Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Towards evidence-based recommendations to guide the evolution of component-based product families

Leon Moonen

Simula Research Laboratory, Oslo, Norway

HIGHLIGHTS

- The sharing of assets in component-based product families complicates evolution.
- Precise change impact analysis across the complete product family is needed.
- The heterogeneity of artifacts in these families hinders fine-grained analysis.
- We devise recommendation technology that supports accountable software evolution.
- We discuss the research challenges that need to be overcome to build such tools.

ARTICLE INFO

Article history:
Received 25 September 2013
Accepted 5 November 2013
Available online xxxx

Keywords:
Program comprehension
Reverse engineering
Change impact analysis

ABSTRACT

Many large-scale software-intensive systems are produced as instances of component-based product families, a well-known tactic to develop a portfolio of software products based on a collection of shared assets. However, sharing components between software products introduces dependencies that complicate maintenance and evolution: changes made in a component to address an issue in one product may have undesirable effects on other products in which the same component is used. Therefore, developers not only need to understand how a proposed change will impact the component and product at hand; they also need to understand how it affects the whole product family, including systems that are already deployed. Given that these systems contain thousands of components, it is no surprise that it is hard to reason about the impact of a change on a single product, let alone assess the effects of more complex evolution scenarios on a complete product family. Conventional impact analysis techniques do not suffice for large-scale software-intensive systems and highly populated product families, and software engineers need better support to conduct these tasks. Finally, for an accountable comparison of alternative evolution scenarios, a measure is needed to quantify the scale of impact for each strategy. This is especially important in our context of safety-critical systems since these need to undergo (costly) re-certification after a change. Cost-effective recommendations should prioritize evolution scenarios that minimize impact scale, and thereby minimize re-certification efforts.

This paper explores how reverse engineering and program comprehension techniques can be used to develop novel recommendation technology that uses concrete evidence gathered from software artifacts to support engineers with the evolution of families of complex, safety-critical, software-intensive systems. We give an overview of the state of the art in this area, discuss some of the research directions that have been considered up to now and, identify challenges, and pose a number of research questions to advance the state of the art.

© 2013 Elsevier B.V. All rights reserved.

E-mail address: leon.moonen@computer.org.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.11.009

Please cite this article in press as: L. Moonen, Towards evidence-based recommendations to guide the evolution of component-based product families, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.11.009

The search for techniques that transcend language boundaries has been a recurring theme in Paul's career. It entered my world when Paul challenged me to investigate solutions for generic data flow analysis as a master student. This paper further explores this theme, driven by the need to better manage the evolution of multi-language software systems. Understanding the interrelations that may interfere with changing such systems requires cross-language analysis. True to form, we use a unified meta-model to represent all relevant information, and build language-independent analyses on top of this model. The application to change impact analysis even remarkably close to the data flow analysis that started all this for me. Paul, I would like to thank you for the inspiration and challenges that led me on this journey.

1. Introduction

Integrated Control and Safety Systems (ICSSs) are complex, large-scale, software-intensive systems to monitor and control safety-critical devices and processes in domains such as process plants, oil and gas production, and maritime equipment. These cyber-physical systems integrate hardware and software components, and they are typically designed as a network of interconnected elements that interact with their environment via physical sensors and mechanical actuators. Consequently, for deployment in concrete situations such ICSSs need to be adapted and configured to the particular safety logic and installation characteristics, such as number and type of sensors and actuators, and the layout and dependencies of safety areas. At the same time there can be considerable similarities between different ICSSs, ranging from high-level requirements to low-level implementation details, in particular for systems in the same domain. For example, consider two off-shore platforms that are alike but not identical in terms of layout or equipment, a very similar observation can be made for their ICSSs. To leverage commonality while accommodating for variations as efficiently as possible, many ICSSs are developed as component-based product families, a well-known tactic to develop a portfolio of software products based on shared assets [1–3].

However, sharing components between software products introduces dependencies that complicate maintenance and evolution of the system because the changes made to address an issue in one product may have undesirable effects on another product in which the changed component is used [4]. Changes arise not only from product-specific improvements, they also originate from rethinking the product family as a whole (known as domain engineering). Shared components can be updated as a result of both *"family-level"* domain engineering activities, as well as product-specific (*"system-level"*) development and maintenance.

For large-scale systems and highly populated product families, a change typically does not come by itself: to achieve a certain effect, developers need to evaluate alternatives for (a) the actual change(s) to be made, (b) additional modifications to address undesired ripple effects of that change, as well as (c) the way in which changes are applied. We refer to these alternatives as evolution scenarios. To choose between alternative evolution scenarios, developers not only need to understand how a change will impact a component and the product in which it is used; they also need to reason how the change will affect the whole product family, including the systems that are already deployed. Given that ICSSs consist of thousands of components, it will be no surprise that it is hard to reason about the effect of a change on a single product, let alone on a complete family of products. Yet, the engineers have to ensure that the components work seamlessly and flawlessly together, also after the proposed changes were made. Despite impressive technological advances, the existing approaches to developing and maintaining software are stretched to the max due to increasing demands, complexity, and scale. Consequently, software is often built and managed based on decisions for which we have insufficient evidence to confirm their suitability, quality, costs, and inherent risks.

Accountable software evolution decisions can be based only on concrete evidence gathered from the actual source artifacts [5]. Change Impact Analysis (CIA) can play a significant role in this process by estimating the ripple effect of a change [6]. CIA takes a set of modifications (the change set), and computes what parts of the program are affected by that change (the impact set) [7]. Source-based CIA estimates the impact of a change by tracking dependencies between program elements. The ripple effects can be found using reachability analysis on this dependence graph. We found that the CIA methods published in scientific literature (and reviewed in the next section) were insufficient for large-scale software-intensive systems and highly populated product families. Challenges include the fact that components can be implemented in various programming languages, and that component composition, initialization and interconnection are typically specified in separate configuration files, ranging from simple key-value pairs to domain-specific languages. This type of artifact heterogeneity complicates many types of system- and family-wide analysis, including change impact analysis [8,9]. Moreover, the conventional methods are aimed at analyzing single systems, and do not take dependencies in a product family into account. A final concern in the context of ICSSs is that the safety-critical nature of these systems requires that they are re-certified after a change. Therefore, cost-effective recommendations should prioritize evolution scenarios that minimize impact scale, and thereby minimize re-certification efforts.

This paper explores how reverse engineering and program comprehension techniques can be used to develop novel recommendation technology that uses concrete evidence gathered from software artifacts to support engineers with the evolution of families of complex, safety-critical, software-intensive systems. Contributions of this paper include an overview of the state of the art in this area and a discussion some of the research directions that have been explored up to now. Next, we identify challenges, and pose a number of research questions that need to be answered to advance the state of the art.

Please cite this article in press as: L. Moonen, Towards evidence-based recommendations to guide the evolution of component-based product families, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.11.009

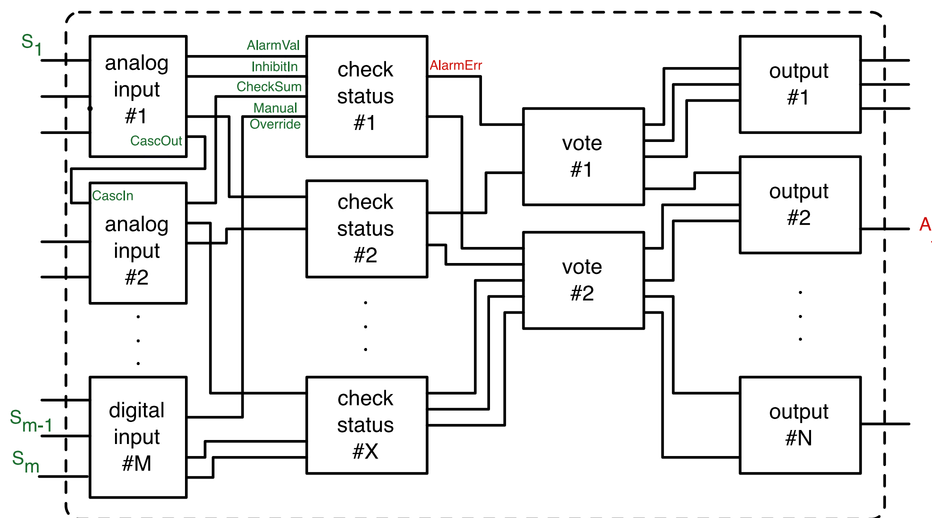

Fig. 1. Component composition network for a simplified example system.

The remainder of this paper is structured as follows: Sections 2 and 3 describe the context of our research and summarize the state of the art. We present the overall approach in Section 4, and conclude in Section 5.

2. 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. Such systems are paramount to ensuring reliability and dependability of marine equipment, and malfunction of such systems can pose severe danger to people and natural environment. They are prime examples of the kind of large-scale, software-intensive, safety-critical embedded systems that we discussed in the previous section.

We use the following (standard) terminology for component-based systems: a *component* is a unit of composition with well-defined interfaces and explicit context dependencies [10]; 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.

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 (Fig. 1). The components are implemented in MISRA C (a safe subset of C [11]), are 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.).

The "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-and-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 of the size, consider that in contrast to the 11 instances and 4 stages shown in Fig. 1, the safety system of a typical real-life installation has 12 to 20 stages in each pipeline, and in the order of 5000 component instances. Since components are so configurable, very few product-specific components need to be developed, and new products can be composed largely by configuring and interconnecting multiple instances from a core set of around 250 components which is shared among all members of the product family.

Based on workshops and interviews with safety domain experts and software engineers at KM [12], we have identified the following causes for variability in this domain: (1) functional requirements of each product category; (2) customer-

Please cite this article in press as: L. Moonen, Towards evidence-based recommendations to guide the evolution of component-based product families, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.11.009

specific requirements; (3) size and structure of each installation; and (4) different deployment configurations. To deal with this variability, our industrial collaborator adopted a component-based product development process that can be regarded as an instance of 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 [1]. They did not adopt more formal PLE activities like variability modeling.

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. [13]); 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 such knowledge may be forgotten or lost when team members leave.

3. State of the art

Lehnert provides a review of software change impact analysis [14]. 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. [15] propose a change impact model to investigate the influence of high-level design on 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. [16] 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. There is an implicit assumption in [15,16] 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.

We conjecture that the separation of an IIS and an FIS to increase precision and scalability is equally beneficial for component-based systems as for object-oriented systems. Obviously, one needs to tune entity abstractions and dependency links to make them suitable for representing the system, and for representing the complete product family. Moreover, the assumption that all dependencies yield equal impact is a simplification that has much larger effects on component-based product families, especially in a setting where we want to compare alternative change scenarios and need a way to somehow approximating *impact scale* (i.e., approximating the size of the affected area).

Some studies use a formal approach to specify component interfaces and component composition mechanisms to conduct CIA, or to assess modifiability of component-based systems using CIA-inspired techniques [17–19]. In a nutshell, these approaches describe a component solely based on its provided/required interface functions. Each of them defines their own variant 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 straightforward 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 [20] conduct CIA at 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 aims at the inter-component level, whereas our goal is to analyze the impact of fine-grained changes at the intra-component level and propagate these to the inter-component and family levels.

Recently, there has been an increased interest in tailoring CIA to software product-lines [21,22]. Díaz et al. [22] 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 at the architectural level. In general, these studies are aimed at exploiting state-of-the-art features of model-driven engineering and product-line engineering, such as domain-specific modeling and variability modeling.

Please cite this article in press as: L. Moonen, Towards evidence-based recommendations to guide the evolution of component-based product families, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.11.009

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.

In our earlier work [23,12], we have presented a technique to reverse engineer a fine-grained system-wide dependence model from the source and configuration artifacts of a component-based system and track the information flow throughout the system. It is our hypothesis that these models could be a suitable starting point for conducting change impact analysis. However, to the best of out knowledge there is no scientific literature on the (approximate) quantification and comparison of impact caused by alternative evolution scenarios, which is one of the requirements for creating cost-effective software evolution recommendation technology.

4. Challenges and research questions

Our overall goal is to develop technology that supports software engineers with the accountable evolution of families of complex, safety-critical, software-intensive systems. To reach this goal, we propose to build on the heterogeneous source-code analysis methods developed in our earlier work [23,24] and investigating the following avenues:

- Automated, scalable, analysis and transformation technology to systematically reverse engineer fine-grained system-wide and family-wide abstractions (models) from the heterogeneous software artifacts that are used for the component-based development of families of software-intensive systems.
- Improving existing software change impact analysis algorithms [14] to enable scalable, precise, change impact analysis based on these system-wide and family-wide models.
- Devising recommendation technology [25] that utilizes the system-wide and family-wide change impact analysis methods to guide software engineers during evolution and help them determine an evolution strategy that minimizes re-certification efforts.

To address the challenges in a scalable fashion, we propose to adopt a model-driven approach that relies on modeling standards such as OMG's Knowledge Discovery Meta-model KDM [26]. In model-based software engineering (MBSE), “models” are simplified representations of complex software products or product families. MBSE will help us to reduce the complexity of analyzing and reasoning about software by raising the level of abstraction. We believe in an evidence-based approach where models are not created as new artifacts, but are reverse engineered from the actual artifacts used to build the system. As such, we have identified the following challenges to achieve this goal:

(1) Precise (fine-grained) CIA algorithms that scale to the product family level: State-of-the-art source-based CIA techniques are limited to providing either a detailed analysis at the level of individual components (e.g. [15,16]), or a course-grained analysis of component-based systems where complete components are the smallest elements that can be considered [20,17–19]. Recent studies address CIA for software product-lines by exploiting state-of-the-art techniques such as domain-specific modeling and feature modeling [21,22]. However, most manufacturers of software product families have not (yet) adopted these advanced software product-line techniques. Research into alternative approaches for family-wide CIA is needed to support the large body of existing software that simply cannot be redeveloped from scratch (e.g., for economic reasons).

RQ1 *How can we construct change impact analysis algorithms that enable the scalable and precise analysis of complete component-based product families, and how can the scale of the observed impact be quantified?*

We propose to address this question by investigating extensions of existing software change impact analysis algorithms [14] that make them applicable for change impact analysis on system-wide, and family-wide models like the ones recovered in the earlier phase. As we have seen before, in the context of software products, CIA can be defined as a graph reachability problem and the most important question is how to create a precise graph. However, in the context of product families with variability, change impact analysis becomes much more complex. It is no longer a question of “which components are affected”, but also a question of “which products are affected”. As a result, CIA on product families is no longer a transitive graph reachability problem but it becomes a constrained graph reachability problem. Since we analyze models that represent (families of) existing systems, which differs from analyzing all possible configurations based on variability specifications like feature models, we avoid many of the theoretical complexity issues associated with this class of reachability. Nevertheless, precautions will need to be taken to avoid a combinatorial explosion during the analysis.

In addition, measures need to be defined to quantify, or approximate, the impact scale of a proposed change and include the computation of this measure in our change impact analysis algorithm. Our current line of thought is that the length of the path that is computed during the reachability analysis is a suitable candidate as a first approximation of impact scale. Refinements to this measure need to be investigated, and validated based on empirical data [27].

(2) Analysis of heterogeneous software artifacts, including component composition and configuration: Analyzing the properties of programs within closed code boundaries is a well-established area [28], and techniques have been implemented in professional program analysis tools [29,30]. The closed world assumptions in conventional approaches limit the analysis to complete, homogeneous systems, written in a single programming language. In contrast, components can be implemented

in various programming languages, and interconnected using a variety of configuration artifacts, ranging from simple key-value maps to elaborate domain-specific configuration languages. The inability to deal with heterogeneous source artifacts complicates the analysis of component-based products [8,9]. Moreover, even though composition and configuration of components are crucial for component-based development, we found that there is surprisingly little support for incorporating this information in static software analysis [23]. Change impact analysis on component-based product families cannot be realized unless these issues are addressed. Earlier we have reported on our initial efforts in this direction [23,12], but a thorough study aimed at the challenges of representing complete product families and suitable for precise software change impact analysis has not been conducted.

RQ2 *How can we automatically reverse engineer precise system-wide and family-wide dependence models from the source artifacts that are used for the development of component-based product families?*

We propose to investigate this question by means of program analysis and transformation: a process of systematically analyzing the data present in software development artifacts and transforming it into another representation, such as the dependence graphs (or models) used for change impact analysis.

The main challenges include: (1) dealing with heterogeneous source artifacts while scaling to the system and family levels; (2) capturing commonality and variability in representations of product families; (3) recognizing domain-specific or other user-defined abstractions; and (4) recovery of detailed models that allow navigation between abstraction layers to enable precise and scalable software change impact analysis. Amongst others, this requires rethinking what dependencies need to be modeled for a precise representation of a complete component-based product. Moreover, novel approaches need to be investigated for capturing the commonality and variability in representations of product families and for dealing with these aspects in a way that avoids a combinatorial explosion during analysis.

Investigating this research question requires systematically exploring the opportunities for identifying both patterns that abstract from details in the source artifacts, as well as relations that summarize dependencies between these abstractions. This includes the inference of domain-specific and user-defined abstractions as this will help system engineers to interpret analysis results with respect to their mental model of the system by decreasing the conceptual gap. Other aspects to consider include: How can we assess the precision and completeness of the recovered models? How can we document the decisions and reasoning during model reconstruction? I.e., how can we achieve traceability?

(3) Comparing the impact of evolution scenarios: CIA techniques in scientific literature make the implicit assumption that all dependencies yield equal impacts. Consequently, they manipulate impact as a Boolean value, i.e., for a given change they can only compute whether or not a program element is impacted by that change. However, to compare alternative evolution scenarios, a measure is needed to assess the impact scale of each strategy. As alluded to before, this is especially important in the context of ICSSs since the safety-critical nature of these systems requires that they are re-certified after a change, and cost-effective evolution decisions should prioritize evolution scenarios that minimize impact scale, and thereby minimize re-certification efforts.

Since it is difficult to understand and reason about the effects of a change on a family of products, our aim is to support engineers in this process by guiding them through the decisions based on source-based evidence. We propose to do this by investigating recommendation systems for software evolution. Recommendation systems are software applications that support navigation through large information spaces by helping people to collect valuable information and make decisions where they lack experience or can't consider all the data at hand [25].

RQ3 *How can we devise novel recommendation technology that uses change impact analysis results to guide engineers during the evolution of software product families?*

Recommendation systems consist of two main parts: (1) a mechanism to collect and organize data in a model, and (2) a recommendation engine to analyze the data and give (ordered) recommendations that satisfy certain criteria. As data collection mechanism, we propose to use the impact analysis and impact scale approximation algorithms from the previous challenge.

The conventional analysis approach in recommendation engines builds on clustering and classification techniques [31,32]. However, the question of finding an evolution strategy with minimal re-certification efforts does not lend itself well to being translated into a clustering or classification question. Instead, we propose to investigate devising a cost-effective recommendation engine by translating this question into a constraint optimization problem. In such an approach, constraint programming (CP) is used to minimize a cost function that uses the impact scale of an evolution strategy as approximation of its associated re-certification effort. In fact, a single-objective cost function is most likely not enough and a multi-objective search is needed to account, for example, for the granularity of re-certification in the analysis: when re-certification is done at component level, it is better to select a strategy where one component has four locations that need updates, than a strategy in which four components each need one update, even when the impact scale of both scenarios is equal.

Constraint solvers can seek the global minimum in such a complex search space using a combination of constraint propagation and filtering, and search heuristics with a branch & bound procedure. One of the main advantages of using CP is that it provides a way to cope with the NP-hardness of this optimization problem by introducing time-contracts which

will compromise the calculation of the global minimum but guarantee a quasi-minimum solution that will still satisfy the given constraints.

5. 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.

This paper explores how reverse engineering and program comprehension techniques can be used to develop novel recommendation technology that support software engineers with making accountable software evolution decisions can only be based on concrete evidence gathered from the actual source artifacts. The contributions of this paper are the following: (1) we survey the specific characteristics of Integrated Control and Safety Systems in our application domain; (2) we discuss the state of the art and major research directions in change impact analysis for component-based product families; (3) we identify challenges, and pose a number of research questions that need to be addressed to develop the proposed recommendation technology.

Apart from techniques based on change impact analysis, another approach for estimating the effects of software change is to investigate how a system has evolved in the past [33]. 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) [34]. There is an emerging trend to integrate the two approaches to increase the precision of software evolution estimations [33,35]. In addition to the approach sketched here, 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.

References

[1] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison–Wesley, 2000.
[2] M. Jazayeri, A. Ran, F.v.d. Linden, Software Architecture for Product Families: Principles and Practice, Addison–Wesley, 2000.
[3] M. Matinlassi, Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA, in: Int'l Conf. Softw. Eng. (ICSE), IEEE, 2004.
[4] M. Svahnberg, J. Bosch, Evolution in software product lines: two cases, J. Softw. Maint.: Res. Pract. 11 (1999) 391–422.
[5] L. Moonen, Building a better map: Wayfinding in software systems, in: IEEE Int'l Conf. Program Comprehension (ICPC), 2011, Keynote.
[6] S. Bohner, R. Arnold, Software Change Impact Analysis, IEEE, 1996.
[7] S. Lehnert, A taxonomy for software change impact analysis, in: Int'l Ws. Principles of Softw. Evolution (IWPSE-EVOL), ACM, 2011, pp. 41–50.
[8] M.J. Harrold, D. Liang, S. Sinha, An approach to analyzing and testing component-based systems, in: ICSE Ws. Testing Distributed Component-Based Systems, 1999.
[9] A. Rountev, Component-level dataflow analysis, in: Int'l Conf. Component-Based Softw. Eng. (CBSE), Springer, 2005.
[10] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed., Addison–Wesley, 2002.
[11] L. Hatton, Safer language subsets: an overview and a case history, MISRA C, Inf. Softw. Technol. (IST) 46 (2004) 465–472.
[12] A.R. Yazdanshenas, L. Moonen, Tracking and visualizing information flow in component-based systems, in: IEEE Int'l Conf. Program Comprehension (ICPC), 2012.
[13] S. Deelstra, M. Sinnema, J. Bosch, Product derivation in software product families: a case study, J. Syst. Softw. 74 (2005) 173–194.
[14] S. Lehnert, A review of software change impact analysis, Report ilm1-2011200618, Techn. Univ. Ilmenau, 2011.
[15] M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, A change impact model for changeability assessment in object-oriented software systems, in: European Conf. Softw. Maintenance and ReEng. (CSMR), IEEE, 1999, pp. 130–138.
[16] X. Sun, B. Li, C. Tao, W. Wen, S. Zhang, Change impact analysis based on a taxonomy of change types, in: Computer Softw. and Applications Conf. (COMPSAC), IEEE, 2010, pp. 373–382.
[17] Z.-j. Wang, X.-f. Xu, D.-c. Zhan, Agility evaluation for component-based software systems, J. Inf. Sci. Eng. 23 (2007) 1769–1783.
[18] L. Yan, X. Li, An interface matrix based detecting method for the change of component, in: Int'l Symp. Information Science and Eng., IEEE, 2008, pp. 38–42.
[19] C. Mao, J. Zhang, Y. Lu, Matrix-based change impact analysis for component-based software, in: Computer Softw. and Applications Conf. (COMPSAC), IEEE, 2007, pp. 641–642.
[20] T. Feng, J.I. Maletic, Applying dynamic change impact analysis in component-based architecture design, in: Int'l Conf. Softw. Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing, IEEE, 2006, pp. 43–48.
[21] H. Cho, Y. Cai, S. Wong, T. Xie, Model-driven impact analysis of software product lines, in: Model-Driven Domain Analysis and Software Development: Architecture and Functions, IGI, 2011, pp. 275–303.
[22] J. Díaz, J. Pérez, J. Garbajosa, A.L. Wolf, Change impact analysis in product-line architectures, in: European Conf. Softw. Architecture, 2011, pp. 114–129.
[23] A.R. Yazdanshenas, L. Moonen, Crossing the boundaries while analyzing heterogeneous component-based software systems, in: IEEE Int'l Conf. Softw. Maintenance (ICSM), 2011.
[24] A.R. Yazdanshenas, L. Moonen, Fine-grained change impact analysis for component-based product families, in: IEEE Int'l Conf. Softw. Maintenance (ICSM), vol. 5, 2012.

Please cite this article in press as: L. Moonen, Towards evidence-based recommendations to guide the evolution of component-based product families, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.11.009

ARTICLE IN PRESS SCICO:1638

[25] M. Robillard, R. Walker, T. Zimmermann, Recommendation systems for software engineering, IEEE Softw. 27 (2010) 80–86.
[26] OMG, Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v1.2, 2010.
[27] F. Shull, J. Singer, D. Sjøberg (Eds.), Advanced Topics in Empirical Software Engineering, Springer, 2008.
[28] D. Binkley, Source code analysis: A road map, in: Future of Softw. Eng. (FoSE), IEEE, 2007, pp. 104–119.
[29] P. Anderson, 90% perspiration: Engineering static analysis techniques for industrial applications, in: IEEE Int'l Working Conf. Source Code Analysis and Manipulation (SCAM), 2008, pp. 3–12.
[30] P. Anderson, T. Reps, T. Teitelbaum, M. Zarins, Tool support for fine-grained software inspection, IEEE Softw. 20 (2003) 42–50.
[31] S. Owen, R. Anil, T. Dunning, E. Friedman, Mahout in Action, Manning, 2011.
[32] I.H. Witten, E. Frank, M.A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2011.
[33] H. Kagdi, J. Maletic, Software-change prediction: Estimated+actual, in: IEEE Int'l Ws. Softw. Evolvability (SE), 2006, pp. 38–43.
[34] H. Kagdi, M.L. Collard, J.I. Maletic, A survey and taxonomy of approaches for mining software repositories in the context of software evolution, J. Softw. Maint. Evol.: Res. Pract. 19 (2007) 77–131.
[35] L. Hattori, G. dos Santos Jr., F. Cardoso, M. Sampaio, Mining software repositories for software change impact analysis: A case study, in: Brazilian Symp. Databases (SBBD), 2008, pp. 210–223.

Please cite this article in press as: L. Moonen, Towards evidence-based recommendations to guide the evolution of component-based product families, Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.11.009