcal{G}_i$ (extracted from activities) which exactly match a given template's t features; e.g. $num\_links(\mathcal{G}_i) = num\_links(t)$. Now, we further evaluate so called *submatches*. Here, we test each $\mathcal{G}_i$ against all available templates in order to recognize if a given composition fulfills *at least* a template's features; e.g. $num\_links(\mathcal{G}_i) \geq num\_links(t)$. As a result the more complex a recognized social composition is, the more often it can be decomposed in simpler submatches and registered multiple times. Figure 17 shows the results[s] for this experiment. In general, 2,825 social compositions (from 4,000

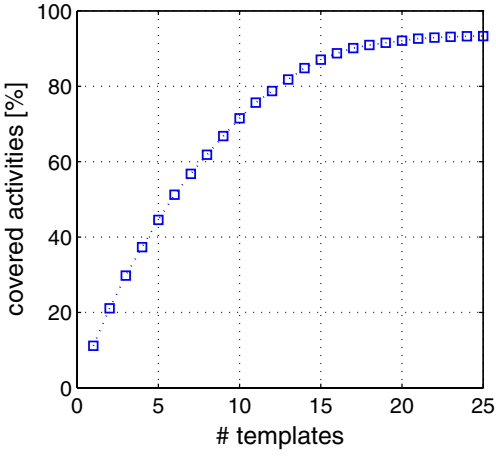


Fig. 16. Amount of covered activities compared to number of applied templates.

[s]Notice the logarithmic scale on the y-axis.

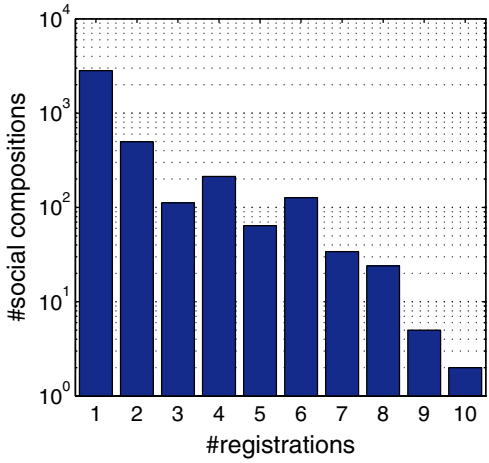
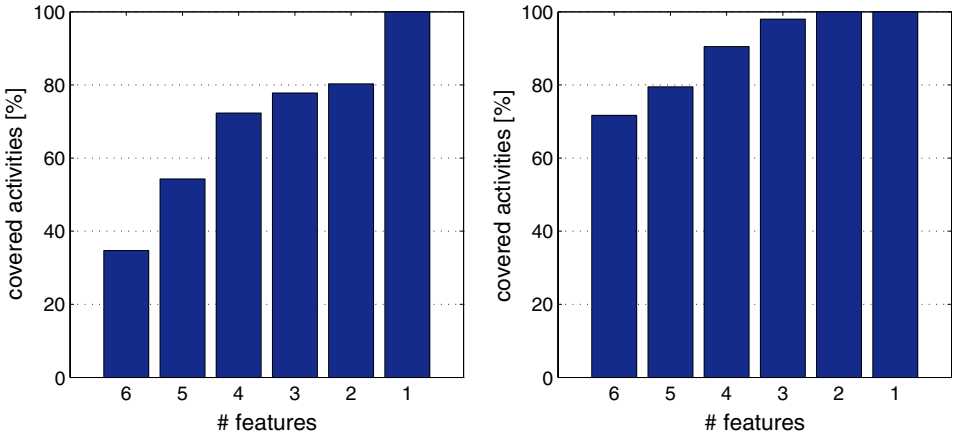


Fig. 17. Number of social compositions that match multiple templates (leading to several registrations).

activities) are registered only once (because those consist only of two nodes and therefore cannot match further relevant subgraphs); 497 compositions ($\approx 12.5\%$) are registered twice. Interestingly, the number of social compositions for higher number of registrations is not decreasing monotonic. For instance, there are more social compositions that are registered four times than three times. The reason is that there are numerous compositions consisting of three nodes (u, v, w), which can be additionally decomposed in three subgraphs consisting of only two nodes ((u, v), (u, w), (v, w)). Thus, in sum, a three-node-composition here is registered four times as long as other features (such as *avg_trust*) do match either.

Template Complexity. Finally, we investigate the impact of template structures and complexity on the indexing process. The question is, how does the amount of social compositions, which is recognized by the indexing process, change for differently complex template definitions. For that purpose we set the number of utilized templates to 10 (fixed) and just vary the number of impacting features. The whole feature list consists of (in this order): *num_nodes*, *num_*{*role*}, *num_links*, *avg_nodedeg*, *num_hubs*, and *avg_trust*. We begin with testing each social composition against each single template by considering all features. Then, we remove the last feature in the feature list (beginning with *avg_trust*) and run the indexing process again. Figure 18 visualizes the results for indexing with exact matches and indexing with submatches. As expected, the more features are tested the lower is the amount of matching social compositions. Furthermore, submatching leads to at least partly indexed structures of more complex social compositions as well as to multiple registrations. Therefore, even larger social compositions become indexed to some extent and the number of covered activities (and their corresponding social compositions respectively) in Fig. 18(b) does not increase as sharply as in Fig. 18(a). A careful *tradeoff between accuracy* of indexed compositions *and amount of covered*



(a) Exact matching (b) Submatching

Fig. 18. Amount of covered activities depending on number of compared template features.

activities by applying a fixed number of templates (here: 10) must be considered. For instance, when indexing by exact matching the *num_nodes* feature, 10 templates are sufficient to cover 100% of activities (matching $num\_nodes = [2, 11]$), because there are no corresponding social compositions with more than 11 users [compare Fig. 6(b)]. However, in that case one would only be able to query for social compositions with an appropriate size but neglecting other features, such as roles and trust links.

4.3.4. *Composition discovery and query relaxation*

For the final tests we create a reference index through template-based indexing with exact matching and using 25 templates which consider *all* features as defined in Table 2. This index covers 87% of activities and their corresponding social compositions respectively. Furthermore, we define three test queries $\mathbf{Q_1}$, $\mathbf{Q_2}$, and $\mathbf{Q_3}$. Query $\mathbf{Q_1}$ is designed to deliver a large amount of results, i.e. querying only for a pair of software users (su) with an average trust relation. Query $\mathbf{Q_2}$ looks up three users, with one hub node to which two other nodes are connected. Query $\mathbf{Q_3}$ defines the search for three users of different roles which are well interconnected. All three query definitions use features *num_nodes*, *num_*{*role*}, *num_links*, *avg_nodedeg*, *num_hubs*, and *avg_trust*. Notice, *maturity* is neglected here (symbol $*$ means any) which is not sufficiently reflected in our test data set. Table 6 shows the details.

Issuing the queries as given delivers result sets of sizes 446, 102, and 89 (Table 7). If the discovered social compositions are currently not available (e.g. actors are involved in other activities, index is outdated), a query relaxation mechanism can be applied in order to find further social compositions that potentially match one's needs. This mechanism gradually removes features from the query and therefore extends the result sets. Table 7 provides an overview of the results for the given

Table 6. Definition of test queries $\mathbf{Q_1}$, $\mathbf{Q_2}$, $\mathbf{Q_3}$.

Feature	$\mathbf{Q_1}$	$\mathbf{Q_2}$	$\mathbf{Q_3}$
num_nodes	2	3	3
num_{roles}	2 su	2 su, 1 pa	1 su, 1 sd, 1 pa
num_links	1	2	3
avg_nodedeg	1	1.5	2
num_hubs	2	1	3
avg_trust	[0.25,0.75]	[0.25,0.75]	[0.25,0.75]
maturity	*	*	*

Table 7. Query relaxation results.

γ	$\lfloor \mathbf{Q_1} \rfloor$	$\lceil \mathbf{Q_1} \rceil$	$\lfloor \mathbf{Q_2} \rfloor$	$\lceil \mathbf{Q_2} \rceil$	$\lfloor \mathbf{Q_3} \rfloor$	$\lceil \mathbf{Q_3} \rceil$
1	446	446	102	102	89	89
0.833	446	767	102	222	89	194
0.667	446	890	102	289	194	226
0.5	446	890	212	356	201	289
0.333	987	2825	282	472	222	312
0.167	1237	2825	385	534	250	501
Increase ($\gamma = 0.5$)	≈100%		107%–183%		126%–225%	

queries. The first column describes the relaxation factor $\gamma = \frac{\#appliedFeatures}{\#allFeatures}$ ($\gamma = 1$ means all features are used in the query). Then two columns, a lower bound and an upper bound, describe the size of the result set. There are two limits because depending on which feature is removed first, the sizes of result sets vary. For instance, removing *num_nodes* from $\mathbf{Q_1}$ has virtually no effect, since *num_{roles}* already strictly define that two software users are required. However, removing *avg_trust* dramatically extends the result set (here: from 446 to 767). The last row in the table summarizes how many more results are generated when neglecting half of the query features ($\gamma = 0.5$). In general, more complex queries profit more from query relaxation because they become considerably simpler. Again, query relaxation introduces fuzziness to the query results and its usefulness heavily depends on the use case, i.e. how strictly results must match to an issued query.

If a query delivers too many results, one can either restrict constraints, e.g. define more search attributes or tighter constraints,[t] or rank query results according to one or several metrics. For instance, ranking hundreds of elements in the result set according to average trust and selecting the top-ranked element is an intuitive use case to pick a certain social composition from a larger list.

4.3.5. *Web services-based implementation*

Service Infrastructure. From the technical (and implementation) point of view, we use a wide variety of state-of-the art Web (service) technologies. Crowd members

[t]Notice that the query mechanisms cannot only deal with the = operator, but also with < and > and their combination in order to define intervals.

(and even their relations) are represented by a large set of individual FOAF profiles that are initially created manually for (and ideally from) each node in the social network. Once registered, these profiles, especially `knows`-relations are then automatically updated based on captured interaction data as further discussed in Ref. 26. Using FOAF, we link these nodes to a list of activity identifiers that reflect respective community members' involvements in certain activity instances. Activities are managed in an external Activity Web service that implements the activity model as discussed earlier in this paper, and is hosted on Axis2. Since reading single FOAF profiles for each query for determining graph structures is time-consuming, we implement a dedicated SocialTrustNetwork Web service that manages an in-memory graph model (with a MySQL backend database to guarantee persistence) and whose *cache* is frequently updated from current FOAF profiles. Together with the the single FOAF profiles, the whole technical infrastructure is hosted on an *Apache Tomcat Web Server*. Potential end-users, i.e. people who query for social compositions, can utilize this infrastructure through *Java Portlets* that are hosted on a Liferay Community Server. These portlets support the configuration of the index management (e.g. defining template features), and the definition of queries (e.g. entering required subgraph feature values of indexed social compositions).

Performance Issues. Our implemented prototype uses a service-oriented backend with dedicated Web services for (i) periodically processing interaction logs to infer (and update) collaboration metrics and behavior (see details in Ref. 8, (ii) managing activities and joint task contexts (see details in Ref. 14), and (iii) managing subgraphs reflecting well-proven social compositions (i.e. create, read, update, delete index entries). Web services offer the great ability to facilitate interoperability and openness, since interfaces are well documented (as WSDL) and easily usable through established frameworks that can create stubs on demand and even perform flexible invocations. However, Web service calls are time-intensive (predominantly because of complex XML processing of SOAP messages) and often cause a major bottleneck in data-intensive applications. Furthermore, processing distributed FOAF profiles and creating/updating the social network graph is a time- and resource-intensive process. Numerous optimization methods can be applied to tackle performance issues, including caching of FOAF documents, bulk-transfers of profiles if applicable, and even bypassing Web services, e.g. here granting the indexing service direct access to back-end databases. Some optimization mechanisms for social network management, including WS caching mechanisms[8] and selective updates of relations,[26] have been discussed in our previous work. Here, the technical infrastructure itself is not in the focus of the work at hand.

Essentially, we utilize a heavily distributed service-based infrastructure which mitigates negative performance impacts through clustering. The actually critical issue is the dynamic usage of the system. The higher the number of created compositions and activities, and number of interactions per time interval, the higher is

the effort to keep the social network, and thus the index, up-to-date. These performance questions need to be clarified in future work that deals with a real world deployment in larger scales.

5. Background and Related Work

CSCW and Knowledge Management Systems. Numerous definitions of Computer-Supported Cooperative Work (CSCW) exist.[36,37] From quite technology-centric views, such as[38] "... addresses how collaborative activities and their coordination can be supported by means of computer systems", to more people/social centric views[39] "...the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques". One particular aim of this kind of software is the composition of effective teams. This so called staffing concept[40] was invented to create harmonic teams that fit to certain task requirements based on their knowledge and expertise reflected in database profiles. However, these staffing systems assume sophisticated knowledge management approaches[12,41] in order to properly represent member properties in the database. A distinct disadvantage of such systems is that they mostly rely on manual data input. In our work, we motivate the application of data mining approaches to update profiles automatically. Moreover, we do not apply the staffing concept for composing a static core team, but use an approach for flexibly discover (and integrate) supporting individuals (i.e. crowd members), who perform certain tasks in comparatively short time intervals. We argue that this aspect makes our approach different from traditional staffing systems, and thus justifies the introduction of a crowdsourcing terminology.

Crowdsourcing. From our understanding Web-based collaboration and crowdsourcing are the natural successors of CSCW software. Crowdsourcing applications[13,42,43] are online, distributed problem-solving and production models that have emerged in recent years. A vast number of registered individuals offer solutions to various problems and provide their workforce online. Crowdsourcing offers some distinct benefits such as multiple redundant workforces that can be utilized on demand;[44] and collective intelligence used to rate items and vote for best results.[45] Many research challenges remain related to the distributed and open nature of crowdsourcing. In this work, however, we assume a *private crowd* environment that is established by employees of large-scale enterprises.[46] Thus, some assumptions can be made and typical issues relaxed, such as the motivation of crowd members to participate in activities and sufficient skills and experience of actors. Since members of enterprises have been hired by human resource offices, there skills and working performance is known in advance and these are no issues in our proposed use cases.

First attempts towards service-oriented crowdsourcing of inter-connected technical artifacts have been elaborated in our previous work. In Ref. 47, we motivate the need for collaboration in crowdsourcing applications and introduce first ideas of required building blocks. In the current paper, we follow up on this approach,

and elaborate in detail a realistic agile software development use case, an advanced social composition discovery approach, and a detailed evaluation using real world data sets. In particular, we formulate the essential research challenges and contribute concrete analytical models and algorithms to solve the problem of matching technical structures to social structures. These contributions have not been presented before.

Socio-technical Dependencies in Software Engineering. Developing complex software systems, requires the involvement of large groups of software designer, developer and tester, and produces an extensive amount of technical artifacts, including, code, specifications, manuals, and reports.[48] As recognized by *Conway's Law*,[2] social structures reflect technical structures and vice versa. That means, there are strong similarities between the coupling of team members (social dependencies) and compositions of artifacts they produce (technical dependencies).[3,49] For instance, tighter coupled software modules require stronger coupled teams, since more technical dependencies demand for thorough coordination and alignment of work. Especially, when applying modern agile software development techniques with short incremental cycles the role of interpersonal interactions and social relations must be revisited. The impact of socio-technical dependencies has been considered to study new approaches on analyzing the fault-proneness of individual software components within a system[50] or to optimize structures of professional software development teams by learning from the organic formation of social structures in open source software development.[51] The controversial topic of how research results from open source software development studies can be transferred to traditional software development is also part of a popular seminal essay.[52] Here, the author characterizes open software development (the bazaar) as a fundamental opposition to professional or organizational software development (the cathedral). Furthermore, file repository logs are a valuable source that reflect technical dependencies between software modules *and* social dependencies such as co-authorship of code.[53,54] Supporting explicitly software development with these principles has been studied by Ref. 49. In this work, dependencies are visualized to support the manual discovery of single developers.

Service-oriented Computing (SOC). SOC promises a world of cooperating services loosely connected, creating dynamic business processes and agile applications that span organizations and platforms.[55] Service-oriented architectures (SOA) have emerged as the defacto standard to design and implement large-scale enterprise collaboration systems on the Web. They allow for loose coupling between single components and enable sophisticated discovery mechanisms based on functional (e.g. supported features) and nonfunctional (e.g. QoS) properties. Web service technology[56] enables cross-organizational interactions in collaborative networks.[57] Major software vendors have been working on standards addressing the lack of human interaction support in service-oriented systems. WS-HumanTask[58]

and Bpel4People[59] were released to address the emergent need for human interactions in business processes. These standards specify languages to model human interactions, the lifecycle of human tasks, and generic role models. Role-based access models[58] are used to model responsibilities and potential task assignees in processes. While Bpel4People-based applications focus on top-down modeling of business processes, *service-oriented crowds* target flexible interactions and compositions of Human-Provided services.[4] This approach is aligned with the vision of the Web 2.0, where people can actively contribute services. In such networks, humans may participate and provide services in a uniform way by using the HPS framework.[4] In our work, we combine SOA concepts and social principles. We consider *open service-oriented crowds* wherein services can be added at any point in time. Following the open world assumption, humans actively shape the availability of services. The concept of Human-Provided Services (HPS)[4] supports flexible service-oriented collaborations across multiple organizations and domains. Similarly, emergent collectives as defined by Ref. 60 are networks of interlinked valued nodes (services). Open service-oriented systems are specifically relevant for future *crowdsourcing applications*.[42] While existing platforms (e.g. Amazon's Mechanical Turk[u]) only support simple interaction models (tasks are assigned to individuals), social network principles support more advanced techniques such as formation, delegation, and adaptive coordination.

Trust in Social Networks. Social Trust[15,61–63] in service-oriented systems has become a very important research area. SOA-based infrastructures are typically distributed comprising a large number of available services and huge amounts of interaction logs. Therefore, trust in SOA has to be managed in an automatic manner.[64] Depending on the environment, trust may rely on the outcome of previous interactions[15] and interest similarity.[16,65] Eventually, social trust is an indicator for the strength and degree of social coupling, which we utilize to discover matching social structures to given technical artifact compositions. In our approach, metrics express social behavior influenced by the context in which collaborations take place.[8] For instance, *reciprocity*[66] is a concept describing that humans tend to establish a balance between provided support and obtained benefit from collaboration partners. Social trust networks can be represented as computational models using the FOAF format.[7]

Subgraph Matching. The fundamental basis for our social comosition discovery approach relies on the *concept of induced subgraph isomorphism*.[20] In order to measure the substructure similarity between a target graph (here: distinct parts of a social network) and a query graph (here: a template describing required actor composition properties), different models[21–23] have been proposed; in particular (i) physical property-based, (ii) feature-based, and (iii) structure-based. In our work we

[u]Amazon MTurk: `http://www.mturk.com`

describe a feature-based approach since it allows to introduce some degree of fuzziness in the discovery process and is not as complex to compute as structure-based models; and thus, fit better to large-scale networks. Further works deal especially with multi-billion node graphs,[67] and probabilistic subgraphs for social networks.[68]

6. Conclusion and Future Work

Our work is motivated by the observation from related studies that there is a direct mapping between social dependencies and technical dependencies in large-scale software projects. We presented an approach that uses this knowledge to enable efficient *collaborative* crowdsourcing of software artifacts (and related activities respectively) by considering technical artifact dependencies to discover matching social compositions of crowd members. Social links are defined by members through their FOAF profiles and enriched with data gathered through an automatic interaction mining process. Using service-oriented architectures enables sophisticated interaction monitoring and thus the calculation of interaction metrics that describe collaboration behavior. We evaluated and proved our concepts using data from a real community, i.e. *SourceForge*. This approach ensures that we design our concepts and prototype implementation for scenarios having realistic properties and scale. Our work has important design implications for future frameworks and platforms supporting socio-computational crowdsourcing applications. We discussed typical properties of large-scale *Web-based* software development use cases and demonstrated the application of monitoring and mining techniques, and social composition indexing and discovery mechanisms.

We argue that our proposed approach of subgraph matching with additional query relaxation for discovering member compositions is a good tradeoff between performance and quality. Although the indexation and relaxation approach might lead to misses of best results for specific queries, the mechanisms can still quickly provide results even in large networks. This is essential if small update cycles are required (as typical for high dynamics in collaborative networks) in order to keep the index up-to-date.

The presented indexing approach is going to be applied and evaluated in real end-user environments in context of the EU project COIN. Results from this evaluation will substantially influence further developments and improvements. Conceptually, we currently use the concept of activities to group crowd members and capture the context of interactions. Hence, primarily social compositions within activities are recognized. In our future work we plan to consider also collaborations across activities, in particular in hierarchical activity structures, in order to capture diverse social compositions in large-scale collaborative settings.

Future research objectives for the short term deal with the evaluation of the feasibility of our approach in real world applications and numerous run-time aspects. Although our evaluation on a (static) `sourceforge` data snapshot reveals that we are able to capture, manage and discover social compositions, dynamic aspects of

system usage and time series analysis of SRDA have been out of scope of our evaluation. In particular, we will study the user acceptance and behavior, and thus, the resulting load on our system. Critical objectives are the evaluation in flexible working environments and the application of appropriate mechanisms to avoid resource deadlocks (i.e. the concurrent claim of social compositions) at run time, delegation of activities to balance work within the community[69] and to ensure optimal work distribution.

A further critical aspect, not related to the technical realization itself, is the overall feasibility and acceptance in rather traditional working environments. Related questions, dealing with the acceptance of and trust in proposed compositions by company managers, need to be evaluated in future work to ensure a wide adoption of the proposed socio-computational crowds besides common collaboration models.

Acknowledgments

This work is financially supported by the EU through the FP7 projects COIN (No. ICT-2008-216256) and S-Cube (No. ICT-2008-215483). We would further like to express our gratitude to Prof. Greg Madey and his team from the Computer Science and Engineering Department of the University of Notre Dame for providing us access to the SRDA.

References

1. C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd edn. (Prentice Hall, 2002).
2. M. E. Conway, How do committees invent, *Datamation* **14**(4) (1968) 28–31.
3. E. Trainer, S. Quirk, C. de Souza and D. F. Redmiles, Bridging the gap between technical and social dependencies with ariadne, in *ETX* (ACM, 2005), pp. 26–30.
4. D. Schall, H.-L. Truong and S. Dustdar, Unifying human and software services in web-scale collaborations, *Internet Comp.* **12**(3) (2008) 62–68.
5. D. Schall and S. Dustdar, Dynamic context-sensitive pagerank for expertise mining, in *Social Informatics*, Vol. 6430 of *LNCS* (Springer, 2010), pp. 160–175.
6. F. Skopik, D. Schall, H. Psaier and S. Dustdar, Adaptive provisioning of human expertise in service-oriented systems, in *SAC* (2011), pp. 1568–1575.
7. F. Skopik, D. Schall and S. Dustdar, Trusted information sharing using soa-based social overlay networks, *IJCSA* **9**(1) (2012) 116–151.
8. F. Skopik, D. Schall and S. Dustdar, Modeling and mining of dynamic trust in complex service-oriented systems, *Info. Syst.* **35** (2010) 735–757.
9. C. Gentry, Z. Ramzan and S. Stubblebine, Secure distributed human computation, in *ACM Conf. Electronic Commerce* (ACM, 2005), pp. 155–164.
10. R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices* (Prentice-Hall, Inc, 2002).
11. C. Schroth and T. Janner, Web 2.0 and soa: Converging concepts enabling the internet of services, *IT Professional* **9**(3) (2007) 36–41.
12. D. E. O'Leary, Enterprise knowledge management, *IEEE Comput.* **31**(3) (1998) 54–61.

13. M. Vukovic, Crowdsourcing for enterprises, in *Congress on Services* (IEEE Computer Society, 2009), pp. 686–692.
14. D. Schall, C. Dorn, S. Dustdar and I. Dadduzio, Viecar — enabling self-adaptive collaboration services, in *Euromicro Conf. Software Engineering and Advanced Applications*, pp. 285–292, 2008.
15. L. Mui, M. Mohtashemi and A. Halberstadt, A computational model of trust and reputation for e-businesses, in *HICSS* (2002), p. 188.
16. J. Golbeck, Trust and nuanced profile similarity in online social networks, *ACM Trans. Web* **3**(4) (2009) 1–33.
17. C. Dwyer, S. R. Hiltz and K. Passerini, Trust and privacy concern within social networking sites: A comparison of facebook and myspace, in *Americas Conf. Information Systems* (2007).
18. M. J. Metzger. Privacy, trust, and disclosure: Exploring barriers to electronic commerce, *J. Comput.-Mediated Commun.* **9**(4) (2004).
19. T. Grandison and M. Sloman, A survey of trust in internet applications, *IEEE Commun. Surveys Tutorials* **3**(4) (2000).
20. D. Eppstein, Subgraph isomorphism in planar graphs and related problems, *J. Graph Algorithms Appl.* **3**(3) (1999).
21. X. Yan, P. S. Yu and J. Han, Substructure similarity search in graph databases, in *SIGMOD Conf.* (ACM, 2005) pp. 766–777.
22. X. Yan, F. Zhu, P. S. Yu and J. Han, Feature-based similarity search in graph structures, *ACM Trans. Database Syst.* **31**(4) (2006) 1418–1453.
23. A. Papadopoulos and Y. Manolopoulos, Structure-based similarity search with graph histograms, in *DEXA Workshop* (1999), pp. 174–178.
24. D. Brickley and L. Miller, Foaf vocabulary specification 0.98 (2010).
25. J. Golbeck and M. Rothstein, Linking social networks on the web with foaf: A semantic web case study, in *AAAI Conf. Artificial Intelligence* (2008), pp. 1138–1143.
26. F. Skopik, D. Schall and S. Dustdar, Computational social network management in crowdsourcing environments, in *Int. Conf. Engineering Complex Computer Systems*, 2011.
27. F. Radlinski and T. Joachims, Query chains: learning to rank from implicit feedback, in *SIGKDD KDD* (ACM, 2005), pp. 239–248.
28. B. D. Goodman, Accelerate your web services with caching, *IBM Advanced Internet Technology*, December 2002.
29. J. Howison, K. Inoue and K. Crowston, Social dynamics of free and open source team communications, in *OSS*, Vol. 203 (Springer, 2006), pp. 319–330.
30. K. Crowston and J. Howison, The social structure of free and open source software development, *First Monday* **10**(2) (2005).
31. M. Van Antwerp and G. Madey, Advances in the sourceforge research data archive (srda). in *Fourth Int. Conf. Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*, September 2008.
32. S. Christley and G. R. Madey, Analysis of activity in the open source software development community, in *HICSS* (IEEE, 2007), p. 166.
33. S. Christley and G. Madey, Social positions at sourceforge.net, 2007.
34. F. Skopik, H.-L. Truong and S. Dustdar, Trust and reputation mining in professional virtual communities, in *Int. Conf. Web Engineering* (2009), pp. 76–90.
35. A. Reka and Barabási, Statistical mechanics of complex networks, *Rev. Mod. Phys.* **74** (2002) 47–97.
36. J. Grudin, Computer-supported cooperative work: History and focus, *Computer* **27**(5) (1994) 19–26.

37. K. Schmidt and L. J. Bannon, Taking cscw seriously, *Computer Supported Coop. Work* **1**(1–2) (1992) 7–40.
38. P. H. Carstensen and K. Schmidt, Computer supported cooperative work: New challenges to systems design, in *Handbook of Human Factors*, ed. K. Itoh (Asakura Publishing, 1999), pp. 619–636.
39. P. Wilson, *Computer Supported Cooperative Work* (Intellect Books, 1991).
40. M. J. Stevens and M. A. Campion, Staffing work teams: Development and validation of a selection test for teamwork settings, *J. Manag.* **25**(2) (1999) 207–228.
41. S. Faraj and L. Sproull, Coordinating expertise in software development teams, *Manag. Sci.* **46**(12) (2000) 1554–1568.
42. D. C. Brabham, Crowdsourcing as a model for problem solving: An introduction and cases, *Convergence* **14**(1) (2008) 75.
43. J. Howe, *Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business* (Crown Business, New York, 2008).
44. A. Kittur, E. H. Chi and B. Suh, Crowdsourcing user studies with mechanical turk, in *CHI* (ACM, 2008), pp. 453–456.
45. O. Alonso, D. E. Rose and B. Stewart, Crowdsourcing for relevance evaluation, *SIGIR Forum* **42**(2) (2008) 9–15.
46. O. Stewart, J. M. Huerta and M. Sader, Designing crowdsourcing community for the enterprise, in *KDD Workshop on Human Computation* (ACM, 2009), pp. 50–53.
47. F. Skopik, D. Schall, H. Psaier, M. Treiber and S. Dustdar, Towards social crowd environments using service-oriented architectures, *it — Inf. Technol.* **53**(3) (2011), 108–116.
48. J. D. Herbsleb, A. Mockus, T. A. Finholt and R. E. Grinter, An empirical study of global software development: Distance and speed, in *ICSE* (IEEE Computer Society, 2001), pp. 81–90.
49. C. de Souza, S. Quirk, E. Trainer and D. F. Redmiles, Supporting collaborative software development through the visualization of socio-technical dependencies, in *GROUP* (ACM, 2007), pp. 147–156.
50. C. Bird, N. Nagappan, H. Gall, B. Murphy and P. T. Devanbu, Putting it all together: Using socio-technical networks to predict failures, in *ISSRE* (IEEE Computer Society, 2009), pp. 109–119.
51. C. Bird, D. Pattison, R. D'Souza, V. Filkov and P. Devanbu, Chapels in the bazaar? latent social structure in open source projects, in *Symp. Foundations of Software Engineering* (ACM, 2008), pp. 24–35.
52. E. S. Raymond, The cathedral and the bazaar, *First Monday* **3**(3) (1998).
53. C. de Souza, J. Froehlich and P. Dourish, Seeking the source: Software source code as a social and technical artifact, in *GROUP* (ACM, 2005), pp. 197–206.
54. M. D'Ambros, H. Gall, M. Lanza and M. Pinzger, Analysing software repositories to understand software evolution, in *Software Evolution*, eds. T. Mens and S. Demeyer (Springer, 2008), pp. 37–67.
55. D. Georgakopoulos and M. P. Papazoglou (eds.), *Service-Oriented Computing* (MIT Press, Cambridge, MA, 2008).
56. G. Alonso, F. Casati, H. Kuno and V. Machiraju, *Web Services — Concepts, Architectures and Applications* (Springer, 2003).
57. L. M. Camarinha-Matos and H. Afsarmanesh, Collaborative networks, in *PROLAMAT* (2006), pp. 26–40.
58. M. Amend *et al.*, Web services human task (ws-humantask), version 1.0 (2007).
59. A. Agrawal *et al.*, Ws-bpel extension for people (bpel4people), version 1.0 (2007).

60. C. J. Petrie, Plenty of room outside the firm, *IEEE Internet Comput.* **14**(1) (2010) 92–96.
61. D. Artz and Y. Gil, A survey of trust in computer science and the semantic web, *J. Web Sem.* **5**(2) (2007) 58–71.
62. A. Jøsang, R. Ismail and C. Boyd, A survey of trust and reputation systems for online service provision, *Decis. Support Syst.* **43**(2) (2007) 618–644.
63. J. Sabater and C. Sierra, Social regret, a reputation model based on social relations. *SIGecom Exchanges* **3**(1) (2002) 44–56.
64. Z. Malik and A. Bouguettaya, Reputation bootstrapping for trust establishment among web services, *Internet Comput.* **13**(1) (2009) 40–47.
65. Y. Matsuo and H. Yamamoto, Community gravity: Measuring bidirectional effects by trust and rating on online social networks, in *World Wide Web Conf.* (2009), pp. 751–760.
66. A. Falk and U. Fischbacher, A theory of reciprocity, *Games Econ. Behav.* **54**(2) (2006) 293–315.
67. Z. Sun, H. Wang, H. Wang, B. Shao and J. Li, Efficient subgraph matching on billion node graphs, *Proc. VLDB Endow.* **5**(9) (2012) 788–799.
68. M. Bröcheler, A. Pugliese and V. S. Subrahmanian, Probabilistic subgraph matching on huge social networks, in *ASONAM* (2011), pp. 271–278.
69. F. Skopik, D. Schall and S. Dustdar, Trustworthy interaction balancing in mixed service-oriented systems, in *ACM Symp. Applied Computing* (2010), pp. 801–808.