Fine-Grained Change Impact Analysis for Component-Based Product Families

Amir Reza Yazdanshenas
Simula Research Laboratory
Lysaker, Norway
Email: amir.yazdanshenas@simula.no

Leon Moonen
Simula Research Laboratory
Lysaker, Norway
Email: leon.moonen@computer.org

***Abstract*—Developing software product-lines based on a set of shared components is a proven tactic to enhance reuse, quality, and time to market in producing a portfolio of products. Large-scale product families face rapidly increasing maintenance challenges as their evolution can happen both as a result of collective domain engineering activities, and as a result of product-specific developments. To make informed decisions about prospective modifications, developers need to estimate what other sections of the system will be affected and need attention, which is known as change impact analysis.**

This paper contributes a method to carry out change impact analysis in a component-based *product family*, based on system-wide information flow analysis. We use static program slicing as the underlying analysis technique, and use model-driven engineering (MDE) techniques to propagate the ripple effects from a source code modification into all members of the product family. In addition, our approach ranks results based on an *approximation* of the scale of their impact. We have implemented our approach in a prototype tool, called Richter, which was evaluated on a real-world product family.

***Keywords*-component-based software development, software product-lines, change impact analysis, information flow**

I. Introduction

Integrated Control and Safety Systems (ICSSs) are complex, large-scale, software-intensive systems where hardware and software components are integrated to control and monitor safety-critical devices and processes. ICSSs are increasingly pervasive in domains like process plants, oil and gas production platforms, and in maritime equipment. These systems interact with their environment via physical sensors and mechanical actuators. Consequently, for deployment in concrete situations, ICSSs need to be adapted and configured to different safety logic and installation characteristics, such as sensor properties and field layout. On the other hand, there can still be a considerable similarity between different installations of an ICSS, ranging from high-level requirements to low-level implementation details (e.g. two off-shore platforms that are quite similar but not exactly identical). To leverage these commonalities and to accommodate variations as efficiently as possible, many ICSSs are developed using product-line engineering (PLE) techniques.

Component-based development is one of the main approaches to realize such software product-lines [1], and a set of shared components commonly constitutes the backbone of ICSSs. Software evolution in *families* of software products is arguably more complex as a result of the increased dependencies between software assets [2]. Shared components might be updated as a result of both family-wide domain engineering activities, and product-specific development and maintenance tasks. For large-scale systems and highly populated product families, the task of updating all products with a new version of a component comes at considerable cost.

Change Impact Analysis (CIA) plays a significant role in the software maintenance process by estimating the *ripple effect* of a change [3]. It takes a set of modified program elements (the *change set*), and computes the set of elements that need to be modified accordingly (the *impact set*) [4]. We found that the CIA methods published in scientific literature (and reviewed in the next section) were not sufficient for component-based product families. In these families, components can be implemented in various programming languages. Moreover, component composition, initialization and interconnection is typically specified in separate configuration files, ranging from simple key-value pairs to domain-specific languages. The heterogeneity of these artifacts complicates many types of system- and family-wide analysis, including change impact analysis [5, 6].

In our earlier work [7], we present a technique to reverse engineer a fine-grained system-wide dependence model from the source and configuration artifacts of a component-based system, i.e. it can be applied to a single ICSS in the product family. This paper builds on that technology and makes the following contributions: (1) we define a method for constructing a fine-grained family-wide dependence graph (FDG) from the source and configuration artifacts of a component-based product family; (2) we extend the approach of [8] to find the *initial impact set* in terms of modifications to a component's interface based on a *set of source code changes*; (3) we define a method that uses the initial impact set in combination with model-driven engineering and program slicing techniques to efficiently compute the *final impact set* across a component-based product family (i.e., to perform change impact analysis); (4) we define a measure to *approximate* the scale of the impact of a change, and use it to rank the analysis results; (5) we implemented and evaluated our approach by building a prototype tool, called Richter.

The remainder of this paper is structured as follows: Sections II and III summarize related work and describe the context of our research. We present the overall approach

978-1-4673-2312-3/12/$31.00

in Section IV, and our implementation in Section V. We evaluate our work in Section VI and conclude in Section VII.

II. Related Work

Lehnert provides an in-depth review of software change impact analysis [9]. In the context of our work, we define Change Impact Analysis (CIA) as the process of estimating what parts will be affected by a proposed change to a software system. The input is the *change set* and the output is the *impact set*. Our focus is on source based CIA. In this case, impact estimation is generally done by analyzing reachability between program elements via some form of dependencies. These dependencies can be expressed as a graph, and the ripple effects of a change can be found by traversing this *dependence graph*. CIA approaches in literature typically vary in: (a) the type of program information that is represented by the nodes and edges in the dependence graph, and (b) the type of graph traversal that is performed.

Chaumun et al. [10] propose a change impact model to investigate the influence of high-level design on the changeability of Object-Oriented programs. They use abstractions of OO entities and relations as the starting point for building their dependence graph. Sun et al. [8] propose a static CIA technique based on a predefined list of *change types* and *impact rules*. They argue that, apart from well-chosen change types and accurate dependency analyses, the precision of a CIA technique can be improved by distinguishing two stages: (1) derivation of an *Initial Impact Set (IIS)* from the change set (based on change types), and (2) propagation of that IIS through the dependence graph to find the *Final Impact Set (FIS)*. They show that a more accurate IIS results in a more precise FIS. Our approach also separates the IIS and FIS to increase precision and scalability and conducts CIA based on abstractions of the system-under-analysis. However, we need different abstractions and dependency links to represent a complete component-based product family.

Moreover, there is an implicit assumption in [8, 10] that all dependencies yield equal impacts. Consequently, they manipulate impact as coarse-grained Boolean expressions, i.e. for a given change they can only compute *whether or not* a class is impacted by that change. In contrast, our approach aims at ranking CIA results based on approximating *impact scale* (i.e., approximating the size of the affected area).

Component-based CIA: A number of studies have taken a formal approach to specify component interfaces and component composition mechanisms either to conduct CIA, or to assess the modifiability of component-based systems using CIA-inspired techniques [11–13]. In a nutshell [11, 12] specify a component, based on its set of provided/required interface functions. Each of them define their own variants of dependency relationships among components, e.g. component adjacency, (transitive) connectivity, change dependency, etc. Having defined dependency relationships in matrices, these studies take advantage of straight-forward matrix manipulation operators (e.g. production and subtraction) to conduct CIA. They focus more on *propagation* of change throughout the system, than on deriving the change set from modified artifacts. Unfortunately, they do not discuss the application of their approaches to real-world systems.

Feng and Maletic [14], conduct CIA on the *architectural level*, to estimate the ripple effects of component replacement in component-based systems. They generate component interaction traces based on the static structure of the components' interface and UML sequence diagrams. They define a short taxonomy of the atomic changes in the externally visible part of a component interface, and *impact rules* for each type of change. Finally, they derive the list of impacted elements (i.e. components, and provider/required interfaces) by slicing the generated interaction traces according to the impact rules. This work is aimed at the *inter-component*-level, whereas our goal is to analyze the impact of fine-grained code changes at the *intra-component*-level and propagate these to the *inter-component*-level and *family*-level.

Recently, there has been an increased interest in tailoring CIA to software product-lines [15, 16]. Diaz et al. [16] propose a meta-model that supports knowledge specific to product-lines (e.g. variability models), and apply traceability analysis on these models to conduct CIA on the *architectural level*. In general, these studies are aimed at exploiting state-of-the-art features of MDE and PLE such as domain-specific modeling and variability modeling. However, there are many manufacturers of software product families that have not adopted these state-of-the-art methods. We aim at devising techniques that can also support this class of practitioners.

We refer to our previous work [7] for a detailed discussion of work related to our method to build system-wide dependence models from heterogeneous source artifacts.

III. Background and Motivation

The research described in this paper is part of an ongoing industrial collaboration with Kongsberg Maritime (KM), one of the largest suppliers of programmable marine electronics worldwide. The division that we work with specializes in "high-integrity computerized solutions to automate corrective actions in unacceptable hazardous situations." It produces a large *portfolio* of highly-configurable products, such as emergency shutdown, process shutdown, and fire and gas detection systems, that will be tailored to specific deployment environments, such as vessels, off-shore platforms, and on-shore oil and gas terminals. These products are prime examples of large-scale, software-intensive, safety-critical embedded systems that interconnect software control components with physical sensors and mechanical actuators.

Terminology: We use the following terminology: a *component* is a unit of composition with well-defined interfaces and explicit context dependencies [17]; a *system* is a network of interacting components; and a *port* is an atomic part of an

interface, a point of interaction with other components or the environment. A component *instance* represents a component in a system, i.e. initialized and interconnected following the product configuration data, and a component *implementation* refers to its source code. There is one implementation and possibly more instances for every component in a system.

Production: Concrete software products are assembled in a component-based fashion and the system's overall logic is achieved by composing a network of interconnected component instances. These "processing pipelines" receive their input from sensors and decide what actuators to trigger. Components can be cascaded to handle a larger number of input signals than foreseen in their implementation. Similarly, the output of a given pipeline can be used as input for another pipeline to reuse the safety conclusions for connected areas. The dependencies from sensors and actuators are described in a decision table that is known as the cause & effect (C&E) matrix. This matrix serves an important role in discussing the desired safety requirements between the supplier and the customers and safety experts. By filling certain cells of a C&E matrix, the expert can, for example, prescribe which combination of sensors needs to be monitored to ensure safety in a given area. As installations become larger, the number of sensors and actuators grow, the safety logic becomes increasingly complex and the products end up interconnecting thousands of component instances. To give an impression, a typical real-life installation has in the order of 5000 component instances in its safety system.

Product Family: Based on workshops and interviews with safety domain experts and software engineers at KM [18], we have identified the following causes for variability in this domain: (1) functional requirements of each product category; (2) customer specific requirements; (3) size and structure of each installation, (4) different deployment configurations. To deal with this variability, our industrial collaborator adopted a component-based product development process that can be regarded product-line engineering (PLE): They maximize predictive reuse by exploiting product commonality using a set of generic and highly configurable shared components that acts as the backbone of the product family [19]. They did not adopt more formal PLE activities like variability modeling. The components are implemented in MISRA C (a safe subset of C [20]), relatively small in size (in the order of 1-2 KLOC), and contain relatively straightforward computations. Their control logic, however, can be rather complex and is highly configurable via parameters (e.g. initialization, thresholds, comparison values etc).

Evolution: There are two sources of evolution in such a product family: (1) once a new product is derived from the core components, changes are required to *adapt* the reused components to product-specific requirements (cf. [21]); and (2) it is not uncommon for product-specific components to "mature" into shared components, for instance due to an improved implementation, bug-fix, or an emerging requirement for the whole product family. In such cases, other (deployed) products of the family often need to be updated with the improved components as well. This can cause a considerable ripple effect throughout the product family. There is currently a designated *retrofit team* whose task is to take an exiting (deployed) product in the product family and update it to the latest revision of the shared components. Correctly updating the product family (and the existing deployed systems) requires a thorough understanding of the potential impact of such a change. Although a considerable amount of documentation exists for each (version of a) component to facilitate understanding, our interviews with safety domain experts and software engineers also indicated that the evolution process still depends on considerable *tacit knowledge*. It is inherently difficult to communicate this information to all developer groups; and it is vulnerable to be forgotten or lost once team members are substituted.

Research Question: The question that drives the research presented in this paper is *"Can we devise techniques to carry out fine-grained family-wide change impact analysis on the source and configuration artifacts of a component-based product family."* The goal is to provide our industrial partner with (prototype) tools to support the component evolution and retrofitting activities on their product portfolio.

IV. Approach

In the remainder, we use *system-level input* and *sensor* interchangeably, and likewise for *system-level output* and *actuator*. We discuss our approach in terms of the studied product family but emphasize that it is also applicable with other inputs and outputs than sensors and actuators.

As discussed in Section II, CIA techniques are characterized by the type of program information that is represented by the nodes and edges in the dependence graph, and the type of graph traversal that is performed. Considering the significance of connections between sensors and actuators in the domain of our case study, we select the information flow from sensors to actuators as backbone of our CIA technique.

Tracking Information Flow: In a previous study [7], we proposed a method for tracking the information flow between sensors and actuators using program slicing [22]. In that work, we also addressed the challenge of constructing a system-wide dependence graph of *a single* component-based system which was successfully used for system-wide program slices. It could be argued that having the necessary tooling to compute system-wide slices in component-based systems makes product-line-specific CIA obsolete. By replicating the method in [7], one could (1) build a separate SDG for each product in the product family, (2) compute a straightforward system-wide slice with each actuator as the slice criterion (to find all the program points with a potential affect on that actuator), (3) and take the intersection of each slice with the change set and report those with a non-null intersection as being *impacted*. However, this straightforward

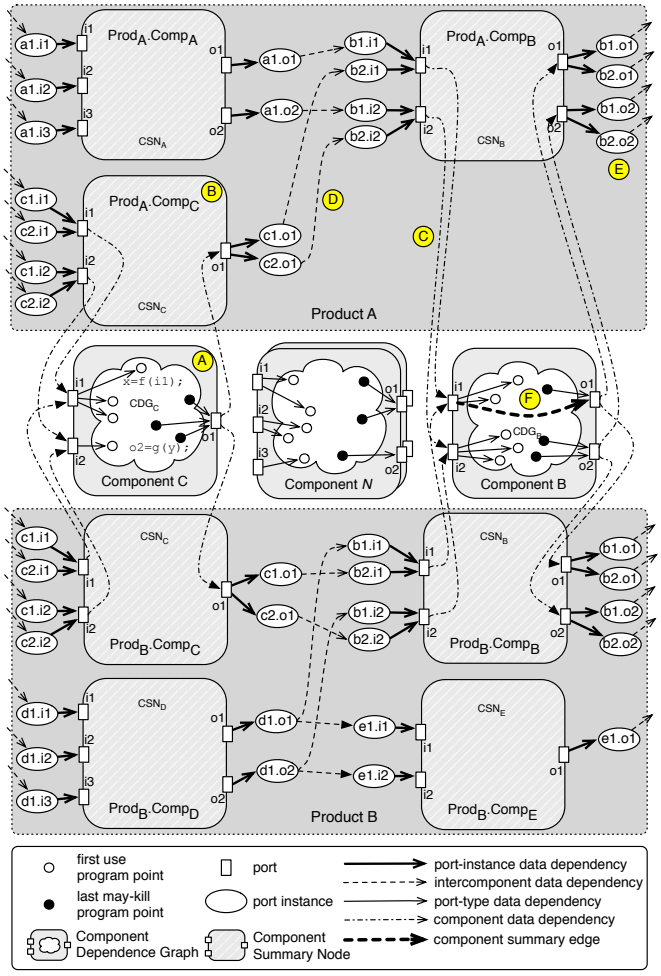

Figure 1. Family Dependence Graph (tags A–F are explained in the text)

approach would certainly not scale up to the hundreds of product installations in our product family and hundreds of actuators in each product. Not capturing the commonality and variability would lead to a combinatorial explosion of alternatives to analyze, especially since components can be included multiple times in a product (e.g. voter components).

Family Dependence Graphs – We build on our previous technique to construct a homogeneous *family-wide* dependence model that can represent all members of a large-scale component-based product family. Apart from scalability, the target dependence model should be amenable to CIA, with comparable precision to fine-grained program slicing. The overall approach to construct this dependence model, which we call the Family Dependence Graph (FDG), is as follows:

1) For each component in the system, we build a *component dependence graph (CDG)* by following the method for constructing inter-procedural dependence graphs [23] and taking the component source code as "system source". This CDG contains the fine-grained program points and data- and control-dependencies from the component's implementation (Figure 1, tag A).
2) To efficiently represent components in members of the product family, we define the notion of a *Component Summary Node* (CSN). A CSN is a projection of a component's CDG from the perspective of its *externally visible* interface, i.e. *without* the fine-grained dependence graph. There's a separate CSN for a given component, and for each product containing an instance of that component (Figure 1, tag B).
3) To link a CDG and its counterpart CSN in a product, dependencies are added from each input port of a CSN to the corresponding input port of the CDG (Figure 1, tag C), and from each output port of that CDG to the corresponding output port of the CSN. This makes the CDG appear "in-line" with its product-specific CSN.
4) For each product, the configuration artifacts are analyzed to build a product-specific *inter-component dependence graph (ICDG)*. This graph captures the *network* of interconnected component instances via their externally visible interfaces. Construction of the ICDG is done in the same way as the component composition framework uses to set up the correct network.
5) The *product system-wide dependence graph (PSDG)* is constructed by integrating the product's ICDG with the CSNs of the components (Figure 1, tag D). Conceptually, the construction can be seen as taking the ICDG and substituting each "component instance node" with the CSN for the given component. As in [7], we use structured IDs for port-instances (Figure 1, tag E) to preserve context during slicing.

The union of PSDGs for all members of the product family forms the FDG, where products are interconnected via their shared components. This homogeneous model of the product family can be sliced using standard graph reachability algorithms with one minor adaptation to preserve the calling context of components: whenever a component CDG is entered via a port-instance, we save the instance ID, and when exiting that component, we only continue slicing on connections that match the saved instance ID. This is analogous to how procedure calls are handled in [23].

To avoid repeating expensive slicing in later stages of our impact analysis, we *enrich* our CDGs with *Component Summary Edges* (CSEs) that capture component-wide dependencies between component input and output ports. CSEs show which input ports can affect which output ports, but abstract away all the details on *how* the information flow is realized. For each component C, we enrich the respective CDG_C by slicing all output ports and including the summary edge $\mathrm{O}_n \rightarrow \mathrm{I}_m$ if input port I_m is included in the slice for O_n (Figure 1, tag F). The size of the slice, i.e. the number of the program points included in that slice, is added as a property to this summary edge. Note that, CSEs can be (and in our case are) one-to-many relations, i.e. more than one

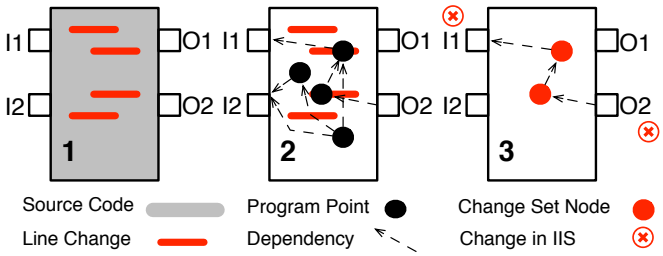

Figure 2. Detecting the CS and IIS

component input port can affect a single output port. In such cases we use an *aggregate summary edge*, which connects a given output port to all of the affecting input ports, and the slice size becomes an attribute of this aggregate edge.

We develop our CIA approach based on the above-mentioned FDG in such a way that it leverages the fine-grained information inside the CDGs and balances them with the coarse-grained CSNs and product-specific ICDGs to trade-off between precision and scalability. The steps are as the following: (1) Detect the Change Set: what has been modified? (2) Find the Initial Impact Set: what has changed from the external interface of a modified component? (3) Find the Final Impact Set: what products, and which sections of those products, will be impacted?

Detect the Change Set (CS): We focus on syntactic changes as no static analysis method can guarantee to infer the semantic differences between two versions of a program. The process retrieves the syntactic differences of two consecutive revisions of a given component using a source-differencing tool available in most of the mainstream software revision control systems (Figure 2, case 1). Using a pure text-based tool, such as SVN "diff", has the benefit that no syntactic change will go undetected. However as a down-side, ineffective trivial modifications to the source code are also retrieved (e.g. adding comments). Such modifications are obviously irrelevant with respect to CIA, and we remove them by comparing the retrieved modifications against the CDG to filter out the *pseudo* changes, i.e. changes in the source code that do not have a counterpart in the component's CDG (Figure 2, case 2). Hereinafter, a *CS node* stands for a program point in the CDG of a component whose counterpart source code has changed.

Find the Initial Impact Set (IIS): To estimate the ripple effects of a modification in a component-based system, we first need to detect the consequences of the source-changes from the perspective of the component's interface — hereinafter called the the Initial Impact Set (IIS). By component interface in this product family, we mean the set of input and output ports of the component. As the IIS will later seed the process of propagating ripple effects throughout the product family, the accuracy of the IIS will have a great impact on the final precision of our CIA. Therefore, considering the safety-critical characteristic of our case study, we do not intend to trade-off precision for scalability at the IIS level.

To this end, we slice through the fine-grained CDG of the *updated* component, with each output port as the slicing criterion. This step tracks the new *intra*-component information flows, and at the same time, extracts the new set of CSEs and their slice sizes. The following two cases will be included in the IIS on the *original* component:

1) Any output port whose program slice has a non-null intersection with the CS nodes (Figure 2, case 3).
2) Any difference between the new enriched CDG and the previous one (with respect to component interface ports, summary edges, and their slice size).

At a first sight, it might appear that the second item in the above list makes the first item obsolete as all cases of the first item are also included in the second one. However, a more detailed look on the cases entails their differences in the context of CIA, as exemplified in the following: Consider the case of two component revisions (with input port I_1 and output port O_1), whose *only* difference is changing the program statement: "$O_1 = I_1 + 1$;" to "$O_1 = I_1 + 1000$;". This source modification does not yield any difference in the structure of the enriched CDG, nor does it change the slice size of O_1 between the two versions. Therefore it will not be caught in the second item, while it will be caught by the first one as the program point(s) that represent the modified source line will result in a non-null intersection with the backwards slice from O_1. As a result, the first item is needed for the purpose of a "safe" analysis. Likewise, the second item is not a subset of the first item since the modifications in the new version of the component, might occur *next to* and *disjoint to* the previous code. (e.g. adding a new pair of input and output ports to a component and not changing code which belongs to previously-existing ports).

We emphasize that one component-wide slice is performed for each port of a given component *type* (approximately 10-30 ports for each component), and not for each component *instance* (up to thousands of instances of each component). Therefore, the scalability of our approach will not depend on the size of the deployed systems or the number of component instances, but on the number of component implementations and the number of ports belonging to each component.

Find the Final Impact Set (FIS): Before discussing the propagation rules, we remark that change requests that alter the overall black-box behavior of a product (e.g. adding a new actuator), are not considered in our CIA approach. Such modifications are represented by a new cause & effect matrix (Section III) by adding/removing sensors or actuators, and definitely call for maintenance effort (e.g. testing and certification). However, it is unlikely that the expected behavior of all sensors/actuators are altered in an existing product. It is in such cases where CIA can be especially valuable as it can limit the maintenance effort to only the new sensor/actuators, should the previously existing behavior of the system be reckoned as *intact* by CIA.

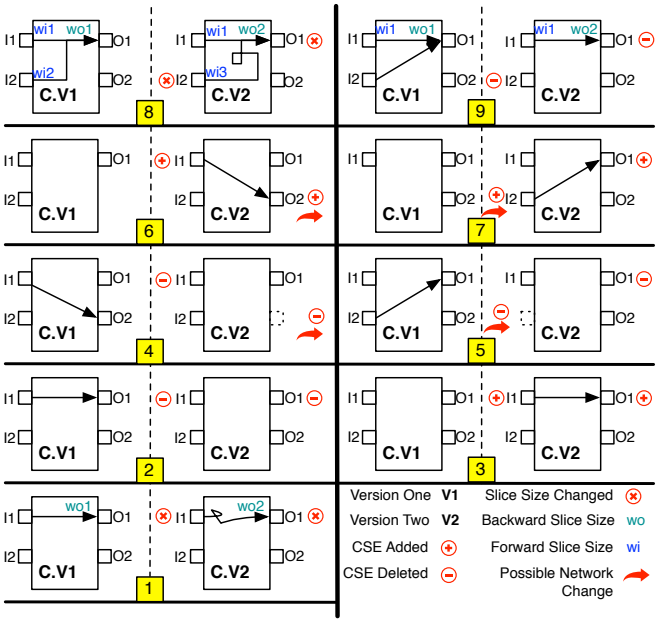

Figure 3. IIS cases and propagation of ripple effects

Similar to most existing CIA techniques, we propagate the *prospective* ripple effects of the detected IIS throughout all products of our product family, by forward and backward traversal of our intermediate system representation, i.e. the FDG in our case. We call this impact set the Final Impact Set (FIS), which contains all sensor-actuator pairs whose information flows have been affected by the source modification.

To discuss different change propagation scenarios, we distinguish several *atomic* IIS cases, graphically shown in as pre-change (V1) and post-change (V2) versions in Figure 3:

1) Only the slice size of a CSE is changed (Figure 3, #1).
2) A CSE between a pair of component input and output is added or removed (Figure 3, #2-3).
3) A component input or output port is added or removed (Figure 3, #4-7).

Figure 3, #8-9 show refinements discussed later.

Realistic IIS cases can, and usually do, contain more than one of the above-mentioned items at the same time. However, we separate such compound cases into the atomic cases to conduct impact analysis and aggregate the results afterwards.

Traversal of the FDG is done by a straightforward reachability analysis on coarse-grained summary dependencies (CSEs) and inter-component dependence graph (ICDG). For clarity, we discuss the highlights with respect to two versions of a hypothetical component (C.V1 and C.V2 in Figure 3).

Case 1 (Figure 3, #1) – Fine-grained slice sizes change and coarse-grained dependencies are the same: we only need to traverse the existing FGD (forwards and backwards), and mark all reaching sensors and actuators as FIS. We take the slice size as an *approximation* of impact, and rank the FIS according to the (absolute) delta in the system-wide slice sizes. In this scheme, a system-wide information flow whose size is changed by V is considered equal to an information flow whose size has changed by $-V$ (with respect to impact), and an information flow with size S_1 is ranked higher than one with size S_2, provided that $S_1 > S_2$.

Case 2 (Figure 3, #2-3) – Intra-component information flows are added/removed but the externally visible interface remains the same: system-wide information flows might change as a result of the changes and may yield different dependence relations between sensor-actuator pairs. Given that multiple instances of a component might participate in a single system-wide information flow, it is not enough to propagate the IIS only in the previous FDG as it only contains the previous CSEs. In this case we need to conduct *two* rounds of propagation: once with the *previous* enriched CDG, and once with the *new* CDG of the component in the FDG. The FIS will be the *union* of the results from the two propagations, as we are interested in every change in the externally-visible behavior of the system (e.g. an actuator engaged due to a sensor that was previously ineffective, and vice versa). Any differences in system-wide information flows will be marked as cases with definite impact. The remaining information-flows will be ranked as in case 1.

Case 3 (Figure 3, #4-7) – Externally-visible interface of the components are changed: we should conduct the propagation in two phases, similar to Case 2. However, in case the inter-component connections (the ICDG in our model) have also adapted to the component's new interface, we should conduct the second round of propagation with an extra twist: with the new ICDG in place. We point out that addition/removal of a component port does not necessarily imply that the ICDG of the product will change, as it is not uncommon to replace a component with a revision which has an extended interface, while the product only uses the previously existing interface. For instance, consider the case of a revised component who contains a number of bug-fixes and also introduces more optional capabilities into the product family. In such cases we do not need to conduct the second round of propagations with the new ICDG.

Refinement (Figure 3, #8-9) – Enriching the FDG with forward component-wide slice sizes can improve the accuracy of change propagation in a number of cases: Figure 3, #8, shows O_1 being affected by I_1 and I_2 in version V1. This could for example be the results of the following two program statements: "$O_1 = I_1 + 1$;" and "$O_1 = I_2 + 2$;". In V2, we change the latter to "$O_1 = (I_2 > 0)\ ?\ 0 : 1$;", and the size of the backward slice from O_1 changes from Wo_1 to Wo_2. This will put O_1, I_1, and I_2 in the IIS. However, considering that forward slices from I_1 in the two versions have not changed, we can conclude that the modifications have only affected the information flow between O_1 and I_2, and not I_1. Therefore, we trim the IIS accordingly and avoid propagating the changes from I_1 as they would be false-positives. Figure 3, #9, shows a more involved case of #8 in which the information flow between O_1 and I_2 have been

completely cut off. Likewise, by considering forward slice sizes we are able reduce false positives.

Distinguishing atomic parts makes it easier to analyze complex cases, but it does not affect the FIS, which is computed by taking the union of the atomic cases. However, some caution is needed for ranking based on slice size: when two atomic changes affect the same system-wide information flow respectively by size V1 and V2, we rank this flow once with size $|V1|+|V2|$ to avoid reporting that flow more than once with different impact scales.

V. Prototype Implementation

This section discusses the implementation of our approach in a tool named Richter. In [7] we developed tooling to reverse engineer system-wide dependence graphs from source artifacts of *a single* component-based system. Richter reuses parts of this work to develop a homogeneous model from source artifacts of a family of products, which will be reported briefly in this section.

To enable flexible integration of individual models to build our FDG, we use OMGs Knowledge Discovery Meta-model (KDM) [24] as a foundation for representing the various intra- and inter-component dependence graphs. KDM was designed as a wide-spectrum intermediate representation for describing existing software systems and their operating environments. KDM is completely language- and platform independent, making it an ideal match for the purpose of modeling product families with heterogeneous artifacts.

We use Grammatech's CodeSurfer [25][1] to create component dependence graphs (CDGs) for the individual components. The top portion of Figure 4 gives an overview of the main information that we collect from component implementations to build the CDGs. Next, these CDGs are traversed using CodeSurfer's API to inject them into KDM. We refer to [7] for more details on the mapping between program elements and KDM classes. The traversal uses the Java Native Interface to drive KDM constructors in the Eclipse Modeling Framework (EMF). For each program point, we include a pointer to its *origin* in the source code for traceability. We enrich the CDGs with additional *summary edges* using a simple slicing tool in Java that we have created as part of our earlier work. Alternatively, we could have defined several "destructive" transformations that create an additional new model for each CDG, but we prefer to enrich our dependence model in order to reuse information in multiple applications. To avoid keeping the whole FGD in memory, we exploit EMF notions of Resource and ResourceSet to compartment our model and to serialize each compartment separately [26]. For each (version of) a component implementation we need to build its fine-grained CDG once, and save this model into a separate EMF Resource. By activating the optional lazy-loading mechanism

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

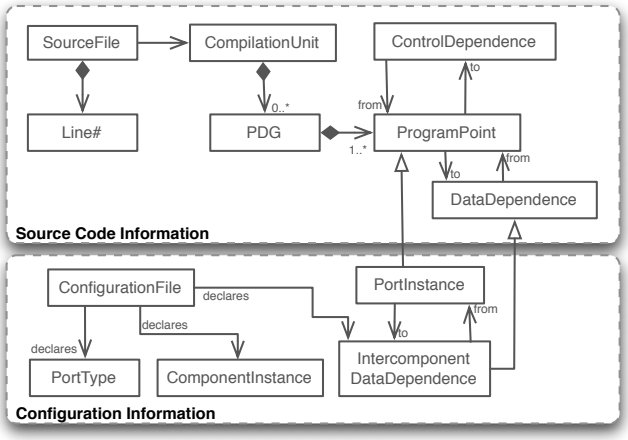

Figure 4. Meta-model describing the main elements used to track information flow across a component-based system.

of EMF Resources, once a CDG is required to build a PSDG for the first time, it will be automatically loaded into memory.

Next, we use Xalan-J to analyze and transform the system configuration artifacts of each product (lower portion in Figure 4) into its inter-component dependence graph (ICDG). Finally, we use KDM *container* elements to add the component summary nodes (CSNs) for each component that is included in a product, and use a straightforward substitution transformation to integrate the CSNs with the ICDG and create the final PSDG. The edges between CDG and CSN interface ports are easily mimicked by adding *stereotyped* ActionRelationShips (a KDM wild-card meta-model class to be extended by new meta-model classes).

Our prototype calls SVN "diff" to detect syntactic changes in the source code, but similar tools can be used instead. With the FDG constructed, we propagate changes throughout the product family by slightly adapting our straightforward Java slicing tool to traverse coarse-grained summary edges and accumulating slice sizes along the way.

VI. Evaluation

In the context of large-scale safety-critical systems, two important factors for evaluating our approach are its accuracy and scalability. Since our CIA technique largely depends on the quality of the family-wide dependence model that is used as the medium to both detect changes, and to propagate ripple effects throughout the product family, we focus on evaluating the accuracy and the scalability of our FDG.

Accuracy: One of the challenges in evaluating the accuracy of our cross-component approach is determining a *gold standard* to compare our results to. This is due to the fact that existing program analysis tools are typically confined to the boundaries defined by the source code of a single component as they can neither incorporate the component configuration information, nor heterogeneous programming languages.

We address this challenge in the same way as we did in [7] by increasing our level of control during the ex-

perimental evaluation. In short, we create two code bases and compare the results of applying (a) our approach, and (b) an existing reliable tool on these code bases: First, we develop two simple in-house component based products that closely resemble the architecture of the products described in Section III. To simulate a product family, these two products have one shared component. Each product mimics the component composition framework of our real-world case study by processing a number of external configuration files and building the inter-component network. Port declarations, component instantiations, and all component interconnections are described using text-based configuration files. The connection mechanism is simple, yet general enough to represent most component-based systems, including our case study. Second, we create another product family which contains the same ingredients as the first one, but everything is implemented as a homogeneous program. This is done by replacing the component composition framework by *hard-coded* connections in the program's source code.

The components and configuration artifacts of the first product family are analyzed using our slicer. Moreover, since the second product family does not depend on external configuration files and since all aspects are programmed in C, it can be analyzed by CodeSurfer to set the gold standard in our evaluation. We evaluate the accuracy by comparing the slices obtained using our tool with the gold standard computed by CodeSurfer, and looking for any differences in the program points, component instances, and port instances that are included in a slice. To maximize the fault-revealing potential and test both system-wide and partial information flow paths, we repeat this comparison for each system and component output port as the slicing criteria.

Our comparisons show that for each configuration and slicing criterion, both slicing tools generate the same output for what concerns the components and their interactions. The slices computed by CodeSurfer also contained the code that was added to the variants to hard-code the component connections. Since our approach abstracts from the framework and directly captures the configuration, those program points have no counterpart in our slices, as was expected. We conclude that we achieve 100% accuracy.

Scalability: We discuss the scalability of our dependence model and fine-grained system-wide slicing in reference to the evaluation in [7] in recognition of the continuity in our industrial collaboration. Afterwards, we discuss the effects of coarse-grained dependencies on the FDG and system-wide dependency analysis, which are specific to our current study.

As mentioned earlier, the System Dependence Graph (SDG) introduce in [7] provided a fine-grained dependency model for a single component-based system. We have developed our FDG based on the same principles and terminology as in [7], but tailored the dependence model with respect to shared vs. product-specific components. However, product-specific components can be regard as a special product-line asset which has been used in only *one* system so far. According to [21], product-specific components can, and in reality do, mature into core components once they enrich their variabilities. In conclusion, if we build our FDG for a hypothetical product whose components are all specific to that product, the FDG becomes identical to our previous SDG. Therefore, we developed our prototype tool (Richter) by reusing our previous implementation reported in detail in [7]. In that paper we demonstrated in detail that both execution time and model size show linear growth with respect to program size (measured by LOC). The growth rate was shown to be constant from a number of industrial code bases ranging from about 100LOC to 100KLOC in size. To give an impression of the resulting model size, we report that the (KDM) model for the mentioned system with 100KLOC is transformed into 600,000 lines of XMI (78MB), once serialized on disk.

To efficiently represent components inside PSDGs, we substitute fine-grained CDGs with CSNs which only contain the externally-visible interface of a component. To implement this scheme we need one node in the CSN for each component port, and one edge (ActionRelationShip in KDM) between the port nodes. Apart from that, we enrich each CDG with coarse-grained CSEs which summarize component-wide information flows by using a single edge for each component input-output pair that is connected by program slicing. These two design choices make our system-wide dependency analysis completely independent from the components' source code size once the FDG is built. The efficiency of our dependency analysis is a linear function of the number of *component instances* that participate in each system-wide information flow, which is approximately in the range of 12-20 in the product family we study. To traverse across each competent instance we need to walk five edges: two edges between port instances and port types in the CSN, two for the edges between a port in CSN and its counterpart node in CDG, and one for the summary edge inside the CDG. Traversing such low number of dependencies in our model takes a trivial time, in the order of milliseconds.

We would like to demonstrate the effectiveness of the coarse-grained dependencies (i.e. CSNs and CSE) with respect to the graph size. Table I reports graph size in four randomly selected components from a subset of our industrial partner's software repository which was accessible to us to evaluate our approach. The first row of the table shows the number of CDG nodes and edges, corresponding to program

Table I
GRAPH SIZE: FINE-GRAINED VS. COARSE-GRAINED

	Component	1	2	3	4
Fine-Grained	Node #	3010	1864	2518	1592
	Dependency#	10386	5915	8702	5220
Coarse-Grained	Node #	23	13	23	21
	Dependency#	44	26	50	50

points and data- and control dependencies in the component source code. The second row shows the number of CSN nodes and CSEs, corresponding to component ports (input and output) and pairs of input-outputs that are connected together by information flow. As a reference for comparison, the component dependence graph for "Component 1" has 3010 program points and 10386 dependencies. Its corresponding coarse-grained graph has only 23 nodes (for each product that has an instance of "Component 1"), and 44 edges (each one connecting input to output directly).

Validity: The above-mentioned evaluation covers the accuracy and scalability of the FDG and the underlying program analysis technique, i.e. slicing. Although they are an important determinant in the efficiency of our CIA approach, we acknowledge that a thorough application of our CIA approach is needed before we can assess its reliability. First and foremost, the precision and recall factors of our CIA needs to be demonstrated in practice using the software repository of our industry partner. The mentioned repository contains the actual evolution history of the product family for almost two decades. We can put to test our CIA approach by choosing a component revision whose actual ripple effects are known in the repository, and compare our FIS against that. Likewise, the intuition to associate the scale (severity) of impact with program slice sizes, which was the basis of our approximate ranking scheme, should be empirically put to test before it can be adopted as a reliable measure. Both of the mentioned tasks require long-term collaborations with our industry partner, to closely monitor the applicability of our approach in day-to-day maintenance tasks in the course of time. This process, in return, requires integrating our approach in the existing development environments of our industry partner. We are currently planning and discussing a number of prospective research avenues with our industry partner to accomplish the mentioned goals.

Discussion: As described in Section IV, a single CDG is built for every component in the product family, regardless of being a shared or being a product-specific asset. Therefore, all PSDGs are built by using a much more lightweight Component Summary Node (CSN). Alternatively, we could build an equally-expressive model without building separate CDGs and CSNs for product-specific components. As such components appear only *once* in the product family, we could embed the original CDGs inside the PSDGs. One could argue that having separate CSNs for product specific components imposes extra nodes into the model, albeit only a handful of nodes. We believe such (low) overheads in model space are negligible given that we get a highly regular, and much simpler, modeling of the domain in return. Also from a technical point of view, having the heavy and fine-grained CDGs in one model compartment (together with the lazy-binding mechanism mentioned in Section V), makes PSDGs extremely lightweight. This makes (potentially frequent) executions of CIA even more cost-effective.

VII. Concluding Remarks

Integrated Control and Safety Systems (ICSSs) are complex, large-scale, software-intensive systems to control and monitor safety-critical devices and processes that are increasingly pervasive in technical industry, such as oil and gas production platforms, and process plants. These systems are highly-configurable and for deployment in concrete situations they need to be adapted and configured to different safety logic and installation characteristics. Component-based development of product families is one of the main approaches to cope with such a high variability space while controlling quality, cost and time to market by maximizing the reuse of components between products.

However, software evolution in such products families is arguably more complex as a result of the increased dependencies that are introduced via shared components. Change Impact Analysis (CIA) can play a significant role in this process by estimating the *ripple effect* of a change, but the heterogeneity of software artifacts hinders a uniform analysis in product families.

Contributions: This paper proposes a technique for Change Impact Analysis in component-based product families using a combination of Model-Driven Engineering with well-established program analysis techniques, such as program slicing. The contributions of this paper are the following: (1) we recover a Family Dependence Graph (FDG) which balances the trade-off with precision and scalability, for the purpose of change impact analysis; (2) we improve the precision of change propagation by detecting the Initial Impact Set (IIS) using fine-grained dependence graphs; (3) we compute the Final Impact Set (FIS) by propagating the IIS throughout a family of products via traversal of lightweight and coarse-grained dependencies — *this choice of where to draw the line between IIS and FIS, and move from fine-grained to coarse-grained dependencies, is the key decision to balancing precision and efficiency in our approach* — (4) we propose a ranking scheme based on *approximations* of the scale of impact using program slice sizes; (5) we present the transformations that helped us to achieve these models, and discuss how we developed a prototype tool (named Richter) based on a standardized language-independent metamodel (KDM) to ensure interoperability and generalizability. The proposed approach is not limited to the proposed domain and can be applied on systems with inputs and outputs other than sensors and actuators. The evaluation indicates that it scales well to the constraints of real-world product families.

Future Work: We see several directions for future work: First, as mentioned in Section VI, we intend to empirically assess our approach to evaluate the precision and recall factors of our analysis in an industrial context. In addition, our approximation of impact scale based on program slice sizes, needs to be validated by closely monitoring how our approach is used in practice, and by gathering feedback from

the so-called *retrofit team* (Section III). We also intend to try out the effect of different weighting schemes on our ranking mechanism, based on the *type* of the program points involved in the slice. For instance, we can assign a larger weight for a node in a condition clause than a node in an assignment statement, assuming that a change in a condition clause should take priority to another change with the same size with no condition clause. These studies require long-term close collaborations with our industry partner.

The externally-visible interface of the components in our case study, is highly "data oriented", i.e. components interact by sending and receiving data to/from each other. This characteristic makes them very amenable to information flow analysis, which is the foundation for our impact analysis. One line of future work is to investigate the application of our approach in component-based system whose interaction is via API calls. One main difference of such systems with our case study is that component interactions follows a "call-and-return" scheme. The effect of such interaction schemes on the homogeneous dependence model needs to be studied.

Apart from CIA techniques, another approach for estimating the effects of software change is investigating how a given system has evolved in the past [27]. Several studies have reported on cases that uncover co-evolution trends among software artifacts, by applying data-mining techniques on the previous versions of the artifacts and other related historical data (e.g. bug reports and the meta-data in version control software) [28]. There is an emerging trend to integrate the two approaches to increase the precision of software evolution estimations [27, 29]. It would be interesting to investigate how our CIA-based estimations can be enhanced using the historical evolution information from our industrial partner's software repository.

Acknowledgments: We thank the safety experts and software engineers from Kongsberg Maritime that participated in our workshop and interviews for their time and feedback.

REFERENCES

[1] M. Matinlassi, "Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA," in *Int'l Conf. Softw. Eng.* IEEE, 2004.

[2] M. Svahnberg and J. Bosch, "Evolution in software product lines: two cases," *J. Software Maintenance: Research and Practice*, vol. 11, no. 6, 1999.

[3] S. Bohner and R. Arnold, *Software Change Impact Analysis*. IEEE, 1996.

[4] S. Lehnert, "A taxonomy for software change impact analysis," in *Int'l Ws. Principles of Softw. Evolution (IWPSE-EVOL)*. ACM, 2011.

[5] M. J. Harrold, D. Liang, and S. Sinha, "An Approach To Analyzing and Testing Component-Based Systems," in *ICSE Ws. Testing Distributed Component-Based Systems*, 1999.

[6] A. Rountev, "Component-Level Dataflow Analysis," in *Int'l Conf. Component-Based Softw. Eng. (CBSE)*. Springer, 2005.

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

[8] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change Impact Analysis Based on a Taxonomy of Change Types," in *Computer Softw. and Applications Conf.* IEEE, 2010.

[9] S. Lehnert, "A Review of Software Change Impact Analysis," Techn. Univ. Ilmenau, Report ilm1-2011200618, 2011.

[10] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman, "A change impact model for changeability assessment in object-oriented software systems," in *European Conf. Softw. Maintenance and ReEng.* IEEE, 1999.

[11] Z.-j. Wang, X.-f. Xu, and D.-c. Zhan, "Agility Evaluation for Component-based Software Systems," *J. Information Science And Engineering*, vol. 23, no. 6, 2007.

[12] L. Yan and X. Li, "An Interface Matrix Based Detecting Method for the Change of Component," in *Int'l Symp. Information Science and Eng.* IEEE, 2008.

[13] C. Mao, J. Zhang, and Y. Lu, "Matrix-based Change Impact Analysis for Component-based Software," in *Computer Softw. and Applications Conf.* IEEE, 2007.

[14] T. Feng and J. I. Maletic, "Applying Dynamic Change Impact Analysis in Component-based Architecture Design," in *Int'l Conf. Softw. Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. IEEE, 2006.

[15] H. Cho, Y. Cai, S. Wong, and T. Xie, "Model-Driven Impact Analysis of Software Product Lines," in *Model-Driven Domain Analysis and Softw. Development: Architecture and Functions*. IGI, 2011.

[16] J. Díaz, J. Pérez, J. Garbajosa, and A. L. Wolf, "Change impact analysis in product-line architectures," in *European Conf. Softw. Architecture*, 2011.

[17] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. AW, 2002.

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

[19] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. AW, 2000.

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

[21] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *J. Systems and Software*, vol. 74, no. 2, 2005.

[22] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, 1982.

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

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

[25] P. Anderson, "90% Perspiration: Engineering Static Analysis Techniques for Industrial Applications," in *IEEE Int'l Working Conf. Source Code Analysis and Manipulation*, 2008.

[26] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. AW, 2009.

[27] H. Kagdi and J. Maletic, "Software-Change Prediction: Estimated+Actual," in *IEEE Int'l Ws. Softw. Evolvability*, 2006.

[28] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, 2007.

[29] L. Hattori, S. Jr, F. Cardoso, and M. Sampaio, "Mining Software Repositories for Software Change Impact Analysis : A Case Study," in *Brazilian Symp. Databases (SBBD)*, 2008.