Tracking and Visualizing Information Flow in Component-Based Systems

Amir Reza Yazdanshenas
Simula Research Laboratory
P.O. Box 134, N-1325 Lysaker, Norway
amir.yazdanshenas@simula.no

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

***Abstract*—Component-based software engineering is aimed at managing the complexity of large-scale software development by composing systems from reusable parts. In order to understand or validate the behavior of a given system, one needs to acquire understanding of the components involved *in combination with* understanding how these components are instantiated, initialized and interconnected in the particular system. In practice, this task is often hindered by the heterogeneous nature of source and configuration artifacts and there is little to no tool support to help software engineers with such a system-wide analysis.**

This paper contributes a method to *track and visualize information flow* in a component-based system at various levels of abstraction. We propose a hierarchy of 5 interconnected views to support the comprehension needs of both safety domain experts and developers from our industrial partner. We discuss the implementation of our approach in a prototype tool, and present an initial qualitative evaluation of the effectiveness and usability of the proposed views for software development and software certification. The prototype was already found to be very useful and a number of directions for further improvement were suggested. We conclude by discussing these improvements and lessons learned.

***Index Terms*—information flow analysis, software visualization, model reconstruction, component-based software systems**

I. Introduction

How well software engineers understand a system's source code affects how well the system will be maintained and evolved. Various studies have shown that program comprehension accounts for a significant part of the development and maintenance efforts (see [1] for an overview) and with today's rapid growth in system size and complexity, software engineers are faced with tremendous comprehension challenges.

Component-based software engineering is aimed at better managing the complexity of large-scale software development by *assembling* systems from ready-made parts, similar to how hardware systems are assembled from integrated circuits. Software systems are composed from reusable components, implemented in one or more programming languages, and connected using configuration artifacts, ranging from simple key-value maps to domain-specific configuration languages.

Even though component-based design supports comprehension by lowering coupling and increasing the cohesion of components, the *overall* comprehension of component-based systems can be prohibitively complicated. This is caused by the fact that the configuration and composition of the components plays an essential part in the overall behavior of such systems. Consequently, to understand a system's behavior, one needs to understand how control and data flow are interlaced through its combination of component and configuration artifacts.

In spite of these challenges, we found that there is little support for system-wide analysis of component-based systems. Most of the tools that are available 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. In practice, this means that software engineers have only their own cognition abilities to rely on for understanding the overall system's behavior.

Another complicating factor in the engineering of large industrial software systems is that it is not just the developers that need to understand what's going on in the code. However, most of the literature on reverse engineering and program comprehension assumes the developer as the default, and only, audience. There is extensive literature on the visualization of *non-source* artifacts to support domain experts (e.g. [2]), but there is considerably less information on the visualization of source code related information for non-developers. After all, why would non-developers need to understand source code?

This paper is motivated by a typical industrial case in which (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 reverse engineered views on the system need to be goal-driven, at a suitable level of abstraction, and based on relevant knowledge of the application domain.

Our earlier work [4] presents a technique to reverse engineer a fine-grained system-wide dependence model from the source and configuration artifacts of a component-based system. The paper concluded with the observations that the technique was promising but *"considering the size and complexity of most industrial systems, there are many opportunities in the direction of visualizing the analysis results"*, and *"a visualization of the information flow at higher levels of abstraction may considerably improve the comprehensibility"*.

The current paper builds on the technology developed in [4] and makes the following contributions: (1) we propose a

978-1-4673-1216-5/12/$31.00 © 2012 IEEE

hierarchy of views that represent system-wide information flows at various levels of abstraction, aimed at supporting both safety domain experts and developers; (2) we present the transformations that help us to achieve these views from the system-wide dependence models and discuss the different trade-offs between scope and granularity; (3) we discuss how we have implemented our approach and views in a prototype tool, named FlowTracker; (4) we report on an initial qualitative evaluation of the effectiveness and usability of the proposed views for software development and software certification. The evaluation results indicated that the prototype was already very useful. In addition, a number of directions for further improvement were suggested.

The remainder of the paper is organized as follows: Section II describes the context of our work. We present the overall approach and the proposed hierarchy of visualizations in Section III, followed by a description of our prototype implementation in Section IV. We discuss the qualitative evaluation of our approach in Section V. We summarize related research in Section VI, and conclude in Section VII.

II. Motivation

The research 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 unacceptable hazardous situations. Examples include emergency shutdown, process shutdown, and fire and gas detection systems for vessels and off-shore platforms. In particular, we study a family of complex safety-critical embedded software systems that connect software control components to physical sensors and mechanical actuators. The overall goal of the collaboration is to provide our partner with tooling that provides *source-based evidence to support software certification*, and assists the development teams with understanding the behavior of *deployed systems*, i.e. systems composed and configured to monitor the safety requirements of a particular installation (execution environment).

The remainder of this section gives a generalized view on how systems are developed in this application domain. We use the following terminology: a *component* is a unit of composition with well-defined interfaces and explicit context dependencies [5]; a *system* is a network of interacting components; and a *port* is an atomic part of an interface, a single point of interaction between a component and other components or the environment. A component *instance* is the representation of a component as it would appear at run-time, specialized and interconnected following the configuration data. A component *implementation* refers to the component's source code artifacts (i.e without configuration information). There is one component implementation and possibly several component instances for each component in the system.

Without loss of generality, we discuss our approach in terms of the system that was studied. This means that we use the general term *system-level input* and the more case specific term

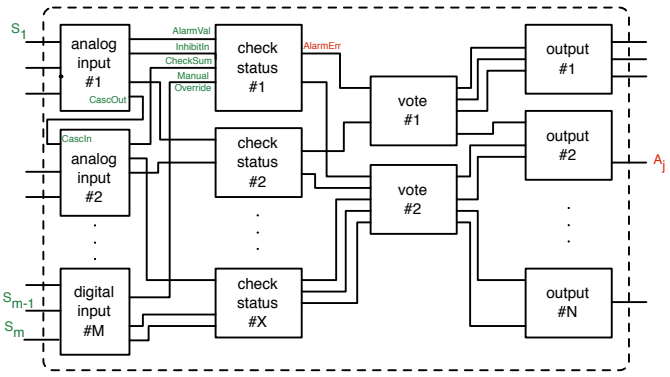


Fig. 1. Component composition network for an example system.

sensor interchangeably, and similar for *system-level output* and *activator*. We emphasize that the proposed approach can also be applied to component based systems with other types of inputs and outputs than sensors and activators.

Concrete software products are assembled in a component-based fashion from a limited collection of approximately 30 reusable components. The components are implemented in MISRA C (a safe subset of C [6]). They are relatively small in size (in the order of 1-2 KLOC) and the computations are relatively straightforward. 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 a network of interconnected component instances (Figure 1). These processing pipelines receive their input values from sensors and process it in various ways, such as measuring, digitizing, voting, and counting before sending the outputs to drivers for the actuators. Components of the same type can be cascaded to handle a larger number of input signals than foreseen in their implementation (shown in Figure 1 for analog inputs #1 and #2). Similarly, the output of a given pipeline can be used as input for another pipeline to reuse the safety conclusions for one area as inputs for a connected area.

Research Question: As installations that are monitored become bigger, the number of sensors and actuators grows rapidly, the safety logic becomes increasingly complex and the induced component networks end up interconnecting thousands of component instances. To give an impression, consider that in contrast to those 11 instances and 4 stages shown in Figure 1, 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 harder and harder to understand and reason about the overall behavior of the system. The main question that drives our research is *"can we provide source-based evidence that signals from the system's sensors trigger the appropriate actuators?"*.

III. Approach

The question if signals from the system's sensors affect the appropriate actuators can be answered by tracking the information flow between sensors and actuators using program

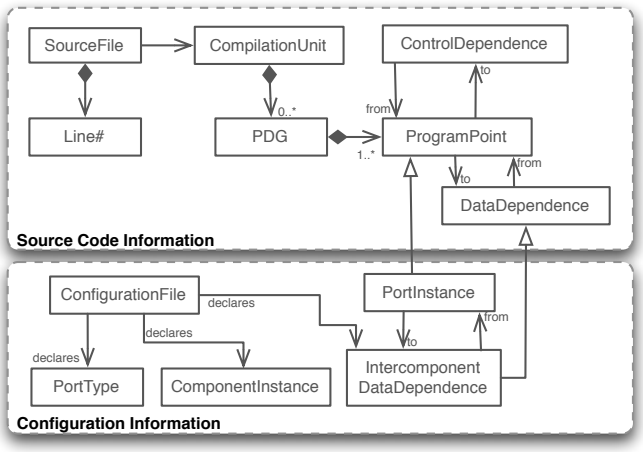


Fig. 2. Main elements from artifacts used to track information flow.

slicing [7]. Program slicing is a decomposition technique that can be used to leave out all parts of the program that are not relevant to a given point of interest, referred to as the *slicing criterion*. In other words, a (backward) slice consists of all the program elements that potentially affect the values at the slicing criterion [8]. Thus, by selecting an actuator as slicing criterion, we can determine which sensors can affect this actuator since these will be contained in its backward slice.

Note that dynamic analysis (tracing) during real-life operation is not an option due to safety hazards. In addition, off-site execution requires advanced stubs and simulators to replace hardware components and to create realistic execution scenarios. Since this infrastructure is already under high demand for testing, we investigate alternatives based on static analysis.

There are two challenges that need to be addressed to successfully apply slicing in our context: (1) program slicing is typically defined within the closed boundaries of source code, whereas our case needs system-wide slicing across a network of interacting components, i.e. including information from component source code and system configuration artifacts; (2) the information that is obtained via slicing typically contains many low level details that can impede comprehensibility.

The first challenge is addressed by reverse engineering a fine-grained system-wide model of the control- and data dependencies in the system based on our previous work [4], which is briefly summarized in Section III-A. To address the second challenge, we propose a hierarchy of five abstractions (views). We discuss how these views are constructed from the system-wide dependence model via a combination of slicing, transformation (abstraction) and visualization in Section III-B.

A. Reverse Engineering a System-wide Dependence Model

This section summarizes the technique and terminology of our earlier work on reconstructing system-wide dependence models [4]. The overall approach is as follows:

1) For each component in the system, we build a *component dependence graph (CDG)* by following the method for constructing inter-procedural dependence graphs [9] and taking the component source code as "system source".
2) The system's configuration artifacts are analyzed to build an *inter-component dependence graph (ICDG)*. This graph captures the *externally visible* interfaces and interconnections of the component instances. Construction of the ICDG is done in the same way as the component composition framework uses to set up the correct network.
3) The *system-wide dependence graph (SDG)* is constructed by integrating the system's ICDG with the CDGs for the individual components. Conceptually, the construction can be seen as taking the ICDG and substituting each "component instance node" with a sub-graph formed by the CDG for the given component.

Figure 2 gives an overview of the main information that we collect from various source artifacts to build these system-wide dependence models.

B. Model Abstraction and Visualization

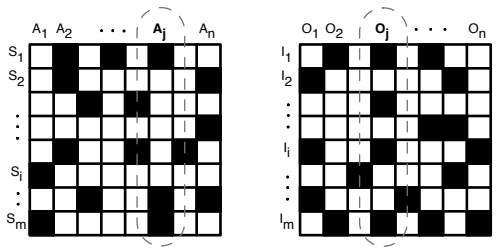
Dependence graphs, and slices through dependence graphs, are complex, often even more complex than the original source artifacts. This is because these models reflect all relevant program points and dependencies from a compiler's perspective, an intrinsic characteristic that makes them well-suited for detailed program analysis, but it makes them less suited for directly supporting comprehension or visualization [4, 10].

In order to make the detailed information contained in an SDG or slice more suitable for comprehension, we propose a hierarchy of five abstractions (views) that are aimed at satisfying 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. These views are constructed from the system-wide dependence model via a combination of slicing, transformation and visualization. The various levels are interconnected via hyperlinks to enable easy navigation and support various comprehension strategies [11]. We distinguish the following views:

(1) System Dependence Survey: This view shows the dependencies between all system-level inputs (sensors) and outputs (actuators) in one single matrix, with sensors and actuators as rows and columns respectively (see Figure 3a). 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 of the SDG 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 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 shows the inter-component information flow from all sensors that can affect a given actuator, i.e., there is a diagram for each actuator in the system. The view hides all intra-component level information in a backward slice through the SDG with actuator A_j as



(a) System-wide dependencies (b) Component dependencies

Fig. 3. System- and Component Dependence Surveys.

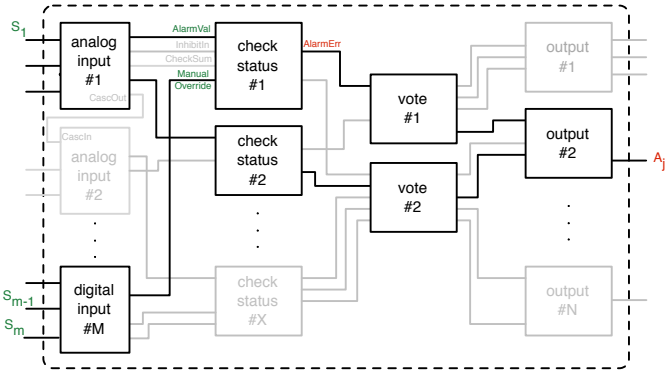


Fig. 4. System Information Flow highlighted for A_j in example system.

slicing criterion. The result highlights the actuator and all related sensors, component instances, and inter-component connections. Figure 4 shows an example 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 hyperlinks for navigation so that a user can click on a component instance to zoom in on a single component, or click outside the diagram to return to a higher level of abstraction.

(3) Component Dependence Survey: Similar to the System Dependence Survey, the Component Dependence Survey summarizes the dependencies between a component's input- and output ports using filled cells in a matrix (see Figure 3b). This black-box view shows which input ports can affect which output ports but hides all details on *how* the information flow is realized. 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 in a column (e.g. O_j) 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. Note that we combine sequences of conditions into aggregated conditions wherever possible to reduce cognitive overhead. The details of this refinement are described later, in Section IV-B.

Figure 5 shows an example with output port "AlarmErr" as slicing criterion (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 conditional nodes have hyperlinks embedded to navigate to the corresponding location in the source code (indicated by marker A in Figure 5).

(5) Implementation View: At the lowest level in our hierarchy, the implementation view shows pretty-printed source code with hypertext navigation facilities, e.g. cross-referencing of program entities with their definition. Higher level views provide links into the source code as a means of traceability and to help minimize user disorientation.

C. Typical Usage Scenario

A typical scenario to use this hierarchy is sketched below:

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 3a).
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 (Figure 4),
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 3b)
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 5).
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.

IV. Prototype Implementation

This section discusses the implementation of the approach described in Section III in a tool named FlowTracker. We distinguish three stages in the implementation, detailed below:

A. Model Reverse Engineering

We reuse our earlier tool to reverse engineer system-wide dependence graphs (SDGs) from source artifacts [4]. It builds on Grammatech's CodeSurfer [12][1] to create component dependence graphs (CDGs) for the individual components. Next, these CDGs are traversed using CodeSurfer's API to inject them into OMG's Knowledge Discovery Metamodel (KDM) [13]. The traversal uses the Java Native Interface to drive KDM constructors in the Eclipse Modeling Framework.

[1] http://www.grammatech.com/

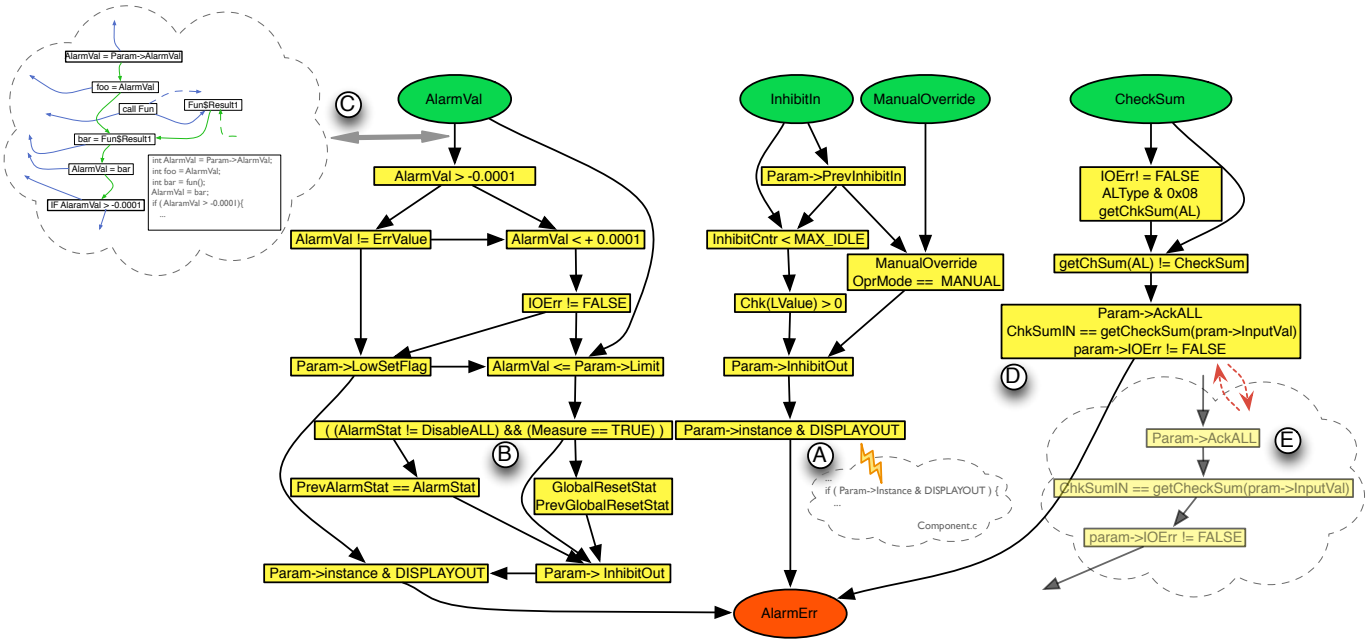


Fig. 5. Component Information Flow example (markers A-E and the cloud-like fragments are used for explanations in the text)

For each program point, we include a pointer to its *origin* in the source code for traceability. Next, we use Xalan-J to analyze and transform the system configuration artifacts into the inter-component dependence graph (ICDG). Finally, we use a straightforward substitution transformation to integrate the CDGs with the ICDG and create the final SDG.

B. View Construction

During view construction, we enrich the SDG with additional *summary edges* and *aggregate nodes* that capture a number of view-specific abstractions that will be used in the presentation stage. Alternatively, we could have defined several "destructive" transformations that create a new model for each view, but we prefer to enrich our SDG model in order to reuse information between views. Our implementation builds on a simple slicing tool in Java that we have created as part of our earlier work. Below we discuss the abstractions that were added for the various views. Their names map trivially on the view names in Section III-B

The SysDep relation is based on slices for each of the system's actuators and includes a summary edge (S_i,A_j) if sensor S_i is in the slice for A_j. Similarly, for each component C, the $\mathsf{CompDep}_C$ relation is based on slicing all output ports and including (I_m,O_n) if input port I_m is in the slice for O_n.

For each actuator A, the relation $\mathsf{SysInfoFlow}_A$ is based on slicing the enriched SDG on A. For each component C_i in the slice, we use the summary edges of $\mathsf{CompDep}_{C_i}$ to hide the internals of C_i. What remains of the slice are summary edges for the connections between (ports of) the component instances involved and connections from the incoming sensors and towards the given actuator. Note that it is not possible to compute this information by simply slicing the ICDG because the ICDG does not contain information about the dependencies between a component's input and output ports.

For each output port O of every component C, the $\mathsf{CompInfoFlow}_{C,O}$ relation is based on three transformations:

(1) Codesurfer splits sub-expressions of a condition over separate program points to increase precision during slicing. When presenting results to the user, this creates a cognitive distance with respect to the original code. We address this issue by merging the sub-expressions of conditions into aggregate nodes that resemble the original code (Figure 5, marker B).

(2) We add summary edges that subsume all nodes that are not input ports, conditions or output ports (Figure 5, marker C). For example, when we have edges (x,y) and (y,z) and y is not an input port, condition or output port, we add a summary edge (x,z). These summary edges are computed transitively, so that they represent the longest path possible.

(3) finally, we analyze the resulting graph to detect so-called *condition chains*. We define condition chains as the (longest possible) paths in the SDG that exclusively consist of single entry/single exit conditional nodes. For each condition chain, we add a special aggregate node to represent the individual conditions in the chain at a higher level of abstraction. This aggregate node is labelled based on the conditions it represents. For an example, see Figure 5, marker D for the aggregate node, and marker E for the condition cluster it represents.

The Implementation View does not require additional summary edges or aggregate nodes to be added to the SDG.

C. Presentation

We present the results of our System- and Component Dependence Surveys as matrices that have been implemented as HTML tables with input and output ports as rows and columns, respectively. This presentation is intentionally chosen

to resemble 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 matrices are made *active* by embedding hyperlinks to the corresponding views on the next lower abstraction level. By clicking one of the cells below a given port or actuator, the user can zoom in on the information flow *leading to* that port or actuator.

We use the KDM API to traverse the view-specific summary edges in our enriched SDG and transform the elements of interest into GDL, a graph description language that can be processed by the aiSee graph layout software.[2] We make use of GDL's provisions for collapsable subgraphs to represent conditional clusters and their aggregate representation so the user can go back and forth between these representations. We include navigation between views by embedding hyperlinks in the nodes representing components and ports. Similarly, we provide traceability by embedding hyperlinks to source code locations in Component Information Flow nodes representing conditions. These hyperlinks are preserved when aiSee computes the layout and renders the graph in Scalable Vector Graphics (SVG) format.

Finally, we create a pretty printed version of the source code using Doxygen.[3] Doxygen is a source code document generator for numerous programming languages, including C. It can be configured to include the source code as part of the generated documents in HTML format and embed various hypertext navigation features.

V. Evaluation

To evaluate our approach we consider the following three aspects: accuracy, scalability and usability. In a context of software certification, the accuracy of our views is of utmost importance and is determined by the accuracy of our model reconstruction and of our slicing tool. Both were evaluated in detail in [4] and showed 100% accuracy when compared to gold standard results from CodeSurfer. The same paper also reported that these steps show linear growth of execution time and model size with respect to program size. This indicates good overall scalability as the views that we construct in this paper are all projections of this model (i.e. smaller in size).

In the remainder of this section we focus on the results of a preliminary qualitative study assessing the *usability* of FlowTracker, and in particular of the proposed views.

A. Study Design

Considering that FlowTracker is still a prototype in early stages of development, our goal is to conduct an *exploratory study* to evaluate the usability and effectiveness of the visualizations, and their fitness for the needs of our industrial partner. To this end, we conduct a *qualitative evaluation* of the tool with a group of six subjects that were selected so that their profiles would match with the various roles of prospective FlowTracker users. We use such a pre-experimental design as it is a cost-effective way to find out the major positive and negative points, and identify missing functionality and required improvements before the tool can be adopted by our industry partner [14]. In addition, this design limits the overhead and impact of our study on the industrial partner, and it decreases the influence of (negative) *anchoring effects* that can rise from having early prototypes evaluated by people that should later adopt the tool [15] (this effect can be paraphrased as "first impressions are hard to change"). This is an important consideration for a domain-specific tool that is dedicated to a specific audience, like ours.

Participant profiles: Three of the participants are senior engineers in Kongsberg Maritime (KM) who work daily with the case study system. Participant P1 is a senior developer who develops and maintains core modules of the studied system, his focus is more on individual modules than complete systems. P2 is both a system integrator and auditor: (a) in some projects, his role is to *audit* systems that are built by other teams to assess their validity and reliability; (b) in other projects, his role is to compose the overall system logic from components, which includes verifying correct component inter-connections. P3 is a safety expert who handles the certification process together with the third party certifiers such as DNV.[4] In addition, she has prior development experience on the system.

We recruited the other three subjects (P4 to P6) from colleagues who were in the final stages of their PhD studies on model-based software verification and validation at Simula Research Laboratory. These subjects are very familiar with the notions of component-based design, model-driven engineering, verification and validation, but they have no previous exposure to the case study system. However, each of them had two to four years of industrial experience prior to starting their PhD study, so we refer to them as junior developers. We include this second group of subjects with a different perspective to decrease the potential bias towards the specific traits of the case study that could be caused by only selecting subjects from our industrial partner [16].

Preparation: The evaluation sessions were conducted independently of each other, and the results were aggregated after all participants finished the evaluation. Each session started with a brief presentation of FlowTracker (~10min.). The presentation included a walk-through of a typical usage scenario, similar to Section III-C. The junior developers were given an extra presentation on the studied system, to clarify the problem statement and the goals of the study. Next, we let the participants play around with the tool until they felt confident in their understanding of its functionality. We concluded this training session with three hands-on exercises which participants had to complete before starting the evaluation. The exercises were designed in a way to engage all the views and the major features of FlowTracker. There were no time limits to complete the exercises and discussion was stimulated.

Data Collection: The evaluation itself consists of a *structured interview* which was guided by a questionnaire consisting of 30 closed questions that used a 5-point Likert scale and 6

[2] http://www.aisee.com/ [3] http://www.doxygen.org/

[4] http://www.dnv.no/

open (discussion) questions. Questions where both positively and negatively phrased to break answering rhythms and avoid steering the subjects [17]. In total, each session lasted between 60 and 90 minutes. 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. Based on the answers, the interviewer included relevant follow-up or clarification questions. We recorded the complete audio of the sessions (training+interviews) and transcribed and analyzed them using the ELAN multimodal annotation tool [18]. This allowed us to collect the answers to our questions, find deeper reasons behind those answers and get more insights into the preferred interactions with FlowTracker.

Workshop: Prior to the evaluation sessions, we organized a workshop meeting at KM to present FlowTracker to various stakeholders with different roles and engineering backgrounds. As the audience of this workshop was different from the evaluation participants, we will also discuss the relevant feedback from this meeting below.

B. Findings

In the remainder of this section we present the major findings, key questions and, the highlights of the feedback we received from the participants. The results are aggregated per view, followed by a discussion of feedback on the overall usage experience. Whenever there are outliers or noteworthy differences between the answers of the group of junior developers versus the group of senior, we will discuss the details.

(1) System Dependence Survey: The responses to questions regarding this view indicated that the engineers very frequently need to find out which system inputs affect a certain output. For example, P2 stated he *"needs that kind of information on a daily basis"*. When asked how they would obtain such information in the absence of FlowTracker, most subjects responded that they would (and currently did) revert to the manual inspection of the source code to find these dependencies, except for P4 who preferred *"to use UML activity diagrams to model the message passing in the system"*.

Overall, the subjects indicated that they found the presentation of information in this view to be intuitive, and that the goal of summarizing system-wide information flow was adequately achieved. They agreed with our choice to designate this view as the starting point for navigation in FlowTracker.

The positive response to this view is not surprising considering that it closely resembles the Cause & Effect specifications that are already used by our partner. Already from the very first meetings, there was a request for tooling that would enable safety domain experts (and certification bodies) to compare the *"as-implemented"* system against the *"as-specified"* safety logic at a single glance, and this view satisfies that goal.

(2) System Information Flow: The subjects were generally satisfied with the functionality of this view: indicating which components, ports and sensors can affect the value of the selected actuator. FlowTracker currently shows all components and ports and *highlights* the elements that affect a given actuator; the others are dimmed. An alternative could be to hide these elements from the diagram. Most subjects favored the current design. P5, for example, remarked that *"this view gives me the big picture as well as the micro answer"*. However, two subjects had some reservations with respect to the amount of information shown in this view; P4 and P6 were concerned that the extra information could lead to confusion. All subjects were positive about the idea of adding more interactive facilities, such as an option to include or hide the dimmed elements on demand in this view.

The view was regarded an appropriate navigation intermediary between the System Dependence Survey and Component Dependence Survey, except for P5 who preferred to have the choice to jump directly from System Dependence Survey to Component Dependence Survey as alternative navigation path. We had considered this option while designing the navigation structure but decided against it in favor of a single predictable navigation structure without shortcuts to avoid disorientation.

The way information is presented was received as intuitive, and *"very beneficial for the needs of system integrators"*. This benefit was also mentioned during the initial workshop where a participant remarked that this view was useful to inspect *"what is happening when there is no system-wide information flow between a sensor-actuator pair that is supposed to be connected"*. Examples that were mentioned included analyzing configuration issues like *dangling connections* that could, for example, result from renaming component port names but not updating existing (external) system configurations.

Subjects also observed that the System Information Flow to some extent duplicates the functionality of one of our partner's current tools, which shows the overall component composition network based on the configuration information. However, the FlowTracker view is based on fundamentally different underlying knowledge: it is based on the system-wide dependencies *across* components instead of just using the configuration information. As such, the System Information Flow gives a more reliable view regarding the *actual* inter-component information flow, because any disruptions that occur *inside* components will be rendered as a broken flow in our view but are not noticed by the existing tool.

During the discussion, P2 (system integrator) pointed out a promising new feature: he mentioned that KM has (preliminary) guidelines for inter-connecting components, for example detailing which port-types are compatible. Although these guidelines do not guarantee correct behavior, having some form of automated checking could save a lot of time by signaling apparent connection mistakes. P2 saw good opportunities for FlowTracker to check such composition guidelines, and to show deviations in the System Information Flow view.

(3) Component Dependence Survey: Similar to the System Dependence Survey, the subjects agreed that this view adequately summarizes the dependencies between input and output terminals. P3 (safety expert dealing with system certification) regarded this view as *"top priority for the certification*

process and a facilitator of the discussions with the third-party certifiers". Module developer P1 stated that he *"must know the input/output relations of the components at all times, but I currently only have the source code to read and hopefully find out about all dependencies"*. P1 did not expect that this view would be beneficial for the certification process, but he emphasized that he had not been directly involved in the certification process. P6 preferred that the matrix would distinguish between the data dependencies and control dependencies between inputs and outputs; input terminals whose *value* is transferred to the output terminals appear differently from the inputs whose value is used to *control* the information flow toward the same output port.

(4) Component Information Flow: We received mixed feedback regarding this view. The most positive responses came from the group of industrial subjects, in particular P1, the module developer. The variety of opinions about this view can perhaps be explained by the fact that it uses an unfamiliar design, which does not resemble the more well-known matrix or UML diagram styles like our other views. Another potential cause is the visual complexity of some of the larger diagrams, which was mentioned by at least one of the subjects.

Five of the subjects agreed that conditions can have a significant effect on the intra-component information flows and should be highlighted and put in perspective to improve comprehension. The subjects also indicated that *"such graphs clearly show the intra-module information flows [and] the effects of conditions on the information flow"*, reportedly *"much better than the source code"*. On the other hand, subject P6 answered that *"one might need to see the assignment statements in the diagram as well to understand the information flows"*. In addition, she would like to see the outgoing edges of condition nodes labelled with True or False to indicate which edge would be used if the condition would be evaluated during actual execution. Finally, she had concerns about the intuitiveness of the diagrams when they grow in size, i.e. she mentioned that *"the larger diagrams are no longer intuitive"*. Subject P5 remarked that this view would *"probably not contain enough information to check safety regulations or design guidelines"*.

Prior to our evaluation, we assumed that the Component Dependence Survey (i.e. one level above this view) would be the lowest abstraction level that would be useful for non-developers such as safety experts. However, safety expert P3 actually regarded this Component Information Flow as *"a very good tool to demonstrate to the external certifiers what we have done"*, i.e. to provide evidence for software certification. During the workshop, participants discussed that this view would make a good point of reference for discussions between different engineering roles, they stated that *"it acts as a bridge between the C programmers and integrators"*.

The subjects would like to see more interactive facilities, especially some measures to better deal with the larger diagrams. In addition to zooming, a concrete suggestion that was made is the option to just see the information flow that starts in a single selected component input port. We foresee that many of these requests can be achieved quite easily by incorporating a better graph viewer than currently used in the prototype.

(5) Implementation View: This view is very similar to the source code in a typical modern IDE (besides not being editable in our prototype). As such, the view by itself doesn't contribute much, but the subjects reported that the inclusion of this view in FlowTracker helped them to relate more easily to higher level views since it *"helps to remove the gap between visualizations and the source code"*.

In particular, subjects considered the hyperlinks from conditions in the Component Information Flow diagram to the respective locations in the source code beneficial for comprehension and traceability. They were less sure that these links would support certification purposes equally well: P4 and P6 said they are useful if only the certifier knows the source code (which they thought unlikely); P1 considered the links beneficial; P2 and P5 refrained from answering this question since they felt not sure about the certification process; Safety expert P3 said that *"certifiers generally do not look at the source code, but in the worst cases where they want to see more evidence, these links will help to find the right locations"*.

Overall Experience: All in all, the subjects were positive about the intuitiveness of the tool as they *"did not need to learn a lot of things before being able to work with FlowTracker"* and *"did not feel that the tool was complex"*.

The subjects would like to see the tool closer integrated into their IDEs, although the junior developers remarked that they did not see immediate benefits from using the tool during the early stages of developing the components. They preferred to *"use FlowTracker during the more matured stages of development, such as integration, testing, or for refreshing [their] understanding of an existing system"*. The industrial subjects, on the other hand, were *"looking forward to use FlowTracker during the development process, and for post-development phases, such as auditing and certification"*.

Overall, FlowTracker received excellent feedback regarding component and system comprehension. When we look at the feedback concerning FlowTracker's support for the certification process, the results were less conspicuous, but still very positive, most notably from the industrial subjects.[5] They argued that FlowTracker supports the certification process by *"enabling discussions between the developers and safety experts"*, and *"demonstrating the safety logic that is actually implemented in a system to the external certifiers"*.

When subjects were asked to think of other tasks where FlowTracker could be helpful, topics included: 1) source code maintenance; 2) track ripple effects of modified source code; 3) track ripple effects of modified configuration files; 4) configuring a new system; 5) debug individual modules; 6) auditing projects; and 7) training new project members.

C. Threats to Validity

We conducted an evaluation study using a group of six subjects. It could be argued that this amount of subjects is

[5] We should add that two junior developers did not comment on this aspect as they felt that they did not know the certification process well enough.

too small to infer generalizable conclusions. We have taken the following measures to reduce this threat: Considering that FlowTracker is a domain-specific tool with a specific industrial target audience, the potential for recruiting a statistically significant number of subjects is rather limited, so we use an exploratory *qualitative* study design to get the best possible results from a limited group of subjects at an early stage. In addition, the subjects were selected such that their profiles would match with the various roles of prospective FlowTracker users and in addition to the industrial subjects, we added a second group of subjects with a different perspective to avoid bias towards the specific traits of the case study.

Since we have conducted researcher-administered interviews, subjects might have been inclined to give socially acceptable positive feedback. We have limited the impact of this threat by including control- and follow-up questions and instructing the subjects that honest answers would in the end give them the most valuable tool. This threat would have been lower for self-administered questionnaires but from other experiences we learned that the amount and the quality of feedback for such studies is much lower.

Another threat is that the reliability of the collected data depends on the interviewer's interpretation of the subject's answers or actions. We have mitigated this threat in two ways: (1) we emphasized that the participants should try to give (or include) closed answers in terms of the Likert categories whenever possible, to limit subjective interpretation on the evaluators side; (2) each of the two authors independently transcribed and analyzed the interviews. Afterwards, the results were compared and differences were re-analyzed (jointly) until an agreement was reached. The latter step was obviously most valuable for the cases where subjects did not (only) give a closed answer but included more discussion.

A potential concern w.r.t. generalizability is that our evaluation only included one subject for each of the different roles of module developer, system integrator, and safety expert. As such, this subject gets a dominant voice in the evaluation and the answers may be based more on personal opinions than on what is needed for the role. We have tried to limit the impact of this threat by organizing a pre-evaluation workshop where we asked the stakeholders to identify the most qualified senior engineers that could represent these roles in the evaluation. In addition, it turned out that subjects with a given role generally also had experience in some of the other roles, which also helps to create a more balanced picture.

VI. Related Work

Maletic et al. [19] identify five dimensions of software visualizations: tasks (why), audience (who), source (what), representation (how), and medium (where). Our work can be summarized as *why:* providing source-based evidence to support software certification, *who:* for safety domain experts and developers, *what:* of implementation artifacts of component-based systems, *how:* by visualizing information flow using a set of hierarchical views, *where:* on a computer screen.

Hermans et al. use leveled data flow diagrams to aid professional spreadsheet users in comprehending large spreadsheets [20] . Their survey showed that the biggest challenges occur when spreadsheets are transferred to colleagues or have to be checked by external auditors. They suggest a hierarchical visualization of the spreadsheets: starting from coarse-grained worksheets, expanding worksheets to view the contained data blocks, and diving into formula view to see "a specific formula and the cells it depends on". Our work is similar in providing a hierarchical visualization of information flow, with each view having a different trade-off between scope and granularity. Another similarity is the inclusion of non-developer, domain experts as users of the visualizations. However, the analysis subject, technique and the underlying entities to be visualized are completely different. Our work analyzes source code to infer system-wide information flows using SDGs that are based on both control- and data flow information, while they analyze data flow dependencies in formula-rich spreadsheets.

Krinke reports on various attempts to visualize program dependence graphs and slices via existing (algorithmic) graph layout tools [10]. He proposes a declarative graph layout, tailored to preserve the relative *locality* of program points to provide a better cognitive mapping back to the source code. A survey showed that the standard representations of program slices were "less useful than expected", and the improved layout is "very comprehensible up to *medium* sized procedures", but "overly complex and non-intuitive" for large procedures. He concludes that a textual visualization of source code is essential and introduces the distance-limited slice to assign each program statement a specific color according to its distance to the slicing criterion. In contrast, we developed multiple layers of abstraction to reduce the complexity of system-wide slices and show only the information that is relevant for the particular task and users. We provide links between the various views which can be navigated down to a textual representation of source as a last resort.

Pinzger et al. [21] use nested graphs to represent static dependencies in source code at various levels of abstraction. They follow a top-down approach similar to ours for representing information about the system, and allow users to adjust the graphs by adding or filtering information, like adding a caller or "keep callees and remove other nodes". In contrast to our approach which creates abstracts from fine-grained data- and control dependencies, they analyze static "uses" dependencies in Java programs at a relatively coarse grained level, considering elements such as package, class, method, method call and field access.

We refer to our previous work [4] for a detailed discussion of work related to our method to build system-wide dependence models from heterogeneous source artifacts.

VII. Concluding Remarks

Component-based software engineering is widely used to manage the complexity of large scale software development. Although correctly engineering the composition and configuration of components is crucial for the overall behavior, there

is surprisingly little support for incorporating this information in the analysis of such systems. Moreover, to get a correct understanding of system's overall behavior, one needs to understand how the control and data flow is *interlaced* through component sources and configuration artifacts. We found that support for such a system-wide analyses is lacking, as it is hindered by the heterogeneous nature of these artifacts.

Contributions: In this paper, we address these issues by proposing an approach that supports system-wide tracking and visualization of information flow in heterogeneous component-based software systems. Our contributions are the following: (1) we proposed a hierarchy of views that represent system-wide information flows at various levels of abstraction, aimed at supporting both safety domain experts and developers; (2) we presented the transformations that help us to achieve these views from the system-wide dependence models and discuss the different trade-offs between scope and granularity; (3) we discussed how we have implemented our approach in a prototype tool; (4) we reported on an initial qualitative evaluation of the effectiveness and usability of the proposed views for software development and software certification. The evaluation results indicated that the prototype was already very useful. In addition, a number of directions for further improvement were suggested.

Future Work: We see several directions for future work: First of all, we want to improve the overall user experience by adding more on-demand interaction facilities such as zooming and hiding or collapsing groups of nodes. Such facilities allow users to be more selective in the amount and type of information they see, according to their information needs at the moment. As briefly mentioned before, we foresee that this can be achieved by using a more elaborate graph viewer than currently used in the prototype. Since the graph presentation is done using SVG, a promising direction forward is investigating the inclusion of some additional scripting based on JavaScript libraries such as Raphäel[6] or D3[7].

Moreover, to improve the scalability of Component Information Flow diagrams, we want to investigate if the hierarchical block structure of the source code can be used to create a hierarchy of collapsable sub-graphs in the visualizations.

Then there were a number of interesting extensions to FlowTracker that were brought up during the evaluation. One example is the possibility to include some kind of automated type checking for component inter-connections or other forms of constraint checking on component composition. Another extension that came up is the ability to analyze and visualize multiple versions of a system at the same time and highlighting the modifications and their impact in the version history.

A final direction for future work is the integration of our tooling with an IDE like the Eclipse platform. Besides the increased ease of adoption, this would also have the added benefit of being able to directly navigate to editable source code and reuse of all existing Eclipse features such as intelligent search, bookmarking etc. Moreover, we will be able to take advantage of Eclipse perspectives and create separate perspectives for safety domain experts and developers to optimize the experience and avoid intimidation or distraction by unneeded detail.

[6] http://raphaeljs.com/ [7] http://mbostock.github.com/d3/

ACKNOWLEDGMENTS

We would like to thank the participants in our workshop and interviews for their valuable time and feedback, without their collaboration the evaluation of this work would not have been possible.

REFERENCES

[1] A. Abran, J. Moore, P. Bourque, R. Dupuis, and L. Tripp, *Guide to the Software Engineering Body of Knowledge - 2004 Version - SWEBOK*. IEEE-Computer Society Press, 2005.

[2] J. Steele and N. Iliinsky, *Beautiful Visualization, Looking at Data through the Eyes of Experts*, 1st ed., J. Steele and N. Iliinsky, Eds. Sebastopol, CA, USA: O'Reilly Media, 2010.

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

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

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

[6] 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, Jun. 2004.

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

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

[9] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

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

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

[12] P. Anderson, "90% Perspiration: Engineering Static Analysis Techniques for Industrial Applications," in *IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*, Sep. 2008, pp. 3–12.

[13] OMG, "Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM) - v1.2," 2010.

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

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

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

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

[18] H. Brugman and A. Russel, "Annotating Multi-media / Multi-modal resources with ELAN," in *Fourth Int'l Conf. on Language Resources and Evaluation (LREC), 2004*. http://www.lat-mpi.eu/tools/elan/.

[19] J. Maletic, A. Marcus, and M. Collard, "A task oriented view of software visualization," in *IEEE Int'l Ws. on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2002, pp. 32–40.

[20] F. Hermans, M. Pinzger, and A. V. Deursen, "Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams Categories and Subject Descriptors," in *Int'l Conf. on Software Engineering (ICSE)*, 2011, pp. 451–460.

[21] M. Pinzger, K. Graefenhain, P. Knab, and H. C. Gall, "A Tool for Visual Understanding of Source Code Dependencies," in *IEEE Int'l Conf. on Program Comprehension (ICPC)*, Jun. 2008, pp. 254–259.