2017 IEEE 25th International Requirements Engineering Conference

# What Requirements Knowledge Do Developers Need to Manage Change in Safety-Critical Systems?

Micayla Goodrum, Jinghui Cheng, Ronald Metoyer, Jane Cleland-Huang
Computer Science and Engineering
University of Notre Dame
{Micayla.J.Goodrum.1; jcheng3; rmetoyer; JaneClelandHuang}@nd.edu

Robyn Lutz
Computer Science
Iowa State University
rlutz@iastate.edu

***Abstract*—Developers maintaining safety-critical systems need to assess the impact a proposed change would have upon existing safety controls. By leveraging the network of traceability links that are present in most safety-critical systems, we can push timely information about related hazards, environmental assumptions, and safety requirements to developers. In this work we take a design science approach to discover the informational needs of developers as they engage in software maintenance activities and then propose and evaluate techniques for presenting and visualizing this information. Through a human-centered study involving five safety-critical system practitioners and 14 experienced developers, we analyze the way in which developers use requirements knowledge while maintaining safety-critical code, identify their informational needs, and propose and evaluate a supporting visualization technique. The insights proposed as a result of this study can be used to design requirements-based knowledge tools for supporting developers' maintenance tasks.**

## I. Introduction

Long-lived software systems evolve over time for many reasons, and therefore must undergo maintenance activities. These activities are performed to correct defects, accommodate new or updated hardware, adapt to changes in adjacent software systems, support new features, and/or modify or remove existing ones [31], [23]. Especially in safety-critical systems, such changes must be introduced carefully. For example, organizations such as NASA have change boards in place which require proposed changes to be analyzed for safety and approved by that board before they are implemented [31]. In such an environment, software change may trigger a repetition of the life-cycle tasks including requirements analysis, design, code, and test prior to deploying the change.

However, there are increasingly many safety-critical software systems on the market and a broad spectrum of rigor at which their software is maintained [8], [32]. While agencies such as NASA have rigorous safety-controls in place, some organizations developing medical devices, communication signals, electronic healthcare systems controlling prescriptions, or unmanned aerial vehicles (UAVs) use less formal processes. Safety-related problems are often introduced when new versions of a product are released to the market. For example, the US Food and Drug Administration (FDA) reported that at one point every Medical Infusion Pump on the US market had been recalled for safety reasons, and further that these recalls were made against new releases of the product – not the original product [27]. Depending on budget and current culture of those organizations, the informal processes may continually be adopted. Thus, supporting developers in managing changes to avoid safety risks within the informal environments has become a prominent issue.

### A. Problem Motivation

In this paper, we investigate developers' information needs for making informed software maintenance decisions on operational, deployed systems. The NASA safety handbook states that the "most common safety problem" during the maintenance phase is "lack of configuration control" which results in "undocumented and poorly understood code." The problem is often caused by the use of disconnected tools. For example, programmers may modify code or models in an Integrated Development/Modeling environment without the benefit of understanding how specific sections of code trace back to safety-related requirements, hazards, and environmental assumptions. Without this knowledge, they may inadvertently make changes to the code which compromise established safety controls (i.e. implementation of a safety requirement in software, hardware, or process to detect, identify or mitigate a hazard).

The solutions for addressing such problems are highly dependent on the development process. For example, in safety-critical products developed using formal methods, source code may be entirely generated automatically from models, and requirements specified as safety properties and proven against the models. As a result, the development is focused on modifying and maintaining the models, rather than the code. However, formal methods are not yet widely adopted, and the majority of safety-critical systems are currently developed using standard development processes augmented with some safety-related practices.

The work we describe in this paper focuses on safe maintenance of software, without the benefit of formal methods. We examine the human side of the problem and seek to discover the information that developers need

2332-6441/17 $31.00 © 2017 IEEE
DOI 10.1109/RE.2017.65

in order to make informed decisions when maintaining a safety-sensitive code base. The current body of work to date has emphasized analysis of code and its dependencies during software maintenance [40], [26], [39], [6]; however, in this study we focus on the role of requirements-related knowledge defined by Maalej et al., as "implicit or explicit information that is created or needed while engineering, managing, implementing, or using requirements" in ways that aid in "answering requirements-related questions" [25]. Requirements knowledge includes diverse areas including "engineering" knowledge which focuses on the content of requirements and their associated artifacts.

### B. Research Overview

To guide our study we pose two research questions:

**RQ1:** What artifacts do developers access for the requirements knowledge they need to make an informed decision in changing safety-related code?

**RQ2:** How should that information be presented and/or visualized to developers in order to aid their analysis task?

To answer these questions we adopted the Design Science approach, which is a solution-oriented research method aimed at investigating artifacts within the context of their use [41]. Design Science is an iterative process in which researchers repeat between design activities, aimed at improving a contextualized problem, and empirical studies that answer knowledge questions and evaluate the proposed solution. This approach is particularly appropriate in investigating visual representations which also requires a highly iterative approach [28]. Our study included two iterations of this process and was initially informed by in-depth interviews with safety-critical system practitioners; Fig. 1 provides an overview of our study process.

The remainder of this paper is laid out as follows. Section II discusses safety-critical project environments. Sect. III describes the results of the practitioner interviews that laid the foundation for our initial design solution. In Sect. IV we introduce the case environment and describe the three change requests used in our study. Sects. V and VI describe the first and second iterative cycles of design activities and user studies. Finally, Sect. VII discusses related work, Sect. VIII discusses threats to validity, and Sect. IX describes our conclusions.

## II. Safety-Critical Project Environments

Safety-critical products are often externally regulated with specific development guidelines. For example, technical guidelines DO-178B/C [1], [37], ISO 26262-6 [18], ECSS-E-40 [9], and FDA [12] are applicable to the industrial domains of aviation, space, automotive, and medical devices, respectively. All four of these guidelines describe the prescribed development life-cycle including its processes, activities, and objectives to be fulfilled in order to demonstrate software safety [36]. Software traceability that establishes relationships among software artifacts is

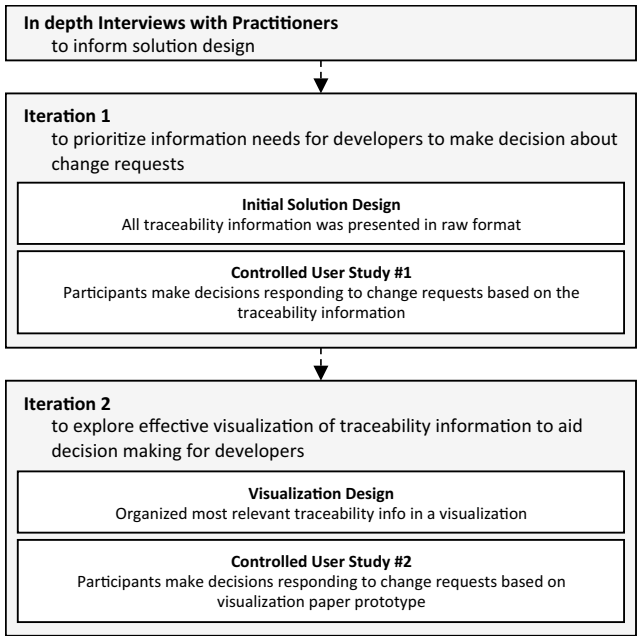



Fig. 1. Our study process implemented using Design Science

one of the objectives. For example, the DO-178B guideline states that for systems at certain levels of criticality "*Traceability between source code and low-level requirements should be provided to enable verification of the absence of undocumented source code and verification of the complete implementation of the low-level requirements.*"

Traceability is achieved in practice through the creation and use of *trace links*, defined as the "specified associations between a pair of artifacts, one comprising the source artifact and one comprising the target artifact" [16], [7]. Software traceability serves an important role in demonstrating that a delivered software system satisfies its software design constraints and mitigates all identified hazards [17]. When executed correctly, traceability demonstrates that a rigorous software development process has been established and systematically followed [12]. For example, the DO178B guidelines prescribe a rigorous hazard analysis to identify potential hazards and contributing faults, so that appropriate mitigating requirements can be specified at both system and software levels. The hazards are risky states that should be avoided [20]. We use FMECA (Failure Modes, Effects and Criticality Analysis) in this study to identify the faults that might cause hazards because it is widely used in safety-critical projects [20], [24].

In practice, trace links and their associated artifacts are often neither readily accessible to developers and other project stakeholders [16], [17], [11], nor integrated into the IDE that developers are using to make changes. As a consequence, developers must actively search for safety-related information needed to evaluate change requests and to safely modify the software product.

## III. Practitioner Interviews

To lay the foundation for an initial solution, we conducted semi-structured phone interviews with five professionals in safety-critical systems industries to understand their perspectives about change management. Borg et al. have conducted an interview-based study with practitioners to understand their current practices, attitudes, and needs around change impact analysis [3]. The purpose of our interviews, however, is tailored to the research questions of this study. Specifically, we aim to understand practitioners' perspectives concerning the support developers need in order to make informed decisions when responding to change requests in safety-critical systems.

Two of our interviewees were from the automotive industry, while the remaining three were from consulting and engineering service companies that cover projects in the automotive, railway, and aviation domains. The job responsibilities of our interviewees varied and included two software developers, two safety analysts, and one research engineer. Participants' experience in the safety-critical systems field ranged from six to 25 years. All five interviewees were based in Europe; two were from Belgium and the remaining three were from Portugal, the United Kingdom, and Sweden respectively. During the interviews, we asked the participants to reflect on their experience in change management and specifically to consider safety-related information that developers might use in order to implement source code changes without negatively impacting system safety. The interviews were audio-recorded and later fully transcribed. We inductively coded the interviews to identify prominent themes [29]; this process involved assigning words/phrases as tokens (i.e. codes) to describe important portions of participants' answers (i.e. quotes) and iteratively combining the codes into higher-level concepts until notable patterns and themes emerged.

### A. Interview Findings

We identified two top-level themes in the interviewees' discussion about change management: (1) factors they considered as challenges/problems in change management and (2) approaches/solutions they believed to be necessary to overcome those challenges and problems.

*1) Challenges and Problems:* Our interviewees discussed factors associated with the challenges and/or problems of change management that fell into one of the following two categories:

• *Understanding the context of the code change.* Three interviewees mentioned that it is usually challenging for developers to see the big picture and understand interconnections among the code to be changed and other related software artifacts. For example, a software developer said: "*When you are coding, you are looking at your current function, or current two or three functions that you are dealing with. But you are not seeing the whole picture.*" Another developer who also had management responsibilities voiced concerns that the trace links among software artifacts are usually not explicitly documented: "*The key in the end is the meta-model of system engineering that expresses [artifact] dependencies. But I think it is a big difference with most approaches where these dependencies are implicit. They are not always explicit.*"

• *Dealing with complexity.* Three interviewees also considered the complexity of safety-critical systems as the most prominent challenge in change management. They mentioned the difficulty of achieving non-interference between safety-sensitive and non-safety-related code. For example, a developer of automotive electronic systems said, "*If I ... overwrite some kernel data structures, then I am in trouble – even if that code is not safety critical because it only turns on the LED.*" A safety analyst from the railway and aviation domains also mentioned that complexity posed challenges in reliable safety analysis related to change impact: "*There are techniques and those safety analyses that you can do. But still many of them are human-based and expert-based analysis. So if you have a less experienced person, it will probably miss many of the situations.*"

*2) Proposed Solutions:* To address those challenges and problems, our interviewees proposed several solutions that we categorized into three major themes:

• *Establishing artifact dependency.* Four interviewees emphasized the importance of establishing and maintaining dependencies among software artifacts to ensure reliable change management. For example, a research engineer focused on automotive systems said, "*If we know about the safety goal and safety requirements of a functionality, then we can actually allocate those safety goals and safety requirements to different levels – to a part of software component or a particular C function that realizes part of that. ... Then the developer would be clearer what he or she should do to comply with that requirement.*"

• *Reinforcing analysis and testing.* Three interviewees emphasized the importance of reinforcing safety analysis and testing to address software errors related to change. For example, a safety analyst mentioned, "*There are static analysis tools that already provide some good feedback about possible areas where problems could happen. ... One of [the other] techniques we used is that we developed a tool for doing fault injection. It is a more dynamic analysis.*"

• *Addressing process and methodology issues.* Two interviewees mentioned that it is important to "*follow the standards*", "*putting the experienced people doing the job*", and maintaining "*easily-understandable documentations*".

### B. Lessons Learned

Several comments were particularly relevant to our study and therefore influenced the design of the initial solution. In particular the observation that developers had difficulty understanding the broader context of the code including its dependencies across other artifacts was noteworthy. The practitioners also described code complexity and its dependencies as problems. Their input guided our

decision to focus the first design/study cycle on discovering which artifacts were of particular importance to developers as they made source code maintenance decisions. Our goal was to focus subsequent cycles on investigating ways to display artifact and traceability data.

## IV. User Study Environment

One of the non-trivial challenges facing a study of this nature is the lack of publicly accessible safety-critical project data that includes basic artifacts such as hazard analyses, faults, safety requirements, functional requirements, environmental assumptions, and source code. Two of the authors previously invested significant time searching for such project data and directly contacting researchers who described datasets in their papers. While there are several active projects, such as the multi-university project focusing on Medical Infusion Pumps [2], the available artifacts are disjointed and/or do not include more than cursory source code. For purposes of this, and other related studies, we are therefore developing a non-trivial software application in the safety critical domain.

### A. Dronology

*Dronology* is an integrated environment currently under development at the University of Notre Dame[1]. Dronology is designed to coordinate the flights of drones, i.e. Unmanned Autonomous Vehicles (UAVs). It supports both virtual and physical drones and aims to ensure their safe operation in shared airspace. The system is considered safety-critical because it could be deployed in populated areas where incidents such as midair drone collisions or hazardous landings could cause personal harm. For purposes of this study we used Version 0.02 which included 13 unique hazards, 44 requirements (of which 17 were safety related), 14 environmental assumptions, 32 java classes consisting of 2,070 lines of code and a total of 158 trace links between various pairs of artifacts as depicted in the Traceability Information Model (TIM) shown in Figure 2.

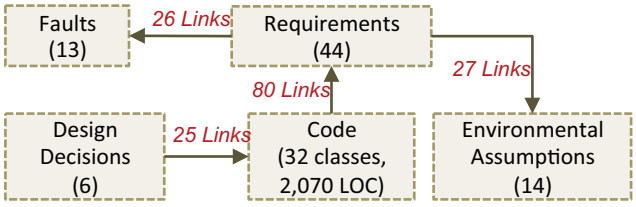



Fig. 2. Artifacts and trace links constructed for Dronology

The user studies we conducted were based upon three specific change requests applied to the Dronology System. We therefore describe each of these requests, and their potential safety impacts, in this section. The first task is illustrated with relevant classes, requirements, faults, and design decisions. Trace matrices capturing associations between artifacts are not shown.

[1]Artifacts are available at http://tinyurl.com/Dronology1

TABLE I
Artifacts relevant to Task #1

**IMPACTED JAVA CLASS**
model.flights.Flights.java

**Requirements associated with ..Flights.java**

| | |
|---|---|
| R9 | Each flight plan will be in one of three states: pending, current, completed. |
| R18 | A drone will always be in one of five flight modes: grounded, awaitingTakeOff, taking-off, flying, or landing. |
| R25* | The maximum number of drones flying in the flight zone shall be limited to two. |

* Only R25 is safety related

**Faults related to R25**

| | | | |
|---|---|---|---|
| F6 | Excessive drones in airspace | Algorithms not designed to maintain safety for more than two drones. Accidents result. | Critical |

**Related Design Decisions**

| | | |
|---|---|---|
| D1 | Roundabout algorithm for safety diversion is currently implemented for two drones only. | System will be developed incrementally. |

**Other requirements associated with fault F6**

| | |
|---|---|
| R19* | Minimum separation distance shall always be maintained between all drones. |
| R20* | A safety component shall direct the flight path of any drone which approaches the minimum separation distance of another drone. |
| R22* | When two or more drones approach minimum separation distance violations will be avoided through forming an aerial roundabout. |
| R35* | A central flight manager will coordinate all drone launches. |

* All requirements are safety related

**Classes associated with these requirements**

| | |
|---|---|
| controller.movement.SafetyManager.java | R19,R20,R22 |
| controller.movement.Roundabout.java | R19,R22 |
| controller.helper.Distance.java | R20,R22 |
| controller.flightZone.FlightZoneManager.java | R35 |

### B. Task 1: Updating a Requirement

Currently only two drones are permitted to fly in the Flight Zone at any time. The change request states "*increase the number of drones flying at the same time to five*"; however, this requested change conflicts with requirement *R25* : "*The maximum number of drones flying in the flight zone shall be limited to two.*" In the current system, the variable **maximumAllowedCurrentFlights** constrains the maximum number of drones in flight to 2.

Relevant artifacts are depicted in Table I. The initially impacted class (i.e. Flights.java) is associated with three requirements (i.e. R9, R18, R25) of which only R25 traces back to a potential fault (i.e. F6 in the FMECA) related to *Excessive drones in airspace*. R25 also traces to a design decision (D1) which explains that the design currently only guarantees minimum separation distance for two

drones at a time. Implementing the requested change to increase the number of drones would therefore adversely impact safety. Tracing fault F6 back to additional mitigating requirements (i.e. R19, R20, R21, and R22) and their associated classes indicates that SafetyManager.java, RoundAbout.java, and Distance.java could all be potentially impacted by the change.

### C. Task 2: Tweaking a Performance Requirement

Currently the SafetyManager.java checks to ensure that no additional drones are in the immediate airspace of a drone during takeoff. This complies with requirement R45 which states that *A drone shall not takeoff unless the overhead airspace is clear of other drones.* A method entitled *checkForViolations()* in SafetyManager.java checks the airspace every five seconds for clearance. A change request has asked for the airspace check to be repeated every two seconds. While SafetyManager.java is associated with several safety-related requirements, none of them are relevant to the requested change.

### D. Task 3: Replacing a Requirement

Currently the simulation runs in virtual or physical mode and runtime switching between modes is not supported. A change request asks for "the current restriction on switching modes at runtime" to be removed. The initial impact point is identified as *model.drone.fleet.RunTimeDroneTypes.java*, specifically two methods: (1) *setPhysicalEnvironment* and (2)*setVirtualEnvironment* where the only state transition allowed by the code is the initial setting to either Physical or Virtual mode. Further transitions are prevented. This code traces to three requirements including *R*2: *The system will either be in physical mode or virtual mode but never simultaneously in both* which in turn is linked to the potential fault *F*9 stating that *Physical drones are orphaned when mode is changed to virtual, causing loss of control of Physical drones.* The developer should therefore *not* remove the code that prohibits state transitions between modes without fully understanding the rationale behind the original requirement.

## V. Iteration 1: Focusing on Information Needs

For purposes of the first user study we focused on identifying key artifacts that would be presented to the user. The first iteration was specifically aimed at answering **RQ1:** "What artifacts do developers access for the requirements knowledge they need to make an informed decision in changing safety-related code?"

### A. Design Activity

As described in Section IV, we selected a set of artifacts that are commonly present in safety-critical project environments (see Figure 2). To address RQ1, we made the deliberate decision to present all artifacts in their most basic format. Three types of artifacts were included: (1) source code listings; (2) artifact lists of Requirements (IDs and specifications), Faults (FMECA), Assumptions, Architectural Diagrams, and Design Decisions; and finally (3) trace matrices depicted as simple tables showing associations as links between pairs of artifacts.

### B. User Study

We conducted a think-aloud study [10] in which each participant was asked to perform the three previously described tasks on the Dronology system. The artifacts were presented to the participants on paper.

*1) Participants:* The first user study included seven participants, all of whom had at least one year of industrial development experience and were current computer science graduate students. Prior industrial experience varied between one to 12 years and included software development in IT, biomedical, and pharmaceutical companies.

*2) Procedure:* Each session began with completion of an informed consent document. The session moderator then described the study process and the artifact documents. We encouraged the participants to "think aloud" as they evaluated each of the change requests and to explain whether the change could be made and under what circumstances. The workspace was videotaped and all audio was captured; the sessions were later fully transcribed.

*3) Data Analysis:* We conducted a qualitative analysis [21] to identify themes about (1) general artifacts participants consulted and (2) the 'correctness' in their reasoning process for making decisions in each task. For the second purpose, the research team first leveraged their expertise and familiarity with the Dronology code base to develop a coding scheme that described the elements in a 'correct' reasoning process for each task (i.e. the 'golden path'); we then identified how much of the participants' reasoning process contained those elements.

For both analysis purposes, two researchers first independently examined three tasks from three different transcripts to refine the definitions of codes in the coding scheme. Agreement was calculated using the Jaccard index (i.e. the intersection of two researchers' codes divided by the size of their union). Codes for the two analysis purposes were iteratively refined until at least 60% agreement (a general threshhold for reliability [22]) was reached. Once reliable code sets were determined, a single researcher then applied the codes to all tasks for all participant transcripts.

*4) Findings:* Figure 3 summarizes the access frequency for each artifact type. Recall that our initial interviews with experts indicated that an important element of making decisions in safety-critical software development is understanding the context of the decision. From Figure 3, we see that the most frequently accessed artifacts were Requirements specifications, with 25 accesses. Trace Links, Design Decisions, FMECA, and Starting Code were also accessed frequently (15-17 accesses). Architectural diagrams and Assumptions were accessed the least (9-10).

Figure 4 illustrates artifact to artifact frequency, or how often one artifact type followed access to another

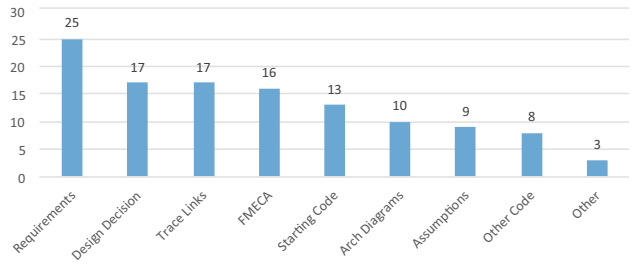
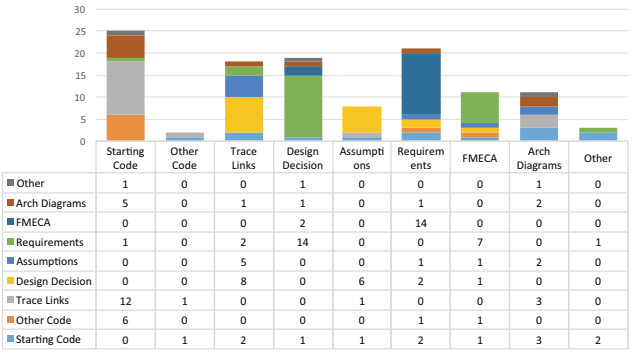



Fig. 3. Frequency of artifact access in Iteration 1 sorted from most to least frequenctly accessed.

artifact type. All participants began each task on the Starting Source Code artifact. From this chart, it appears that a common strategy was to immediately consult the Trace Links, or relationships, from the starting file. The Trace Links most often were followed by Design Decisions and Assumptions. Participants appear to want to delve into the highest level design information immediately and from Assumptions, they typically went directly to Design Decisions as well. From Design Decisions, participants by and large moved to Requirements and from Requirements to FMECA. Alternatively, Design Decisions and Assumptions may have been accesssed often and early because of the order that they were listed in the trace matrix artifact (Assumptions->Design Decisions->Requirements).




| | Starting Code | Other Code | Trace Links | Design Decision | Assumptions | Requirements | FMECA | Arch Diagrams | Other |
|---|---|---|---|---|---|---|---|---|---|
| Other | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| Arch Diagrams | 5 | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 0 |
| FMECA | 0 | 0 | 0 | 2 | 0 | 14 | 0 | 0 | 0 |
| Requirements | 1 | 0 | 2 | 14 | 0 | 0 | 7 | 0 | 1 |
| Assumptions | 0 | 0 | 5 | 0 | 0 | 1 | 1 | 2 | 0 |
| Design Decision | 0 | 0 | 8 | 0 | 6 | 2 | 1 | 0 | 0 |
| Trace Links | 12 | 1 | 0 | 0 | 1 | 0 | 0 | 3 | 0 |
| Other Code | 6 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| Starting Code | 0 | 1 | 2 | 1 | 1 | 2 | 1 | 3 | 2 |

Fig. 4. Frequency of artifact to artifact transitions (i.e. how often participants moved from one artifact to another).

If we assume no file-ordering effects, then a possible preferred strategy is to trace backward to the requirements, identifying design decisions that may constrain their implementations, and then to follow the links to FMECA-identified faults that may give insight into the requirements' rationales. An effort to identify the potential impact of proposed changes appears to guide the pattern of sequential accesses. The variety of access patterns seen in the experiment suggests that availability of traces from requirements directly to multiple other artifacts may help support practitioners' varied approaches to managing change. For changing safety-critical code, the fact that different practitioners consulted the FMECA at different points in their analysis makes providing bi-directional links between requirements and faults especially important.

Table II shows correctness scores for each participant for each task. Correctness was computed as the percentage of necessary reasoning elements mentioned by the participant during the think-aloud. Clearly, some participants performed much better than others, perhaps due to experience with safety critical software or simply experience with professional software development.

TABLE II
CORRECTNESS SCORES (%) FOR ITERATION 1.

| | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** | **P7** | **Ave.** |
|---|---|---|---|---|---|---|---|---|
| **Task1** | 100 | 75 | 0 | 100 | 66.7 | 0 | 100 | 63 |
| **Task2** | 75 | 100 | 0 | 100 | 100 | 0 | 100 | 68 |
| **Task3** | 100 | 100 | 33 | 100 | 0 | 0 | 100 | 62 |

## VI. Iteration 2: Focusing on Organizing Information

The second iteration focused on answering **RQ2:** "How should that information be presented to and/or visualized for developers in order to aid their analysis task?" To structure our visualization design decisions, we utilized the Nested Model framework [30] to guide our design decisions. The first level of the model requires that the designer understand the user's needs through techniques such as interviews or observations. This level was addressed through our interviews described in Section III and through the user study analyses in Iteration 1. This data informed the second iteration of design in which we completed levels 2 and 3 of the Nested Model framework: Data/Task abstraction and Encoding.

### A. Design Activity

The Data and Task Abstraction level of the Nested Model framework requires that we move away from domain language (e.g. software safety analysis) to abstract descriptions of data and user tasks in order to more easily map the existing design problem to known design principles that apply to those abstract data and task types.

*1) Data Abstraction:* While most of the artifacts consist of tabular data with nominal and text attributes (e.g. descriptions), the raw Traceability Matrix represents a network of nodes and relationships where the artifacts are the nodes of the network and relationships are the trace connections.

*2) Task Abstraction:* Task Abstraction requires that we identify tasks that users need to perform on the data. From the initial interviews of industry experts and the think-aloud study of Iteration 1, the most important task was: *Browse the context of the current change request.* In our observations from Iteration 1, this task always started with the current source code file under question and consisted of (1) looking up artifacts of interest to gather more information about them; and (2) identifying artifacts related to the current artifact under consideration. This

process was then typically repeated until the developer felt that she had collected enough evidence to make the decision. Thus the abstract tasks become (1) lookup values in the network and (2) identify paths.

*3) Encoding:* In this level, we seek to choose the most effective visual marks (e.g. points, lines, etc.) and channels (e.g. color, size, etc.) given the dataset type, attribute types and primary user tasks. The network traceability information can be shown in two primary ways; as a matrix, or as a node-link diagram. We chose the node-link diagram because the primary tasks of *browsing the network* and *identifying paths* are more easily accomplished with a node-link representation than with a matrix representation [15], [19]. Based on Iteration 1, we also decided to only show the relevant portion of the traceability data. We use a simple graph traversal algorithm to get the subset of the traceability network that contains all artifacts immediately relevant given the current focus artifact (e.g. the starting source file).

The "Lookup" task requires that the user be able to examine the details of any artifact of interest. We encode the artifact IDs directly into the network representation as named nodes to support identification and we additionally encode the artifact type with color to visually "group" the node types. More detailed information regarding the artifact (e.g. text description of a requirement) can be obtained through interaction. When the user places the mouse over an artifact node, a linked "pane" displays all text information for that artifact. We use a left to right ordering of artifact nodes consistent with the traceability connectivity which is essentially a directed graph from source to Design Decisions and Assumptions or to FMECA via Requirements. Requirements are important as evidenced by how often they were consulted, so they should be presented prominently. We center the graph on Requirements where much of the interaction occurs.

### B. User Study

To evaluate our design, we conducted a Wizard-of-Oz study of a paper prototype implementation shown in Figures 5 and 6. This approach, in which a "wizard" simulates the system's responses to a user in real-time, is particularly appropriate for evaluation of low fidelity prototypes. In this particular study, participants were aware of our intentions to serve as the "wizard" to implement the interactive features for the paper prototype. The wizard (first author of this paper), sat opposite the user as they conducted each of the three Dronology change tasks and manually provided updates to the paper prototype as the user interacted with it (e.g. mousing over a node). Again, particpants were instructed to "think-aloud" during the study to articulate their reasoning process.

### C. Participants

The study included seven participants, all of whom had at least one year of industrial development experience. One participant was a post-doctoral researcher and all others were current computer science graduate students.

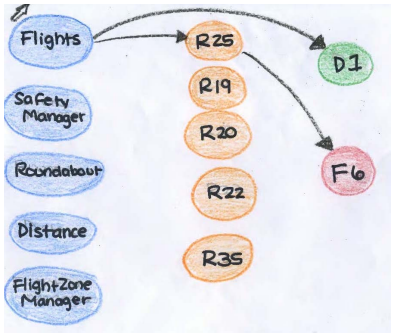



Fig. 5. Paper prototype representation of the network design for Task 1 (See Table I). The network always opens with all nodes, but only showing paths traced directly from the starting source file (always in upper left corner). This provides context without the visual clutter of the edges.

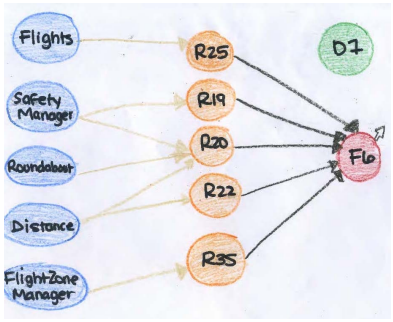



Fig. 6. In this example, the user has placed the mouse over F6. We show all one-step relationships in black and "gray out" all other relationships that lead to that node. Again, this reduces visual clutter while providing relevant context. Information for F6 was shown as text in another "pane" (not shown here).

### D. Procedure and Analysis

The procedure was equivalent to that in Iteration 1 with the exception that participants were given a brief tutorial on the paper prototype design and how to interact with it. They were briefed on the role of the "wizard" and how the study would proceed. We performed a qualitative analysis of the video data exactly as described for Iteration 1.

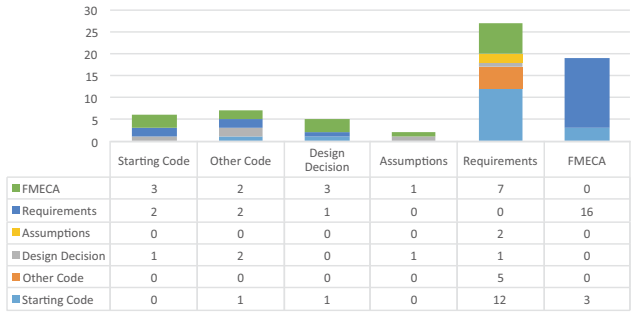
### E. Findings

Figures 7 and 8 and Table III show the access frequency, access order frequency, and the correctness for all participants in Iteration 2 respectively.

TABLE III
CORRECTNESS SCORES (%) FOR ITERATION 2. MISSING DATA FOR P8 WAS DUE TO A VIDEO MALFUNCTION.

| | **P8** | **P9** | **P10** | **P11** | **P12** | **P13** | **P14** | **Ave.** |
|---|---|---|---|---|---|---|---|---|
| **Task1** | 100 | 100 | 0 | 0 | 100 | 0 | 100 | 57 |
| **Task2** | N/A | 100 | 50 | 75 | 100 | 25 | 50 | 67 |
| **Task3** | N/A | 67 | 33 | 67 | 67 | 67 | 67 | 61 |


Fig. 7. Frequency of artifact access in Iteration 2 sorted from most to least frequently accessed.




| | Starting Code | Other Code | Design Decision | Assumptions | Requirements | FMECA |
|---|---|---|---|---|---|---|
| FMECA | 3 | 2 | 3 | 1 | 7 | 0 |
| Requirements | 2 | 2 | 1 | 0 | 0 | 16 |
| Assumptions | 0 | 0 | 0 | 0 | 2 | 0 |
| Design Decision | 1 | 2 | 0 | 1 | 1 | 0 |
| Other Code | 0 | 0 | 0 | 0 | 5 | 0 |
| Starting Code | 0 | 1 | 1 | 0 | 12 | 3 |

Fig. 8. Frequency of artifact to artifact transitions (i.e. how often participants moved from one artifact to another) for Iteration 2.

The frequency of access results are consistent with Iteration 1 in that Requirements again are the most often referenced. Note that TraceLinks are no longer considered because they are implicit in the network and Design Decisions are referenced much less frequently, perhaps due to the 'order bias' no longer existing in the network version. We see a much more prominent pattern of movement between Requirements and FMECA in Iteration 2.

In comparison to Iteration 1 results, and considering our small sample sizes, participants performed similarly in terms of correctness. These results are promising for several reasons. First, our participants were given very little instruction regarding the network design as we hoped to understand how easily they were able learn to use it. Second, the "wizard of oz" approach with a paper prototype is somewhat awkward and it took some time for participants to get comfortable with treating the interface as an interactive "application". Lastly, our design, while based on sound design principles, still suffers from several usability issues that we noticed in Iteration 2 and that will be the subject of the next design and evaluation round. Subjectively, we observed that participants more easily navigated between elements in Iteration 2 and as a result, they tended to explore the connections more thoroughly. Participants also appeared more confident in their decisions in Iteration 2.

Finally, as we suspected in Iteration 1, it is possible that Design Decisions should play a much more prominent role. Figures 9 and 10 show clearly that those who performed well in Iterations 1 and 2 accessed design decisions more often, and source code less often, respectively, than those who performed poorly.

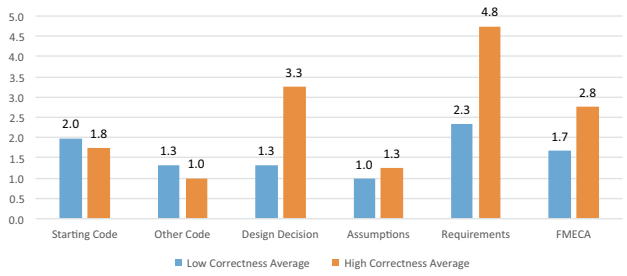



Fig. 9. Frequency of artifact access in Iteration 1 by poor performers and good performers.

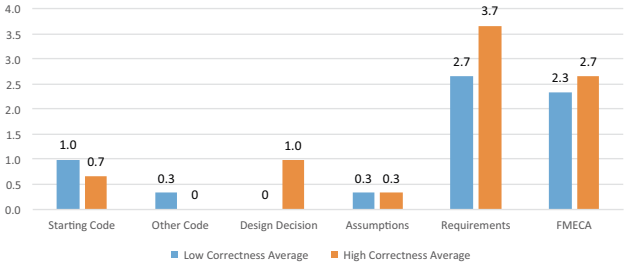



Fig. 10. Frequency of artifact access in Iteration 2 by poor performers and good performers.

## VII. Related Work

Our work is closely related to other studies that focused on (1) understanding industrial practitioners' perspectives on change management, (2) discovering strategies used by developers in maintaining source code, and (3) visualizing software artifacts.

### A. Practitioners' Perspectives on Change Management

A recent paper by de la Vara, et al. reports rich results from a survey of practitioners regarding safety evidence change impact analysis [8]; they surveyed safety analysts ranging from safety engineers to certification authorities. Their survey confirms that software maintenance *occurs regularly in most safety-critical systems.* Nearly a quarter of the survey respondents (20 of 97, or 23.8%) reported that in some of their projects, there was "Modification of a system during its maintenance" involving change impact analysis of safety evidence. Another 22.6% of respondents reported that in "Most projects" or "Every project" there was "Modification of a system during its maintenance" involving change impact analysis of safety evidence. The survey also confirmed the *need for improved support in practice for updating safety artifacts when software changes.* Our work differs from this study in that we focus specifically on the needs of the software developers responsible for maintaining the software system and responding to software change requests.

Similarly, Borg et al. interviewed practitioners from two development teams to understand their current practices, attitudes, and needs around change impact analysis [3]. Their study also confirmed that "information overload"

associated with the complexity of the system and the corresponding artifacts is a prominent challenge in change impact analysis. Additionally, their interviewees voiced needs of tools that help maintain traceability information for change management. Our work specifically focuses on software developers' needs in making informed decisions responding to change requests. We proposed a potential visualization design focused on providing relevant traceability information to software developers.

Similar to our tool-centered approach, Charrada et al. presented a solution to support maintainers by detecting impacted requirements when the source code changes [4]. They propose displaying a ranked list of requirements most likely to have been impacted or highlighting potentially impacted parts of the requirements specification. In contrast, our work, focuses on the needs of the practitioners and the visualization design of a tool to satisfy those needs.

### B. Understanding Software Maintenance Strategies

Many researchers have observed developers at work with the goal of identifying strategies that they use for debugging and maintaining source code. Examples include Mark Weiser's observation in 1982 that programmers use slices when debugging – a fact that is well known now, but was insightful at the time and led to new tools and techniques for automating slicing [40]. Maalej et al. identified strategies that developers used to comprehend source code including a "problem-solution-test" work pattern, and interacting "with the UI to test expected program behavior" [26]. Sutherland et al. observed code annotation strategies of experienced programmers and identified their major motivations for annotating as supporting code navigation, recording working notes, and communicating with co-workers [39]; their findings provided insights for tools that leveraged code annotations. Codoban et al. examined the strategies developer used for understanding software history and proposed a three-lens framework that separates developers' "immediate", "awareness", and "archaeology" needs [6]. They argued that the software history information "should have different representations, based on the age of commits", indicating new considerations in designing version control tools.

While many of the related studies focused on strategies for source-code management and comprehension, our study focuses on strategies for acquiring and leveraging broader knowledge about safety-related requirements, associated hazards, and underlying design decisions. As with these previous studies, we are interested in understanding developers' strategies in order to lay a foundation for tool-building and future research.

### C. Visualizing Software Artifacts

Two common presentation approaches in visualization are Focus+Context and Overview+Detail. Focus+Context representations present a detailed view of the data but maintain the details within the context of the larger overview (e.g. "fisheye" magnification of the Mac OS Dock). Overview+Detail representations provide an overview of the data with detail shown not necessarily in context (e.g. Adobe Acrobat reader left pane page overview and main pane detailed page view).

Most recently, Ghazi et al. presented a requirements visualization design based on the Focus+Context concept. [14]. Their design uses a magnet-based metaphor [38], combined with a fisheye view [13] to allow the user to navigate large graphs and semantically zoom [35] in on areas of focus while zooming out in the areas further from the focus (e.g. where the magnet is located).

Our design employs an Overview+Detail approach where the detail is provided in a separate pane linked through a mouse "hover" interaction. There have been many studies to evaluate the pros and cons of Focus+Context and Overview+Detail approaches [5], articulating benefits of both approaches for specific situations. We chose Overview+Detail because of the known problems associated with distortion in fisheye views and because linked Overview+Detail has been demonstrated to be successful in similar data exploration tasks [34].

## VIII. Threats to Validity

This study investigates the informational needs of developers as they maintain safety-critical systems with the aim of understanding which artifacts and their associated trace links should be presented to developers during change maintenance. There are several threats to validity of this study. First, the numbers of our interviewees and user study participants were limited. However, we have found recurring themes from discussion with our interviewees who had diverse backgrounds, indicating that major and representative issues were captured. Additionally, the design science approach is iterative in nature and even a limited number of participants provides useful usability feedback to iteratively refine a design [33]. Second, our study was conducted in a controlled university setting, rather than in the context of a real project and while our participants had industrial experience, most had not worked on safety critical projects. However, this is appropriate for early phases of discovery. The study is specifically related to interviewed practitioners' needs in maintaining safety-critical systems and to our findings regarding developers' frequency of access to safety information in FMECAs and requirements in such situations. Based on findings in this study, we plan to develop real tools that can be evaluated within the context of industrial projects. Answering fundamental questions about users' informational needs and the way they process information to make change impact decisions is the primary contribution of this paper and an essential precursor to building such tools.

## IX. Conclusion

Results from the two user studies we designed and conducted suggest that the answer to **RQ1:** "What artifacts

do developers access for the requirements knowledge they need to make an informed decision in changing safety-related code?" is that developers tended to use an implicit, sequential strategy. To determine whether a safety-related change can be made without impacting existing safety controls, developers most often accessed the requirements specification. To determine whether specific safety risks would exist if the proposed change was made they tended to access fault information from the FMECA. They used this contextual information to identify what tasks should be performed in order to implement the change. These results are consistent with the concerns expressed by the practitioners in interviews that organizing the information is especially challenging in safety-critical systems.

Results from the user studies suggest that the answer to **RQ2:** "How should that information be presented and/or visualized to developers in order to aid their analysis task?" is that the way the information was organized was important to the developers and that the Overview + Detail approach to visualizing the software artifacts and their relationships gave good results in terms of correctness of actions taken. The contribution of the studies is to identify what practitioners do to meet their needs for information when maintaining safety-critical code and to identify an approach to visualization that can guide tool design in better supporting them going forward.

## X. Acknowledgments

The work in this paper was partially funded by the US National Science Foundation Grants CCF:1647342 and CCF:1513717.

## References

[1] *RTCA/EUROCAE. DO-178B/ED-12B: Software considerations in airborne systems and equipment certification.* 2000.

[2] The Generic Infusion Pump (GIP). https://rtg.cis.upenn.edu /gip/, 2017.

[3] M. Borg, J. L. de la Vara, and K. Wnuk. Practitioners' Perspectives on Change Impact Analysis for Safety-Critical Software – A Preliminary Analysis. In *SAFECOMP Workshops*, pages 346–358, 2016.

[4] E. B. Charrada, A. Koziolek, and M. Glinz. Supporting requirements update during software evolution. *Journal of Software: Evolution and Process*, 27(3):166–194, 2015.

[5] A. Cockburn, A. Karlson, and B. B. Bederson. A review of overview+ detail, zooming, and focus+ context interfaces. *ACM Computing Surveys (CSUR)*, 41(1):2, 2009.

[6] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: A study on why and how developers examine it. In *Proc. of ICSME*, pages 1–10. IEEE, 2015.

[7] CoEST: Center of Excellence for Software Traceability, http://www.CoEST.org.

[8] J. L. de la Vara, M. Borg, K. Wnuk, and L. Moonen. An industrial survey of safety evidence change impact analysis practice. *IEEE Trans. Software Eng.*, 42(12):1095–1117, 2016.

[9] ECSS. ECSS-E-40C: principles and requirements applicable to space software engineering, 2009.

[10] K. A. Ericsson and H. A. Simon. Verbal reports as data. *Psychological Review*, 87(3):215, 1980.

[11] O. G. et al. The quest for ubiquity: A roadmap for software and systems traceability research. In *Requirements Engineering*, pages 71–80, 2012.

[12] Food and Drug Administration. *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*, 2002.

[13] G. W. Furnas. *Generalized fisheye views*, volume 17. ACM, 1986.

[14] P. Ghazi, N. Seyff, and M. Glinz. Flexiview: a magnet-based approach for visualizing requirements artifacts. In *Proc. of REFSQ*, pages 262–269, 2015.

[15] M. Ghoniem, J.-D. Fekete, and P. Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[16] O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of RE*, pages 94 –101, April 1994.

[17] J. D. Herbsleb and M. B. Dwyer, editors. *Proceedings of the on Future of Software Engineering, FOSE*. ACM, 2014.

[18] ISO 26262-6:2011-11: Road vehicles - Functional safety - Part 6: Product development at the software level. International Organization for Standardization, Nov 2011.

[19] R. Keller, C. M. Eckert, and P. J. Clarkson. Matrices or node-link diagrams: which visual representation is better for visualising connectivity models? *Information Visualization*, 5(1):62–76, 2006.

[20] J. Knight. *Fundamentals of Dependable Computing for Software Engineers*. Chapman Hall/CRC, 2011.

[21] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage, 2004.

[22] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. pages 159–174. JSTOR, 1977.

[23] N. G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2012.

[24] R. R. Lutz and R. M. Woodhouse. Requirements analysis using forward and backward search. *Ann. Soft. Eng.*, 3:459–475, 1997.

[25] W. Maalej and A. K. Thurimella, editors. *Managing Requirements Knowledge*. Springer, 2013.

[26] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4):31:1–31:37, 2014.

[27] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang. Strategic traceability for safety-critical projects. *IEEE Software*, 30(3):58–66, 2013.

[28] S. McKenna, D. Mazur, J. Agutter, and M. Meyer. Design activity framework for visualization design. *IEEE Trans. Vis. Comput. Graphics*, 20(12):2191–2200, 2014.

[29] M. B. Miles, A. M. Huberman, and J. Saldana. *Qualitative data analysis: a methods sourcebook*. Sage, 2014.

[30] T. Munzner. A nested model for visualization design and validation. *IEEE Trans. Vis. Comput. Graphics*, 15(6), 2009.

[31] NASA. *Software Safety Standard NASA-STD-8719.13C*. 2013.

[32] R. Nevalainen, P. Clarke, F. McCaffery, R. V. O'Connor, and T. Varkoi. Situational factors in safety critical software development. In *Proc. of EuroSPI*, pages 132–147, 2016.

[33] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In *CHI*, pages 206–213, 1993.

[34] C. North and B. Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *Proc. of AVI*, pages 128–135. ACM, 2000.

[35] K. Perlin and D. Fox. Pad: an alternative approach to the computer interface. In *SIGGRAPH*, pages 57–64. ACM, 1993.

[36] P. Rempel, P. Mäder, T. Kuschke, and J. Cleland-Huang. Mind the gap: assessing the conformance of software traceability to relevant guidelines. In *Proc. of ICSE*, pages 943–954, 2014.

[37] L. Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[38] A. S. Spritzer and C. M. Freitas. A physics-based approach for interactive manipulation of graph visualizations. In *Proc. of AVI*, pages 271–278. ACM, 2008.

[39] C. J. Sutherland, A. Luxton-Reilly, and B. Plimmer. An observational study of how experienced programmers annotate program code. In *INTERACT*, pages 177–194. Springer, 2015.

[40] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[41] R. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.