Uncovering information centres in requirements traceability networks

Inah Omoronyia, Guttorm Sindre and Tor Stålhane
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
e-mail:{inah1, Guttorm.Sindre, Tor.Stalhane} @idi.ntnu.no

Abstract—Non-trivial software project are likely to result in the generation of a dynamic and complex requirements traceability network of numerous artefacts, use cases and collaborating developers. Such complexity makes the process of creating and maintaining requirements traceability links expensive. Also, obtaining useful project related information from existing traceability links is difficult. This is because understanding vital information about a software process requires insight into where information centres exist within the software project. For instance, which of the entities within the traceability network are potential single points of failure and where are the bottlenecks within the software project.

This paper presents an approach that enables an automated generation of requirements traceability network of use cases, code artefacts and developers. These networks are further combined with information centrality measures that can be used to provide useful insight into the potential entities that can hold important information about the software project. An initial study suggests that requirements traceability networks combined with such centrality measure can provide vital cues during software development processes. Lessons learnt during this research are also presented.

Keywords— Requirements Traceability; Information centres; Network Analysis; Collaborative Software Development

I. INTRODUCTION

Requirements Traceability is aimed at defining and utilizing relationships between stakeholder requirements and artefacts produced during software processes [1]. Traceability provides an important means of fostering software understanding, accountability and enhance testing, validation and verification processes [2]. For non-trivial software projects, however, studies have shown that the benefits of requirements traceability are hard to achieve. One reason for this is that there exists a non-scalable growth of up to \( n \) trace dependencies for \( n \) requirements entity instances associated with a software project [3]. Also, traces have to be identified and recorded between numerous, heterogeneous entity instances (use cases, models, code,…). It is challenging to create a meaningful trace network within such a complex set-up, much less derive useful information from existing traces. Furthermore, these traces are in a constant state of flux since they may change whenever requirements or other development entities changes.

This trace complexity demands the need for automated processes for harvesting of requirements traceability links. Furthermore, the approach for which such traceability links are harvested should enable stakeholders to easily derive useful information about the software development process. In a previous paper [4], we proposed an automated process for harvesting requirement traceability links that relate code artefacts and developers to use cases and assigned a relative importance measure to these links based on developer activities during software development. Trace links between code artefacts and developers with use cases, were formed by monitoring events initiated by a developer working in the context of a use case on a code artefact. The relative importance, or relevance, of a code artefact or developer to a selected use case is based on a number of factors: what kind of actions a developer performs e.g. create, edit or view; how often the action is performed; how dominant the entities are - that is, how many other entities they are already associated with - their sphere of influence in the system.

This paper explores how the harvested requirement traceability links that relate code artefacts and developers to use cases can be used to generate a complete traceability network for a software development project. Furthermore, we investigates how the contextual relative relevance of developers and code artefacts to different use cases and other symmetric representations can be combined and translated to provide a context free insight into the centrality of each developer, use case and code artefact to the software project as a whole. Centrality refers to a structural attribute of nodes within a network and provides insight into the importance, influence and prominence of a node in the network. In this research, centrality refers to the relative importance of an entity within a requirements traceability network. It is proposed that such centrality measure can provide vital information required for enhancing and managing non trivial software development processes.

The next section provides a simple example scenario of how trace networks can be formed. We the present a detailed discussion on how to derive trace networks, followed by our approach for determining the centrality of entities within these traceability networks. The implementation of our approach is presented and insights obtained from a preliminary evaluation of the usefulness of generated requirements traceability networks is discussed. An overview of lessons learnt during this research is then presented. The paper ends with a discussion on related work, research conclusion and potential for further work.
II. EXAMPLE SCENARIO

Bill, Amy and Ruben are members of a team collaborating to implement an online cinema ticketing system called TickX. There are two front-end use cases required to accomplish TickX: Purchase Tickets and Browse Movies. In addition, there will be some use cases for system administrators which are not included here. A number of code artefacts are being developed to achieve TickX, including Account.java, Cinema.java, Booking.java, Ticket.java, Customer.java, Movie.java, and MovieCatalog.java. While Amy and Bill have been collaborating to implement the Purchase Tickets use case, Ruben has been responsible for the Browse Movies use case. The following interaction trails were observed as these collaborators worked on their associated use cases:

- While Amy worked on Purchase Tickets she created and updated the Account.java and Customer.java code artefacts. She viewed and updated Booking.java a number of times. She also viewed MovieCatalog.java and Cinema.java.
- In Bill’s early work on Purchase Tickets use case, he viewed Account.java and MovieCatalog.java. This was subsequently followed by his creation and update of Ticket.java and Booking.java.
- Ruben’s work on the Browse Movies use case involved the creation and update of MovieCatalog.java, Cinema.java and Movie.java. He also viewed Ticket.java a number of times.

The detailed interaction event trails is as shown in figure 2. Any selected time-point corresponds to at least one event associated with a use case, a developer, and a code artefact. For instance, at time-point 1, a create event associated with Account.java was executed by Amy while working on the Purchase Tickets use case. Similarly, time-point 7 has two events: Ruben updated Cinema.java (absolute update delta 50) while working on Browse Movies, and Bill viewed Account.java as he worked on Purchase Tickets.

In this scenario, the Purchase Tickets use case is associated with Bill, Amy and a number of code artefacts. Also, MovieCatalog.java is associated with the three collaborators as well as the two use cases. On the whole, within such a rather small and seemingly uncomplicated scenario involving only 2 use cases, 3 developers and 8 code artefacts, 27 different traceability links can be identified. Furthermore, given that this is a quadratic traceability problem and assuming that symmetric and homogenous relations exist, the number of trace dependencies could be as high as 85 (i.e., \((2+3+8)/2\)) over the TickX software life cycle. For non-symmetric trace relation, there could then exist up to 169 trace dependencies. The remaining part of this paper uses this simple scenario to describe how complex traceability networks can be automatically harvested with semantic insight on centrality of involving entity instances.

III. DERIVING TRACE NETWORKS

The monitoring and analysis of events within a development tool and the sphere of influence of project entities is used to derive requirements trace networks. Rather than monitoring the entire space of interactions that can occur, we focus on a core set of event types that influence the changing state of a software project - create, update and view. A create event causes the manifestation of a code artefact within a collaboration space. Associated with an update is the update delta- the absolute difference in the number of characters changed or added to the code artefact before and after the event. A view event indirectly affects the state of artefacts, possibly enhancing the understanding of a developer in order to update the same artefact or other artefact instances.

During collaboration different work contexts - associations between use case, developer and artefact entities - are formed. These work contexts are constantly changing in response to events, and entities may participate in several work contexts. Figure 1 shows example work contexts (represented as graphs) for Amy, Purchase Tickets and MovieCatalog.java. In figure 1a, Amy is the entity that forms the perspective of the work context graph while Purchase Tickets task and MovieCatlog, Account, Customer, Booking and Cinema are all the entities relevant to Amy’s work context.

<table>
<thead>
<tr>
<th>Interaction type</th>
<th>View</th>
<th>Update</th>
<th>Create</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighting factor</td>
<td>0.001</td>
<td>0.0001*Δ</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Δ - Absolute update delta (magnitude of the update)
Weights are assigned to each interaction event type as shown in Table I. The weights were derived from the study of CVS records in real development projects [5], and are in line with related work by Fritz et al. [6] that identified the importance of the creator of code artefacts. In addition, studies conducted by Zou and Godfrey [7] suggested the need to distinguish between random and relevant view events. Thus, viewing is weighted relatively lightly compared to creates and updates (weighted by the size of the update in terms of the absolute number of characters changed).

This research assumes that the size of an entity’s work context or the number of other entities that it exacts its presence on, is proportional to its relative influence in the collaboration space. A use case implemented by several developers and artefacts is considered to hold more information about the state of a project than a use case associated with only a small number of developers and artefacts. This dimension is captured by the concept of sphere of influence (SOI).

SOI is a general concept used to capture both geographic and semantic groupings, and provides a well-defined boundary for interactions [8]. SOI indicates the region over which an entity exacts some kind of relevance (determined by interaction events) and is defined by its work context. The SOI ratio of an entity is defined as the number of unique entity instances directly associated with an entity (the size of its work context) divided by the number of unique entity instances in the whole collaboration space. For the motivating example the SOI ratio of Amy is 6/9 (entities in Amy’s work context /total number of entities - 2 tasks and 7 classes).

The concepts of interaction events combined with SOI ratio within a development work context graph such as the one shown in Figure 1 forms the basis for deriving trace networks with semantic insight on centrality of involving entity instances. Figure 1 demonstrates a basic graph \( G \) consisting of two sets of information: a set of nodes \( E = \{ e_1, e_2, \cdots, e_n \} \) with each member representing an entity instance (\( n \) is the number of entities in a context graph) and a set of arcs \( L = \{ l_1, l_2, \cdots, l_m \} \) (\( m \) is the number of arcs in a context graph) that represent entity relations resulting from interaction events. Arcs are ordered pairs of distinct entities \( l_k = < e_i, e_j > \). The arc \(< e_i, e_j >\) is directed from \( e_i \) (the entity that forms the perspective of work context) to \( e_j \) (the entity that is relevant to the work context of \( e_i \)). Thus, \(< e_i, e_j > \neq < e_j, e_i >\).

\( G \) is a 3-partite graph since its entities \( E \) can be partitioned into three subsets \( E_c, E_d \) and \( E_a \) that each represent the set of use cases, developers and code artefact instances that form the graph. All arcs connect entities in different subsets and no entities in the same subset are adjacent. This assumption suggests for example that two use cases are related only if they both have a relation to the same code artefact or developer instance.

The weight attribute of each arc in \( L \) is specified by the accumulative linear combination of weights gained as a result of events associated with that arc and the sphere of influence of the entity that forms the perspective of work context. More formally, the cumulative weight \( x \) associated with an arc \(< e_i, e_j >\) in response to an event is given by equation (1), where \( t \) is the type of event (possible values shown in Table I), \( s \) the SOI ratio of \( e_i \) and \( n \) the total number of interactions associated with the arc \(< e_i, e_j >\). Thus, the weight attributed for the arc \(< e_i, e_j >\) after \( n \) interactions is based upon its previous value plus the value of the last interaction multiplied by the SOI ratio of \( e_i \).

\[
x_{(n)}<e_i,e_j> = x_{(n-1)}<e_i,e_j> + t_{(n)}<e_i,e_j> \ast s_{(n)}<e_i>
\] (1)
In network analysis, centrality indices are normally used to convey the intuitive feeling that in most networks some vertices or edges are more central than others [9], [10]. A typical example of a centrality index which suits the requirements traceability networks definition is the Markov centrality. This is because Markov centrality can be applied to directed and weighted graphs. To obtain the centrality of entities in this research, the weighted requirements traceability network shown in figure 3 is viewed as a Markov chain. White and Smyth [11] described a Markov chain as a single 'token' traversing a graph in a stochastic manner for an infinitely long time, and the next node (state) that the token moves to is a stochastic function of the properties of the current node. White and Smyth also interpreted the fraction of time (sojourn time) that the token spends at any single node as being proportional to an estimate of the global importance or centrality of the node relative to all other nodes in the graph. From the viewpoint of this research, a Markov chain enables the characterisation of a token moving from a developer to a selected use case as an indication of the relative importance of the use case instance to the developer. Similarly, a token moving from a use case instance to a code artefact indicates the importance of the artefact instance in achieving the use case.

The appropriate derivation of centrality from a Markov chain lies in the ability to derive a transition matrix from the weighted requirements traceability network. This is achieved by assuming that the likelihood of a token traversal between two nodes is proportional to the weight associated with the arc linking the nodes. Thus, higher transition probabilities are associated with arcs with higher weights. The weights in a traceability network are then converted to transition probability weights by normalising the weights on arcs associating entities with a work context to one. Thus, transition probability is dependent on each arc weight value and the total number of entities within a work context. Figure 4 represent the transition matrix derived from the requirements traceability network of TickX project. Figure 4 shows that the transition probability of a token from Ticket to Browse Movies use case is 0.0339 while the reverse of a token traversal from Browse Movie to Ticket is 0.0044. Similarly, the transition probability of a token traversal from Ticket to Purchase ticket use case is 0.4661. It is important to note that each of the rows in the transition matrix sums to one. The algorithm and computational processes for the derivation of transition matrix and the subsequent centrality of entities was carried out using the Java network/graph framework (JUNG) [12].

Figure 7 shows a graphical representation of a requirements traceability network for TickX project where the size of each entity is proportional to its Markov centrality in the network. This figure shows the relatively higher centrality that MovieCatalog.java has achieved in the collaboration space.
II. MODEL IMPLEMENTATION AND INITIAL STUDY

The implementation of a prototype in this research envisages a scenario where requirement analyst can express the use cases required for achieving a system in a shared collaboration space. These use cases can be updated or removed over the life time of the project and new ones can be added. Developers are then able to select any use case they are interested in implementing. Finally, the traceability model is achieved as the use case selected is automatically traced to every update, create and view event that the developer carried out on code artefacts while achieving the system.

The requirements traceability approach in this paper has been implemented as a client server architecture, where the Eclipse IDE for each developer is a client and the model processing logic and storage of event data is performed on the server. The client server approach models a shared collaboration space. The client monitors sequences of view, update and create events executed within Eclipse (chosen because of its open plug-in architecture). When a network connection exists, this event data is offloaded to the server and synchronised with that of other developers. While there is no connection (or a slow connection) the client will temporarily store event data locally and perform local model processing logic to give the developer a partial view of current trace links and their relative centrality - offline mode. The implementation architecture is as shown in Figure 5. The architecture is distributed across client and server ends, and consists of four core layers: the model, event, messaging and Rich Client Platform (RCP). The client end of each layer is plugged into the Eclipse platform while the server end resides on an Apache Tomcat web application server.

![Figure 4. Transition matrix derived from TickX requirements traceability network](image)

![Figure 5. Implementation architecture](image)
developer are traced to the work context of an activated use case. The RCP layer is also responsible for generating visualisations of requirements traceability network of developers, artefacts and use cases. The generation of the traceability network as shown in figure 7 is triggered by a system developer using the button labelled 3 in figure 6. A selected node in the network can be moved around within the visual interface to enhance clarity of trace relations for increasingly complex trace networks.

The workflow requires that each time a developer wants to carry out a coding activity, they log in and activate an existing use case located in the central repository or create a new one. For each client workstation, only one use case can be active at a selected time, working on another use case requires that the developer activates the new use case which automatically deactivates the previous one. Similarly, the active code artefact is the current artefact being viewed, updated or created. Switching to another artefact automatically deactivates the previous artefact. This workflow enables cross cutting relations amongst artefacts, developers and use cases since, over their lifetime, and as they are used to achieve different aspects of a project, each can be associated with any number of other instances. As events generated by the developer are traced to the work context of an active use case and artefact on the server, the centrality value of each entity instance involved in the traceability network is recalculated.

A. Initial study

To obtain insight into the potential usefulness of generated requirements traceability networks, a six weeks study involving ten software engineering students in the third year of their Masters/Honours programme was carried out. All participants had at least 2.5 years of object-oriented development experience using Java. All were participating in project developing 'Gizmoball' - an editor and simulator for a pinball table - working in groups of three [5]. During the study, use cases were modelled and tagged with meaningful short form descriptions or acronyms that was easy to understand by the collaborators. Furthermore, to minimize intrusion and closely mimic real collaboration scenarios, use cases were defined by developers on an ad hoc basis and used as a basis for tasks assignment.

At the end of the six weeks, structured interviews were conducted with eight of the participants (the two remaining participants were unavoidably absent). The interviews were personalised based on the use cases and code artefacts that the participant had worked on. Also, permission was granted by participants to obtain an audio record of the interviews which was then transcribed to follow up the interesting responses. All data were anonymised for analysis and presentation. Feedback from participants suggested that the tool captured between 60-90% of the interaction events carried out over the study period.

At the initial phase of the interview each participant was presented with a random list of code artefacts he/she had worked on while achieving different use cases, then asked to select the four they had put in the most coding effort over the six weeks. Some insight on how developers formed perception of work effort was revealed. One participant formed perception of work effort based on related clusters of code artefacts. For instance, the snippet from the participant ‘Paul’ is as shown below:

Paul: *I am not sure of Wall.java, I dont think I put as much coding effort into Wall.java as I put into the flipper related classes...*

Paul’s work context graph is as shown in figure 8 and demonstrates that he had worked on 2 use cases and used 26 code artefacts. Furthermore, the analysis of the events associated with Paul while working on Wall and flipper related classes showed that his perception could have been flawed. This is because although the total effort of Paul on the flipper classes, based on events he generated, was greater than the effort used for Wall, none of them individually exceeded Wall (for Wall 12 views and a total update delta of 1400 characters were recorded, for Flipper it was 8 and 214, for LeftFlipper 7 and 778, and for RightFlipper 19 and 25). It is expected that such clustering of related code artefacts while developing a perception of work effort will be more frequent as Paul works on more artefacts. This is particularly so if more code artefacts share or are related to a higher level system functionality over the project lifetime.

![Figure 6. Snapshot of Eclipse view of visualization components](image)

![Figure 7. Trace graph for TickX](image)
Other instances where the pattern of formation of work effort could be flawed was demonstrated when one participant discounted a code artefact because it was only a small driver module, but then acknowledged that it was modified each time the use case tagged user interface was tested; another was the perceived difficulty associated with an artefact – one participant ranked an artefact with lower work effort as it was perceived as straightforward with many edits that were not hard to implement; and lastly a participant discounted an artefact because it was ... simply copied and pasted from an online source.

Each of these instances demonstrates that there existed a need for a requirements traceability network embedded with semantic information on the centrality of entities. Such traceability network can assist collaborators in tracing how work effort is being dissipated across a software project.

In the second phase of the interview, participants were asked to narrate their usage experience of generated requirements traceability networks over the study period. An example is the collaboration that generated the traceability network as shown in figure 9 and involved collaboration between Greg, Boris and Blair to achieve Gizmoball. Two requirements use case tags were identified - 'From Demo to Final' (Translate game demo to final mode), and JUnit Tests (generate test cases for each gizmo object). 45 artefacts were identified as being used to achieve these use cases. While the major responsibility of achieving 'From Demo to Final' was assigned to Boris, the responsibility for JUnit Tests was mainly assigned to Blair. Snippet from Boris demonstrating insight he obtained while navigating the traceability network generated as a result of their collaboration (figure 9) is shown below:

Boris: ...If we have done 'JUnit Test' how come it only relates to only Gizmo.java, Square.java and GizmoModel.java...? Because I know that it should be looking at virtually all of the code... ...there is more work to be done in 'JUnit Tests'

This feedback suggests that Boris was expecting JUnit Tests to have a higher centrality in the network. He also expected the use case to be related to more code artefacts. This is because they had decided to use test driven development and as such needed every code artefact to be assigned a test case. While they had agreed and documented their decision on test driven development in their previous group meeting, the traceability network of the current state of the project rather suggested that there was still much work to be done to achieve their agreed objective.
Other collaboration insights also suggested that the requirements traceability network helped to reveal issues that system developers normally would not have thought of. For instance, the graph shown in figure 10 involved collaboration between Luke, Alex and Tony to achieve the Gizmoball project. Feedback suggested that they used the graph to visualize where the bigger challenges in the system were. These graphs were also used to get a grasp of the work pattern of system developers and which use case had changed more considerably recently.

III. DISCUSSION

An advantage of our approach is that harvesting requirements traceability links is automated and that the network is constantly updated to reflect the current state of the project. Furthermore, entities that are more likely to hold greater information about the project are made explicit by calculating their centrality within the requirements traceability network. The Markov centrality of an entity is determined based on the type of events they have been associated with and their sphere of influence.

On the whole, insights obtained from the initial study suggest that potentially useful information can be obtained from requirements traceability network representation. These include the ability to easily determine bottleneck entity instances in a software project. For instance, deleting artefacts, updating tasks descriptions, or removing developers that are all attributed with high information centrality in a shared project collaboration space could be detrimental to the information/knowledge flow for that project. Furthermore, requirements traceability network can enhance the realtime project risk assessment processes, since centrality of entities also suggest high volatility and a potential single point of failure of the entity instance in the network.

There have been a number of lessons learned from the modelling, implementation and subsequent initial study of our requirements traceability network. One of the important lessons learned from the modelling of traceability network is that the SOI ratio can be central in revealing a number of latent properties of a collaborative software development process. For instance, a high SOI ratio for a developer may suggest that he/she is working on many parts of the system and hence central to the development process. Furthermore, if most developers tend to be associated with a high SOI ratio, then it might imply a shared code ownership development model such as extreme programming. If a use case has a high SOI ratio then this can indicate its importance to the development process. On the other hand it might also indicate poor use case definition and allocation practice the use case has for instance not been broken down enough or that the development process has not been well segmented. The use of SOI as the basis of a forensic analysis of the design and its development has rich potential for future work.

A drawback observed from the study was that the traceability network became increasingly cluttered as the number of entities associated with a project increases. Thus, while a selected entity from a traceability network could be moved around within the implementation interface for visual clarity, this was a difficult process for complex networks. To help overcome this drawback, a Fisheye visualisation (figure 11) based on centrality of entity instances associated with a project has been implemented. Fisheye view has been shown to be an efficient mechanism to enhance clarity for complex visualisations with increasing number of nodes [13]. Figure 11 is a Fisheye visualisation to support the requirements traceability network shown in figure 10. Entities with higher centrality values are clearly shown at the top of the view.

An implied workflow constraint, based on the implementation of the model, requires that systems analysts and developers need to explicitly be working within the context of a selected use case. This is achieved by activating the use case within the development tool. Insight obtained from the initial study suggest that such workflow constraint can sometimes be difficult to achieve, especially when developers have cumbersome project schedules and date lines. Feedback from study shows that the explicit activation of a use case during development work is sometimes not a primary concern of the participant and he/she thus might forget to formally carry out the use case activation processes within Eclipse IDE.

Finally, while our choice of allowing the definition of use cases by developers on an ad hoc basis during the evaluation minimised intrusion, it also had some implications. Typically, this choice made it impossible to monitor and verify potential conceptual gaps in the definition of use
cases. This is shown in the study as potential tasks for the development process were rather modelled as use cases. This outcome suggests that there need to be a more formal than ad hoc approach in the definition of use cases.

IV. RELATED WORK

A simplistic way to reduce the complexity of traceability network resulting from multiple entity dependencies is the enforcement of strict partitioning. An example includes cases where system developers are strictly assigned to use cases and there is then strict enforcement of code ownership. Such scenarios are difficult to achieve in non-trivial software development settings. This is because such development settings are normally attributed with high degree of collaboration such as open source, and distributed software engineering projects where artefact ownership is not always obtainable [14], [15].

Research direction for the automation of trace link creation and maintenance covers a wide spectrum of approaches. These include the application of text mining and information retrieval techniques to recover trace links between software artefacts and their abstract model representations [16], [17]. Here, trace links are generated by computing the similarity between a query and each artefact that comprises a software project. The probabilistic nature of this approach sometimes results in software developers manually analysing and discarding high numbers of false positives. Another approach involves the derivation of trace links from existing ones [18]. This approach requires the existence of a set of initial trace links between artefacts and their abstract models. An approach which is closely related to the research presented in this paper is the creation and maintenance of trace links by monitoring users’ modifications and analyzing change history [19], [20].

Mader et al. [20] proposed an approach for the automated update of existing traceability relations during the evolution and refinement of UML analysis and design models. The approach observes elementary changes applied to UML models, recognises the broader development activities and triggers the automated update of impacted traceability relations. The elementary change events on model elements include add, delete and modify. The broader development activity is also recognised using a set of rules which helps in associating an elementary change as constituent parts of intentional development activity. The key similarity between the approach in this paper and Mader et al.'s approach is the focus on maintaining up-to-date post requirement traceability relations. In addition, our approach provides a perception of the centrality of traced entities based on the amount of development activity required to achieve and maintain the associating links.

V. CONCLUSION

The main aim of this paper has been to identify information centres in a complex network of requirements engineering entities such as use cases, artefacts and developers. This has been achieved via an automatically harvested requirements traceability network. The centralities of entity instances in the network are then determined. The type and amount of events that has been associated with an entity and its sphere of influence are the two main constructs used for centrality prediction. Initial study has demonstrated that visualisation of requirements network embedded with such centrality measures has the potential of enabling stakeholders to obtain a perception of critical and volatile entities responsible for the evolutionary state of a software project.

Further work focuses on enhancing the accuracy of centrality values. This involves understanding the effect of the nature of development task (e.g. maintenance, debugging, refactoring or simply forward engineering) on centrality of entities. Secondly, homogenous relations amongst entities of the same type were not considered in the prediction of the centrality of entities. It is expected that such homogenous relations can provide further insight into centrality prediction. Enforcing homogenous relations would require further static and dynamic analysis of code artefacts to identify dependency relations amongst use cases. It also demands an understanding of complex group dynamics of how developers and use cases depend on each other beyond relationships derived from structural and more explicit constraints.

REFERENCES


Inah Omoronyia, Guttorm Sindre, Tor Stålhane © 2009. The authors assign to AWRE a non-exclusive licence to publish this paper in full in the AWRE09 Proceedings. The paper may be published on the World Wide Web, CD-ROM, in printed form, and on mirror sites on the World Wide Web. Any other usage is prohibited without the express permission of the authors.