

Chapter 5

Understanding Architectural Elements from Requirements Traceability Networks

Inah Omoronyia, Guttorm Sindre, Stefan Biffl, and Tor Stålhane

Abstract The benefits of requirements traceability to understand architectural representations are still hard to achieve. This is because architectural knowledge usually remains implicit in the heads of the architects, except the architecture design itself. The aim of this research is to make architectural knowledge more explicit by mining homogenous and heterogeneous requirements traceability networks. This chapter investigates such networks achieved by event-based traceability and call graphs. Both traces are harvested during a software project. An evaluation study suggests the potential of this approach. Traceability networks can be used in understanding some of the resulting architectural styles based on the real time state of a software project. We also demonstrate the use of traceability networks to monitor initial system decisions and identify bottlenecks in a software project.

5.1 Introduction

In spite of substantial research progress in the areas of requirements engineering and software architectures, little attention has been paid to how to bridge the gap between the two [1]. It is essential to know how to transition from requirements to architecture and vice versa, and to understand the impact of architectural design on existing and evolving software requirements. This research focuses on investigating how requirement traceability approaches can be used to bridge this gap between software requirements and architectural representations.

When software requirements evolve, appropriate traceability mechanisms can provide an understanding and better management of the linking between requirements and associated artefacts during evolving project cycles [2]. Evolution of software requirements suggests a similar evolution in architecture because changes to software requirements normally imply updates to the different components used to achieve the system. Such component updates can trigger a change in structure of the system and the relationship of updated components with other components of the system.

Thus, the main research question is how software requirements evolution impacts on the underlying architecture of the system. This question will be addressed by investigating how traceability relations between software requirements and different components in a system reveal its architectural implications. Turner et al. [3] describe a requirement feature as “a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective.” We envisage a scenario where decisions are previously taken on the desired architecture to be used in implementing a specified feature in the system. We subsequently harvest homogenous and heterogeneous requirements traceability networks. Such traceability networks can also represent semantic graphs from which the actual architectural representation of the system can be inferred. The aim then is to compare and validate the desired architecture against the real-time inferred system architecture used to implement a desired user feature.

In the remaining part of this chapter, Sect. 5.2 first provides the background on requirements traceability and software architectures and discusses the architectural information needs of different stakeholders. Section 5.3 presents the automated requirements traceability mechanism that is used to realize our traceability networks. A system architectural inference mechanism based on extracted requirements traceability networks is explained. Section 5.4 presents an evaluation of our approach based on an implemented prototype. Section 5.5 presents related work and subsequently our conclusion and further work in Sect. 5.6.

5.2 Requirements Traceability and Software Architectures

5.2.1 *Inferring Architectural Rationale from Traceability Networks*

Software architecture seeks to represent the structure of the system to be developed. The structure is defined by components, their properties and inter-relationships [4]. As pointed out by Bass et al. [5], a project’s software architecture acts as a blueprint, serves as a vehicle for communication between stakeholders and contains a manifest of earliest design decisions. An architecture is the first artifact that can be analyzed to determine how well the quality attributes of an ongoing project are being achieved. In line with other chapters in this book, architectural characterisations range from the actual architecture of the system to its inferred or intended architecture. Such architectures do not exist in isolation, rather they are influenced by external factors such as the system requirement features and quality goals from the customer and developing organization, task breakdown structure and developer task assignment, lifecycle etc. Architectural rationale is thus a means to understand design architectures by considering the external factors that has influenced their realisation. Architectural rationale is realised as stakeholders endeavour to satisfy their architectural information needs by asking questions that have architectural

implications (see example of questions in Sect. 5.2.2). Architectural rationale is essential to access if a desired architectural plan for achieving a specified system’s requirement is being realized in the tangible real-time representation of the system.

There are different viewpoints on traceability, but mostly aimed at addressing the same research problem of enhancing conformance and understanding during software development processes. Palmer [6] claims “traceability gives essential assistance in understanding the relationships that exist within and across software requirements, design and implementation.” Requirements traceability enables the harmonization between the stakeholder’s requirements and the artifacts produced along the software development process. Alternatively, requirements traceability is aimed at identifying and utilizing relationships between system requirements and other artefacts produced during a software project’s lifecycle [7]. Typically, such artefacts include external documents, code segments, hardware components and associated stakeholders. Traceability facilitates software understanding, accountability, and validation & verification processes. These benefits of traceability have particularly been realized between explicit software artifacts, such as homogenous relationships between instances of requirement and heterogeneous relationships between requirements and code artefacts [8]. Relationships have varying degrees of relevance depending on the stakeholder involved.

The benefits of requirements traceability to software architectural representations are still little explored. This is because architectural knowledge which consists of architecture design, design decisions, assumptions and context, usually remains implicit in the minds of the architects, except the delivered architecture design itself [9].

Figure 5.1 represents our approach to investigating architectural representations from harvested traceability links. This scenario assumes that at the early phase of

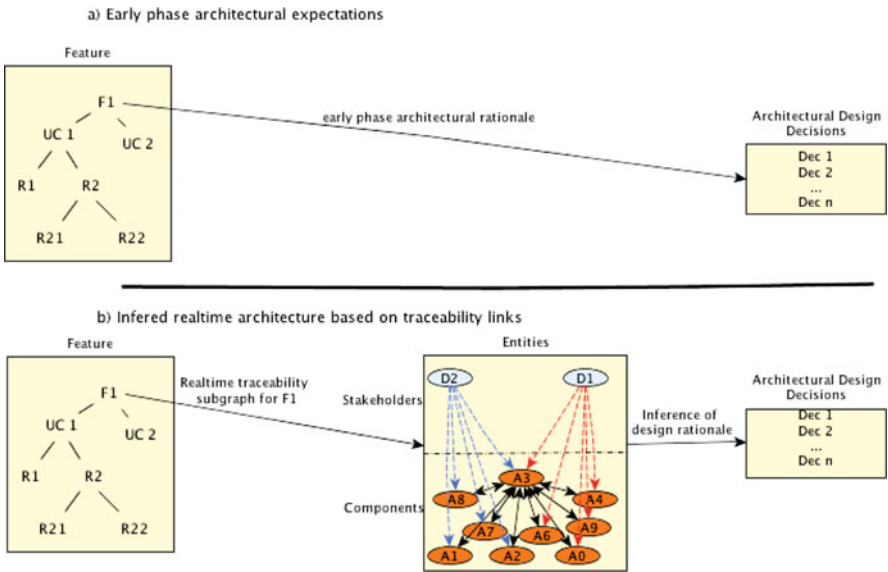


Fig. 5.1 Inferring architectural rationale from traceability links

the project, architectural decisions are made to implement specific features of the system. Such features can further generate a set of use cases and more concrete requirements that achieve the use case (Fig. 5.1a). Subsequently, different stakeholders use a set of components to achieve a specified system feature as shown in Fig. 5.1b. Thus, the main concern here is deriving some real time architectural insight based on the trace links generated between system features or use cases, components and stakeholders.

It is not straightforward to achieve architectural insight based on tangible representation of the system and harvested traceability links since different components of a system can be associated with multiple desired features of the system and in most cases worked on by different stakeholders from varying perspectives, to achieve different tasks or features. Hence, traces between system components, features and stakeholders will result in a complex web from which the core challenge is to infer an earlier guiding design rationale. The aim of this research is to reveal architectural rationale from such a real-time traceability viewpoint. This is achieved by mining homogenous and heterogeneous requirements traceability networks. In this chapter, we focus on a subset of possible architectural insights classified either as explicit or implicit. Explicit insight can be directly inferred from generated traceability networks, e.g., the architecture style revealed by real-time links between project entities. Implicit architectural insight is the additional information that can be assumed based on the interaction between the different project entities, e.g., development model, feature and requirements breakdown structure, decomposition and allocation of responsibility, assignment to processes and threads, etc.

5.2.2 *Stakeholder Needs for Architectural Information*

Palmer's [6] viewpoint of requirements traceability suggests that the relationships between different project entities can allow architects to show compliance with the requirements and help early identification of requirements not satisfied by the initial architecture. Hence, some information needs must be satisfied by the traceability links. It is expected that a stakeholder's needs will vary depending on role (e.g., architect/programmer/tester) and task (e.g., initial development/maintenance). As an example, consider software developer D who needs to make some changes to a software product due to a requirements change, e.g., the customers have expressed a wish for altering a certain use case. Unless the task is trivial, there are a number of questions that D might ask, for example:

- Which code artefacts (e.g., classes) are involved in implementing the use case and how do they affect the architecture of the system, e.g., a predefined architectural style or the system feature and requirements breakdown structure?
- If the class C is modified to fulfil the requirements change, what other use cases or system features might also be affected by this modification? And what

adaptation is required in the initial architecture, (e.g., the assignment of class components to processes and threads)?

- Who were the stakeholders involved in writing the use case, or the class C and other classes that are relevant for the use case? How has the desired change affected the decomposition and allocation of responsibility in the initial software architecture?

We refer to these requests as architectural information needs, defined as the traceability links between entity instances, system features or use cases, stakeholders and code artefacts. Such information is essential to understand the architecture. Ideally the traceability links required to answer such questions might have been explicitly captured during the project, e.g., which developer contributes to which artefacts, and which artefacts are related to each other? But this rarely happens – at best such traceability information is incomplete and outdated because many developers find it too time-consuming to update it.

In the remaining part of this research, we investigate traceability links harvested automatically by event based tracing and the use of call graphs. We then evaluate a number of architectural representations inferred from the harvested traceability links.

5.3 Deriving Requirements Traceability Networks for Inferring Architectural Representations

In this section, we present an automated method for harvesting a traceability network based on the scenario below:

Scenario. Bill, Amy, and Ruben are members of a team developing an online cinema ticketing system, TickX. There are two front-end use cases required: Purchase Tickets and Browse Movies. Additional use cases for system administrators are not discussed here. A number of code artefacts are being developed to realise TickX, including Ticket.java, Customer.java, Account.java, Booking.java, Movie.java, MovieCatalog.java, and Cinema.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:

- While Amy was collaborating 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 the initial phase of Bill's collaboration on the Purchase Tickets use case, he viewed Account.java and MovieCatalog.java. Then he created and updated Ticket.java and Booking.java.

- Ruben’s implementation of the Browse Movies use case involved the creation and further updating of MovieCatalog.java, Cinema.java, and Movie.java. Ruben also viewed Ticket.java a number of times.

Traceability links can be homogenous, e.g., a code component being related to another code component, or heterogeneous, e.g., a relationship between a developer and code component, or a use case and component. The detailed interaction event trails is as shown in Fig. 5.2. 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 [magnitude of the update based on character difference]) 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 developers as well as the two use cases. On the whole, within such a rather small and seemingly uncomplicated scenario involving only two use cases, three developers and eight code artefacts, 27 different traceability links can be identified. To make sense of such number of dependencies, they must be ranked for relevance. For instance, the relevance of traceability links between a use case and developer is dependent on the number of interaction events generated over time by the developer in achieving the use case. A relevance measure of trace links between two entities is non-symmetric. This is because the relevance measure is firstly dependent on the number of other entity instances a selected entity can be traced to, and secondly the amount of interaction events generated as a result of each trace link.

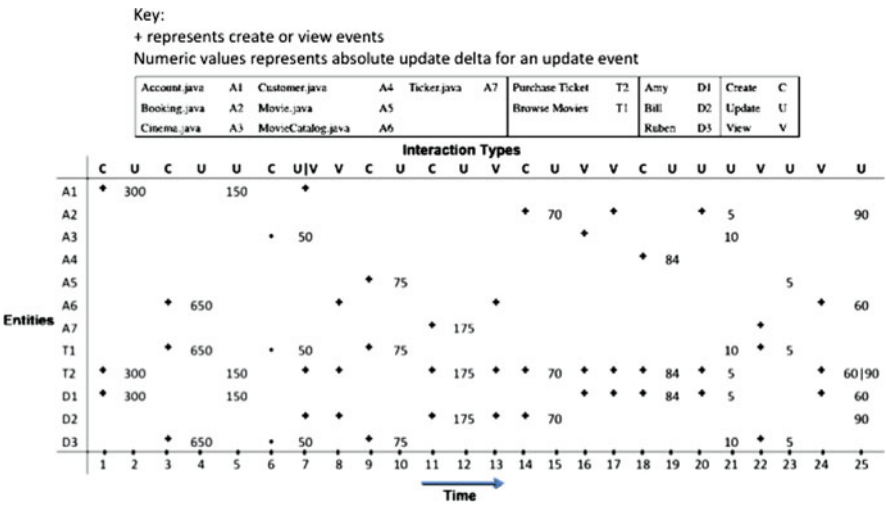


Fig. 5.2 Monitored interaction trails used to achieve TickX across 25 time-points

This demonstrated scenario poses a number of questions. Firstly, what are the possible automated methods for harvesting a traceability network? Secondly, how can the system's architectural representations be revealed by these networks? In addressing the first question, this research investigates an event based mechanism for retrieving interaction events for the subsequent generation of traceability networks. This involves capturing navigation and change events that are generated by the developers. The advantage of this approach is the opportunity to automatically harvest real-time trace links. The event based approach also provides a basis for inferring real-time architectural representations. Some ideas for such an approach has been presented in earlier work [10]. In this section, we present an event based linear mechanism to generate traceability links and rank their relevance. Since event based approaches are sometimes prone to generating false positives, we also use call graphs to validate the event based networks.

5.3.1 Event-Based Mechanism for Capturing Trace Links

In a previous paper [2], we proposed an automated event-based process for harvesting requirement traceability links relating code artefacts and developers to use cases. Trace links 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 was based on the type and frequency of developer actions, e.g., create, edit or view; and on the entity's *sphere of influence* in the system, i.e., how many other entities they are associated with .

This chapter explores how the harvested requirement traceability links can be used to generate a complete traceability network for a software development project. Furthermore, it is investigated how the relative importance of trace links can be used to provide insight into the centrality of each developer, use case and code artefact to the software project as a whole. Centrality is a structural attribute of nodes within a network and provides insight into the importance, influence and prominence of that particular node. Based on the centrality of entities in traceability networks, we investigate architectural rationale that can be inferred.

The event based mechanism uses events generated within a development tool and the sphere of influence of project entities 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. 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 (system features), 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 5.3 shows example work contexts for Amy, Purchase Tickets, and MovieCatalog.java. In Fig. 5.3a, Amy is the entity that forms the perspective of the work context graph while the Purchase Tickets use case and the classes MovieCatalog, Account, Customer, Booking and Cinema are all the entities relevant to Amy’s work context.

Weights are assigned to each interaction event type as shown in Table 5.1. The weights were derived from the study of CVS records in real development projects [11], and are in line with related work by Fritz et al. [12] that emphasized the importance of the creator of code artefacts. In addition, studies conducted by Zou and Godfrey [2] 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). Typically, changing one line of code is much less significant compared to rewriting an entire module.

This research assumes that the size of an entity’s work context 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 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 [13]. SOI indicates the region over which an entity exerts some kind of relevance and is defined by its work context. The *SOI ratio* is used to represent the relative influence an entity has on the collaboration space. The SOI ratio of an entity is defined as the

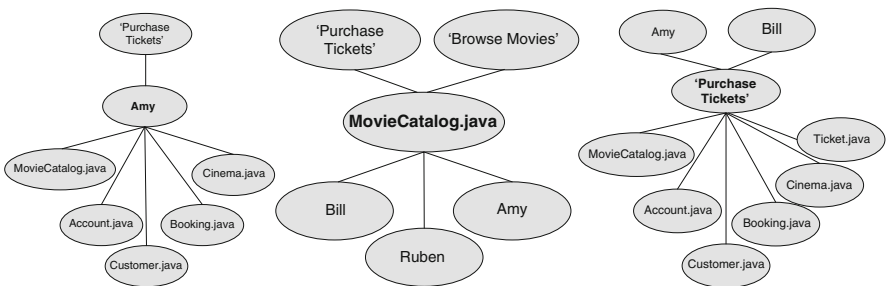


Fig. 5.3 Work context graphs

Table 5.1 Interaction type weightings			
Interaction type	View	Update	Create
Weighting factor	0.001	0.0001*Δ	0.01
Δ, absolute update delta			

number of unique entity instances directly associated with it 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 – two use cases and seven classes).

The concepts of interaction events combined with SOI ratio forms the basis for deriving trace networks with semantic insight on centrality of involved entity instances. Figure 5.3 shows three directed graphs. In general, a graph G has a set of nodes $E = \{e_1, e_2, \dots, e_n\}$ and a set of arcs $L = \{l_1, l_2, \dots, l_m\}$ which are ordered pairs of distinct entities $l_k = \langle e_i, e_j \rangle$. The arc $\langle e_i, e_j \rangle$ is directed from e_i to e_j . Thus, $\langle e_i, e_j \rangle \neq \langle e_j, e_i \rangle$. In our usage, the graphs are three-partite since their entities E can be partitioned into three subsets E_c , E_d and E_a (use cases, developers and code artefacts). All arcs connect entities in different subsets.

The weight attribute of each arc 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 $\langle e_i, e_j \rangle$ in response to an event is given by (5.1), where t is the type of event (possible values shown in table 1), s the SOI ratio of e_i , and n the total number of interactions associated with the arc $\langle e_i, e_j \rangle$. Thus, the weight attributed for the arc $\langle e_i, e_j \rangle$ after n interactions is based upon its previous value plus the value of the last interaction multiplied by the SOI ratio of e_i .

$$x_{(n)\langle ei, ej \rangle} = x_{(n-1)\langle ei, ej \rangle} + t_{(n)\langle ei, ej \rangle} s_{(n)ei} \quad (5.1)$$

As a further illustration of how to use events generated while developing TickX as shown in Fig. 5.2, time-point 8 represents a view event carried out by Bill while working on the Purchase Tickets use case using MovieCatalog.java. Subsequent to this, time-points 1, 2, 5 and 7 are other events carried out by Amy and Bill using Account.java within the work context of Purchase Tickets. Thus, the SOI of Purchase Tickets at time-point 8 is 0.67 (four artefacts and developers in the Purchase Tickets work context divided by six artefacts and developers in total). The weight of the arc tracing MovieCatalog.java to the work context of Purchase Tickets at time-point 8 is 0.0007. The next event involving the use of MovieCatalog.java within Purchase Tickets is represented in time-point 13, and the weight gained as a result of this event is 0.0006 and the cumulative weight is 0.0013. By the end of the trail in time-point 25 the relation from MovieCatalog.java to Purchase Tickets work context has obtained a cumulative weight value of 0.0069.

The total number of context graphs in a software project depends on the unique number of use cases, code artefacts and developers in the project. A use case may be related to a number of code artefacts and developers, and vice versa. This produces a complex network combining the results of different work context graphs. A typical example of such a network for TickX project is shown in Fig. 5.4.

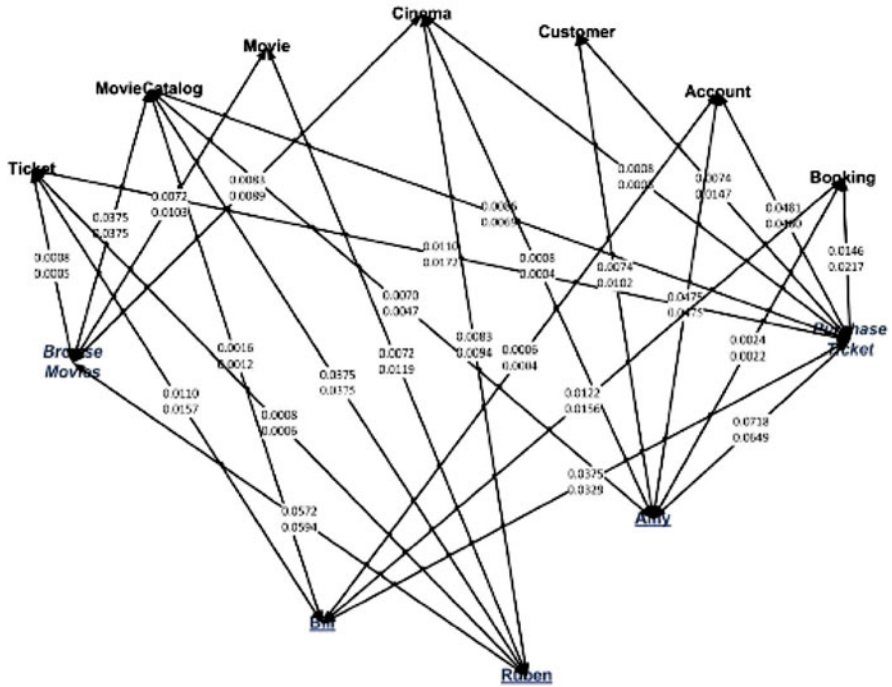


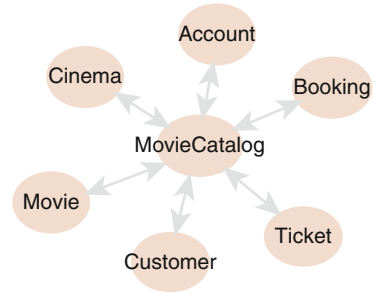
Fig. 5.4 Traceability network for TickX

5.3.2 Capturing Trace Links Between Components Via Call Graphs

The event-based approach is suitable for capturing heterogeneous links between stakeholders, use cases/features and associated code components. From an architectural viewpoint, there is also need to capture direct links between different components. Here, we investigate the use of *call graphs* to achieve homogenous traceability between software components. Call graphs are directed graphs that represent calling or message passing relationships between components of a system. From a software engineering perspective, call graphs are either dynamically generated (at execution time) or statically generated (at compile time). The core focus of this work is on the use of static call graphs generated by message passing between code components. Figure 5.5 is an example of a call graph for TickX. Figure 5.5 shows that MovieCatalog is a central component through which other components pass or receive messages.

On the whole, a requirements traceability network is a merge of homogenous and heterogeneous traceability links. Thus, a requirements traceability network is a graph of system components, use cases/desired system features and stakeholders of the system. An example of such a traceability network is shown in Fig. 5.8.

Fig. 5.5 Graphical representation of a call graph between different TickX components



5.3.3 Centrality of Entities in Traceability Networks

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 [14, 15]. A centrality index which suits the requirements traceability networks definition is the Markov centrality, which can be applied to directed and weighted graphs. To obtain the centrality of entities in this research, the weighted requirements traceability network shown in Fig. 5.4 is viewed as a Markov chain. White and Smyth [16] 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. They 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.

Centrality is calculated by deriving a transition matrix from the weighted requirements traceability network, assuming that the likelihood of a token traversal between two nodes is proportional to the weight associated with the arc linking the nodes. 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 5.6 gives the transition matrix for TickX. The transition probability of a token from Ticket to Browse Movies use case is 0.0339 while the reverse probability is 0.0044. 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) [17].

Figure 5.8 shows a graph for TickX where the size of each entity is proportional to its Markov centrality. This figure shows the relatively higher centrality that MovieCatalog.java has achieved in the collaboration space.

	Amy	Bill	Ruben	Purchase ticket	Browse movies	Movie	MovieCatalog	Booking	Cinema	Account	Ticket	Customer
Amy	0	0	0	0.5000	0	0	0.0359	0.0167	0.0023	0.3658	0	0.0787
Bill	0	0	0	0.5000	0	0	0.0178	0.2374	0	0.0061	0.2387	0
Ruben	0	0	0	0	0.5000	0.1005	0.3154	0	0.0794	0	0.0047	0
Purchase ticket	0.3284	0.1716	0	0	0	0	0.0315	0.0993	0.0035	0.2196	0.0786	0.0673
Browse movies	0	0	0.5000	0	0	0.0897	0.3281	0	0.0773	0	0.0044	0
Movie	0	0	0.5000	0	0.5000	0	0	0	0	0	0	0
MovieCatalog	0.0759	0.0174	0.4067	0.0933	0.4067	0	0	0	0	0	0	0
Booking	0.0822	0.4178	0	0.5000	0	0	0	0	0	0	0	0
Cinema	0.0440	0	0.4560	0.0440	0.4560	0	0	0	0	0	0	0
Account	0.4938	0.0062	0	0.5000	0	0	0	0	0	0	0	0
Ticket	0	0.4661	0.0339	0.4661	0.0339	0	0	0	0	0	0	0
Customer	0.5000	0	0	0.5000	0	0	0	0	0	0	0	0

Fig. 5.6 Transition matrix for TickX requirements traceability network

5.3.4 Model Implementation

The implementation of a prototype envisages a scenario where the requirement analysts can specify the use cases or features 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 implementing that use case.

The requirements traceability approach in this chapter 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 view, update and create events executed within Eclipse. When a network connection exists, event data are offloaded to the server. 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 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.

The model layer is the main event processing unit in the architecture. This layer is responsible for the formation of entity work contexts and their related SOI ratios, and also generates the centrality values for entities associated with monitored trace links. The model layer also generates a call graph by parsing the abstract syntax tree representing a java component in Eclipse IDE. The event layer is responsible for capturing and archiving interaction event sequences generated within a software

project. The log.event component is the clearing centre and data warehouse of all events generated by the project collaborators. The messaging layer carries out asynchronous processing of request/response messages from the server. The offline.emulator component emulates the server end functions of the model and event layers while a developer is generating interaction events in the offline mode. Finally, the RCP layer resides only on the client end, and provides the minimal set of components required to build a rich client application in Eclipse.

Figure 5.7 shows a snapshot of an Eclipse view of the visualisation.rpc component. System developers can open, activate and deactivate their use cases of interest by using the popup menu labelled 7 in Fig. 5.7. All events generated by the developer are traced to the work context of an activated use case. The RCP layer is also responsible for generating visualisations of requirements traceability networks of developers, artefacts and use cases. A system developer using the button labelled 3 in Fig. 5.7 triggers the generation of the traceability network shown in Fig. 5.8. The size of each node corresponds to its centrality in the traceability network. 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.

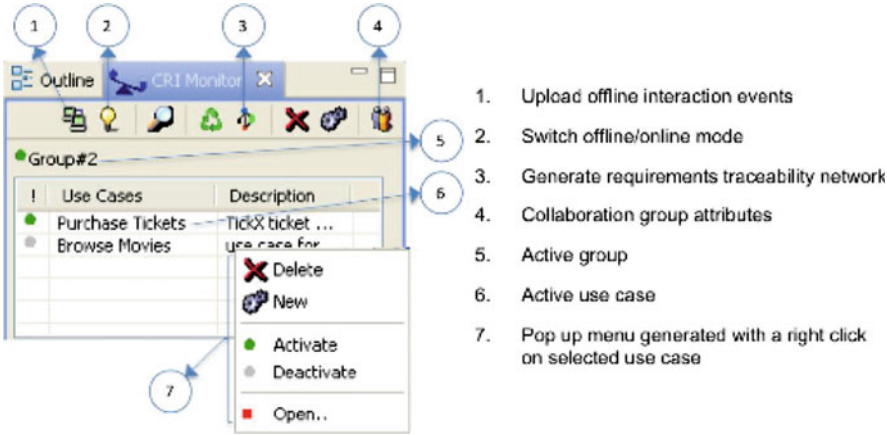


Fig. 5.7 Snapshot of eclipse view of visualization components

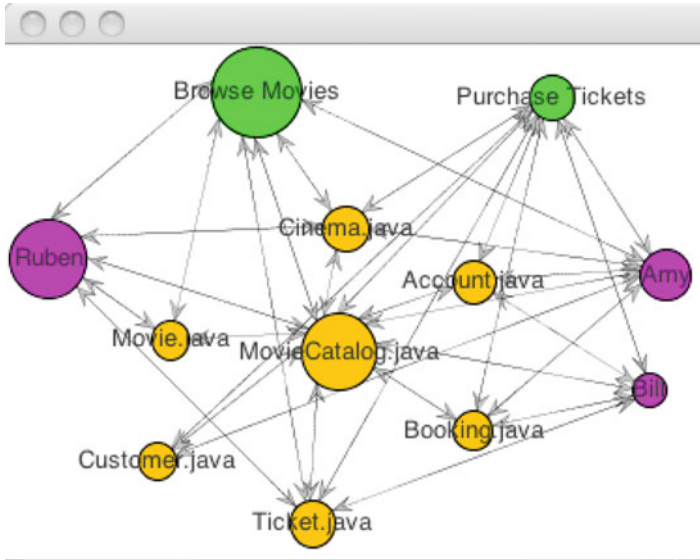


Fig. 5.8 Trace graph for TickX

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.

5.4 Preliminary Study: Inferring Architectural Representations from Traceability Networks

In this section, we aim to provide possible avenues to addressing the questions on architectural information needs discussed in Sect. 5.2.2. We achieve this by discussing insights from the use of traceability networks to infer architectural representations. The discussion is based on a repository of event based requirements traceability networks and call graphs generated during a six weeks study involving ten software engineering students. The students were in the third year of their Masters/Honours programme. 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 [11]. During the study, use cases and system features were modelled and tagged with meaningful short form descriptions or acronyms that were easy to understand by the collaborators. Furthermore, to minimize intrusion and closely mimic real collaboration scenarios, use cases and system features were defined by developers and subsequently used as a basis for tasks assignment. At the end of the 6 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/system features and code artefacts that the participant had worked on. All data were anonymized 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. The remaining part of this section first presents how traceability networks are used to provide insight on architectural styles, then how they help validate initial system decision and identify potentially overloaded components, critical bottlenecks and information centres with ensuing architectural implications.

5.4.1 Understanding Architectural Style

Our expectation is that layouts of architectural styles are unfolded and realised with the accumulation of trace events generated by stakeholders. Thus, if traceability networks harvested from events associated with the achievement of system features and desired requirements is realised, then it is also possible to infer the architectural style used to realize the specified feature or system requirement.

Insights were obtained from our initial study on the inference of architectural styles from event based traceability networks. Figure 5.9 demonstrates a traceability network for the feature ‘File Demo’ in Gizmoball (a feature requirement that users should be able to load gizmos from file). The figure shows the different code artefacts that the developer ‘Tony’ used to realize the desired feature and the trace

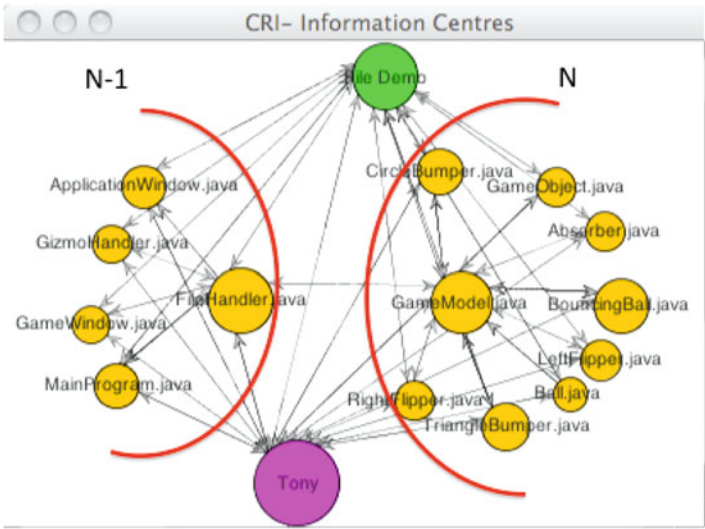


Fig. 5.9 Revealed architectural styles associated in the achievement of gizmoball feature – ‘File Demo’

links between the artefacts. A visual arrangement and repositioning of the artefacts in the traceability network reveals that a 2-tier architectural style is being used by Tony to achieve ‘File Demo.’ Furthermore, each of the tiers reveals a possible blackboard approach. This example demonstrates how different architectural styles can be combined to achieve a specified system feature.

It is important to note that we do not claim here that the discovery and combination of architectural styles is trivial. While some styles such as n-tier or batch sequential are more easily recognised from visualisation of traceability networks, other styles such as the blackboard requires more investigation. Also, call graphs in non trivial cases does not provide the information needed to infer styles. An example is in cases where communication between the clients of a blackboard and the blackboard could be via data sharing, middleware, or network communication. Secondly, the traceability network in Fig. 5.9 demonstrates the pivotal role displayed by the artefacts FileHandler and GameModel in realising the architectural style associated with File Demo. The two artefacts are responsible for the linking of the two different blackboard styles to reveal a 2-tier architectural style. This becomes obvious due to our use of different node sizes based on centrality, thus demonstrating the advantage of this visualization.. Furthermore, for every new link amongst artefacts that is subsequently introduced by collaborators to the network, the trace network reveals corresponding adaptation that is required in the initial architectural rationale for the associated feature of the system.

This study also reveals that traceability networks for non-trivial projects can be overwhelming with hundreds or thousands of components. The implied architectural style used to achieve ‘File Demo’ was revealed by a simple manual visual rearrangement of existing nodes in the network. To give support for bigger projects, further work is needed to focus on the automatic machine learning of architectural styles based on a given traceability network.

5.4.2 Monitoring Initial System Decision and Identifying Critical Pointers

One of the important lessons learned from the repository of event-based traceability networks during the 6 weeks study, is related to information that can be derived from an entity’s centrality measures. An entity’s centrality is useful in revealing a number of latent properties in the trace relation between requirements, code components and the underlying system/software architecture. For instance, a high centrality measure for a developer may suggest that they are working with many parts of the system. Such high centrality for developers can further suggest that the components and system features they are working on are crucial to achieving the system and hence are central to the development process.

The study showed that stakeholders built a perception of their expected centrality measures for entities in the trace network. These expectations are envisaged

based on previous decisions made on achieving the system. Such expectations are then used to monitor the state of the system. An example is the traceability network shown in Fig. 5.10 and involving collaboration between Greg, Boris and Blair to achieve Gizmoball. Two project stages were identified – ‘From Demo to Final’ (Translate game demo to final mode), and *JUnit Tests* (generate test cases for each gizmo object). Forty five 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. A snippet from Boris demonstrating insight he obtained while navigating the traceability network generated as a result of their collaboration (Fig. 5.10) is shown below:

Boris: ...If we have done ‘JUnit Test’ how come it only relates to 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.

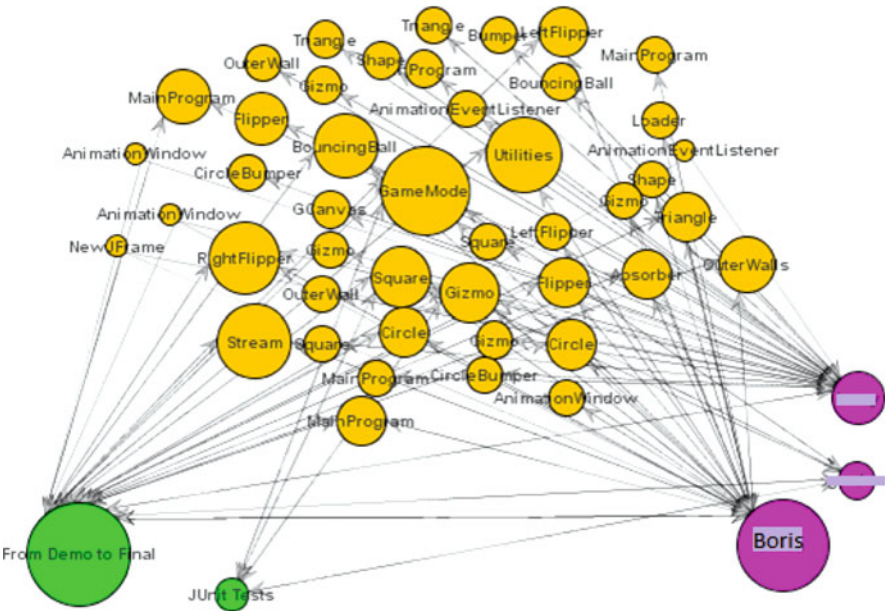


Fig. 5.10 Requirements traceability network with 44 code artefacts

If most developers tend to be associated with a high and equal measure of centrality, then it might imply a shared code ownership development model such as extreme programming. In this case, the architectural design rationale associated with extreme programming practices can then be assumed. This scenario is demonstrated in the case where the centralities of Tony, Alex and Luke in relation to the use case ‘Build Mode’ were closely similar. Transcripts from the interview session confirmed that the three collaborators all worked together in an interchanging pair fashion to realise the ‘Build Mode’ feature of Gizmoball.

The initial study also showed that the requirements traceability network helped to reveal issues that developers would easily have overlooked. For instance, interview transcripts from the collaboration between Luke, Alex and Tony to achieve the Gizmoball project suggested that they used the graph to visualize where the bigger challenges in the system were. The centrality of nodes in traceability networks were also used by the group to get a grasp of which use case or system feature had changed more considerably recently or over the lifetime of the project. Finally, it can be expected that if a requirements use case or system feature has a high centrality relative to other use cases, then this can indicate its importance to the development process. On the other hand it might indicate poor architectural design and use case definition/allocation practice -for instance, the use case has not been broken down enough or the architecture has not been well segmented. Figure 5.11 demonstrates an example of poor segmentation and allocation of components to system features. The feature ‘User Interface’ clearly attained a higher centrality measure relative to other system features. Further insight on the artefacts associated with the identified system feature revealed that it was associated with components necessary for realizing build mode (configuration of gizmos) and play mode (running of gizmos), which are the two main interfaces through which a user can interact with the gizmoball game. This suggests that the ‘User Interface’ feature could more appropriately be further decomposed into two other system features.

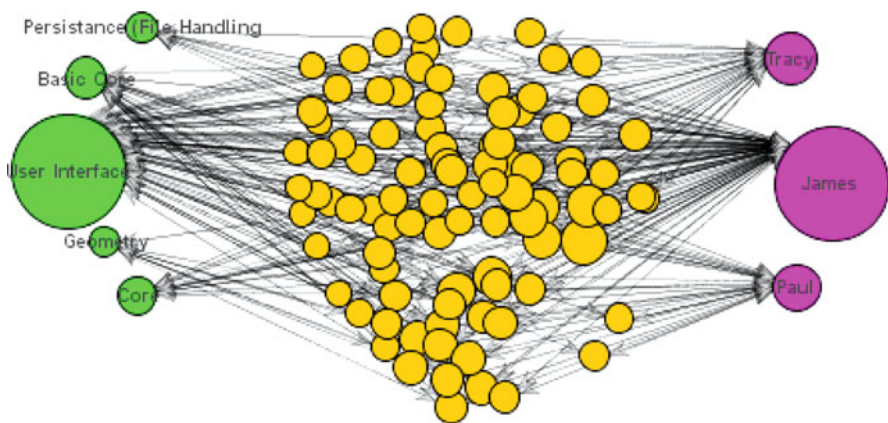


Fig. 5.11 Requirements traceability network involving 92 code artefacts, five system features and three developers

5.4.3 *Lessons Learned and Limitations of Traceability Approach*

An advantage of our approach is that requirements traceability links are automatically harvested and constantly updated to reflect the current state of the project. Furthermore, entities that are more likely to hold greater information about the project are emphasized by their larger centrality values. The use of call graphs is essential to harvesting homogenous traceability links between software components. The focus of this work has been on the use of abstract syntax tree representing a software component to generate its static call graphs. Homogenous traceability links were harvested for top-level static function callers.

A challenge is that the traceability network becomes increasingly cluttered as the number of entities 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 based on centrality has been implemented. Fisheye view has been shown to be an efficient mechanism to enhance clarity for complex visualisations with increasing number of nodes [18]. Another challenge related to scalability is the performance overhead that arises with increasing volume of captured developer interaction events. Finally, the use of interaction patterns to make inference on system decision and identifying critical pointers is based only on the small set of participants in the study. Thus, there is need for more empirical data in subsequent studies.

An implied workflow constraint, based on the implementation of the traceability model, is that systems analysts and developers explicitly need to be working within the context of a selected system feature. This is achieved by activating the desired features or use cases within the development tool. Insight obtained from the initial study suggests that such workflow constraint can sometimes be difficult to achieve, especially when developers have strict project schedules. Feedback from our 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 might forget to formally carry out the use case activation processes within Eclipse IDE. Also coding on a real project would not necessarily be for a specific use case, but “utility” code needed by other modules such as generic data access or manipulation routines.

5.5 Related Work

There are some methods and guidance available that help in the development and tracing system requirements into an architecture satisfying those requirements. The work presented by Grünbacher et al. [19, 20] on CBSP (Connector, Bus, System, Property) focuses on reconciling requirements and system architectures. Grünbacher et al.’s approach has been applied to the EasyWinWin requirements negotiation

technique and the C2 architectural models. The approach taken in our work differs from Grünbacher et al. as our focus is rather on the use of requirements traceability approach to help collaborating developers understand the architectural implications of each action they perform.

A closely related work is that presented on architectural design recovery by Jakobac et al. [21–23]. The main motivation for their work is based on the frequent deviation of developers from the original architecture causing architectural erosion – *a phenomenon in which the initial architecture of an application is (arbitrarily) modified to the point where its key properties no longer hold*. The approach assumes that a given system’s implementation is available, while the architecturally relevant information either does not exist, is incomplete, or is unreliable. Jakobac et al. then used source code analysis techniques for architectural recovery from the systems implementation. Finally, architectural styles were then leveraged to identify and reconcile any mismatch between existing and recovered architectural models. A distinction of our work from Jakobac et al. approach is the associations of requirement use cases or desired system features to the subsequent tangible architectural style used to realize the feature or use case. Furthermore, our traceability links are harvested real time as the system is being realized. Harvested traces are subsequently used to provide developers with information about the revealed architecture based on the work that is currently carried out. We provide pointers to potential bottlenecks and information centres that exist as a result of an initial architectural rationale.

There are a number of other reverse engineering approaches by which the architectures of software systems can be recovered. For instance, the IBIS and Compendium originating from the work of Werner and Rittel [24], presents the capability to facilitate the management of architectural arguments. Mendonca and Kramer [25] presented an exploratory reverse engineering approach called X-ray to aid programmers in recovering architectural runtime information from a distributed system’s existing software artifacts. Also, Guo et al. [26] used static analysis to recover software architectures. Guo et al.’s approach extracted software architecture based on program slicing and parameter analysis and dependencies between the objects based on relation partition algebra. However, these approaches do not directly focus on how such extracted architectures are related to stakeholders’ requirements of the system. Again, there are different approaches to harvesting traceability networks. This research has focused on an event based approach for automated harvesting of heterogeneous relations, and call graph to retrieve homogenous trace links between components achieving the system. Other automated mechanisms for harvesting traceability networks include the use of information retrieval mechanisms and scenario driven approach. Traceability networks generated from information retrieval techniques are based on the similarity of terms used in expressing requirements and design artefacts [27–29]. The scenario-driven approach is accomplished by observing the runtime behaviour of test scenarios. Observed behaviour is then translated into a graph structure to indicate commonalities among entities associated with the behaviour [30].

Mader et al. [24] 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 research 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.

5.6 Conclusion and Further Work

This chapter was motivated by the potential of requirements traceability to understanding architectural representations, responding to some typical architectural information needs during a software project lifecycle. It has presented a technique for the automatic harvesting of traceability networks for inferring architectural rationale. Our technique is based on the use of event-based mechanisms to capture heterogeneous trace links, while call graphs are used to generate homogenous traceability links between components. The heterogeneous and homogenous trace links were then combined to form a unified traceability network of system components, use cases/desired system features and stakeholders (developers) of the system. The advantage of our approach is that the relative potential and architectural implications of each node in the traceability network can then be determined.

An evaluation using a prototype tool implementation has demonstrated the usefulness of our approach. Using event data captured from a student-based project carried out over 6 weeks, we demonstrated how traceability networks are used to provide insight on architectural styles. We also detail how the participants in our study used the traceability tool to understand the architectural implications of the different interaction events carried out during their project. Such architectural implications included impact of executed events on initial system decision and also identifying bottlenecks and information centres in the software project.

The focus of further work is twofold. First, we aim to investigate the accuracy of centrality values. This involves understanding the effect various tasks (e.g. maintenance, debugging, refactoring or simply forward engineering) on centrality of entities. Second, for non-trivial projects, traceability networks can be overwhelmingly complex. Thus, we aim to focus on enhancing the process of inferring architectural rationale, offering a machine learning approach to supplement manual analysis. We also plan to find ways to gain better insight from the complex traceability networks resulting from non-trivial projects.

References

1. Galster M, Eberlein A, Moussavi M (2006) Transition from requirements to architecture: a review and future perspective
2. Omoronyia I et al (2009) Use case to source code traceability: the developer navigation view point
3. Turner CR, Fuggetta A, Lavazza L, Wolf AL (1999) A conceptual basis for feature engineering. *J Syst Softw* 49(1):3–15
4. Eden AH, Kazman R (2003) Architecture, design, implementation. ICSE, Portland
5. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison Wesley, Reading
6. Palmer JD (1997) Traceability. In: Thayer RH, Dorfman M (eds) Software requirements engineering. IEEE Computer Society Press, Los Alamitos, pp 364–374
7. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. *IEEE Trans Software Eng* 27(1):58–93
8. Egyed A (2003) A scenario-driven approach to trace dependency analysis. *IEEE Trans Software Eng* 29(2):116–132
9. Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: Hofmeister C (ed) QoSA-Quality of software architecture. Springer, Vasteras, pp 43–58
10. Omoronyia I, Ferguson J, Roper M, Wood M (2009) Using developer activity data to enhance awareness during collaborative software development. *Comput Supported Coop Work* 18(5–6 December 2009):509–558
11. Omoronyia I (2008) Enhancing awareness during distributed software development. Ph.D. Dissertation, University of Strathclyde, Glasgow, Scotland
12. Fritz T, Murphy GC, Hill E (2007) “Does a programmer’s activity indicate knowledge of code?” in ESEC/SIGSOFT FSE 341–350
13. Gutwin C, Greenberg S, Roseman M (1996) Workspace awareness in real-time distributed groupware: framework, widgets, and evaluation. In: BCS HCI, London, UK, pp 281–298
14. Brandes U, Erlebach T (2007) Network analysis -methodological foundations – introduction, ser. Lecture notes in computer science. vol 3418. Springer-Verlag, Berlin (2005)
15. Latora V, Marchiori M (2007) A measure of centrality based on network efficiency. *New J Phys* 9:188
16. White S, Smyth P (2003) Algorithms for estimating relative importance in networks. In: Getoor L, Senator TE, Domingos P, Faloutsos C (eds) KDD. ACM, Washington, pp 266–275
17. Java universal network/graph framework. [Online] Available: <http://jung.sourceforge.net>
18. Hornbaek K, Hertzum M (2007) Untangling the usability of fisheye menus. *Acm Transactions On Computer-Human Interaction* 14: 2
19. Grünbacher P, Egyed A, Medvidovic N (2001) Reconciling software requirements and architectures: the CBSP approach. Fifth IEEE international symposium on requirements engineering (RE’01)
20. Grünbacher P, Egyed A, Medvidovic N (2000) Dimensions of concerns in requirements negotiation and architecture modelling
21. Jakobac V, Medvidovic N, Egyed A (2005) Separating architectural concerns to ease program understanding. *SIGSOFT Softw Eng Notes* 30(4):1
22. Jakobac V, Egyed A, Medvidovic N (2004) ARTISAn: an approach and tool for improving software system understanding via interactive, tailorable source code analysis, TR USC-CSE-2004-513. USC, USA
23. Medvidovic N, Egyed A, Gruenbacher P (2003) Stemming architectural erosion by coupling architectural discovery and recovery
24. Mader P, Gotel O, Philippow I (2008) Enabling automated traceability maintenance by recognizing development activities applied to models., pp 49–58

25. Mendonça N, Kramer J (2001) An approach for recovering distributed system architectures. *Automated Softw Eng* 8(3–4):311–354
26. Guo J, Liao Y, Pamula R (2006) Static analysis based software architecture recovery, computational science and its applications – ICCSA 2006
27. Antoniol G et al (2002) Recovering traceability links between code and documentation. *Softw Eng IEEE Trans* 28(10):970–983
28. Oliveto R (2008) Traceability management meets information retrieval methods: strengths and limitations. In: *Proceedings of the 2008 12th European conference on software maintenance and reengineering*. IEEE Computer Society, Athens, Greece
29. Lormans M, van Deursen A (2005) Reconstructing requirements coverage views from design and test using traceability recovery via LSI. In: *Proceedings of the 3rd international workshop on traceability in emerging forms of software engineering*, ACM, Long Beach
30. Egyed A (2006) Tailoring software traceability to value-based needs. In: Stefan Biffl AA, Boehm B, Erdogmus H, Grünbacher P (eds) *Value-based software engineering*, Egyed A., Springer-Verlag, pp 287–308

Relating Software Requirements and Architectures

Avgeriou, P.; Grundy, J.; Hall, J.G.; Lago, P.; Mistrík, I.
(Eds.)

2011, XXVIII, 387 p., Hardcover

ISBN: 978-3-642-21000-6