User Evaluation of a Domain Specific Program Comprehension Tool

Leon Moonen
Simula Research Laboratory
P.O. Box 134, N-1325 Lysaker, Norway
leon.moonen@computer.org

***Abstract*—The user evaluation in this paper concerns a domain-specific tool to support the comprehension of large safety-critical component-based software systems for the maritime sector. We discuss the context and motivation of our research, and present the user-specific details of our tool, called FlowTracker. We include a walk-through of the system and present the profiles of our prospective users. Next, we discuss the design of an exploratory qualitative study that we have conducted to evaluate the usability and effectiveness of our tool. We conclude with a summary of lessons learned and challenges that we see for user evaluation of such domain-specific program comprehension tools.**

***Keywords*-user evaluation, domain specific tooling, program comprehension, software visualization.**

I. Introduction

Component-based software engineering aims to manage the complexity of large-scale software development by *assembling* systems from ready-made parts. It has known benefits on the comprehensibility of the individual components, by separating concerns, reducing coupling, and increasing cohesion [1]. However, the *overall* comprehension of component-based systems is complicated by the fact that the configuration and composition of the components plays such an essential role in the system's behavior. To understand a component-based system, one needs to understand the interplay between its components and configuration artifacts.

In spite of this clear need, we found that there is little support to assist software engineers with this task [2]. Most tools available to professional developers have strict limitations on the programming languages that can be processed. This typically means that information from external configuration artifacts can not be included, effectively inhibiting system-wide analysis and confining it to the boundaries defined by the source code of a single component.

A second comprehension challenge is that it's not always just developers that need to understand what's going on in the code: Our work is motivated by an industrial case where (non-developer) safety domain experts need to understand the logic that is implemented in the system to support software certification. These safety domain experts need to see the system's source artifacts represented in a context that is relevant to them – not just what the code *does*, but what it *means* [3]. Consequently, any views on the system need to be goal-driven, at a suitable level of abstraction, and based on relevant knowledge of the application domain.

We have developed an approach to reverse engineer a fine-grained system-wide dependence model from the source and configuration artifacts of a component-based system [2], and defined a hierarchy of views on this model to visualize these dependencies and possible information flow between inputs and outputs at various levels of abstraction [4]. The views are aimed at supporting both the developers and safety domain experts of our industrial partner in their understanding of the system. The whole process of model reverse engineering and visualization is supported by a prototype tool, FlowTracker, that we would like to evaluate with our users.

II. Context and Motivation

The work described in this paper is part of an ongoing industrial collaboration with Kongsberg Maritime (KM), one of the largest suppliers of programmable marine electronics worldwide. The division that we work with specializes in computerized systems for safety monitoring and automated corrective measures to mitigate hazardous situations, such as emergency shutdown, process shutdown, and fire & gas detection systems for vessels and off-shore platforms. In particular, we study a family of safety-critical embedded software systems that connect control components to physical sensors and mechanical actuators.

Concrete software products are assembled in a component-based fashion from a collection of approximately 30 reusable components, implemented in MISRA C (a safe subset of C [5]). The components are relatively small in size (1-2 KLOC), have a well-defined interface of input- and output ports to connect to other components or the environment, and perform relatively straightforward computations. Their control logic, however, can be rather complex and is highly configurable via parameters (e.g. initialization, thresholds, comparison values etc).

The system's overall logic is achieved by composing pipelines of interconnected component *instances* that receive values from system input ports (sensors) and process it in various ways, such as measuring, digitizing, voting, and counting, before sending the results to system output ports (actuators). Figure 1 shows an example (the colors are not important for now). Components of the same type can be cascaded to handle a larger number of input signals than foreseen in their implementation. Similarly, the output of a pipeline can be put into another pipeline to use the safety conclusions for one area as input for a connected area.

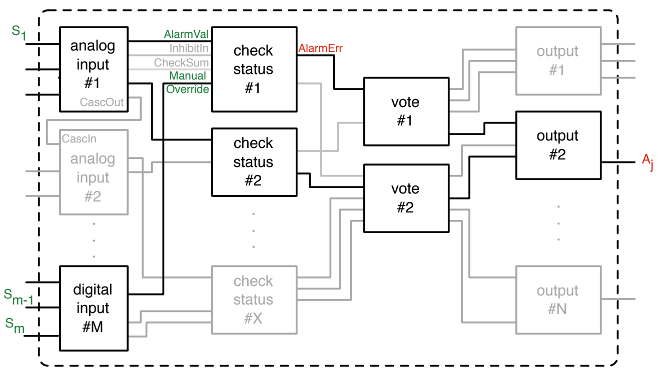


Figure 1. Component composition network for an example system with the System Information Flow for actuator A_j highlighted in black.

Research goal: As installations that are monitored become bigger, the number of sensors and actuators grows rapidly, the safely logic becomes increasingly complex and the induced component networks end up interconnecting thousands of component instances. To give an impression, a typical real-life installation has 12 to 20 stages in each pipeline, and approximately 5000 component instances in its safety system. As a result, it becomes very hard to understand the overall behavior and dependencies in the system. The goal is to investigate feasibility of providing *source-based evidence* to support software certification and assists with the understanding of *deployed systems*, i.e. systems composed and configured to monitor the safety requirements of a particular installation. One of the most important questions in this context is the question *"can signals from the appropriate sensors reach a given actuator?"*.

III. FlowTracker Overview

In theory, the question at the end of Section II can be answered using program slicing [6]: A backward slice of a program contains all parts that potentially affect the values at a given point of interest, known as the slicing criterion [7]. Thus, by examining the backward slice for a given actuator, we can determine which sensors can affect that actuator .

In practice, there are two challenges, given our context: First, program slicing is defined within the closed boundaries of source code, whereas we need to slice *across* the complete component-based system. To this end, FlowTracker analyzes component sources and system configuration artifacts to reverse engineer a fine-grained system-wide dependence graph (SDG) that captures intra- and inter-component dependencies, and can be sliced using standard algorithms [2].

Second, dependence graphs, and slices through dependence graphs, contain many low level details that make them unfit for supporting comprehension or visualization, especially when targeting non-developers [8]. To make this type of information more tractable, we have defined a hierarchy of five abstractions (views), aimed at the needs of safety experts and developers, ranging from a black-box survey of the system, via a number of intermediate levels, to a hypertext version of the source code. FlowTracker constructs these views from the system-wide dependence model via a combination of slicing, transformation and visualization [4]. The views are rendered using HTML and SVG, and are interconnected via hyperlinks to support navigation and enable various comprehension strategies [9].

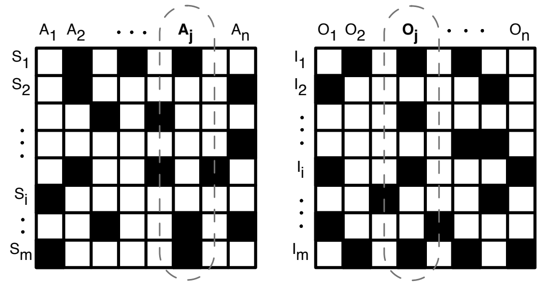
(1) System Dependence Survey: This view shows the dependencies between all system inputs (sensors) and outputs (actuators) in one single matrix, with sensors and actuators as rows and columns respectively (see Figure 2a). A filled cell indicates that there is at least one path along which information can flow from that sensor to that actuator. This view gives a black-box summary that hides all details on *how* the information flow is realized. Engineers can use it to quickly find what sensors can affect a specific actuator, and vice versa. The presentation intentionally resembles our industrial partner's specifications of the safety logic, known as Cause & Effect matrices, to enable easy comparison of the implemented dependencies with the specified safety logic.

The System Dependence Survey serves as a starting point for navigation. To this end, we make the matrix *active* by embedding hyperlinks to corresponding views on the next abstraction level. By clicking one of the cells in the column for a given actuator (e.g. A_j), the user can zoom in on the System Information Flow for that specific actuator.

(2) System Information Flow: This view depicts the inter-component information flow that can affect a given actuator, i.e., there is a diagram for each actuator in the system. The view shows a backward slice through the system with actuator A_j as criterion while *hiding* all intra-component level information. The result highlights the actuator and all related sensors, component instances, and inter-component connections, as was shown in Figure 1 for actuator A_j.

Apart from showing the elements that influence an actuator, this view serves as an intermediate level between system level views and component level views. It includes navigation hyperlinks so that a user can click on a component instance to zoom in on that component, or click outside the diagram to return to a higher level of abstraction.

(3) Component Dependence Survey: Similar to the System



(a) System-wide dependencies

(b) Component dependencies

Figure 2. System- and Component Dependence Surveys.

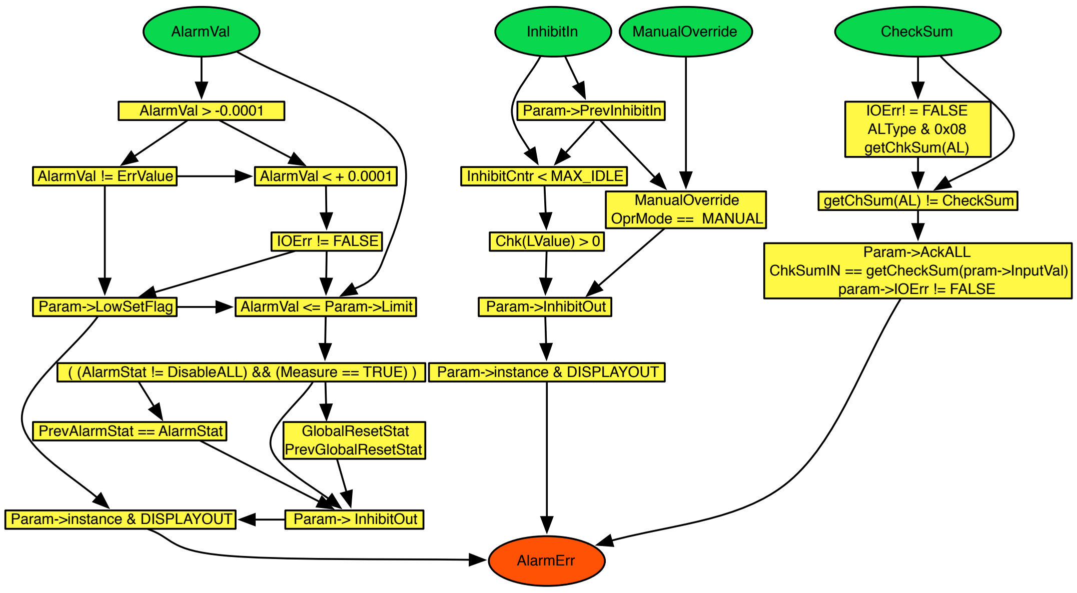


Figure 3. Component Information Flow example

Dependence Survey, the Component Dependence Survey is a black-box view that summarizes the dependencies between a component's input- and output ports using filled cells in a matrix (see Figure 2b). There is one dependency matrix for each component, independent of its instances, because the dependencies are induced by the component source code. Users can navigate to more detailed views by clicking one of the cells to zoom in on the Component Information Flow for the corresponding output port.

(4) Component Information Flow: For a given component and output port, this view shows the *intra-component* information flow from all input ports that can affect that output port (i.e., there is a diagram for each output port of the component). In addition to the in- and output ports involved, the graph includes all conditions that control the information flow towards the selected output port. The view hides all other information (e.g. assignments, computations) and it combines sequences of conditions into aggregated conditions wherever possible to reduce cognitive overhead.

Figure 3 shows an example for output port "AlarmErr" (red node at the bottom). The input ports that can affect AlarmErr are at the top (green nodes) and the conditions that control the information flow are shown as yellow squares. The condition nodes have hyperlinks embedded to navigate to the corresponding location in the source code.

(5) Implementation View: At the lowest level, this view shows pretty-printed source code with hypertext navigation facilities, e.g. cross-referencing of program entities with their definition. It is very similar to the source code in an IDE (besides not being editable) and we foresee that it could be replaced by an IDE in future versions. Higher level views provide links into the source code as a means of traceability and to help minimize user disorientation.

IV. Typical Usage Scenario

1) Users start navigating the system from the System Dependence Survey. In this view, they can immediately identify those sensors that can (or can not) influence a given actuator (Figure 2a);
2) By clicking on one of the actuator columns, the users zoom in on the System Information Flow that helps them find the components and inter-component connections that play a role in transferring the values from sensors to that actuator (highlighted in Figure 1);
3) By selecting on one of the component instances, they navigate to the Component Dependence Survey. This view can be used to identify which input ports can (or can not) affect which output ports (Figure 2b);
4) By clicking on one of the output port columns, the users focus on the Component Information Flow, that shows the conditions that control how information from the input ports can reach the selected output port (Figure 3);
5) Finally, the user can click on one of the conditions to navigate to the corresponding location in the source code for traceability and further (manual) inspection.

V. Towards User Evaluation of FlowTracker

User profiles: We organized a workshop at our partner's site to present FlowTracker to the various stakeholders, and to identify the roles of prospective Flowtracker users:

Module Developers: engineers that develop and maintain the modules of the studied system. Their focus is on individual modules rather than the complete system.

System Integrators: experts in system composition and configuration. Their tasks include checking component inter-connections to verify consistency, and auditing systems to verify that they meet the costumer's needs.

Safety Experts: design the system's Cause & Effect specification(s) with the customer and handle the certification process with third party certifiers.

Exploratory Qualitative Evaluation: Before embarking on a full-blown user evaluation, we performed an *exploratory study* to evaluate the usability and effectiveness of our visualizations for the needs of KM. We conducted a *qualitative evaluation* of the tool with six subjects, selected to match the roles discussed above. Such a pre-experimental design is a cost-effective way to get initial feedback, identify missing functionality and required improvements [10].

We prepared the subjects with a brief 10 min. presentation of FlowTracker that included a walk-through like Section IV. Next, we let the subjects play around with the tool until they felt confident in their understanding of its functionality. We concluded the training with three hands-on exercises which participants had to complete before starting the evaluation. These exercises were designed in a way to engage all the views and the major features of FlowTracker.

The evaluation itself consists of a *structured interview* which was guided by a questionnaire consisting of 30 closed questions using a 5-point Likert scale and 6 open (discussion) questions. The questions varied in positive and negative phrasing to break answering rhythms and avoid steering [11]. Researcher-administered interviews were chosen over self-administered questionnaires to elicit as much feedback as possible. Participants were instructed to bring up any question or comment during the training exercises, questions, and the open-ended discussion, similar to think-aloud sessions. We recorded the complete audio of the sessions (training+interviews). These recordings were independently analyzed by two researchers to avoid interpretation bias. The evaluation results indicate that the prototype was already very useful and a number of directions for further improvement were suggested. For more details, we refer to [4].

VI. Challenges and Lessons Learned

We see a number of challenges around the user evaluation of a domain specific program comprehension tool targeted at a specific industrial audience, such as FlowTracker.

As researchers, we want to develop a sustainable relation with our industrial partner, which requires that we limit the overhead and impact of our research activities on their daily work. A related concern, especially when targeting a small user population, is that the final adoption may be hindered by (negative) *anchoring effects* that result from exposing potential users to (early) research prototypes [12]. As such, we found that the potential for recruiting a statistically significant number of subjects to conduct is rather limited, making it hard to conduct a *quantitative evaluation*.

Our exploratory study is based on six subjects, which, can be argued, is too small to infer generalizable conclusions. We have tried to limit this threat using a *qualitative* study design. In addition, the subjects were selected so that their profiles would match with the various roles of prospective users. Finally, we added a second group of subjects (colleague researchers) to decrease the potential bias that could be caused by only selecting subjects from our industrial partner [13].

A remaining concern is that we include one industrial subject for each of the user profiles. As such, the person gets a dominant voice and the answers (and our conclusions) may be based more on personal opinions than on the role.

We used interviews over self-administered questionnaires to collect more and better quality data. However, the live interaction with the researchers might have led the subjects to give more socially acceptable positive feedback.

Even though the tool was very positively evaluated, we had underestimated how much attention would be drawn to user interface aspects, which we had ignored since our focus was getting the views themselves right. A next time, we would address these issues before evaluation, to improve the subject's focus and avoid distraction.

Acknowledgments: The FlowTracker research is joint work with Amir R. Yazdanshenas. We would like to thank Kongsberg Maritime for the collaboration, and in particular thank the participants in our workshop and exploratory evaluation study for their valuable time and feedback.

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[2] A. R. Yazdanshenas and L. Moonen, "Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems," in *Int'l Conf. Software Maintenance*, 2011.

[3] M. Petre, "Mental imagery and software visualization in high-performance software development teams," *J. Visual Languages & Computing*, vol. 21, no. 3, pp. 171–183, 2010.

[4] A. R. Yazdanshenas and L. Moonen, "Tracking and Visualizing Information Flow in Component-Based Systems," in *Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2012.

[5] L. Hatton, "Safer language subsets: an overview and a case history, MISRA C," *Information and Software Technology (IST)*, vol. 46, no. 7, pp. 465–472, 2004.

[6] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[7] K. Gallagher and D. Binkley, "Program slicing," in *Frontiers of Software Maintenance (FoSM)*. IEEE, 2008, pp. 58–67.

[8] J. Krinke, "Visualization of program dependence and slices," in *Int'l Conf. Software Maintenance (ICSM)*. IEEE, 2004, pp. 168–177.

[9] M.-A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," in *Int'l Ws. Program Comprehension (IWPC)*. IEEE, 2005, pp. 181–191.

[10] D. T. Campbell and J. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Wadsworth, 1963.

[11] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. Continuum, 1992, vol. 30, no. 3.

[12] A. Tversky and D. Kahneman, "Judgment under Uncertainty: Heuristics and Biases." *Science*, vol. 185, no. 4157, pp. 1124–31, 1974.

[13] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *SIGCHI Conf. Human Factors in Computing Systems*, vol. 17, no. 1. ACM, 1990, pp. 249–256.