Information and Software Technology 77 (2016) 34–55

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Analyzing and visualizing information flow in heterogeneous component-based software systems

Leon Moonen[a,*], Amir Reza Yazdanshenas[b]

[a] Simula Research Laboratory, Oslo, Norway
[b] Testify AS, Oslo, Norway

ARTICLE INFO

Article history:
Received 3 July 2015
Revised 2 May 2016
Accepted 4 May 2016
Available online 12 May 2016

Keywords:
Information flow analysis
Component-based software systems
Model reconstruction
Program comprehension
Software visualization

ABSTRACT

Context: Component-based software engineering is aimed at managing the complexity of large-scale software development by composing systems from reusable parts. To understand or validate the behavior of such a system, one needs to understand the components involved *in combination with* understanding how they are configured and composed. This becomes increasingly difficult when components are implemented in various programming languages, and composition is specified in external artifacts. Moreover, tooling that supports in-depth system-wide analysis of such heterogeneous systems is lacking.

Objective: This paper contributes a method to *analyze and visualize information flow* in a component-based system at various levels of abstraction. These visualizations are designed to support the comprehension needs of both safety domain experts and software developers for, respectively, certification and evolution of safety-critical cyber-physical systems.

Method: We build system-wide dependence graphs and use static program slicing to determine all possible end-to-end information flows through and across a system's components. We define a hierarchy of five abstractions over these information flows that reduce visual distraction and cognitive overload, while satisfying the users' information needs. We improve on our earlier work to provide interconnected views that support both systematic, as well as opportunistic navigation scenarios.

Results: We discuss the design and implementation of our approach and the resulting views in a prototype tool called FlowTracker. We summarize the results of a qualitative evaluation study, carried out via two rounds of interview, on the effectiveness and usability of these views. We discuss a number of improvements, such as more selective information presentations, that resulted from the evaluation.

Conclusion: The evaluation shows that the proposed approach and views are useful for understanding and validating heterogeneous component-based systems, and address information needs that could earlier only be met by manual inspection of the source code. We discuss lessons learned and directions for future work.

1. Introduction

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

Component-based software engineering [2] 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 of 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 *detailed* analysis and comprehension of the *complete* component-based system can be prohibitively complicated. This challenge stems from the fact that the configuration and composi-

* Corresponding author.
E-mail addresses: leon.moonen@computer.org (L. Moonen), amirry@simula.no (A.R. Yazdanshenas).

http://dx.doi.org/10.1016/j.infsof.2016.05.002
0950-5849/

tion of the components play an essential part in the overall behavior of such systems [3]. Consequently, to understand the intricacies of a system's behavior, one needs to understand how control and data flow are interlaced through its combination of component and configuration artifacts. Note that for this analysis it is not enough to treat the components as black boxes: similar to what was earlier found for testing of component based systems, white-box approaches are needed to collect the required information from the components' source code [4–7].

In spite of these challenges, we found that there is little support for system-wide analysis of component-based systems from their source artifacts. This shortage of system-wide analysis in component based systems can be traced back to *language heterogeneity*: software components can be implemented using one or more programming languages, while configuration artifacts are often encoded using smaller declarative languages such as XML. We follow Strein et al. [8] by referring to such multi-lingual component-based systems as *heterogeneous systems*. Most of the available tools have strict limitations on the programming languages they can process, i.e., they are only fit for mono-lingual systems. This typically means that information from external configuration artifacts cannot be included, effectively inhibiting system-wide analysis in heterogeneous systems 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 engineering large industrial software systems is that it is not just the developers who need to understand what's going on in the code: also non-developers, for instance safety domain experts, need to understand what is actually implemented in the code to assess whether the system properly adheres to given safety requirements. However, most of the literature on reverse engineering and program comprehension assumes that the developers are the default, and the only, audience. There is extensive literature on the visualization of *non-source* artifacts to support domain experts [e.g., 9], but considerably less information exists 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 implemented in the system so that they can conduct software certification. These safety domain experts need to see the system's source artifacts in a context relevant to them – not just what the code *does*, but what that *means* for safety concerns [10]. 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 [11] 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 observation 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 work builds on the technology developed in [11] and makes the following contributions: (1) We propose 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 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 a qualitative evaluation study, carried out via two rounds of interviews, on the effectiveness and usability of the proposed views for software development and software certification.

The study reported here was carried out in collaboration with our industry partner in two stages: The first stage, consisting of a feasibility study of the approach as well as an initial industrial evaluation of the prototype (Flowtracker V1), was discussed in our conference paper [12]. This paper extends that work by following up on several ideas for improvement that came out of the initial evaluation. We motivate and implement a number of extensions to the prototype and improve the ways in which the views can be navigated (Flowtracker V2). Moreover, we conduct a second round of evaluations to assess if these changes indeed lead to the anticipated improvements. For the sake of clarity and completeness, this paper elaborates on the work performed in both stages. Since the second stage is a continuation of the first one, we present the results in a single narrative, only distinguishing them with respect to individual stages when the discussion of changes or results mandates such a distinction.

The remainder of the paper is organized as follows: Section 2 describes the context of our work. We present the overall approach and the proposed hierarchy of visualizations in Section 3, followed by a description of our prototype implementation in Section 4. We discuss the qualitative evaluation of our approach in Section 6. We summarize related research in Section 7, and conclude in Section 8.

2. Background and motivation

Case description: 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 in 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 [2]; a *system* is a network of interacting components; and a (component) *port is an atomic part of an (component) 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.

Generality of the approach: We discuss our approach in terms of the concrete case that was studied. This means that we use the general term *system-level input* and the more case-specific term *sensor* interchangeably, and, similarly, for *system-level output* and *activator*. We use the general term *system port* to refer to both system-level inputs and outputs. However, we emphasize that the proposed approach is not specific to the specific system studied.

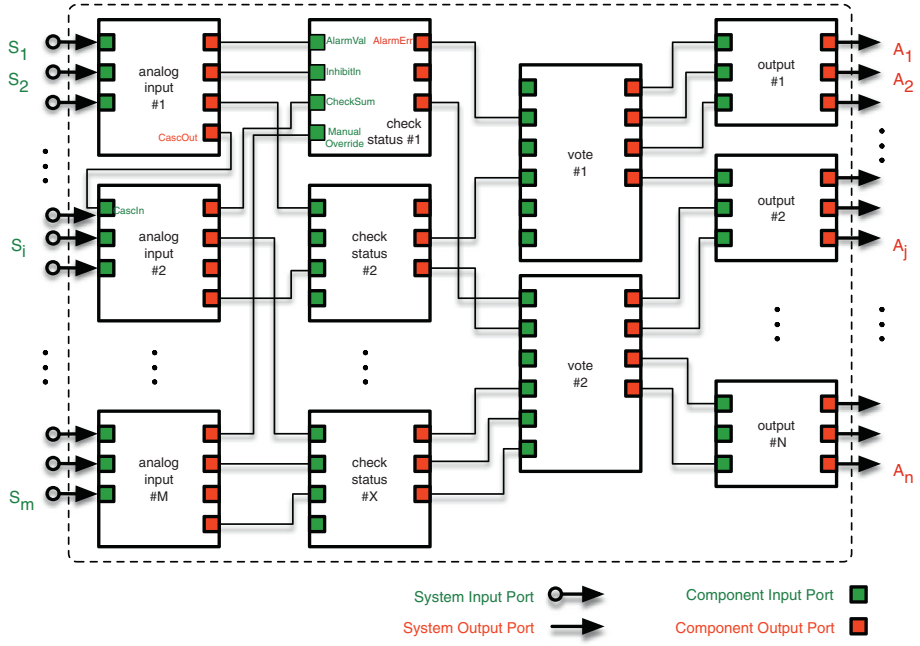


Fig. 1. Component composition network for an example system.

It can be applied to component-based systems with other types of input and output than sensors and activators, and it supports other component models than the proprietary one of our industrial partner. For example, quite diverse options such as the Koala [13] and the Spring [14] component models are supported by our approach by adding a simple parser for their configuration artifacts.

Configuration artifacts: Concrete software products are assembled in a component-based fashion from reusable components. Our industrial partner uses a limited collection of approximately 30 reusable components that are implemented in a safe subset of C [15] and have control logic that is highly configurable via parameters (e.g., initialization, thresholds, comparison values, etc.). These parameters and the component's interface, i.e. the in- and output ports with their respective data types, are externally described in a so-called component definition file, which is one type of *configuration artifacts* that needs to be considered.

The system's overall logic is achieved by composing a network of interconnected component instances (Fig. 1). These processing pipelines receive their input values from sensors and process them 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 Fig. 1 for analog inputs #1 and #2). Similarly, the output of a pipeline can be used as input for another pipeline to reuse the safety outcomes for one area as inputs for a connected area. The composition is specified in a second type of *configuration artifacts* that need to be considered. For a concrete system, such an artifact defines: (1) the various component instances in the system, with values for configurable parameters where needed, and (2) the connections between in- and output ports of these component instances.

Motivation: As monitored installations 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 make this more concrete, consider that in contrast to those 11 instances and 4 stages shown in Fig. 1, a typical real-life installation has 12 to 20 stages in each pipeline, and approximately 5,000 component instances in its safety system. As a result of these numbers, it becomes increasingly difficult to understand and reason about the overall behavior of the system.

Research question: The *main* question that drives our research is: *"Can we provide source-based evidence that the signals from the system's sensors trigger the appropriate actuators?"*

Goals: In addition to this primary goal of supporting software certification questions, our industry partner indicated that they had a secondary goal: During the collaboration, their developers and system integrators recognized that such system-wide program comprehension techniques had the potential to support various development and maintenance tasks. For example, they were looking for support that helped them to predict the consequences of a change in a given component on the complete system (i.e., impact analysis). Similarly, they also wanted to understand better what parts of the system could actually affect the state of a given component. To support both the primary and secondary goals, we set out to provide black-box and white-box visualizations of the system at various levels of abstraction, aimed to satisfy the needs of the various users and tasks foreseen by our collaborator, and allowing for a trade-off between detail and cognitive complexity.

The following list summarizes the goals and characteristics of this work:

1. To track all possible system-wide channels for information flow
   (a) in heterogeneous component-based systems,
   (b) spanning across software components and configuration artifacts
   (c) by means of static analysis, at design/compile time.
2. To present these information channels to end users,
   (a) using a high-level black-box view aimed at supporting certification,
   (b) as well as white-box views detailing the software elements that enable the information channels,
   (c) at levels of detail that support software developers and system integrators with their tasks.
3. To visualize information flows in the system,
   (a) to enhance software comprehension in general, and

(b) to satisfy information needs typical to dependence analysis and impact analysis,
(c) using notation that is intuitive and familiar to the stakeholders.

3. Approach

The question if signals from the system's sensors affect the appropriate actuators can be answered by analyzing the information flow between sensors and actuators using *program slicing* [16]. Program slicing is a decomposition technique that can be used to leave out all parts of the program that are irrelevant 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 [17]. Thus, by selecting an actuator as the slicing criterion, we can determine which sensors can affect this actuator, since these will be contained in its backward slice. Conversely, a *forward slice* consists of all the program points that are potentially affected by the slicing criterion [17]. Thus, by selecting a sensor as the slicing criterion, we can determine which actuators can be affected by this sensor, since they will be contained in its forward slice. In the remainder, analysis *direction* refers to the direction of the slicing, and *forward (backward) information flow* refers to an information flow analyzed via forward (backward) slicing.

Two challenges 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 the components' source code and the system configuration artifacts; (2) The information 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 [11], which is briefly summarized in Section 3.1. To address the second challenge, we propose a hierarchy of five abstractions (views), with close consideration to the existing abstractions already known to our industry partner, and the software elements that have a decisive influence on end-to-end information flows. The motivation is that this will help define views that are intuitive for our industry partner. In Section 3.2 we discuss the defining elements and the characteristics of these views. We also present the methods used to construct each of these views from the system-wide dependence model via a combination of slicing, transformation (abstraction), and visualization. In Section 3.3 we present a walk-through of a typical top-down navigation of the system using the proposed views: what the user sees at each stage, and his/her next options to browse. In Section 3.4, we describe how is it possible for the users to break free from a rigid top-down navigation schema and browse the system in an opportunistic manner.

3.1. Reverse engineering a system-wide dependence model

This section summarizes the technique and terminology of our earlier work on reconstructing system-wide dependence models [11]. 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 interprocedural dependence graphs [18] and taking the component source code as "system source."
2. The system's configuration artifacts are analyzed to build an *intercomponent 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 sets 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 component.

Fig. 2 gives an overview of the main *types of information* that we collect from various source artifacts to build the SDG. The dashed boxes are used to indicate the parts of the meta-model that represent source code information respectively configuration information and show how they are connected. To construct the CDG, the fundamental information are the nodes (program points), and the edges (data and control dependencies). Informally speaking, a program point is a fragment of source code that could be traced to a location in the source code. For instance, the value of variable v at program point p is a distinguishable element, and can be used as a slicing criterion. We refer to Horwitz et al. [18] for a detailed discussion on the aforementioned elements, and the construction of dependence graphs in procedural languages. For traceability purposes, we also extract some properties of program points, such as their location in the source files and the physical structure of the software artifacts. This information is extracted from the components' implementation. For the ICDG, the information is a component's inputs and outputs, parameters (see Section 3.5), instances, and intercomponent connections. This information is extracted from the configuration artifacts.

Further details on our reverse engineering of the system-wide dependence models go beyond the scope of this paper and we refer to our earlier work [11].

3.2. 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. This characteristic does, however, make them less suited for directly supporting comprehension or visualization [11,19].

To make the detailed information contained in an SDG or slice more suitable for comprehension, we propose a hierarchy of five abstractions (views) aimed at satisfying the needs of safety experts and developers. These information needs range from a black-box survey of the system, via a number of intermediate levels, to a hypertext version of the source code. In addition to providing the required information, these abstractions should be appealing to the users with respect to familiarity and intuitiveness. To achieve this, we aim to reuse visualizations that are already known to our industrial partners as much as possible. In cases where such visualizations do not already exist in the daily activities of our industrial partner, we devise new visualizations that build on the same or very similar visual elements and beacons as the existing ones. These views are constructed from the system-wide dependence model via a combination of slicing, transformation and visualization.

Since, in our case, understanding the system-wide dependencies is required in both directions (i.e., forward and backward), all visualization levels accommodate abstractions over both forward and backward slices. Depending on the nature of the visualization, the abstractions over forward and backward analyses are either visualized in separate diagrams, or, whenever possible, in a single diagram enriched to accommodate both directions of analysis.

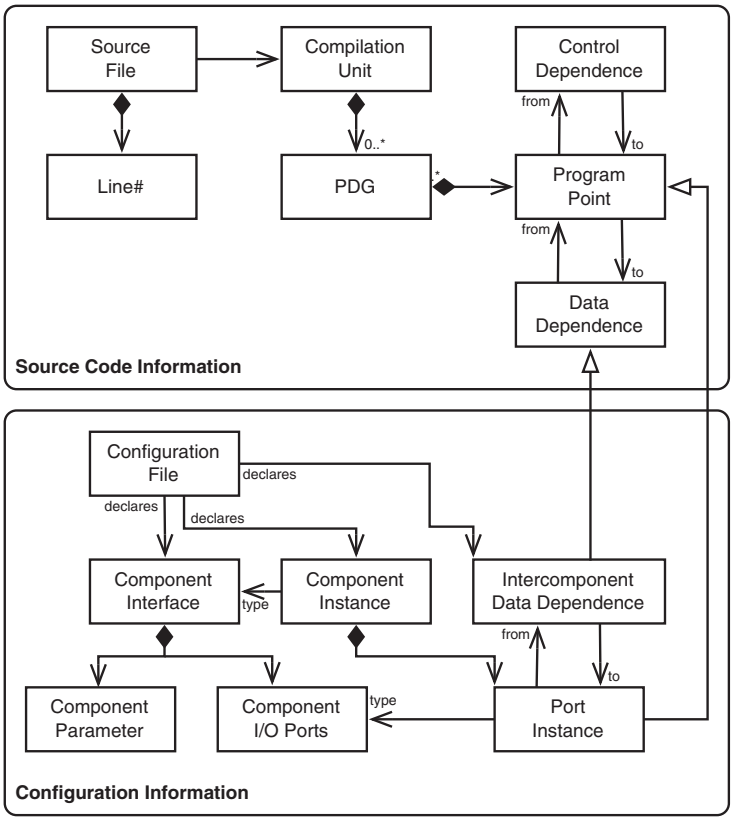


Fig. 2. The main elements from various artifacts used to track information flow.

Such separate (set of) diagrams that belong to the same abstraction layer target the same *type* of desired system elements, and can address similar, but not identical, comprehension requirements. In our description of the abstraction levels, we distinguish between the two analysis directions, if needed.

The various levels are interconnected via hyperlinks to enable easy navigation and to support various comprehension strategies [20]. We distinguish the following hierarchy of views, which are discussed in more detail below:

1. System dependence survey: a black-box view that summarizes possible information flow between the system's sensors and actuators in a matrix;
2. System information flow: a grey-box view that details the information flow between selected sensors and actuators, including the components and ports involved, but abstracts from the component contents;
3. Component dependence survey: a black-box view that summarizes information flow between input and output ports of a component in a matrix;
4. Component information flow: a grey-box view that details the information flow between input and output ports of a component, and the conditions that control this flow, but hides the lower level aspects of a component;
5. Implementation view: a white-box view of the source code enriched with navigation facilities.

(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 Fig. 3).[1] 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.

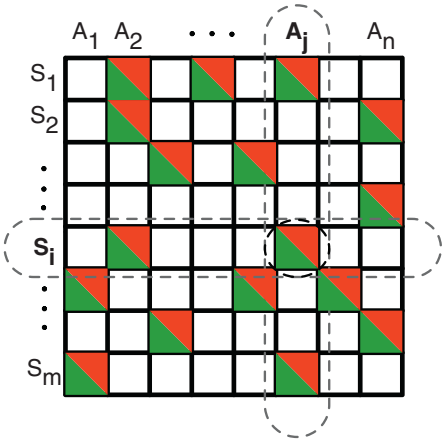


Fig. 3. System-wide dependence survey.

In addition to providing a succinct black-box view, we specifically used this matrix-based presentation as it is already well

[1] Note that the graphs in the paper are actual figures as they are created our tool and shown using a standard webbrowser. They were taken from the study with our industrial collaborator after some renaming for anonymization/nondisclosure purposes.

known to our industry partners. The dependencies from sensors and actuators in each deployment are described in a decision table that is known as a Cause&Effect matrix [21,22]. This matrix serves a reference contract in discussing the desired safety requirements between the supplier and the customers and safety experts. By filling certain cells of a this matrix, the expert can, for example, prescribe which combination of sensors needs to be monitored to ensure safety in a given area. The System Dependence Survey mimics the same visualization over the end-to-end dependencies as they appear in the *implementation* (as opposed to in the requirements documents). Another advantage of such straight-forward matrix-based visualizations is that they are easy to interpret by software engineers and system integrators.

Note that there is only one System Dependence Survey and there is no need to distinguish between forward and backward analysis directions in this view. Although the information contained in forward and backward slices is different in general, in this particular case where slicing is used to identify the end-to-end (abstract) dependencies between sensors and actuators, the overall system-wide information (for all end-points combined) will be the same, no matter what slicing direction is used.

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. To provide navigation to both analysis directions on the next lower level, matrix cells are diagonally divided in two. In a given matrix cell (for example the one at index [S_i, A_j] shown in Fig. 3), the lower left half corresponds to the respective sensor S_i, and the upper right half to the actuator A_j. By clicking on the lower left half of the cell, the user can zoom in on the System Information Flow for that specific sensor, in order to view the forward information flows *originating from* that sensor. By clicking on the upper right half of the cell, the user can zoom in on the System Information Flow for that specific actuator, in order to view the backward information flows that *end in* that actuator.

(2) System information flow: This grey box view shows the intercomponent information flow between particular sensors and actuators, i.e., it shows system-wide slices through the complete system (Fig. 4). In the design of the visualizations at this abstraction level, our goal is to represent component-based systems with an *intuitive* and *familiar* notation, which bears a resemblance to UML 2.0 Component Diagrams [23]. As shown in Fig. 4, components are represented by a rectangular shape (Fig. 4, marker A). Component input ports are represented by a stack of green boxes on the left side of the component, and the output ports with red boxes on the right side. All visualized elements (i.e., sensors, actuators, components, ports, and connections) that are not part of the target information flow are low-lighted, and in gray. We distinguish two variants based on analysis direction:

Backward system information flow: For each actuator, there is a diagram that shows the intercomponent information flow from all sensors to that actuator. The view hides all intracomponent level information in a backward slice through the SDG with actuator A_j as the slicing criterion. The result highlights the actuator and all related sensors, component instances, and intercomponent connections. Fig. 4a shows an example for actuator A_j.

Forward system information flow: For each sensor, there is a diagram that shows the intercomponent information flows that originate from that sensor. The view hides all intracomponent level information in a forward slice through the SDG with sensor S_i as the slicing criterion. The result highlights the sensor and all related actuators , component instances, and intercomponent connections. Fig. 4b shows an example for actuator S_i.

Apart from showing the elements that a sensor influences, or 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 cells in a matrix (see Fig. 5). This black-box view shows which input ports can affect which output ports but hides all details on *how* the information flow is realized. Again there is no need to provide separate matrices for the forward and backward analysis directions because the summarized information is identical. There is one dependency matrix for each component, independent of its instances, because the dependencies are induced by the component source code.

To enable navigation to both analysis directions in more detailed views, matrix cells are diagonally divided in two. Clicking on the lower left half of a cell brings the user to the Component Information Flow for the corresponding input port that shows which *intracomponent* forward information flows *can be affected by* that input port. Clicking in the upper right half of a cell brings the user to Component Information Flow for the corresponding output port that shows which *intracomponent* backward information flows *can affect* that output port.

(4) Component information flow: For a given component and input- or output port, this grey-box shows the intracomponent information flows connected to that port (i.e., there are diagrams for each input port and for each output port of every component). In addition to the input and output ports involved, this view includes all conditions that control the information flow between those ports. We distinguish two variants based on analysis direction:

Backward component information flow: For each output port, there is a diagram that shows the intracomponent information flow from all input ports to that output. Fig. 6a shows an example with output port "AlarmErr" as the slicing criterion (single 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.

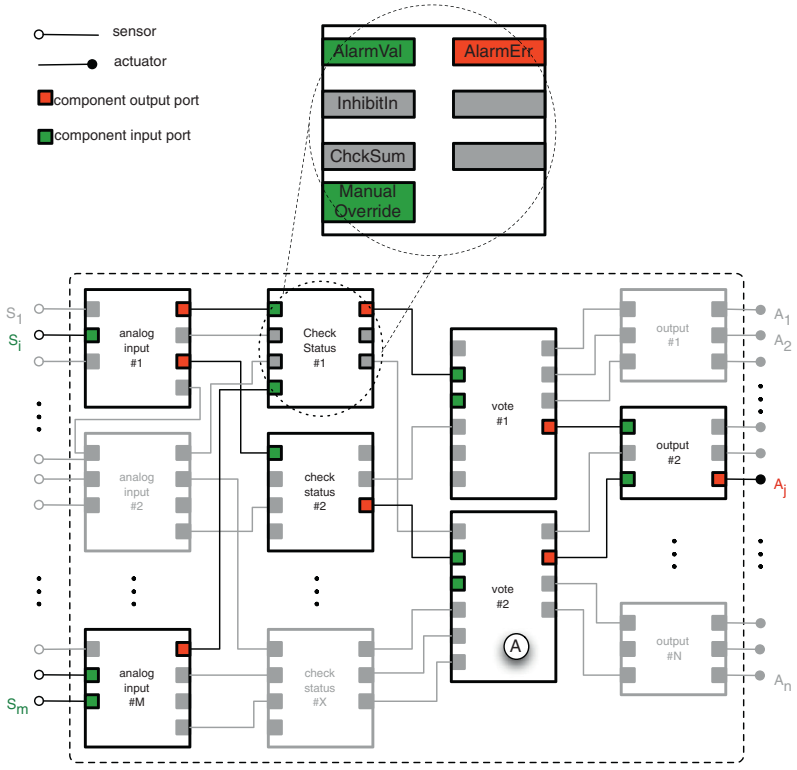
Forward component information flow: For each input port, there is a diagram that shows the intracomponent information flows that originate from that input. Fig. 6b shows an example with input port "InhibitIn" as the slicing criterion (single green node at the top). The output ports affected by InhibitIn are at the bottom (red nodes), and the conditions that control the information flow are shown as yellow squares.

Note that we have chosen to show both the forward and the backward flow in a top-down fashion with inputs at the top and outputs at the bottom to make it easier for a user to orient themselves while changing views. In addition, we combine sequences of conditions into aggregate nodes to reduce cognitive overhead. The details of this refinement are described later, in Section 4.2. The conditional nodes have hyperlinks embedded to navigate to the corresponding location in the source code (indicated by marker A in Fig. 6b).

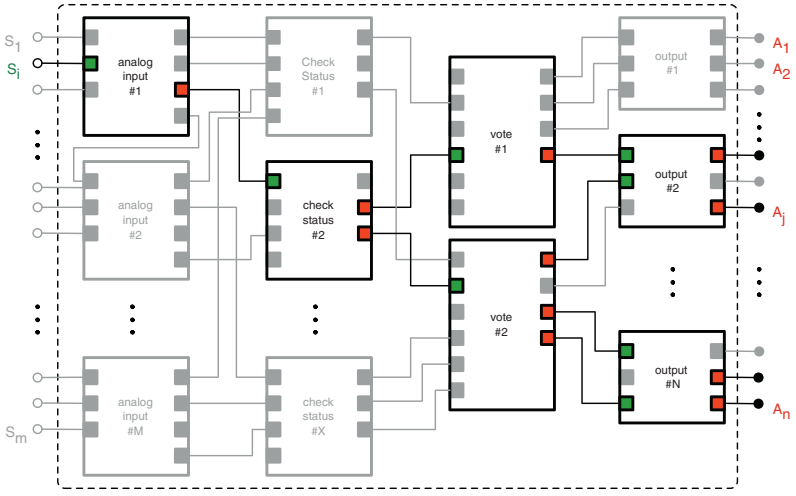
(5) Implementation view: At the lowest level in our hierarchy, the white-box implementation view shows pretty-printed source code with hypertext navigation facilities, e.g., cross-referencing of program entities with their definition. Higher-level views provide links to the source code as a means of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

3.3. Typical usage scenario

A typical scenario takes advantage of the hierarchical design of the abstractions, and is sketched as the following:



(a) Backward slice criterion: A_j



(b) Forward slice criterion: S_i

Fig. 4. Backward and forward system information flows for A_j and S_i, respectively (marker A is used for explanations in the text). (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

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 an actuator (Fig. 3).
2. By selecting a sensor-actuator pair in the matrix, the users zoom in on the System Information Flow that helps them find the components and intercomponent connections that play a role in transferring the values from the selected sensor to the selected actuator (Fig. 4). Depending on which half of the matrix cell is clicked, the users either see the outgoing information flows from a sensor (Fig. 4b), or the incoming information flows towards an actuator (Fig. 4a).
3. By selecting on one of the component instances, they navigate to the Component Dependence Survey. This view can be used to identify which component input ports can (or can not) affect which component output ports (Fig. 5).
4. By selecting a component input-output pair in the matrix, the users focus on the Component Information Flow. This shows the conditions that control how information from the selected input port can reach the selected output port (Fig. 6). Depending on which half of the matrix cell is clicked, the users either see the outgoing information flows from an input port

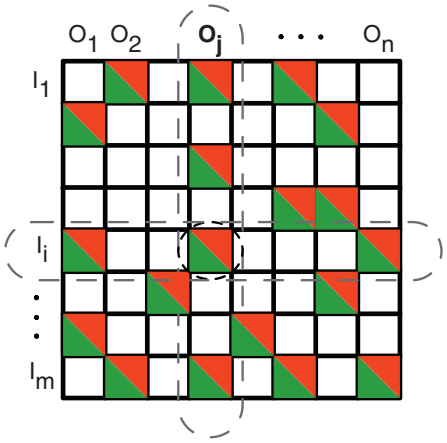


Fig. 5. Component dependence survey.

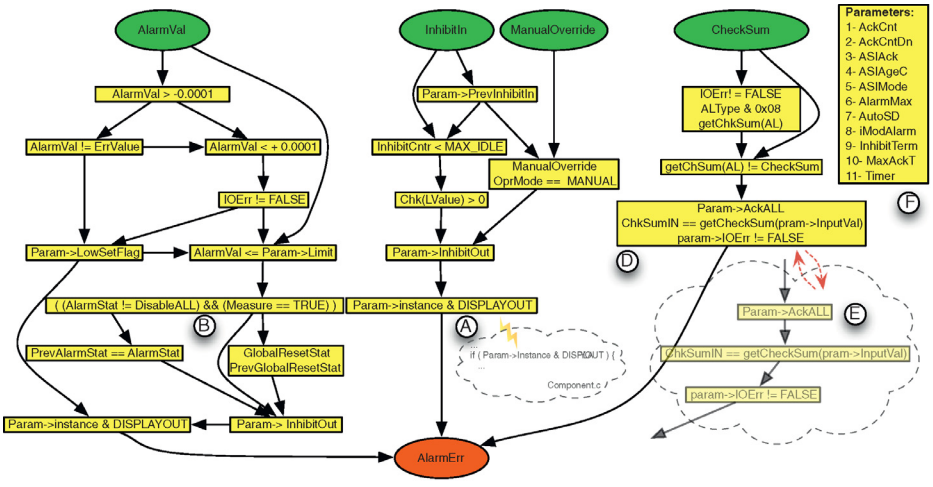
(Fig. 6b), or the incoming information flows towards an output port (Fig. 6a).

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.

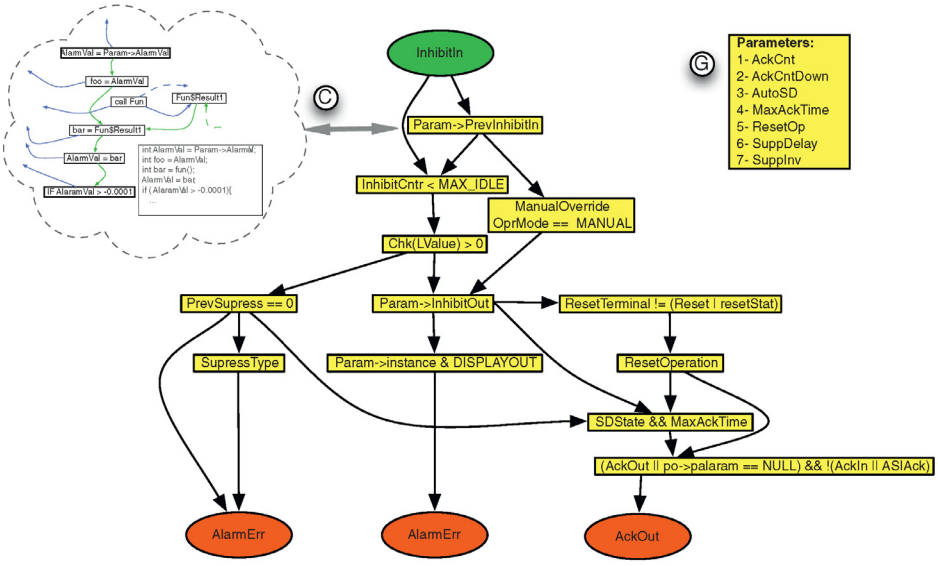
3.4. Enhanced navigation

The aforementioned typical usage scenario of the visualizations supports top-down exploration and comprehension of the information flows. As the user starts from the topmost layer and descends the abstraction hierarchy, the scope of the information flows decreases (i.e., from system-wide to intracomponent), and the amount of details increases (from the system's black-box view to the source code). Apart from that, there are two – conceptually similar – types of information at every abstraction layer: forward and backward information flows.

This highly structured navigation profile helps the novice users in finding their way through the system and prevents that they get lost during their explorations. However, requiring the user to



(a) Backward Component Information Flow



(b) Forward Component Information Flow

Fig. 6. Forward and backward component information flow examples (markers A–G and the cloud-like fragments are used for explanations in the text). (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

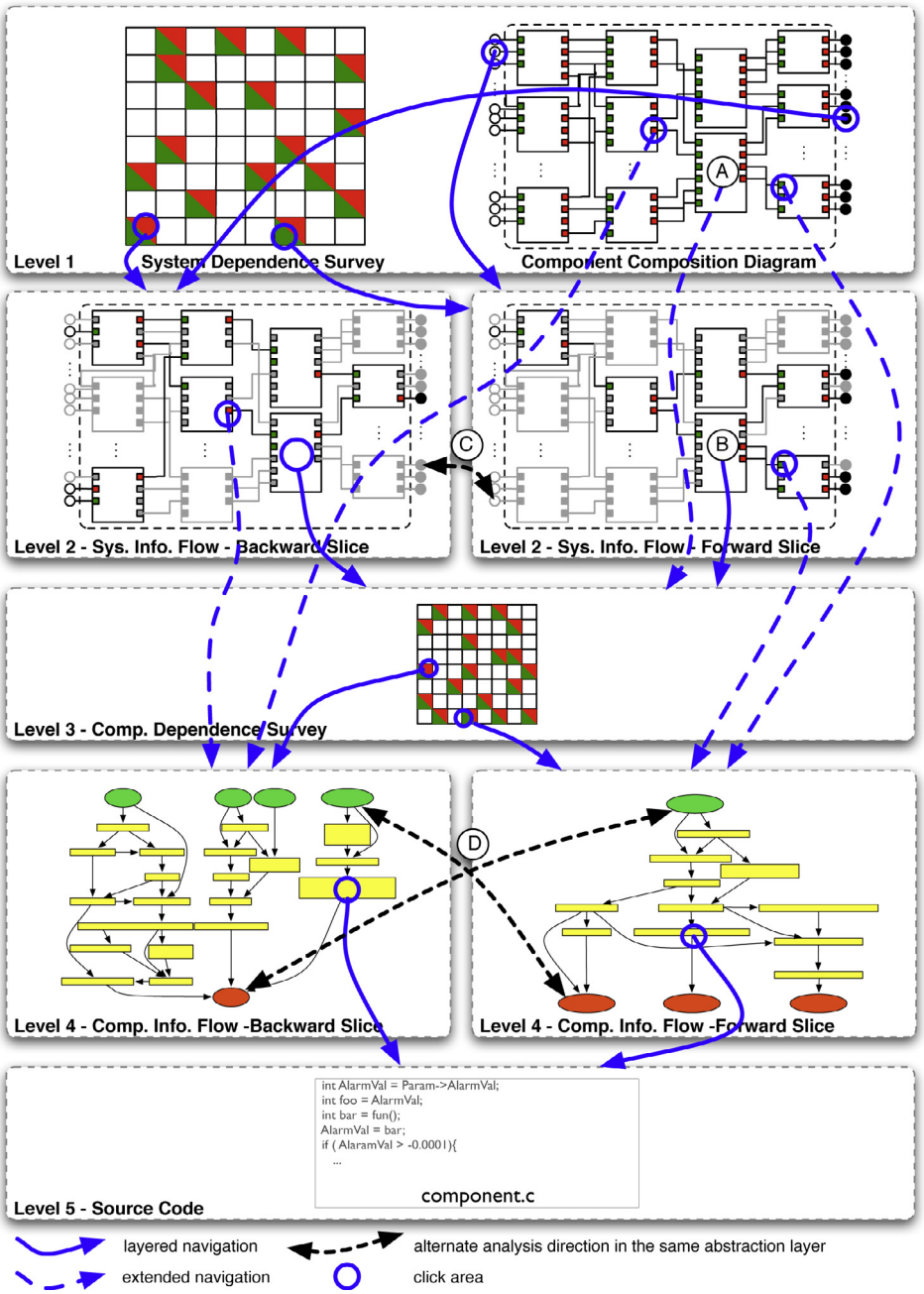


Fig. 7. The navigation structure of the visualizations. Every visualized element, except connections and edges, has a hyperlink (markers A–D are explained in the text). (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

always go through such a fixed five-layer schema would be too strong a restriction. In addition, the structure includes premature commitment to analysis direction which is known to negatively affect usability [24]. Frequent users who are already familiar with the various parts of the system, do not need the structure provided by the layered navigation. Instead, they need more flexible ways to browse and navigate the collected information [20], for example, quickly remind himself of multiple information flows halfway through a maintenance task. Forcing them to stick to a rigid and cumbersome navigation schema would reduce their overall usability experience up to a point where they might eventually abandon the tool.

To address this threat, we provide an enhanced navigation schema that allows users to browse the system more freely and more spontaneously, as well as allowing them to follow the previously described hierarchical navigation. Below, we provide a detailed explanation of the overall navigation structure with the help of Fig. 7, which contains all abstraction levels and highlights the links between the levels.

First, we enhance the system dependence survey with the component composition diagram of the complete system (Fig. 1). This diagram shows how the system is currently configured, with the same graphical notation as in the System Information Flow, and is rendered next to the matrix in the System Dependence Survey (Fig. 7, Level 1). It can be seen as a generic System Information Flow before slicing, and thereby helps to build a coherent mental model for navigating through the system. We deliberately represent component input and output ports in the same color (green and red, respectively) in matrix-based and

Component-Diagram-based visualizations to take advantage of color as a graphical beacon throughout the whole system.

Second, we *activate* almost every visualized element with navigation hyperlinks. In Fig. 7, clicking on any element other than a connection leads to a diagram that shows the information flow with respect to that element. The following navigation rules apply consistently across all layers:

1. Every *system input* port points to the corresponding Forward System Information Flow
2. Every *system output* port points to the corresponding Backward System Information Flow
3. Every *component input* port points to the corresponding Forward Component Information Flow
4. Every *component output* port points to the corresponding Backward Component Information Flow

In addition to these rules, clicking on the graphical representation of a component (Fig. 7, Level 1 and 3) leads to the respective component dependence survey (Fig. 7, markers A and B). The resulting navigation schema is highly intuitive, and enables the users to navigate two or three abstraction layers in one step. Users can browse from the component configuration diagram (Fig. 7, Level 1) to system information flow or component information flow by clicking system or component ports, respectively (without going through system dependence survey or component dependence survey).

Apart from that, the users are not bound to a specific analysis direction once they descend the abstraction hierarchy (i.e., forward and backward), and can alternate the direction at any step. For instance, while investigating a pair of sensor-actuator (S_i, A_j), the users can quickly view the outgoing information flows from S_i and the incoming information flows to A_j by a single click, and without backtracking to higher-level diagrams (Fig. 7, marker C). The same direction switching is provided for component information flow (Fig. 7, marker D).

3.5. Component parameters

As mentioned in Section 2, our industry collaborator is a major producer of various safety and control systems. These products and systems (that is, the individual instances of a product that are installed in the real world) share considerable similarities which is exploited by assembling products in a component-based fashion from reusable components [25]. Nevertheless, there is also a considerable variation between any two concrete installations and the safety systems that monitor and control them. Examples include variations in the actual sensors and actuators that are used, and the respective thresholds at which they trigger, and process specific variables such as what levels are considered hazardous, and variations that follow from complying to different safety standards.

In the systems that we studied, these reusability and variability concerns have been addressed by developing *generic* implementations for the components that are highly customizable and configurable with the help of *component parameters*. These component parameters are used to concretize the functionality of these components in a specific installation of a specific product. The parameters can be set either at system configuration time, or at execution time by user actions. In either case, they can be regarded as *input* to the components; however, they are separate and different from component input ports. The set of each component's parameters is declared in (XML-based) configuration files, similar to the component's ports. For example, for a given component, the user may be able to set a threshold value that some input needs to reach, before the signal is propagated to an output. This threshold is communicated to the components as a parameter and is not considered for determining the normal (process-specific) sensor-actuator information flows.

Because parameters can significantly affect the behavior of components and the way in which they transfer information from their input to output ports, it is essential that developers are aware of, and understand, the parameters' potential influence on the intracomponent information flows. In addition, even though the parameters are described in the component documentation, there can be many parameters for a component and not all of them are relevant for every information flow. Moreover, the discussion of the states and effects of a parameter is scattered over various use-case specific sections, making it difficult to get a comprehensive overview. We propose to highlight the parameters that can affect a given intracomponent information flow in the visualization of that flow. To this end, we enrich component information flow diagrams with a table showing all component parameters that participate in that information flow (Fig. 6, markers F and G).

This presentation has the added advantage of *filtering* the potentially long lists of parameters of a component to the ones that are relevant for the task at hand (i.e. the information flow under consideration). The more effective this filtering is, the lower the overall comprehension overhead. To analyze its effectiveness, Table 1, second row, reports the total number of parameters in a subset of the studied components (selected randomly). The table also shows the minimum, maximum, and average number of component parameters listed in forward and backward component information flow diagrams. The last row of the table shows what percentage of the component parameters is included in component information flow diagrams on average, disregarding the port direction. In other words, this last row can be regarded as an effectiveness measure of the filtering that was achieved. Considering the high filtering percentages in Table 1, we conclude that presenting the relevant component parameters as part of component information flow has a tangible effect on facilitating the comprehension of intracomponent information flows.

Table 1
Component parameters.

Component		1	2	3	4	5
Total number of parameters		79	197	43	36	80
In forward flow	min	0	12	0	3	25
	max	48	29	42	29	25
	avg	21.20	17.75	21.5	10.92	25
In backward flow	min	5	7	0	9	1
	max	72	29	24	35	6
	avg	24.56	14.5	11.92	18.33	3.97
Average # parameters in flow		23.76	15.48	16.08	15.05	18.34
Percentage of total filtered		69.93%	92.15%	62.61%	58.20%	77.08%

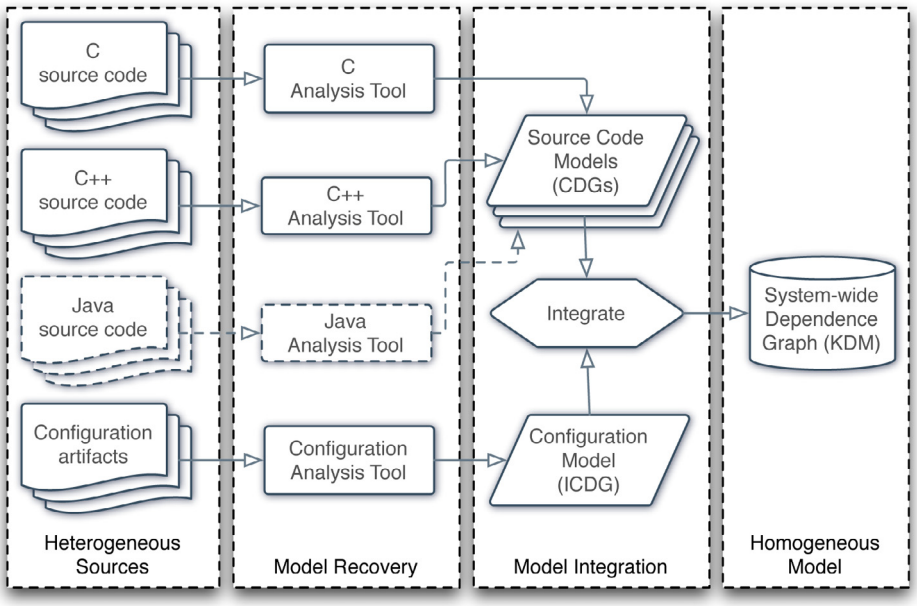


Fig. 8. Integrating models derived from heterogeneous sources into a homogeneous model.

4. Prototype implementation

This section discusses the implementation of the approach described in Section 3 in a tool named FlowTracker. We distinguish three stages in the implementation:

1. Model Reverse Engineering: constructing the homogeneous model of the system from components' source code and configuration files.
2. View Construction: building the hierarchy of the views based on the homogeneous model.
3. Presentation: packaging and presenting the end result to the users.

The rest of this sections provides the mentioned stages in more details.

4.1. Model reverse engineering

We reuse our earlier tool to reverse engineer system-wide dependence graphs (SDGs) from source artifacts [11] in a model-driven approach. Fig. 8 shows a high level overview of our approach. Constructing a homogeneous model from a collection of heterogeneous sources is executed in two phases: (1) a model recovery phase in which we reverse engineer the dependency models of interest from individual source artifacts; (2) a model integration phase in which we merge these individual models into a single homogeneous model of the system. To build the homogeneous model, we use OMG's Knowledge Discovery Metamodel (KDM), a meta-model aimed at representing reverse engineering knowledge [26]. KDM comes with a reference implementation based on the Eclipse Modeling Framework (EMF)[27], which we use in our implementation.

We build on Grammatech's CodeSurfer[2] to create component dependence graphs (CDGs) for the individual components [28]. CodeSurfer is a standalone tool that offers an API in the C programming language by which the CDGs can be programmatically traversed. The bridge between CodeSurfer and Eclipse is implemented in C using Codesurfer's API and the Java Native Interface (JNI) to call methods in Eclipse. Upon traversing every node and edge of the CDGs, we call the appropriate KDM constructor API in Eclipse to inject the corresponding element into the homogeneous model. 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 intercomponent dependence graph (ICDG).[3] Finally, we use a straightforward substitution transformation to integrate the CDGs with the ICDG, and create the final SDG.

As mentioned in Section 3, all dependencies in the SDG need to be interpreted in both forward and backward direction to compute the various information flows. In our original reverse engineered SDG, each dependency is represented by an instance of a stereotyped ActionRelationship class in KDM, where stereotyping ActionRelationship is used to distinguish between different types of dependencies, such as data-, control-, and intercomponent dependencies (for more details on the mapping of SDGs into KDM, we refer to [11]). Unfortunately, KDM does not support ActionRelationships that can be interpreted bidirectionally. To enable slicing by 'natively' traversing the KDM representation, we therefore choose to represent each dependency in our SDG by two (uni)directional ActionRelationships, one for each direction. A downside of this implementation decision is that the number of ActionRelationship objects doubles. This increases the model size, which has a tangible effect on the initial model loading time.[4] On the upside however, there is no cost penalty for computing the system-wide slices, because the traversal of the dependencies in each direction is independent of the presence of dependencies of the opposite direction. This clearly outbalances the alternative where, on average, for half of the slicing computations the inverse dependence relation would have to be computed.

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

[3] https://xml.apache.org/xalan-j/

[4] To give an impression of the model size and execution time, we refer to two of the components in our studied system. A component with 13787 nodes and 46276 edges took 1.996 s to be transformed to KDM. Another component with 61507 nodes and 216956 edges took 9.938 s. Using unidirectional edges, one the other hand, enabled us to compute program slices in trivial time (in the order of ms). See our previous publication for more details [11].

4.2. View construction

During view construction, we enrich the SDG with additional *summary edges* and *aggregate nodes* that capture a number of view-specific abstractions 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 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 [11]. Subsequently, we discuss the abstractions that were added for the various views. The names were chosen so that they map trivially on the names of the views in Section 3.2.

The SysDep relation is based on slices for each of the system's ports and includes the summary edge $(S_i, A_j)_{\text{Backward}}$ if sensor S_i is in the backward slice for A_j, and the summary edge $(S_i, A_j)_{\text{Forward}}$ if actuator A_j is in the forward slice for S_i. Similarly, for each component C, the CompDep_C relation is based on slicing all component ports and including $(I_m, O_n)_{\text{Backward}}$ if input port I_m is in the backward slice for O_n, and $(I_m, O_n)_{\text{Forward}}$ if output port O_n is in the forward slice for I_m.

For each system port P, the relation $\text{SysInfoFlow}_{direction,P}$ is based on slicing the enriched SDG on P with the correct direction. The direction has to be backward for the actuators, and forward for the sensors. For each component C_i in the slice, we use the summary edges of $\text{CompDep}_{C_i,direction}$ 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 toward the 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 port P of every component C, the $\text{CompInfoFlow}_{direction,C,P}$ 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 increases the 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 (Fig. 6, marker B).
2. We replace edges by summary edges that subsume all nodes that are not input ports, conditions, or output ports (Fig. 6, 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 replace both edges (and node y) by a single summary edge (x,z). These summary edges are computed transitively, so that they represent the longest path possible.
3. 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 labeled based on the conditions it represents. For an example, see Fig. 6, marker D for the aggregate node, and marker E for the condition cluster it represents.

Finally, the construction of the Implementation View does not require any additional summary edges or aggregate nodes to be added to the SDG.

4.3. 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 [21,22], 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 port or actuator, the user can zoom in on the information flow *leading to* that port or actuator.

To render the Component Information Flow, 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.[5] We use 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 by using Doxygen.[6] 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.

A positive side-effect of implementing all visualizations in HTML is that we inherit all benefits of the familiarity and features of modern web browsers as part of our user interface. These browsers are widely available and well-known to all prospective users of FlowTracker. Moreover, they provide standard navigation features such as browsing history and bookmark creation that help users to maintain a breadcrumb trail and to store landmarks or points-of-interest for later recall. This helps the users to maintain awareness about of their position, keep an overview of where they have been, and backtrack to earlier locations without considerable burden on their own memory, and supporting various strategies for comprehending software systems [20].

Fig. 9 shows a screenshot of the various Flowtracker views in separate browser windows. The navigation between the views is as described in Fig. 7 and would normally open up in the same browser window. For space reasons, we omitted the component composition diagram as it is similar to the level 2 view without components greyed out, and the opening page which shows background and usage information. Note that the level 2 and level 4 information flow graphs are the same as in Fig. 4(a) and Fig. 6(a), were they are more legible.

5. Discussion

5.1. Static versus dynamic analysis

Our approach is based on static analysis of the system which, by its very nature, computes an approximation of the actual relations that exist in a system at runtime. In theory, more precise information could be obtained by using dynamic analysis, which aims to capture exactly those relations that can be observed on a running system. However, in the context of software intensive control systems, there are a number of limitations: First, in-vivo dynamic analysis of these systems in their real operating environment is generally not an option, due to safety hazards. Second,

[5] http://www.aisee.com/
[6] http://www.doxygen.org/

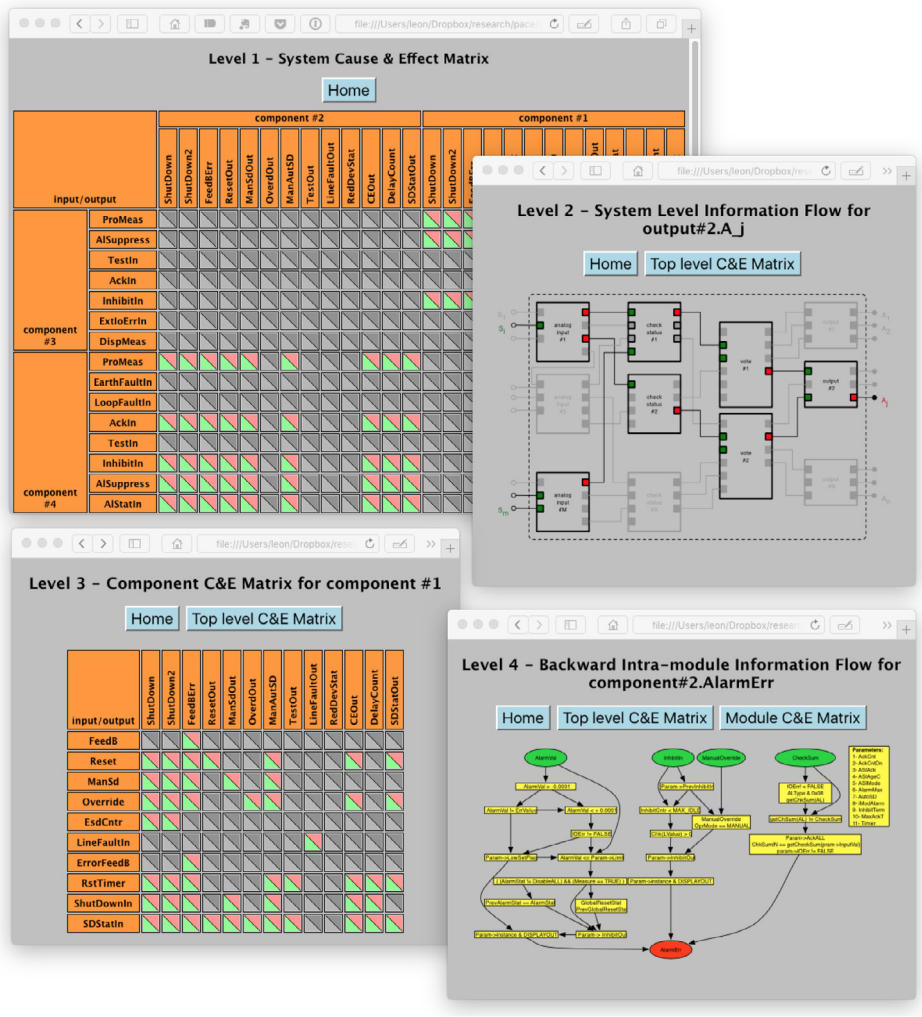


Fig. 9. Screenshot of the various Flowtracker views in separate browser windows.

in-vitro dynamic analysis of such systems requires advanced, expensive, stubs and simulators to replace hardware components and to create realistic execution scenarios, which is typically only available at limited development sites. Finally, this infrastructure for in-vitro analysis is generally in high demand for product development and testing. Faced with these limitations, we set out to investigate an alternative approach based on static analysis.

Since the introduction of static program slicing [16], there have been several improvements to compute more *accurate* slices [see e.g., [29,30]]. However, static program slicing remains a *conservative approximation*, i.e., there might be statements in a slice that have no relation to the slicing criterion. This characteristic has the following effects on the information flows that we compute: (1) They are *safe*; conservativeness guarantees that no dependency goes undetected between component input and output ports, and eventually between system inputs and outputs. Therefore, the users can be assured that when FlowTracker does not show a dependency between a pair of system (or component) input and output, there is no possibility of influence from that input to the output. (2) They may contain *false positives*; it is not guaranteed that a reported input actually *does* influence the value of a given output. In other words, the extracted information flows are a superset of the actual information flows.

5.2. Forward versus backward slicing

As discussed in Section 3, addressing the users' information needs requires computing both forward and backward slices through the SDG to determine information flow. Apart from discussing the suitability of forward and backward slices as we did in that chapter, here we would like to highlight some observations on these two slicing directions, and especially focus on the differences in slice sizes, as this directly affects the cognitive load involved with understanding the various diagrams that we create.

From a black-box point of view, the effects of both directions of slicing are the same, i.e., they detect the same abstract dependence relations between inputs and outputs, as shown in the system dependence survey and component dependence survey matrices. This is true of system-level slices, as well as component-level ones. This observation in our diagrams is a direct implication of the definition of dependence graphs [18]. If a program point P1 is included in the backward slice from program point P2, then P2 is definitely included in a forward slice from P1. Following the same argumentation, the *average* size of forward and backward slices is the same. However, the *distribution* of slice sizes does not have to be the same [31], which makes it interesting to investigate a bit deeper.

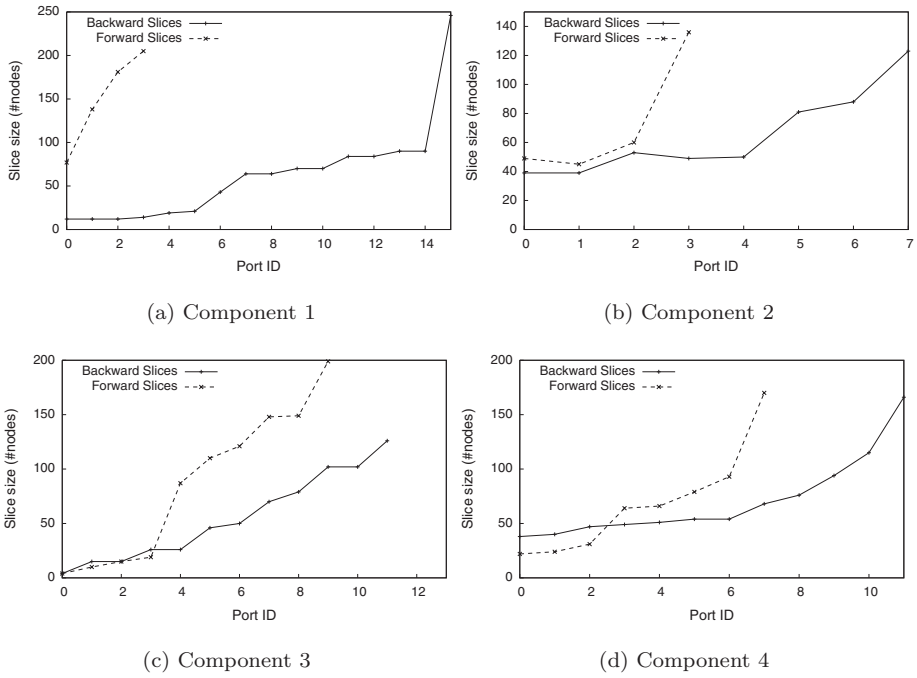


Fig. 10. A comparison of the number of program points (nodes) included in intracomponent slice sizes in the forward vs. backward direction. Forward slices are computed from the component input ports, and backward slices from the output ports. Component ports are sequentially numbered on the horizontal axis (starting from zero), in the order of ascending slice sizes (vertical axis).

From a white-box point of view, i.e., from the perspective of *intracomponent* slices, one *can* observe differences between forward and backward slices. Fig. 10 depicts the slice sizes for a randomly selected subset of the components studied. Slice sizes are measured by the number of program points in each slice. As mentioned before, forward slices are computed from component input ports, and backward slices from component output ports. Component ports are indexed on the horizontal axis in the ascending order of the slice sizes, which is projected on the vertical axis.

In the selected components in Fig. 10, forward slices are clearly bigger than backward slices. Consequently, we observe more complex diagrams in component information flow for forward information flows than backward information flows, which can be considered an obstacle for comprehension. This observation confirms our initial intuition to extract information flows based on backward slices to facilitate comprehension [12]. The same observation could also apply to the number of dependencies (a.k.a. edges in the SDG) included in the forward and backward slices (shown in Fig. 11). The number of edges is not commonly used in measuring slice sizes. However, this number is of interest to us, as it has an indirect, yet major, impact on the complexity of the final diagrams in component information flow.

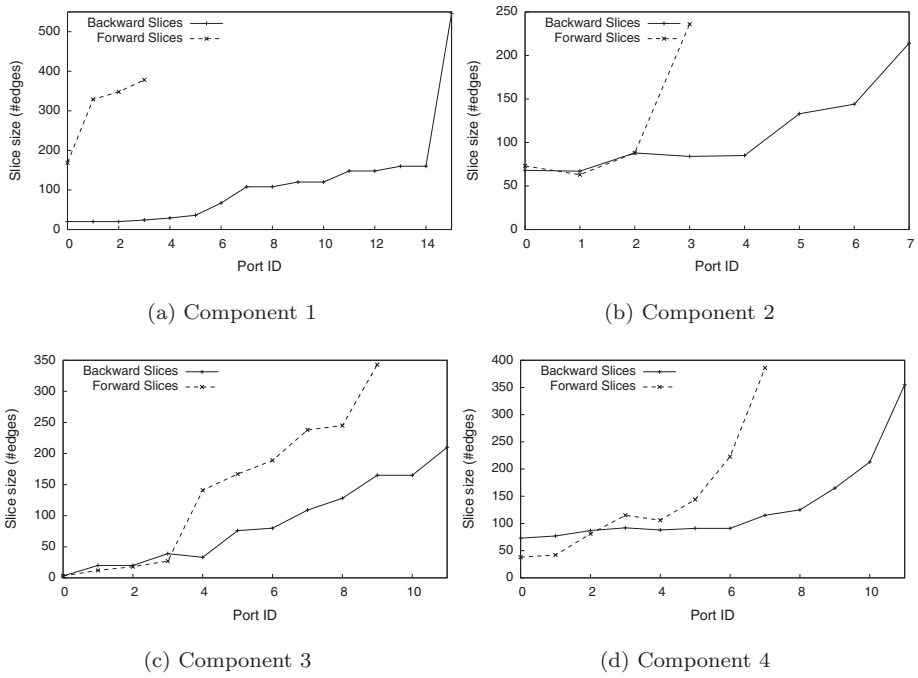
At first, these observations seem to contradict to Binkley and Harman's empirical study which concludes "forward slices are smaller than backward slices" [31]. In this study, the authors provide evidence that the *distribution* of slices sizes for forward slices leans toward smaller numbers compared with backward slices. Their claim is strongly supported by: (1) Computing both forward and backward slices from *every* program point; and (2) Computing forward slices from all inputs and backward slices from all outputs. They gather statistically significant data from a large code base containing a wide range of programs (accumulating over 1 million lines of code), which averages out the majority of architectural- and source code specific characteristics that could affect their conclusions.

The systems that we studied however, do follow a specific component-based architecture, and we need to take into account any special characteristics that this design may have on the analyzed components, before drawing our conclusions. Indeed, a closer look at Fig. 10 reveals that the number of input ports is typically smaller than the number of output ports (indicated by the fact that there are fewer data points for forward slices that for backward slices). Assuming that there is no *unreachable code*[7] in the components, the union of forward slices from all input ports should cover all program points in the component. Likewise, the union of the backward slices from all output ports should cover all program points. Therefore, having fewer input ports than output ports implies having bigger forward slices than backward slices. Closer inspection of the component interfaces showed that most components in the system we studied follow the same pattern, i.e., they have considerably fewer inputs ports than output ports.

The difference in the number of input and output ports might not be the only reason behind the difference in forward and backward slice sizes. Certain other characteristics of the components' source code could be a complementary reason: Binkley and Harman [31] show that the effects of control dependence are the major cause of difference between forward and backward slice sizes. They propose an unproved conjecture that the "tree like" structure of control dependencies in SDGs, which has roots in the typical control statements in structured programming (e.g., "for," "while," and "if" statements), is the decisive factor behind the aforementioned difference in slice sizes. Closer inspection of the source code of the components sampled in Fig. 10 indicated that they do *not* contain loop structure, but include 490 conditional statements in total ("if" and "else if").

This lack of loop structures is indeed one of the special code characteristics of the system under study and follows directly from

[7] Unreachable code are those parts of the source code of a program that can never be executed because there exists no control flow path to that code from the rest of the program [32].



(a) Component 1 (b) Component 2

(c) Component 3 (d) Component 4

Fig. 11. A comparison of the number of dependencies between program points (edges) included in intracomponent slice sizes in the forward vs. backward direction. As in Fig. 10, forward slices are computed from the component input ports, backward slices from the output ports, and component ports are sequentially numbered on the horizontal axis (starting from zero), in the order of ascending slice sizes (vertical axis).

the way in which these systems were architected: In general, component parameters (see Section 3.5) are used to configure the conditional clauses that decide how the data in input ports should be directed to the output ports (see Section 2). This design shows the "data-oriented" nature of the majority of interactions in these systems, on both the intra- and inter-component levels that could also explain part of the gap between our observations and those by Binkley and Harman [31].

We would like to emphasize that the aim of this study was not to replicate Binkley and Harman's study, and our results should not be interpreted as contradicting their claims. However, we conclude that the differences in forward and backward slice sizes deserve more architecture-aware, and perhaps even domain-specific, studies. In addition, we conclude that in the context of our case study, the design and code characteristics are such that backward slices are typically smaller and generate information flow diagrams that are therefore easier to comprehend than the ones that originate from forward slices.

6. 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 [11] 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.

6.1. 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 the 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 we would cover the different roles of prospective FlowTracker user groups. We use such a pre-experimental design, because 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 [33]. 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 [34] (this effect can be paraphrased as "first impressions are hard to change"). This is an important consideration for a domain-specific tool dedicated to a specific audience, like ours.

The evaluation study is conducted two rounds of interview with the same participants, corresponding to the two versions of FlowTracker we have developed so far (distinguished here as V1 and V2). In the first round we evaluate the core functionality of FlowTracker V1, focusing on the aforementioned five abstraction levels and their functionality. As the changes in Flowtracker V2 were driven by the initial evaluation of V1, the second round of evaluations with the same participants focuses on the delta between the two versions to assess whether the changes led to the anticipated improvements. The new features of Flowtracker V2 include:

1. Supporting both forward and backward analysis of information flows in each abstraction level (Section 3.2);
2. Inclusion of component parameters in component information flow (Section 3.5); and
3. Providing an enhanced navigation schema across abstraction levels (Section 3.4).

In addition to evaluating the deltas, we include a number of "overall" questions to capture a holistic view of the positive and negative aspects of FlowTracker usability. As FlowTracker V2 is a mere continuation of its predecessor, for the benefit of cohesion and readability, we present the results of the two rounds of interview together as a single complete evaluation study. However, we do present the results from the each round of interview separately, whenever necessary to maintain accuracy.[8]

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 system studied; his focus is more on individual modules than complete systems. P2 is both a system integrator and a system 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 interconnections. P3 is a safety expert who handles the certification process together with third-party certification bodies, such as DNV GL or TÜV.[9] 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 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 doctoral studies, so we refer to them as junior developers. We include this second group of subjects with a different perspective to decrease the potential bias toward the specific traits of the case study, a bias that could be caused by only selecting subjects from our industrial partner [35].

Preparation: In both rounds, all evaluation sessions were conducted independently of one another, and the results were aggregated after all participants finished the evaluation. Each session started with a brief presentation of FlowTracker ($\sim$ 10 min). The presentation included a walk-through of a typical usage scenario, similar to Section 3.3. The junior developers were given an extra presentation on the system studied, 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, that 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. In both rounds, we continued training until the participants acknowledged full confidence in their ability to work with FlowTracker, before switching to the actual evaluation.

Data collection: The evaluation itself consists of *semi-structured, researcher-administered, interviews* that, for consistency over various participants, were driven by a common questionnaire that served as an interview plan. This questionnaire consisted of 30 closed questions for which the answers were ranked on a 5-point Likert scale and 6 open (discussion) questions in the first round, and 24 closed questions in the second round. Questions where both positively and negatively phrased to break answering rhythms and avoid steering the subjects [36]. In total, each session lasted between 60 and 90 min in the first round, and 45 to 60 min in the second round.

[8] Note that the results reported in our earlier paper [12] were exclusively from the first round of evaluation on FlowTracker V1.

[9] For more information about these certification organizations we refer to their home pages, respectively http://www.dnvgl.com/ and http://www.tuv.com/.

In line with our goal to conduct a qualitative evaluation, we consciously choose to conduct researcher-administered interviews over having participants fill out the questionnaires themselves. Based on the answers, the interviewer could elicit as much feedback as possible by means of relevant follow-up or clarification questions. In addition, 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), and transcribed and analyzed them using the ELAN multimodal annotation tool [37]. 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 first round of interview, 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.

6.2. Findings

In the remainder of this section, we present the major takeaways, 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 developers, 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 that 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 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.

With respect to the use of each half of a matrix cells as a way to zoom into respectively forward and backward information flows (a feature only in V2), the general response was highly positive. All participants viewed this feature as an added value that outweighs the additional complexity on the user interface ("has far better functionality," according to P4). Two of the junior developers (P5 and P6) wanted more visual aids on the graphical user interface to help the users. They believed that having different cells "creates the expectation" that the cells would lead to different places. However in FlowTracker's matrices, the *right-half* section of all the cells in a single *column* leads to the same diagram representing the information flows leading to that *output* (i.e. there is redundancy in the user interface). Likewise, the *left-half* section of all the cells in a single *row* leads to a single diagram that represents the information flows originating from that *input*. In this situation, P5 and P6

Table 2
Summary of the results on system dependence survey.

Subjects	Feedback
All	Frequent need to know about system I/O dependencies.
All	Alternative solution is manual code inspection.
All	Intuitive presentation, familiar to our industry partners.
All	Added value to have dependencies both in forward and backward analysis.
P5, P6	Needs more graphical features, such as highlighting upon mouse hover.

would like the user interface to emphasize all elements that point to the same destination, e.g., by highlighting all "equivalent" elements at the moment one of them is "hovered" over by a mouse. At one point during the evaluation, P6 indicated that she would like to be able to click on the *title row* and the *left-most* column of the matrix to navigate to the corresponding forward and backward information flows (i.e., directly on the port names instead of the matrix cells). However, she later reconsidered her choice, fearing that the users would have to *drag* the mouse too much to be able to switch between forward and backward information flows in larger matrices. We foresee that both these requests can be easily, and transparently, addressed with some additional client-side scripting and further extending the enhanced navigation scheme.

For better accessibility, Table 2 contains a summary of key takeaways on System Dependence Survey.

(2) System information flow: The subjects were generally satisfied with the functionality of this view, that indicates which components, ports, and sensors can affect the value of a given actuator, and in V2 also indicates which of these elements are affected by a given sensor. FlowTracker currently shows all components and ports, and *highlights* only those elements that participate in the target information flow; the others are dimmed. An alternative could be to hide the unused elements in 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 about 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* intercomponent 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.

Table 3
Summary of the results on system information flow.

Subjects	Feedback
All	Generally satisfied with the functionality of this abstraction level.
P1, P2, P3, P5	Preferred to see all elements, distinguished by high/low light.
P4, P6	Extra information may lead to confusion (better to remove elements).
All	More interactive navigation needed, e.g. hiding unwanted elements.
All - P5	Appropriate intermediary between System Dependence Survey and Component Dependence Survey.
P5	Would like to jump directly from System Dependence Survey to Component Dependence Survey.
All	Intuitive presentation. Very beneficial for system integrators.
P2	Suggests to add type checking among connected ports to this view.

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.

Considering the forward and backward information flows together, one could argue that there is a certain degree of redundancy in the visualizations (i.e., portions of information flows could be repeated multiple times in forward and backward system-wide information flows in V2). However, none of the participants regarded this as a problem and believed both directions of information flows are necessary to visualize. Table 3 contains a summary of key takeaways on System Information Flow.

(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. Table 4 contains a summary of key takeaways on Component Dependence Survey.

Considering FlowTracker V2 and using the matrix cell halves for navigation, the feedback was consistent with the feedback we received for the System Dependence Survey.

(4) Component information flow: We received mixed feedback regarding this view. The most positive responses came from the

Table 4
Summary of the results on system information flow.

Subjects	Feedback
All	Generally satisfied with the functionality of this abstraction layer.
P3	Top priority for the certification process.
P1	Alternative solution: manual code inspection.
P1	Doubts if the view can be used for certification purposes.
P6	Data and control dependencies should be distinguishable.

Table 5
Summary of the results on system information flow.

Subjects	Feedback
-	Mixed feedback regarding the functionality of this abstraction layer.
P1,P2,P3	Industry partners were generally more positive.
All - P6	Conditional points in source code are relevant for information flows, and deserve to be visualized.
P6	Not intuitive. True and false edges of conditional statements should be distinguishable.
P5	Probably not useful for certification purposes.
P3	A means to demonstrate actions taken to external certifiers.
All	More interactive facilities needed for the diagrams, e.g. zooming and hiding nodes.
All	Highly positive to show forward and backward information flows.

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. The design does not resemble the more well-known matrix or UML diagram styles of 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 intracomponent information flows, and should be highlighted and put in perspective to improve comprehension. The subjects also indicated that *"such graphs clearly show the intracomponent 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 labeled with "True" or "False" to indicate which edge would be used if the condition were 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 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 measures to better deal with the larger diagrams. In addition to zooming, another concrete suggestion was to have the option of seeing exclusively the information flow that starts from a single (selected) component input port. We foresee that many of these requests can be fulfilled quite easily by incorporating a graph viewer that is better than the one that is now used in the prototype.

With respect to the visualization of forward and backward information flows in FlowTracker V2, all participants gave highly positive feedback and mentioned that this helped them better with answering program comprehension questions. This positive feedback for including both analysis directions was consistent over all layers of abstraction.

Table 5 contains a summary of key takeaways on Component Information Flow.

Component parameters: A new feature in FlowTracker V2 is that the component parameters that are relevant to an information flow are shown in the view (see Section 3.5). All participants, especially P3, regarded component parameters as highly important for the better comprehension of the intracomponent information flows, and therefore, appropriate to be visualized. In the first round of evaluations, before this feature was added, P3 regarded them as a top priority to visualize. Filtering out the irrelevant component parameters for each information flow was seen as highly positive for the comprehension by all participants. Considering the large number of parameters for each component, and the sharp decrease that is achieved by filtering the relevant parameters for each information flow (as shown in Table 1), this is hardly surprising (especially since the only alternative is manual inspection).

However, most participants stated that showing the relevant components parameters in a list was not enough for them. For example, P3 expressed that she was certain that component parameters have influence on the information flows and did not perceive them as a subsidiary feature, but as an intrinsic part of the information flow. In other words, the participants wanted to see better where a given component parameter could affect the component information flow. P3 and P6 would like to see the *"relevant"* portions of the diagrams highlighted when the user *"hovers"* the mouse over any of the component parameters in the list. P5 wanted the parameters to be visualized in separate nodes visually connected to the relevant portions of the information flows. In a nutshell, the consensus of P5 and P6 was that the relation between the component parameters and the information flows should be more explicit. In a follow-up discussion P3 indicated that she would also like to see the effect of *"changing the value of component parameters"* on the information flows *"on the fly"* and interactively. For example, once the *new* value of a component parameter causes a conditional clause to be evaluated as "False", the infeasible portions of the information flow are to be hidden. In other words, P3 wants the information flows to be *"animated"* with respect to the values of component parameters at the execution time.

In contrast to the previously mentioned sophisticated features of component parameter visualizations, P4 wanted only to see more information about the *data types* of the parameters in the current listing. In his view, this information can help the developers, especially in cases where the component parameters are of "enumeration" or "Boolean" types. He stated that adding more information about the component parameters would make the diagrams too complicated and decrease comprehensibility.

Table 6 contains a summary of key takeaways on component parameter in FlowTracker V2.

(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 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

Table 6
Summary of the results on component parameters.

Subjects	Feedback
All	Component parameters are relevant for the understanding of information flows.
All	Only listing relevant component parameters is not enough for understanding their effect on information flows.
P3, P6	Would like to see the relevant subset of the information flow highlighted, upon hovering the mouse on component parameters.
P5	Component parameters should be visualized as separate nodes, and connected to the relevant nodes on the respective information flows.
P3	Would like to see the effect of changing component parameters on the fly.
p4	Would like to see data types of component parameters stated directly in the diagram.

Table 7
Summary of the results on the implementation view.

Subjects	Feedback
All	Inclusion of this view helps to remove the gap between visualizations and the source code.
All	Direct hyperlinks from the diagrams to the source code beneficial for comprehension and traceability.
P3	Beneficial for certification purposes.
P4, P6	Only beneficial for certification if the certifier knows the code base.
P2, P5	No comment on usability for certification purposes.

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 only if 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 unsure 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."*

Table 7 contains a summary of key takeaways on the implementation view.

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."*

All participants believed that providing the users with visualizations that represent forward information flows, as well as backward information flows, is highly beneficial for comprehension (V2). Unanimously, the participants said that being able to distinguish between the following two closely related questions helps them to find: (1) what elements have an effect a given output?, and (2) what elements are affected by a given input? The benefit was acknowledged both at the system-level, and component-level information flows. P4 also mentioned that having access to both directions of information flows is not only beneficial for comprehension, but also helps the users to *"follow the flow more easily"* and readily.

All participants regarded the enhanced navigation schema (see Section 3.4) as a major improvement over the fixed, layered navigation method. The choices of elements that lead to other visualizations (at the same or other abstraction layer) were deemed *"intuitive"* by most of the participants. P4 found the choice of input (output) ports to jump the forward (backward) information flows very effective and *"easy to remember, but only after the first introduction to the tool"* (which could be interpreted as indicating that this may not be the most intuitive choice). Using the enhanced navigation, all participants were confident in saying that user information needs can be satisfied faster, and the overall user experience in V2 is better than V1.

Table 8
Summary of the results on the overall usability issues.

Subjects	Feedback
All	Satisfies by the intuitiveness of the diagrams in the tool.
All	Visualizing information flows from forward and backward perspectives is beneficial. Information needs of the users can be satisfied more quickly.
P1	Beneficial to have FlowTracker integrated into their programming IDE.
P4,P5,P6	FlowTracker is more beneficial at later stages of development like integration. No acute need of integration to programming IDEs.
All	FlowTracker is excellent for system comprehension.
All	FlowTracker is beneficial for certification, but only to some extent.

The subjects would like to see the tool more closely 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 using 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.[10] 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.

Table 8 contains a summary of key takeaways on the overall usability of FlowTracker.

6.3. Threats to validity

It could be argued that the number of subjects in our study is 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 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 the various roles of prospective FlowTracker users. In addition to the industrial subjects, we added a second group of subjects with a different perspective to avoid bias toward 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 questions and follow-up questions, and instruct-

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

ing 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 of the evaluators; (2) Each of the two authors independently transcribed and analyzed the interviews. Afterward, 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 in which subjects did not (only) give a closed answer but included more discussion.

A potential concern with respect to generalization is that our evaluation included only 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, in which we asked the stakeholders to identify the most qualified senior engineers who could represent these roles in the evaluation. In addition, it turned out that subjects with a given role generally also had experience with some of the other roles, which also helps to create a more balanced picture.

With respect to threats to external validity, our solution to constructing homogeneous dependence models of heterogeneous component-based system, is designed for component models that use distinct directional inter-component communication between addressable data points. Examples of such component models include the Koala [13] and Spring [14] component models, and the proprietary solution used by our industrial collaborator. In earlier work, we have developed prototype tool-support for computing system-wide slices in component-based systems built upon on the Spring framework [14,38] and the Java programming language. Spring offers a component model for modern Java-based enterprise applications based on Inversion of Control (IoC) [39]. Our initial evaluations on smaller Java systems show promising results, however, as we were lacking industry-quality tools for dependence graph generation (comparable to what CodeSurfer gives us for C/C++ [28]), we have not been able to replicate the study with real-world industrial systems. Nevertheless, although our approach is not specifically tied to the component model used in our case study, there are various other forms of inter-component communication that our approach has not been tested against, and may require further extensions of the model extraction of model merging stages depicted in Fig. 8.

7. Related work

Maletic et al. [40] 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. [41] use leveled data flow diagrams to aid professional spreadsheet users in comprehending large spreadsheets. 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 data flow and control flow information, while they analyze data flow dependencies in formula-rich spreadsheets.

Krinke [19] reports on various attempts to visualize program dependence graphs and slices via existing (algorithmic) graph layout tools. 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 from the slicing criterion. In contrast, we developed multiple layers of abstraction to reduce the complexity of system-wide slices and show only the information relevant to the particular task and users. We provide links between the various views that can be navigated down to a textual representation of source as a last resort.

Pinzger et al. [42] 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, such as 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.

Rountev [43] investigates data-flow analysis in component-based systems that have a similar communication model as our case study system, i.e. systems that do not contain call-backs. In contrast to the work presented here, their approach does not incorporate the configuration information into the analysis, as their method does not work with a system-wide dependence structures. The study also does not present any tool support.

Yang et al. [44] tackle the problem of control-flow analysis in framework-based and event-driven systems in the Android ecosystem. They contribute a new *call-back control flow graph* and apply a variation of graph reachability to compute a conservative sequences of callbacks. Their approach is different to ours in not including data dependencies into their analysis. Also their study does not concern the problem of how best presenting the extracted control-flows in a human-readable manner. However, their study is of special interest to us as it opens up new opportunists for extending our method to component-based systems that have more complicated communication models based on call-backs.

One should note that the system-wide analysis of software systems has been studied by several authors (see for example Mysore et al. [45], Enck et al. [46], and Lee et al. [47]). However, these studies generally use (variations of) dynamic analysis as their means to overcome complicating factors such as language-heterogeneity, or component-based implementation. The fact that dynamic analysis techniques can be used to circumvent these challenges should hardly come as a surprise, as the complications

manifest themselves at design/compile time. However, for large safety-critical cyber-physical systems like the ones built by our industrial collaborator, dynamic analysis is simply not an option, as instrumentation would affect operation, and simulation of realistic inputs is not possible. For this reason, our study explores the opportunities that are offered by pure static analysis on design/compile time software artifacts.

We refer to our previous work Yazdanshenas and Moonen [11] for a detailed discussion of work related to our method to build system-wide dependence models from heterogeneous source artifacts.

8. 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 a 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 analysis is lacking, because it is hindered by the heterogeneous nature of these artifacts.

8.1. 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 propose a hierarchy of views that represent system-wide information flows at various levels of abstraction, aimed at supporting both safety domain experts and developers (addressing Goals 1 and 2 from Section 2);
2. We present the transformations that help us achieve these views from the system-wide dependence models and discuss the different trade-offs between scope and granularity (addressing Goals 1 and 2 from Section 2);
3. We discuss how we have implemented our approach in a prototype tool;
4. We report on two qualitative evaluations of the effectiveness and usability of the proposed views for software development and software certification (addressing Goal 3 from Section 2). The evaluation results indicated that the prototype was already very useful. In addition, a number of directions for further improvement were suggested.

8.2. 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[11] or D3[12].

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.

There were some interesting extensions to FlowTracker that were brought up during the evaluation. One example is the possibility of including some kind of automated type checking for component interconnections 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.

Based on the last discussions with our industry partner, there is a desire to better understand what effects changing a given parameter values will have on the information flows that can be achieved. Based on our earlier experiences with (static) value range propagation in source code for embedded systems, we do not foresee that these questions can be answered using static analysis [48]. An alternative is to extract this parameter change information using dynamic analysis methods. Leveraging the capabilities of static and dynamic analysis not only helps to increase the accuracy of the information flows, but also opens many options toward better visualization of intracomponent information flows using concrete values of the component parameters and conditional clauses at run time.

A final direction for future work is the integration of our tooling with an IDE, such as 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 and bookmarking. 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.

[11] http://raphaeljs.com/

[12] 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. This work was partly funded by Simula Research Laboratory and by the Research Council of Norway through the projects inspectIT (#191171), evolveIT (#221751) and Certus SFI (#203461).

References

[1] A. Abran, J. Moore, P. Bourque, R. Dupuis, L. Tripp, Guide to the Software Engineering Body of Knowledge - 2004 Version - SWEBOK, IEEE-Computer Society Press, 2005. ISBN 0-7695-2330-7.

[2] C. Szyperski, Component Software: Beyond Object-Oriented Programming, second, Addison-Wesley, 2002. ISBN 0-201-74572-0.

[3] J. Li, R. Conradi, O. Slyngstad, M. Torchiano, M. Morisio, C. Bunse, A state-of-the-practice survey of risk management in development with off-the-shelf software components, IEEE Transactions on Software Engineering 34 (2) (2008) 271–286. ISSN 0098-5589.

[4] Y. Wu, D. Pan, M.-H. Chen, Techniques for testing component-based software, in: Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems, 2001, pp. 222–232. ISSN 1050-4729.

[5] S. Beydeda, V. Gruhn, State of the art in testing components, in: Third International Conference on Quality Software, 2003. Proceedings, IEEE, 2003, pp. 146–153. ISBN 0-7695-2015-4.

[6] S. Beydeda, V. Gruhn, Testing Commercial-off-the-Shelf Components and Systems, Springer, 2005. ISBN 3-540-21871-8.

[7] H.-G. Gross, No Component-Based Software Testing with UML, Springer, 2005. ISBN 3-540-20864-X.

[8] D. Strein, H. Kratz, W. Lowe, Cross-language program analysis and refactoring, in: Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on, 2006, pp. 207–216.

[9] J. Steele, N. Iliinsky, Beautiful Visualization, Looking at Data through the Eyes of Experts, first, O'Reilly Media, 2010. ISBN 978-1-4493-7986-5.

[10] M. Petre, Mental imagery and software visualization in high-performance software development teams, J. Vis. Lang. Comput. 21 (3) (2010) 171–183. ISSN 1045926X.
[11] A.R. Yazdanshenas, L. Moonen, Crossing the boundaries while analyzing heterogeneous component-based software systems, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 193–202. ISSN 1063-6773.
[12] A.R. Yazdanshenas, L. Moonen, Tracking and visualizing information flow in component-based systems, in: IEEE International Conference on Program Comprehension (ICPC), 2012.
[13] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, The koala component model for consumer electronics software, Computer 33 (3) (2000) 78–85. ISSN 00189162.
[14] P. Manickam, S. Sangeetha, S. Subrahmanya, Component-Oriented Development and Assembly: Paradigm, Principles, and Practice using Java, first, Infosys Press, CRC Press, 2013. ISBN 1466580992.
[15] L. Hatton, Safer language subsets: an overview and a case history, MISRA c, Inf. Softw. Technol. 46 (7) (2004) 465–472. ISSN 09505849.
[16] M. Weiser, Program slicing, in: International Conference on Software Engineering (ICSE), IEEE, 1981, pp. 439–449. ISBN 0897911466.
[17] K. Gallagher, D. Binkley, Program slicing, in: 2008 Frontiers of Software Maintenance, IEEE, 2008, pp. 58–67. ISBN 978-1-4244-2654-6.
[18] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Program. Lang. Syst. 12 (1) (1990) 26–60. ISSN 01640925.
[19] J. Krinke, Visualization of program dependence and slices, in: IEEE International Conference on Software Maintenance (ICSM), 2004, pp. 168–177. ISBN 0-7695-2213-0.
[20] M.-A. Storey, Theories, tools and research methods in program comprehension: past, present and future, Softw. Qual. J. 14 (3) (2006) 187–208. ISSN 0963-9314.
[21] X.-G. Lin, Structural techniques in the design of control systems, University of Queensland, 1991 Ph.d. thesis.
[22] L. Hopkins, P. Lant, B. Newell, Output structural controllability: a tool for integrated process design and control, J. Process Control 8 (1) (1998) 57–68. ISSN 09591524.
[23] T.O.M. Group, Unified modeling language: superstructure v2.0, 2005, http://www.omg.org/spec/UML/2.0/Superstructure/PDF/.
[24] T.R.G. Green, M. Petre, Usability analysis of visual programming environments : a cognitive dimensions ' framework, Vis. Lang. Comput. 7 (1996) 131–174.
[25] S. Deelstra, M. Sinnema, J. Bosch, Product derivation in software product families: a case study, J. Syst. Softw. 74 (2) (2005) 173–194. ISSN 01641212.
[26] OMG, Architecture-driven modernization (ADM): Knowledge discovery metamodel (KDM) - v1.2, 2010, http://www.omg.org/spec/KDM/1.2/.
[27] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, Addison-Wesley, 2009. ISBN 0321331885.
[28] P. Anderson, 90% perspiration: engineering static analysis techniques for industrial applications, in: IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2008, pp. 3–12. ISBN 978-0-7695-3353-7.
[29] F. Tip, A survey of program slicing techniques, J. Program. Lang. 3 (3) (1995) 121–189.
[30] J. Silva, A vocabulary of program slicing-based techniques, ACM Comput. Surv. 44 (3) (2012) 1–41. ISSN 03600300.
[31] D. Binkley, M. Harman, Forward slices are smaller than backward slices, in: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), 2005, pp. 15–24. ISBN 0-7695-2292-0.
[32] S.K. Debray, W. Evans, R. Muth, B. De Sutter, Compiler techniques for code compaction, ACM Trans. Program. Lang. Syst. 22 (2) (2000) 378–415. ISSN 01640925.
[33] D.T. Campbell, J. Stanley, Experimental and Quasi-Experimental Designs for Research, Wadsworth, 1963. ISBN 0395307872.
[34] A. Tversky, D. Kahneman, Judgment under uncertainty: heuristics and biases., Science 185 (4157) (1974) 1124–1131. ISSN 0036-8075.
[35] J. Nielsen, R. Molich, Heuristic evaluation of user interfaces, in: SIGCHI Conference on Human Factors in Computing Systems, ACM, 1990, pp. 249–256. ISSN 1556-3669.
[36] A.N. Oppenheim, Questionnaire Design, Interviewing and Attitude Measurement, Continuum, 1992. ISBN 1855670437.
[37] H. Brugman, A. Russel, Annotating multi-media / multi-modal resources with ELAN, in: Fourth International Conference on Language Resources and Evaluation (LREC), 2004, 2004. http://www.lat-mpi.eu/tools/elan/.
[38] C. Walls, Spring in Action, third, Manning Publications, 2011. ISBN 1935182358.
[39] R.E. Johnson, B. Foote, Designing reusable classes abstract designing reusable classes, J. Object-Orient. Program. 1 (1988) 22–35. ISSN 0896-8438.
[40] J. Maletic, A. Marcus, M. Collard, A task oriented view of software visualization, in: IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2002, pp. 32–40. ISBN 0-7695-1662-9.
[41] F. Hermans, M. Pinzger, A.V. Deursen, Supporting professional spreadsheet users by generating leveled dataflow diagrams categories and subject descriptors, in: International Conference on Software Engineering (ICSE), 2011, pp. 451–460.
[42] M. Pinzger, K. Graefenhain, P. Knab, H.C. Gall, A tool for visual understanding of source code dependencies, in: IEEE International Conference on Program Comprehension (ICPC), 2008, pp. 254–259. ISBN 978-0-7695-3176-2.
[43] A. Rountev, Component-level dataflow analysis, in: International Conference on Component-Based Software Engineering (CBSE), Springer, 2005, pp. 82–89.
[44] S. Yang, D. Yan, H. Wu, Y. Wang, A. Rountev, Static control-flow analysis of user-driven callbacks in android applications, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, 2015, pp. 89–99. ISBN 978-1-4799-1934-5 ISSN 02705257.
[45] S. Mysore, B. Mazloom, B. Agrawal, T. Sherwood, Understanding and visualizing full systems with data flow tomography, ACM SIGPLAN Notices 43 (3) (2008) 211. ISSN 03621340.
[46] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, A.N. Sheth, Taintdroid, ACM Trans. Comput. Syst. 32 (2) (2014) 1–29. ISSN 07342071.
[47] Y.K. Lee, J.Y. Bang, J. Garcia, N. Medvidovic, ViVA: a visualization and analysis tool for distributed event-based systems, in: Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014, ACM Press, New York, New York, USA, 2014, pp. 580–583. ISBN 9781450327688.
[48] C. Boogerd, L. Moonen, On the use of data flow analysis in static profiling, in: International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2008, pp. 79–88. ISBN 978-0-7695-3353-7.