Towards System-Wide Analysis of Heterogeneous Component-Based Software Systems

by

Amir Reza Yazdanshenas

Thesis submitted for the degree of Philosophiae Doctor
Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
September 2015

Abstract

One way to manage the complexity of software systems is to compose them from reusable components, instead of starting from scratch. Components may be implemented in different programming languages and are tied together using configuration files, or glue code, that entail their instantiation, initialization and interconnections. Although correctly engineering the composition and configuration of components is crucial for the overall behavior, there is little support for incorporating this information in the static verification and validation of such systems. Analyzing the properties of programs *within* closed code boundaries has been studied for some decades and is a well-established research area, where as analyzing *heterogeneous component-based* systems is not as mature. The heterogeneous nature of source code and configuration artifacts often hinders system-wide analysis of component-based systems.

This thesis contributes a method to support analysis *across* the components of component-based systems by, firstly, building upon the Knowledge Discovery Metamodel to reverse engineer homogeneous models of the system. This homogeneous model is then used as the building block to support various activities crucial for efficient maintenance and evolution of heterogeneous systems, namely: information flow analysis to support *quality assurance*, visualizations needed for *comprehension*, and *change impact analysis*.

For end-to-end information flow analysis of component-based systems, we apply *program slicing* through and across the components. Dependencies between pairs of inputs and outputs are revealed at both component-level, as well as system-level. This method is implemented in a prototype tool, that has been successfully used to track information flow across the components of a large-scale industrial system.

The homogeneous model repository is extended to allow for *visualization* of information flows in component-based systems at *various* levels of abstraction, and from two complementary perspectives. We propose a hierarchy of five interconnected views to support the comprehension needs of both safety domain experts and developers from our industrial partner. The abstractions are selected to minimize visual distraction and reduce the cognitive load while satisfying information needs of the users. The views are interconnected in a way that supports both systematic, as well as opportunistic navigation scenarios. We discuss the implementation of our approach in a prototype tool, called FlowTracker, and present the results of two qualitative evaluation studies on the effectiveness and usability of the proposed views for software development and software certification. Based on the received feedback we can report positive results, and a number of in-depth insights for future improvements.

As a third step, we devise a method to support the evolution of component-based systems as *members of product portfolios*, where evolution can happen both as a result of collective domain engineering activities, as well as product-specific developments. Developing software product-lines based on a set of shared components is a proven tactic to enhance reuse, quality, and time to market. Adaptations of product-lines software engineering is heavily used by our industry partner, Kongsberg Maritime. To help developers make informed decisions about prospective modifications, we contribute a method to estimate what other sections of the system, as well as what other products in the same product portfolio, will be affected and need "maintenance attention." We use static program slicing as the underlying analysis technique, and adapt the aforementioned framework of homogeneous model repositories to accommodate large-scale product portfolios. Our method trades off precision and scalability in a way to maintain maximum precision at intra-component analyses, while being highly scalable upon the propagation of ripple effects. Our approach also ranks the results based on an *approximation* of the scale of impact. We have implemented our approach in a prototype tool, called Richter, which was evaluated on a real-world product family.

As the final section of this thesis, we conduct a study to asses the state of the art in cross-lingual program analysis. We contribute a systematic literature review on the available literature, and discus several implications for future research and practice as a basis for the improvement of software evolution in multi-language systems. We search through seven digital libraries to find the relevant primary studies on cross-language program analysis, and identify additional studies with manual snowballing, which results in 75 studies. We classify the studies based on several criteria, including their purpose (why), the adopted or suggested approach (how), the information leveraged in each programming language or artifact (what), and the conducted evaluation (quality). The results include objective findings on the diversity of the applied techniques, application domains, programming languages, and reliability of the approaches. Based on the findings of this literature review, several implications for research and practice are discussed, including potential breakthroughs, and negative effects of the shortage of community-driven research.

Acknowledgments

First and foremost, I am grateful to Simula Research Laboratory and Simula School of Research and Innovation for creating a fantastic framework to conduct industry-driven research, as well as a pleasant working atmosphere. Specifically, I want to thank Aslak Tveito, Are Magnus Bruaset, and Molly Maleckar for their support of my PhD program.

I want to express my gratitude to Kongsberg Maritime for the opportunity of this research collaboration. I am grateful to Lionel Briand for his involvement in initializing the collaboration with Kongsberg Maritime. Special thanks goes to Anne-Heidi Mills as an instrumental contributor to this collaboration.

I want to cordially thank Magnus Jørgensen, my secondary supervisor, for his support and insightful comments at times most needed. I would like to show my gratitude to my previous secondary supervisor and my current employer, Erik Arisholm, for his support and appreciation of my PhD program and for granting me the necessary time to finish what I started almost six years ago.

I am most thankful to Leon Moonen, my mentor and principle supervisor, for accompanying me on this journey. His insights, scientific rigor, thoroughness in learning, and constant urge for self-improvement have helped me to become a better learner. I wish to thank him again for going an extra mile in finalizing this rather prolonged research program, despite several ups and downs.

I am thankful to my many colleagues and friends at Simula for creating such an inspiring and joyous working atmosphere. It was a privilege to interact with so many smart people with various cultural backgrounds: Hadi, Razieh, Nina, Shokoofeh, Raj, Aiko, Andrea, Shuakat, Zoheib, Sunil, Sagar, Tor, Hans Christian, Kari, and many more.

Last but by no means least, I am truly in debt to my family for their relentless support throughout these years, and not letting me feel absent in the many years I was. Dad, thanks for all that support. Mom, This is for you.

List of Papers

Paper I
Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems
Amir Reza Yazdanshenas, Leon Moonen
In Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)

Paper II
Analyzing and Visualizing Information Flow in Heterogeneous Component-Based Software Systems
Amir Reza Yazdanshenas, Leon Moonen
Submitted to Journal of Information and Software Technology, 2015. This is an extended version of the paper that was published in Proceedings of the 27th IEEE International Conference on Program Comprehension (ICPC 2012)

Paper III
Fine-Grained Change Impact Analysis for Component-Based Product Families
Amir Reza Yazdanshenas, Leon Moonen
In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)

Paper IV
Cross-language program analysis for the evolution of multi-language software systems: a systematic literature review
Amir Reza Yazdanshenas, Leon Moonen
Submitted to Journal of Software: Evolution and Process, 2015.

Contents

Summary

Summary

1 Introduction

Automated or semi-automated systems are increasingly prevailing in modern society, and so is their essential element: software. Software systems contribute to almost every aspect of our society, such as energy, heavy industries, aerospace, commerce, consumer products, entertainment, information technology, and many more. This dominating trend in software brings forth two other inevitable and ever-increasing trends:

1. The necessity of quality assurance
2. The complexity of the software's behavior and structure

The more software systems become essential for us, the more important is their integrity. This necessity is maximized in safety-critical systems, in which a single defect could pose a major hazard for the humans or the environment. In this situation, it is to be expected that enough attention should be paid to the quality of software systems prior to their deployment. Depending on the expected reliability of the system, the integrity of the software system should be assessed thoroughly, often using some form(s) of testing and/or program analysis. The degree of thoroughness in the quality assurance process depends on many factors, such as the criticality of the system, the number of users, the cost of a prospective defect, the budget at a project's disposal, etc. For safety-critical systems the stakes are so high that the common desire is to ace on the quality assurance process, however, in practice, the extent of such processes is limited to a trade-off versus their costs. In general, quality assurance is not a trivial task. For some systems (e.g., safety-critical systems) the costs of quality assurance could even exceed the cost of building the system, and therefore, the success and failure of the quality assurance process could be the main determining factor in the faith of such projects.

On the other hand, as software systems tend to constantly grow in functionality, so *might* their size and structural complexity. This trend can come at odds with the growing criticality of quality assurance, unless appropriate measures are taken in advance to enhance quality assurance processes, for instance by extended tool support. The growth of the complexity of a system does not necessarily root from more use cases at the end-users' disposal, but sometimes it is simply due to cope with more users or more complex distribution models. For instance, once a monolithic desktop application expands the scope of its service delivery to users across multiple locations, it might adapt a more complex distributed architecture while its set of use cases remain essentially the same from the end-users' perspective.

Software developers tackle the system's increasing complexity by dividing the system in smaller units (e.g., subsystems, components, classes, methods), perhaps with each unit being implemented in a programming language that fits best the purpose of that unit. This ubiquitous divide-and-conquer method in software engineering can be a double-edged sword. It facilitates the design and implementation of individual units, while it obstructs the comprehension, manipulation, and quality assurance of the overall system. The latter phenomenon is partly due to the (potential) language heterogeneity of the different system units. This heterogeneity often renders the available language-specific testing and program analysis methods simply inapplicable, or at best only applicable after extensive adaptations.

Component-Based Software Engineering (CBSE) can be considered as a special school of the more general divide-and-conquer design technique. CBSE emphasizes software reuse by composing software systems from prefabricated units, called components [1]. CBSE could be viewed as a continuation of the goals and principles of Object-Oriented (OO) software design, however, components are generally expected to be *coarse-grained* enough to deliver richer value than the medium-sized objects. CBSE has gained considerable attention since its introduction in the domain of software, and today several successful component-based technology stacks exist. Nonetheless, despite several decades of research and practice in CBSE, few people might argue that the goal of seamless hot-pluggable "software-ICs" [2] [3] have been fully realized. There could be several technical and non-technical reasons behind this illusive target, but one major reason is perhaps the lack of a universal coordination model among components.

In CBSE, components are only the building blocks, and the final system still needs to be synthesized using a coordination model which enables the components to interact. This coordination mechanism, by itself, adds a level of heterogeneity to the final product. Apart from that, the lack of a common agreement in modeling, specifying, and implementing component interactions (despite several standardization attempts [1]), hinders any cross-component analysis. This characteristic of component-based systems, in turn, puts the already-lagging program analysis technologies several steps behind, compared to mono-lingual non-component-based systems.

This thesis investigates the issue of language heterogeneity in software analysis, and contributes an approach to overcome the problem of conducting system-wide program analysis in multi-lingual component-based systems. Our objective is to facilitate the development and maintenance of such systems by providing intuitive abstractions over intra- and inter-component system elements that are capable of satisfying the information needs of various system stakeholders. To be relevant to real world systems, we need to devise a practical approach with close attention to applicability and cost-effectiveness. We propose a method that (i) combines Model-Driven Engineering (MDE) – a more recent engineering paradigm in software systems to tackle complexity – with program analysis techniques to develop a homogeneous model of heterogeneous component-based systems, (ii) enables analysis across the components of a component-based system, (iii) computes system-wide slices as a basis for various higher-level analysis techniques, (iv) exploits the flexibility of modern standards in MDE to visualize multiple abstractions over the collected knowledge about a system, and (v) enables impact analysis in families of component-based products. Information flows play a pivotal role in our approach, as the knowledge of a system or component's I/O is essential for correct manipulation of it. For the same reason, advanced tool support for identification and comprehension of the information flows is highly sought by our industry partner[1] in this project. We have developed two prototype tools up to this point, and a number of evaluations are conducted to assess the applicability of the tools to our industry partner. Detailed descriptions of how the above goals are realized, together with concise guidelines on how to cost-effectively build upon reusable well-established program analysis technologies are also provided. Taking a broader perspective than CBSE, this thesis also investigates the challenges posed by language heterogeneity in software systems and summarizes the proposed solutions as the state of the art in *cross-lingual* program analysis.

Contributions This thesis focuses on a *subset of* obstacles introduced to program analysis in the presence of multi-lingual component-based systems. Contributions are related to both design and implementation of new cross-component analysis techniques, as well as presenting a concise overview on cross-lingual program analysis techniques. In particular:

1. We have identified a generalizable problem of identifying and verifying information flows in component-based systems based on program slicing. The problem domain as well as the essential characteristics of viable solutions were identified with collaboration with our industry partners, based on a battery of real-world, large-scale, safety critical systems. In particular the inherent need for maintaining system-wide as well as component-wide scopes while conducting and presenting the findings of the analysis is one of the main objectives of this thesis.

2. Extending on the notion of dependence graphs, we define System-wide Dependence Graphs (SDGs) to include (intra-)Component Dependence Graphs (CDGs) as the building blocks, and Inter-component Dependence Graphs (ICDGs) as the gluing material. In that respect, our approach is distinguished from the original and seminal method of inter-procedural program slicing [4] in covering both *source code* and *configuration artifacts.*

[1]Kongsberg Maritime, http://www.km.kongsberg.com

3. We present a generalizable approach that combines model-driven engineering with program analysis techniques to support analysis across and through the components of heterogeneous component-based systems. For the purpose of cost-effectiveness, amongst others, we build upon the foundations laid out by OMG‘s Knowledge Discovery Metamodel (KDM) to reverse engineer homogeneous (dependence) models from heterogeneous software artifacts and use this model as the basis for our analysis. By doing so, we add a point of reference to the use and extension of KDM in an industrial setting, extending an area of literature that is currently underdeveloped. We have implemented and evaluated our slicing approach by building a prototype tool which has been successfully used to track information flow in a component-based system using program slicing, and has satisfied industry-driven scaling and efficiency requirements.

4. Building upon the slicing framework, we devise and implement a hierarchy of views that visualize system-wide information flows at various levels of abstraction, aimed at supporting both safety domain experts and developers with different trade-offs between scope and granularity. The views were selected to reduce visual distraction and reduce cognitive overload and were interconnected in a way that supports both systematic as well as opportunistic navigation scenarios ending eventually to the source code. Information flows are visualized and navigable both from inputs' perspective (forward), and from outputs (backward).

5. We conducted two qualitative evaluations of the effectiveness and usability of our prototype tool (called FlowTracker) in collaboration with our industry partner. Based on the received feedback from the domain experts, key takeaways and lessons learned are discussed as a reference point for further improvement.

6. We devised Family-wide Dependence Graphs (FDGs) as a means to support quality assurance across a *portfolio* of similar products. Using this homogeneous and highly scalable model, we devise a method to trace ripple effects across component-based product families (change impact analysis). The model trades off precision and scalability in a way to achieve maximum precision while detecting intra-component change sets, and to achieve linear scalability while detecting the impact set across the whole product portfolio. We propose a measure to *approximate* the scale of the impact of a change based on program slice sizes and use it to rank the analysis results before the end user. We implemented our approach by building a prototype tool, called Richter.

7. We conducted a systematic literature review on the available literature on cross-language program analysis techniques. Our review identified 75 conference and journal papers, which are analyzed to identify numerous trends in the respective body of research and to answer eight research questions. In addition, the findings

of the review reveals a number of implications for future research initiatives, such as the open areas wherein research still falls short. The review also reflects on the strength of the findings and puts forward a number of suggestions to the interested research community to overcome the present shortcomings.

Thesis Structure The thesis is compiled as a collection of papers and is organized as the following:

Summary: This part presents a summary of the research conducted in this project, together with the main contributions of each paper. Section 2 summarizes the underlying concepts and the background information needed to understand the rest of the thesis. In Section 3, the overall solutions of the thesis are briefly discussed, followed by the methodology used to conduct this body of research in Section 4. Section 5 highlights the main results of each of our studies. Section 6 outlines the future directions to this research, and Section 7 concludes the thesis.

Papers: The second part of the thesis consists of the published or submitted papers, which present the results of this research project. Three of the papers have been already published in international and peer-reviewed conferences. Two articles have been submitted to journals, however, a shorter version of one article had been refereed and published in a conference and selected for a journal extension. Paper I lays out the overall framework and covers items one, two and three in the aforementioned list. Paper II discusses items four and five in visualizing the information flows; while Paper III addresses item six in the context of software product families. Item seven is reported in Paper IV.

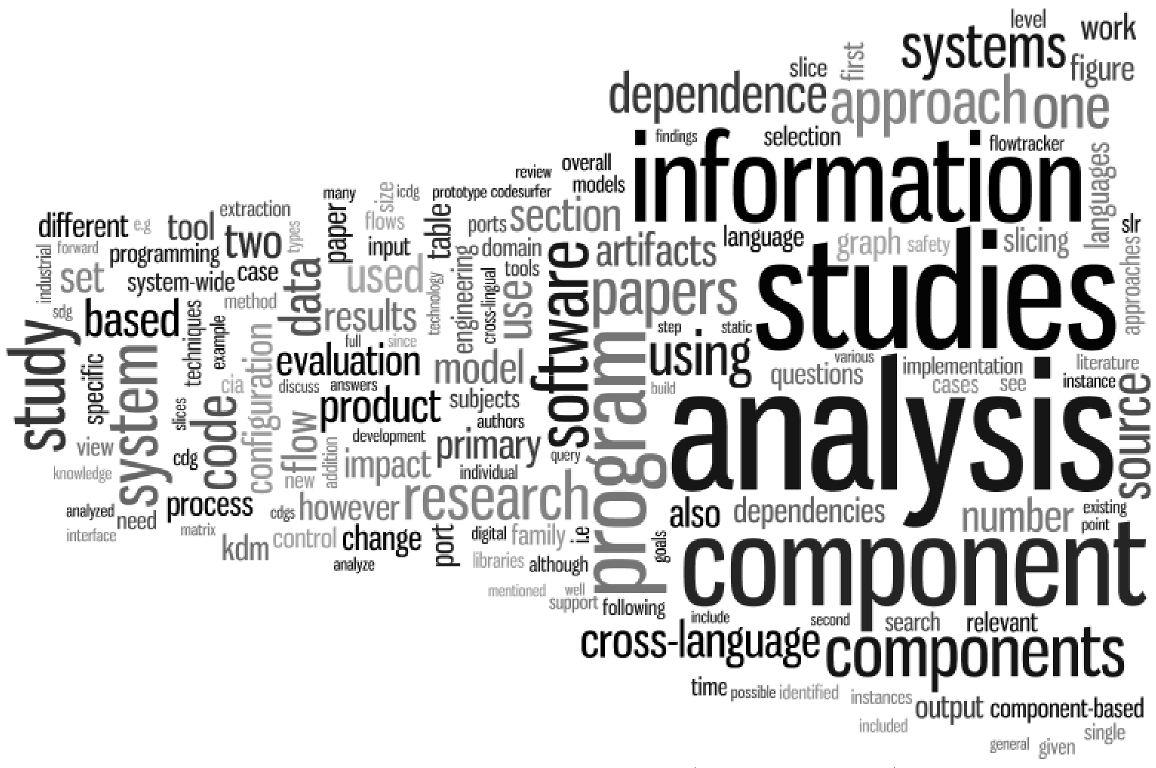


Figure 1: Word cloud of this thesis (`www.wordle.net`)

2 Background

This section presents a brief introduction to the material upon which this project is based on. The purpose is not to be exhaustive, but rather to present the preliminary knowledge to understand the rest of the thesis, to contextualize our research in reference to the sate-of-the-art, and to guide the interested reader to more comprehensive references. For clarity, it also mentions some of the "adjoining" subjects that are *not* addressed in this thesis.

As described in the following four subsections, we see four relevant research and practice domains (see Figure 2).

2.1 Program Analysis

The ISO/IEC/IEEE 24765 standard defines source code as the "computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator" [5]. Others have taken a more relaxed definition like "any fully executable description of a software system" [6], and Binkley also considers the "documents needed to execute or compile the program" as source code [7].

In the broadest sense, any (semi)automatic investigation of a software program to gain more knowledge about its structure or behavior can be categorized as **program**

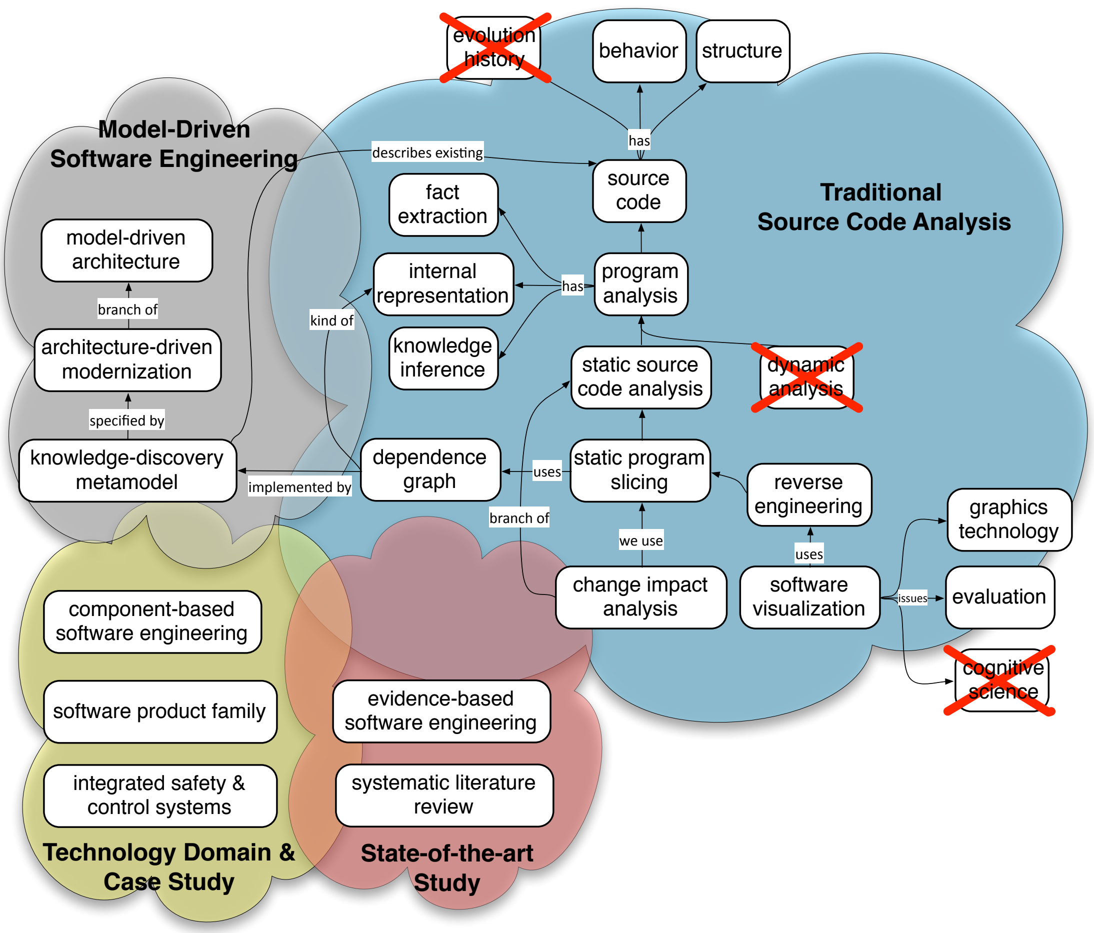


Figure 2: The four main research and industrial domains relevant to this thesis: (1) traditional source code analysis, (2) model-driven software engineering, (3) component-based systems, and (4) systematic literature reviews. Items with the red cross, are *not* investigated in this thesis. The relevance of each item is described in Section 2.1 to Section 2.4.

analysis. In the academic circles pertinent to program analysis, there has been a growing interest in recent years to include non-source code resources to gain more insight into the system and especially its development process. Examples of such secondary information resources are the metadata in the version control systems (commit logs), bug reports, mailing archives, developer activities in social medias, collective tacit knowledge in development teams, etc (e.g., look at [8] [9] [10] [11] [12] [13]). Once the analyzed material is limited to a system's source code, the respective analysis method can also be referred to as **source code analysis**. However, *program analysis* and *source code analysis* are occasionally used interchangeably, and we choose to do so in the thesis.

The analysis of software programs have been a long-standing active research topic, perhaps as old as the history of modern computing [14]. There are numerous application areas for program analysis and numerous approaches to conduct such analyses [15] [7],

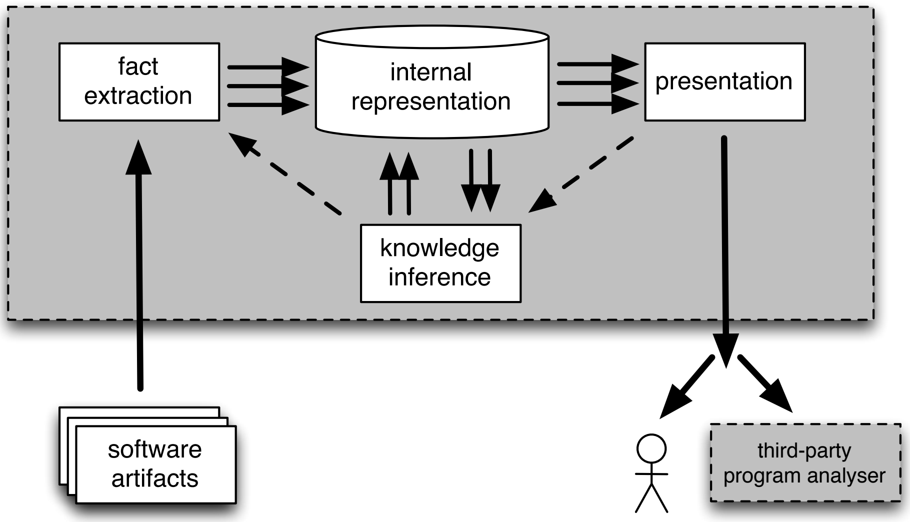


Figure 3: The anatomy of a source code analyzer.

from which we only point out the ones that are relevant to this thesis.[2] Most, if not all, program analysis approaches are composed of three components (see Figure 3)[7]:

1. Fact extraction (e.g. parsing)
2. An internal representation (or information repository)
3. Analysis (knowledge inference or abstraction).

Should you aim at analyzing a subject, you set out to gather *enough* information about it (step one). For future reference, you need to keep track of the gathered information in a suitable manner (step two), and then you are ready to crunch the gathered information to infer higher order knowledge about the system (step three). Of course, the resulting knowledge can be stored back into the repository and the analysis process can continue iteratively to produce more valuable knowledge. The output of a program analyzer should be either directly at the disposal of the end users (e.g., via report documents or the GUI), or to other program analyzers as input (e.g., via a custom API). The efficiency and user friendliness of the way a program analyzer presents the output to the users could not be overrated.

With the introduction of several programming languages (and programming paradigms), numerous application domains, and various end goals in mind, the academics has been kept very busy pushing forward the boundaries of each of the aforementioned steps. From that perspective, the contributions of this thesis are also in the direction of

[2]Harman argues that the topic of source code analysis will always be relevant (at least as long as computing is relevant). In his view, its importance will grow dramatically with the growing prevalence of software, up to a point where "source code will to be seen to be one of the most fundamental and pivotal materials with which humankind has ever worked" [14].

improving the same three steps, mainly to facilitate the maintenance of component-based and heterogeneous software systems.

Binkley lists a number of challenges which has been addressed and also a number of emerging challenges yet to be addressed by the research community [7]. Cross-language or multi-language program analysis, which is one of the contributions of this thesis, is rightly mentioned as one of the existing challenges [16].

2.1.1 Static Program Analysis

One of the main dichotomies in source code analysis is *static* versus *dynamic*. Static program analysis is conducted without executing the system. The analysis is usually performed using the design time textual representation of the source code, or at times using the compiled version of the source code (e.g., Java bytecode [17] [18] [19]). Dynamic analysis, on the other hand, is performed using program executions [20]. The required information is gathered through program instrumentation or profiling.

In many respects, static and dynamic analysis are the dual of each other [20] [21]. Static analysis does not concern program inputs, whereas dynamic analysis is actually bound to one or more inputs. Therefore, for static analysis to be sound, the results should be valid for all possible executions of the program. This characteristic forces static analysis to always make *safe* approximations, whenever the flow of data or control cannot be fully determined without knowing the run time values of the program variables. In such cases static analysis can resume, but only after taking all possibilities into account. This generally leads to a substantial increase in the number of false positives. However, this restriction does not apply to dynamic analysis, as the flow of control and data in already known to us. In short, static analysis is complete, safe, and imprecise, while dynamic analysis is incomplete and precise.

Despite, or indeed because of, the inherent differences between static and dynamic analysis, several authors have advocated for possible synergies of the two approaches [22], hoping to leverage the best of the two worlds [21] [7] [15]. Nevertheless, there are occasions in which only one of the two approaches is feasible. For instance, in cases where the source code is not accessible, one might be obliged to directly instrument the executable code and perform dynamic analysis, whereas not having access to an executable (sub)system (e.g., an incomplete system under construction) might incline the analyst to choose a source-based approach.

This thesis puts forward a case in which static analysis is the only feasible approach. As pointed out in Section 3, and in Paper I, this research project is motivated by a safety-critical integrated safety and control system. The safety-critical characteristic of the system prohibits the instrumentation and analysis of the system execution in the intended habitat of the system. Moreover, the *embedded* characteristic of the system makes it largely dependent on various proprietary hardware and software platforms, which in turn prohibits practicing the system in laboratory conditions without having access to the necessary stubs and emulators. Therefore, in this project we are limited to what static analysis can offer.

2.1.2 Program Dependence Graphs

As mentioned in Section 2.1, most program analysis approaches utilize an internal representation of the system under study, which fits best the purpose of the analysis. The most common internal representation is in the form of graphs [7], which can vary substantially depending on the type of information elements chosen as the graph nodes, and the type of dependencies representing the edges of the graph.

One main class of dependence graphs which is labeled as Program Dependence Graph (PDG), reflects the fine-grained control and data dependencies present in a program's source code using a directed graph [23] [4]. There is a single PDG for each procedure, and in a simple definition, the assignment statements and control predicates of that procedure constitute the nodes of the PDG. There is a control dependence edge from the vertex v_1 to v_2, if executing v_1 determines whether v_2 is executed or not. There is a data dependence from v_1 to v_2, when (1) v_1 defines variable x, (2) v_2 uses x, and (3) execution order can reach v_2 after v_1, and there is no other vertex in-between which defines x.

Horwitz et al., [4] extended the notion of PDGs to encompass *inter*procedural dependencies, and devised System Dependence Graph (SDG). This seminal progress enabled various analyses on complete (structured) programs with the call-return control model and paved the path for future extensions of dependence graphs for object-oriented [24], multi-thread [25], and web-based systems [26]. Apart from the PDG of each procedure, several other structured *internal representations* are needed to construct a complete SDG: abstract syntax tree, control flow graph, points-to graph, definition and use of variables [27].

Throughout this thesis, we refer to SDGs numerous times, and extend this notion to fit one of our goals: computing system-wide slices in component-based systems. While we are within component boundaries, our dependence graph is essentially the same as the traditional SDG defined in [4]. We add the required elements to construct the system-wide dependence graph in a way that is scalable and easy to construct. Implementing a reliable SDG constructor for a general-purpose language is not a trivial task. Therefore, we intend to choose an approach that maximizes reuse from the available professional tools, to increase the applicability of our approach in realistic settings.

One of the major applications of PDGs and SDGs is, of course, slicing.

2.1.3 Program Slicing

Program slicing is a technique to determine a subset of a program that can affect the computation at a point of interest, known as the *slicing criterion*. Alternatively, from a negative perspective, a slice can be decomposed from a program by detecting what sections of the program are *not* relevant to the slicing criterion. Weiser coined the notion of slicing about three decades ago [28] [28]. In his view, slicing was a variant of the ubiquitous divide-and-conquer technique and was inspired by the way developers understand the behavior of a subset of a program: ignoring the irrelevant parts, and resuming inspection on the relevant parts. His proposed (static) slicing approach was based on dependence graphs.

Since its inception, program slicing has been an active area of research. Numerous program slicing techniques have been proposed for different languages and different precision levels. Apart form that, various applications of program slicing have been proposed for software inspection and maintenance [29] [30] [31]. Debugging, impact analysis, testing, program comprehension, software metrics (e.g., cohesion measurement), reverse engineering, program decomposition (e.g., via diagnosing program clusters), program differencing are among the many applications of program slicing used to assist software development and evolution.[3]

As the output of slicing is (a subset of) the program's source code and not a higher order knowledge, it is formally regarded as program *manipulation* rather than program *analysis* in more recent papers [7] [14]. In this thesis we are largely indifferent to this taxonomy. However, we argue that our application of slicing bears some resemblance to the definition of program analysis, as it unveils the existing dependencies in the system which would be otherwise hidden to the "naked eye" due to the sheer size of today's industrial systems. An overly cluttered presentation of data hides the essential information from the user's perception [32] [33] [34], a phenomenon which is often referred to as "less is more" in the domain of information visualization. In this research project, we use static slicing to unveil input/output dependencies in and across components in large-scale systems. In that respect, we argue that the output of our slicing can be very well viewed as higher-order enough to be regarded as program analysis.

2.1.4 Software Model Reconstruction

Reverse engineering, *design recovery*, *software model reconstruction*, *architecture recovery*, and *software archeology* are only a handful of terms used to refer to a division of the broad topic of program analysis, which revolves around analyzing an existing system to infer (high order) knowledge about its structure, behavior, or the way it has been developed [35] [36] [37] [38]. Of course, there are (sometimes fuzzy) differences between each of the mentioned terms, however from an abstract point of view, all try to facilitate the comprehension and maintenance of the existing complex software systems. For instance, *architecture recovery* can be regarded as a *design recovery* method whose output is as abstract as the high level architecture of the system (e.g., determining

[3]Harman [14] succinctly reports the number of papers on the topic of slicing during a decade of holding the International Working Conference on Source Code Analysis and Manipulation (SCAM) [6].

coarse-grained subsystems and layers), instead of fine-grained design elements (e.g., procedures). Likewise, the main visible difference between *software archeology* and the traditional *reverse engineering* is that the former connotes strongly with *legacy systems* according to some researchers [37] [39] [40]. However, there is no consensus on what exactly is legacy code. Traditionally, legacy code was regarded as a heavily outdated piece of software, with some main elements (e.g., operating systems) that are no longer supported. On the contrary, some authors regard any high maintenance software [39], or a system without test code as legacy [41].

Indifferent to the minute differences of the aforementioned topics, this thesis is certainly relevant to the general topic of software model reconstruction. We conduct low-level program analyses to obtain high-level knowledge about the system. Later, we convey the obtained knowledge using various models for the benefit of the developers and other system stakeholders.

2.1.5 Software Visualization

As mentioned in Section 2.1, the third step in a general program analysis method is to infer abstract knowledge and present the result to the user in an appropriate manner. One approach is to use diagrams and graphical presentations, mainly to facilitate human comprehension. This need gradually introduced software visualization as a distinct research domain with dedicated conference venues [42]. In short, software visualization is the use of, possibly interactive, computer graphics and animations to enhance the interface to the user for understanding software artifacts and their evolution [32] [43].

Vision is the topmost dominant perception channel in humans [44]. Quite naturally so, the subject of software visualization has grabbed the attention of numerous researchers and practitioners. Although the use of graphics in computers is a long-standing practice, the use of visualizations to support the understanding of software systems is a relatively recent movement. Early works in the domain of software visualization seem to be inspired more by expert intuitions rather than based on informed choices (understanding how human perceive visual information). Pioneers would "simply" propose a certain visualization or notation they thought would best fit the metaphor they intended to convey - pure instincts. However, there has been a growing interest among computer scientists to study the human cognitive processes, particularly the visual perception, and design a visualization which fits best the human psyche [45] [46] [47]. In that respect, various research domains could contribute more and more to software visualization in the future, such as cognitive science, neuroscience, psychology, and psychophysics.

Apart from the complex enigmas of human cognition, there are at least two other major challenges to overcome in this domain: (1) graphics technology, and (2) evaluation. There are several design criteria for a successful software visualization tool [47] [48]. Easy and intuitive user interface, spontaneous and directional navigation, scalable and uncluttered diagrams, automated and user-driven layout algorithms, satisfying interactive experience, and aesthetic look-and-feel of the visualizations are only a handful of design elements that contribute to the overall *usability* of the visualization tool [33]. Contrary to one might think, there are not many general-purpose tools that are flexible enough to be used in more than one context, without extensive customization and high implantation overheads.

The main purpose of software visualization is to facilitate comprehension, and comprehension is inherently subjective. Therefore, evaluating the usability and efficiency of visualization tools is not straightforward and does not lend itself very well to quantitative measurements. In this situation, researchers often opt for a human-based qualitative, and sometimes purely heuristic-based evaluations [49]. Involving human subjects in a study introduces numerous challenges (e.g., designing questionnaires and priming), and perhaps it will never be "accurate," as such [50] [51] [52].

In this thesis, we do not touch upon the cognitive aspects of visual perception. In Paper II, we propose a number of visualizations which we think fit the requirements of component-based systems (and our industry partners), implement them using a number of available libraries cost-effectively, and evaluate their fitness using human-based qualitative studies.

2.1.6 Change Impact Analysis

Consistency, in general, is the holy grail of most of our daily activities as a software engineer. Software artifacts can have various (direct and indirect) dependencies, and once one is modified the engineers need to know what other artifacts should be updated accordingly. With the increasing complexity and size of today's software systems, determining dependencies using only developer's memory and comprehension becomes prohibitively difficult. This ubiquitous issue has opened a line of research, referred to as Change Impact Analysis (CIA) [53]. The results of CIA can be utilized in many typical activities in software change management, such as cost (effort) estimation, testing, and issue tracking [54] [55]. Many approaches in CIA use a variant of dependence graphs [56] [57] [58], and PDGs are also highly amenable for such purposes [59] [60].

We directly concern this topic in Paper III, in which we propose a *family-wide* dependence graph to conduct CIA in a family of component-based systems.

2.2 Model-Driven Engineering

Model-driven engineering is a relatively recent, but promising, approach in computer science, which tries to overcome the complexity of software systems by using abstract models. To conduct our research, we try to take advantage of the benefits of model-driven approaches, such as their language and platform independence, flexibility, and extensibility.

Model-driven engineering, or its OMG version Model-Driven Architecture (MDA) [61] [62], is more concerned with the forward engineering of software systems. However, we are more interested in the maintenance and evolution of *existing* systems, and that directly leads us to Architecture-Driven Modernization.

2.2.1 Architecture-Driven Modernization

As soon as one piece of software is labeled as "legacy," modernization becomes a debatable solution, and consequently a challenge. Architecture-Driven Modernization (ADM) is a line of research initiated by the Object Management Group (OMG) to tackle the modernization problem, using the new opportunities offered by model-driven approaches. ADM's mission statement, "creating specifications and promote industry consensus on modernization of *existing* applications," rightly does not limit its scope to only legacy systems [63].

The way we utilize ADM approaches through the thesis is exactly in line with their mission statement. We heavily rely on the their standardized specifications to facilitate the evolution and quality assurance of existing component-based systems.

2.2.2 Knowledge Discovery Metamodel

Knowledge Discovery Metamodel (KDM) is the cornerstone of ADM standards, with the goal of *representing* the existing software systems [64]. KDM is a metamodel, rich enough to cover arguably all elements and characteristics of existing software systems. KDM's scope spans from coarse-grained software elements, such as directories and files, to fine-grained elements, such as a single variable access. Inventory, Code, Build, Structure, Data, Business Rules, User Interface, Event, and Platform constitute KDM's main domain knowledge.

The original idea behind KDM initiative was to boost, or basically enable, *interoperability* between various program analysis tools [65]. With the current complex systems, which might be constructed from several heterogeneous subsystems, traditional program analysis and quality assurance tools will not be adequate, since they are commonly limited to a single programming language, a single platform, a single analysis target, and a proprietary output. With KDM's standardization, tool builders have access to a "common ontology and (data) interchange format" that facilitates a holistic system-wide quality assurance.

In this thesis we utilize a subset of KDM packages to represent our target software artifacts: components and component configuration files. This representation is used and enriched by higher order analysis tool-sets that eventually deliver the abstract knowledge we seek in a desired format. In case a certain element is not directly presented in the metamodel, we use KDM's own extension mechanism to enrich the metamodel according to our needs, while the "standard" ontology is not breached. Our experiences with KDM are described in our papers, to some extent.

KDM is specified based on Meta Object Facility (MOF), which is the cornerstone (Standardized) specification by OMG's MDA Task Force [66]. Currently, an implementation of KDM is available as open source [67] [65], through the Eclipse Modeling Framework (EMF) [68].

2.3 Component-Based Software Engineering

Along with researchers and practitioners' endeavors to overcome software complexity Component-Based Software Engineering (CBSE) was conceived. Raising the abstraction level of software artifacts, increasing reuse, improving separation of concerns, and facilitating off-the-shelf software market has been the major motivations behind CBSE. Software components have often been associated with hardware components, and even called "software ICs" by some of the pioneers of CBSE [1]. However, the original idea of complete plug-and-play software components have not been fully realized, arguably, mainly due to lack of standards for the component composition mechanism (also known as the "component bus"). Nevertheless, component technology have thrived enormously, and nowadays, software projects are rarely developed entirely from scratch and with no reusable components. There are numerous successful component-based technology platforms available, which are backed by well-established open-source or proprietary vendors. This topic has witnessed numerous dedicated conferences and workshops [69] [70], and literally hundreds of papers [71] [72] [73]. Also, our industrial partner in this project relies heavily on CBSE to deliver reliable mission-critical software, efficiently and in a predictable manner.

Components are not the only products of following the suggested software design principles. Objects, components, modules, and packages have a great deal of similarity, and their definitions sometimes overlap. Apart from that, these terms are highly overloaded, and they have (sometimes confusing) different definitions within the different programming paradigms (e.g., Ada packages and UML packages). Throughout this thesis we follow what Szyperski [1] provides as the definition of a component:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Szyperski perceives software components as *units of*: abstraction, accounting, compilation, deployment, maintenance, and – more relevant to this thesis – *analysis*. This characteristic of components to be the natural unit of analysis is perhaps the root cause of this thesis, as it overlooks cross-component analysis and quality assurance. Therefore, we deliberately steer our contributions toward component-based software systems in this research project.

2.3.1 Software Product Families

Software Product Line Engineering (SPLE) is an engineering approach to maximize *predictive* reuse of software by creating coarse-grained reusable software packages, which can cover a portfolio of products, rather than a single product. In this discipline of engineering reuse is deemed to not only include source code, but also frameworks, architecture, tools, methodologies, etc. In today's software market, where customization of the product according to various costumer needs is essential, most long-standing companies have no choice but to adapt a variation of SLPE to succeed, or merely to survive [74] [75]. Most SPLE approaches rely heavily on component-based software design, and therefore, there is considerable common grounds among the two design approaches. In this thesis, Paper III covers change impact analysis in component-based product families. As a concrete case study, we work on a family of safety-critical Integrated Control and Safety Systems (ICSS), which is developed based on a considerable shared component repository, developed in C.

2.4 Systematic Literature Reviews

Evidence-Based Software Engineering (EBSE) is a relatively recent shift in conducting research on software engineering, inspired by a similar paradigm in medical studies, to "provide the means by which best evidence from research can be integrated with practical experience in the decision making process regarding the development and maintenance of software" [76] [77]. The main goal of EBSE is to increase the *dependability* of the processes by which software engineers and researchers perform, by systematically investigating ideally all the available (or the best) evidence for a certain issue. Systematic Literature Reviews (SLR) are one of EBSE's toolset members used for "identifying, evaluating and interpreting all available research relevant to a particular research question." The main difference between an SLR and a traditional survey study is that SLRs should be *repeatable* by other researchers, and this mandates a detailed report of the research questions, a complete description of the evidence finding process, and a transparent synthesis of the gathered evidence [78] [79].

As pointed out in this section, the state of the art and practice in program analysis, software maintenance, quality assurance and component-based software engineering is enormously vast. There is a considerable number of papers in each of these domains and even in each of their sub-domains. In such circumstances, it is not improbable that researchers overlook a highly relevant and valuable piece of evidence and even reinvent the wheel due a fragmentary view over the matter. While conducting our individual studies presented in this thesis, we encountered several occasions in which we felt an urgent need to use the results of a relevant SLR to make informed choices, rather than based on our intuitions and (inevitably-) partial studies of the available evidence. However, several searches for such an SLR proved futile, and therefore, we decided to conduct an SLR on the topics we felt most needed a systematic survey. Paper IV is reports our findings and lessons learned in conducting an SLR on cross-lingual program analysis.

3 Cross-Component Analysis

The main goal in this thesis is to tackle the program analysis challenges in industrial component-based multi-lingual systems as a means to facilitate the typical tasks needed for software maintenance, namely comprehension, impact analysis, testing, and quality assurance. In the final section of this thesis, we intentionally drop the constraint of *component-based* to provide a deeper understanding into the specifics of *cross-lingual* program analysis in all multi-lingual systems. In this section, we first describe the common context in the development of industrial component-based systems, and explain the challenges of system maintenance imposed within this context. The context is motivated by the case of our industry partner, but it is by no means limited exclusively to our case studies. Then we present a brief overview of our contributions toward addressing a number of maintenance challenges.

3.1 Pitfalls of large-scale component-based systems

Component-based design is an effective approach to tackle the (accidental) complexity of software systems. However, even the best component-based design approach is no silver bullet to eliminate the essential and accidental complexity of modern real-world software systems [80] [81].

Large-scale component-based systems are typically built from a relatively limited number of reusable components, often implemented in general-purpose programming languages (e.g., C, C++, Java). Well-designed components are customizable (e.g., via parameters), nevertheless, the components might have to undergo further adaptations and customizations to fit the project at hand [82]. These parameterized code-bases (a.k.a. components) account for the lion's share of the reusability gain of the component-based design strategy.

Preparation of the individual components is only a prelude to the more sensitive task of assembling the components, to form the final product. In practice, the number of components is rarely trivial. In large-scale systems, especially the ones with fine-to-medium-grained components, the number of components and the total number of run-time instances of the components can easily mount to hundreds or thousands [83]. Apart form *instantiation*, *initialization* of the components is also a substantial task, depending on the "parameterizability" of the components. In addition, the final wiring or *configuration* of the components is often most demanding task. Even in the simplest cases, in which each component has only a handful of input and output ports, the total state space of component interconnections is overwhelmingly large. The three tasks of component instantiation, initialization, and configuration are often simply called *configuration* in the rest of this thesis, and also in general.

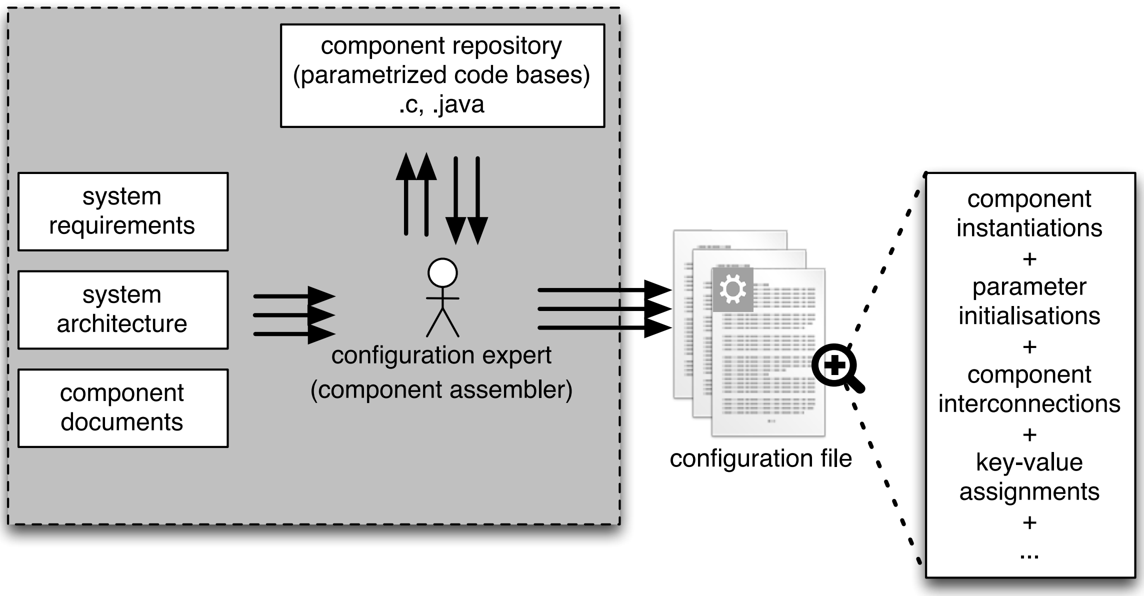


Figure 4: The overall configuration process.

The output of the configuration process is usually an overwhelmingly large configuration file, which together with the component code-bases constitute the complete system (see Figure 4). This configuration file conveys a a considerable, or the major, portion of the system specification. This specification is then read and processed by a variant of component framework (or component container), whose responsibilities are analogous to the "main" function in a procedural program. The component framework loads the specified components into the memory, makes enough number of instances of each component, sets the initial values, and sets up the component inter-communication channels (if necessary). The integrity of the configuration file, and its compliance with the correct component communication requirements, cannot be overrated with respect to the integrity of the complete system. Some studies indicate that a considerable portion of errors in highly component-based systems are the result of faulty configurations (e.g., up to 40% in [84]).

This configuration is by no means a trivial task and requires a considerable effort from the system engineers. Although the process of component configuration is not totally different from the traditional programming, there are a number of peculiarities that complicate this task:

- **Insufficient or lack of tool support.** For decades, the tremendous demand for better programming tool-sets have driven researchers and practitioners to constantly improve the capabilities, efficiency, and the overall user experience of the Integrated Development Environments (IDE). Modern IDEs provide a wide range of services to assist the developers in their daily tasks. Smart code completion, advanced refactoring, on-the-fly code analysis, automated build, facilities to manage and share code are only a few examples of what modern IDEs can offer. Services like "project explorers," syntax highlighting, text-based searching, debugging facilities (e.g., breakpoints, "watches," and runtime stack viewers) are taken granted in mainstream IDEs from a long time ago. On the contrary, similar facilities for component configuration files are either non-existing or offered at minimal levels. Only the highly popular component frameworks that have a considerable contributor crowd (e.g., Spring) offer the very basic services of syntax highlighting, code completion, and basic (XML-derived) syntax checks. In proprietary component frameworks, the developers are often left with nothing but a basic text editor to manipulate a gigantic text file.[4]

- **Complicated comprehensibility.** Comprehension is hindered in component-based systems due to two factors: (1) comprehensibility of the configuration file per se, and (2) the overall system comprehension. *Modal properties* are features in the source artifacts that help the developers' *associative memory* to maintain associations (or conceptual links) between points of interest [46]. These associations are essential for comprehension and navigation of the source artifacts. Modal properties can be essentially anything "that emphasizes (reminds of) a specific aspect of interest," such as lexical, structural, spatial, and syntactical beacons [46]. As mentioned before, configuration files can sometimes be lengthy, mono-colored, text-based, flat-structured files with few opportunities to form the needed modal properties, to maintain the required mental associations. On the other hand, having the overall program logic scattered among several software artifacts increases programmers' cognitive overheads; an issue known as *disorientation* [47]. The problem of disorientation is not limited to component-based systems, however, scattering program logic in more than one design paradigm (e.g., configuration specification versus programming languages) can increase the cognitive overheads.

- **Obstacles of program analysis.** Components are known to be the unit of program analysis in component-based systems ([1] p. 141). To be fair, this characteristic helps to solve a range of program analysis obstacles, however on the other hand, it introduces a range of other obstacles. Once software is structured into cohesive and independent components, memory or computationally intensive analysis algorithms are more feasible (e.g., massive slicing [85]), as there is less demand to conduct a global analysis on the complete system which might mount to millions of lines of code. Apart from that, component-based design facilitates

[4]Configuration files can be prohibitively lengthy as the mechanisms of *modular* design (such as classes and packages in object-oriented paradigm) has not yet been realized in many (proprietary) component configuration frameworks. Consequently, all the configuration data ends up in a single, flat (text) file.

the analysis and verification of portions of the system before other portions are developed, which helps to reduce the overall risk as early as possible in a project life cycle. However, there is always a limit to independent analysis of components in the overall system. According to the level of (inevitable) coupling between a group of components, some properties of the other components need to be considered during the analysis of a certain component, and to conduct a cross-component analysis certain properties of all components are needed. As the component interactions are needed to conduct a cross-component analysis, and as such interactions are specified mostly in the configuration artifacts, conducting system-wide analyses is not a trivial task. Moreover, components are regarded as "units of independent deployment" ([1], p. 36) and are, ideally, amenable for creating a third-party component market. Having heterogeneous components implemented in different programming languages, and third-party components whose source code might not be available at the time of analysis, complicates most system-wide analyses by leaps and bounds.

In addition to the aforementioned issues which directly root from the state-of-the-practice in component-based software development, there are also a number of generic problems that could sometimes be intensified due the application of component-based design guidelines. One such example is the problem of documentation. Outdated documents are a hassle in every software engineering methodology, however, as components could be developed independently by different organizations, this problem could cause more problems, and there are higher chances of having non-uniform documentations and terminologies. As these issues are not inherently the result of component-based design, we will not focus on them in this thesis. Nevertheless, one should remember that all such factors can amount to the challenges of evolution, maintenance, and quality assurance in component-based systems.

3.2 Toward system-wide analysis: an overview

One main motivation of this thesis is to devise and implement a cost-effective approach to address the aforementioned challenges in the development and evolution of component-based systems. Our immediate aim is to facilitate intra and inter-component program slicing, which in turn, can be the bed rock of various types of analyses, such as information flow analysis and visualization, and change impact analysis. Each of these analysis methods are a tool in the quality assurance tool-set, which requires comprehension, knowledge inference, and leveraging any source of information that can assist testing and fault discovery. The contributions of this thesis are presented step-wise, which corresponds to the incremental implementation road-map in reality.

3.2.1 System-wide dependence graphs: bridging modelware to grammarware

Grammarware refers to the whole technological space that revolves around the concept of grammars in programming languages, including the formalism and mechanisms used to specify and implement the grammar itself, in addition to the technology stack that is built on top of grammars [86]. From this perspective, all grammar notations and formalism, compilers, parse trees, abstract syntax trees, and most of the groundbreaking technologies used for language engineering are considered well within this technological space. This technological domain is among the most well-established ones in computer science and historically accounts for a considerable portion of the linguists' and computer scientists' contributions. A major advantage of this technological domain is the strong formalism behind grammars, which makes the derivatives to a large extent *unambiguous*,[5] *verifiable*, and machine *executable*. This close reliance to formalism, however, comes at the cost of reducing the flexibility, modularity, and interoperability of most technologies in this domain. A traditional C/C++ compiler might not be forgiving enough to compile a wrong or incomplete piece of source code (e.g., without the necessary header files); nor able to process source artifacts in other programming languages or even other dialects of the same language.

Island grammars are one proven way to increase the flexibility and robustness of parsers [87]. The idea behind island grammars is to maintain grammar precision in only specific areas and trade off precision with flexibility in areas that are not interesting for the task at hand. Another opening is to prospect the modelware technological space (Model-driven engineering (MDE)) for new opportunities. Most MDE standards are specified with close attention to *flexibility*, *extensibility*, and *customizability*. A well-balanced approach, which harvests each of grammarware and modelware in their own natural territory, can exploit the best of the two technological spaces: precision of grammars, and flexibility of (meta)models. This idea of having hybrid program analysis approaches in not unprecedented [88] [89]. We adapt the same idea to our context to enhance the overall applicability of our solution, brief description follows.

As pointed out in Section 2.1.3, program slicing has various applications in software evolution and quality assurance, and program dependence graphs are one major enabler of program slicing (see Section 2.1.2). Therefore, as our first step, we set off to build a homogeneous and system-wide dependence graph for component-based systems. Constructing robust and precise PDGs for general purpose programming languages is a mammoth task, and enforcing that as a prerequisite will severely undermine the applicability of our approach. Fortunately though, PDGs are fairly well established in grammarware, and there are a number of open source and commercial PDG constructors (e.g., CodeSurfer for C/C++ [27]).

[5]In the presence of ambiguities in the grammar, there are often assisting mechanisms to resolve them in most mainstream programming languages.

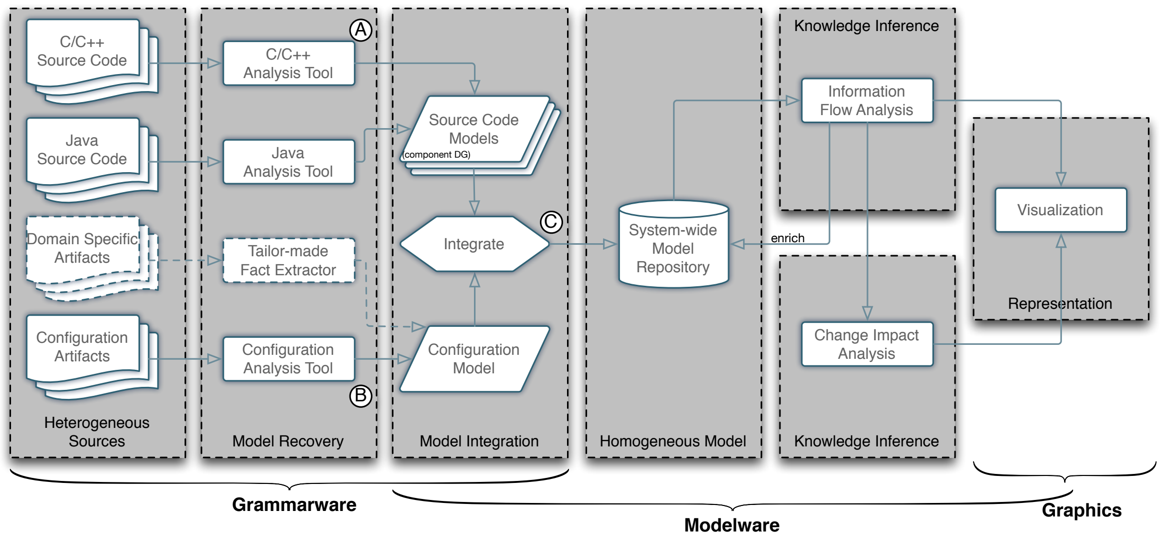


Figure 5: Overall approach.

Therefore, we devise our approach in a way to reuse the available state-of-the-practice in grammarware and construct robust and precise dependence graphs with minimum overheads (Figure 5, marker A). Of course, this trick works as long as we stay within the boundaries of individual components, as these are the only portions of the system developed in programming languages. Afterward, we analyze the configuration file(s) to understand component interactions needed to build the respective partial *intercomponent dependence graph* (Figure 5, marker B). Text or XML-based configuration files – as lengthy as they might be – often have less rigorous grammar than programming languages, and developing a special-purpose parser (or fact extractor) usually requires much less effort than those of, say, C++. Once we retrieve the dependence graphs of each component, in addition to the intercomponent dependence graph from the configuration information, we are ready to combine all these partial dependence graphs to construct the final system-wide dependence graph. This is where the flexibility and extensibility of modelware technologies are utilized at best to construct a homogeneous dependence graph from a number of heterogeneous partial graphs (Figure 5, marker C).

To pave the path for prospective integrations of our approach to other program analyses used for quality assurance, we choose to use a *standard* metamodel to represent our system-wide dependence graph. Knowledge Discovery Metamodel (KDM) is utilized for this purpose, as demonstrated in Paper I, is extensible enough to easily accommodate all the needed elements of program dependence graphs as well as any secondary attribute of source code elements, such as line number. We point out that this integrated information repository is not limited to the dependence graph and contains detailed information about various elements of the source code.

At this point we have the information necessary to compute intra and inter-component slices using a variation of traditional reachability analysis algorithms (see

Paper I).

3.2.2 Information flow analysis and comprehension

Computing slices on critical points of the system (e.g., inputs and outputs) can be a valuable source of information to the developers, however, the level of abstraction is still too low to be conveniently consumed. For this reason, we perform slicing not as an end, but as a means to provide higher order knowledge about the system. We use slicing as a decomposition technique to identify *information flows*, which are the channels in which information can be transferred within the system.[6]

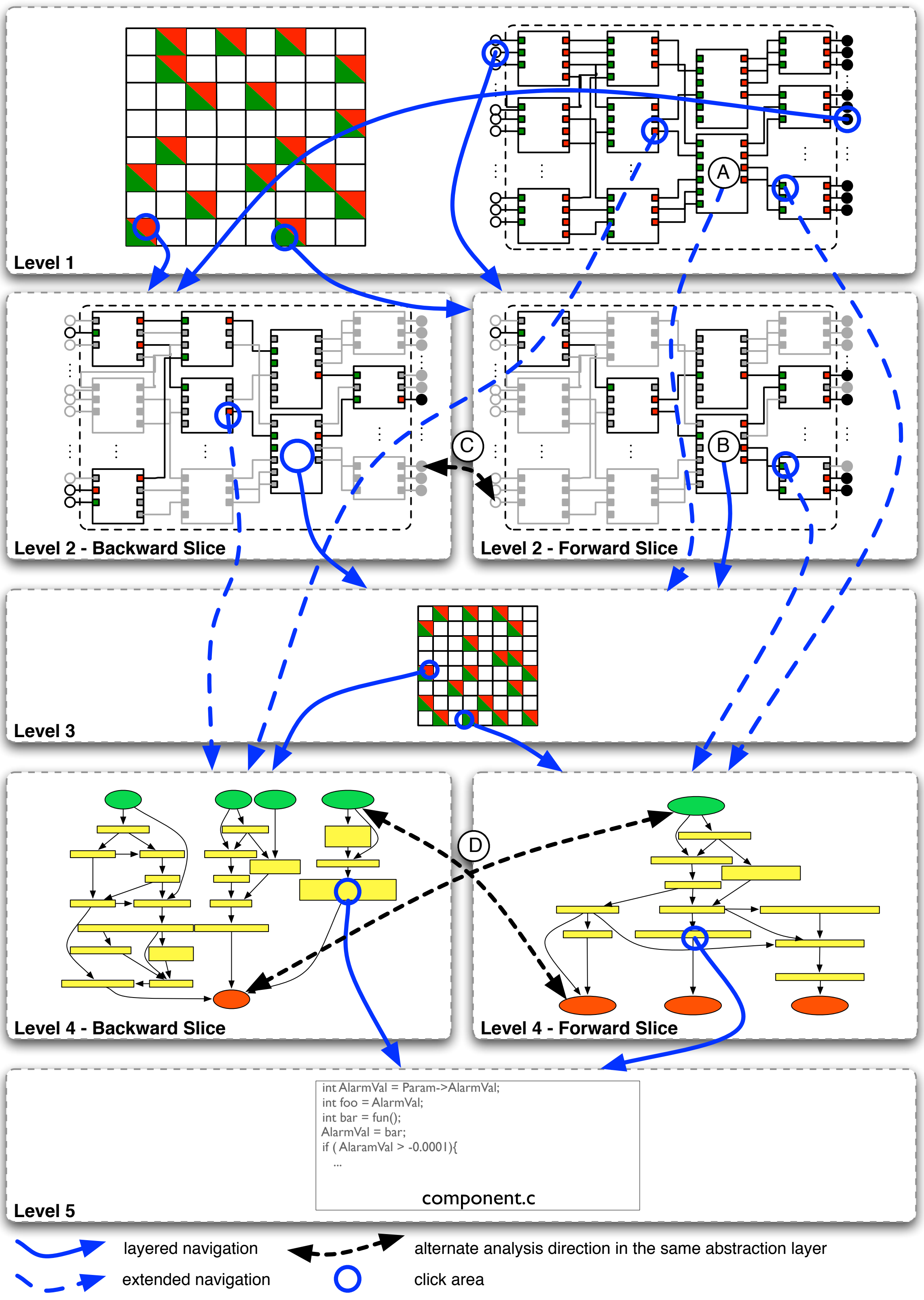
Comprehension is arguably the most fundamental prerequisite to any development and maintenance task. Considering the aforementioned difficulties in the comprehension of large-scale component-based systems (and also considering the top priority requirements of our industrial partner), we intend to utilize the extracted information flows as a means to address comprehension challenges. We devise a number of abstractions over the extracted information flows, and visualize each abstraction layer, ideally, in the most intuitive and succinct manner.

Although we have the advantage of conducting *system-wide* slices in component-based systems, we analyze the information flows in two scopes: system-wide, and component-wide. Our homogeneous model of the system is capable of slicing at any program point, however, we focus on inputs and outputs at both component and system level. Apart form that, the dependencies in our homogeneous model can be traversed in both directions of forward and backward and this characteristic helps us to create two perspectives over the information flows: *forward information flows*, and *backward information flows*.

To avoid the computational overheads of computing slices (especially in large-scale systems) each time we are about to analyze a specific information flow, we enrich the homogeneous model with the knowledge about the abstracted information flows. Such modifications to the dependence graph are proven to be trivial, considering the flexibility and extensibility of our modelware. The newly added information can be, and is indeed, used to iteratively and incrementally enrich the homogeneous model.

All the resulting information flows (i.e., system or component-wide, and forward and backward) are presented to the user in a single package, which is also integrated to the components' source code. However, the toolset is designed in a way that the user can choose the level of abstraction according to his/her needs, and therefore, it can be used by various system stakeholders (e.g., safety experts). See Figure 6 for an example depiction of the various visualization layers and possible navigations amongst them.

[6]Identification of information flows using program slicing is not unique to our project, and has been put to test in other contexts and programming languages as well [90] [91].



```
int AlarmVal = Param->AlarmVal;
int foo = AlarmVal;
int bar = fun();
AlarmVal = bar;
if ( AlaramVal > -0.0001){
   ...
```

component.c

Figure 6: The navigation structure of the visualizations. Every visualized element, except connection arrows, functions as a hyperlink to a target visualization (markers A-D are explained in detail in Paper III).

3.2.3 Change Impact Analysis in component-based product families

Large-scale software systems are not the result of the work of a few developers over a course of weeks. Rather, they are often developed by tens or hundreds of engineers and constantly maintained for many years. Consequently, it is not practical for the developers to understand and remember the dependencies between different systems elements just by relying on their cognitive power and memory. As pointed out in Section 2.1.6, having the necessary tool support to help developers identify such dependencies in software systems is highly beneficial, and is commonly referred to as Change Impact Analysis (CIA). CIA could be put to both prospective and retrospective usage scenarios: (1) what software elements need to be changed if a certain element is to be modified?, and (2) what software elements need to be changed now that a certain element has been modified?

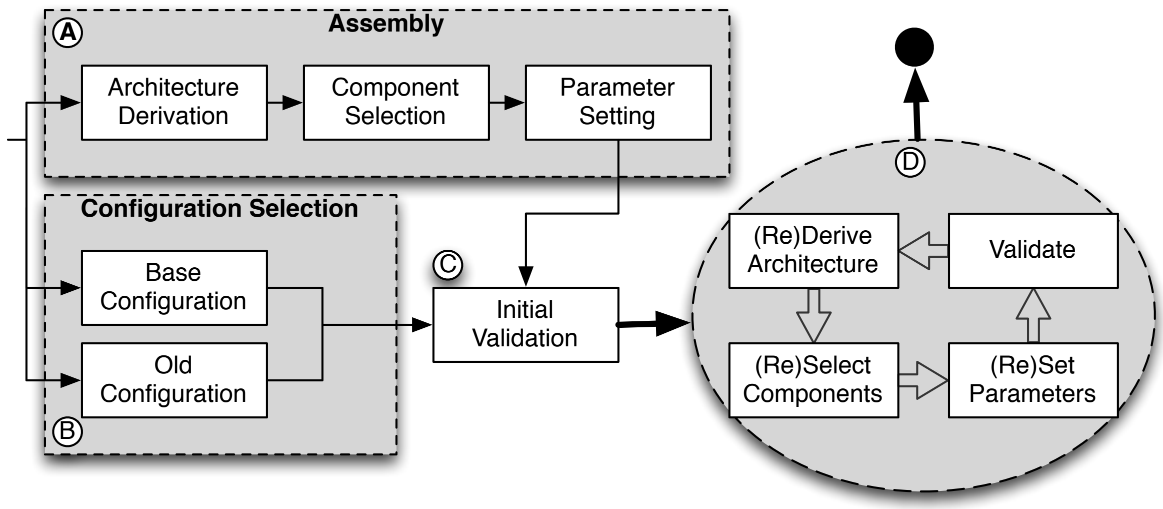


Figure 7: Product derivation process in product families, using reusable and adaptable components.

In large-scale component-based systems, which might constitute thousands of components each developed by a different team, the need for an efficient CIA mechanism is arguably more than other systems. Apart from the *size* of industrial component-based product families, it is their development process that calls for efficient CIA mechanisms. Deelstra et al. describe the *product derivation* process in component-based product families as a two phase process [82]:

Initial phase to build a skeleton product by either assembling a subset of the shared product family assets (Figure 7, Marker A), or by selecting a closest matching existing configuration (Figure 7, Marker B). An *initial validation* might conclude this phase by checking whether the skeleton implementation "sufficiently implements the desired product" with the available shared assets (Figure 7, Marker C).

Iterative phase in which software components and/or product configuration is modified and validated against costumer requirements until the product is deemed ready (Figure 7, Marker D).

Following the same process description, there are two main sources of component *evolution* in such product families: (1) once a new product is derived from the core components, changes are required to adapt the reused components to product-specific requirements; 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 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. To exemplify the monumental maintenance overheads of this process we can point out to the case of our industrial partner, Kongsberg Maritim, which assembled a designated *retrofit team* whose task was to take an exiting (deployed) product in the product family and update it to the latest revision of the shared components, just to escape the complications of dealing with several working versions of several hundred shared components. Correctly updating the product family (and the existing deployed systems) requires a thorough understanding of the potential impact of such a change.

Using CIA the developers can rely on tool support to ease the evolution process to some extent. We utilize the aforementioned cross-component slicing mechanism as a means of conducting CIA in component-based systems. To this end, we enhance the system-wide system dependence graph described in Paper I, in a way to efficiently accommodate all members of a large-scale component-based product family. 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 [4] 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 8, tag A). 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.

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 8, tag B).

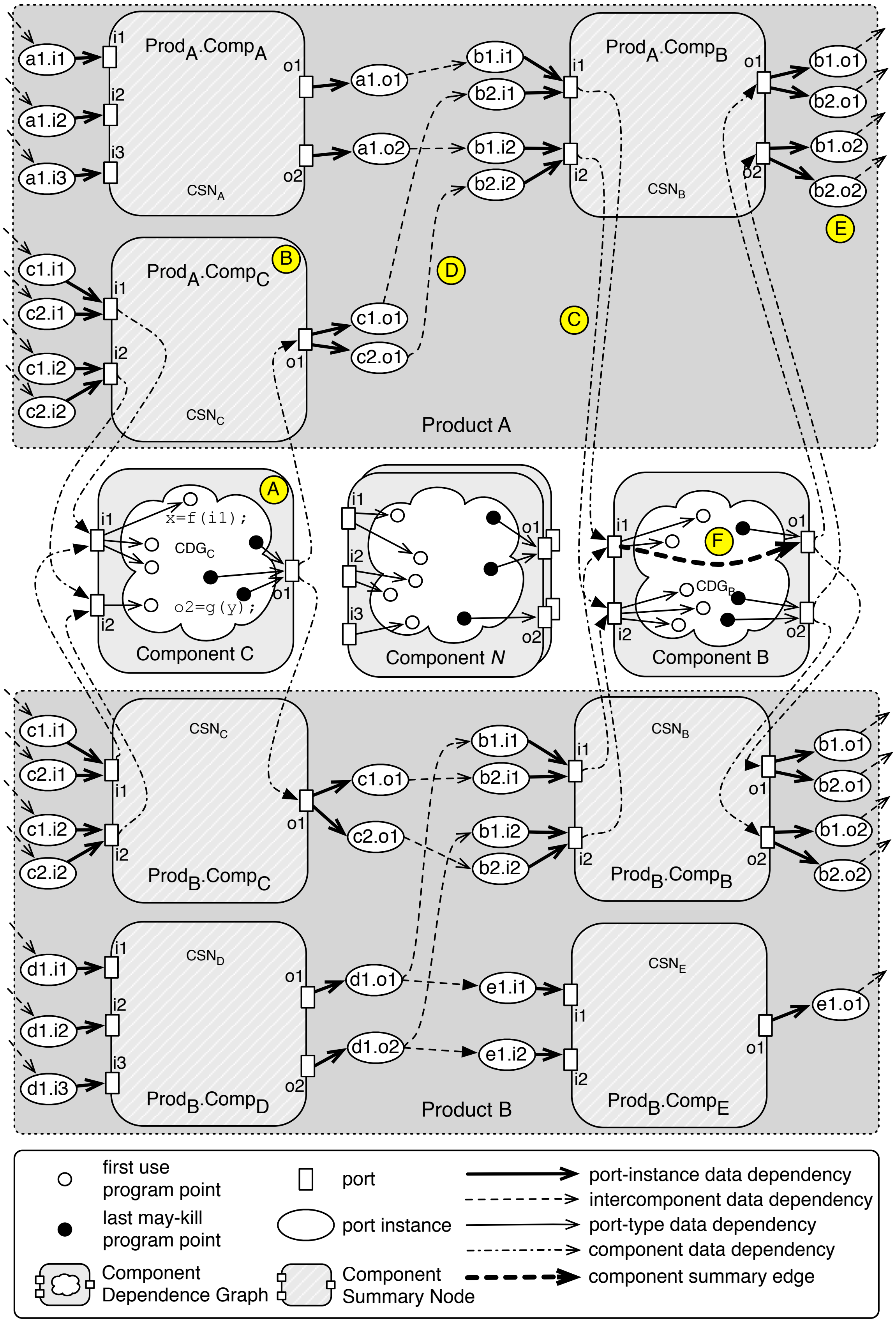


Figure 8: Family Dependence Graph (tags A–F are explained in the text)

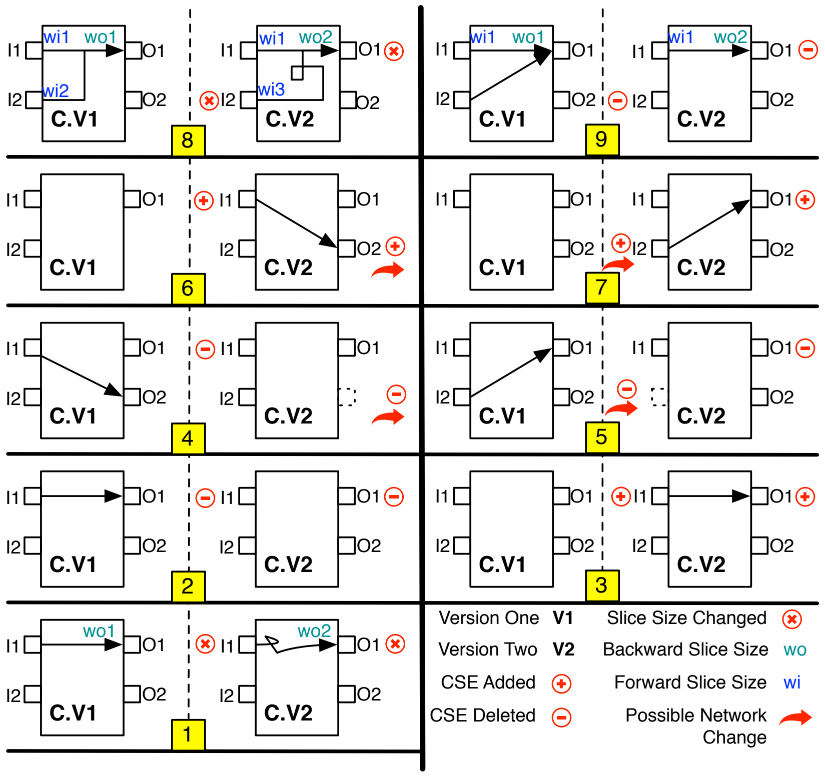


Figure 9: IIS cases and propagation of ripple effects. See Paper III for detailed discussions.

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 8, 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 8, 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.

The union of PSDGs for all members of the product family forms the FDG.

We conduct CIA in 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 overall steps are as the following:

1. Detect the Change Set (CS): this is accomplished using text-based analysis and checking against fine-grained CDGs.
2. Find the Initial Impact Set (IIS): requires slicing through the fine-grained CDG of the *updated* component, with each *component* output port as the slicing criterion. However computationally expensive is this step, it takes place within the low-scale boundaries of individual components.
3. Find the Final Impact Set (FIS): requires forward and backward traversal of the FDG all *system* input-output pairs whose information flows have been affected by the source modification are found. This traversal is very cost-effective using coarse-grained summary edges in the FDG. The propagation scenarios are defined based on the extent and type of the seeding IIS (see Figure 9 for a depiction of IIS varieties).

The details of each step, as well as evaluation results, are described in full detail in Paper III.

3.3 Cross-language program analysis: a systematic literature review

In the 1990s, at least one third of the applications developed in USA were known to utilize two languages, and 10% of the applications were estimated to use three or more languages [92]. Nowadays, these percentages have grown extensively, and it is no longer surprising to see a team of software engineers use up to 30 different programming, scripting, markup, and configuration languages to build large-scale software systems [93]. However cacophonous it may sound, this large variety in programming languages is a natural consequence of the absence of a "silver bullet" programming language: barely surprising. If utilized wisely, variety in the choice of programming languages can be highly beneficial for the development of software systems. On the other hand, this language heterogeneity complicates most system-wide tasks in the evolution of such *multi-language* systems, as *cross-language* dependencies and interactions are substantially more difficult to identify, comprehend, and manage. In the last part of this thesis, we intend to provide a basis for the improvement of software evolution of multi-language systems, by assessing the state of the art in cross-language program analysis, and discussing the implications for research and practice.

We define *cross-language program analysis* as the analysis of multi-lingual systems as a single entity; not only covering artifacts of several languages individually, but also incorporating the structural or behavioral relations that are realized through *inter*-language interactions *across* artifacts. Figure 10 is a symbolic depiction of cross-language program analysis, and its distinction criteria from single-language and multi-language analysis methods. To exemplify cross-language analysis, we can point out Paper I,

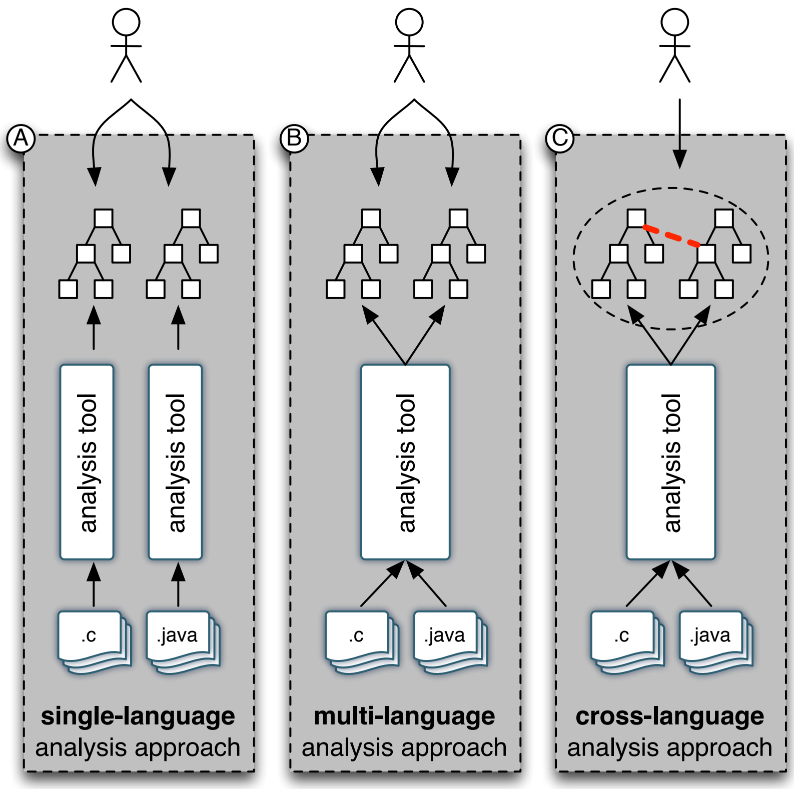

A
B
C
analysis tool
analysis tool
analysis tool
analysis tool
.c
.java
.c
.java
.c
.java
single-language analysis approach
multi-language analysis approach
cross-language analysis approach

Figure 10: Single-, multi-, and cross-language analysis approaches applied on multi-language systems. Only the cross-language analysis approach covers the interaction of multiple languages. See Paper IV for a more detailed discussion, following markers A, B, and C.

Paper II, and PaperIII in this thesis which track system-wide information flows through a combination of C modules and a third party component composition framework.

3.3.1 Conduct of the survey

With respect to investigating the available literature, the general scope of this study can be described as:

- *Population:* Published scientific literature reflecting on cross-lingual program analysis.
- *Intervention:* Devising new and/or applying cross-language program analysis methods.
- *Outcomes:* The extent of the studied cross-lingual relations, and the languages involved.
- *Experimental designs:* No restrictions. All primary studies that concern a relevant intervention are accepted on the condition that they demonstrate their relevance

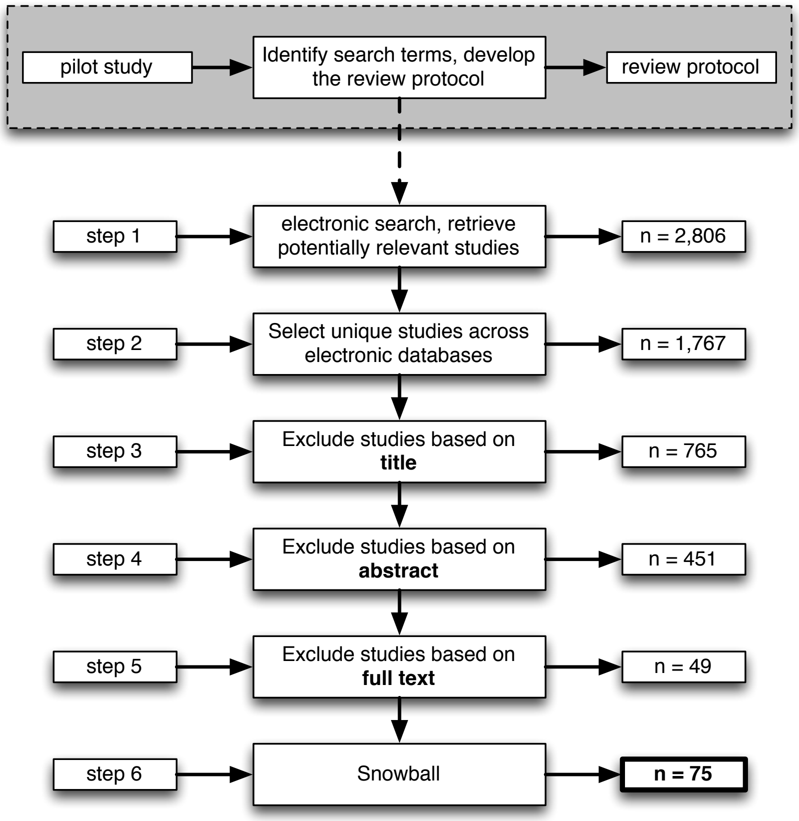

pilot study
Identify search terms, develop the review protocol
review protocol
step 1
electronic search, retrieve potentially relevant studies
n = 2,806
step 2
Select unique studies across electronic databases
n = 1,767
step 3
Exclude studies based on **title**
n = 765
step 4
Exclude studies based on **abstract**
n = 451
step 5
Exclude studies based on **full text**
n = 49
step 6
Snowball
n = 75

Figure 11: The study selection process, and the number of primary studies at each selection stage.

with enough objective data. Our quality assessment criterion is whether the paper provides an answer for one or more of our research questions.

Following Kitchenham's guidelines for conducting systematic literature reviews in software engineering [78], we conduct our study as a series of discrete steps, which are collectively called the systematic review protocol. The protocol entails: (1) the research questions, (2) the search strategy, (3) the study selection (i.e. inclusion and exclusion) criteria, (4) data extraction procedures, and (5) data synthesis methods. The review protocol itself is defined after several iterations of an initial pilot study to repeatedly test and improve all the aforementioned items until the possibility of researcher bias is reduced to a minimum. One key feature of a review protocol is that it should be well-documented so that the "readers can assess the rigor, completeness, and repeatability of the process" [78]. Therefore, Paper IV presents the review protocol, and the key decision points that lead to the protocol, in great detail.

Our search strategy consists of two consecutive stages: (1) searching seven well-established online digital libraries, and (2) manual snowballing. The sole criterion to determine the relevance of the gathered primary studies is whether they contain a non-trivial pertinence to the topic of cross-language program analysis (as defined earlier in this thesis). Any paper devising, experimenting, surveying, or advocating cross-language program analysis is considered relevant, even though it might lack some aspects of a full-fledged study, such as evaluation. We judge the papers, in respective steps, by their title, abstract, and full-text until their relevance to our review can be fully established. Our selection process is based on *exclusion*, rather than inclusion: i.e. at each step only those studies that are *clearly* not fit for our review are filtered out, and the rest are left for more detailed judgments (Figure 11). All *borderline* cases, or cases in which either of the two authors had doubts about, were double checked separately by both authors and discussed until consensus was reached.

All identified papers were subject to an in depth analysis to answer the following eight research questions:

RQ1: What approaches have been used for cross-language program analysis?

RQ2: What fact extraction methods are common in cross-language program analysis?

RQ3: What types of facts are typically extracted for cross-language program analysis?

RQ4: What internal representations of software artifacts are used?

RQ5: What higher level goals are targeted using cross-language program analysis?

RQ6: Which languages and types of software artifact have been analyzed?

RQ7: Which technological domains attracted most attention in literature?

RQ8: How rigorously are newly proposed approaches tested and evaluated?

To collect the information that is needed to answer our research questions, we classify the studies along several criteria. To guide this process, we developed a data extraction table in the pilot study, whose columns are derived from the research questions. To avoid ad hoc interpretation upon data collection into this table (and to minimize the effects of terminology mismatch between reviewers), we chose to limit the options for answering to Y/N check-boxes, or selection from a limited set of options (using drop-down lists) over free text answers whenever possible. We used the pilot study to define the taxonomy of possible answers by open and axial coding techniques from grounded theory [94]. For most columns the number of alternatives in the taxonomy stabilized early in the pilot study, and the taxonomy of possible answers were succinctly defined. For other columns we largely remained loyal to the original terminology in the primary study to avoid researcher bias as much as possible.

Data synthesis and presentation were fairly straightforward for data columns with small taxonomies. For other columns, however, we had to employ another step of axial coding, grouping related concepts into more general abstractions [94]. This synthesis was iteratively conducted along both axes of the table: across co-related primary studies, and across the concepts in the taxonomy. Using study-specific terminology during

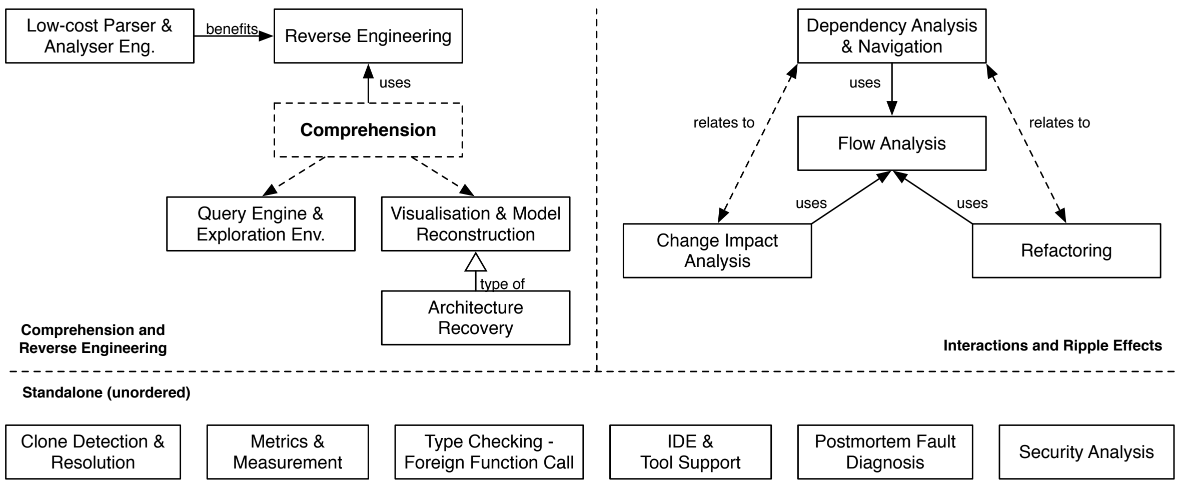

Low-cost Parser & Analyser Eng.
benefits
Reverse Engineering
uses
Comprehension
Query Engine & Exploration Env.
Visualisation & Model Reconstruction
type of
Architecture Recovery
Comprehension and Reverse Engineering
Dependency Analysis & Navigation
uses
Flow Analysis
relates to
relates to
uses
uses
Change Impact Analysis
Refactoring
Interactions and Ripple Effects
Standalone (unordered)
Clone Detection & Resolution
Metrics & Measurement
Type Checking - Foreign Function Call
IDE & Tool Support
Postmortem Fault Diagnosis
Security Analysis

Figure 12: Overview of the study goals that were identified, and their relations.

data extraction and creating cross-study abstractions during data synthesis (when the informative value of each individual concept became more clear) allowed us to make informed decisions about balancing the line between precision and conciseness.

3.3.2 Summary of the results

Our investigation identified **75** relevant papers, published between 1995 to 2014, in 37 different publication channels. The majority of primary studies were published in conferences and workshops (71 of 75, i.e., 95%), while only 4 studies were published in scientific journals. Having published almost half of all relevant papers, five conferences and workshops clearly stand out as premier channels for the research on cross-language program analysis: SANER[7], ICPC/IWPC, ICSM, WSE, and SCAM.

Upon analyzing the content of the studies, 10 publications were removed from further analysis as their content is largely repeated in other publications of the same author. In cases of highly similar publications, to avoid publication bias, only the most complete paper is accounted. The remainder of the SLR reflects on the results of **65** unique primary studies, which fall into the following broad categories:

1. technology papers (58 studies, 89%)

2. position papers (7 studies, 11%)

Technology papers propose a new, or evaluate an existing analysis method. Position papers contain general views that are independent from any specific method.

Within the 58 technology papers, we identified 16 distinct analysis goals. Figure 12 depicts the studied goals, as well as their interrelations. *Reverse Engineering* is the practice of analyzing a system to gain desired information, and generally benefits further

[7]In 2014 WCRE and CSMR merged into the SANER conference. Relevant publications in either of the conferences before the merge have been accounted for SANER in here.

comprehension attempts. Comprehension can either be addressed using graphical *Visualizations and Model Reconstruction*, or by providing the user with non-graphical *Query Mechanisms* to explore the system interactively. *Architecture Recovery* is a specialized type of model reconstruction and visualization, whose output is aimed at the abstract architectural level. A number of studies put a special price on *Cost-effective Parser and Analyzer Engineering*, which in turn facilitates day-to-day reverse engineering needs. We recorded considerable commonalities between the topics of *Dependency Analysis and Navigation*, *Change Impact Analysis (CIA)*, and to a lesser extent with the topic of *Refactoring.* Some form of *Flow Analysis* on data and/or control across multiple languages is typically used do drive these analyses, but flow analysis was also observed as an independent primary goal in some of the selected studies. A number of studies have paid exclusive attention to *Foreign Function Calls*, mainly by providing cross-language *Type Checking* facilities. Although most authors have acknowledged the importance of tool support, only few have pursued the implementation of their work in native *IDEs* or as *Generic Tool Support.* Established techniques for *Software Metrics*, *Fault Diagnosis*, *Clone Detection and Resolution*, and *Security Analysis* have been adapted in the context of cross-language analysis. Although the aforementioned goals are distinct enough to form respectable groups, there are noteworthy overlaps among the problem space and the findings of the respective studies. For example, most studies targeting model reconstruction also have a respectable contribution to basic fact extraction and reverse engineering techniques.

Our study identifies the primary analysis goal of each study, in addition to marking studies with an additional "secondary" goal. We highlight some of the more frequently visited research goals in here:

1. *Comprehension* is the most sought after goal, with 15 papers (26%) having it as their primary goal, either using a text-based query mechanism (4) or some form of graphical visualization (9) or view reconstruction (2).

2. Considering auxiliary and primary goals together, one-third of studies has pursued *graphical visualizations* (9 + 9 + 2 = 20, 34%).

3. *Dependency identification and analysis* is the second most popular target (9, 16%).

4. With *reverse engineering* as prerequisite to *comprehension* in our classification, we see 31% of studies have investigated ways to *extract* and *abstract* cross-lingual facts. Four additional papers (7%) focus on making reverse engineering more cost-effective.

5. Considering auxiliary and primary goals together, 10 papers analyze cross-lingual flow of control or data (17%).

6. Although several studies contribute task-specific tools, only two studies (3.4%) aim at holistic tool support comparable to a general-purpose IDE.

The aforementioned data exemplifies a portion of the results of our SLR, with respect to RQ5. To avoid repetition, as well as over-simplification of our multifaceted results, we refer interested readers to Paper IV for further objective discussions on the research questions.

Based on our -to some extent subjective- interpretation of the primary findings of the research questions, a set of discussion points for future research are presented in the final section of the SLR. Here follows a summary of the highlights:

There are several fact points in our synthesized answers to the research questions, that when put together convey a sense of immaturity regarding the state of the art in cross-language analysis research. The abundance of studies with no or only proof-of-concept implementations, the non-negligible number of short papers, the low number of journal articles, and the light-weight evaluation conducted in many studies are some of the symptoms. We argue that the limited occurrence of cross-references among the primary studies, in addition to the aforementioned observations, indicates a shortage of incremental and community-driven research and evaluation initiatives (e.g., by means of a “bake-off” or analysis challenge at a workshop or conference, were participants apply their approach on a common subject system and compare and possibly integrate their results).

The considerable amount of language- and technology-specific analyses, together with a noticeable lack of (semi-)generic methods are clear signs that *generalizablity* is a major challenge as well as a potential key breakthrough for the future of cross-language program analysis. Considering the high pace of technology and language development, research on highly adaptable and generalizable methods might be the only viable approach to close the gap between industry-quality tool support and the growing needs in software maintenance. This need for more generic approaches makes us doubt the trade-off of developing techniques that are highly dependent on heuristics, over investing in sound theoretical frameworks as a basis for future generations of analysis tools. While knowledge repositories have been used in half of the relevant primary studies, there is little evidence that the state-of-the-art is capable of accommodating our future needs, given the sporadic use of each repository technology. Model-driven technologies, supported by standards and open-source movements, may change this trend, but still need to stand the test of time.

4 Research Methodology

Various research methods have been applied in different parts of this research initiative. Here follows a brief description of the methods, and areas of activities leading to the results presented in this thesis.

4.1 Investigating the industrial context

This research project was conducted with close collaboration with our industry partner, Kongsberg Maritime (KM), which is a leading technology provider for a extensive range of products for on- and offshore oil and gas platforms, commercial maritime, subsea installations, fisheries and naval vessels, etc. As a producer of large-scale cyber-physical systems, KM needs to apply the necessary resources to develop the software needed to run their end-to-end systems. Moreover, as the operation of such systems are **safety-critical**, they also need to spend additional resources on quality assurance, testing, and certification of each product before they can be deployed on the target installation plant. Developing industry-quality software is time consuming, and it is hardly surprising that substantial amount of time and resources are needed to drive the software production and certification processes at our industry partner. KM is one of the largest in-house software development bodies in Norway, and the subdivision in collaboration with us is in charge of developing and maintaining a safety (sub-)system used in fire and gas detection, process shutdown, and emergency shutdown systems.

The initial phase of our collaborations with our industry partner were dedicated to understand the context: the general requirements of the overall system, the role of software in the system, the devised software architecture, the quality assurance processes ahead of software development, and more importantly the areas in which quality assurance either fell short or were faced with extremely time-consuming and expensive tasks. We had a number of meetings, and were given access to several technical and non-technical documents, as well as an adequate portion of their source code. In a nutshell, we found that KM had (1) a portfolio of similar products, (2) built upon a highly scalable, and strictly component-based architecture, (3) developed in MISRA C, and configured using a propitiatory XML-based component-configuration framework, (4) shared and maintained incrementally across more than one department in KM, (5) with unwieldy quality assurance and certification processes whose key concern is to ensure correct passage of information from systems inputs to the outputs. We also found out that it is not only software (component) developers who are involved, but also safety experts, testers, and product developers who combine and configure the components for each new installation. Moreover, it became clear that due to extensive dependencies on specific hardware and processing units to run the system, and due to the prohibitive cost and risk of running our analysis on running systems, our solution space is bound to static analysis.

Using this information, we performed domain analysis, in several iterations, to characterize viable solutions that could facilitate the quality assurance process at KM. After each iteration, the results were discussed with our industry partner to ensure a correct understanding of the problem domain, and fitness of the proposed solution to the expectations of our partners in terms of scope (e.g. system-wide or component-wide), level of detail for various user groups, and tool support. Paper I, Paper II and Paper III each describe and address an aspect of the problem domain.

4.2 Literature review and tool evaluations

Having the attributes of the problem in place, we conducted a review over the available literature and the state-of-the-practice to find a viable solution to the problem at hand. A number of key constraints in the problem domain were crucial in how we navigated a vast body of literature:

1. slim chances of executing the (sub-)system, which rules out dynamic analysis
2. complying with component-based systems
3. involving at least two types of development artifacts, namely source code and configuration artifacts
4. the crucial need for ensuring correct dependencies between inputs and outputs

In the review, we explored and evaluated several third-party tools which could be reused to deliver (a portion of) the solution space. Software visualization tools, component configuration frameworks, adaptable compilers, model recovery tools are some of the explored areas. Of particular importance was investigation over the existing technology space on grammarware and tools providing dependence graphs from source code. Likewise, our review over the available literature on program slicing proved crucial for our basic approach in detecting component- and system-wide information flows. As also mentioned in Paper I however, we found very little existing work on multi-language software systems pertinent to our problem domain.

4.3 Devising a model-based approach to enhance grammarware

Having identified the aforementioned gap in literature and tool-support, we defined our approach to analyze information flows through and across components of a large-scale software system. Our approach is devised in a way to reuse the existing mechanism and tools as much as possible. This characteristic is vital in an industry-driven research collaboration, where practicality and plausibility of the solutions is a key factor.

We built on the seminal work of Horwitz et al. [4], and used program slicing as a means to find component-wide information flows. One benefit of this approach is the possibility of reusing well-established tool support, namely CodeSurfer [27]. To fill the gap in slicing in multi-language component-based systems, we developed a model-based approach to leverage the extensive flexibility of models and the emerging tool support in this domain, such as Eclipse Modeling Framework. Paper I explains our proposed approach for cross-component program slicing, adapted to the notion of information flows in our case study system. Paper II presents our solution for visualizing system-wide information flows, to fill the gap in tool-assisted comprehension for both software developers, and safety experts in Kongsberg Maritime. In Paper III, we present our solution for conducting change impact analysis in a family of products, which is required for the long-term maintenance and evolution of large-scale component-based systems.

4.4 Empirical studies and expert-based evaluations

Each stage of our research collaboration resulted in a prototype tool to demonstrate the applicability of our approach. This enabled us to provide hands-on experience for our industry partner, and address deficiencies based on the received feedback in a number of iterations.

In particular, we conducted two rounds of evaluation with six participants, to evaluate the core functionalities of FlowTracker as well as the *overall* questions to capture a holistic view of the positive and negative aspects of FlowTracker usability. Evaluation sessions were conducted independently of one another, and the results were aggregated after all participants finished the evaluation. After an initial introduction of FlowTracker, the participants had a training session with three hands-on exercises. The evaluation session were driven by a structured interview following a questionnaire, consisting of both closed questions and a number of open (discussion) questions. The gathered feedback gave us deep insight into the usability issues of the provided tool set, which lead to an improved version afterwards. Paper II presents the results of two expert-based rounds of evaluation.

4.5 Systematic literature review

In the final stage of this research initiative, we intend to provide a basis for the improvement of software evolution of multi-language systems by assessing the state of the art in cross-language program analysis. Therefore we conducted a systematic review over the available literature on cross-language program analysis, to gather the diversity of the applied techniques, application domains, programming languages, and on the strength of the findings. The findings of the SLR gave us enough insight to categorize the applied approaches, identify possible trends in research, and identify possible implications for research.

This study was conducted following the established guidelines of conducting Systematic Literature Reviews, described in [78]. We provide extensive documentation over the applied process in Paper IV; a necessary measure for the purpose of repeatability in systematic literature reviews.

5 Summary of Results

In this section we list the main results of our research initiative. Detailed discussions are covered in the four papers included in this thesis.

5.1 Paper I

Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems. *Amir Reza Yazdanshenas, Leon Moonen.* Published in the proceedings of the 27th IEEE International Conference on Software Maintenance, 2011.

This paper describes our case study system at Kongsberg Maritime and explains the substantial challenges ahead of quality assurance and certification processes. It entails the main attributes of the subject system, as well as the main characteristics of viable solutions and usage scenarios. We clarify the root cause that hinders our industry-partner to perform any system-wide analysis in the subject system at design time.

The following contributions can be listed with reference to Paper I:

- Sets the case for system-wide, or cross-component, program analysis for the purpose of quality assurance. With no loss of generality, the case is set in reference a range of large scale safety-critical systems at our industry partner.
- Clarifies where state-of-the-practice falls short in conducting system-wide analyses in multi-lingual component-based systems. Highlights the urgent shortage of tool support in cross-component analysis, despite wide acceptance of component-based design methods.
- Argues the significance of information flows in our industry partner as well as similar systems with large-scale networks of data-intensive components.
- Discuses possibilities of extending traditional intra-component analysis methods to cross-component analysis methods, capable of producing system-wide insights.
- Introduces the notion of inter-component dependence graph (ICDG).
- Devises a method to cost-effectively combine traditional (intra-component) program dependence graphs and inter-component dependencies: yielding homogeneous system-wide dependence models from a system's heterogeneous source and configuration artifacts. The method solely uses design time artifacts (static analysis).
- Builds upon the foundations of OMG's Knowledge Discovery Metamodel (KDM) and demonstrates how this standardized and modeling specification could be used to produce language-independent *intermediate representations* amenable for program analysis.

- Applies program slicing across heterogeneous software artifacts, based on a homogeneous internal representation of a software system.

- Reports on building a prototype tool which has been successfully applied on our case study system. Discusses positive and negative points of the proposed approach based on a number of “in vitro” evaluations.

- Adds a point of reference to the use and extension of KDM in an industrial setting.

5.2 Paper II

Tracking and Visualizing Information Flow in Component-Based Systems. *Amir Reza Yazdanshenas, Leon Moonen.* Submitted to Journal of Information and Software Technology, 2015. This is an extended version of the paper that was published in Proceedings of the 27th IEEE International Conference on Program Comprehension (ICPC 2012).

Paper II continues with the same case study as in Paper I, however, focuses on comprehension issues rather than quality assurance. Nevertheless, program comprehension is the fundamental prerequisite for arguably any activity in program maintenance and evolution, including quality assurance. Moreover, in the case of our industry partner tool-assisted program comprehension can be a direct input to demonstration of safety and certification processes.

Paper II offers contributions on the following areas:

- Sets the case for additional comprehension challenges across heterogeneous software artifacts.

- Builds upon the achievements of Paper I and successfully uses forward and backward slicing to navigate the flow of information across heterogeneous software systems: starting from inputs to outputs and vice versa.

- Devises a hierarchy of five interconnected views to support the comprehension needs of various stakeholders, such as safety domain experts and software developers. These hierarchy of visualizations are devised in a way to enable the user to “zoom in and out” over the information flows on demand.

- Proposes traditional matrix-based visualizations to succinctly present dependencies among pairs of input and outputs, both at component and system level.

- Devises *System Information Flow* and *Component Information Flow* abstractions to visualize the exact flow of information throughout the system, with different levels of granularity.

- Presents *Component Information Flow* capable of visualizing conditions that control the information flow. Presents a method to reduce the visual cluttering of large dependence graphs by building concise summary edges from information flows.

- Presents navigable visualizations with use of active hyperlinks, allowing both systematic (e.g. top-down) as well as opportunistic navigation scenarios.

- Presents a method to directly connect visualizations to exact positions in source code as a means of traceability and to minimize user disorientation.

- We implement our approach in a prototype tool and present two qualitative evaluation studies on the effectiveness and usability of the proposed views for software development and software certification. The evaluations provide insight how users would best benefit from visualized information flows, how intuitive they find each visualization type, and areas in which presented visualizations fall short of users' expectations.

5.3 Paper III

Fine-Grained Change Impact Analysis for Component-Based Product Families. *Amir Reza Yazdanshenas, Leon Moonen.* Published in the proceedings of the 28th IEEE International Conference on Software Maintenance, 2012.

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. Paper III 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.

In short, the following contribution points can be listed for Paper III:

- Builds upon the achievements of Paper I, and uses static program slicing to support change impact analysis in heterogeneous component-based systems.

- Devises a method for constructing fine-grained family-wide dependence graphs (FDG) from the source and configuration artifacts of a component-based product family.

- Devises a method to conduct change impact analysis based on a taxonomy of change types. Uses fine-grained and well-established text-based analysis methods as a seed for conducting change impact analysis (Change Set).
- Defines the notion of Initial Impact Set (IIS) as the set of modifications to a component's interface on a given change set. Adapts fine-grained analysis of program dependence graphs to calculate the IIS with maximum precision.
- Computes the Final Impact Set (FIS) by propagating the IIS throughout a family of products via traversal of lightweight and coarse-grained dependencies.
- Discusses an optimum balance between the precision of IIS and the required efficiency of FIS and argues why components' interface is the crucial breaking point between the two.
- Presents the transformations needed to achieve homogeneous models of product families and the lessons learned in implementing a prototype tool based on a standardized language-independent metamodel (KDM) to enhance interoperability and generalizability.
- Proposes a ranking scheme based on approximations of the scale of impact using program slice sizes.
- Presents the result of an "in vitro" evaluation.

5.4 Paper IV

Cross-language program analysis for the evolution of multi-language software systems: a systematic literature review. *Amir Reza Yazdanshenas, Leon Moonen.* Submitted to Journal of Software: Evolution and Process.

In Paper 4, which concludes this thesis, we provide a basis for the improvement of software evolution of multi-language systems by assessing the state of the art in cross-language program analysis and discussing the implications for research and practice.

The following contribution points can be listed for Paper IV:

- Defines *cross-language program analysis*, and highlight its distinction from single- and multi-language analysis methods.
- Identifies 75 papers addressing cross-language program analysis.
- Identifies 58 technical and seven position papers and provides in-depth analysis customized to the characteristics of each group.
- Presents a mapping study over the identified primary studies, highlighting the more popular publication challenges relevant to cross-language analysis. Presents the

distribution of conducted studies over a continuous course of 20 years. Identifies the more efficient online digital libraries, with extensive coverage of relevant primary studies.

- Classifies the studies based on several criteria, including their purpose (why), the adopted or suggested approach (how), the information leveraged in each programming language or artifact (what), and the conducted evaluation (quality).
- Extracts data from all primary studies and synthesizes data to provide in-depth answers to eight research questions (see Paper 4 for the list of research questions). The answers objectively reveal several trends in the relevant body of research and systematically put the available knowledge into perspective.
- Based on the research questions, presents a set of implications for research and the interested community. Here follows a short list:
  - Argues for research on more language-generic analysis approaches.
  - Explains why reducing dependence of analysis methods on "heuristics" can be beneficial.
  - Objectively reveals that the state-of-the-art in *knowledge repositories* has been far from conclusive with respect to the research on cross-language program analysis.
  - Highlights the potential impact of generic name-resolution mechanisms.
  - Reveals a shortage of industry-driven research on cross-language program analysis.
  - Reveals a shortage of incremental research among the interested community and proposes possible initiatives.
  - Highlights the lack of common terminology, and possible implications.
- Contributes to the discipline of conducting systematic literature reviews in the domain of software engineering by providing a number of lessons learned while conducting the study. Explains how the research protocol developed incrementally by conducting a pilot study.

6 Future Directions

Based on the results so far and our gained insight in this thesis, we foresee a number of directions for future research.

The first extension could be the adaptation of our cross-component slicing tool to several other programming languages, and component composition/configuration frameworks. Although the gist of many component container design principles are essentially the same, there can be numerous wide-spread component containers which are radically different from our case study system. For instance component interactions based on API calls (call-and-return) might not be amenable to information flow analysis as much as data-oriented component frameworks that interact by sending and receiving data on a set of well-defined data ports. Possible implications of more moderns design principles, such as Inversion of Control (IoC), need to be adapted to our model construction method, and put to test. We have developed a proof-of-concept tool connecting Java and configuration files in the Spring framework. However, we need to experiment beyond toy-example systems.

Based on the results of expert-based evaluations on visualized information flows in Paper II, we acknowledge that the overall user experience needs to be improved by adding more *on-demand* interaction facilities, such as zooming and hiding or collapsing groups of nodes. Such facilities allow users to be more selective in the amount and type of information they see and spontaneously adjust the visualizations according to their information needs at the moment. We speculate that a portion of the necessary improvement can be achieved by using a more elaborate graph viewer than what we currently use in our prototype tool, called FlowTracker. A major improvement to the future of our visualization method is integration with a general-purpose IDE such as Eclipse. This integration has the benefit of minimizing user disorientation, and avoids the common difficulties of familiarizing seasoned developers to new working toolsets. A number of extensions to FlowTracker were advised by our industry collaborators during the evaluation sessions. Namely, "visualizing multiple versions" of a system at the same time and highlighting the implied *delta* on the information flows were requested by the developers.

With respect to our devised method for change impact analysis in families of component-based systems, one immediate next step would be to empirically evaluate the precision and recall factors of our analysis in an industrial context and demonstrate applicability in real-world evolution scenarios. 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 more industry partners. One could also 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.

7 Conclusion

Heterogeneity of software artifacts poses several challenges in the development, evolution, comprehension, and quality assurance of multi-language software systems. One main source of heterogeneity in contemporary software systems is widespread component-based design principles, which is an effective tool in managing complexity of large-scale software systems by composing them from reusable parts. Another source of language heterogeneity is the (legitimate) tendency to develop each part of a system with the programming language that fits best. The urgency of challenges introduced by language heterogeneity comes into perspective once we realize that contemporary software systems are *rarely* implemented uniformly in one programming language, and delivered in one type of development artifact. This thesis contributes a number of approaches to tackle heterogeneity in component-based systems in addition to providing objective insight over the existing body of literature on cross-language program analysis.

As the first step we devise a method to build homogeneous dependence models from component-based systems. We use model-driven engineering methods to integrate traditional program dependence graphs (extracted from source code) with partial intercomponent dependence graphs (extracted from component configuration files). We use OMG's Knowledge Discovery Metamodel (KDM) to accommodate this system-wide model in our prototype tool. The resulting unified model can be traversed forward and backward to compute program slices *through and across* components to detect channels of *information flow* in the system. System-wide information flows, or relations among systems' input and outputs, are a major point-cut to verify correct behavior and correct configuration in component-based systems. The extracted information flows can be consulted in several quality assurance activities of our industry partner, Kongsberg Maritime.

Secondly, we utilized the same system-wide information flows to assist system stakeholders in comprehending component-based systems. Comprehension is of utmost value in arguably any development and maintenance task in any system, specially large-scale component-based systems. We propose a hierarchy of five interconnected views to support the comprehension needs of various stakeholders, namely safety domain experts and developers in our industrial partner. The abstractions are stacked on top of each other in a way to allow the end user to easily "zoom in and out" a given information flow; ranging from high-level traditional matrix views, to intra-component information flows presenting condition points in the source code, and finally to the source code. Using active hyper-links, the views are interconnected in a way to support both top-down, as well as ad hoc navigation scenarios. A prototype tool is delivered to our industry partner, and two qualitative evaluation studies reveal overall promising results with respect to effectiveness and usability, as well as a number of insightful feedback on the shortcomings.

Thirdly, we contribute a method to carry out change impact analysis in a component-based product families based on system-wide information flows. Change impact analysis concerns estimating the ripple effects of a change through out a system. Tool-assisted change impact analysis can substantially facilitate long-term evolutions of large-scale component-based software systems. We build upon the same model-driven engineering techniques as before and devise a method to cost-effectively build homogeneous *family-wide* dependence graphs. We devise a method to conduct change impact analysis based on a taxonomy of change types. The Initial Impact Set (IIS) is defined as the modifications to a component's interface on a given change set, and the Final Impact Set (FIS) is marked on a system(s)' interface by propagating the IIS throughout a family of products via traversal of lightweight and coarse-grained dependencies. We also propose an auxiliary mechanism to rank the impact of a change on information flows according to an *approximation* of the scale of impact using changes in program slice size.

Finally, we seek to provide a basis for the improvement of software evolution in multi-language systems by assessing the state of the art in cross-language program analysis. To this end, we conducted a systematic review over the available literature in several digital libraries and by manual snowballing. Our search revealed 75 primary studies, which were systematically put to an in-depth analysis to answer eight research questions. In the hindsight of the research questions, we reveal a number of areas in which current state of research falls short and discuss several implications for improving future research initiatives.

Bibliography

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

[2] M. D. McIlroy, "Mass Produced Software Components," in *Software Engineering Report on a Conf. sponsored by the NATO Science Committee*, ser. {NATO} Software engineering conference, P. Naur and B. Randell, Eds., vol. 1, no. October 1968, NATO. NATO Science Committee, 1968, pp. 138–155.

[3] B. Cox and A. Novobilski, *Object-oriented Programming; An Evolutionary Approach*, 2nd ed. Addison-Wesley, 1986.

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

[5] "24765-2010 - Systems and software engineering – Vocabulary," p. 418, 2010.

[6] "Working Conference on Source Code Analysis and Manipulation (SCAM)," Dec. 2012. [Online]. Available: http://www.ieee-scam.org/2012/SCAM/index.html

[7] D. Binkley, "Source Code Analysis: A Road Map," in *Future of Software Engineering (FoSE)*. IEEE, May 2007, pp. 104–119.

[8] "Working Conference on Mining Software Repositories," Dec. 2013. [Online]. Available: http://2013.msrconf.org/index.php

[9] A. Kuhn and M. Stocker, "CodeTimeline: Storytelling with versioning data," in *2012 34th Int'l Conf. on Software Engineering (ICSE)*. IEEE, Jun. 2012, pp. 1333–1336.

[10] A. Hindle, M. W. Godfrey, and R. C. Holt, "Software process recovery using Recovered

Unified Process Views," in *2010 IEEE Int'l Conf. on Software Maintenance.* IEEE, Sep. 2010, pp. 1–10.

[11] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin, "Future of Mining Software Archives: A Roundtable," *IEEE Software*, vol. 26, no. 1, pp. 67–70, Jan. 2009.

[12] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *2009 IEEE 31st Int'l Conf. on Software Engineering.* IEEE, 2009, pp. 1–11.

[13] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT Int'l Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16.* ACM Press, 2008, pp. 13–23.

[14] M. Harman, "Why Source Code Analysis and Manipulation Will Always be Important," in *2010 10th IEEE Working Conf. on Source Code Analysis and Manipulation.* IEEE, Sep. 2010, pp. 7–19.

[15] D. Jackson and M. Rinard, "Software analysis: a roadmap," in *Proceedings of the Conf. on The future of Software engineering - ICSE '00.* ACM Press, 2000, pp. 133–145.

[16] D. Strein, H. Kratz, and W. Lowe, "Cross-Language Program Analysis and Refactoring," in *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE Int'l Ws. on*, 2006, pp. 207–216.

[17] R. Pawlak, C. Noguera, and N. Petitprez, "Spoon: Program Analysis and Transformation in Java," Tech. report #inria-00071366, INRIA, 2006.

[18] S. Genaim and F. Spoto, "Information Flow Analysis for Java Bytecode," in *The 6th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, 2005, pp. 346–362.

[19] J. Zhao, "Dependence analysis of Java bytecode," in *Proceedings 24th Annual Int'l Computer Software and Applications Conf.ĊOMPSAC2000.* IEEE Comput. Soc, pp. 486–491.

[20] T. Bell, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, Nov. 1999.

[21] M. D. Ernst, "Invited Talk Static and dynamic analysis: synergy and duality," in *Proceedings of the ACM-SIGPLAN-SIGSOFT Ws. on Program analysis for software tools and engineering - PASTE '04.* ACM Press, 2004, pp. 35–35.

[22] R. Gupta, M. L. Soffa, and J. Howard, "Hybrid slicing: integrating dynamic information with static analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 370–397, Oct. 1997.

[23] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[24] L. Larsen and M. Harrold, "Slicing object-oriented software," in *Int'l Conf. on Software Engineering (ICSE).* IEEE, 1996, pp. 495–505.

[25] J. Krinke, "Static slicing of threaded programs," in *ACM SIGPLAN-SIGSOFT Ws. on Program Analysis for Software Tools and Engineering (PASTE)*, Jul. 1998, pp. 35–42.

[26] F. Ricca and P. Tonella, “Web application slicing,” in *IEEE Int’l Conf. on Software Maintenance (ICSM)*, vol. 12, no. 2, Apr. 2001, pp. 148–157.

[27] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins, “Tool support for fine-grained software inspection,” *IEEE Software*, vol. 20, no. 4, pp. 42–50, Jul. 2003.

[28] M. Weiser, “Program slicing,” in *Int’l Conf. on Software Engineering (ICSE).* IEEE, 1981, pp. 439–449.

[29] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[30] M. Harman and R. Hierons, “An overview of program slicing,” *Software Focus*, vol. 2, no. 3, pp. 85–92, Jan. 2001.

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

[32] B. Price, I. Small, and R. Baecker, “A taxonomy of software visualization,” in *Proceedings of the Twenty-Fifth Hawaii Int’l Conf. on System Sciences*, no. January. IEEE, 1992, pp. 597–606 vol.2.

[33] H. M. Kienle and H. a. Muller, “Requirements of Software Visualization Tools: A Literature Survey,” in *IEEE Int’l Ws. on Visualizing Software for Understanding and Analysis (VISSOFT)*, Jun. 2007, pp. 2–9.

[34] J. Steele and N. Iliinsky, *Beautiful Visualization, Looking at Data through the Eyes of Experts*, 1st ed. O’Reilly Media, 2010.

[35] E. Chikofsky and J. Cross, “Reverse engineering and design recovery: a taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[36] M. L. Nelson, “A Survey of Reverse Engineering and Program Comprehension,” Old Dominion University, Tech. Rep., 1996.

[37] A. Hunt and D. Thomas, “Software archaeology,” *IEEE Software*, vol. 19, no. 2, pp. 20–22, 2002.

[38] G. Canfora and M. Di Penta, “Frontiers of reverse engineering: A conceptual model,” in *Frontiers of Software Maintenance (FoSM).* IEEE, Sep. 2008, pp. 38–47.

[39] T. Kuipers, “Techniques for Understanding Legacy Software Systems,” Ph.D. dissertation, Universiteit van Amsterdam, 2002.

[40] B. Moyer, “Software Archeology. Modernizing Old Systems,” *Embedded Technology Journal*, pp. 1–4, 2009.

[41] M. Feathers, *Working Effectively With Legacy Code.* Prentice Hall, 2004.

[42] “IEEE Working Conference on Software Visualization (VISSOFT),” 2013. [Online]. Available: http://icsm2013.tue.nl/VISSOFT/

[43] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, 2007.

[44] N. Wade and M. Swanston, *Visual Perception: An Introduction.* Psychology Press, 2001.

[45] D. Moody, “The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, Nov. 2009.

[46] C. Parnin and S. Rugaber, “Programmer information needs after memory failure,” in

2012 20th IEEE Int'l Conf. on Program Comprehension (ICPC). IEEE, Jun. 2012, pp. 123–132.

[47] M. Storey, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, Jan. 1999.

[48] M. M. Ber, D. Cruz, M. Jo, and R. Uzal, "Evaluation Criteria of Software Visualization Systems used for Program Comprehension," in *Conferência Interacção Pessoa-Máquina*, 2008.

[49] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *SIGCHI Conf. on Human Factors in Computing Systems.* ACM, 1990, pp. 249–256.

[50] M. Sensalire, P. Ogao, and A. Telea, "Evaluation of software visualization tools: Lessons learned," in *IEEE Int'l Ws. on Visualizing Software for Understanding and Analysis (VISSOFT)*, Sep. 2009, pp. 19–26.

[51] R. Molich and J. S. Dumas, "Comparative usability evaluation (CUE-4)," *Behaviour & Information Technology*, vol. 27, no. 3, pp. 263–281, May 2008.

[52] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement.* Continuum, 1992.

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

[54] 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. on Software Maintenance and Reengineering (CSMR).* IEEE, 1999, pp. 130–138.

[55] H. Kagdi and J. Maletic, "Software-Change Prediction: Estimated+Actual," in *IEEE Int'l Ws. on Software Evolvability (SE)*, Sep. 2006, pp. 38–43.

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

[57] ——, "A taxonomy for software change impact analysis," in *Int'l Ws. on Principles of Software Evolution (IWPSE-EVOL).* ACM, 2011, pp. 41–50.

[58] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, pp. n/a–n/a, Apr. 2012.

[59] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495–509, Jun. 2003.

[60] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceeding of the 33rd Int'l Conf. on Software engineering - ICSE '11.* ACM Press, 2011, p. 746.

[61] "A Proposal for an MDA Foundation Model," 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?ormsc/05-04-01

[62] "Model-Driven Architecture," 2013. [Online]. Available: http://www.omg.org/mda/index.htm

[63] "Architecture-Driven Modernization," 2013. [Online]. Available: http://adm.omg.org/

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

[65] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems," *Computer Standards & Interfaces*, vol. 33, no. 6, pp. 519–532, Nov. 2011.

[66] O. M. T. Force, "OMG Meta Object Facility (MOF) Core Specification," 2003.

[67] G. Barbier, H. Brunelière, F. Jouault, Y. Lennon, and F. Madiot, "MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases," in *Information Systems Transformation: Architecture-Driven Modernization Case Studies*, W. M. Ulrich and P. Newcomb, Eds. Morgan Kaufmann, 2010, ch. 14, pp. 365–400.

[68] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley, Dec. 2009.

[69] "International ACM Sigsoft Symposium on Component-Based Software Engineering," 2013. [Online]. Available: http://cbse-conferences.org/2013/

[70] "ICSE Workshop Series on Component-Based Software Engineering," 2013. [Online]. Available: http://www.icse-conferences.org/2003/

[71] S. Mahmood, R. Lai, and Y. S. Kim, "Survey of component-based software development," *IET Software*, vol. 1, no. 2, pp. 57–66, 2007.

[72] S. P. Shashank, P. Chakka, and D. V. Kumar, "A systematic literature survey of integration testing in component-based software engineering," in *2010 Int'l Conf. on Computer and Communication Technology (ICCCT).* IEEE, Sep. 2010, pp. 562–568.

[73] O. Slyngstad, M. Torchiano, M. Morisio, and C. Bunse, "A State-of-the-Practice Survey of Risk Management in Development with Off-the-Shelf Software Components," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 271–286, Mar. 2008.

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

[75] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[76] B. A. Kitchenham, T. Dybå, and M. Jorgensen, "Evidence-based Software Engineering," in *Int'l Conf. on Software Engineering (ICSE)*, 2004, pp. 273–281.

[77] T. Dyba, B. Kitchenham, and M. Jorgensen, "Evidence-based software engineering for practitioners," *IEEE Software*, vol. 22, no. 1, pp. 58–65, Jan. 2005.

[78] B. A. Kitchenham, "Guidelines for performing Systematic Literature Reviews in Software Engineering," Keel University & University of Durham, Tech. Rep., 2007.

[79] J. Biolchini, P. G. Mian, A. C. C. Natali, and G. H. Travassos, "Systematic Review in Software Engineering," UFRJ, Tech. Rep. May, 2005.

[80] F. P. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[81] T. Mens, "On the Complexity of Software Systems," *Computer*, vol. 45, no. 8, pp. 79–81, Aug. 2012.

[82] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *Journal of Systems and Software*, vol. 74, no. 2, pp. 173–194, Jan. 2005.

[83] R. Behjati, "A Model-Based Approach to the Software Configuration of Integrated Control Systems," Ph.D. dissertation, Faculty of Mathematics and Natural Sciences, University of Oslo, 2012.

[84] R. Behjati, T. Yue, L. Briand, and B. Selic, “SimPL: A product-line modeling methodology for families of integrated control systems,” *Information and Software Technology*, vol. 55, no. 3, pp. 607–629, Oct. 2012.

[85] D. Binkley, M. Harman, and J. Krinke, “Empirical study of optimization techniques for massive slicing,” *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 1, pp. 3–es, Nov. 2007.

[86] P. Klint, R. Lämmel, and C. Verhoef, “Toward an engineering discipline for grammarware,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 3, pp. 331–380, 2005.

[87] L. Moonen, “Generating robust parsers using island grammars,” in *Proceedings Eighth Working Conf. on Reverse Engineering.* IEEE Comput. Soc, pp. 13–22.

[88] M. Wimmer and G. Kramler, “Bridging grammarware and modelware,” in *Ws. in Software Model Engineering (WiSME).* Springer, 2006, pp. 159–168.

[89] T. Reus, H. Geers, and A. Van Deursen, “Harvesting Software Systems for MDA-Based Reengineering,” in *European Conf. on Model Driven Architecture-Foundations and Applications (ECMDA-FA).* Springer, 2006, pp. 213–225.

[90] B. Li, “Analyzing information-flow in java program based on slicing technique,” *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 5, pp. 98–103, Sep. 2002.

[91] J.-F. Bergeretti and B. a. Carré, “Information-flow and data-flow analysis of while-programs,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 37–61, Jan. 1985.

[92] C. Jones, *Estimating Software Costs : Bringing Realism to Estimating: Bringing Realism to Estimating*, 2nd ed., ser. McGraw-Hill’s AccessEngineering. Mcgraw-hill, 2007.

[93] R.-H. Pfeiffer and A. Wasowski, “Cross-language support mechanisms significantly aid software development,” in *Proceedings of the 15th Int’l Conf. on Model Driven Engineering Languages and Systems*, ser. MODELS’12. Springer-Verlag, 2012, pp. 168–184.

[94] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* SAGE, 1998.

Paper I

Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems

Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems

Amir Reza Yazdanshenas, Leon Moonen

Software Engineering Department, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

Abstract – One way to manage the complexity of software systems is to compose them from reusable components, instead of starting from scratch. Components may be implemented in different programming languages and are tied together using configuration files, or glue code, defining instantiation, initialization and interconnections. Although correctly engineering the composition and configuration of components is crucial for the overall behavior, there is surprisingly little support for incorporating this information in the static verification and validation of these systems. Analyzing the properties of programs *within* closed code boundaries has been studied for some decades and is well-established. This paper contributes a method to support analysis *across* the components of a component-based system. We build upon the Knowledge Discovery Metamodel to reverse engineer homogeneous models for systems composed of heterogeneous artifacts. Our method is implemented in a prototype tool that has been successfully used to track information flow across the components of a component-based system using program slicing.

Keywords – program analysis, reverse engineering, model reconstruction, KDM, component-based software systems, SDG

1 Introduction

Component-based software engineering is a frequently advocated approach for the development of large software systems. It is based on the notion that the complexity of software development can be better managed by *assembling* systems from reusable parts, similar to how hardware systems are constructed from ready-made components. Many of today's software systems are built following these principles: they are composed from reusable components, implemented in one or more programming languages, and connected using a variety of configuration artifacts, ranging from simple key-value maps to elaborate domain specific configuration languages.

Since correctly engineering the composition and configuration of components is no less challenging or error-prone than source code, one could assume that the analysis of such artifacts is an intrinsic part of professional software development methods and tools. However, we found that even though these aspects are crucial for the overall behavior of such systems, there is surprisingly little support for incorporating this information in static verification and validation.

Analyzing the properties of programs *within* closed code boundaries is a well-established area that has been studied for some decades [1], and techniques have successfully been implemented in professional program analysis tools [2, 3]. However, most of these tools have strict limitations on the programming languages that can be processed. In the context of component-based systems, this typically means that information from configuration artifacts can not be included, effectively inhibiting system-wide analysis and confining it to the boundaries defined by the source code of a single component. We address this issue with an approach that allows crossing the boundaries between components, enabling system-wide analysis of component-based systems.

The contributions of this paper are the following: We present a method that combines model-driven engineering with program analysis techniques to support analysis *across* the components of a component-based system. In particular, we build upon the foundations laid out by OMG's Knowledge Discovery Metamodel (KDM) [4] to reverse engineer a homogeneous system-wide dependence model from a software system's heterogeneous source- and configuration artifacts, and use this model as the basis for our analysis. We have implemented and evaluated our approach by building a prototype tool which has been successfully used to track information flow in a component-based system using program slicing. Finally, we add a point of reference to the use and extension of KDM in an industrial setting, extending an area of literature that is currently underdeveloped.

The remainder of the paper is organized as follows: Section 3 describes the background of this study. We describe our approach in Section 4, and report on our prototype implementation in Section 5. We evaluate our approach and prototype in Section 5, discuss the related work in Section 7, and conclude in Section 7.

2 Background and Motivation

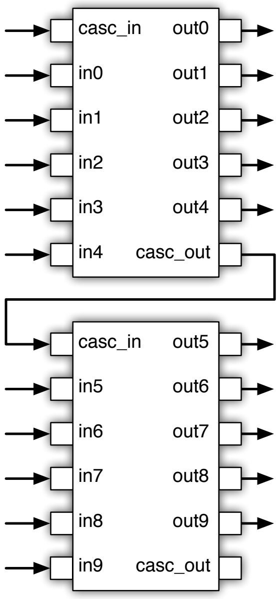
The research described in this paper is part of an ongoing industrial collaboration with Kongsberg Maritime (KM), one of the largest suppliers of systems for dynamic positioning, navigation and automation to vessels and on- and offshore installations worldwide. The division that we work with specializes in computerized systems for safety monitoring and automatic corrective actions on unacceptable hazardous situations. Examples include emergency shutdown, process shutdown, and fire & gas detection in installations such as drilling vessels, and offshore oil and gas terminals. In particular, we study a family of complex safety-critical embedded software systems that connect software control components to physical sensors and mechanical actuators. The overall goal of the collaboration is to supply our partner with software analysis tooling that provides *source based evidence* to support software certification.

The remainder of this section gives a generalized view on how systems are developed in this domain. We use the following terminology: a *component* is a unit of composition with well-defined interfaces and explicit dependencies; a *system* is a network of interacting components; and a *port* is an atomic part of an interface, a single point of interaction between components or components and the environment.

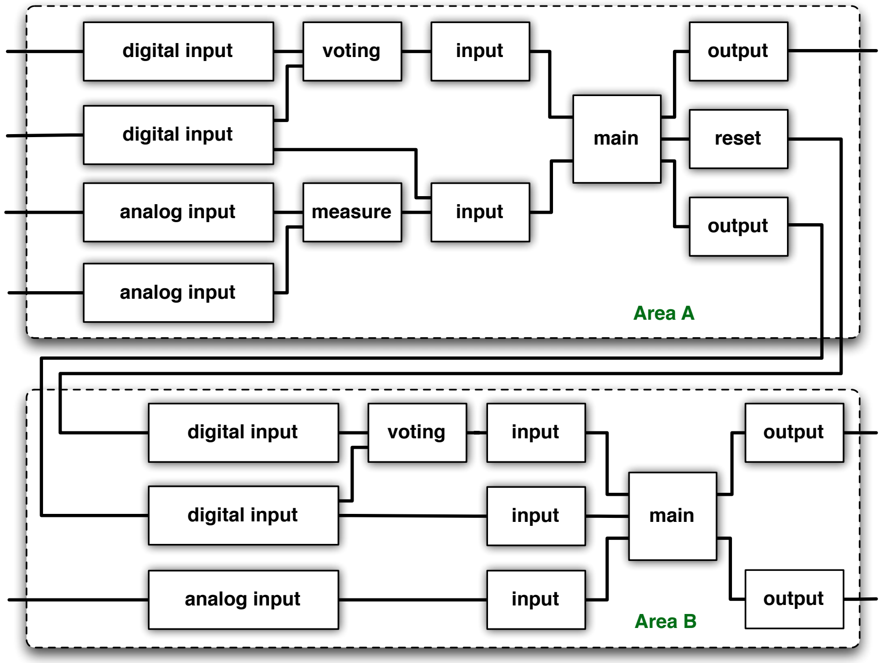
Concrete software products are assembled in a component-based fashion from a limited collection of reusable components. Components are implemented in a safe subset of C called MISRA C [5]. They are relatively small in size and the computations are relatively straightforward. The control logic, however, can be rather complex and is highly configurable via parameters (e.g. initialization, thresholds, multipliers etc). This flexibility is taken to the max in the control components, which are configured using a *cause and effect matrix*. This is basically a decision table that defines what action should be triggered when a given situation arises.

The system's overall logic is composed as a network of interconnected component instances. The control components play a central role and receive inputs that are derived from raw sensor data via a series of components that implement tasks such as measurement, voting, and counting. The control components' outputs are read by a series of components that trigger and drive the system's actuators.

Components can be cascaded to handle larger numbers of input signals (Figure 1a), and the output of a given network can be used as input signal for another (Figure 1b). The latter is used, for example, to reuse the conclusion for one area as input for a connected area. As the installations that are monitored become bigger, the numbers of sensors and actuators grow rapidly, the safety logic becomes increasingly complex and the induced component networks end up interconnecting hundreds of component instances.



(a) Cascading components.



(b) Combining component networks (the drawing is simplified for readability by representing multiple connections between modules as a single line).

Figure 1: Examples of component configurations

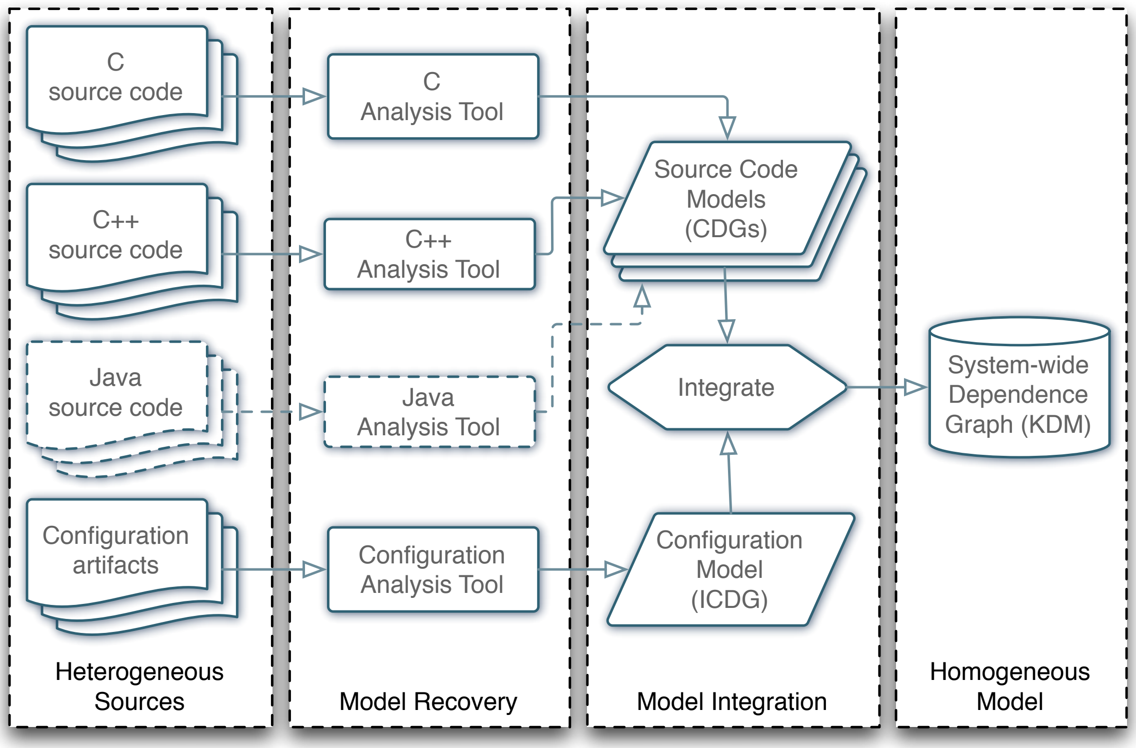


Figure 2: Integrating models derived from heterogeneous sources into a homogeneous model

3 Approach

3.1 Tracking Information Flow

It will not be surprising that one of the main software certification questions asks for evidence that signals from the sensors trigger the appropriate actuators. In program analysis terms, this amounts to tracking the information flow between sensors and actuators through the network of components that makes up the system. Conceptually, this question lends itself well to being answered by means of *program slicing* [6].

Program slicing is a decomposition technique that leaves out all parts of the program that are not relevant to a given point of interest, referred to as the *slicing criterion*. In other words, the program slice consists of the parts of the program that potentially affect the values at the slicing criterion [7]. When we select a given actuator as slicing criterion, the program slice of our system would contain exactly those sensors that may have an effect on the given actuator.

The predominant way of computing program slices is based on traversing the system dependence graph (SDG) [8], and one of the main challenges that a program slicing tool has to tackle is the construction of this SDG from a system's source code. In extension to the original approach which was defined on procedural code, various authors have proposed methods to construct SDGs for other paradigms, such as object oriented and parallel programming. In contrast to our expectations, an investigation of the scientific literature did not bring up any work on the construction of SDGs for heterogeneous component-based systems. As discussed in the introduction, this gap in literature is mirrored by the state of the art in program analysis tools which are typically confined to the boundaries defined by the source code of a single component because they can not construct an SDG that incorporates information from configuration artifacts.

To enable program slicing *across the components* of our subject systems, we devise a method to construct a *system-wide* dependence graph that *integrates* the dependencies from both the components and the configuration artifacts.

3.2 Construction of A System-wide Dependence Graph

This section describes a model-driven approach to construct a system-wide dependence graph that incorporates and integrates the dependence's from all components and configuration artifacts. A high level overview of our approach is shown in Figure 2. We distinguish two main phases in the process: (1) *model recovery* in which we reverse engineer the dependency models of interest from individual source artifacts; (2) *model integration* in which we merge the individual models into a single homogeneous system model.

The overall process of creating a system-wide dependence graph can be described using the following steps. The first two steps are concerned with model reconstruction, the third is concerned with model integration. The process can be completely automated (as shown in Section 5):

1. For each component in the system, we build an (intra-) *component dependence graph (CDG)*. The construction of these CDGs can be done following the SDG construction method in [8], with the component's implementation as "system" source code.
2. The system's configuration artifacts are analyzed to build an *inter-component dependence graph (ICDG)*. This is a dependence graph at a higher level of abstraction than the CDG: the ICDG captures the *externally visible* interfaces and interconnections of components and component instances. These facts can be derived from the configuration files since they are also needed by the component composition framework to set up the correct network. Because the format of the configuration files is specific to the component composition framework, we need to write a dedicated language processor to analyze its configuration files. However, this is not a demanding task as these "languages" are typically very straightforward, most often in the form of key-value pairs or a simple XML based configuration.
3. The *system-wide dependence graph (SDG)* is constructed by integrating the system's ICDG with the CDGs for the individual components. Conceptually, the construction of the SDG can be seen as a process that creates a copy of the ICDG

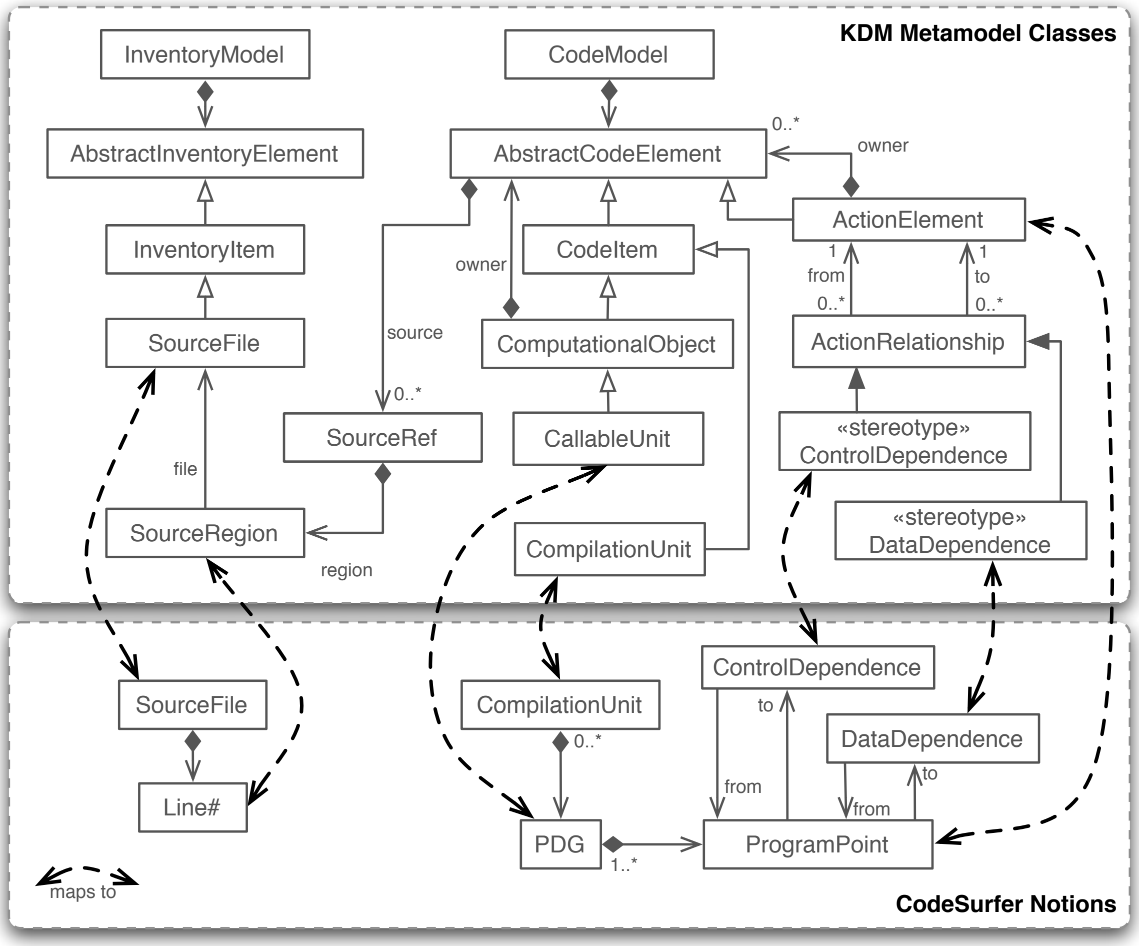


Figure 3: Target KDM metamodel classes and their mapping to CodeSurfer constructs

and replaces each high level “component” node in that copy with a sub-graph that is the CDG for that component.

To enable flexible integration of individual models in step (3), we propose to use OMG’s Knowledge Discovery Metamodel (KDM) [4] as a foundation for representing the various intra- and inter-component dependence graphs. The KDM was designed as a wide-spectrum intermediate representation for describing existing software systems and their operating environments. It is uniform, language- and platform independent. Its goal is to ensure interoperability between tools for maintenance, evolution, assessment and modernization. One of the key concepts is that of a container: an entity that owns other entities. This enables the representation of software systems at various levels of abstraction. The KDM supports incremental analysis that can be used to augment an initial representation based on new knowledge. In addition, it has an extensibility mechanism that allows adding domain-, application- or implementation-specific knowledge. By using the KDM as a basis for our models, we become language agnostic. In the next section we will discuss the concrete mapping from SDG elements to entities in the KDM.

Finally, we want to point out that it is possible to reuse existing program analysis tools for the construction of the individual CDGs in step (1). In this case we will also benefit from the KDM as it helps us to become tool independent. We distinguish the following two sub-steps: (1a) use a third party tool to build the CDG; (1b) apply a model transformation that converts the internal representation of the tool into a KDM-based representation of the CDG. Obviously, the tool should provide access to its internal representation or be able to emit it in some structured format. We will discuss a concrete example of this setup in the next section where we use the CodeSurfer program analysis tool to recover CDGs for components written in C language.

4 Prototype Implementation

In this section we discuss a prototype implementation of the approach that was sketched in Section 4. First, we discuss how we derive CDGs for the individual components by building on functionality provided by a third party tool and transforming the tool's internal representation into dependence graphs represented using KDM. Next, we describe how we analyze configuration artifacts to combine these individual CDGs into a system-wide SDG.

4.1 Component Dependence Graphs

Our prototype builds on Grammatech's CodeSurfer to derive the CDG. CodeSurfer is a program analysis tool that can construct dependence graphs for C and C++ programs [2]. It provides an API that can be used to make your own analysis plugin that can query and traverse the *internal representation* that CodeSurfer builds to analyze a system.

We have built a CodeSurfer plugin that traverses the internal representation and uses the Java Native Interface (JNI) to build a counterpart of the dependence graph by driving a Java implementation of KDM in the Eclipse Modeling Framework (EMF). This relieves us from having to deal with the challenging idiosyncrasies of analyzing C code, including parsing the various dialects and performing pointer analysis.

Figure 3 shows a simplified excerpt of the KDM together with the mapping between CodeSurfer constructs and metamodel classes that we used to represent dependence graphs in the KDM. Although dependence graphs are not "natively" supported in the KDM, the metamodel contains appropriate fine-grained entities that can be used (or extended) to represent such graphs. As is shown in the figure, we can define a direct mapping for most constructs and we use KDM's lightweight extension mechanism to create appropriate stereotypes for constructs that have no direct mapping. Note that we only need a small part of the KDM; the KDM-specific classes in this figure belong to three of the twelve KDM packages: Source, Code and Action. These are respectively shown in the left, middle and right “columns” of the KDM-specific part of Figure 3.

The Code package represents ”implementation level program elements and their associations”, and the Action package expresses ”implementation-level behavior descriptions”. Both packages complement each other to build a CodeModel of the system, capable of describing almost any valid element in a programming language in KDM. For instance, a CallableUnit represents ”a basic stand-alone element that can be called, such as a procedure or a function”. We use this container class to include the information about each PDG. An ActionElement, ”a basic unit of behavior”, is used to represent a program point in PDG, and can be linked to the original representation through the SourceRef element.

The Action package defines several relationship classes to represent relations between ActionElements or between ActionElements and DataElements, such as EntryFlow, GuardedFlow, Calls, Reads, Writes, etc. However, none of these map to the control and data dependency relations in PDGs. ActionRelationship is a ”wild-card element to define new metamodel elements through the KDMs light-weight extension mechanism”. We use ActionRelationships together with the stereotyping mechanism in KDM to express control, data, forward, and backward dependencies among program points.

In addition to the dependence graph, we also extract additional information from CodeSurfer to make our model more complete, such as information regarding compilation units, functions, etc. that is used to populate the *inventory model.* The inventory model is part of KDM’s Source package and is used to represent the physical artifacts in the system [4]. Although we do not need this model to perform slicing, we use it to add traceability to our models. This information can be used, for example, to highlight the source code that is the result of a slice. We use the SourceFile and SourceRegion classes to save the location (file:line#) of each program point.

Based on this mapping we can build CDGs in KDM. This enables us to compute an *intra*-component slice: when we select an output port as slicing criterion, we can determine which of the component’s input ports can affect the value on that output port. Although one could argue that the transition from proprietary program analysis tool to KDM-enabled platform opens up many interoperability opportunities, we still can not do anything more than CodeSurfer already does out of the box. To change this, we need to take the next step and assemble a system-wide dependence graph.

4.2 The Inter-Component Dependence Graph

Before we can assemble all individual CDGs into a system-wide dependence graph (SDG), we analyze the configuration artifacts to derive information about component instantiations and interconnections. We capture this information in an inter-component dependence graph (ICDG) which models the *externally visible* interfaces and interconnections of components and component instances. The nodes in this graph represent

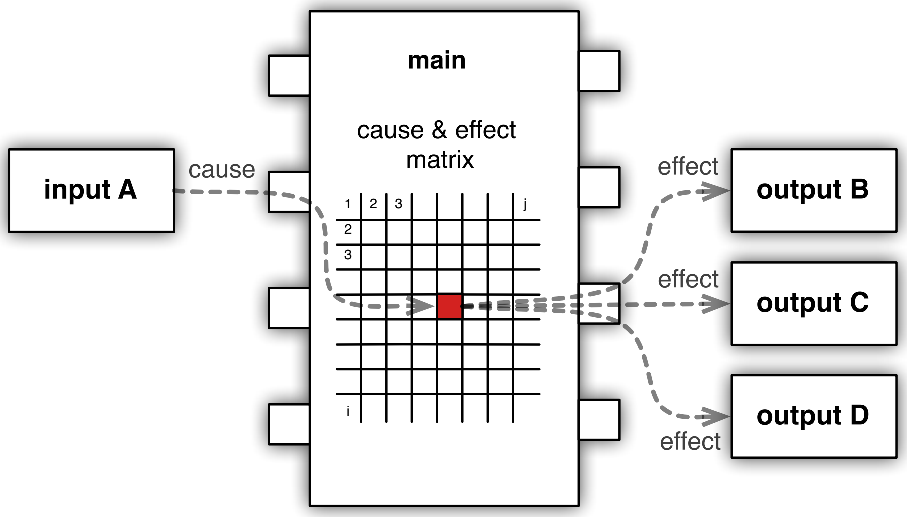


Figure 4: Cause&Effect matrix: a domain-specific inter-component communication mechanism based on shared memory.

component instances and port instances and we use data dependence edges to represent connections between port instances, and control dependence edges to associate components with their input/output ports. We refer to Figure 5 later in this paper for an overview of the nodes and edges in the ICDG (but note that this figure serves another role and the ICDG does not contain the parts that are shown *inside* the grey component nodes). In our case study, the configuration files are in XML and we use Xalan-Java for processing (but other XSLT processors could have been used as well).

We distinguish two types of inter-component communication: the first type are the common *port-based* connections where an output port of component A is connected to an input port of component B. These connections are explicitly defined in the configuration files and can directly be translated into dependencies between port instances in our ICDG.

The second type of connections are made via the *cause & effect matrix*, a domain-specific inter-connection mechanism that needs some explanation: At the core of the system, the inputs (causes) are processed by a control component that decides what outputs (effects) to trigger. The mapping from causes to effects is encoded 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.

The C&E matrix is implemented as shared memory in the main control component (see Figure 4). It creates a blackboard architecture to which components have read or write access. Each input component handles one cause and can only write to a single cell in the C&E matrix. Multiple output components can read that same cell, effectively

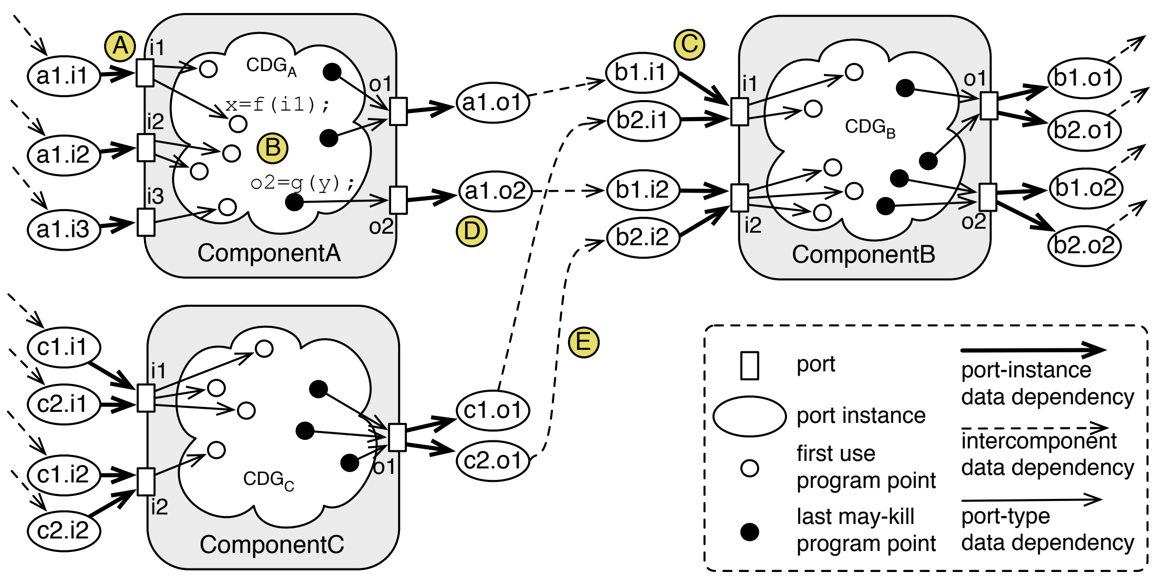


Figure 5: Assembling the SDG from the individual CDGs and the ICDG (note that markers A–E are explained in the text).

ensuring that a cause could trigger multiple effects. The cells to which the inputs can write and from which the output modules can read are described in (XML-based) configuration files.

Note that this form of connections is of special interest because static analysis tools in general have trouble with following the flow through such a block of shared memory. Even the tools that use sophisticated pointer-analysis algorithms will at some point need to make the trade-off between precision and analysis time/resources. For most tools this trade-off means that they will analyze pointers down to the level of directly addressable (named) memory locations, but not consider the individual elements of arrays or matrices: these are lumped together as a single array or matrix object. This conservative estimate has unfortunate effects in this situation, as the C&E matrix will be seen as a single object that is being written by all input- and read put all output components. Although this is a *safe* approximation that does not miss any potential dependencies, it is prohibitively sub-optimal as it creates numerous false positives.

We address this issue as follows: During processing of the configuration artifacts, whenever we find a pair of input-output components that write and read from the same matrix cell respectively, we capture this *indirect* connection via the C&E by adding a *direct* inter-component data dependency between the input and output components to the ICDG. This will enable program analysis (and slicing) to “pass through” the C&E matrix from output component instances to exactly those input component instances that write to the same cell in the C&E matrix as the output components read from.

Although this example is specific to our case, we believe that the proposed solution is general enough to be used as a template for other inter-component communication mechanisms, such as message passing, sockets, and pipes.

4.3 The System-wide Dependence Graph

The method of assembling a system-wide dependence graph from CDGs closely resembles the method of building an SDG from a collection of PDGs in [8] with the exception that there is no call-return relation between a couple of connected components so we adapt our description accordingly. The concrete assembly process is implemented as follows: Based on the information in the ICDG, we add an ActionElement for each port to the owner component (CompilationUnit) in our KDM model (Figure 5, marker A). These ActionElements (ports) play exactly the same role to a component, as a formal parameter plays to a procedure.

Analogous to the intra-procedural dependency edges of each formal parameter, we need to add the intra-component data dependencies of each port (Figure 5, marker B). The *output* ports have a data dependency to the last "may-kill" program points for that port, i.e. those locations at which the value communicated over the port can be defined [9]. Similarly, there is a data dependence between an input port and the first "uses" of values received over that port. Note that a component may contain multiple functions that read or write values to ports and we need to add the above data dependencies for each of them.

We model component instantiation analogous to procedure calls in [8] with the exception that there is no return flow. For each port instance in the ICDG, we add an ActionElement and add a data dependency to the element representing the port (Figure 5 marker C). Such port-instance nodes roughly correspond to actual parameters in procedure calls. Note that the structured names of port-instance ActionElements (Figure 5, marker D) play an important role in our method as these are used to associate the input ports of a component instance to the output ports of the same component instance. This helps preserving *context* during slicing.

Finally, we add the component interconnections to the model: wherever we see a data dependence between two port instances in the ICDG, we simply add a data dependency edge between the corresponding ActionElements of those port-instances (Figure 5, marker E).

4.4 Slicing

Now we have a homogeneous model representing the system-wide dependence graph, we can slice it to gather evidence to support our original certification questions.

We have created a simple slicing tool in Java which compute slices by traversing the dependencies (ActionRelationships) in our SDG using the standard graph reachability algorithms with one minor adaptation for context preservation: when entering a component via a port instance we save the component instance name, and when exiting a component, we only ascend to those port *instances* that belong to the same component instance as the saved one.

5 Evaluation

In order to evaluate our approach and the implemented prototype tool, we will consider two aspects: First, we evaluate accuracy by comparing the results of our slicing method with a gold standard set by CodeSurfer. Second, we evaluate performance and scalability by converting and analyzing a series of large industrial code bases.

5.1 Accuracy

One of the challenges in evaluating the accuracy of our approach is determining a *gold standard* to compare our results to. Remember that one of the motivations was that existing approaches and tools are not able to handle the type of systems that we want to analyze.

We have solved this challenge by increasing our level of control during the experimental evaluation: First, we have developed a simple component based system that closely resembles the architecture of the ones described in Section 3. Our system consists of a "framework" (main function) that reads a number of external configuration files that describe how it should instantiate and interconnect a network of components (represented by other functions). We follow a similar component-based design and use the same component interconnection mechanisms as the system in our case study. 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. The characteristics of this system are described as System A in Table 1.

Second, because we have full control over this system, we can trivially create a variant A' in which we replace the framework code that reads the configuration files by code that directly instantiates and interconnects components. To minimize the differences, this *hard-coded* variant A' uses the same instantiation and interconnection functions as the configuration file reader to programmatically build a network of components. We program A' to create a network that corresponds exactly to the network that is specified in the configuration files of system A.

Since system A' 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. The components and configuration artifacts of the original system (A) are analyzed using our prototype tool-set: we generate an SDG in KDM using the tooling described in Section 5-A&B and slice it for a given set of slicing criteria using the tool described in Section 4.4.

We evaluate the accuracy by comparing the slices obtained for system A using our tool-set with the gold standard computed by CodeSurfer on system A', 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, we have repeated this comparison for all elements in a set of slicing criteria that was increased in a guided-random fashion until the complete set of slices covered the SDG (i.e., in each increment we add a randomly selected element from the program points that were not yet covered as new slicing criterion, until we have covered the whole SDG). Moreover, we have repeated this process for three different configurations (adding variants A" and A"').

Our comparisons showed that for each configuration and slicing criterion, both slicing tools generated 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 programmatically set up the component connections. Since our approach by design 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.

5.2 Scalability

For this step we use our prototype to analyze the source code of three industrial code bases of increasing size and create the corresponding SDGs in KDM. These systems are shown as systems B, C and D in Table 1. Note that the number of components that is reported refers to the number of component *types* in each code base. Each of these types may be instantiated numerous times in an actual configuration.

Analysis of the results shows that the number of nodes (ActionElements) in the KDM SDG is equal to the sum of all program points of the individual CDGs in CodeSurfer, as long as there are no component instantiations. When component instantiation is included, the difference between these two is a linear function of the number of instances of each component and the number of input/output ports of the instantiated components. This shows the main advantage of the way how we model component instances compared to the alternative, where the complete CDG is duplicated for each component instance. The latter approach would yield a high risk of scalability problems in our case, since the typical scenario in our application domain is to create large numbers of instances from a limited set of components.

The model reconstruction and transformations are performed on a general-purpose laptop with 2.66 GHz Intel Core 2 Duo CPU, 4 GB of memory, running Mac OS X V10.6. The value reported as "Total CodeSurfer time" is the sum of the times that it takes CodeSurfer to create all individual CDGs, including the time for parsing and full pointer analysis. The value reported as "SDG construction time" includes reconstructing the ICDG from the configuration artifacts, transforming all CDGs into KDM representation and assembling these parts into a single homogeneous SDG.

To minimize potential performance fluctuations of a multitasking system, we profile 22 executions of our model transformation, omit both the longest and shortest execution times and report the average time of remaining 20 executions. We should

Table 1: Characteristics of analyzed systems and resulting models

System	A	B	C	D
# Distinct Components	4	6	30	60
LOC	207	16181	54053	101393
Total CodeSurfer time (sec.)	3.181	13.064	65.022	132.381
SDG construction time (sec.)	0.246	1.996	9.938	19.755
# Nodes in final SDG	2074	13787	61507	121197
# Dependencies in final SDG	3784	46276	216956	431042

remark that the transformation times had very little variation, so this precaution was probably not needed. However, the purpose of these tests was not so much to analyze the execution times but to assess the scalability. Our results show that both execution time and model size grow linear as the system size grows (see also Figure 6, the small dent can be explained by startup overhead which has more impact for A than for the other systems). The growth rate is constant, even for the largest code base which measures a little over 100KLOC in size. The serialized KDM model for this code base results in an impressive 600,000 lines of XMI (78MB). In all cases, the execution of the slicing algorithm takes a trivial time, in the order of milliseconds.

5.3 Threats to validity

We have identified the following threats to the validity: *Internal Validity:* Since our accuracy evaluation is based on a form of "regression testing" where we compare the slices generated by our approach for random slicing criteria with the slices that are generated by CodeSurfer, there are two factors that could affect the evidence that supports our claims: (1) The statically configured systems A', A" and A"' that are analyzed by CodeSurfer may differ from the original system A that is analyzed by our approach (and cannot be analyzed by CodeSurfer). While designing the example software systems, we have taken all possible measures to prevent differences between these systems with the required exception of the way in which the component network is configured. The fact that all the resulting slices are identical supports our belief that we were successful in mitigating this threat of changing the instrument. (2) The evaluation is based on randomly selected slicing criteria that by chance may not expose problems in our implementation. We have minimized this risk by taking a sufficiently large number of slices and use a random selection of slicing criteria to increase the coverage of the SDG by the generated slices.

External Validity: We have identified the following two threats to the generalizability of our results: (1) In addition to our own example system, the study only includes industrial code from one particular company. As a result, there may be a bias in our approach towards specifics of that particular codebase. In general, this is hard to

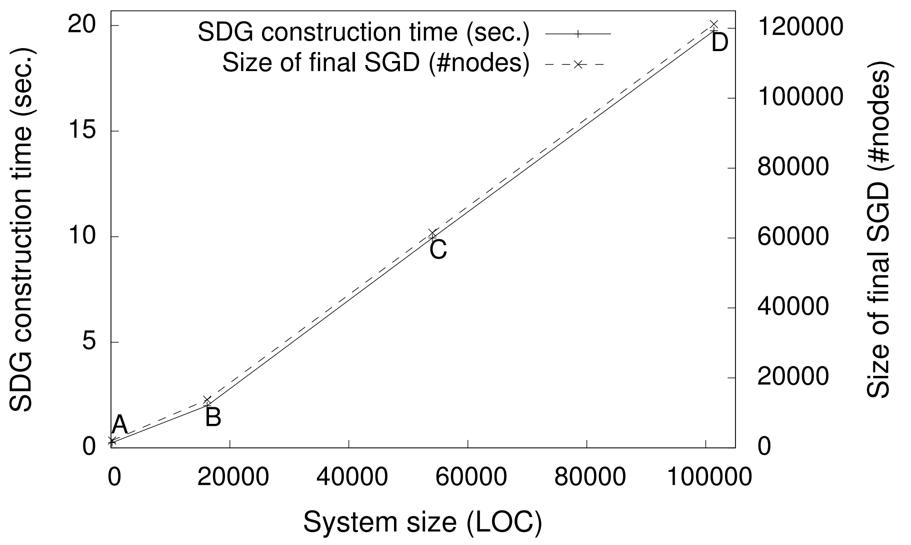


Figure 6: Transformation time and SDG size vs. system size.

avoid in an industrial collaboration, and specifically so in a setting as described in this paper where one needs to develop a small dedicated language processor to derive facts from the configuration artifacts. However, we have identified two general component interconnection mechanisms and present a solution that can be used as template for other interconnection mechanisms. We leave the demonstration on other cases as future work. (2) The evaluation is based based on one particular tool (CodeSurfer) to generate CDG's. Although this tool supports both C and C++, the generalizability to tools that process other languages is not evaluated. The extension of the SDG to represent other languages has been described by many papers and we do not expect problems with mapping those extensions to KDM. However there seems to be only a limited amount of industrial strength tools that can create PDGs or SDGs from given source code, so this may be a practical challenge to the generalizability.

6 Related Work

Architecture Driven Modernization: In the recent years, several studies have been published that follow the ADM approach and use KDM to capture knowledge about legacy systems [10–13]. Although there are similarities in the approach and use of KDM, both the type and abstraction level of the information that is recovered in these studies is very different from ours. The MARBLE framework [11] creates KDM models from database schemes and SQL statements embedded in Java source code which are used for data contextualization: the recovery of links between source code and the relevant parts of any databases that are used. In [12, 13], KDM models are recovered from PL/SQL triggers in Oracle Forms applications. These models are used to measure the coupling between code and UI as this was recognized as major factor influencing the time and effort required to migrating the applications.

MoDisco is an Eclipse plug-in aimed at supporting model-driven software modernization by reverse engineering models from (Java) source code [14]. It consists of a “model discoverer”, which uses the Eclipse Java Development Tools (JDT) parser and its resulting AST to create models from Java source code files. These models conform to a detailed Java metamodel defined in Ecore, and can be browsed by the MoDisco model browser. In addition, they can be analyzed and explored by all tools that can process Ecore models, such as transformation and querying engines. Finally, MoDisco includes transformations to transform their internal Java models into models that conform to the Knowledge Discovery Metamodel (KDM) and the Software Metrics Metamodel (SMM). Several other researchers have investigated the reverse engineering of fine-grained model from source code, resulting in tools such as Spoon [15], and JaMoPP [16] but at the time of writing, these approaches do not generate KDM compliant models. The main difference between our work and the approaches mentioned above is that those are based on building *structural* models of the code entities and their direct relations, such as function calls and control flow, whereas our approach is aimed at models that include the higher level semantic relations needed for program analysis (such as control and data dependence). As such, the KDM models that are recovered by MoDisco are orthogonal to ours for the same set of source artifacts, and one of the main advantages of building on KDM is that they can easily be merged together to recover an even richer model.
Program analysis: Ricca and Tonella describe the construction of system dependence graphs for web application slicing [17]. Their approach addresses a problem similar to ours in that they need to combine dependence information from the server side programming language PHP with dependence information from the client side programming language JavaScript. They extend the traditional SDG to one that contains specific dependencies for web applications.

Several authors have studied slicing at the architectural level. In general, all these approaches aim at raising the abstraction level of the analysis to the component level: the (dependency) relations that are captured are *between* components and not *within* components. As such, these approaches cannot be used to conduct a detailed analysis *across* components, as we aim at in our work. The authors typically aim at answering *impact analysis* questions such as “What other components are required when one component is to be reused in another system?”, “What other components might be affected when a given component is changed?”, “What is the minimal set of components that must be inspected when a system fails at a given component?”. Both Zhao [18] and Stafford et al. [19] have studied the analysis and slicing of software architectures based on their specification in an Architecture Description Language (ADL). In both approaches, the components and relations in a software system’s architecture are first (manually) modelled in a domain specific language before they are analyzed. In addition to the difference in abstraction levels described above, these approaches differ from ours in that we aim at automatically reconstructing our analysis models from the system's source artifacts (code and configuration).

Li et al. introduce the component dependency graph (and component dependence adjacency matrix) to explicitly represent dependencies in a component-based system [20]. They find components in C++ or Java source code by identifying all classes and interfaces, and derive component dependencies from the **#include** directives. The difference with our work is that the granularity of dependencies in their approach is at the component level, where the Boolean cells in their adjacency matrix indicate the existence or absence of a dependency between the two components. Such information can be used to estimate the impact of changes, and for finding the set of components that is required to support the reuse of a component in another system. However, the granularity is too coarse for the type of detailed program analysis that we aim to support with our technique, such as program slicing and information flow analysis, which need dependencies at the granularity of individual program points.

Eichberg et al. define an approach that uses static analysis expressed in Datalog for the continuous checking of constraints on *structural* program dependencies [21]. The granularity of their dependencies is more fine-grained than that of Li et al. discussed above and ranges from intra-class dependencies to the level of architectural building blocks. Their approach is designed to check architectural and design level constraints and they provide a domain-specific language to easily specify these constraints. The main difference with our work is that their approach is limited to identifying and reasoning over structural dependencies between source elements. Since they do not capture semantic dependencies such as control- and data-dependence, their approach cannot be used to analyze (constraints on) the information flow in a system, as opposed to our work.

7 Concluding Remarks

Many of today's software systems are composed from reusable components, implemented in one or more programming languages, and connected using a variety of configuration artifacts. Correctly engineering these configuration artifacts is no less challenging or error-prone than source code. We found that even though these are crucial for the overall behavior of these systems, there is surprisingly little support for incorporating them in the static verification and validation. In this paper, we remediate this situation.

Contributions of this paper include: We present a method that combines model-driven engineering with program analysis techniques to support analysis *across* the components of a component-based system. Our approach is based on (1) recovering intra-component dependence graphs (CDGs) for each component; (2) recovering an inter-component dependence graph (ICDG) from the configuration artifacts; and (3) integrating the ICDG with the various CDGs to reconstruct a system-wide dependence graph (SDG).

We build on the Knowledge Discovery Metamodel (KDM) to reverse engineer a homogeneous model from heterogeneous artifacts. We leverage KDM to become programming language agnostic and tool independent. This enables us to reuse existing tools for constructing the individual CDGs.

We have implemented and evaluated our approach by building two prototype tools which have been successfully used to recover models from component-based systems and track information flow using program slicing. We have tested the scalability of our approach on industrial code bases up to 100 KLOC, and the results show a linear growth in execution time and model size, as the system size increases.

Future Work: We see several directions for future research. The first (and obvious) one is the extension of our prototype and experiments to include the analysis of more source languages and component composition/configuration languages.

Next, considering the size and complexity of most industrial systems, there are many opportunities in the direction of visualizing the analysis results. So far, we have used the SourceRegion objects in our KDM model as traceability links between the analysis results and the source code, but a visualization of the information flow at higher levels of abstraction may considerably improve the comprehensibility. More abstract visualizations are of special interest to our industrial partner because it is not just the developers but also the (non-developer) safety domain experts that could use the recovered information to support software certification.

Another interesting direction is the “injection” of our SDG *back* into CodeSurfer by modifying its internal representation. This would enable us to reuse the visualization, exploration and analysis capabilities of CodeSurfer.

Finally, by integrating the SDG into a graph exploration tool, it may be possible to provide more user-friendly visualization and navigation facilities. This can, for example, enable the user to “zoom” from a high-level view of the system showing information flow, to a fine-grained view showing CDG internals. Such variations in abstraction level support the requirements imposed by different maintenance tasks, for instance debugging a single component, or finding an ill-configured system before deployment.

Bibliography

[1] D. Binkley, “Source Code Analysis: A Road Map,” in *Future of Software Engineering*. IEEE, May 2007, pp. 104–119.

[2] P. Anderson, “90% Perspiration: Engineering Static Analysis Techniques for Industrial Applications,” in *IEEE Int’l Working Conf. on Source Code Analysis and Manipulation*, Sep. 2008, pp. 3–12.

[3] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins, “Tool support for fine-grained software inspection,” *IEEE Software*, vol. 20, no. 4, pp. 42–50, Jul. 2003.

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

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

[6] M. Weiser, “Program Slicing,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[7] K. Gallagher and D. Binkley, “Program slicing,” in *Frontiers of Software Maintenance.* IEEE, Sep. 2008, pp. 58–67.

[8] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM TOPLAS*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[9] M. S. Hecht, *Flow analysis of computer programs.* North Holland, 1977.

[10] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies.* Morgan Kaufmann, 2010.

[11] R. Pérez-Castillo, I. García-Rodríguez de Guzmán, M. Piattini, and O. Ávila garcía, “On the Use of ADM to Contextualize Data on Legacy Source Code for Software Modernization,” in *Working Conf. on Reverse Engineering*, 2009, pp. 128–132.

[12] J. L. C. Izquierdo and J. G. Molina, “A Domain Specific Language for Extracting Models in Software Modernization,” in *European Conf. on Model Driven Architecture-Foundations and Applications (ECMDA-FA).* Springer, 2009, pp. 82–97.

[13] J. L. C. Izquierdo and J. G. Molina, “An Architecture-Driven Modernization Tool for Calculating Metrics,” *IEEE Software*, vol. 27, no. 4, pp. 37–43, Jul. 2010.

[14] H. Brunelière, J. Cabot, F. Jouault, and F. Madiot, “MoDisco: a generic and extensible framework for model driven reverse engineering,” in *IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE).* ACM, 2010, pp. 173–174.

[15] R. Pawlak, C. Noguera, and N. Petitprez, “Spoon: Program Analysis and Transformation in Java,” Tech. report #inria-00071366, INRIA, 2006.

[16] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, “Closing the Gap between Modelling and Java,” in *Software Language Engineering (SLE).* Springer, 2009, pp. 374–383.

[17] F. Ricca and P. Tonella, “Construction of the system dependence graph for Web application slicing,” in *IEEE Int'l Ws. on Source Code Analysis and Manipulation*, 2002.

[18] J. Zhao, “A slicing-based approach to extracting reusable software architectures,” in *European Conf. on Software Maintenance and Reengineering.* IEEE, 2000, pp. 215–223.

[19] J. A. Stafford and A. L. Wolf, “Architecture-Level Dependence Analysis for Software Systems,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 04, pp. 431–451, Aug. 2001.

[20] B. Li, Y. Zhou, Y. Wang, and J. Mo, “Matrix-based component dependence representation and its applications in software quality assurance,” *ACM SIGPLAN Notices*, vol. 40, no. 11, pp. 1–29, Nov. 2005.

[21] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, “Defining and continuous checking of structural program dependencies,” in *Int'l Conf. on Software Engineering.* ACM, 2008, pp. 391–400.

Paper II

Analyzing and Visualizing Information Flow in Heterogeneous Component-Based Software Systems

Analyzing and Visualizing Information Flow in Heterogeneous Component-Based Software Systems

Amir Reza Yazdanshenas, Leon Moonen

Software Engineering Department, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

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

This paper contributes a method to *analyze and visualize information flow* in a component-based system at various levels of abstraction, and from two complementary perspectives. We propose a hierarchy of five interconnected views to support the comprehension needs of both safety domain experts and developers from our industrial partner. The abstractions were selected to reduce visual distraction and reduce cognitive overload while satisfying the prospective users' information needs. The perspectives focus on either impact analysis or dependence analysis. The views are interconnected in a way that supports both systematic, as well as opportunistic navigation scenarios. We discuss the implementation of our approach in a prototype tool, and present two qualitative evaluation studies on the effectiveness and usability of the proposed views for software development and software certification. During the evaluation, the prototype has already been found to be very useful, and a number of directions for further improvement were suggested. We conclude by discussing these improvements and the lessons learned.

Keywords – information flow analysis, component-based software systems, model reconstruction, program comprehension, software visualization

1 Introduction

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

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

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

In spite of these challenges, we found that there is little support for system-wide analysis of component-based systems from their source artifacts. Most of the available tools have strict limitations on the programming languages they can process. This typically means that information from external configuration artifacts can not be included, effectively inhibiting system-wide analysis and confining it to the boundaries defined by the source code of a single component. In practice, this means that software engineers have only their own cognition abilities to rely on for understanding the overall system's behavior.

Another complicating factor in engineering large industrial software systems is that it is not just the developers who need to understand what's going on in the code: also non-developers, for instance safety domain experts, need to understand what is actually implemented in the code to assess whether the system properly adheres to given safety requirements. However, most of the literature on reverse engineering and program comprehension assumes that the developers are the default, and the only, audience. There is extensive literature on the visualization of *non-source* artifacts to support domain experts [e.g., 21], but considerably less information exists on the visualization of source code-related information for non-developers. After all, why would non-developers need to understand source code?

This paper is motivated by a typical industrial case in which (non-developer) safety domain experts need to understand the logic implemented in the system so that they can conduct software certification. These safety domain experts need to see the system's source artifacts in a context relevant to them – not just what the code *does*, but what that *means* for safety concerns [18]. Consequently, any reverse-engineered views on the system need to be goal-driven, at a suitable level of abstraction, and based on relevant knowledge of the application domain.

Our earlier work [29] presents a technique to reverse engineer a fine-grained, system-wide dependence model from the source and configuration artifacts of a component-based system. The paper concluded with the observations that the technique was promising but *"considering the size and complexity of most industrial systems, there are many opportunities in the direction of visualizing the analysis results,"* and *"a visualization of the information flow at higher levels of abstraction may considerably improve the comprehensibility."*

The current paper builds on the technology developed in [29] and makes the following contributions: (1) We propose a hierarchy of views that represent system-wide information flows at various levels of abstraction, aimed at supporting both safety domain experts and developers; (2) We present the transformations that help us to achieve these views from the system-wide dependence models and discuss the different trade-offs between scope and granularity; (3) We discuss how we have implemented our approach and views in a prototype tool, named FlowTracker; (4) We report on two qualitative evaluations of the effectiveness and usability of the proposed views for software development and software certification. The results from the first evaluation indicated that the prototype was already very useful and a number of directions for further improvement were suggested. Based on these suggestions, several changes and extensions were implemented which were in turn evaluated in the second study. We have reported on the initial results from the first study in [30]. In this paper we elaborate on those results, discuss the changes that were made to improve on the views and the tooling and report on the followup evaluation.

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

2 Motivation

The research described in this paper is part of an ongoing industrial collaboration with Kongsberg Maritime (KM), one of the largest suppliers of programmable marine electronics worldwide. The division that we work with specializes in computerized systems for safety monitoring and automated corrective measures to mitigate unacceptable hazardous situations. Examples include emergency shutdown, process shutdown, and fire-and-gas detection systems for vessels and off-shore platforms. In particular, we study a family of complex, safety-critical embedded software systems that connect software control components to physical sensors and mechanical actuators. The overall goal of the collaboration is to provide our partner with tooling that provides *source-based evidence to support software certification*, and assists the development teams in understanding the behavior of *deployed systems*, i.e., systems composed and configured to monitor the safety requirements of a particular installation (execution environment).

The remainder of this section gives a generalized view on how systems are developed in this application domain. We use the following terminology: a *component* is a unit of composition with well-defined interfaces and explicit context dependencies [24]; a *system* is a network of interacting components; and a (component) *port* is an atomic part of an (component) interface, a single point of interaction between a component and other components or the environment. A component *instance* is the representation of a component as it would appear at run-time, specialized and interconnected following the configuration data. A component *implementation* refers to the component's source code artifacts (i.e., without configuration information). There is one component implementation and possibly several component instances for each component in the system.

Without loss of generality, we discuss our approach in terms of the system we studied. This means that we use the general term *system-level input* and the more case-specific term *sensor* interchangeably, and, similarly, for *system-level output* and *activator*. We emphasize that the proposed approach can also be applied to component-based systems with other types of input and output than sensors and activators. In cases where the direction of a system interaction point is not significant, we use the general term *system port* to refer to both system-level inputs and outputs.

Concrete software products are assembled in a component-based fashion from a limited collection of approximately 30 reusable components. The components are implemented in MISRA C (a safe subset of C [10]). They are relatively small (in the order of 1-2 KLOC), and the computations are relatively straightforward. Their control logic, however, can be complex and is highly configurable via parameters (e.g., initialization, thresholds, comparison values, etc).

The system's overall logic is achieved by composing a network of interconnected component instances (Figure 1). These processing pipelines receive their input values from sensors and process them in various ways, such as measuring, digitizing, voting, and counting before sending the outputs to drivers for the actuators. Components of the same type can be cascaded to handle a larger number of input signals than foreseen in their implementation (shown in Figure 1 for analog inputs #1 and #2). Similarly, the output of a pipeline can be used as input for another pipeline to reuse the safety outcomes for one area as inputs for a connected area.

Research Question – As monitored installations become bigger, the number of sensors and actuators grows rapidly, the safety logic becomes increasingly complex, and the induced component networks end up interconnecting thousands of component instances.

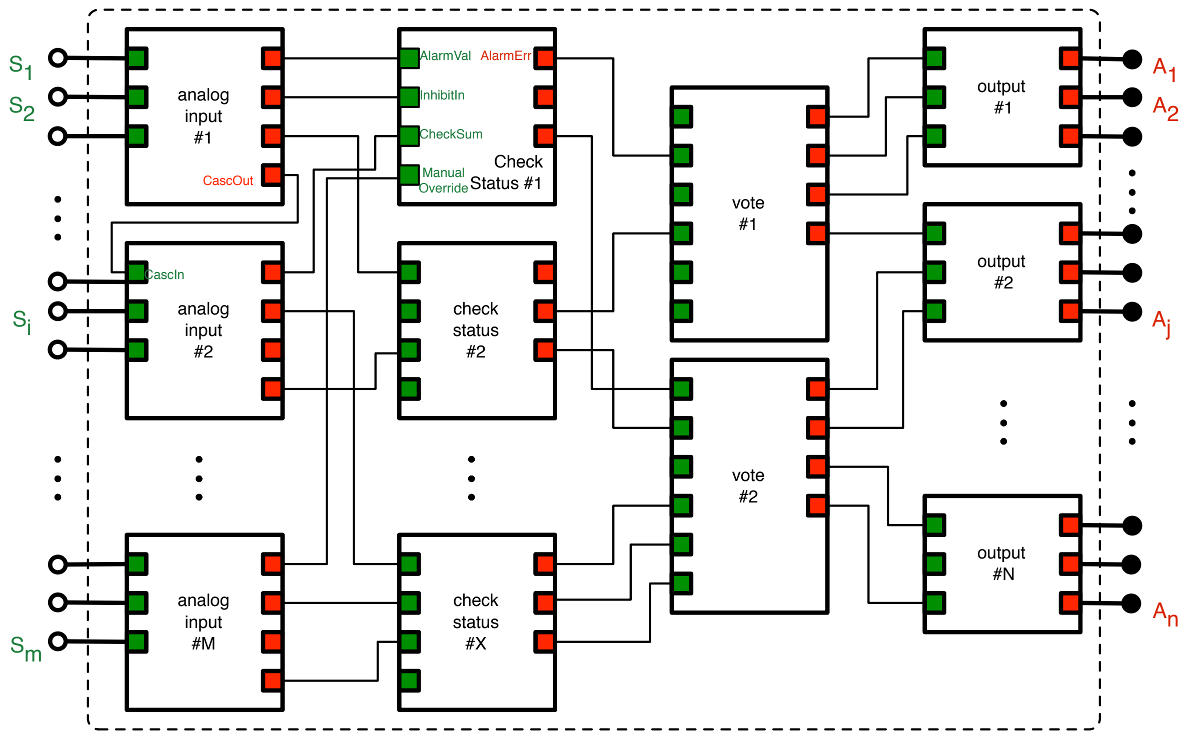


Figure 1: Component composition network for an example system.

To make this more concrete, consider that in contrast to those 11 instances and 4 stages shown in Figure 1, a typical real-life installation has 12 to 20 stages in each pipeline, and approximately 5,000 component instances in its safety system. As a result of these numbers, it becomes increasingly difficult to understand and reason about the overall behavior of the system. The *main* question that drives our research is: *"Can we provide source-based evidence that the signals from the system's sensors trigger the appropriate actuators?"*

In addition to this primary goal of supporting software certification questions, our industry partner indicated that they had a secondary goal: During the collaboration, their developers and system integrators recognized that such system-wide program comprehension techniques had the potential to support various development and maintenance tasks Storey [23]. For example, they were looking for support that helped them to predict the consequences of a change in a given component on the complete system (i.e., impact analysis). Similarly, they also wanted to understand better what parts of the system could actually affect the state of a given component. To support both the primary and secondary goals, we set out to provide black-box and white-box visualizations of the system at various levels of abstraction, aimed to satisfy the needs of the various users and tasks foreseen by our collaborator, and allowing for a trade-off between detail and cognitive complexity.

3 Approach

The question if signals from the system's sensors affect the appropriate actuators can be answered by analyzing the information flow between sensors and actuators using program slicing [28]. Program slicing is a decomposition technique that can be used to leave out all parts of the program that are irrelevant to a given point of interest, referred to as the *slicing criterion*. In other words, a *backward* slice consists of all the program elements that potentially affect the values at the slicing criterion [8]. Thus, by selecting an actuator as the slicing criterion, we can determine which sensors can affect this actuator, since these will be contained in its backward slice. Conversely, a *forward* slice consists of all the program points that are potentially affected by the slicing criterion [8]. Thus, by selecting a sensor as the slicing criterion, we can determine which actuators can be affected by this sensor, since they will be contained in its forward slice. In the remainder, analysis *direction* refers to the direction of the slicing, and *forward (backward) information flow* refers to an information flow analyzed via forward (backward) slicing.

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

The first challenge is addressed by reverse engineering a fine-grained, system-wide model of the control and data dependencies in the system based on our previous work [29], which is briefly summarized in Section 3.1. To address the second challenge, we propose a hierarchy of five abstractions (views). We discuss how these views are constructed from the system-wide dependence model via a combination of slicing, transformation (abstraction), and visualization in Section 3.2.

3.1 Reverse Engineering a System-Wide Dependence Model

This section summarizes the technique and terminology of our earlier work on reconstructing system-wide dependence models [29]. The overall approach is as follows:

1. For each component in the system, we build a *component dependence graph (CDG)* by following the method for constructing interprocedural dependence graphs [12] and taking the component source code as "system source."

2. The system's configuration artifacts are analyzed to build an *intercomponent dependence graph (ICDG)*. This graph captures the *externally visible* interfaces and interconnections of the component instances. Construction of the ICDG is done in the same way as the component composition framework sets up the correct

network.

3. The *system-wide dependence graph (SDG)* is constructed by integrating the system's ICDG with the CDGs for the individual components. Conceptually, the construction can be seen as taking the ICDG and substituting each "component instance node" with a sub-graph formed by the CDG for the component.

Figure 4 gives an overview of the main types of information that we collect from various source artifacts to build the SDG. To construct the CDG, the fundamental information is the program points, and the data and control dependencies. For traceability purposes, we also extract some properties of information points, such as their location in the source files and the physical structure of the software artifacts. This information is extracted from the components' implementation. For the ICDG, the information is a component's inputs and outputs, parameters (see Section 3.5), instances, and intercomponent connections. This information is extracted from the configuration artifacts.[8]

3.2 Model Abstraction and Visualization

Dependence graphs, and slices through dependence graphs, are complex, often even more complex than the original source artifacts. This is because these models reflect all relevant program points and dependencies from a compiler's perspective, an intrinsic characteristic that makes them well-suited for detailed program analysis. This characteristic does, however, make them less suited for directly supporting comprehension or visualization [13, 29].

To make the detailed information contained in an SDG or slice more suitable for comprehension, we propose a hierarchy of five abstractions (views) aimed at satisfying the needs of safety experts and developers. These needs range from a black-box survey of the system, via a number of intermediate levels, to a hypertext version of the source code. These views are constructed from the system-wide dependence model via a combination of slicing, transformation and visualization. Since, in our case, understanding the system-wide dependencies is required in both directions (i.e., forward and backward), all visualization levels accommodate abstractions over both forward and backward slices. Depending on the nature of the visualization, the abstractions over forward and backward analyses are either visualized in separate diagrams, or, whenever possible, in a single diagram enriched to accommodate both directions of analysis. Such separate (set of) diagrams that belong to the same abstraction layer target the same *type* of desired system elements, and can address similar, but not identical, comprehension requirements. In our description of the abstraction levels, we distinguish between the two analysis directions, if needed. The various levels are interconnected via hyperlinks

[8] [29] contains more information about mapping the extracted information points to KDM.

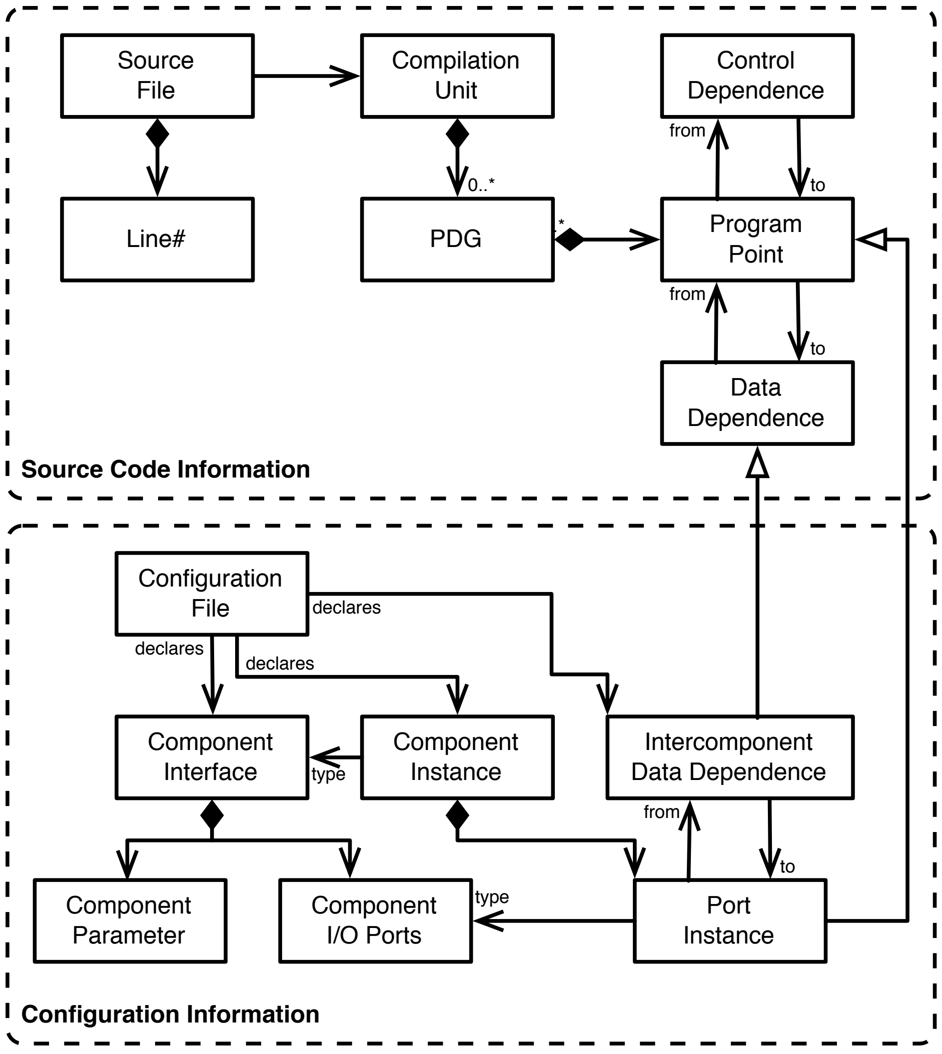


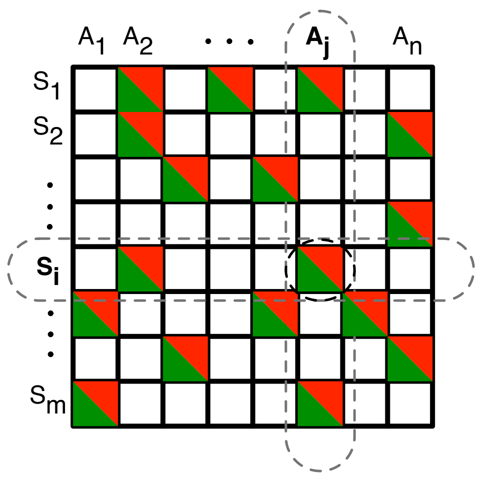
Figure 2: The main elements from various artifacts used to track information flow.

to enable easy navigation and to support various comprehension strategies [22]. We distinguish the following views:

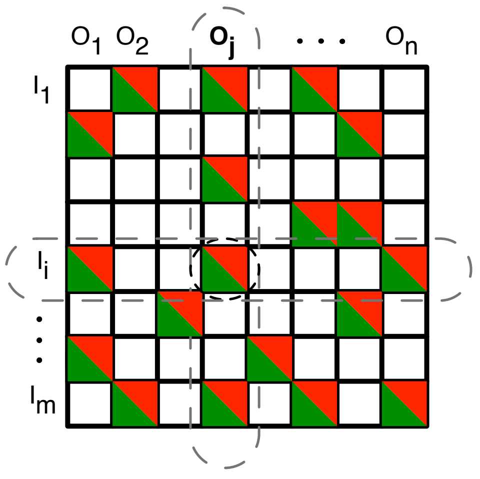
(1) System Dependence Survey: This view shows the dependencies between all system-level inputs (sensors) and outputs (actuators) in one single matrix, with sensors and actuators as rows and columns, respectively (see Figure 3A).[9] A filled cell indicates that there is at least one path along which information can flow from that sensor to that actuator. This view gives a black-box summary of the SDG that hides all details on *how* the information flow is realized. Engineers can use it to quickly find what sensors can affect a specific actuator, and vice versa.

By definition, there is no need to distinguish between forward and backward analysis directions in this view. Apart from that, although forward and backward slices

[9]Note that the views in this paper are actual views taken from the study with our industrial collaborator after some renaming for anonymization/nondisclosure purposes.



(a) System-wide dependencies



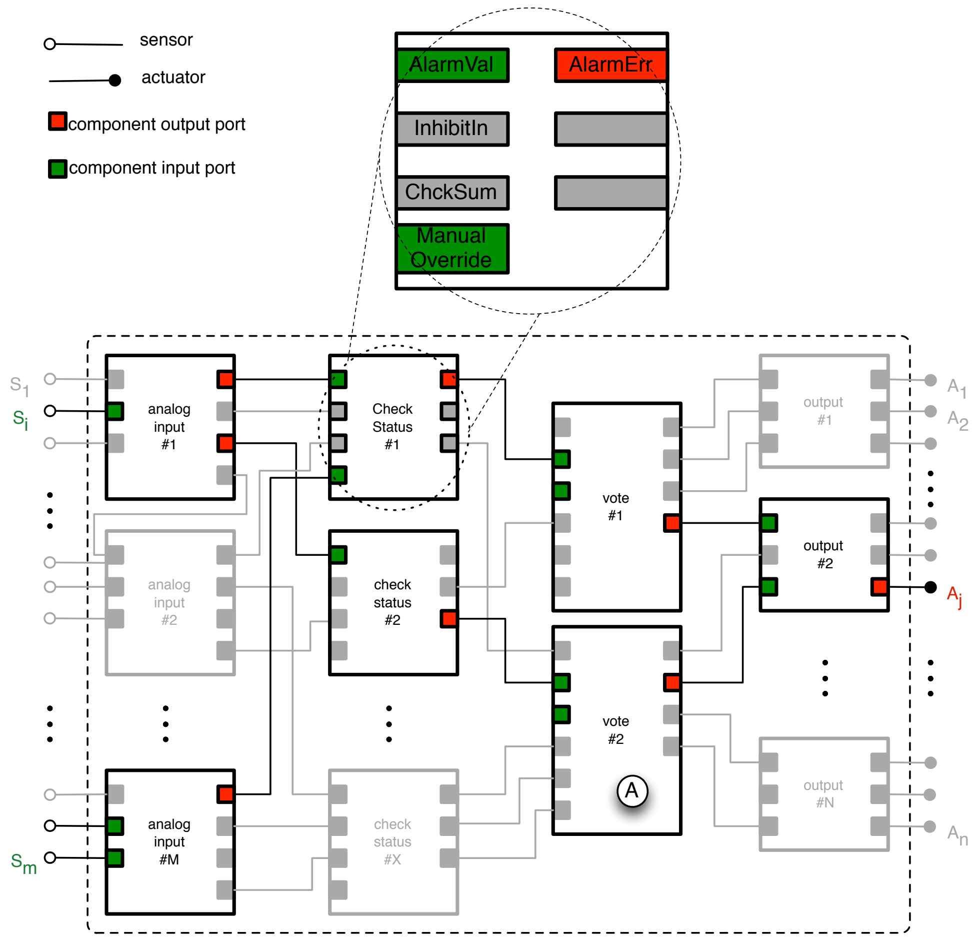
(b) Component dependencies

Figure 3: System- and Component Dependence Surveys.

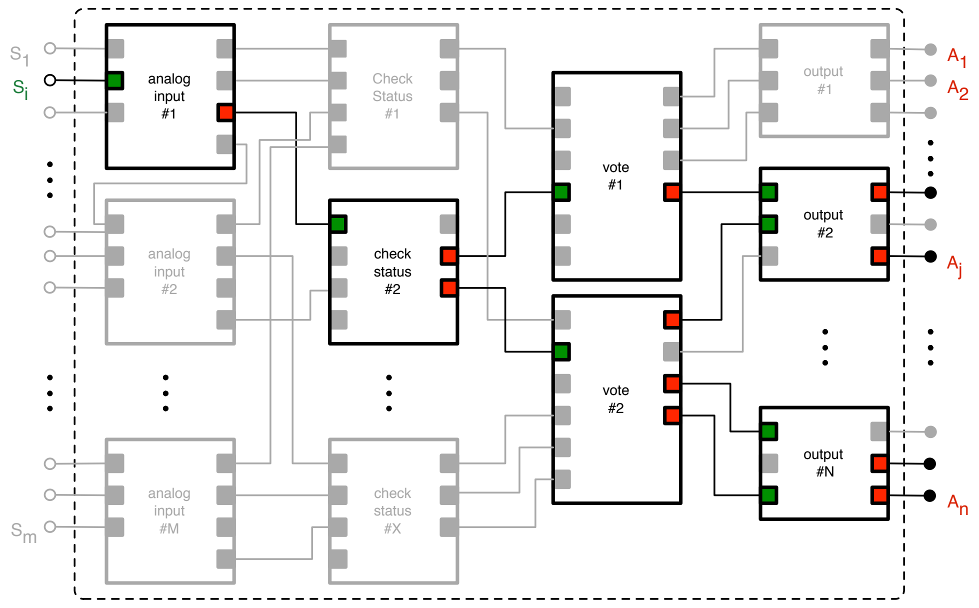
would be different in general, in this particular case were slicing is used to identify the end-to-end (abstract) dependencies between sensors and actuators, the overall system-wide view (for all end-points combined) will be the same, no matter what slicing direction is used.

The System Dependence Survey serves as a starting point for navigation. To this end, we make the matrix *active* by embedding hyperlinks to corresponding views on the next abstraction level. To provide navigation to both analysis directions on the next lower level, matrix cells are divided in two diagonally. In a given matrix cell (e.g., (S_i, A_j)), the lower left half corresponds to the respective sensor (S_i), and the upper right half to the actuator (A_j) (Figure 3a). By clicking on the lower left half of the cell, the user can zoom in on the System Information Flow for that specific sensor, in order to view the forward information flows *originating from* that sensor. By clicking on the upper right half of the cell, the user can zoom in on the System Information Flow for that specific actuator, in order to view the backward information flows that *end in* that actuator.

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



(a) Backward slice criterion: A_j



(b) Forward slice criterion: S_i

Figure 4: Backward and forward system information flows for A_j and S_i, respectively (marker A is used for explanations in the text).

of the target information flow are low-lighted, and in gray. We distinguish two variants based on analysis direction:
Backward System Information Flow: For each actuator, there is a diagram that shows the intercomponent information flow from all sensors to that actuator. The view hides all intracomponent level information in a backward slice through the SDG with actuator A_j as the slicing criterion. The result highlights the actuator and all related sensors, component instances, and intercomponent connections. Figure 4a shows an example for actuator A_j.
Forward System Information Flow: For each sensor, there is a diagram that shows the intercomponent information flows that originate from that sensor. The view hides all intracomponent level information in a forward slice through the SDG with sensor S_i as the slicing criterion. The result highlights the sensor and all related actuators , component instances, and intercomponent connections. Figure 4b shows an example for actuator S_i.

Apart from showing the elements that a sensor influences, or the elements that influence an actuator, this view serves as an intermediate level between system-level views and component-level views. It includes hyperlinks for navigation so that a user can click on a component instance to zoom in on a single component, or click outside the diagram to return to a higher level of abstraction.
(3) Component Dependence Survey: Similar to the System Dependence Survey, the Component Dependence Survey summarizes the dependencies between a component's input and output ports, using cells in a matrix (see Figure 3b). This black-box view shows which input ports can affect which output ports but hides all details on *how* the information flow is realized. Again there is no need to provide separate matrices for the forward and backward analysis directions because the summarized information is identical. There is one dependency matrix for each component, independent of its instances, because the dependencies are induced by the component source code.

To enable navigation to both analysis directions in more detailed views, matrix cells are diagonally divided in two. Clicking on the lower left half of a cell brings the user to the Component Information Flow for the corresponding input port that shows which *intracomponent* forward information flows *can be affected by* that input port. Clicking in the upper right half of a cell brings the user to Component Information Flow for the corresponding output port that shows which *intracomponent* backward information flows *can affect* that output port.
(4) Component Information Flow: For a given component and input- or output port, this view shows the intracomponent information flows connected to that port (i.e., there are diagrams for each input port and for each output port of every component). In addition to the input and output ports involved, the graph includes all conditions that control the information flow between those ports. We distinguish two variants based on analysis direction:
Backward Component Information Flow: For each output port, there is a diagram

that shows the intracomponent information flow from all input ports to that output. Figure 5a shows an example with output port “AlarmErr” as the slicing criterion (single red node at the bottom). The input ports that can affect AlarmErr are at the top (green nodes), and the conditions that control the information flow are shown as yellow squares.

Forward Component Information Flow: For each input port, there is a diagram that shows the intracomponent information flows that originate from that input. Figure 5b shows an example with input port “InhibitIn” as the slicing criterion (single green node at the top). The output ports affected by InhibitIn are at the bottom (red nodes), and the conditions that control the information flow are shown as yellow squares.

Note that we have chosen to show both the forward and the backward flow in a top-down fashion with inputs at the top and outputs at the bottom to make it easier for a user to orient themselves while changing views. In addition, we combine sequences of conditions into aggregate nodes to reduce cognitive overhead. The details of this refinement are described later, in Section 4. The conditional nodes have hyperlinks embedded to navigate to the corresponding location in the source code (indicated by marker A in Figure 5b).

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

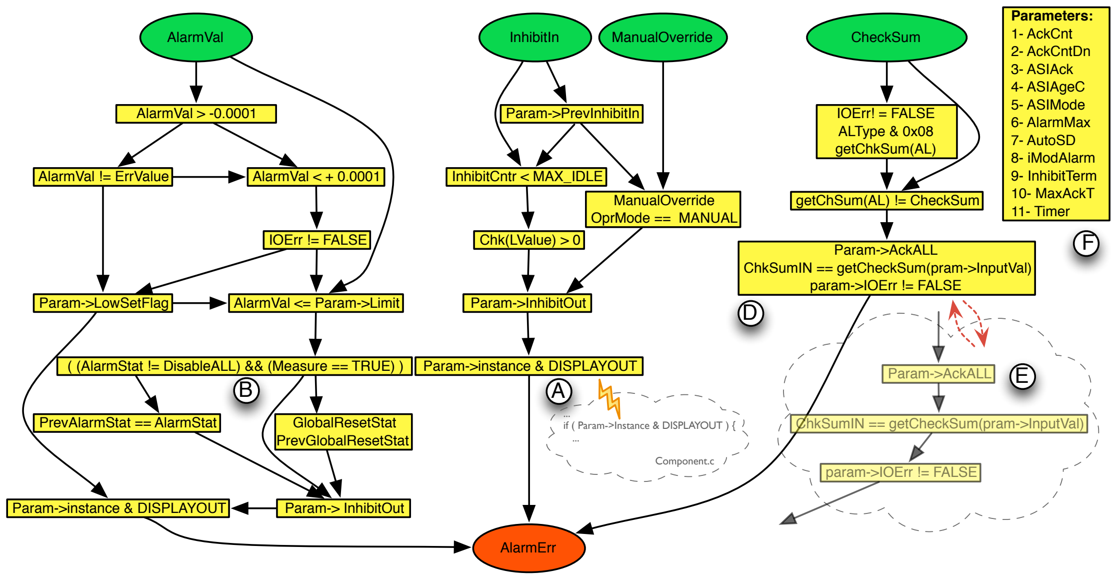
3.3 Typical Usage Scenario

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

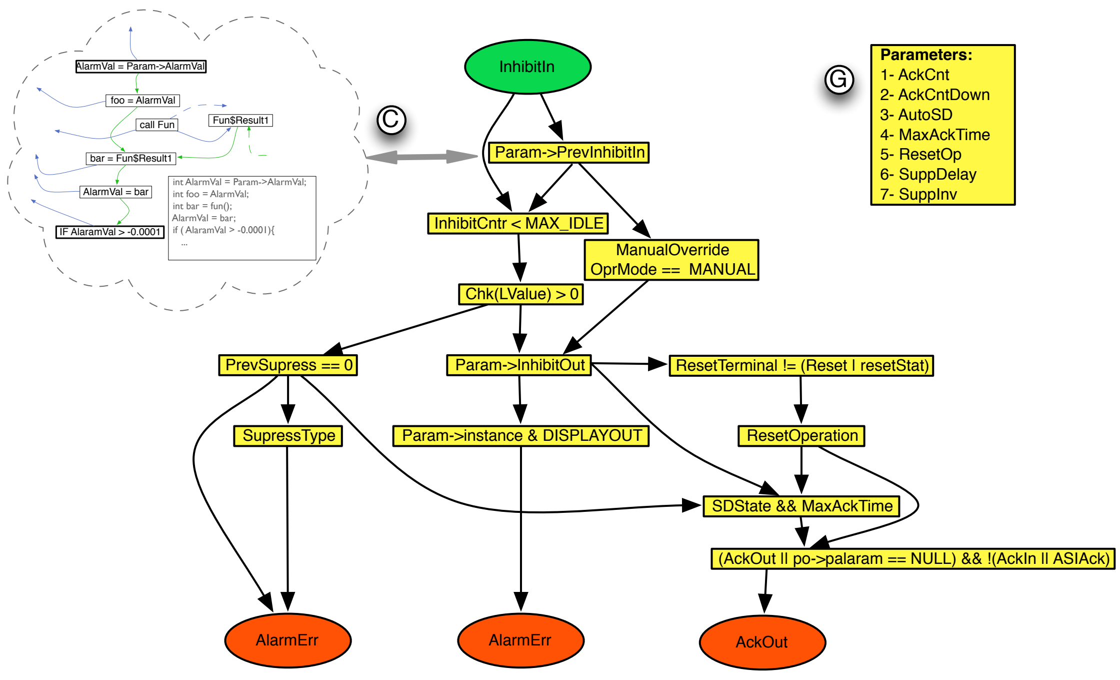
1. Users start navigating the system from the System Dependence Survey. In this view, they can immediately identify those sensors that can (or can not) influence an actuator (Figure 3a).

2. By selecting a sensor-actuator pair in the matrix, the users zoom in on the System Information Flow that helps them find the components and intercomponent connections that play a role in transferring the values from the selected sensor to the selected actuator (Figure 4). Depending on which half of the matrix cell is clicked, the users either see the outgoing information flows from a sensor (Figure 4b), or the incoming information flows towards an actuator (Figure 4a).

3. By selecting on one of the component instances, they navigate to the Component Dependence Survey. This view can be used to identify which component input ports can (or can not) affect which component output ports (Figure 3b).



(a) Backward Component Information Flow



(b) Forward Component Information Flow

Figure 5: Forward and backward Component Information Flow examples (markers A-G and the cloud-like fragments are used for explanations in the text).

4. By selecting a component input-output pair in the matrix, the users focus on the Component Information Flow. This shows the conditions that control how information from the selected input port can reach the selected output port (Figure 5). Depending on which half of the matrix cell is clicked, the users either

see the outgoing information flows from an input port (Figure 5b), or the incoming information flows towards an output port (Figure 5a).

5. Finally, the user can click on one of the conditions to navigate to the corresponding location in the source code for traceability and further (manual) inspection.

3.4 Enhanced Navigation

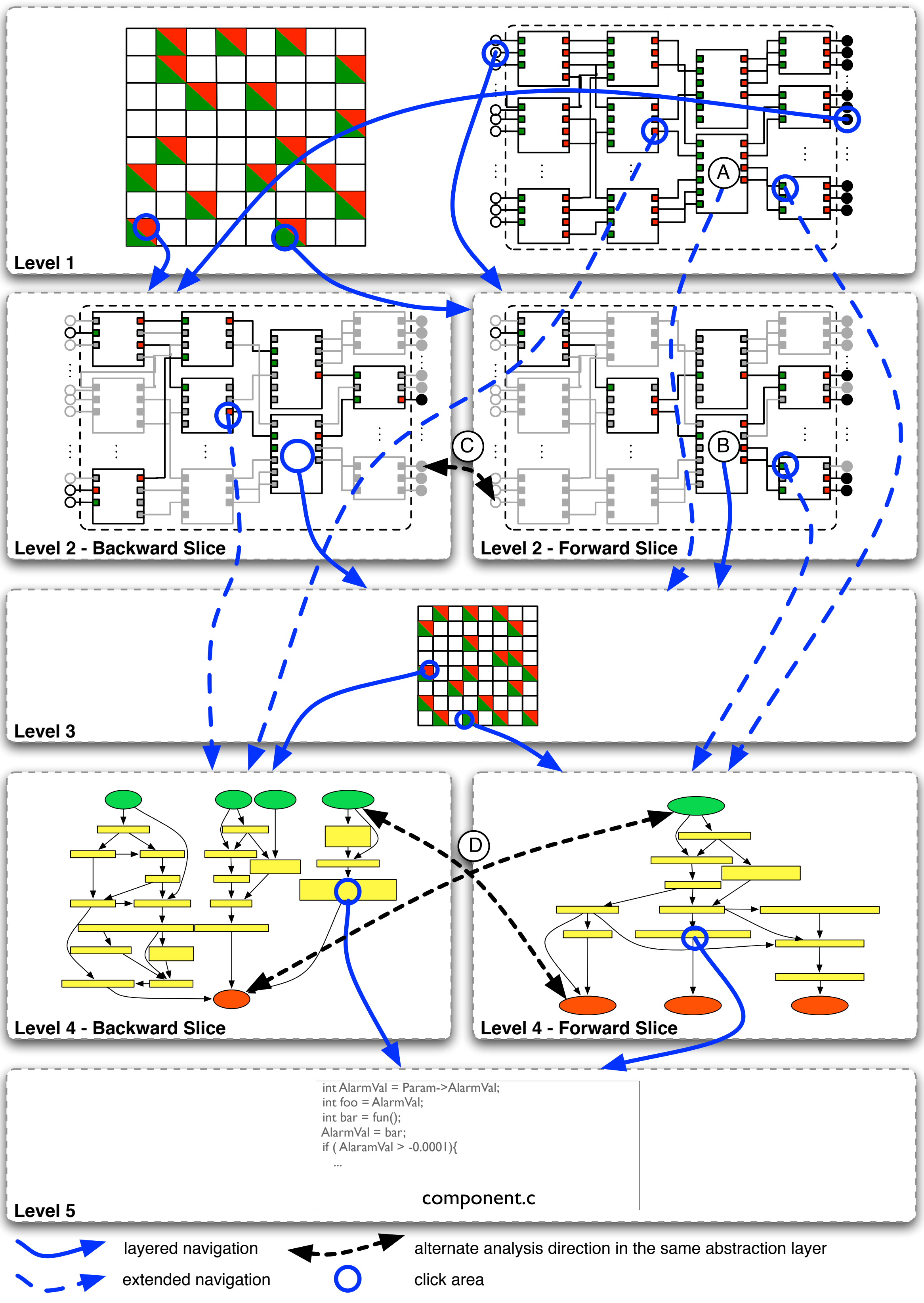
The aforementioned typical usage scenario of the visualizations supports top-down exploration and comprehension of the information flows. As the user starts from the topmost layer and descends the abstraction hierarchy, the scope of the information flows decreases (i.e., from system-wide to intracomponent),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

This highly structured navigation profile helps the novice users in finding their way though the system and prevents that they get lost during their explorations. However, requiring the user to always go through such a fixed five-layer schema would be too strong a restriction. In addition, the structure includes premature commitment to analysis direction which is know to negatively affect usability [9]. Frequent users who are already familiar with the various parts of the system, do not need the structure provided by the layered navigation. Instead, they need more flexible ways to browse and navigate the collected information [23], for example, quickly remind himself of multiple information flows halfway through a maintenance task. Forcing them to stick to a rigid and cumbersome navigation schema would reduce their overall usability experience up to a point where they might eventually abandon the tool.

To address this threat, we provide an enhanced navigation schema that allows users to browse the system more freely and more spontaneously, as well as allowing them to follow the previously described hierarchical navigation. Below, we provide a detailed explanation of the overall navigation structure with the help of Figure 6, which contains all abstraction levels and highlights the links between the levels.

First, we enhance the system dependence survey with the component composition diagram of the complete system (Figure 1). This diagram shows how the system is currently configured, with the same graphical notation as in the System Information Flow, and is rendered next to the matrix in the System Dependence Survey (Figure 6, Level 1). It can be seen as a generic System Information Flow before slicing, and thereby helps to build a coherent mental model for navigating through the system. We deliberately represent component input and output ports in the same color (green and red, respectively) in matrix-based and Component-Diagram-based visualizations to take advantage of color as a graphical beacon throughout the whole system.

Second, we *activate* almost every visualized element with navigation hyperlinks.



```
int AlarmVal = Param->AlarmVal;
int foo = AlarmVal;
int bar = fun();
AlarmVal = bar;
if ( AlaramVal > -0.0001){
   ...
```

Figure 6: The navigation structure of the visualizations. Every visualized element, except connections and edges, has a hyperlink (markers A-D are explained in the text).

In Figure 6, clicking on any element other than a connection leads to a diagram that shows the information flow with respect to that element. The following navigation rules apply consistently across all layers:

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

In addition to these rules, clicking on the graphical representation of a component (Figure 6, Level 1 and 3) leads to the respective component dependence survey (Figure 6, markers A and B). The resulting navigation schema is highly intuitive, and enables the users to navigate two or three abstraction layers in one step. Users can browse from the component configuration diagram (Figure 6, Level 1) to system information flow or component information flow by clicking system or component ports, respectively (without going through system dependence survey or component dependence survey).

Apart from that, the users are not bound to a specific analysis direction once they descend the abstraction hierarchy (i.e., forward and backward), and can alternate the direction at any step. For instance, while investigating a pair of sensor-actuator (S_i, A_j), the users can quickly view the outgoing information flows from S_i and the incoming information flows to A_j by a single click, and without backtracking to higher-level diagrams (Figure 6, marker C). The same direction switching is provided for component information flow (Figure 6, marker D).

3.5 Component Parameters

As mentioned in Section 2, our industry collaborator is a major producer of various safety and control systems. These products and systems (that is, the individual instances of a product that are installed in the real world) share considerable similarities which is exploited by assembling products in a component-based fashion from reusable components [7]. Nevertheless, there is also a considerable variation between any two concrete installations and the safety systems that monitor and control them. Examples include variations in the actual sensors and actuators that are used, and the respective thresholds at which they trigger, and process specific variables such as what levels are considered hazardous, and variations that follow from complying to different safety standards.

In the systems that we studied, these reusability and variability concerns have been addressed by developing *generic* implementations for the components that are highly customizable and configurable with the help of *component parameters*. These component parameters are used to concretize the functionality of these components in a specific installation of a specific product. The parameters can be set either at system configuration time, or at execution time by user actions. In either case, they can be regarded as *input* to the components; however, they are separate and different from component input ports. The set of each component's parameters is declared in (XML-based) configuration files, similar to the component's ports. For example, the user can prescribe the so-called *operation mode* of (part of) the system in some cases. When the user sets the operation mode to "Test Mode," the "Alarm" output of the components is treated as normal in the computations of the system; however, no actuator is engaged (e.g., no alarm bell goes off). This operation mode is communicated to the components as a parameter and is not considered for determining the normal (process-specific) sensor-actuator information flows.

Because parameters can significantly affect the behavior of components and the way in which they transfer information from their input to output ports, it is essential that developers are aware of, and understand, the parameters' potential influence on the intracomponent information flows. In addition, even though the parameters are described in the component documentation, there can be many parameters for a component and not all of them are relevant for every information flow. Moreover, the discussion of the states and effects of a parameter is scattered over various use-case specific sections, making it difficult to get a comprehensive overview. We propose to highlight the parameters that can affect a given intracomponent information flow in the visualization of that flow. To this end, we enrich component information flow diagrams with a table showing all component parameters that participate in that information flow (Figure 5, markers F and G).

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

Table 1: Component Parameters

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

4 Prototype Implementation

This section discusses the implementation of the approach described in Section 4 in a tool named FlowTracker. We distinguish three stages in the implementation, detailed below:
Model Reverse Engineering: We reuse our earlier tool to reverse engineer system-wide dependence graphs (SDGs) from source artifacts [29]. It builds on Grammatech's CodeSurfer [2][10] to create component dependence graphs (CDGs) for the individual components. Next, these CDGs are traversed using CodeSurfer's API to inject them into OMG's Knowledge Discovery Metamodel (KDM) [16]. The traversal uses the Java Native Interface to drive KDM constructors in the Eclipse Modeling Framework. For each program point, we include a pointer to its *origin* in the source code for traceability. Next, we use Xalan-J to analyze and transform the system configuration artifacts into the intercomponent dependence graph (ICDG). Finally, we use a straightforward substitution transformation to integrate the CDGs with the ICDG, and create the final SDG.

As mentioned in Section 4, all dependencies in the SDG need to be interpreted in both forward and backward direction to compute the various information flows. In our original reverse engineered SDG, each dependency is represented by an instance of a stereotyped ActionRelationship class in KDM, where stereotyping ActionRelationship is used to distinguish between different types of dependencies, such as data-, control-, and intercomponent dependencies (for more details on the mapping of SDGs into KDM, we refer to [29]). Unfortunately, KDM does not support ActionRelationShips that can be interpreted bidirectionally. To enable slicing by 'natively' traversing the KDM representation, we therefore choose to represent each dependency in our SDG by two (uni)directional ActionRelationShips, one for each direction. A downside of

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

this implementation decision is that the number of ActionRelationship objects doubles. This increases the model size, which has a tangible effect on the initial model loading time. On the upside however, there is no cost penalty for computing the system-wide slices, because the traversal of the dependencies in each direction is independent of the presence of dependencies of the opposite direction. This clearly outbalances the alternative where, on average, for half of the slicing computations the inverse dependence relation would have to be computed.
View Construction: During view construction, we enrich the SDG with additional *summary edges* and *aggregate nodes* that capture a number of view-specific abstractions in the presentation stage. Alternatively, we could have defined several "destructive" transformations that create a new model for each view, but we prefer to enrich our SDG model to reuse information between views. Our implementation builds on a simple slicing tool in Java that we have created as part of our earlier work. Subsequently, we discuss the abstractions that were added for the various views. The names were chosen so that they map trivially on the names of the views in Section 3.2.

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

For each system port P, the relation $\mathsf{SysInfoFlow}_{\mathit{direction},\mathsf{P}}$ is based on slicing the enriched SDG on P with the correct direction. The direction has to be backward for the actuators, and forward for the sensors. For each component C_i in the slice, we use the summary edges of $\mathsf{CompDep}_{\mathsf{C}_i,\mathit{direction}}$ to hide the internals of C_i. What remains of the slice are summary edges for the connections between (ports of) the component instances involved and connections from the incoming sensors and toward the actuator. Note that it is *not* possible to compute this information by simply slicing the ICDG, because the ICDG does not contain information about the dependencies between a component's input and output ports.

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

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

2. We replace edges by summary edges that subsume all nodes that are not input ports, conditions, or output ports (Figure 5, marker C). For example, when we have edges (x,y) and (y,z), and y is not an input port, condition, or output port, we

replace both edges (and node y) by a single summary edge (x,z). These summary edges are computed transitively, so that they represent the longest path possible.

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

Finally, the construction of the Implementation View does not require any additional summary edges or aggregate nodes to be added to the SDG.
Presentation: We present the results of our System- and Component Dependence Surveys as matrices that have been implemented as HTML tables with input and output ports as rows and columns, respectively. This presentation is intentionally chosen to resemble our industrial partner's specifications of the safety logic, known as Cause & Effect matrices, to enable easy comparison of the implemented dependencies with the specified safety logic. The matrices are made *active* by embedding hyperlinks to the corresponding views on the next-lower abstraction level. By clicking one of the cells below a port or actuator, the user can zoom in on the information flow *leading to* that port or actuator.

We use the KDM API to traverse the view-specific summary edges in our enriched SDG and transform the elements of interest into GDL, a graph description language that can be processed by the aiSee graph layout software.[11] We use GDL's provisions for collapsable subgraphs to represent conditional clusters and their aggregate representation so the user can go back and forth between these representations. We include navigation between views by embedding hyperlinks in the nodes representing components and ports. Similarly, we provide traceability by embedding hyperlinks to source code locations in Component Information Flow nodes representing conditions. These hyperlinks are preserved when aiSee computes the layout and renders the graph in Scalable Vector Graphics (SVG) format.

Finally, we create a pretty printed version of the source code by using Doxygen.[12] Doxygen is a source code document generator for numerous programming languages, including C. It can be configured to include the source code as part of the generated documents in HTML format and embed various hypertext navigation features.

A positive side-effect of implementing all visualizations in HTML is that we inherit all benefits of the familiarity and features of modern web browsers as part of our user interface. These browsers are widely available and well-know to all prospective users of FlowTracker. Moreover, they provide standard navigation features such as browsing

[11]http://www.aisee.com/
[12]http://www.doxygen.org/

history and bookmark creation that help users to maintain a breadcrump trail and to store landmarks or points-of-interest for later recall. This helps the users to maintain awareness about of their position, keep an overview of where they have been, and backtrack to earlier locations without considerable burden on their own memory, and supporting various strategies for comprehending software systems [23].

5 Discussion

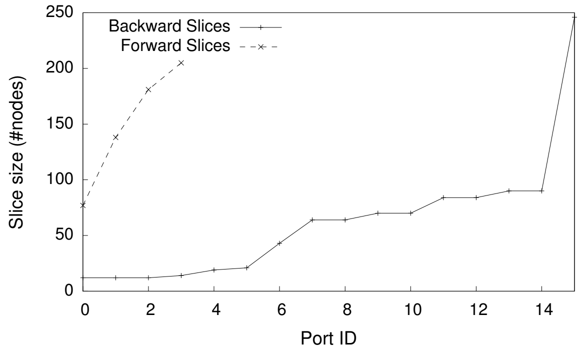
5.1 Static versus Dynamic Analysis

Our approach is based on static analysis of the system which, by its very nature, computes an approximation of the actual relations that exist in a system at runtime. In theory, more precise information could be obtained by using dynamic analysis, which aims to capture exactly those relations that can be observed on a running system. However, in the context of software intensive control systems, there are a number of limitations: First, in-vivo dynamic analysis of these systems in their real operating environment is generally not an option, due to safety hazards. Second, in-vitro dynamic analysis of such systems requires advanced, expensive, stubs and simulators to replace hardware components and to create realistic execution scenarios, which is typically only available at limited development sites. Finally, this infrastructure for in-vitro analysis is generally in high demand for product development and testing. Faced with these limitations, we set out to investigate an alternative approach based on static analysis.

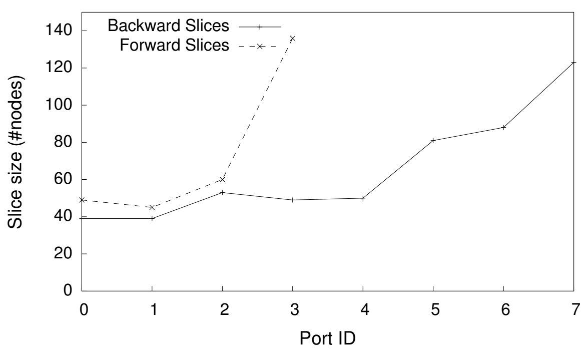
Since the introduction of static program slicing [27], there have been several improvements to compute more *accurate* slices [see e.g., 20, 25]. However, static program slicing remains a *conservative approximation*, i.e., there might be statements in a slice that have no relation to the slicing criterion. This characteristic has the following effects on the information flows that we compute: (1) They are *safe*; conservativeness guarantees that no dependency goes undetected between component input and output ports, and eventually between system inputs and outputs. Therefore, the users can be assured that when FlowTracker does not show a dependency between a pair of system (or component) input and output, there is no possibility of influence from that input to the output. (2) They may contain *false positives*; it is not guaranteed that a reported input actually *does* influence the value of a given output. In other words, the extracted information flows are a superset of the actual information flows.

5.2 Forward versus Backward Slicing

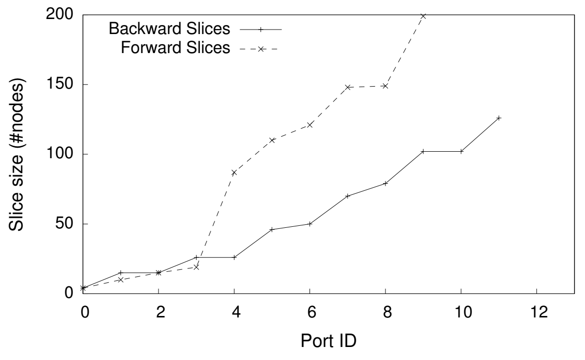
As discussed in Section 4, addressing the users' information needs requires computing both forward and backward slices through the SDG to determine infomation flow. Apart from discussing the suitability of forward and backward slices as we did in that chapter, here we would like to highlight some observations on these two slicing directions, and



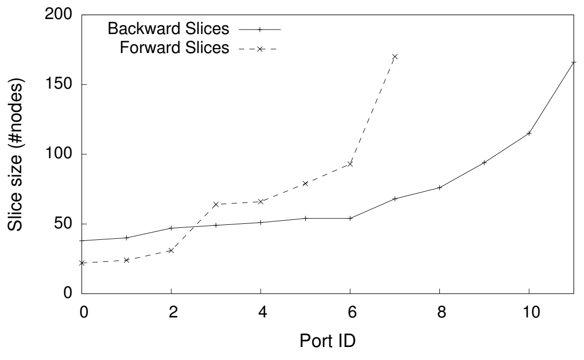
(a) Component 1



(b) Component 2



(c) Component 3



(d) Component 4

Figure 7: A comparison of intracomponent slice sizes in the forward vs. backward direction. Forward slices are computed from the component input ports, and backward slices from the output ports. Component ports are sequentially numbered on the horizontal axis (starting from zero), in the order of ascending slice sizes (vertical axis).

especially focus on the differences in slice sizes, as this directly affects the cognitive load involved with understanding the various diagrams that we create.

From a black-box point of view, the effects of both directions of slicing are the same, i.e., they detect the same abstract dependence relations between inputs and outputs, as shown in the system dependence survey and component dependence survey matrices. This is true of system-level slices, as well as component-level ones. This observation in our diagrams is a direct implication of the definition of dependence graphs [12]. If a program point P1 is included in the backward slice from program point P2, then P2 is definitely included in a forward slice from P1. Following the same argumentation, the *average* size of forward and backward slices is the same. However, the *distribution* of slice sizes does not have to be the same [3], which makes it interesting to investigate a bit deeper.

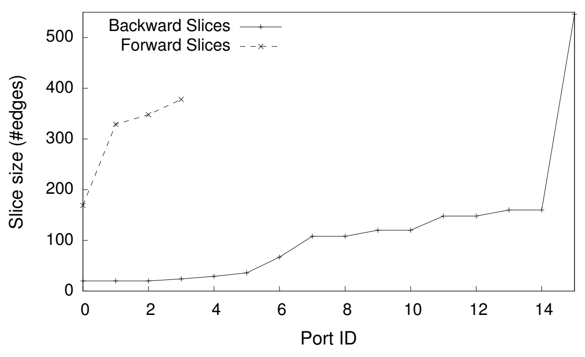
From a white-box point of view, i.e., from the perspective of *intracomponent* slices, one *can* observe differences between forward and backward slices. Figure 7 depicts the slice sizes for a randomly selected subset of the components studied. Slice sizes are measured by the number of program points in each slice. As mentioned before, forward slices are computed from component input ports, and backward slices from component output ports. Component ports are indexed on the horizontal axis in the ascending order of the slice sizes, which is projected on the vertical axis.

In the selected components in Figure 7, forward slices are clearly bigger than backward slices. Consequently, we observe more complex diagrams in component information flow for forward information flows than backward information flows, which can be considered an obstacle for comprehension. This observation confirms our initial intuition to extract information flows based on backward slices to facilitate comprehension [30]. The same observation could also apply to the number of dependencies (a.k.a. edges in the SDG) included in the forward and backward slices (see Figure 8). The number of edges is not commonly used in measuring slice sizes. However, this number is of interest to us, as it has an indirect, yet major, impact on the complexity of the final diagrams in component information flow.

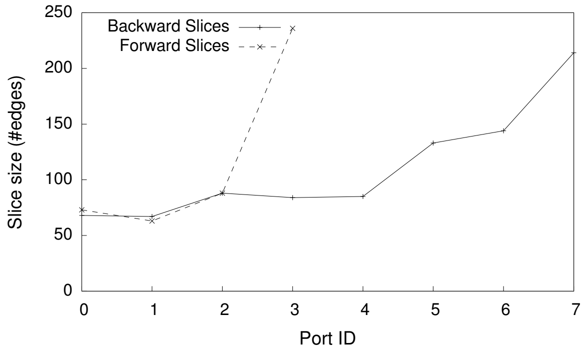
At first, these observations seem to contradict to Binkley and Harman's empirical study which concludes "forward slices are smaller than backward slices" [3]. In this study, the authors provide evidence that the *distribution* of slices sizes for forward slices leans toward smaller numbers compared with backward slices. Their claim is strongly supported by: (1) Computing both forward and backward slices from *every* program point; and (2) Computing forward slices from all inputs and backward slices from all outputs. They gather statistically significant data from a large code base containing a wide range of programs (accumulating over 1 million lines of code), which averages out the majority of architectural- and source code specific characteristics that could affect their conclusions.

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

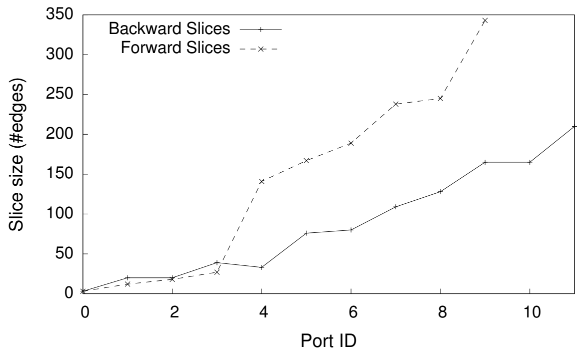
The difference in the number of input and output ports might not be the only reason behind the difference in forward and backward slice sizes. Certain other characteristics of the components' source code could be a complementary reason: Binkley and Harman [3] show that the effects of control dependence are the major cause of difference between forward and backward slice sizes. They propose an unproved conjecture that the "tree



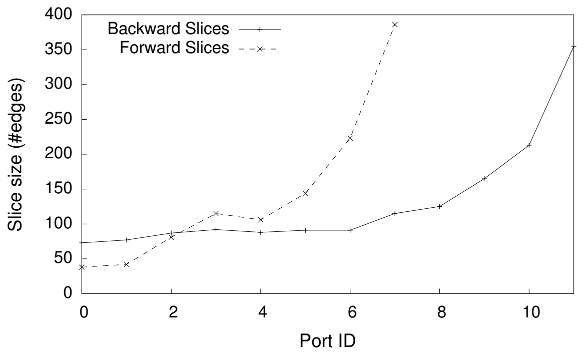
(a) Component 1



(b) Component 2



(c) Component 3



(d) Component 4

Figure 8: A comparison of intracomponent slice sizes in the forward vs. backward direction. In all subfigures, the size of the slice in measured by the number of included dependencies (a.k.a. edges in the SDG). Component ports are sequentially numbered on the horizontal axis (starting from zero), in the order of ascending slice sizes (vertical axis).

like" structure of control dependencies in SDGs, which has roots in the typical control statements in structured programming (e.g., "for," "while," and "if" statements), is the decisive factor behind the aforementioned difference in slice sizes. Closer inspection of the source code of the components sampled in Figure 7 indicated that they do *not* contain loop structure, but include 490 conditional statements in total ("if" and "else if").

This lack of loop structures is indeed one of the special code characteristics of the system under study and follows directly from the way in which these systems were architected: In general, component parameters (see Section 3.5) are used to configure the conditional clauses that decide how the data in input ports should be directed to the output ports (see Section 2). This design shows the "data-oriented" nature of the majority of interactions in these systems, on both the intra- and inter-component levels that could also explain part of the the gap between our observations and those by Binkley and Harman [3].

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

6 Evaluation

To evaluate our approach, we consider the following three aspects: accuracy, scalability, and usability. In a context of software certification, the accuracy of our views is of utmost importance and is determined by the accuracy of our model reconstruction and of our slicing tool. Both were evaluated in detail in [29] and showed 100% accuracy when compared to gold-standard results from CodeSurfer. The same paper also reported that these steps show linear growth of execution time and model size with respect to program size. This indicates good overall scalability, as the views that we construct in this paper are all projections of this model (i.e., smaller in size).

In the remainder of this section, we focus on the results of a preliminary qualitative study assessing the *usability* of FlowTracker, and, in particular, of the proposed views.
Study Design: Considering that FlowTracker is still a prototype in early stages of development, our goal is to conduct an *exploratory study* to evaluate the usability and the effectiveness of the visualizations, and their fitness for the needs of our industrial partner. To this end, we conduct a *qualitative evaluation* of the tool with a group of six subjects that were selected so that we would cover the different roles of prospective FlowTracker user groups. We use such a pre-experimental design, because it is a cost-effective way to find out the major positive and negative points, and identify missing functionality and required improvements before the tool can be adopted by our industry partner [6]. In addition, this design limits the overhead and impact of our study on the industrial partner, and it decreases the influence of (negative) *anchoring effects* that can rise from having early prototypes evaluated by people that should later adopt the tool [26] (this effect can be paraphrased as “first impressions are hard to change”). This is an important consideration for a domain-specific tool dedicated to a specific audience, like ours.

We conduct two rounds of evaluation, corresponding to the two versions of FlowTracker we have developed so far (V1.0 and V1.1). In the first round we evaluate the core functionalities of FlowTracker (V1.0). As V1.1 is an incremental iteration of FlowTracker where features were only added or extended, and not removed, the second round of evaluation focuses on the delta of the two versions. The new features of V1.1 include: (1) Supporting both forward and backward analysis of information flows (Section 3.2); (2) Inclusion of component parameters in component information flow (Section 3.5); and (3) Providing an enhanced navigation schema (Section 3.4). In addition to evaluating the deltas, we include a number of "overall" questions to capture a holistic view of the positive and negative aspects of FlowTracker usability. For the benefit of cohesion and readability, we present the results of the two rounds of evaluation together. However, we distinguish between the two rounds of evaluations whenever necessary to maintain accuracy.[13]

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

We recruited the other three subjects (P4 to P6) from colleagues who were in the final stages of their PhD studies on model-based software verification and validation at Simula Research Laboratory. These subjects are very familiar with component-based design, model-driven engineering, verification and validation, but they have no previous exposure to the case study system. However, each of them had two to four years of industrial experience prior to starting their doctoral studies, so we refer to them as junior developers. We include this second group of subjects with a different perspective to decrease the potential bias toward the specific traits of the case study, a bias that could be caused by only selecting subjects from our industrial partner [15].

Preparation – In both rounds, all evaluation sessions were conducted independently of one another, and the results were aggregated after all participants finished the evaluation. Each session started with a brief presentation of FlowTracker ($\sim$10 minutes). The presentation included a walk-through of a typical usage scenario, similar to Section 3.3. The junior developers were given an extra presentation on the system studied, to clarify the problem statement and the goals of the study. Next, we let the participants play around with the tool until they felt confident in their understanding of its functionality. We concluded this training session with three hands-on exercises, that participants had to complete before starting the evaluation. The exercises were designed in a way to engage all the views and the major features of FlowTracker. There were no time limits to complete the exercises, and discussion was stimulated. In both rounds, we continued training until the participants acknowledged full confidence in their ability to work with FlowTracker, before switching to the actual evaluation.

Data Collection – The evaluation itself consists of a *structured interview* guided by a

[13]Note that the initial results reported in [30] were exclusively from the first round of evaluations on FlowTracker V1.0.

questionnaire. The questionnaire consisted of 30 closed questions that used a 5-point Likert scale and 6 open (discussion) questions in the first round, and 24 Likert scale questions in the second round. Questions where both positively and negatively phrased to break answering rhythms and avoid steering the subjects [17]. In total, each session lasted between 60 and 90 minutes in the first round, and 45 to 60 minutes in the second round.

Researcher-administered interviews were chosen over self-administered questionnaires to elicit as much feedback as possible. Participants were instructed to bring up any question or comment during the training exercises, questions, and the open-ended discussion, similar to think-aloud sessions. Based on the answers, the interviewer included relevant follow-up or clarification questions. We recorded the complete audio of the sessions (training+interviews), and transcribed and analyzed them using the ELAN multimodal annotation tool [5]. This allowed us to collect the answers to our questions, find deeper reasons behind those answers, and get more insights into the preferred interactions with FlowTracker.

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

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

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

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

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

With respect to the use of each half of a matrix cells as a way to zoom into respectively forward and backward information flows (a feature only in V1.1), the general response was highly positive. All participants viewed this feature as an added value that outweighs the additional complexity on the user interface ("has far better functionality," according to P4). Two of the junior developers (P5 and P6) wanted more visual aids on the graphical user interface to help the users. They believed that having different cells "creates the expectation" that the cells would lead to different places. However in FlowTracker's matrices, the *right-half* section of all the cells in a single *column* leads to the same diagram representing the information flows leading to that *output* (i.e. there is redundancy in the user interface). Likewise, the *left-half* section of all the cells in a single *row* leads to a single diagram that represents the information flows originating from that *input*. In this situation, P5 and P6 would like the user interface to emphasize all elements that point to the same destination, e.g., by highlighting all "equivalent" elements at the moment one of them is "hovered" over by a mouse. At one point during the evaluation, P6 indicated that she would like to be able to click on the *title row* and the *left-most* column of the matrix to navigate to the corresponding forward and backward information flows (i.e., directly on the port names instead of the matrix cells). However, she later reconsidered her choice, fearing that the users would have to *drag* the mouse too much to be able to switch between forward and backward information flows in larger matrices. We foresee that both these requests can be easily, and transparently, addressed with some additional client-side scripting and further extending the enhanced navigation scheme.

(2) System Information Flow: The subjects were generally satisfied with the functionality of this view, that indicates which components, ports, and sensors can affect the value of a given actuator, and in V1.1 also indicates which of these elements are affected by a given sensor. FlowTracker currently shows all components and ports, and *highlights* only those elements that participate in the target information flow; the others are dimmed. An alternative could be to hide the unused elements in the diagram. Most subjects favored the current design. P5, for example, remarked that *"this view gives me the big picture as well as the micro answer."* However, two subjects had some reservations about the amount of information shown in this view; P4 and P6 were concerned that the extra information could lead to confusion. All subjects were positive about the idea of adding more interactive facilities, such as an option to include or hide the dimmed elements on demand in this view.

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

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

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

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

Considering the forward and backward information flows together, one could argue that there is a certain degree of redundancy in the visualizations (i.e., portions of information flows could be repeated multiple times in forward and backward system-wide information flows in V1.1). However, none of the participants regarded this as a problem and believed both directions of information flows are necessary to visualize.

(3) Component Dependence Survey: Similar to the System Dependence Survey, the subjects agreed that this view adequately summarizes the dependencies between input and output terminals. P3 (safety expert dealing with system certification) regarded this view as *"top priority for the certification process and a facilitator of the discussions with the third-party certifiers."* Module developer P1 stated that he *"must know the input/output relations of the components at all times, but I currently only have the source code to read and hopefully find out about all dependencies."* P1 did not expect that this view would be beneficial for the certification process, but he emphasized that he had not been directly involved in the certification process. P6 preferred that the matrix would distinguish between the data dependencies and control dependencies between inputs and outputs; input terminals whose *value* is transferred to the output terminals appear differently from the inputs whose value is used to *control* the information flow toward the same output port.

Considering FlowTracker V1.1 and using the matrix cell halves for navigation, the feedback was consistent with the feedback we received for the System Dependence Survey.

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

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

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

The subjects would like to see more interactive facilities, especially measures to better deal with the larger diagrams. In addition to zooming, another concrete suggestion was to have the option of seeing exclusively the information flow that starts from a single (selected) component input port. We foresee that many of these requests can be fulfilled quite easily by incorporating a graph viewer that is better than the one that is now used in the prototype.

With respect to the visualization of forward and backward information flows in FlowTracker V1.1, all participants gave highly positive feedback and mentioned that this helped them better with answering program comprehension questions. This positive feedback for including both analysis directions was consistent over all layers of abstraction.

Component Parameters: A new feature in FlowTracker V1.1 is that the component parameters that are relevant to an information flow are shown in the view (see Section 3.5). All participants, especially P3, regarded component parameters as highly important for the better comprehension of the intracomponent information flows, and therefore, appropriate to be visualized. In the first round of evaluations, before this feature was added, P3 regarded them as a top priority to visualize. Filtering out the irrelevant component parameters for each information flow was seen as highly positive for the comprehension by all participants. Considering the large number of parameters for each component, and the sharp decrease that is achieved by filtering the relevant parameters for each information flow (as shown in Table 1), this is hardly surprising (especially since the only alternative is manual inspection).

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

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

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

In particular, subjects considered the hyperlinks from conditions in the Component Information Flow diagram to the respective locations in the source code beneficial for

comprehension and traceability. They were less sure that these links would support certification purposes equally well: P4 and P6 said they are useful only if the certifier knows the source code (which they thought unlikely); P1 considered the links beneficial; P2 and P5 refrained from answering this question, since they felt unsure about the certification process. Safety expert P3 said that *"certifiers generally do not look at the source code, but in the worst cases where they want to see more evidence, these links will help to find the right locations."*
Overall Experience – All in all, the subjects were positive about the intuitiveness of the tool, as they *"did not need to learn a lot of things before being able to work with FlowTracker"* and *"did not feel that the tool was complex."*

All participants believed that providing the users with visualizations that represent forward information flows, as well as backward information flows, is highly beneficial for comprehension (V1.1). Unanimously, the participants said that being able to distinguish between the following two closely related questions helps them to find: (1) what elements have an effect a given output?, and (2) what elements are affected by a given input? The benefit was acknowledged both at the system-level, and component-level information flows. P4 also mentioned that having access to both directions of information flows is not only beneficial for comprehension, but also helps the users to *"follow the flow more easily"* and readily.

All participants regarded the enhanced navigation schema (see Section 3.4) as a major improvement over the fixed, layered navigation method. The choices of elements that lead to other visualizations (at the same or other abstraction layer) were deemed *"intuitive"* by most of the participants. P4 found the choice of input (output) ports to jump the forward (backward) information flows very effective and *"easy to remember, but only after the first introduction to the tool"* (which could be interpreted as indicating that this may not be the most intuitive choice). Using the enhanced navigation, all participants were confident in saying that user information needs can be satisfied faster, and the overall user experience in V1.1 is better than V1.0.

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

Overall, FlowTracker received excellent feedback regarding component and system comprehension. When we look at the feedback concerning FlowTracker's support for the certification process, the results were less conspicuous, but still very positive, most notably from the industrial subjects.[14] They argued that FlowTracker supports the certification process by *"enabling discussions between the developers and safety experts,"* and *"demonstrating the safety logic that is actually implemented in a system to the external certifiers."*

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

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

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

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

A potential concern with respect to generalization is that our evaluation included only one subject for each of the different roles of module developer, system integrator, and safety expert. As such, this subject gets a dominant voice in the evaluation, and the answers may be based more on personal opinions than on what is needed for the role.

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

We have tried to limit the impact of this threat by organizing a pre-evaluation workshop, in which we asked the stakeholders to identify the most qualified senior engineers who could represent these roles in the evaluation. In addition, it turned out that subjects with a given role generally also had experience with some of the other roles, which also helps to create a more balanced picture.

7 Related Work

Maletic et al [14] identify five dimensions of software visualizations: tasks (why); audience (who); source (what); representation (how); and medium (where). Our work can be summarized as *why:* providing source-based evidence to support software certification; *who:* for safety domain experts and developers; *what:* of implementation artifacts of component-based systems; *how:* by visualizing information flow using a set of hierarchical views; *where:* on a computer screen.

Hermans et al [11] use leveled data flow diagrams to aid professional spreadsheet users in comprehending large spreadsheets. Their survey showed that the biggest challenges occur when spreadsheets are transferred to colleagues or have to be checked by external auditors. They suggest a hierarchical visualization of the spreadsheets: starting from coarse-grained worksheets, expanding worksheets to view the contained data blocks, and diving into formula view to see "a specific formula and the cells it depends on." Our work is similar in providing a hierarchical visualization of information flow, with each view having a different trade-off between scope and granularity. Another similarity is the inclusion of non-developer, domain experts as users of the visualizations. However, the analysis subject, technique, and the underlying entities to be visualized are completely different. Our work analyzes source code to infer system-wide information flows using SDGs that are based on both data flow and control flow information, while they analyze data flow dependencies in formula-rich spreadsheets.

Krinke [13] reports on various attempts to visualize program dependence graphs and slices via existing (algorithmic) graph layout tools. He proposes a declarative graph layout, tailored to preserve the relative *locality* of program points to provide a better cognitive mapping back to the source code. A survey showed that the standard representations of program slices were "less useful than expected," and the improved layout is "very comprehensible up to *medium* sized procedures," but "overly complex and non-intuitive" for large procedures. He concludes that a textual visualization of source code is essential and introduces the distance-limited slice to assign each program statement a specific color according to its distance from the slicing criterion. In contrast, we developed multiple layers of abstraction to reduce the complexity of system-wide slices and show only the information relevant to the particular task and users. We provide links between the various views that can be navigated down to a textual representation of source as a last resort.

Pinzger et al [19] use nested graphs to represent static dependencies in source code at various levels of abstraction. They follow a top-down approach similar to ours for representing information about the system, and allow users to adjust the graphs by adding or filtering information, such as adding a caller or "keep callees and remove other nodes." In contrast to our approach, which creates abstracts from fine-grained data and control dependencies, they analyze static "uses" dependencies in Java programs at a relatively coarse-grained level, considering elements such as package, class, method, method call, and field access.

We refer to our previous work [29] for a detailed discussion of work related to our method to build system-wide dependence models from heterogeneous source artifacts.

8 Concluding Remarks

Component-based software engineering is widely used to manage the complexity of large-scale software development. Although correctly engineering the composition and configuration of components is crucial for the overall behavior, there is surprisingly little support for incorporating this information in the analysis of such systems. Moreover, to get a correct understanding of a system's overall behavior, one needs to understand how the control and data flow is *interlaced* through component sources and configuration artifacts. We found that support for such a system-wide analyses is lacking, because it is hindered by the heterogeneous nature of these artifacts.
Contributions – In this paper, we address these issues by proposing an approach that supports system-wide tracking and visualization of information flow in heterogeneous, component-based software systems. Our contributions are the following: (1) We proposed a hierarchy of views that represent system-wide information flows at various levels of abstraction, aimed at supporting both safety domain experts and developers; (2) We presented the transformations that help us achieve these views from the system-wide dependence models and discuss the different trade-offs between scope and granularity; (3) We discussed how we have implemented our approach in a prototype tool; and (4) We reported on two qualitative evaluations of the effectiveness and usability of the proposed views for software development and software certification. The evaluation results indicated that the prototype was already very useful. In addition, a number of directions for further improvement were suggested.
Future Work – We see several directions for future work: First of all, we want to improve the overall user experience by adding more on-demand interaction facilities, such as zooming and hiding or collapsing groups of nodes. Such facilities allow users to be more selective in the amount and type of information they see, according to their information needs at the moment. As briefly mentioned before, we foresee that this can be achieved by using a more elaborate graph viewer than currently used in the prototype. Since the

graph presentation is done using SVG, a promising direction forward is investigating the inclusion of some additional scripting based on JavaScript libraries, such as Raphäel[15] or D3[16].

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

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

Based on the last discussions with our industry partner, there is a desire to better understand what effects changing a given parameter values will have on the information flows that can be achieved. Based on our earlier experiences with (static) value range propagation in source code for embedded systems, we do not foresee that these questions can be answered using static analysis [4]. An alternative is to extract this parameter change information using dynamic analysis methods. Leveraging the capabilities of static and dynamic analysis not only helps to increase the accuracy of the information flows, but also opens many options toward better visualization of intracomponent information flows using concrete values of the component parameters and conditional clauses at run time.

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

Acknowledgements

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

[15]http://raphaeljs.com/
[16]http://mbostock.github.com/d3/

Bibliography

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

[2] Anderson P (2008) 90% Perspiration: Engineering Static Analysis Techniques for Industrial Applications. In: IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 3–12

[3] Binkley D, Harman M (2005) Forward slices are smaller than backward slices. In: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp 15–24

[4] Boogerd C, Moonen L (2008) On the Use of Data Flow Analysis in Static Profiling. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 79–88

[5] Brugman H, Russel A (2004) Annotating Multi-media / Multi-modal resources with ELAN. In: Fourth International Conference on Language Resources and Evaluation (LREC), 2004, http://www.lat-mpi.eu/tools/elan/

[6] Campbell DT, Stanley J (1963) Experimental and Quasi-Experimental Designs for Research. Wadsworth

[7] Deelstra S, Sinnema M, Bosch J (2005) Product derivation in software product families: a case study. Journal of Systems and Software 74(2):173–194

[8] Gallagher K, Binkley D (2008) Program slicing. In: Frontiers of Software Maintenance (FoSM), IEEE, pp 58–67

[9] Green TRG, Petre M (1996) Usability Analysis of Visual Programming Environments : a âĂŸ cognitive dimensions âĂŹ framework. Visual Languages and Computing 7:131–174

[10] Hatton L (2004) Safer language subsets: an overview and a case history, MISRA C. Information and Software Technology (IST) 46(7):465–472

[11] Hermans F, Pinzger M, Deursen AV (2011) Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams Categories and Subject Descriptors. In: International Conference on Software Engineering (ICSE), pp 451–460

[12] Horwitz S, Reps T, Binkley D (1990) Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(1):26–60

[13] Krinke J (2004) Visualization of program dependence and slices. In: IEEE International Conference on Software Maintenance (ICSM), pp 168–177

[14] Maletic J, Marcus A, Collard M (2002) A task oriented view of software visualization. In: IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), pp 32–40

[15] Nielsen J, Molich R (1990) Heuristic evaluation of user interfaces. In: SIGCHI Conference on Human Factors in Computing Systems, ACM, pp 249–256

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

[17] Oppenheim AN (1992) Questionnaire Design, Interviewing and Attitude Measurement. Continuum

[18] Petre M (2010) Mental imagery and software visualization in high-performance software development teams. Journal of Visual Languages & Computing 21(3):171–183

[19] Pinzger M, Graefenhain K, Knab P, Gall HC (2008) A Tool for Visual Understanding of Source Code Dependencies. In: IEEE International Conference on Program Comprehension (ICPC), pp 254–259

[20] Silva J (2011) A Vocabulary of Program Slicing-Based Techniques. ACM Computing Surveys (to appear)

[21] Steele J, Iliinsky N (2010) Beautiful Visualization, Looking at Data through the Eyes of Experts. O'Reilly Media, p 416

[22] Storey MA (2005) Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In: IEEE International Workshop on Program Comprehension (IWPC), pp 181–191

[23] Storey MA (2006) Theories, tools and research methods in program comprehension: past, present and future. Software Quality Journal 14(3):187–208

[24] Szyperski C (2002) Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley

[25] Tip F (1995) A survey of program slicing techniques. Journal of Programming Languages 3(3):121–189

[26] Tversky A, Kahneman D (1974) Judgment under Uncertainty: Heuristics and Biases. Science 185(4157):1124–31

[27] Weiser M (1981) Program slicing. In: International Conference on Software Engineering (ICSE), IEEE, pp 439–449

[28] Weiser M (1982) Programmers use slices when debugging. Communications of the ACM 25(7):446–452

[29] Yazdanshenas AR, Moonen L (2011) Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems. In: IEEE International Conference on Software Maintenance (ICSM)

[30] Yazdanshenas AR, Moonen L (2012) Tracking and Visualizing Information Flow in Component-Based Systems. In: IEEE International Conference on Program Comprehension (ICPC)

Paper III

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

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

Amir Reza Yazdanshenas, Leon Moonen

Software Engineering Department, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

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

1 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 2 and 3 summarize related work and describe the context of our research. We present the overall approach in Section 4, and our implementation in Section 5. We evaluate our work in Section 6 and conclude in Section 7.

2 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.

3 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.

4 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 2, 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

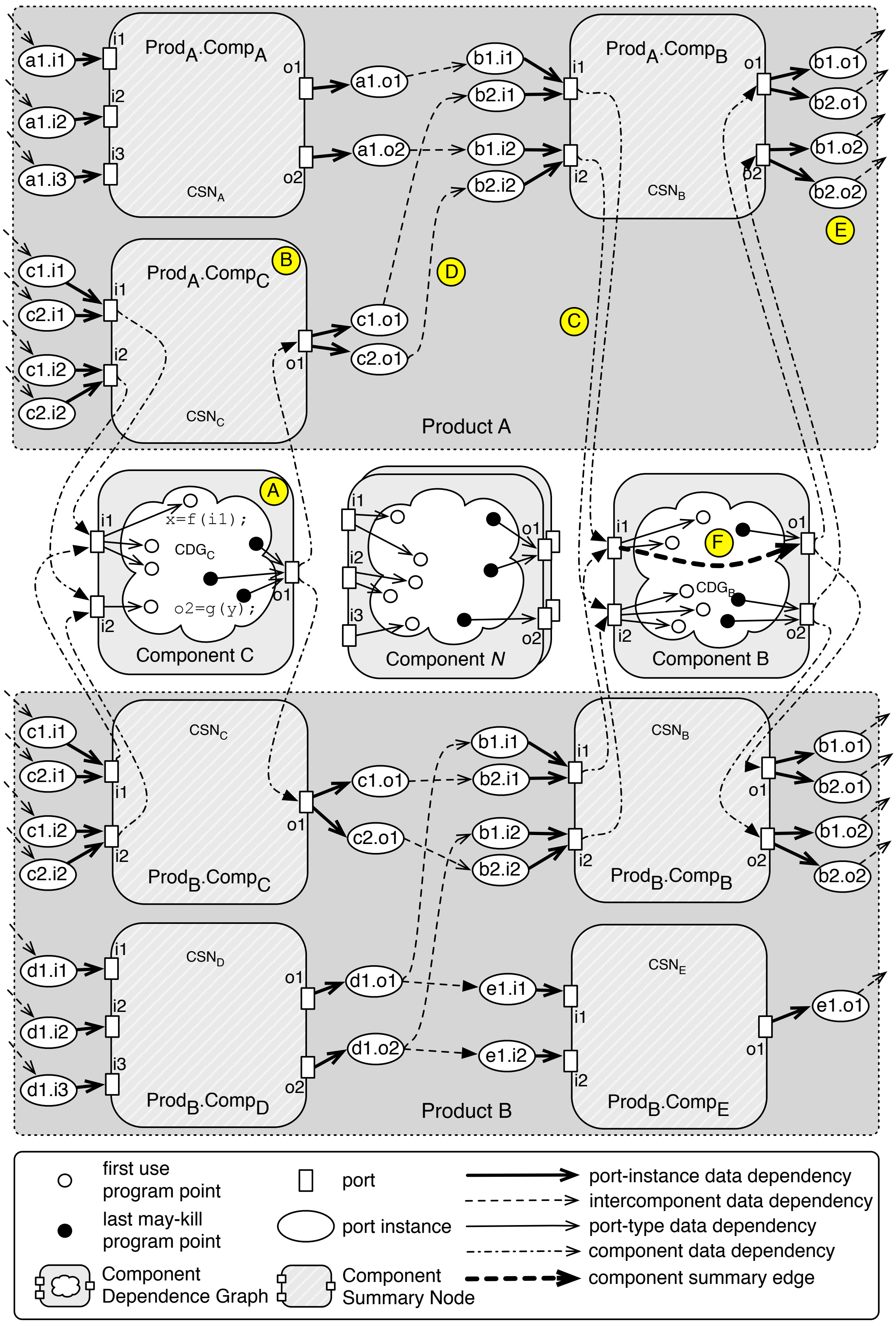


Figure 1: Family Dependence Graph (tags A–F are explained in the text)

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 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 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 downside, 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

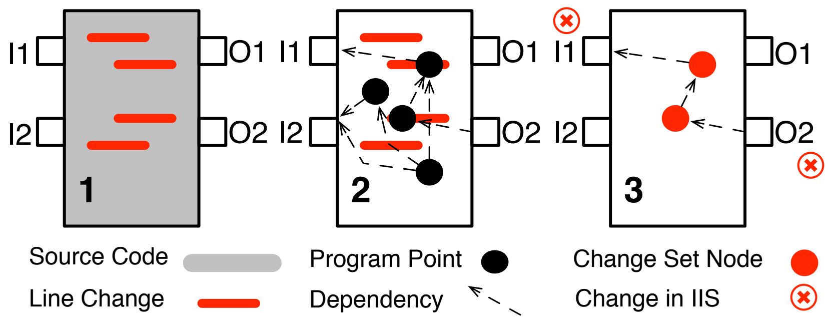


Figure 2: Detecting the CS and IIS

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: "$\mathsf{O}_1 = \mathsf{I}_1 + 1$;" to "$\mathsf{O}_1 = \mathsf{I}_1 + 1000$;". 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 3) 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.

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*

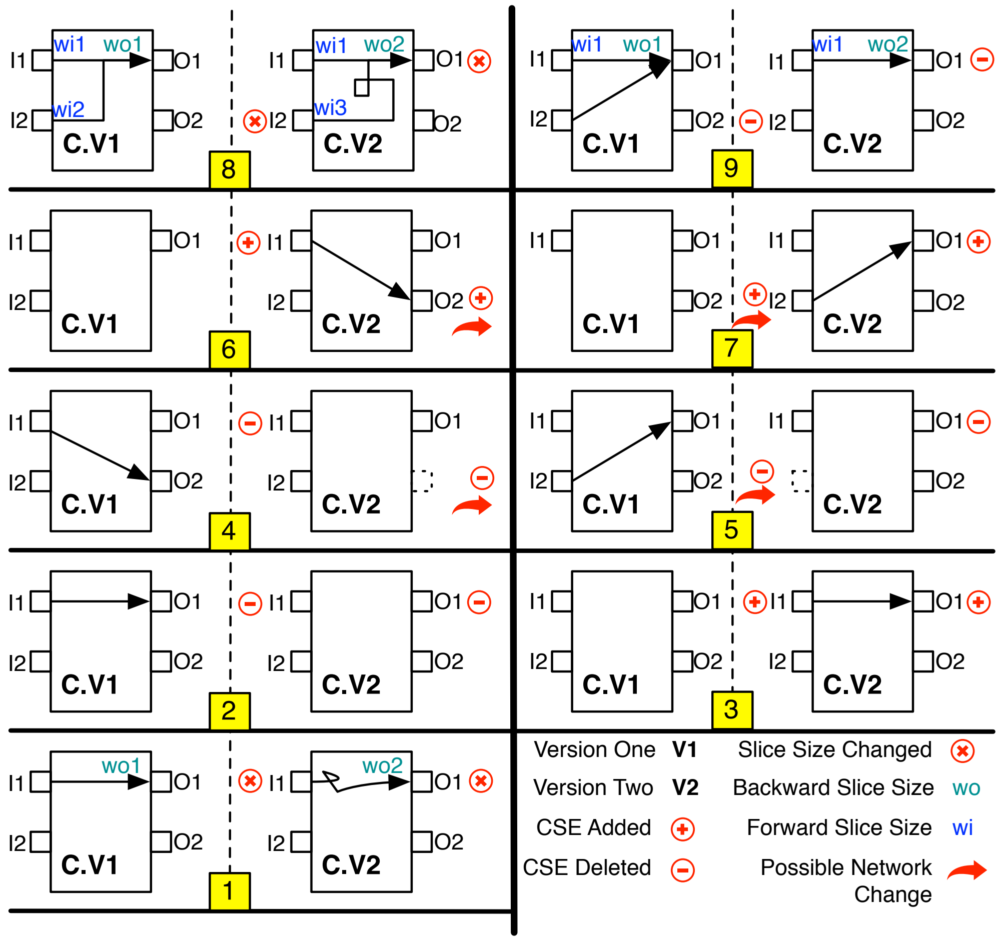


Figure 3: IIS cases and propagation of ripple effects

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: "$\mathsf{O}_1 = \mathsf{I}_1 + 1$;" and "$\mathsf{O}_1 = \mathsf{I}_2 + 2$;". In V2, we change the latter to "$\mathsf{O}_1 = (\mathsf{I}_2 > 0)\ ?\ 0 : 1$;", 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.

5 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

OMGÕs 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][17] 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 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.

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

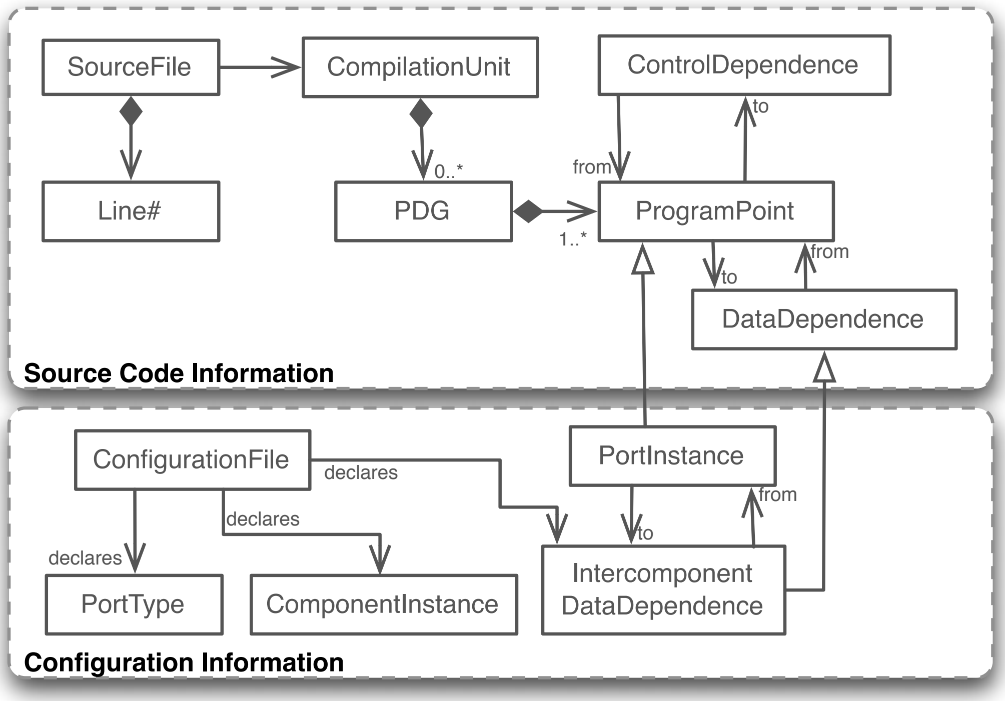


Figure 4: Meta-model describing the main elements used to track information flow across a component-based system.

6 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 experimental 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 3. 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

Table 1: 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

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 1 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 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 4, 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 5), makes PSDGs extremely lightweight. This makes (potentially frequent) executions of CIA even more cost-effective.

7 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 meta-model (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 6, 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 3). 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.

Bibliography

[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.

Paper IV

Cross-language program analysis for the evolution of multi-language software systems: a systematic literature review

Cross-language program analysis for the evolution of multi-language software systems: a systematic literature review

Amir Reza Yazdanshenas, Leon Moonen

Software Engineering Department, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

Abstract – Contemporary software systems are rarely implemented uniformly in one programming language, and delivered in one type of development artifact. Instead, it is common practice to use the programming language that fits best to develop each part of the system. Moreover, in large-scale legacy software systems, the maintainance process often has to deal with several programming languages and artifacts simultaneously. This *heterogeneity* complicates most system-wide tasks in the evolution of such *multi-language* systems, as *cross-language* dependencies and interactions are substantially more difficult to identify and manage.

This paper seeks to provide a basis for the improvement of software evolution of multi-language systems, by assessing the state of the art in cross-language program analysis, and discussing the implications for research and practice. We conduct a systematic review over the available literature in seven digital libraries, to find the relevant primary studies on cross-language program analysis, and identify additional studies with manual snowballing. We classify the studies based on several criteria, including their purpose (why), the adopted or suggested approach (how), the information leveraged in each programming language or artifact (what), and the conducted evaluation (quality). Our investigation identified 75 relevant papers, which were analysed in depth to answer eight research questions. The results include objective findings on the diversity of the applied techniques, application domains, programming languages, and reliability of the approaches. Building on these findings, several implications for research and practice are discussed, including potential breakthroughs based on use of historic data, and possible negative effects of having a shortage of community-driven research.

Keywords – systematic literature review, cross-language, program analysis, source code analysis, multi-language software systems

1 Introduction

This paper reviews the available literature on the topic of *cross-language program analysis*, which is applied to the maintenance of *Multi-Language Software Systems* (MLSS). By MLSS, we mean those systems that are implemented in more than one programming language, or a combination of programming languages and other types of software development artifacts, such as web page scripts, configuration artifacts, deployment descriptors, database-related artifacts, etc.

The domain of programming language design and implementation witnesses a constant stream of new work aimed at developing "better" languages, for example by improving the ability to define abstractions, adding constructs for particular design paradigm, or improving aspects such as expressive power, simplicity, readability, portability, extensibility, efficiency, and many more. This phenomenon has resulted in hundreds, if not thousands, of general-purpose and domain-specific languages

However, like most other engineering problems, there is no silver bullet in language design, and designers need to trade off many of the (often contradictory) aforementioned criteria. Faced with the effects of these trade-offs, it has become common to use multiple programming languages in a single software system, each solely for the purpose they were originally built for, following the idea of language-as-a-tool. In the 1990s, at least one third of the applications developed in USA were known to utilize two languages, and 10% of the applications were estimated to use three or more languages [76]. Nowadays, these percentages have grown extensively, and it is no longer surprising to see a team of software engineers use up to 30 different programming-, scripting-, markup-, and configuration languages to built large-scale software systems [77].

The progress in component-based development, popularity of domain-specific languages [78], ubiquity of web-based applications, and widespread use of software frameworks and middle-ware have all contributed to the rise in MLSS. Apart from developing new systems using multiple languages, the integration of legacy software in systems-of-systems is another source of MLSS. For all these reasons, many have perceived *language heterogeneity* as a perpetual issue in software engineering [79].

A recurring research topic in software engineering is *program analysis:* the (semi-) automated investigation of software artifacts to understand and reason about the structure and behavior of a system [80]. Program analysis is an enabling technique for numerous essential activities during software development and maintenance, ranging from efficient compilation and comprehension of a system, to more specific tasks such as refactoring and clone detection. There are a number of long-standing challenges that stand in the way of efficient and wide-scale adoption of program analysis tools in software engineering [80]. One of these challenges is dealing with the linguistic heterogeneity in MLSS, which is the topic of this systematic literature review.

This paper reviews the available literature concerning cross-language program analysis on the diversity of the applied techniques, application domains, programming languages, and on the strength of the findings. We seek to categorize the proposed approaches, and identify trends where possible. Our aim is to help researchers in identifying proven techniques in exercised areas, as well as the open areas wherein research still falls short. Moreover, we aim to support practitioners and tool builders as well, by systematically putting the available knowledge into perspective.

Following the established guidelines of conducting Systematic Literature Reviews (SLR) [81], we provide detailed documentation on the proceedings of our study. This includes, amongst others, the challenges that we faced, peculiarities of the domain literature we came across, as well as a number of "lessons learned." We have two-fold goals with this extensive documentation: (1) promoting the reproducibility of our results, and (2) contributing to the discipline of conducting systematic literature reviews in the domain of software engineering.

The remainder of the paper is organized as follows: Section 2 presents some background material, as well as the scope of this review. The review protocol is documented in Section 3, including the research questions, study selection criteria, and the process of data extraction and synthesis. The findings of our study are presented in Section 4, starting with an analysis of the publication (meta)data, and continuing with a detailed analysis of facts extracted for the primary studies, and the answers to our research questions. Our interpretation of the findings and their possible implications for future research is presented in Section 5. We discuss the limitations of our study in Section 6, before concluding in Section 7.

2 Scoping and Terminology

Aligning our study with the common terminology in evidence-based software engineering [82], "multi-language software systems" is the *population*, and "program analysis" is the *intervention* that is analyzed in this study. Our target primary studies dwell somewhere within the intersection of the mentioned areas, however, not all such papers fall within the scope of this study, and due to the reasons we shall see, finding the relevant studies is not always straightforward. In this section we present the necessary background, and define our target subject. The outcomes of this discussion will later be used upon protocol development (Section 3), query formulation, and definition of the study selection criteria.

2.1 What is program analysis?

Program analysis has been a topic of active research from the early days of modern computing [80, 83]. Applications of program analysis have been sought arguably in every aspect of software development, for every technological domain, in several (sometimes disconnected) research communities, and over a long period of time. Some achievements have been the result of opportunistic attempts to adopt some known technique in a new application areas, whereas others have been nurtured by long-term and planned research initiatives. These factors have resulted in a lack of standardization of terminology, and considerable vocabulary mismatch in the available literature.[18] The (probably healthy) tendency among authors for creative writing and exploiting the full range of the English language has perhaps amplified the problem, too. This issue might not stand much in the way of senior academics, but it does pose challenges for the junior audience, and more important in our context, the resulting phrase-sensitivity is an impediment for systematically searching through digital libraries that contain scientific literature.

We follow Binkley's definition of source code analysis as *"the process of extracting information about a program from its source code or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools. Source code is any static, textual, human readable, fully executable description of a computer program that can be the description can include documents needed to execute or compile the compiled automatically into an executable form"* [80].

In this SLR, we focus on cross-language program analysis in the context of supporting maintenance and evolution of *existing* systems. In this context, many tasks revolve around the process of "reverse engineering" information from source artifacts by means of *fact extraction* and inference of higher level abstractions and relations from lower level facts (*knowledge inference* or *knowledge discovery*). As such, we consider studies that exclusively focus on a "forward engineering" context (e.g. model-to-code transformation papers) to be outside our target scope.

Note that we are not only concerned by *static* program analysis, but also consider dynamic analysis (in an evolution context) [85] to be within the scope of our study (as shown by the inclusion of execution traces in the aforementioned definition). However, since we are interested in analyzing challenges raised by language heterogeneity, studies that do not discuss cross-linguistic aspects (e.g. analysis of language-agnostic execution traces) will not be included in this SLR.

2.2 What is cross-language?

Program analysis is generally perceived as a language-specific process, and unless explicitly stated otherwise, it implies application on a single programming language. To get better insight into the terminology that is used when describing program analysis studies that do *not* concern application on a single programming language, we followed the established guidelines for conducting systematic reviews [86, 87], and ran a pre-review to test the relevance of our research questions and the reliability of our query terms, This proved to be a valuable step to identify the phrases that are commonly used by researcher and practitioners in the domain. The pre-review revealed that target

[18] Note that the use of mismatching, or even contradictory, terminology has been observed between different standardization bodies, and in some cases among different task forces within a single standardization body [84].

Table 1: The main focus points for characterization identified in the pre-review.

System artifacts	Analysis method	Development method
multi-language	multi-language	polyglot programming
multilingual	multilingual	
mixed-language	cross-language	
heterogeneous	language-independent	
polylingual	language-agnostic	

papers typically[19] characterize themselves by either focusing on the *system's artifacts*, on the *analysis method* used, or on the *development method* used. We give an overview of the various characterizations in Table 1. One could argue that polyglot programing is an instance of artifact focus, but we decided to include it as separate category because those papers treat it as a development method, not as a system characteristic, and the resulting terminology would affect the choice of search terms in any follow-up studies.

In order to better scope the target of this study, we distinguish the following conceptual approaches to the analysis of multi-language software systems:

1. *Single-language analysis:* the use of two or more *intra*-language analyses to study each single-language portion of the system independently and individually, each using a separate analysis tool. Obviously, in this case portions of the overall structure and behavior of the system that are realized only through *inter*-language interactions are not covered in the analysis (Figure 1a). As an example of this type of analysis, consider language specific clone detection techniques that typically operate at the function level or below.

2. *Multi-language analysis:* the use of one analysis method (or tool) that, generally by means of a common representation, is capable of analyzing different languages, but still analyses the artifacts of each language independently (Figure 1b). For example, the consistent visualization of software control structure and complexity for all the modules in a multi-language software system that is provided by the GRASP tool [88].

   This approach is not fundamentally different from the first one, but rather a natural evolution of it. It primarily results from making the implementation of program analysis tools more cost-effective by enabling the reuse of tools and techniques already developed for other languages through the use of common intermediate representations. This research is closely connected to topics such as language-independent or language-agnostic code analysis and manipulation, intermediate source code representation [89, 90], standard exchange formats in-between program analysis tools [91], retargetable analysis tools [92], etc.

[19] Counter examples are covered in Section 3.3.

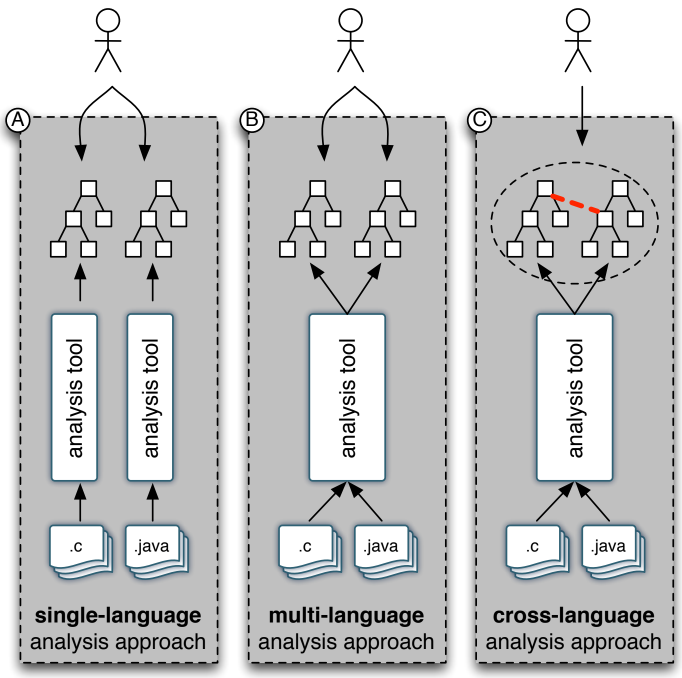

A
B
C
analysis tool
analysis tool
analysis tool
analysis tool
.c
.java
.c
.java
.c
.java
single-language analysis approach
multi-language analysis approach
cross-language analysis approach

Figure 1: Single-, multi-, and cross-language analysis approaches applied on multi-language systems. Only the cross-language analysis approach covers the interaction of multiple languages.

3. *Cross-language analysis:* the analysis of the whole system as a single entity; not only covering all software artifacts, but also incorporating the structural or behavioral relations that are realized through *inter*-language interactions *across* artifacts (Figure 1c). As an example of this analysis, consider the system-wide information flow tracking through a combination of C modules and a third party component composition framework, as discussed in [93], or the visualization of system-wide call graphs in a MLSS.

The scope of this study is cross-language analysis as described in the last item above. However, to avoid missing potentially relevant work, the focus is *not* exclusive: all papers that discuss at least *some* cross-language analysis aspects are considered relevant to our study, even if the analysis approach would not primarily be categorized as cross-language analysis.

2.3 Borderline studies

Heterogeneity is a challenge that applies to various areas of software engineering. Although *programming language* is only one source of *heterogeneity*,[20] it is already broad enough to intersect with several research topics.

While we are not aware of any study that explicitly opposes to the distinction

[20] Other examples include processor and data representation heterogeneity.

between cross-language and multi-language analysis discussed in the previous section, making this distinction is far from commonplace. Many authors do not use the term “cross-language” (or “cross-lingual”) analysis, but instead report on their approaches as a form of “analysis of multi-language systems.” Note that this does not settle the ambiguity whether their approach includes cross-lingual relations or not. In some cases, both authors of this SLR had to review a paper in great detail, and debated all the available information, before a consensus could be reached.

On the other hand, there is a large amount of studies that merely mentions cross-language program analysis in passing, and their treatment of the topic is limited to a few auxiliary clauses. For instance, in [94] the authors propose a methodology to *specify* cross-language interactions among DSLs (using a separate DLS), and as a prerequisite they do mention that cross-language interactions need to be identified before hand. However, as the focus of the paper is elsewhere, the discussion then proceeds in other directions. The authors of this SLR reviewed the paper and discussed its contributions to reach a consensus on inclusion (Section 3.4). We decided to exclude this paper from the study, and do so with similar papers in which cross-language program analysis is merely an auxiliary part of the paper. A good indicator for evaluating such papers was to how many of our research questions (see Section 3.2) the paper provided data points.

Although the number of cases were both evaluators disagreed in their verdict was relatively small, it made us aware of the intricate judgments that surround the design and discussion of multilingual and cross-lingual analysis, and the wide range of possible interpretations by different authors on the topic. For reproducibility, we discuss a number of research topics that we considered *not* relevant to this specific SLR, but which were found during the pre-review based on our query terms:

1. Data integration in heterogeneous data base systems by means of ontologies synthesized from run time data.

2. Studies with an *exclusive* focus on compiler technology with no direct connection to software evolution.

3. Enabling interoperability between (a heterogeneous collection of) *standalone* software systems via (black-box) bridging technologies, such as web-services, service orchestration, software agents, etc [95].

4. Studies that focus on Natural Language Processing (NLP) to analyze multiple natural languages (e.g., multi-language textual content of webpages [96]).

5. Studies that purely analyze models of software [97, 98], with no consideration of programming languages or model extraction/reconstruction from source artifacts.

6. Low-level embedded software, such as System-on-Chip engineering, FPGA programming, digital signal processing, hardware description languages like VHDL and Verilog, etc.

7. Studies that concern recording/playback and analysis of black-box behavior of subsystems (e.g. their network interactions, [99]).
8. Retargetable program analyzers that can be configured cost-effectively to handle other languages, but the paper does not discuss analysis of MLSSs as a unity [100]. Although these approaches are likely adaptable to cross-lingual situations, we do not include them unless this application is explicitly discussed in the paper.
9. Studies whose approach *could* be applied to cross-language analysis, but the paper does not demonstrate or refer to such applications (e.g. [101, 102]).
10. Purely language-agnostic, or language-transparent [103] techniques that are completely insensitive to the underlying programming language. We focus on techniques to go *across* the language boundaries, not in techniques that avoid them.
11. Studies that are aimed at integrating several (heterogeneous) program analysis techniques on a single programming language [104].
12. Studies that analyze if various dynamically generated outputs of a program conforms to a different grammar. For instance, [105, 106] use abstract-parsing techniques to check whether the generated output of a, say, JSP script qualifies as a well-formed HTML or SQL. The analysis of each language happens separately, and cross-lingual interactions are not included.

We remind the interested reader that some of these excluded topics can be of interest when investigating approaches for cross-language analysis. Nevertheless, we consider such papers outside the scope and purpose of this SLR, because they do not explicitly contribute to literature on cross-language analysis.

Overall, we take a conservative approach while judging the relevance of such borderline studies, and consider only those papers that: (1) are relevant to program analysis *and* software evolution, (2) analyze information derived from artifacts written in at least two different (computer) languages, of which at least one programming or scripting language, and (3) contain a non-trivial discussion of the relevant cross-lingual analysis aspects.

3 Review Protocol

This SLR was carried out as a series of discrete steps, closely following Kitchenham's guidelines for conducting systematic literature reviews in software engineering [81], and adapted to the characteristics of our study topic. This section presents the *review protocol*, which details the procedure that was followed during the course of the study. The protocol entails: the research questions, the search strategy, the study selection (i.e. inclusion and exclusion) criteria, data extraction procedures, and data synthesis methods.

Table 2: Most frequent bigrams (with window size 2, 3 or 4) in the titles, abstracts and keywords of documents in our test collection.

Bigram	Freq.	Bigram	Freq.	Bigram	Freq.
reverse engineering	45	abstract software	12	web applications	7
source code	33	software engineering	10	source models	7
software systems	23	software development	10	prototype tool	7
multi language	19	reverse tools	10	programming language	7
case study	16	program analysis	10	multi software	7
programming languages	15	paper presents	10	language systems	7
analysis tools	13	multiple languages	10	language dependencies	7
meta model	12	language software	9	island grammars	7
language independent	12	cross language	9	intermediate representation	7
engineering tools	12	conceptual model	8	dependency analysis	7
component based	12	abstract paper	8		

3.1 Pilot Study

When defining a review protocol, it is strongly suggested to conduct a small pilot study to check the effectiveness of the (draft) protocol and possibly make corrections, before one sets out to conduct the full study.

For the pilot study, we created an initial collection of about 40 papers, all of which were considered highly relevant to the topic of our study, that is intended as *test set* for our developing our protocol and queries. This collection was compiled from (1) hand-picked papers, based on our knowledge of the domain, (2) the related work mentioned in milestone cross-lingual analysis papers, and (3) querying the most obvious phrases of “multi-language” and “cross-language” against Google Scholar, and the IEEE and ACM digital libraries.

The papers in the test set were used in three different ways: First, the test set was used to identify phrases that are commonly used in the domain, to support query formulation. To this end, the titles, keywords, and abstracts are extracted from the individual papers to create a corpus, which, after filtering out stop-words (e.g. a, of, who, been, etc.), is subjected to a standard n-gram phrase analysis.[21] To cancel out the effects of having different word order or interspersed words (e.g. “analyzing multi-language systems,” and “analysis of multi-language software systems”), different bigrams are extracted using window sizes of resp. 2, 3, and 4. Although most of the generated bigrams were too general to be effective search terms (such as “source code” and “case study”), there were other bigrams, such as “web applications” and “component-based”, that hinted at ways to strengthen our query terms. An overview of the most frequent bigrams is shown in Table 2).

[21] We use count.pl from T. Pedersen’s Ngram Statistics Package (NSP), available from http://search.cpan.org/dist/Text-NSP/

Second, the test set served as a minimum acceptance test while experimenting with different query formulation strategies on the various digital libraries [107]. The query formulation and search strategy will be discussed in more detail below, in Section 3.3.

Third, full text of the test case papers was carefully analyzed to help define/refine the data extraction form. Having the test set at our disposal, we could identify what aspects of cross-language analysis methods were stressed in most papers, and could complement the data extraction protocol accordingly. The data extraction protocol will be discussed in more detail below, in Section 3.7.

As the scoping of the study became more clear, about 20 papers from this initial test set made their way into the final results. Although the rest of the papers did not pass our criteria to be assessed as cross-lingual and were later excluded from the study, these papers still helped to fine-tune our scope and terminology, and increased our insight into the topic. In fact, most of these papers discuss borderline research topics, such as the language-independent and retargetable program analyzers, that were discussed previously in Sections 2.2 and 2.3.

3.2 Research Questions

The starting point of this SLR can be summarized as: "What cross-language program analysis methods have been studied so far?" This question, by itself, might closer relate to a systematic *mapping study* than a systematic literature review. Although the border between the two types of studies is not rigid, mapping studies are more aimed at providing a overview of a research area, classifying and quantifying the available literature, the common publication forums, and often contains an analysis of publication trends over time. In contrast, systematic reviews aim at a deeper analysis of the primary studies, to establish the state of the evidence [81, 108]. In this perspective, the first phase of our study can be categorized as a mapping study, aimed to maximize the coverage of the field, and the second phase adds the deeper analysis of a systematic literature review. Indeed, some authors advise a mapping study as a prerequisite to a deeper, complementary systematic review [108].

Note that our analysis does not stop at a coarse-grained statistical analysis of the study topics, nor do we limit ourselves to the titles and abstracts of the primary studies. Our study analyses the full text of the papers and seeks answers to specific research questions, prior to aggregating a qualitative review of the primary studies.

The list of research questions that structure our review is shown in Table 3. Some of the questions are strictly within the boundaries of a mapping study, whereas others need a full analysis of the primary studies.

Table 3: Research questions.

RQ1	What approaches have been used for cross-language program analysis?
RQ2	What fact extraction methods are commonly used during cross-language program?
RQ3	What types of facts are typically extracted for cross-language program analysis?
RQ4	What internal representations of software artifacts are used?
RQ5	What higher level goals are targeted using cross-language program analysis?
RQ6	Which languages and types of software artifact have been analyzed?
RQ7	Which technological domains attracted most attention in literature?
RQ8	How rigorously are newly proposed approaches tested and evaluated?

3.3 Data Sources and Search Strategy

With respect to investigating the available literature, the general scope of this study can be described as:

- *Population:* Scientific literature reflecting on cross-lingual program analysis.
- *Intervention:* Devising new and/or applying cross-language analysis methods.
- *Outcomes:* Extent of the studied cross-lingual relations, and the languages involved.
- *Experimental designs:* No restrictions. All primary studies that concern a relevant intervention are accepted, on the condition that they demonstrate their relevance with enough objective data. Our quality assessment criterion (see [81]) is whether the paper provides an answer for one or more of our research questions.

Our search strategy consists of two consecutive stages: (1) searching digital libraries, and (2) snowballing.
Digital Library Search – We consult the following seven digital libraries to search for primary studies:

IEEE Xplore	`http://ieeexplore.ieee.org/`
ACM Digital library	`http://dl.acm.org/`
Wiley Online Library	`http://onlinelibrary.wiley.com/`
IET Inspec	`http://inspecdirect.theiet.org/`
ISI Web of Science	`http://webofknowledge.com/`
ScienceDirect	`http://sciencedirect.com/`
Scopus	`http://scopus.com/`

We do not constrain the search by date, i.e. all entries in these libraries at the time of the search (March 2014) are potentially of interest. We do limit the search to peer reviewed papers and books in those repositories were it was possible to specify document types.

Table 4: Sub-expressions used to build the queries.

E1:	("multi-language" OR "multilingual" OR "cross-language" OR "heterogeneous" OR "heterogeneity" OR "language independent" OR "language agnostic" OR "multiple languages" OR "polyglot" OR "polylingual")
E2:	("program analysis" OR "source code analysis" OR "program analyzer" OR "source code analyzer" OR "impact analysis" OR "dependency analysis" OR "analyzing dependency" OR "analyzing dependencies" OR "traceability analysis" OR "fact extraction" OR "exchange format" OR "intermediate representation" OR "reverse engineer" OR "reverse engineering" OR "reengineer" OR "reengineering" OR "program comprehension" OR "comprehending" OR "comprehend" OR "system comprehension" OR "program understanding" OR "software understanding" OR "understanding software" OR "understanding program" OR "modernize" OR "modernization" OR "modernizing" OR "modernise" OR "modernisation" OR "software maintenance" OR "software evolution" OR "maintainability" OR "evolvability" OR "comprehensibility" OR "software metrics")
E3:	(software OR system OR program OR "source code" OR "application" OR "applications" OR "software development" OR "software engineering" OR "component-based" OR "web application" OR "web applications" OR "web-based")

Since this is not the case for all libraries, the results still include a number of non-peer reviewed documents, which are removed in the study selection process (Section 3.4).

The overall query string is constructed by a conjunction of three sub-expressions: E1 AND E2 AND E3, corresponding to the *population*, *intervention*, and the *general context* respectively (see Table 4). To ensure that the digital library search was as exhaustive as possible, we built on the experiences from the pilot study (Section 3.1) and extended the query sub-expressions with a series of alternative search terms, while keeping the number of false positives under control using the conjunction.

The selection of phrases in the query sub-expressions is the result of several test runs during the pilot study. E1 and E3 form a straightforward characterization of the population based on results of the pilot study. E3 is particularly useful for decreasing the number of false positives from general-purpose digital libraries that are not limited to computer science, such as Scopus. Moreover, E3 is beneficial for queries on computer science digital libraries, such as the ACM and IEEE Xplore digital libraries, as it helps to restrict generic query terms, such as "analysis" and "analyse" in E2, to the more restrictive n-grams that were found in the pilot study.

Wherever possible, we search within the paper meta-data (i.e., the title, abstract and keywords), because searching against the full text of the papers in a library makes the precision of the queries very low (i.e., the queries return many false positives) [107].

Note that the query shown in Table 4 is generalized. In practice, each of the mentioned digital libraries employs their own specific search technology, leaving the end-user with a non-uniform interface and set of options. Differences that should be carefully taken into account upon query articulation include: the availability of *word expansion* services,[22] the possibility of limiting (sub)queries to titles, abstracts, keywords, or full text, the ability, and the syntax, to form nested queries, To support replication of our study, we report on our specific queries for the various digital libraries in Appendix A.

The limitations of the aforementioned query facilities, their brittle syntactical sensitivity, and our lessons learned w.r.t. query formulation are not within the scope of this paper. However, we do want to point out that the articulation of the query, and its *normalization and unification* across the different digital libraries, took considerably more time and effort than what we originally expected. We want to highlight the instrumental role of (1) the pilot study, (2) the initial test set of known relevant papers, and (3) documenting the evolution of the query strings with lessons learned for each search engine, as a means to effective query articulation.

Snowballing – There are three main approaches to search for primary studies for the purposes of a systematic review: automated searching in digital libraries, manually investigating the records of known publication venues, and (backwards) *snowballing*: searching for studies in the references of initial *seeds* or in the references of studies that were found before. There is no best technique among these when it comes to completeness, precision, or the required manual effort [109, 110]. Rather, it is advised to not exclusively depend on a single search technique, but to complement the main method with the help of a second, or even a third. In our SLR, we use snowballing to complement the automated search described in the previous section. This step is completely manual and starts from the set of studies identified in the digital library search. The criteria to include a paper during snowballing are the same criteria as are used to check for the relevance during the digital library search.

3.4 Study Selection and Management

In this section we present the process and the criteria used to select the relevant primary studies that will be included in the final analysis of the review. Figure 2 depicts the main steps of the study selection process. Our selection approach was based "exclusion," rather than inclusion: i.e. at each step only those studies that were clearly not fit for our review were filtered out, and the rest were left for more thorough selection steps.

Bibliography Management – A drawback of using automated searches over a number of digital libraries is that they require considerable manual effort to unify the results of the individual libraries. In our study, we have opted to use the (free) Mendeley Reference Manager[23] as a means for bibliography management, from capturing query results for individual digital libraries, to identifying duplicate results (e.g., the same study was found in different libraries), and all successive selection steps. The use of Mendeley

[22] For example, stemming a user's query containing the term "analyzing", and matching related terms such as analyze, analysis, analytical, etc.

[23] `http://www.mendeley.com`

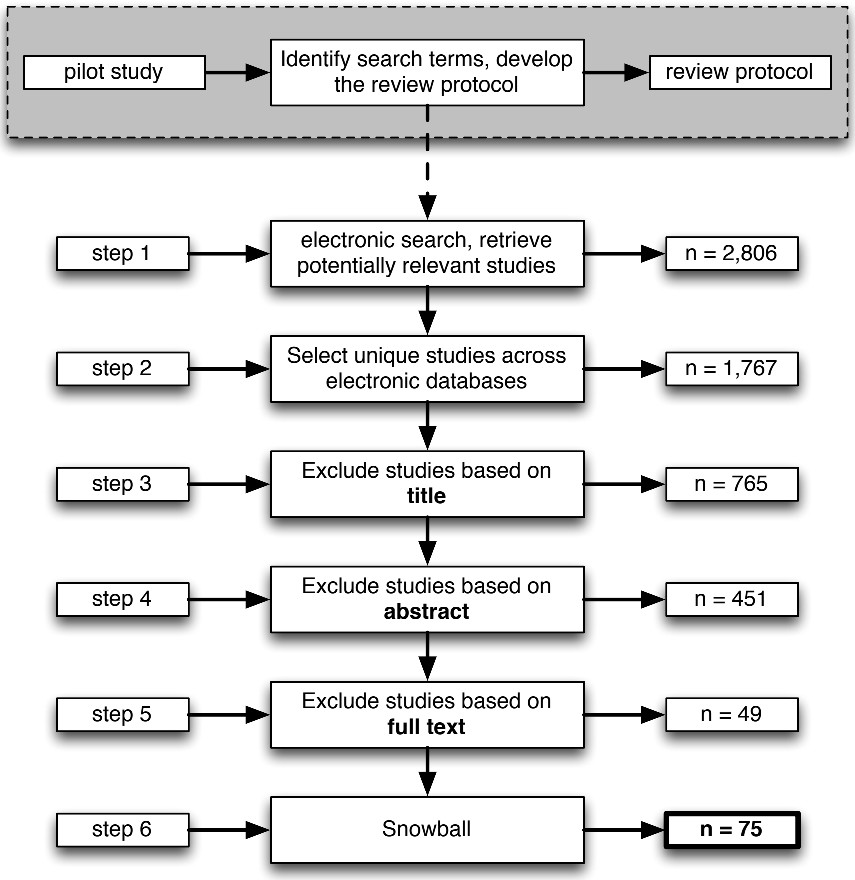

pilot study
Identify search terms, develop the review protocol
review protocol
step 1
electronic search, retrieve potentially relevant studies
n = 2,806
step 2
Select unique studies across electronic databases
n = 1,767
step 3
Exclude studies based on **title**
n = 765
step 4
Exclude studies based on **abstract**
n = 451
step 5
Exclude studies based on **full text**
n = 49
step 6
Snowball
n = 75

Figure 2: Study selection process.

has the advantage that we can easily annotate references and organize them using tags. These annotations can be shared between the authors of the SLR, and any updates are synchronized to all parties. Moreover, in the later stages of the selection process, full-text PDFs can be attached, annotated and synchronized.

Step 1. Automated Search – Results from the automated searches using the query described in Table 4 were gathered in BibTeX format, in a separate file for each individual digital library. The search resulted in a grand total of 2806 hits, i.e., potentially relevant papers, for all seven libraries (step 1 in Figure 2). An overview of the results for the individual libraries (and various selection steps) is presented in Table 5.

Step 2. Removing duplicate results – We observed that the results from different search engines had considerable overlap, leading to hundreds of duplicate entries. These were identified by importing and tagging the results in Mendeley on a library by library basis. To this end, the BibTex files created in step 1 were imported into separate groups (repositories) in Mendeley, and tagged according to their origins (e.g., "search_results_acm").

While importing entries, Mendeley matches them with existing entries, and whenever a reference already exists, the newly added entry will be tagged with the same

Table 5: Search result in digital libraries: initial hits, unique entries, number of papers remaining after selection on title, on abstract and on full text.

	Digital Library	Hits	Unique	Title	Abstract	Full
1	IEEE	535	535	370	240	36
2	ACM	282	173	112	53	9
3	Wiley	213	194	22	7	0
4	Inspec	426	243	112	60	0
5	Web of Science	359	149	48	31	1
6	Science Direct	78	48	8	7	1
7	Scopus	913	425	93	53	1
	Total	2806	1767	765	451	49

tags as were set on the existing one. As a result, duplicate references could be easily identified, because they contained multiple origin tags, leaving us with 1767 unique entries (step 2, Figure 2).[24]

We processed the digital libraries in the order indicated by the first column of Table 5. Duplicate entries were removed from the non-original libraries: for a paper published in IEEE, but present in both IEEE and ACM search results, the entry in the ACM results was deleted. Table 5 shows the number of unique entries from each of the search engines.

Step 3. Selection on Title – The third step in the study selection process is based on the titles of the papers. Entries that were out of the scope of computer science and software engineering were easy targets for exclusion. Entries that had *clearly* nothing to do with program analysis were also filtered out. In addition, non-peer reviews studies such as poster sessions, announcements for panel discussions, keynote presentations, workshops summaries, interviews, and the like, were removed. PhD theses and their summaries were also removed from the study, but papers that were at the basis of the theses were manually added to the results if they seemed relevant to the study (e.g. [111, 112]). One paper that was flagged as plagiarism in IEEE Xplore was also removed. The excluded papers were tagged as excluded in Mendeley, rather than deleted from the repository to make bookkeeping and double-checking easier. All other papers were given the benefit of the doubt and forwarded to the next step of study selection.

Step 4. Selection on Abstract – The next source of information to judge the relevance of the studies are the abstracts. As already mentioned, the terminology relevant to cross-language program analysis is not standardized, and authors use various phrases and writing styles to describe their work. Also, not all abstracts clearly state the goal, the proposed technique, and the conclusion of the study.

In fact, we observed that a considerable portion of the papers in our result set did

[24] Note that some duplication still remained due to small differences in the bibliographic data kept by different digital libraries which prevented Mendeley from matching earlier imported entries. These duplicates were manually removed as part of the later stages.

not have clear indicators in their titles and abstracts. As such, we took a conservative approach and only filtered away those papers that were clearly irrelevant to our research topic. The exclusion was implemented, again, by means of tags on our Mendeley repository.
Step 5. Selection on Full Text – Finally, we conducted a full review of the remaining papers to decide on their inclusion. Due to our conservative approach in excluding studies in the previous steps, we were left with a considerable number of papers subject to full text review, and this step took significant manual effort for completion.

The criterion to include a study was demonstration of relevance to the topic of cross-language analysis (see Section 2). As a first step, the information present in the abstract, introduction and conclusion sections of the papers were considered together to determine inclusion or exclusion. The majority of papers that remained in our repository contained enough details in these sections to enable a decision. However, some papers required a full review, and even then there were a few difficult cases where there was not enough information for a single reviewer to confidently take a decision. Some examples include:

- papers that include cross-language analysis as a goal in the introduction, but do not discuss the topic explicitly in the remainder of the paper;
- papers that characterize themselves as doing multi-language analysis, and do not contain enough details on their approach to infer specific cross-language aspects;
- papers that contain cross-language analysis aspects, but do not present the work accordingly, or present it with a sufficiently distant terminology to be unrecognizable;
- papers that present a broad study, of which only a small portion has "something to do" with cross-language analysis.

All "borderline" cases, or cases in which either of the two authors had doubts about, were double checked separately by both authors, and discussed until consensus was reached. The conclusions and lessons learned from such discussions were captured, and used to judge the relevance of similar studies in a uniform manner. The bottom-line criterion was the application of cross-language program analysis; basically checking if the proposed solution considered information derived from two or more languages to conduct an analysis.
Step 6. Snowballing – After selection on full text, we are left with 49 primary studies. To ensure proper coverage of our study, we applied backward snowballing and checked the references (of references) of selected studies to find additional relevant studies. Borderline papers, that were excluded from papers selected for the study, were also subjected to snowballing to find potentially relevant material.

In addition to reference based snowballing, the *publication records* of authors of selected papers were manually investigated to search for relevant mat Finally, we used Google to check additional phrase combinations that were derived from the meta-data of papers in our selection, and potentially not covered by our initial query. The snowballing phase made a significant contribution to the final results with 26 additions (34% of the total set).

3.5 Reliability of Selection

This systematic literature review is the result of the collaborations of the two authors of this paper. The first author was the main responsible for the execution of the various selection steps. However, to ensure the reliability of inclusion decisions, the second author made independent inclusion/exclusion decisions on a subset of the studies, and the results were compared and discussed using Cohen's Kappa inter-rater agreement statistic [81, 113].

Before the execution of each of the selection steps 3, 4, and 5, at least 10% of the inputs to that step were randomly selected as the screening set: 180 papers for selection by title, 77 papers for abstract, and 45 papers for full text analysis. Next, both authors independently judged on inclusion/exclusion of the papers in the respective screening set. Individual results were compared, and conflicting verdicts were analyzed and discussed until a consensus was reached. In addition to settling the decision on that particular paper, these also served as lessons-learned for later decisions in the selection step, which is why the screenings were conducted as the first phase of these steps.

In addition to the studies that were double-checked in the screening sets, all studies for which the first author was not completely certain during the selection process, were tagged and then later double-checked by the second author. As with the screening sets, the final selection verdict for these studies was reached by mutual consensus.

Overall inter-rater agreement levels during the screenings were high, and there was only one study for which we could not settle an agreement without investigating the full text version. We argue that the combination of witnessing high inter-rater agreement levels for the different screenings, and leaving no "suspicious" inclusion decisions to the judgment of only one author, helps to achieve a high confidence in the reliability of the selection process.

3.6 Study Quality Assessment

In addition to the selection based on general inclusion/exclusion criteria, the evaluation of additional quality appraisal criteria for primary studies may lead to a refinement of the selection. For example, in order to enable systematic assessment of the evidence that is available with respect to a certain intervention, systematic literature reviews often require the use of rigorous empirical evaluation as a mandatory quality criterion for inclusion of a primary study [108]. On the other hand, it is known that the initial literature on new research topics often lacks methodological rigor in its evaluation. For example, Mendes showed that only 5% of the studies in the domain of web engineering follow the methodological rigor required for empirical evaluation, and lightweight evaluation methods such as “experience report” and “proof of concept” are frequently mistaken for more elaborate methods like “experiments” and “case studies” [114].

As mentioned, the first part of our study is essentially a systematic mapping study, aimed at covering the full range of attention that cross-language program analysis has received in the research community. During our pilot (and later also during the full study), we observed a situation similar to that of Mendes, were the lack of methodological rigor in potentially relevant studies implied that requiring the use of rigorous empirical evaluation would overly restrict the coverage of our study. As such, we decided to not introduce additional quality appraisal criteria for primary studies, other than addressing our focus area.

3.7 Data Extraction Strategy

After selection of the relevant primary studies, they need to be systematically analyzed to collect the information that is needed to answer our research questions (Table 3). To guide this process, we developed a data extraction table (included as Appendix B), to record the data extracted from the primary studies. The columns in the table are derived from the research questions to be answered. Most research questions were mapped to a single column in a straightforward way, while some were mapped to multiple columns, depending on the amount of aspects relevant to answering the respective question.

To avoid ad hoc interpretation during data synthesis, and to minimize the effects of terminology mismatch between reviewers, we chose to try and limit the options for answering to Y/N check-boxes, or selection from a limited set of options (using drop-down lists) over free text answers whenever possible.[25]

After piloting the extraction process on a representative random sample of 20% of the primary studies to identify recurring concepts, the taxonomy of possible answers was created by open and axial coding techniques from grounded theory [115], When assessing the initial extraction results from this pilot, we realized that answers to the same questions had a wide variation for a number of reasons, such as:

- the wide range of analysis goals, application domains and languages that were analyzed;
- range of terminology used by the authors of the primary studies, come from different research communities (e.g. model-driven engineering vs. language design and compiler technology) and have varied experience levels;

[25] The data extraction table was implemented in Microsoft Excel and a small amount of Visual Basic scripting for the drop-down lists.

- differences between claims (or outsets) and the actual contributions

We concluded that the most reliable approach was to iteratively synthesize our taxonomy using open (unrestricted) coding, followed by grouping these into a limited amount of categories using axial coding, based on mutual consensus between the reviewers on the relations between the open codes [115].[26]

For some of the table columns, the number of alternatives in the taxonomy stabilized early in the pilot. For other columns, however, the more studies we reviewed, the more options we had to add to the respective group to ensure keeping precision in the characterization. A typical example was the group reflecting the type of data extracted by a study to enable cross-lingual analysis. It quickly became clear that the pilot would not show us the whole range of options, so to accommodate as many answers as needed, we promoted this column to open-ended answers. However, even in such cases we opted to maintain a growing list of answers over pure free text answers to promote reuse of answers. We largely remained loyal to the original terminology in the study, only allowing for minimal interpretation for in cases where a similar item was already present in the list.

Overall, this proved to be an effective strategy for systematic data extraction: with the taxonomy in place there were only minor disagreements in characterizing the primary studies.

3.8 Data Synthesis

The next stage, data synthesis, is aimed at creating an overview of all facts gathered in the data extraction table. The goal is to allow an unambiguous insight into the matter, yet abstract enough to be presentable to the reader in an accessible way.

As a result of our systematic data extraction strategy, a number of columns were already limited to a manageable number of distinct facts. Most of these columns could readily be presented without needing further processing or generalization. The data in other columns, especially the aforementioned open-ended answers, needed additional processing to make the information presentable in an accessible way.

To this end, we employed another step of axial coding, grouping related concepts into more general abstractions [115]. This synthesis was iteratively conducted along both axes of the table: across co-related primary studies, and across the concepts in the taxonomy. For example, during data extraction we distinguished between *intra-*, and *inter*-procedural control flow graphs to maximize precision, and to be able to identify potential trends in usage of either of these concepts. However, during the data synthesis phase, we realized that only two studies ([30] and [7]) distinguish between these types

[26] Note that this process provides a less rigid alternative to the meta-ethnographic analysis [116, 117], that is suggested in [81]. We chose not to conduct a full meta-ethnographic analysis due to the large number of primary studies and wide diversity of subject areas in our selection, as translating all studies into a single common format would hide too much interesting details.

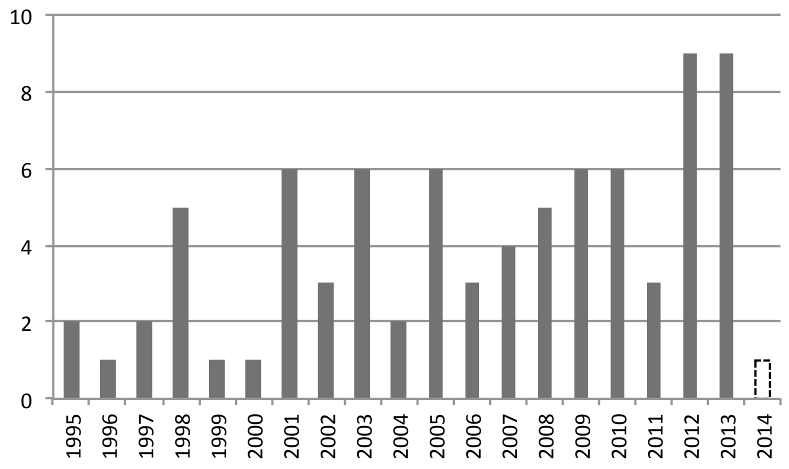


Figure 3: Distribution of studies over time. 2014 is not included as a full year.

of control flow, and it we can abstract to the more general "Control Flow", without loosing valuable insights.

The use of study-specific terminology during data extraction, and generalizing to cross-study abstractions during data synthesis, when the informative value of keeping individual concepts became more clear, allowed us to make informed decisions about balancing the line between precision and conciseness.

4 Findings

This section discusses the findings of our SLR based on analysis of the selected primary studies, which are presented in the first 75 references in the bibliographic section (1 to 75). First, we will analyze the publication data of the studies, without reflecting on their content. Next, we will consolidate the content of the selected studies by developing a number of classifications with respect to various aspects of the studies. We will make use of tabular presentations to provide a succinct united perspective on the available body of knowledge. Finally, this section addresses the individual research questions in more detail, staying close to the extracted (verbatim) data, and referring to concrete technologies wherever suitable.

4.1 Analysis of publications

In this part section our goal is to position the available literature across the interested research communities. As such, we include all studies, including those where the same research was presented in more than one publication (in later analyses these will be unified to avoid bias in quantitative assessments).

Time – The earliest primary study in our review dates back to 1995, and the latest is from March 2014. Figure 3 shows the publication frequency between these times. As can be seen in the figure, there were no years in this period without relevant studies being published. Moreover, although the number of published studies fluctuates over time, the distribution hints at a general increase in attention for the topic in the recent years.

Publication channels – Table 6 gives an overview of the type of the publications, and publication channels. The majority of primary studies were published in conferences and workshops (71 of 75, i.e., 95%), while only 4 (5%) of the studies were published in scientific journals.

The 75 primary studies were published in 37 publication channels in total, and 25 of these channels only published a single study. From Table 6, it is obvious that cross-lingual program analysis has enjoyed considerable attention in conferences in the area of software maintenance and evolution. The top five publication channels are dedicated to this area and contain 48% of the relevant primary studies, the premier channel being the SANER conference, which is the merger of the former WCRE and CSMR conferences on respectively software reverse engineering, and software maintenance and re-engineering. Another noteworthy fact is that the call for papers of one of these (WSE - Symposium on Web System Evolution) explicitly mentions "analysis of multilingual systems" as a topic of interest.

Source of results – With respect to our search strategy, Table 7 presents the number of studies selected from each of the seven digital libraries and in the successive snowballing stage. Note that the numbers in this table are affected by the order in which we processed the different digital libraries (which is specified in Table 5), and its consequences for duplicate removal (i.e., we only count the first occurrence of a paper in cases were different digital libraries contain duplicates, as discussed in Section 3.4). Nevertheless, the table hints at the fact that approximately 95% of the relevant primary studies could have been retrieved using an electronic search on the IEEE and ACM libraries, together with snowballing on those results. Although we chose to use several digital libraries (for the benefit of completeness), this finding might encourage other researchers to limit their search to the mentioned libraries, should they follow different criteria for cost-effectiveness.

4.2 Analysis of studies

To avoid publication bias, 10 studies are removed from further synthesis as their content is largely repeated in other papers, for example in the form of journal extensions of conference papers, or full conference papers that build on initial short papers. In such cases, only the most complete paper is analyzed and accounted. The remainder of this SLR reflects on the results of the 65 *unique* primary studies.

There is also a number of studies that have a certain degree of overlap, yet the contents are considerably different that neither study can be removed from our analysis

Table 6: Distribution of publication channels and types.

Publication Channel	Type	Number	Percent	
SANER (WCRE+CSMR)*	Conference	13	17.3	
ICPC/IWPC	Conference	7	9.3	
ICSM	Conference	7	9.3	
WSE	Conference	5	6.7	
SCAM	Conference	4	5.3	– 48.0%
ICSE	Conference	3	4.0	
WRT (Refactoring)	Workshop	3	4.0	
ECMFA	Conference	2	3.0	
ESEM	Conference	2	2.7	
FSE	Conference	2	2.7	
PLDI	Conference	2	2.7	– 66.7%
ASE	Conference	1	1.3	
ASF+SDF	Conference	1	1.3	
C3S2E	Conference	1	1.3	
CASCON	Conference	1	1.3	
ECOOP	Conference	1	1.3	
ESOP	Conference	1	1.3	
IASTED	Conference	1	1.3	
ICECCS	Conference	1	1.3	
ICPP	Conference	1	1.3	
ICSOFT	Conference	1	1.3	
ISADS	Conference	1	1.3	
ISISE	Conference	1	1.3	
MODELS	Conference	1	1.3	
PASTE	Conference	1	1.3	
QUATIC	Conference	1	1.3	
SEKE	Conference	1	1.3	
SYNASC	Conference	1	1.3	
TACAS	Conference	1	1.3	
WEC	Conference	1	1.3	
MiSE	Workshop	1	1.3	
TOPI	Workshop	1	1.3	
Elsevier SCP	Journal	1	1.3	
Springer ANSOFT	Journal	1	1.3	
Springer ISSE	Journal	1	1.3	
Wiley JSME	Journal	1	1.3	
Total		75	100%	

* In 2014 WCRE and CSMR merged into the SANER conference; of the 13 papers, 1 was from a joint event, 9 were from WCRE and 3 were from CSMR prior to the merge.

Table 7: Sources for the selected studies after the snowballing stage.

Source	Number	Percent
IEEE	36	48
Snowballing (manual)	26	35
ACM	9	12
Web of Science	2	3
Scopus	1	1
Science Direct	1	1
Inspec	0	0
Wiley	0	0
Total	75	100

without loosing value. For instance, in [2], Alalfi et al. envision a conceptual solution to reverse engineer access control models of PHP applications using both static and dynamic analysis, while in [3], they present and evaluate the dynamic analysis approach to analyze the database interactions of PHP applications in more detail. We identify such cases in the presentation of the results, and account for these overlaps in our quantitative analyses, so that they do not bias our results when it comes to answering the research questions.

A number of concepts in the data extraction table pertain to cross-cutting features that reflect on all primary studies. Such cross-cuttings features will unified in the presentation, with no further categorisation, to provide a succinct answer to the respective research question. On the other hand, a number of research questions only apply to certain types of primary studies, and some research questions have a wide variety in their possible answers, which poses challenges to using a systematic flat presentation across all studies at the same time. To avoid having an unwieldy large and sparse presentation, we group studies into a set of coherent sets, and present an overview for each of these at an appropriate level of abstraction. This allows us to add more specific details for those groups of studies without loosing overall accuracy.

At the top-most level, we distinguish the following broad categories of studies:

1. technology papers (58 studies, 89%)

2. position papers (7 studies, 11%)

Where *technology papers* propose a new, or evaluate an existing, analysis method, and *position papers* contain general views that are independent from any specific method. In addition to answering our research questions, the studies within each category will be presented and discussed in more detail. To that end, technology papers will be classified further into coherent subgroups based on the goals of the studies and the applied techniques.

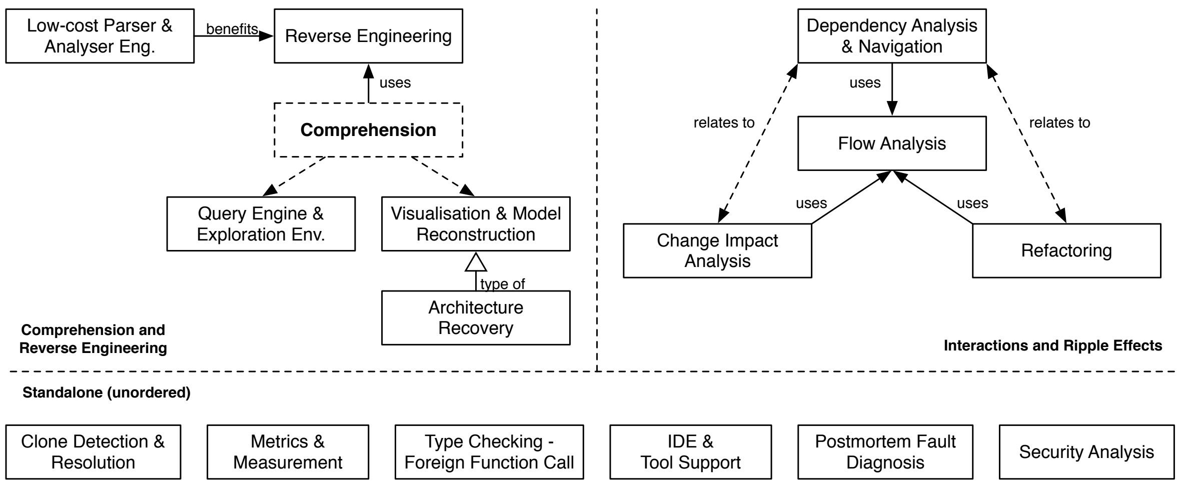

Low-cost Parser & Analyser Eng.
benefits
Reverse Engineering
uses
Comprehension
Query Engine & Exploration Env.
Visualisation & Model Reconstruction
type of
Architecture Recovery
Comprehension and Reverse Engineering
Dependency Analysis & Navigation
uses
relates to
Flow Analysis
relates to
uses
uses
Change Impact Analysis
Refactoring
Interactions and Ripple Effects
Standalone (unordered)
Clone Detection & Resolution
Metrics & Measurement
Type Checking - Foreign Function Call
IDE & Tool Support
Postmortem Fault Diagnosis
Security Analysis

Figure 4: Overview of the study goals that were identified, and their relations.

4.2.1 Study Goals

Within the 58 technology papers, we identified 16 distinct goals that were aimed at by the authors. A number of these goals are frequently targeted by several authors, while others have received significantly less attention. For clarity, we decided to keep all the identified goals (and not categorize the less frequent ones into "general" or "other" categories) despite the sparseness of some of the resulting groups. Likewise, we distinguished important subcategories of the more generic maintenance goals. This resulted in a number of conceptually correlated goals, as well as a number of standalone goals. An overview of the 16 goals and their relations is presented in Figure 4. Most identified study goals have self-explanatory titles. Here follows a brief description of the identified goals.

Reverse Engineering is the practice of analyzing a system to gain desired information, and generally benefits further "comprehension" attempts. Comprehension can either be addressed using graphical *Visualizations and Model Reconstruction*, or by providing the user with non-graphical *Query Mechanisms* to explore the system interactively. *Architecture Recovery* is a specialized type of model reconstruction and visualization, whose output is aimed at the abstract architectural level. A number of studies put a special price on *Cost-effective Parser and Analyzer Engineering*, which in turn facilitates day-to-day reverse engineering needs. We recorded considerable commonalities between the topics of *Dependency Analysis and Navigation*, *Change Impact Analysis (CIA)*, and to a lesser extent with the topic of *Refactoring*. Some form of *Flow Analysis* of data and/or control across multiple languages is typically used do drive these analyses, but was also observed as a primary goal in the selected studies. A number of studies have paid exclusive attention to *Foreign Function Calls*, mainly by providing cross-language *Type Checking* facilities. Although most authors have acknowledged the importance of tool support, only few have pursued the implementation of their

Table 8: Goals of the 58 technology papers, in alphabetical order on first author.*

	related	short paper	reverse eng.	query	visualization	arch. recovery	parser eng.	dependency	CIA	refactoring	flow analysis	clones	type check	metrics	IDE	fault detect	security	testing
Al-Omari+, 2012, [1]		·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·
Alalfi+, 2009, [2]	a	✓	⊙	·	·	·	·	·	·	·	·	·	·	·	·	·	✓	·
Alalfi+, 2009, [3]	a	·	✓	·	·	·	·	·	·	·	⊙	·	·	·	·	·	·	·
Amalfitano+, 2013, [6]		·	·	·	✓	·	·	·	·	·	⊙	·	·	⊙	⊙	·	·	⊙
Ayers+, 2008, [7]		·	·	·	·	·	·	·	·	·	⊙	·	·	·	·	✓	·	·
Barrett, 1996, [8]		·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
Bellettini+, 2004, [9]	b	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
Bellettini+, 2005, [10]	b	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
Boldyreff+, 2002, [11]		·	⊙	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·
Chase+, 1998, [13]		·	⊙	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·
Chen+, 2005, [15]		✓	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·
Cossette+, 2007, [16]		·	·	·	·	·	⊙	✓	·	·	·	·	·	·	·	·	·	·
Cossette+, 2010, [17]		·	·	·	·	·	⊙	✓	·	·	·	·	·	·	·	·	·	·
Deruelle+, 2001, [18]	c	·	·	·	⊙	·	·	·	✓	·	·	·	·	·	·	·	·	·
Deruelle+, 2001, [19]	c	·	·	·	⊙	·	·	·	✓	·	·	·	·	✓	·	·	·	·
DiLucca+, 2004, [22]		·	·	·	✓	·	·	⊙	·	·	⊙	·	·	·	·	·	·	·
Druk+, 2013, [23]		✓	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·
Furr+, 2005, [24]	d	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
Furr+, 2006, [25]	d	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·
Hassan+, 2003, [28]		·	·	·	·	✓	·	·	⊙	·	·	·	·	·	·	·	·	·
Hayes+, 2000, [29]		·	⊙	·	⊙	·	✓	·	·	·	·	·	·	·	·	·	·	·
Hessellund+, 2008, [30]		·	·	·	·	·	·	⊙	·	·	✓	·	·	·	·	·	·	·
Hsu+, 1999, [31]		·	·	·	·	·	·	⊙	✓	·	·	·	·	·	·	·	·	·
Kamp, 1998, [32]	e	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Kempf+, 2008, [33]		✓	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·
Kolsch, 1998, [35]		·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Kraft+, 2008, [36]		✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·
Kullbach+, 1998, [37]	e	·	·	✓	⊙	·	·	·	·	·	·	·	·	·	·	·	·	·
Lange+, 2001, [38]	e	·	·	✓	⊙	·	·	·	·	·	·	·	·	·	·	·	·	·
Lehnert+, 2013, [39]		·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
Linos, 1995, [40]		✓	·	·	✓	·	·	⊙	·	·	⊙	·	·	·	·	·	·	·
Linos+, 2003, [41]		·	·	·	⊙	·	·	✓	·	·	·	·	⊙	·	·	·	·	·
Linos+, 2007 [42]		·	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·
Marinescu+, 2007, [43]		·	✓	·	·	·	·	⊙	·	·	·	·	·	·	·	·	·	·
Mayer+, 2012, [44]		·	·	·	·	·	·	⊙	·	✓	·	·	·	·	·	·	·	·
Moise+, 2005, [46]		·	⊙	·	·	·	·	✓	·	·	·	·	⊙	·	·	·	·	·
Moise+, 2006, [47]		·	⊙	·	⊙	·	·	✓	·	·	·	·	⊙	·	·	·	·	·
Moonen, 1997, [49]		·	⊙	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
Morris+, 2010, [50]		·	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·
Nguyen+, 2012, [51]		✓	·	·	·	·	·	⊙	·	✓	·	·	·	·	·	·	·	·
Perin+, 2010, [53]		·	⊙	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
Pfeiffer+, 2011, [54]		·	·	·	·	·	⊙	✓	·	·	·	·	·	·	·	·	·	·
Pfeiffer+, 2012, [55]		·	·	·	⊙	·	·	⊙	·	⊙	·	·	⊙	·	✓	·	·	·
Polychniatis+, 2013, [57]		✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·
Ricca+, 2001, [58]		·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	✓
Salah+, 2003, [59]		·	·	✓	⊙	·	·	·	·	·	·	·	·	·	·	·	·	·
Schink, 2013, [60]		✓	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·
Strein+, 2006, [63]		·	·	·	·	·	·	⊙	·	✓	·	·	·	·	⊙	·	·	·
Synytskyy+, 2003, [64]		·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·
Synytskyy+, 2003, [65]		·	⊙	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·
Tomassetti+, 2013, [67]		✓	·	·	·	·	·	·	·	⊙	·	·	⊙	·	✓	·	·	·
Tomassetti+, 2014, [68]		✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·
Vetro+, 2012, [70]		✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·
Weijun+, 2008, [71]		✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
Yazdan+, 2011, [72]	f	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
Yazdan+, 2012, [73]	f	·	·	·	✓	·	·	·	·	·	⊙	·	·	·	·	·	·	·
Yazdan+, 2012, [74]	f	·	·	·	·	·	·	·	✓	·	⊙	·	·	·	·	·	·	·
Zheng+, 2007, [75]		·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	✓
Primary			*3*	*4*	*9*	*2*	*2*	*9*	*6*	*7*	*3*	*4*	*3*	*3*	*2*	*1*	*1*	*2*
Auxilliary			9	0	9	0	3	9	1	2	7	0	5	1	2	0	0	1
Total			**12**	**4**	**18**	**2**	**5**	**18**	**7**	**9**	**10**	**4**	**8**	**4**	**4**	**1**	**1**	**3**

* Primary goals are indicated by ✓, secondary goals are indicated by ⊙, related studies are indicated by the same letter in the "related" column.

work in native *IDEs* or as *Generic Tool Support*. Established techniques for *Software Metrics*, *Fault Diagnosis*, *Clone Detection and Resolution*, and *Security Analysis* have been adapted in the context of cross-language analysis. Although the aforementioned goals are distinct enough to form respectable groups, there are noteworthy overlaps among the problem space and the findings of the respective studies. For example, most studies targeting model reconstruction also have a respectable contribution to basic fact extraction and reverse engineering techniques.

Table 8 summarizes the distribution of the high-level goals targeted by the *58* technology papers. Overlapping studies that were based on the same or closely related work are labeled by the same letter in the "related" column. A number of papers are dedicated to one specific goal, while others address more goals. In the latter case, it is common that not all goals of the study are treated with the same amount of detail. In recognition of this, Table 8 distinguishes between the *primary* and *auxiliary* contributions. Three studies [19, 58, 75] were identified as serving two primary goals, as both goals were treated to an equal extent (by consensus of the two authors of this SLR, who felt that downgrading either of the identified goals to auxiliary would hurt accuracy). [19] can alternatively be viewed as two independent sub-studies: one using static analysis for CIA, and the other using dynamic analysis for runtime profiling. [58] and [75] aim for model reconstruction and CIA, respectively, and use the outcome to drive further testing steps.

4.2.2 Study Characterisation

The taxonomy that resulted from our analysis over the selected papers is presented in Table 9. The table contains categories of attributes that are formed by identifying the main concepts from the primary studies, which are also related to our research questions. The attributes in each category are the results of synthesizing the extracted data from individual primary studies, first by open coding of the findings, which was followed by axial coding to group these into a limited set of (more abstract) classes.

We identified 15 attributes that could characterize the *main approach* taken in the primary studies. Note that the attributes are not mutually exclusive and can be at different levels of abstraction. This means that a primary study can be characterized by multiple attributes. By following the attributes, at first, we know whether a study uses static analysis, dynamic analysis, or both. Then we can get more information about, say, the applied dynamic analysis method using other attributes. For instance, we identified two granularity levels for dynamic analysis, one using machine-level execution traces, and one by user-level, coarse-grained flow of events. We also identified code instrumentation and runtime interception as two major techniques of carrying out dynamic analysis. One perspective would be to classify these attributes as information extraction techniques, but we decided to keep them as an approach attribute since they characterize the overall approach of the study to such a large

Table 9: Synthesized attribute framework of the studies.

	Attributes	Description
Approach	static analysis	application of static analysis approaches.
	execution trace analysis	application of dynamic analysis, using machine-level execution traces.
	navigation flow analysis	application of dynamic analysis, using user-level flow of hypertext navigations.
	automation level	fully automated, semi-automated, or manual (resp. indicated as '✓', '·', and 'M').
	evaluation study	presenting a benchmark or evaluation of existing approaches.
	unified model	uses a homogeneous information repository, populated from heterogeneous artifacts.
	modelware technology	uses model-driven technologies, e.g., Ecore metamodels, or XMI transformations.
	multiple parsers	uses multiple language-specific parsers to gather information from MLSSs.
	retargetable parser	uses a generic, or adaptable, parser to process multiple artifact types, or multiple languages in one file.
	island grammar	approaches directly inspired by island grammars, or fuzzy parsing.
	naming conventions	has an explicit dependency on naming conventions, and shared identifiers.
	statistical analysis	approaches based on predictive statistics; approximations based on data mining.
	formalism & theory	studies presenting a respectable formal model as the basis of their approach.
	code instrumentation	an exercise of dynamic analysis based on code instrumentation.
	runtime interception	an exercise of dynamic analysis based on intercepting events during execution.
Information Extraction	parsing	uses traditional parsing/syntactical analysis to gather data from system artifacts.
	lexical analysis	uses light-weight and selective analysis of artifacts, often targeting small code structures, and no final parse tree.
	search & regular expr.	manipulation of regular expressions, or use of pre-fabricated search API to scan for data (e.g. the Eclipse search API).
	GUI event listener	intercepts GUI communication to gather data about runtime behavior.
	decompiler	use of a decompiler or disassembler to gather system data.
	debugger interception	overrides a feature of the runtime environment, such as a debugger or browser console, to gather data.
	web crawler	exercising a web crawler to trigger dynamic analysis.
	profiling	application of single or hybrid profiling methods, e.g. probe-based and sample-based.
Artifacts	source code	artifacts containing general-purpose programming or scripting languages.
	compiled artifacts	binary code or any other machine-readable code such as Java Bytecode or .NET CIL.
	web pages	static, dynamic, or Web 2.0 Rich Internet Application, web documents.
	SQL artifacts	standalone or embedded database-related artifacts, mostly in a dialect of SQL.
	configuration & deployment info.	any peripheral artifact containing data about configuration or deployment of a system, including component interface descriptions (such as CORBA IDL).
Domain	web based	web applications
	component based	systems built with a full-blown or partial component-based design and technology.
	distributed	distributed systems
	parallel systems	systems with a high degree of parallelism, as seen in high-performance computing.
	Java frameworks	contemporary, often configuration-rich, Java frameworks.
	.NET	studies tightly relying on the .NET technology and its languages.
	general	generic software systems with no specificity.
Implement	none	studies presenting no implementation.
	proof-of-concept	implementations explicitly characterised by the authors as "prototype," "partial," "preliminary," "lightweight," etc.
	laboratory	an implementation offered by the authors with no specific characteristic revealing maturity level.
	commercial	commercial or commercial-quality implementations.
	eclipse-plugin	implementations based upon and/or distributed using the eclipse plugin platform.
	publicly available	yes, if a publicly-available release of the implementation is stated in the paper.
Evaluation	none	no evaluation is presented in the study.
	toy example	a highly preliminary evaluation, often using a minimal code base.
	open source	evaluation(s) based on open source systems.
	industrial codebase	evaluation(s) based on real world industrial code bases.
	human subjects	involving third-party human subjects in the evaluation, i.e., usability studies, controlled experiments and questionnaires.
Languages		languages involved in the study, reported verbatim, as claimed by authors

Table 10: Studies incorporating static analysis.*

		Approach										Info. Extract				Artifacts					Domain					Implementation					Eval.				Languages
		Unified Model	Modelware Techn.	Retargetable Parser	Multiple Parsers	Island Grammar	Naming Conventions	Formal & Theoretic	Statistical Analysis	Evaluation Study	Fully Automated	Parsing	Lexical Analysis	Decompile	Search & Reg. Exp.	Source Code	Compiled Artifacts	Web page	SQL Artifacts	Config. & Deploy.	General	Web Based	Component	Java Frameworks	.NET	Proof-of-concept	Lab Implementation	Commercial Quality	Eclipse Plugin	Publicly Available	Toy Example	Open Source	Industrial Code	Humane Subj.	
R.E.	[35]	·	·	·	·	·	·	·	·	·	M	·	·	·	·	✓	·	·	✓	·	✓	·	·	·	·	–	–	–	–	–	–	–	–	–	Cobol, Assembler
	[43]	✓	✓	·	✓	·	✓	·	·	·	✓	·	✓	·	·	✓	·	·	✓	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	✓	·	Java, SQL
Query	[32]	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	✓	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	Cobol, CSP, JCL
	[37]	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	✓	✓	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	Cobol, JCL, SQL
	[38]	·	·	·	·	·	·	·	·	✓	–	–	–	–	–	✓	·	·	·	·	✓	·	·	·	·	–	–	–	–	–	·	·	✓	·	C/C++, Java
Visual	[40]	✓	·	·	✓	·	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	*C, Lisp*
	[53]	✓	·	·	·	·	✓	·	·	·	✓	✓	✓	·	·	✓	·	·	✓	✓	·	·	·	✓	·	·	✓	·	·	✓	·	·	✓	✓	Java, XML, SQL
	[73]	✓	✓	·	✓	·	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	✓	✓	C, XML
Arch	[13]	·	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	✓	·	·	·	✓	·	·	·	·	·	✓	·	C, Fortran, Unix API
	[28]	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	·	·	✓	✓	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	SQL, HTML, VBScript
Parse	[29]	✓	·	✓	·	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	C, TCL/Tk
	[65]	·	·	·	·	✓	·	·	·	·	✓	✓	·	·	·	·	·	✓	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	JS, HTML, VB
Dependency Analysis	[16]	·	·	·	✓	✓	·	·	·	✓	·	·	✓	·	·	✓	·	·	✓	·	✓	·	·	·	·	·	✓	·	·	·	·	✓	·	·	Java, SQL, XML
	[17]	·	·	·	·	✓	✓	·	·	·	·	·	✓	·	✓	✓	·	·	✓	✓	✓	·	·	·	·	·	✓	·	✓	·	·	✓	✓	✓	Java, XML
	[41]	·	·	·	·	·	·	·	·	·	✓	·	✓	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	✓	Java, JNI, C/C++
	[46]	✓	·	·	✓	·	✓	·	·	·	✓	✓	✓	·	·	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	·	·	✓	✓	·	·	Java, JNI, C/C++
	[47]	✓	✓	·	✓	·	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	·	·	·	✓	·	·	C, Perl, Python, TCL
	[54]	✓	✓	✓	·	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	✓	·	·	✓	·	·	·	✓	·	·	✓	·	✓	·	·	∞ (EMF metamodel)
	[57]	·	·	·	·	·	·	·	✓	·	·	·	✓	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	✓	·	✓	∞
	[68]	·	·	·	✓	·	·	·	✓	·	✓	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	✓	✓	✓	·	·	∞
	[70]	·	·	·	·	·	·	·	·	·	✓	–	–	–	–	–	–	–	–	–	✓	·	·	·	·	·	·	·	·	·	·	✓	·	·	∞
CIA	[18]	✓	·	✓	·	·	·	✓	·	·	✓	✓	·	·	·	✓	·	·	✓	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	✓	·	*Java,* CORBA, SQL
	[31]	✓	✓	·	·	·	·	·	·	·	✓	–	–	–	–	·	·	✓	✓	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	HTML, SQL
	[39]	✓	✓	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	✓	✓	✓	·	·	Java, Junit, UML
	[74]	✓	✓	·	✓	·	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	✓	·	C, XML
	[75]	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	✓	✓	·	·	✓	✓	·	·	✓	·	·	·	·	·	·	✓	·	Java, SQL, HTML, Perl, C
Refactoring	[15]	·	·	·	·	·	·	·	·	·	✓	✓	✓	·	✓	✓	·	·	·	✓	·	·	·	✓	·	✓	·	·	✓	·	✓	·	·	·	Java, XML, JSP
	[23]	·	·	·	·	·	·	·	·	·	✓	·	·	·	✓	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	✓	✓	·	·	·	·	Java, Fit Test
	[33]	·	·	·	·	·	·	·	·	·	✓	·	✓	·	✓	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	✓	·	·	·	·	·	Java, Groovy
	[44]	·	✓	·	✓	·	✓	·	·	·	·	✓	·	·	·	✓	·	✓	✓	·	✓	·	·	·	·	✓	·	·	✓	·	·	·	·	·	Ruby, XML, Java
	[51]	✓	·	·	·	·	·	·	·	·	✓	·	✓	·	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	✓	·	·	✓	·	·	JS, HTML, PHP
	[60]	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	✓	·	✓	·	·	·	·	✓	·	·	·	✓	✓	·	·	·	Java, SQL
	[63]	✓	·	·	✓	·	·	✓	·	·	✓	✓	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	·	·	✓	·	✓	·	✓	·	·	.NET, Java, ASP, HTML, XML
Flow	[30]	·	·	·	✓	·	·	✓	·	·	✓	✓	·	·	·	✓	·	·	·	✓	·	·	·	✓	·	✓	·	·	✓	✓	·	✓	·	·	Java, XML
	[49]	✓	·	·	✓	·	✓	·	·	·	–	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	–	–	–	–	–	∞
	[72]	✓	✓	·	✓	·	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	✓	·	C, XML
Clones	[1]	·	·	·	·	·	·	·	·	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	·	✓	✓	·	·	·	·	·	✓	·	·	C#, J#, VB.NET
	[11]	✓	·	·	·	·	·	·	·	·	✓	✓	·	·	·	✓	·	✓	✓	·	·	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	HTML, ASP, JS, CSS
	[36]	✓	·	✓	·	·	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	·	·	·	·	✓	·	✓	·	·	·	·	✓	·	·	.NET languages
	[64]	·	·	·	·	✓	·	·	·	·	✓	✓	·	·	·	✓	·	✓	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	JSP, VBScript, HTML
Type ch	[8]	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	·	✓	·	C++, CLS
	[24]	·	·	·	✓	·	·	✓	·	·	✓	✓	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	✓	·	✓	·	·	C, OCaml
	[25]	·	·	·	✓	·	·	✓	·	·	✓	✓	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	·	Java, C, JNI
M	[42]	·	·	·	·	·	·	·	·	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	·	✓	✓	·	·	·	·	·	✓	✓	✓	.NET CIL
IDE	[55]	✓	✓	·	·	·	·	·	·	·	✓	·	✓	·	·	✓	·	✓	✓	✓	✓	·	·	·	·	✓	·	·	·	✓	·	✓	·	✓	∞
	[67]	·	✓	·	·	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	✓	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	Java, XML
	46	25	11	4	23	4	6	5	2	2	28	31	11	2	4	39	5	9	14	13	28	7	7	3	3	28	13	1	7	10	12	18	13	7	
																										no impl:4					*no eval:10*				

* The letter M in the first column indicates the Metrics goal. Cell value '–' means that respective data was not provided.
Fully Automated column values are either '✓': fully automated, '·': semi-automated, or 'M': manual.
Languages in *italics* have less evidence of support, and '∞' indicates language-neutral or language-parametrized approaches.

Table 11: Studies incorporating dynamic analysis.*

		Approach							Info Extraction					Artifacts				Domain			Implement.				Eval.				Languages
		Execution Trace	Navigation Flow	Code Instrumentation	Runtime Interception	Hybrid analysis	Evaluation Study	Fully Automated?	Web Crawler	GUI Event listener	Decompiler	Debugger Interception	Profiling	Source Code	Compiled Artifacts	Web page	SQL Artifacts	Web Based	Distributed	Parallel Systems	Proof-of-concept	Lab Implementation	Commercial Quality	Publicly Available	Toy Example	Open Source	Industrial Code	Human Subj.	
reverse engineering	[3]	✓	✓	✓	·	✓	·	✓	·	✓	·	·	·	·	·	✓	✓	✓	·	·	✓	·	·	·	·	✓	·	·	PHP, SQL
	[6]	✓	✓	·	✓	·	·	✓	·	✓	·	✓	·	·	·	✓	·	✓	·	·	·	✓	·	✓	✓	·	✓	·	JS, HTML, DOM
	[9]	·	✓	·	·	✓	·	·	·	·	·	·	·	·	·	✓	·	✓	·	·	✓	·	·	·	✓	·	·	·	JS, HTML, ASP
	[10]	·	·	·	·	·	✓	–	–	–	–	–	–	·	·	✓	·	✓	·	·	✓	·	·	·	✓	·	·	·	HTML, PHP
	[22]	·	✓	·	✓	✓	·	·	·	✓	·	·	·	✓	·	✓	·	✓	·	·	✓	·	·	·	·	·	✓	✓	JS, HTML, ASP, PHP
	[58]	·	✓	·	·	✓	·	·	✓	·	·	·	·	·	·	✓	·	✓	·	·	✓	·	·	·	·	·	✓	·	JS, HTML
	[59]	✓	·	·	✓	·	·	·	·	·	·	✓	·	✓	✓	·	·	·	✓	·	·	✓	·	·	✓	·	·	·	Java, C/C++, VB
	[71]	✓	✓	✓	·	✓	·	–	·	·	·	·	·	·	·	✓	✓	✓	·	·	·	·	·	·	·	·	·	·	–
metrcs	[19]	✓	·	✓	·	✓	·	✓	·	·	✓	·	·	✓	✓	·	·	·	✓	·	✓	·	·	·	·	·	✓	·	Java, C/C++
	[50]	✓	·	·	✓	✓	·	✓	·	·	·	·	✓	✓	·	·	·	·	·	✓	✓	·	·	·	·	✓	✓	·	C/C++, Fortran, Python
other	[2]	✓	✓	✓	·	✓	·	✓	·	·	·	·	·	·	·	✓	✓	✓	·	·	·	·	·	·	·	·	·	·	PHP, SQL
	[7]	✓	·	✓	·	·	·	✓	·	·	·	·	·	·	✓	·	·	·	✓	·	·	·	✓	·	✓	·	✓	·	Java, C/C++, .NET
	12	8	7	5	4	8	1	6	1	3	1	2	1	5	4	8	2	8	3	1	7	2	1	1	5	2	6	1	
																					no impl:2				*no eval:2*				

* In this and all remaining tables, the symbol '–' indicates cases where no data is presented in the paper, or cases where the aspect is *not applicable* to the respective study.

extent. Unified model refers to the use of a central knowledge repository to represent all relevant information regarding the system artifacts that is needed by the later analyses. This pattern comes in many different forms, at various granularity levels, and using a wide range of implementation technologies. Homogeneous/centralized model repository, unifying model, internal/intermediate representation, common schema, universal language representation, are among the variations used in the primary studies, which were all classified as unified model.

We identified eight main techniques for *information extraction*, six types of *artifacts*, and seven *technology domains* that were subjected to cross-lingual analysis. As these attributes are typically referred to using specific, and widely shared names, there were no significant challenges in determining the relevant attributes in the primary studies.

Six attributes were identified to describe the provided *implementation*, and five for the reported *evaluation*. These two aspects are largely aiming at characterizing the overall maturity level. We found that not all studies presented enough information to establish the maturity level. Moreover, among those studies with "enough" information, the lack of a common vocabulary makes it difficult to objectively determine the maturity in a uniform manner across different studies. For instance, prototype implementations reported in some studies appeared to be more involved than full implementations reported in other studies. Therefore, we took a conservative approach by (1) choosing a minimal classification, (2) leaning towards the author's self-characterization in studies

that provide such arguments, and (3) building a mutual consensus in judging papers with insufficient supportive information, or cases where the original vocabulary seemed out of place. We suggest a liberal interpretation of the identified attributes, nevertheless, the provided information still provides a decent insight over the maturity levels provided. See Table 9 for a full description of the attributes.
Characterisation Results – We report the final classification in two tables. Table 10 presents studies exclusively based on static analysis. Table 11 covers studies that use dynamic analysis, either exclusively, or in a hybrid approach in combination with static analysis. For the hybrid case, the tabular presentation focuses on the aspects related to dynamic analysis. Together, both tables provide a succinct characterization of the 58 technology papers. Note that a relatively small number of primary studies did not contain explicit information on all aspects of our characterization. In such cases, we refrained from inserting our own interpretations beyond the available information. However, as a result of this, some studies are characterized with seemingly insufficient attributes with respect to our classification.

4.3 Research Questions

This section addresses our target research questions based on the selected papers. Note that since the seven position papers do not present a full study tackling cross-lingual analysis, they are not included in the analysis in this section. We refer to Section 4.3.1 for a discussion of those position papers. The discussion in this section revisits the research questions and information derived from the 58 technology papers. It is largely based on the synthesized information in Table 11 and Table 10, and were possible enriched with concrete examples from the primary studies that were not presented in the tables, to connect the discussion to specific technologies. Finally, note that the static-analysis aspects of those studies taking a hybrid (i.e., static + dynamic) approach shown in Table 11 are fully taken into account in this section.

RQ1 *What approaches have been used for cross-language program analysis?*

Figure 5 shows the distribution of the attributes identified as the general approach. For the purpose of cross-language analysis, it is clear that static analysis has been (far) more applied than dynamic analysis in the studies (79% comparing to 7%). In addition, eight studies (14%) have applied a hybrid approach of static and dynamic analysis, to take advantage of (or trade-off) the characteristics of both approaches.

Most studies used *some form* of unified model (32, 56%) to tackle cross-language analysis. This observation is not too surprising as this architectural pattern is also very common among single-language analysis tools, and resembles the use of an intermediate representation to connect alternative front-ends to the same back-end in compiler building [80]. It is also noticeable how model-driven engineering techniques (and

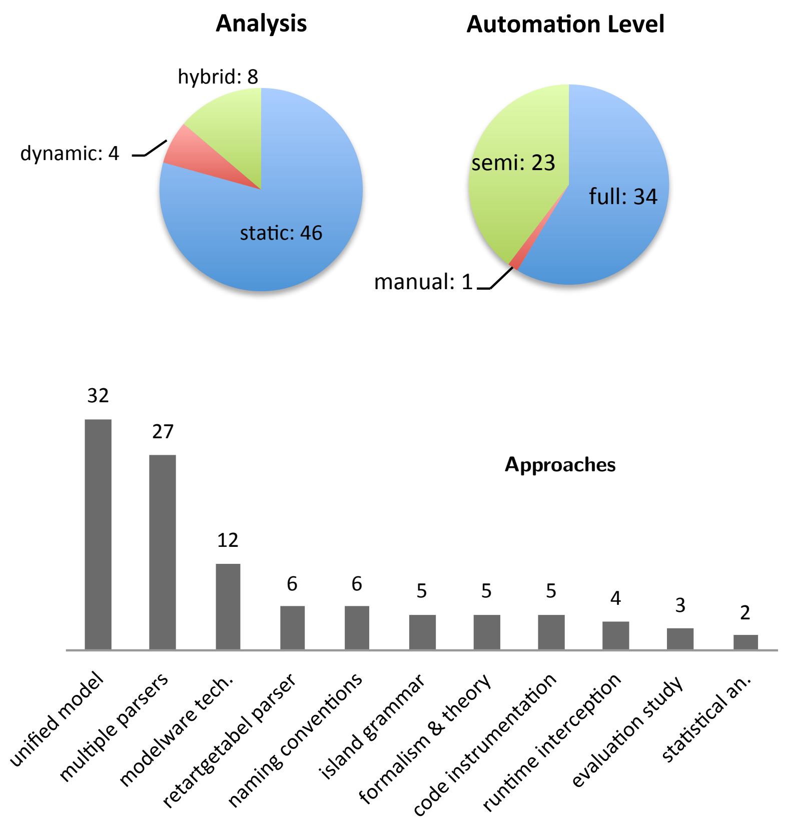


Figure 5: Frequencies and high level classification of the applied approaches.

technologies) have contributed to the domain of program analysis with 12 studies (21%), half of which were published after 2011.

To detect cross-language dependencies, six of the studies (10%) explicitly rely on (case-specific) naming conventions and identifiers, which is arguably a quite brittle (i.e. fault prone) approach, not just for dependency analysis but also during forward engineering.

Finally, only three of the studies (5%) explicitly aimed at reflecting on existing (third party) approaches for cross-language analysis, using benchmarks and evaluations. This low number hints at a certain immaturity in the community with respect to adequate evaluation initiatives, or a lack of collective evaluation efforts, such as call for contests and open challenges. Moreover, as such third-party evaluation studies aim at minimizing bias or preference towards a particular approach, this low number of 2nd-tier studies could negatively affect the overall confidence in results in this area.

RQ2 *What fact extraction methods are commonly used for analysis across multiple languages?*

The identification of information extraction methods was fairly straightforward in most papers, due to direct references to well-known techniques and tool-sets. Nev-

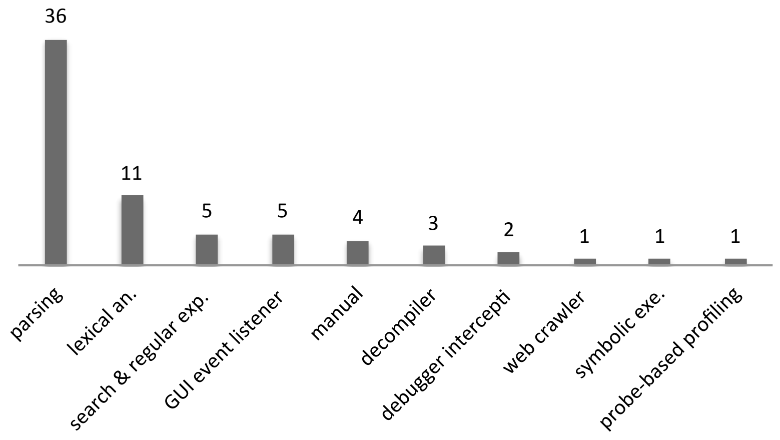


Figure 6: Frequency of information extraction methods.

ertheless, a number of papers did not provide enough information on this aspect to objectively identify the applied method. For example, in some cases the descriptions remained vague with respect to the differences between lightweight/island parsing and lexical analysis; instead island parsing was used as a catch-all phrase to refer to some form of robust semi-structured textual analysis. In such cases we made a judgment by building mutual consensus using all the information at hand, e.g. the implementation and evaluation details and the granularity of the extracted information.

The majority of the primary studies used parsing[27] (36, 62%), either due to extensive information needs, or simply because an already-available parser is the most cost-effective implementation. On the contrary, 11 studies used limited-scope lexical analyzers (19%), and 5 studies used regular expressions and the available search APIs to gather the desired information (9%). Four studies relied on manual fact extraction, either partially or completely [35]. In addition to the aforementioned facets in Table 11 and Table 10, there is one study that applies symbolic execution for cross-language analysis [50].

RQ3 *What types of facts are typically extracted to enable cross-language program analysis?*

Most primary studies report on the types of information that underlies their analysis (of course with a varying degree of detail). Some approaches are built on one major information type, while others use a wide range of extracted facts. Apart from very few exceptions, primary studies generally do not distinguish between the types of data that are needed for *intra-language* versus *inter-language* analysis. In other words, cross-language analysis aspects are not discussed as an extension to a general-purpose single/multi language analysis method, but rather as an integrated approach. This

[27]Either accurate or lightweight parsing.

made it difficult for many papers to decide what are the specific types of information enable the cross-language aspects of their analysis. When formulating this question, we set out to identify that specific information to provide developers of new cross-language analysis approaches with specific starting points for extraction. Based on these findings, Table 12 reflects on *all* types of information that is extracted and used in the primary studies. Note that studies are repeated if they extract multiple types of information and related studies have been grouped together and are indicated by underlining them.

As we can see, some of the identified information types have a certain degree of overlap. For instance, studies who extract function/method signatures use a portion of the program entities, too. However, as the constructing of a mutually-exclusive ontology of information pertaining to source code artifact was not the goal of this section, we decide to leave this overlap in place, and report on extracted information in close reference to the vocabulary used by the original authors.

A first look at the usage frequencies shows the significance of program entities to cross-lingual program analysis, with 35 (60%) primary studies stating a direct need to extract program identifiers, variables, methods, class names, and etc. This is mainly due to the necessity of a (cross-language) name resolution mechanism in majority of the studies - be it improvised and partial, or canonical and complete. String literals in the program are also gathered for essentially the same purpose (7, 12%), and if we add up the respective groups we can conclude that 37 primary studies have had the need for a form of name resolution. Coarse-grained structure of software artifacts, such as containment relation in the file structure or package inclusions, comes at the second place in usage frequency (14, 24%). Such relations can indicate a lot when it comes to detecting cross-language relations.

Program elements with direct impact on the *behavior* of software systems are frequently targeted in cross-language analyses. (1) Control flow and/or function call trees, (2) remote procedure calls (RPC) and/or cross-language procedure calls, and (3) network socket calls are used in 11, 4, and 2 studies, respectively. Method signatures are also used in 5 studies, although this refers to a completely static look up of the signature without considering the fan- in/out interactions. Note that although most data is collected using white-box analysis of the source code, back-box observation techniques such as client-server network traffic snooping have also been used.

We would like to highlight that the essence of many of the approaches not only resides in the information they capture, but perhaps even more importantly in the information that they filter out. Studies using island grammars are a prime example for this alternative perspective [64].

RQ4 *What internal representations of software artifacts are used to achieve cross-language analysis?*

During our investigation it became clear that primary studies have polarized views on the general issue of knowledge repositories. Some studies treat the design and

Table 12: Data extracted to support cross-language analysis.*

Data type	#	studies
program entities	34	[8], [9], [15], [17], [16], [19], [22], [18], [23], [28], [29], [32], [33], [37], [38], [39], [40], [42], [43], [44], [47], [46], [51], [55], [54], [57], [59], [63], [68], [67], [74], [72], [73], [75]
artifact structure	14	[9], [18], [19], [22], [28], [31], [39], [32], [37], [38], [40], [44], [54], [58]
control flow graph	11	[30], [7], [6], [13], [38], [40], [47], [49], [53], [63], [71]
abstract syntax tree	9	[13], [23], [28], [29], [30], [36], [55], [63], [68]
hyperlinks	9	[2], [3], [9], [22], [28], [31], [58], [71], [75]
database queries	8	[2], [3], [28], [43], [53], [60], [71], [75]
HTML tags	7	[9], [11], [28], [58], [64], [71], [75]
keyword/annotation	7	[16], [25], [24], [41], [47], [46], [53]
string literals	7	[2], [3], [17], [25], [24], [51], [55]
configuration data	6	[30], [53], [54], [72], [73], [74]
database schema	6	[2], [3], [18], [31], [43], [60]
data flow	6	[13], [30], [31], [40], [49], [75]
method signatures	5	[8], [24], [25], [41], [44]
PDG/SDG	4	[72], [73], [74], [75]
RPC	4	[59], [41], [46], [75]
cookie/session vars	3	[2], [3], [71]
workflow description	3	[32], [37], [38]
execution traces	2	[6], [7]
network socket calls	2	[13], [59]
bytecode strings	1	[1]
client-server traffic	1	[6]
code block structure	1	[16]
commit history	1	[70]
GUI interaction	1	[6]
runtime call stack	1	[50]
symbol table entries	1	[47]

* Related studies are indicated by underlining them.

implementation of this aspect as the key ingredient to tackling the problem of cross-language analysis, and, naturally, dedicate a significant portion of the paper to provide a detailed description of the proposed knowledge repository. On the other hand, other studies provide no, or very limited information; implicitly dismissing the topic as irrelevant. Nevertheless we decided to keep this research question in our SLR, and provide answers based on the available information.

Table 13 summarizes the available data on the implementation of knowledge repositories, or internal program representations. It reports the most specific description of the knowledge repositories is picked from the primary studies. For instance, in the studies included here, OMG's KDM models are implemented using EMF metamodels, and these EMF models are realized through XML, however, we report KDM with no further generalization.

Overall, the provided information implies a rather sporadic use of each technology, with little trace of continued use of (meta)models among different authors. 15 studies provided to little information to determine if and what intermediate representation was used, and nine more studies effectively mentioned no more specific than the fact that they using relational databases. KDM, Extended Entity-Relationship (EER), and proprietary Abstract Syntax Trees (AST) are each used by three studies, with the former two used by overlapping studies. KDM is the only standardized model schema in here [118]. It is noticeable how the Eclipse Modeling Framework (EMF) has been utilized increasingly often in the very recent years. So far, it is arguably the only implementation technology that gained a noticeable momentum within the community.

RQ5 *What higher level goals are targeted using cross-lingual program analysis?*

Using the information in Table 8, we can reflect on the higher level goals that are aimed at using cross-language program analysis. Some highlights that become apparent include:

1. *Comprehension* is the most sought after goal, with 15 papers (26%) having it as their primary goal, either using a text-based query mechanism (4) or some form of graphical visualisation (9) or view reconstruction (2).

2. Considering auxiliary and primary goals together, one-third of studies has pursued *graphical visualisations* (9 + 9 + 2 = 20, 34%).

3. *Dependency identification and analysis* is the second most popular target (9, 16%).

4. With *reverse engineering* as prerequisite to *comprehension* in our classification, we see 31% of studies have investigated ways to *extract* and *abstract* cross-lingual facts. Four additional papers (7%) focus on making reverse engineering more cost-effective.

Table 13: Use of internal representations and knowledge repositories.

Format	#	Studies
not specified	15	[8], [9], [10], [13], [16], [24], [25], [30], [31], [40], [57], [58], [63], [64], [71]
none	10	[1], [6], [15], [17], [23], [33], [35], [50], [65], [70]
relational db	9	[2], [3], [11], [22], [28], [41], [42], [59], [75]
EMF-based	5	[39], [44], [47], [54], [55]
AST	3	[29], [68], [67]
EER model	3	[32], [37], [38]
OMG's KDM	3	[72], [73], [74]
IR	2	[43], [49]
XML-based	2	[18], [19]
CodeDOM	1	[36]
D-Model	1	[51]
execution traces	1	[7]
FAMIX	1	[53]
GXL	1	[46]
jgraph	1	[60]
Total	58	

5. Considering auxiliary and primary goals together, 10 papers analyze cross-lingual flow of control or data (17%).

6. Although several studies contribute task-specific tools, only two studies(3.4%) aim at holistic tool support comparable to a general-purpose IDE.

RQ6 *Which languages and types of software artifacts have been subjected to cross-lingual analysis?*

Most studies, both technology as well as position papers, provide sufficient information to identify the *type of artifacts* on which they (intend to) apply cross-language analysis. However, we found that the position papers did not target specific *programming languages*, so we only use the technology papers to identify specific programming languages.

Table 14 gives an overview of the types of artifacts that are typically analyzed, and the number of studies that targeted this type. Note that studies are repeated if they analyze multiple artifact types and related studies have been grouped together and are indicated by underlining them. As could be expected, source code analysis is used in the majority of the selected studies, which is of course heavily influenced by the domain and scope of this SLR. Next, we see that database-related artifacts, web pages and configuration and build files have attracted considerable attention in the community - each appearing in a quarter of the primary studies. Compiled software

Table 14: Artifact types subjected to multi-language analysis.

Artifact	#	Studies
source code	43	[8], [11], [13], [15], [17], [16], [22], [19], [50], [18], [23], [25], [24], [29], [30], [32], [33], [35], [36], [37], [38], [39], [40], [41], [43], [44], [46], [47], [49], [53], [55], [54], [57], [60], [59], [63], [64], [68], [67], [74], [72], [73], [75]
database artifacts	18	[2], [3], [11], [16], [17], [18], [28], [31], [35], [37], [43], [44], [53], [55], [60], [61], [71], [75]
web page	18	[2], [3], [6], [9], [10], [11], [22], [28], [31], [44], [51], [55], [56], [58], [64], [65], [71], [75]
config/build files	13	[15], [17], [30], [45], [48], [53], [54], [55], [66], [67], [72],[73],[74]
binary/byte code	9	[1], [7], [9], [19], [24], [25], [42], [59], [71]
license	1	[12]
not specified	3	[10], [34], [70]

artifacts have also been subject to analysis in nine papers (14%), with two of them being related. Finally, three studies do not specify artifact types, and there is one "outlier" that includes license agreements into the analysis [12].

Table 15 summarizes the programming languages that are analyzed. Java and C/C++ are by far the most frequent studied languages, which could be explained by their widespread use, both in industry and academia. Given the heterogeneous nature of contemporary software systems, the significant research attention spent on database/SQL-related, HTML, and XML artifacts is also hardly a surprise. Another areas where multi-language artifacts are common is that of dynamic web applications, which is evidenced by the considerable attention to the scripting languages used to develop such applications . In total 13 primary studies (22%) consider web scripting languages, divided over JSP, ASP, PHP, VBScript, and JavaScript (JS).

When looking at languages whose interaction is typically analyzed together, we can identify the following pairs:

- *Java - C/C++* is targeted in eight studies; three of which specifically analyze the JNI interaction mechanism.
- *Java - XML* is analyzed in eight studies.
- *Java - SQL* is targeted in five studies.
- *C/C++ language extensions:* the interaction of native C/C++ code with external languages such as Fortran, Python, Perl, Tcl, OCaml, CLOS is analyzed in seven studies [8], [13], [24], [29], [40], [47], and [50].

A clear trend is the raised interest in approaches that are not tied to specific programming languages (marked by the symbol '∞' in Table 10). Since 2012, five studies have aimed

Table 15: Languages included in cross-language program analysis.

Language	#	Studies
Java	22	[7], [15], [16], [17], [18], [19], [23], [25], [30], [33], [38], [39], [41], [43], [44], [46], [53], [59], [60], [63], [67], [75]
C/C++	18	[7], [8], [13], [19], [24], [25], [29], [38], [40], [41], [46], [47], [50], [59], [72], [73], [74], [75]
HTML	13	[6], [9], [10], [11], [22], [28], [51], [31], [63], [64], [65], [75], [71]
SQL	11	[2], [3], [16], [18], [28], [31], [37], [43], [53], [60], [75]
XML	11	[15], [16], [17], [30], [44], [53], [63], [67], [72], [73], [74]
JS	7	[6], [9], [11], [22], [51], [58], [65]
generic	6	[49], [54], [55], [57], [68], [70]
.NET	5	[1], [7], [36], [42], [63]
ASP	4	[9], [11], [22], [63]
PHP	4	[2], [3], [10], [22], [51]
Cobol	3	[32], [37], [35]
Python	2	[47], [50]
Fortran	2	[13], [50]
JSP	2	[15], [64]
unknown	2	[10], [71]

at tackling the challenges of cross-language program analysis using approaches that can (ideally) handle “all” programming languages in a generic manner, with no (or little) re-implementation costs.

Finally we should point out that there is considerable variation in granularity level among the selected studies. For example, the level of engagement in language constructs could make it seem that studies such as [68] and [70] should be considered “borderline” studies or even should not be included in our selection. Nevertheless, as their research goals fall well within the scope of this SLR, we decided that they should be included.

RQ7 *Which technological domains have attracted more attention in the literature on cross-lingual analysis?*

Identifying the technology domains in which cross-language program analysis is (to be) applied is relevant to both position papers as well as technology papers. Table 16 reflects on the frequencies of the studied domains, based on verbatim data as presented in the primary studies, with minor generalization. Note that a single study can be associated with more than one technological domain and related studies are grouped together and underlined.

Only one position paper investigates systems with multiple and heterogeneous distribution licenses; The same is true for product lines. Three studies present an

Table 16: Overview of targeted technology domains.

Domain	#	Studies
multi-licensed	1	[12]
product lines	1	[48]
parallel	1	[50]
.NET	3	[1], [36], [42]
distributed	3	[7], [19], [59]
Java frameworks	4	[15], [30], [45], [53]
component-based	9	[12], [13], [18], [48], [54], [75] [72], [73], [74],
web-based	16	[2], [3], [6], [9], [10], [11], [22], [28], [31], [34], [51], [58], [64], [65], [71], [75]
general	32	[8], [13], [17], [16], [23], [25], [24], [29], [30], [32], [33], [35], [37], [38], [39], [40], [41], [43], [44], [47], [46], [49], [55], [56], [57], [60], [61], [63], [66], [67], [68], [70]

approach exclusively based on the inherent characteristics of the .NET framework, while five studies are focused on the ubiquitous configuration-rich Java frameworks. Parallel systems are tackled by three primary studies - all of which using a dynamic analysis approach. Component-based systems are investigated by nine studies - seven use static analysis, and two are position papers. The top most special-purpose visited domain is web applications, which is not surprising considering the common design and implementation techniques of such applications (and was predicted in one of the earlier position papers by Kienle and Muller in 2001 [34]). 31 studies discuss a solution that is considered general, i.e. not tied to any particular technology domain (other than being *multilingual*, of course).

RQ8 *How rigorously are newly proposed approaches tested and evaluated?*

This question naturally only applies to the technology papers. As the evaluation is co-related with tool support for, we start with a brief overview of the reported implementations, before we continue to answer this research question.
Tool support – Of the 57 technology papers, six do not report on implementation initiatives, and two report commercial quality tool support. The remaining 49 papers present laboratory implementations, 35 of which were self-described as "prototype," "partial," "preliminary," or "lightweight." Although the maturity level varies greatly among the instances in this laboratory implementation category (from bare-bone hacks to moderately designed tool-sets with some usability features), they all share the common characteristic of not being deemed ready for unsupervised use by end-users. In fact, only 11 of the 49 laboratory implementations are described as being publicly available. Finally it was interesting to see that, from all implementations, seven were made using Eclipse as their implementation platform and packaged as an Eclipse extensions (only two of these were publicly available).

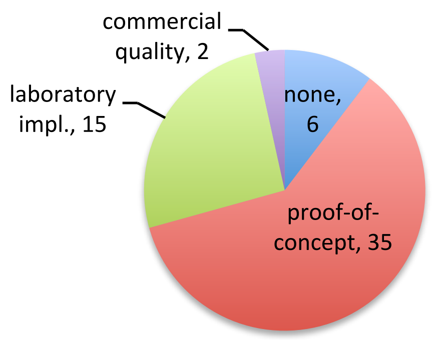


Figure 7: Conveyed maturity level of tool support.

Evaluations – As mentioned before, the lack of methodological rigor among the primary studies prevents us from directly reusing evaluation characterizations made by the authors. For example, some studies use the well-established term *empirical study* to characterize evaluation efforts that, at best, can be regarded as a "preliminary evaluation" by today's standards. Consequently, we used our own, more moderate but more objective, characterization of the evaluation processes.

Figure 8 summarizes these evaluation characteristics. It shows that 12 studies did not discuss evaluation, while another 10 exclusively rely on preliminary evaluations based on toy examples. Overall, we can conclude that over a third (38%) of the studies have little or no experimental evidence to back up their findings, whereas 36 studies (62%) do use larger open source and/or industrial code bases to evaluate their approach.

As we have seen in the discussion of RQ5, the main goal of (cross-language) program analysis is to support the developer with maintenance and evolution of (multi-language) software systems. In that light, we found it noteworthy to discover that *only eight studies (14%) involve human subjects as part of their evaluation*, either in the form of usability studies, stakeholder questionnaires, and the like.

The challenges of conducting unbiased, repeatable, experimental evaluation using human-subjects is long known to the scientific community [119, 120]. Moreover, access to sufficient domain experts as reliable assessors is another obstacle in such evaluations. However, despite these challenges, the low number of studies involving human subjects is striking, even though we should point out that equally low rates have been documented in SLRs that have a somewhat different focus, but a largely overlapping community and domain [85].

Based on our, possibly subjective, interpretation of available information, we can only conclude that the state of real-life evaluation in cross-language analysis research is inadequate. There is almost no demonstration of day-to-day applicability of proposed approaches to industrial cases, and this gets even worse if we consider evaluation to formal empirical studies. In addition to the aforementioned lack of collective initiatives or frameworks for evaluation that were discussed for RQ1, these observations point to a

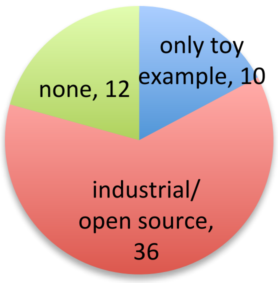


Figure 8: Frequency of evaluation subjects.

lack of maturity of this particular research area, and a considerable gap to practice. We speculate that future research efforts could gain extensive benefits if this gap would be addressed, and evaluations would be conducted more rigorously.

4.3.1 Position papers

Table 17 gives an overview of the seven papers who do not propose a new analysis method or technology, but rather share their experiences, insights, and possible directions for future research to the community. In the remainder of this section, we will discuss some highlights from these papers.

Obviously all primary studies in this review value the topic of cross-lingual program analysis, however, *five* papers spend a significant portion to advocate the necessity of making progress along this direction before the research community – often including figures on the abundance of multi-language software systems. *One* study ([56]) backed

Table 17: Overview of position papers.

		Contribution				Maintenance Task						Artifact Type					Domain				
	short paper	advocacy	future research	taxonomy	feasibility an.	CIA	reverse eng.	visualization	refactoring	navigation	license compat.	source code	config. & deploy	license	web page	SQL artifacts	component	mutli-licensed	web based	Java frameworks	product Lines
Boughanmi, 2010, [12]	✓	✓	✓	·	·	✓	·	·	·	·	✓	✓	·	✓	·	·	✓	✓	·	·	·
Kienle+, 2001, [34]	·	✓	·	·	·	·	✓	·	·	·	·	–	–	–	–	–	·	·	✓	·	·
Mayer+, 2013, [45]	·	·	·	✓	·	–	–	–	–	–	–	✓	✓	·	·	✓	·	·	·	✓	·
Moonen, 2013, [48]	·	✓	✓	·	·	✓	✓	·	·	·	·	✓	✓	·	·	·	✓	·	·	·	✓
Pfeiffer+, 2012, [56]	·	✓	·	·	·	·	·	✓	✓	✓	·	✓	·	·	✓	·	–	–	–	–	–
Schink+, 2011, [61]	✓	·	·	·	✓	·	·	·	✓	·	·	✓	·	·	·	✓	–	–	–	–	–
Tomassetti, 2013, [66]	✓	✓	·	✓	·	–	–	–	–	–	–	✓	✓	·	·	·	–	–	–	–	–

up their speculations using a *controlled experiment*, demonstrating the benefits of using IDEs equipped with cross-language analysis to various software maintenance tasks.[28]

Two recent studies ([45, 66]) argue that the community's understanding of the structure and context of cross-language interactions is inadequate, and use grounded theory to gain a better insight. Both studies analyse examples of contemporary, framework-rich, Java applications, with heavy use of DSLs for configuration, to identify generic patterns of cross-language interactions. [66] uses the findings of another overlapping study as corpus [70], which relies on cross-language commits to "spot the presence of" (but not pinpointing) logical interactions. [29]

Roughly published at the same time, Moonen [48] reasons how progress in cross-lingual program analysis could benefit quality assurance, by enabling recommendation systems that oversee the evolution of *families* of component-based cyber-physical systems. This study also considers the involvement of configuration DSLs as a necessity in analysing component-based systems.

[12] is the only paper in our SLR that speculates about the application of cross-language analysis to software licensing. The author proposed to identify and update subsystems' interactions for systems built on top of *heterogeneously-licensed* subsystems, to analyse (and possibly refactor) how these interactions could affect licensing. However, this line of work seems to have been abandoned, as no related paper has been published by the author since.

From the publication data in Table 17 it is evident that the majority of position papers is from 2010 or later. However, keeping in mind that our selection started from 1995, there is one outlier that is not all that recent: In 2001, at the start of Web 2.0 initiatives and an increase of web applications, Kienle and Müller ([34]) made the case that the state-of-the-art reverse engineering methods of that time should be extended with multilingual and cross-lingual capabilities, to be able to cope with the prospective flood of web-based systems. The findings of this SLR confirm that many studies have addressed this challenge in the decade following the publication of that paper.

However, the fact that today's capabilities to analyze web-based systems are far from perfect, signals that this is still a relevant open research question. We argue that a significant increase in attention paid to cross-lingual program analysis is needed, partly to address these web-based systems, but even more so in the light of the abundance of open-source software frameworks and cross-platform mobile applications.

[28]Although the term "controlled experiment" has been used by at least two other studies in our SLR, this is actually the only study that presents an experiment which has sufficient methodological rigor.

[29]In addition to the position papers, a number of technology papers present taxonomies for cross-language relation types, such as [40] and [55]. However, such taxonomies are often partial and closely bounded to the proposed analysis method.

5 Discussion

This SLR reviews cross-language program analysis papers published in peer-reviewed scientific journals and conferences. The aim is to support the ongoing research by compiling a library of related papers, and a classification of tackled problems, analysis approaches, analysis subjects, and evaluation methods. Based on our interpretation of the gathered data, we discuss a number of recommendations that may help the research community.

5.1 Implications for research

Research on more language-generic approaches – A major portion of the available scientific literature on cross-language program analysis consists of studies that aim at specific technology stacks, specific languages, and specific types of cross-lingual dependencies. While such specificity should not necessarily be regarded as a drawback (in particular for pioneering exploratory studies), a lack of generalizable solutions will likely lead to challenges support the increasing numbers of multi-language software systems.

While fully generic approaches may seem beyond reach at the moment, we recognize the substantial potential in research on generalizable approaches with respect to programming languages. We want to highlight the five recently-published papers identified in this SLR that address that line of work as part of their research agenda (see RQ6, Table 15). We consider this a highly promising avenue for further research, and one that will prove to be essential for dealing with the ever-increasing complexity of modern heterogeneous software systems and their maintenance needs.

Reduce dependance on heuristics – We identified 11 studies which explicitly mention that their proposed solution depends on heuristics. Typically, this means that they leverage certain, non-generalizable, patterns in extracted facts, such as case-specific naming conventions and directory structures, or even the structure of specific development teams [26]. In addition to these, a number of other studies also depend on similar case-specific solutions, although they do not explicitly mention the use of heuristics. The disadvantage of using heuristics is that they often limit the generalizability of the solution, which is especially problematic in cases where the context in which the heuristics do, or do not, apply is not clearly described. Unfortunately, we noticed a tendency among the studies that employed heuristics to leave out some of these details.

On the other side of the spectrum, we also identified only five studies that base their approach on formal or theoretical underpinnings (see RQ1, Figure 5). One main advantage of building on such an underlying formalism is that reasoning about the applicability of the solution in other domains is more accessible, hence increasing the odds for repeatability and incremental work in the community. Based on our observations while conducting this SLR, we speculate there is still considerable open space for new approaches that are developed around a sound model, an unambiguous grammar, or a verifiable theory, rather than depending on potentially brittle and hard to generalize heuristics.

Revisiting knowledge repositories – With many research studies heavily relying on the design and implementation of their knowledge repository, and several others completely circumventing the topic, we sense a polarized view over the importance of the issue. The quest for an ideal knowledge repository (or internal representation), is not exclusive to cross-language analysis, and has attracted a significant research effort over the years [91, 118]. Confirmed by our findings in RQ4, the sporadic usage of each repository technology implies that the state of the practice is still far from a widely accepted solution. Although, at the syntax level, the (faint) trend to use standardized, open source, solutions like KDM and EMF seems plausible, we consider more customized and semantic-rich solutions for cross-language analysis to be a fruitful research topic.

Universal name-resolution mechanisms – We observe that the significant use of program entities and string literals (see RQ3, Table 12) stems from the recurring need for cross-language name resolution. The studies on cross-language dependency discovery and refactoring are prime examples of such use. In the light of the aforementioned need for more generic approaches, this observation points us in the direction of a building block that could further cross-language analysis research: a universal name resolution mechanism. The term universal should not be interpreted too strictly; there is need for a reusable approach without strong dependencies on the actual programming languages analyzed. Although there is a substantial body of research on name resolution for various programming languages, we did not identify any generic approach in the primary studies included in this SLR.

More collaboration and evaluation with industry – Our findings show that empirical studies on real-world systems are extremely rare in the domain of cross-language program analysis, and only few studies have demonstrated their applicability to day-to-day maintenance activities. This can be explained by the fact that realistic empirical studies in industry are only possible with fairly mature and user-friendly tool; whereas many primary studies in this SLR present either no or only proof-of-concept prototypes. However, it is important to address this aspect by closer collaboration with industry, as the lack of empirical evaluation in real-world scenarios will likely lead to lower confidence in reported findings, and decreased opportunities for adoption by practitioners.

5.2 Implications for the community

In addition to research oriented implications, the findings collected in this SLR also led to some insights in the research community conducting cross-lingual program analysis research, and to ideas for potential improvements.

Limited incremental work – Table 18 gives an overview of cross-citations among the studies included in this SLR, without considering self-citations. The overview shows that

Table 18: List of cross-citations among the 65 primary studies. Self-citations are not considered.

Study	Cited Studies
[1]	[36]
[3]	[22], [65]
[9]	[20], [34], [58]
[10]	[20], [34], [58]
[12]	[37], [40], [41], [42], [46], [47]
[16]	[25], [46], [65]
[17]	[25], [46]
[22]	[11], [26]
[25]	[8]
[32]	[40]
[34]	[27], [37], [58]
[36]	[41], [42], [46], [63]
[37]	[32]
[42]	[19], [63]
[44]	[15], [17], [19], [33], [37], [40], [41], [42], [46], [54], [61], [63]
[45]	[17], [42], [54], [55], [56], [61], [63]
[46]	[19], [28], [37], [41]
[47]	[19], [27], [28], [37], [41]
[51]	[33], [62]
[52]	[43]
[53]	[43]
[55]	[63]
[56]	[15], [37]
[57]	[17], [54]
[59]	[27]
[60]	[44], [63]
[61]	[15], [33], [42], [63]
[63]	[37]
[64]	[11]
[65]	[11] [27]
[68]	[45], [55], [56]
[67]	[44], [55], [56]
[66]	[44], [55], [56]
[70]	[54]
[71]	[9], [20], [22], [26]
other 30	no citations of other papers that were included in this SLR

30 papers (46%) have not cited any other paper selected for the SLR, while 20 papers (31%) have cited only one or two. This relatively small number of cross-references is not due to inadequate literature review by the original authors, but rather a sign of limited incremental work within the "community" of cross-lingual program analysis researchers. The majority of studies have used existing work on single-language program analysis, and extended it to produce cross-language program analysis. Given the origins of cross-lingual program analysis research, this observation is neither disconcerting nor problematic. However, it does highlight limited occurrence of incremental research [121], which in turn, indicates a general immaturity in the domain of cross-lingual program analysis. We hope this SLR raises awareness on the body of relevant literature that the community could build on with further ideas.

Lack of journal papers – We find it noteworthy that only 4 out of 65 studies in this SLR have been published in scientific journals. Although publication channel is not a definite measure for quality, this low number of journal publications corroborates the impression of immaturity of the work in this domain. Extensively-developed and well-evaluated research studies typically do not conform very well with strict page limitations of conferences. A tentative conclusion of this observation is that it is another sign of the need for more involved industrial cases and empirical evaluation.

Common terminology – One immediate observation while conducting this SLR was the significant variation, mismatch, or even contradictory terminology used in the primary studies. For instance, considering the various descriptors for "systems," "artifacts," and "programming models," we listed well over 15 different expressions used to characterize MLSS. The case is not better when it comes to characterizing sub-elements of analysis methods, such as fact extraction, and evaluation processes. This issue hinders translation of the findings among different studies, and increases chances for misinterpretations. While this is not specific to cross-language program analysis (nor to program analysis in general), it again shows that many of the research in this domain is performed largely in isolation.

Dedicated Research Channels – To the best of our knowledge, there is no dedicated workshop or conference aimed at cross-lingual program analysis. Although such a dedicated research channel is not a prerequisite for progress, it can advance the state of research by fostering a more integrated community. Considering the ubiquity of MLSS and the respective maintenance needs, we believe it is inevitable that cross-language program analysis is needed in the near future. This SLR shows that the amount of publications in recent years is already sufficient to initiate a small workshop, or designated session in existing conferences. Moreover, if such initiatives would be combined with collaborative challenges or "bake-offs", they could promote more detailed mutual understanding of approaches, and lead more incremental research efforts. Finally, considering the close conceptual, as well as practical, interactions among the topics of multi-language and cross-language program analysis, we speculate that joint initiatives would pay off for both areas.

6 Limitations to this systematic literature review

Conducting an SLR requires interpretation over the available literature throughout several stages, and the current SLR is no exception to this process. We acknowledge that researcher bias is a major challenge in the course of publication selection, data extraction, and data synthesis. To overcome this challenge, the existing guidelines for literature reviews [81, 86] were extensively examined and discussed to identify the best applicable practices to avoid known pitfalls. To test and refine the research protocol prior to conducting the SLR, , the main study was preceded by a pilot study. This iteration helped us to weed out ambiguity, define stronger criteria and a more systematic process, which all helped to avoid bias during processing of the individual studies for the main SLR. To maintain direction and uniformity of decisions over a relatively long period of time, we documented the research protocol in a technical report prior to conducting the study, and presented the necessary details in the paper to promote transparency and repeatability of the results.
Search strategy – In the pilot study, we started with a set of research questions, and a set of 40 known relevant studies. We used n-gram analysis to extract the most frequently used terms, and combinations of terms, in the relevant papers. The results were used to iteratively develop and adapt our the search query to each of the digital libraries, using the collection of 40 known studies as a regression test. To address the sensitivity of the search engines to minute syntactical issues, we decided to complemented the automated search with manual reference checking of the identified studies (also known as snowballing), and continued the snowballing process until no new studies could be identified (fix-point). Using the combination of these two search methods ensured our confidence in the completeness of our selection.
Study selection and scoping – We defined the target scope of this review in the pilot phase to avoid the risk of scope creep, however, the greatest risk is in judging the relevance of candidate studies with respect to the defined scope. To minimize bias, we adapted a multi-stage selection process, where each stage was subjected to inter-rater agreement screening tests. Throughout the selection process, we took a conservative approach by removing only those studies that were clearly out of the scope for this SLR at that respective stage (i.e., respectively based on title, abstract, and finally on the full text). To limit selection bias as a result of ad-hoc decisions, we required mutual consensus among both authors in the presence of even minor uncertainty about potential removal of a study.

Despite such measures, it is not possible to claim or demonstrate absolute completeness. For one, some studies concerning web applications, in particular, are ambiguous with respect to the level of cross-lingual analysis involved, and their inclusion could be argued either way. In all such cases, we tried to objectively determine their relevance using a full text analysis, and by their ability to contribute possible answers to each of our research questions. Rigidly following this selection process increases the confidence

that we have not excluded particularly major or well-cited studies by mistake.
Data extraction and analysis – Early during the pilot run of the data extraction process, we realized that depending solely on verbatim data as presented in individual primary studies would pose a major threat to the precision of our findings. The lack of common terminology, lack of uniformity in the discussion of technical details, and varying levels of methodological rigor in conducting and reporting the primary studies contribute to this realization. On the other hand, unifying the findings of the primary studies based on our personal interpretations would substantially increase the risk of researcher bias in our overall results.

To address this challenge, we applied open and axial coding techniques from grounded theory to iteratively synthesize a taxonomy of possible answers for each research question. A data extraction table was defined, and structured sets of possible answers were encoded into the table using check-boxes and lists wherever possible. A pilot study with a representative sample of primary studies helped us iteratively define the structured lists for many of the columns, and also revealed for which columns we had to use open-ended coding to maintain precision in the data extraction phase. This process made room for mutual consent by providing an succinct overview toward an iteratively-defined data set, and minimized the aforementioned threats by avoiding pure free text answers, and ad-hoc interpretations during data extraction.

The structured data extraction table proved as a reliable starting point for the data synthesis process as well. Table columns with limited list of possible answers could readily be presented with no need for further analysis and abstraction by us. For open-ended coded columns we postponed the comparison and translation of the findings after all studies and their respective footprint on the taxonomies were settled. Conducting the second round of axial coding at this stage, enabled repeatedly testing and verifying of our interpretations against other co-related studies, and other items in the respective taxonomy, as well as building mutual consensus whenever needed. This played a key role in avoiding individual interpretations.

We believe that the applied approach helped us minimize researcher bias to a large extent, and raised our confidence in the findings. Nevertheless, due to the inherent subjective nature of interpretations that still plays a role when depending on consensus between two researchers, it is difficult to objectively assess (or claim) the overall repeatability of our results – a highly-valued (and to some extent illusive) characteristic of systematic reviews.

7 Conclusion

We identified 75 studies addressing cross-language program analysis, searching through 1767 studies using digital search methods and several hundred manual look-ups while snowballing. Since some of the studies are reported in more than one paper, 10 papers were removed from further analysis. A total of 58 papers were associated with specific analysis technologies, while the other seven contribute insights and propositions. The publication meta-data for the selected studies was analyzed to reveal characteristics of the involved community and the relevant publication channels. Next, information was systematically extracted from the selected studies, based on a data extraction table that was developed and refined using grounded theory, i.e. we iteratively synthesized the extraction taxonomy using open (unrestricted) coding and axial coding based on mutual consensus between the authors. Based on our, possibly subjective, interpretation of the extracted data, we formulated the answers to our initial research questions, and drew a number of conclusions regarding the implications for research, and for the community interested in the topic of cross-language program analysis.

A total of 16 overall goals were identified in the primary studies, among which "comprehension" and "visualizations" were the most frequently targeted, closely followed by "dependency" identification and the analysis of "ripple-effects." Approaches based on static analysis proved to be far more popular than (exclusive) dynamic approaches, with 40% of the studies reaching only semi-automation levels. The majority of studies have adopted the well-known "unified model" pattern to create a homogeneous representation of information derived from heterogeneous artifacts. In recent years, technologies such as model-driven engineering have had a noticeable impact on how approaches are implemented. The type of information that is extracted to conduct cross-language analysis varies greatly among the studies. As could be expected, source code information forms the most frequently included data for cross-language analysis. After source code, there was an equal amount of attention for either database-related artifacts, web pages, or configuration files. Although compiled artifacts and even license agreements play a role in some studies, the most extensively studied cross-language interactions are between Java and C/C++. In addition, there was considerable attention for the analysis of the Java-XML combination and for the Java-SQL combination.

With respect to evaluation methods, the results show a that over one third of the studies had no or only very limited empirical evaluation, more importantly, even though many of the goals are related to human aspects such as comprehension of multi-language systems, there were only very few studies that actually involved human subjects (other than the authors) in the evaluation.

The abundance of studies with no or only proof-of-concept implementations, the non-negligible number of short papers, the low number of journal articles, and the light-weight evaluation conducted in many studies, all convey a sense of immaturity regarding the state of the art in cross-language analysis research. We argue that the limited occurrence of cross-references among the primary studies, in addition to the aforementioned observations, indicates a shortage of incremental and community-driven research and evaluation initiatives (e.g., by means of a 'bake-off' or analysis challenge at a workshop or conference, were participants apply their approach on a common subject system, and compare and possibly integrate their results).

Based on our interpretation of the findings of the SLR, a set of implications for research and the community were discussed. The considerable amount of language- and technology-specific analyses, together with a noticeable lack of (semi-)generic methods are clear signs that generalizablity is a major challenge, as well as a potential key breakthrough for the future of cross-language program analysis. Considering the high pace of technology and language development, research on highly adaptable and generalizable methods might be the only viable approach to close the gap between industry-quality tool support and the growing needs in software maintenance. This need for more generic approaches makes us doubt the trade-off of developing techniques that are highly dependent on heuristics, over investing in sound theoretical frameworks as a basis for future generations of analysis tools. While knowledge repositories have been used in half of the studies, there is little evidence that the state-of-the-art is capable of accommodating our future needs, given the sporadic use of each repository technology. Model-driven technologies, supported by standards and open-source movements, may change this trend, but still have to stand the test of time. We discussed signs that seem to indicate a shortage of incremental and community-driven research in cross-language program analysis, and challenges that rise from a lack of common terminology. Establishing some dedicated research channels can be an effective starting point to overcome some of these challenges, and promote incremental research.

Considering the steady increase in multi-language software systems, we advocate for increased research attention to cross-language analysis of such software systems. It is our goal that the findings of this systematic literature review will benefit those efforts, by providing a starting point for identifying related work in the area, providing a taxonomy that helps to better articulate contributions and research questions, and exposing parts of the research area that could benefit from increased attention.

Appendices

A Concrete Queries

Table 19 shows the concrete queries that we used used during the electronic search. The queries are constructed based on the three sub-expressions E1, E2, and E3, discussed earlier in Table 4. In addition to the table, we provide some complementary information to explain some of the differences in the concrete queries in different search engines.

IEEEXplore: Using the provided interface in the “Command Search” tab of IEEEXplore’s “Advanced Search” page[30] we were able to use a very straightforward query: (E1 AND E2 AND E3). It is possible to execute the query in two modes: “Metadata Only” and “Full Text and Metadata”. Using the latter resulted in

[30]Available at: `http://ieeexplore.ieee.org/search/advsearch.jsp?expression-builder`

Table 19: Concrete queries for the different digital libraries

Search Engine	**Query**
IEEEXplore	E1 AND E2 AND E3
ACM	(Owner:GUIDE) AND ((Abstract: (E1)) OR (Title: (E1)) OR (Keywords: (E1))) AND ((Abstract: (E2 AND E3$'$)) *where* E3$'$: (software OR system OR program OR "source code" OR "application" OR "applications" OR "software development" OR "software engineering" OR "component-based" OR web)
Wiley	create E1$^-$, E2$^-$ and E3$^-$ by removing the hyphens from E1, E2, and E3; insert E1$^-$ AND E2$^-$ AND E3$^-$ for "Abstract" and insert E3$^-$ for "Article Title".
Inspec	create E1$^-$, E2$^-$ and E3$^-$ by removing the hyphens from E1, E2, and E3; insert each sub-expression as a separate "Abstract"-field in the "Multi-Field Search" page and create a conjunction using the AND operator from the user interface.
Web of Sci.	SU=(Computer Science OR Engineering) AND TS=(E1 AND E2 AND E3)
Science. Dir.	TITLE-ABS-KEY(E1 AND E2 AND E3)
Scopus	TITLE-ABS-KEY(E1 AND E2 AND E3)

an order-of-magnitude increase in the number of hits (about 14,000), however, manual inspection of the results clearly indicated a considerable increase in the number of false positives. Therefore, we executed the query against the metadata of papers, which includes, title, abstract and keywords.

ACM: The notion of "metadata" is not readily available in ACM Digital Library search engine in the same way as it is in IEEEXplore. Therefore, we simulate a similar search using the keywords that cover the title, abstract and keywords of the papers. It is crucial to note that ACM offers two databases to search: "Publication from ACM and Affiliated Organizations" and "The ACM Guide to Computing Literature". At the time of this study, the former contained about 400,000 entries and the latter had 2,200,000 entries. Our pilot study convinced us to use the larger collection, as it covered more of the papers in our hand-selected *test set* (see Section 3.1). The term "Owner:GUIDE" in 19 instructs the search engine to do so ("Owner:ACM" is the other option).

Wiley & Inspec: Our tests indicate that it is more reliable to remove the hyphens from the query sub-expressions. Wiley and Inspec do not provide a plain command-based interface, hence, the final query has to be constructed using the logical operands embedded into the web interface.

Science Direct & Scopus: No special remarks, the queries were used as in Table 19.

B Data Extraction Table

Table 20 shows the form that was used for data extraction.

Field	Description	Result Type
Basic		
1. ID	A unique identifier for the study	...
2. Bibliographic data	Title, author, publication year	...
3. Publisher	Original publisher of the study	...
4. Publication type	What type of publication	Journal, conference, workshop
5. Publication channel	Name of journal, conference, or workshop	...
6. Number of pages	Publication size (#pages)	...
Search		
7. Study search source	How was the study retrieved?	IEEE, ACM, Wiley, Inspec, WebSci, SciDirect, Scopus, Snowball
RQ1		
8. General Category	Main approach of the study	Open coding
9. Analysis type	Basic analysis method of artifacts	Static, dynamic (execution trace), dynamic (navigation flow), N/A
10. Knowledge repository	Does the study use a knowledge repository?	Yes / No
RQ2		
11. Capturing technique	Method used for fact extraction	Open coding
RQ3		
12. Information captured	Type of information extracted	Open coding
RQ4		
13. IR format	What intermediate representation	Open coding
RQ5		
14. Goal	The objectives of the study	Open coding
RQ6		
15. Automation level	Does the approach require human interaction?	Fully automated, semi-automated, fully manual, N/A
RQ7		
16. Artifact type	Types of analyzed artifacts	Open coding
17. Language	Programming languages analyzed	
RQ8		
18. Implementation	Tool implementation	Commercial quality, lab. implementation, proof-of-concept, N/A
19. Tool availability	Is the tool publicly available?	Yes / No
20. Evaluation	Presence and extent of evaluation	Controlled experiment, industrial code base, open source system, toy example, none, N/A
21. Limitations		
Reviewer		
22. Reviewer Name	Initial reviewer	AY / LM
23. Needs double check?		Yes/No/Done
24. Other Notes		

Table 20: Data Extraction Table.

Bibliography

[1] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, “Detecting Clones Across Microsoft .NET Programming Languages,” in *Reverse Engineering (WCRE), 2012 19th Working Conf. on.* Knowledge Systems Institute Graduate School, 2012, pp. 405–414.

[2] M. H. Alalfi, J. R. Cordy, and T. R. Dean, “A verification framework for access control in dynamic web applications,” in *Proceedings of the 2009 C3S2E Conf. on - C3S2E ’09.* ACM Press, 2009, p. 109.

[3] ——, “WAFA: Fine-grained dynamic analysis of web applications,” in *2009 11th IEEE Int’l Symposium on Web Systems Evolution.* IEEE, Sep. 2009, pp. 141–150.

[4] D. Amalfitano, A. R. Fasolino, A. Polcaro, and P. Tramontana, “Comprehending Ajax Web Applications by the DynaRIA Tool,” in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh Int’l Conf. on the*, 2010, pp. 122–131.

[5] ——, “DynaRIA: A Tool for Ajax Web Application Comprehension,” in *Program Comprehension (ICPC), 2010 IEEE 18th Int’l Conf. on*, 2010, pp. 46–47.

[6] ——, “The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis,” *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 41–57, Apr. 2013.

[7] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, “TraceBack: first fault diagnosis by reconstruction of distributed control flow,” in *Proceedings of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation*, ser. PLDI ’05. ACM, 2005, pp. 201–212.

[8] D. J. Barrett, A. Kaplan, and J. C. Wileden, “Automated support for seamless interoperability in polylingual software systems,” in *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering - SIGSOFT ’96.* ACM Press, 1996, pp. 147–155.

[9] C. Bellettini, A. Marchetto, and A. Trentini, “WebUml: Reverse Engineering of Web Applications,” in *Proceedings of the 2004 ACM symposium on Applied computing - SAC ’04.* ACM Press, 2004, p. 1662.

[10] ——, “Validation of Reverse Engineered Web Application Models,” in *The Second World Enformatika Conf. (WEC’05)*, ser. Proceedings of World Academy of Science Engineering and Technology, C. Ardil, Ed., vol. 4. WASET, 2005, pp. 125–128.

[11] C. Boldyreff and R. Kewish, “Reverse engineering to achieve maintainable WWW sites,” in *Proceedings Eighth Working Conf. on Reverse Engineering.* IEEE Comput. Soc, 2002, pp. 249–257.

[12] F. Boughanmi, “Multi-Language and Heterogeneously-licensed Software Analysis,” in *Reverse Engineering (WCRE), 2010 17th Working Conf. on*, 2010, pp. 293–296.

[13] M. P. Chase, S. M. Christey, D. R. Harris, and A. S. Yeh, “Managing recovered function and structure of legacy software components,” in *Reverse Engineering, 1998. Proceedings. Fifth Working Conf. on*, 1998, pp. 79–88.

[14] ——, “Recovering software architecture from multiple source code analyses,” in *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Ws. on Program analysis for software tools and engineering - PASTE ’98.* ACM Press, 1998, pp. 43–50.

[15] N. Chen and R. Johnson, “Toward refactoring in a polyglot world: extending automated refactoring support across Java and XML,” in *Proceedings of the 2nd Ws. on Refactoring Tools*, ser. WRT ’08. ACM, 2008, pp. 4:1—-4:4.

[16] B. Cossette and R. J. Walker, “Polylingual Dependency Analysis Using Island Grammars: A Cost Versus Accuracy Evaluation,” in *2007 IEEE Int’l Conf. on Software Maintenance.* IEEE, Oct. 2007, pp. 214–223.

[17] ——, “DSketch: lightweight, adaptable dependency analysis,” in *Proceedings of the eighteenth ACM SIGSOFT Int’l symposium on Foundations of software engineering*, ser. FSE ’10. ACM, 2010, pp. 297–306.

[18] L. Deruelle, M. Bouneffa, N. Melab, and H. Basson, “A change propagation model and platform for multi-database applications,” in *Software Maintenance, 2001. Proceedings. IEEE Int’l Conf. on*, 2001, pp. 42–51.

[19] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson, “Analysis and manipulation of distributed multi-language software code,” in *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE Int’l Ws. on*, 2001, pp. 43–54.

[20] G. Di Lucca, A. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, “WARE: a tool for the reverse engineering of Web applications,” in *Proceedings of the Sixth European Conf. on Software Maintenance and Reengineering.* IEEE Comput. Soc, 2002, pp. 241–250.

[21] G. Di Lucca, A. Fasolino, P. Tramontana, and U. De Carlini, “Abstracting business level UML diagrams from Web applications,” in *Fifth IEEE Int’l Ws. on Web Site Evolution, 2003. Theme: Architecture. Proceedings.* IEEE Comput. Soc, 2003, pp. 12–19.

[22] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, “Reverse engineering web applications: the WARE approach,” *Journal of Software Maintenance and Evolution*, vol. 16, no. 1-2, pp. 71–101, Jan. 2004.

[23] M. Druk and M. Kropp, “ReFit: A Fit test maintenance plug-in for the Eclipse refactoring plug-in,” in *Developing Tools as Plug-ins (TOPI), 2013 3rd Int’l Ws. on*, May 2013, pp. 7–12.

[24] M. Furr and J. S. Foster, “Checking type safety of foreign function calls,” in *Proceedings of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI’05.* ACM, 2005, pp. 62–72.

[25] ——, “Polymorphic Type Inference for the JNI,” in *Proceedings of the 15th European Conf. on Programming Languages and Systems (ESOP’06)*, vol. 3924. Springer-Verlag, 2006, pp. 309–324.

[26] A. E. Hassan and R. Holt, “Towards a better understanding of Web applications,” in *Proceedings 3rd Int’l Ws. on Web Site Evolution. WSE 2001.* IEEE Comput. Soc, 2001, pp. 112–116.

[27] A. E. Hassan and R. C. Holt, “Architecture recovery of web applications,” in *Proceedings of the 24th Int’l Conf. on Software Engineering - ICSE ’02.* ACM Press, 2002, p. 349.

[28] ——, “A Visual Architectural Approach to Maintaining Web Applications,” *Annals of Software Engineering- Special Volume on Software Visualization*, vol. 16, 2003.

[29] J. Hayes, W. G. Griswold, and S. Moskovics, “Component design of retargetable program analysis tools that reuse intermediate representations,” in *Proceedings of the 22nd Int’l Conf. on Software Engineering*, ser. ICSE ’00. ACM, 2000, pp. 356–365.

[30] A. Hessellund and P. Sestoft, “Flow Analysis of Code Customizations,” in *The 22nd European Conf. on Object-Oriented Programming (ECOOP 2008).* Springer Berlin Heidelberg, 2008, pp. 285–308.

[31] C.-L. Hsu, H.-C. Liao, J.-L. Chen, and F.-J. Wang, “A Web database application model for software maintenance,” in *Autonomous Decentralized Systems, 1999. Integration of Heterogeneous Systems. Proceedings. The Fourth Int’l Symposium on*, 1999, pp. 338–344.

[32] M. Kamp, “Managing a multi-file, multi-language software repository for program comprehension tools: a generic approach,” in *Program Comprehension, 1998. IWPC ’98. Proceedings., 6th Int’l Ws. on*, 1998, pp. 64–71.

[33] M. Kempf, R. Kleeb, M. Klenk, and P. Sommerlad, “Cross language refactoring for Eclipse plug-ins,” in *Proceedings of the 2nd Ws. on Refactoring Tools - WRT ’08.* ACM Press, 2008, pp. 1–4.

[34] H. Kienle and H. Muller, “Leveraging program analysis for Web site reverse engineering,” in *Proceedings 3rd Int’l Ws. on Web Site Evolution. WSE 2001.* IEEE Comput. Soc, 2001, pp. 117–125.

[35] U. Kolsch, “Object-oriented re-engineering of information systems in a heterogeneous distributed environment,” in *Reverse Engineering, 1998. Proceedings. Fifth Working Conf. on*, 1998, pp. 104–114.

[36] N. A. Kraft, B. W. Bonds, and R. K. Smith, “Cross-Language Clone Detection,” in *the 20th Int’l Conf. on Software Engineering and Knowledge Engineering (SEKEâĂŹ08)*, 2008, pp. 54–59.

[37] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, “Program comprehension in multi-language systems,” in *Proceedings Fifth Working Conf. on Reverse Engineering (Cat. No.98TB100261)*, ser. WCRE ’98. IEEE Comput. Soc, 1998, pp. 135–143.

[38] C. Lange, H. M. Sneed, and A. Winter, “Comparing graph-based program comprehension tools to relational database-based tools,” in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th Int’l Ws. on*, 2001, pp. 209–218.

[39] S. Lehnert, Q. Farooq, and M. Riebisch, “Rule-Based Impact Analysis for Heterogeneous Software Artifacts,” in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conf. on*, Mar. 2013, pp. 209–218.

[40] P. K. Linos, “PolyCARE: a tool for re-engineering multi-language program integrations,” in *Engineering of Complex Computer Systems, 1995. Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP, Proceedings., First IEEE Int’l Conf. on.* IEEE Computer Society, 1995, pp. 338–341.

[41] P. K. Linos, Z.-h. Chen, S. Berrier, and B. O’Rourke, “A Tool For Understanding Multi-Language Program Dependencies,” in *Proceedings of the 11th IEEE Int’l Ws. on Program Comprehension*, ser. IWPC ’03. IEEE Computer Society, 2003, pp. 64–73.

[42] P. Linos, W. Lucas, S. Myers, and E. Maier, “A metrics tool for multi-language software,” in *Proceedings of the 11th IASTED Int’l Conf. on Software Engineering and Applications*, ser. SEA ’07. ACTA Press, 2007, pp. 324–329.

[43] C. Marinescu, “Identification of Relational Discrepancies between Database Schemas and Source-Code in Enterprise Applications,” in *Ninth Int’l Symposium on Symbolic*

and Numeric Algorithms for Scientific Computing (SYNASC 2007). IEEE, Sep. 2007, pp. 93–100.

[44] P. Mayer and A. Schroeder, "Cross-Language Code Analysis and Refactoring," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th Int'l Working Conf. on*, 2012, pp. 94–103.

[45] ——, "Patterns of cross-language linking in java frameworks," in *Program Comprehension (ICPC), 2013 IEEE 21st Int'l Conf. on*, May 2013, pp. 113–122.

[46] D. L. Moise and K. Wong, "Extracting and Representing Cross-Language Dependencies in Diverse Software Systems," in *Proceedings of the 12th Working Conf. on Reverse Engineering*, ser. WCRE '05. IEEE Computer Society, 2005, pp. 209–218.

[47] D. L. Moise, K. Wong, H. J. Hoover, and D. Hou, "Reverse Engineering Scripting Language Extensions," in *Proceedings of the 14th IEEE Int'l Conf. on Program Comprehension*, ser. ICPC '06. IEEE Computer Society, 2006, pp. 295–306.

[48] L. Moonen, "Towards evidence-based recommendations to guide the evolution of component-based product families," *Science of Computer Programming*, vol. 97, pp. 105–112, 2013.

[49] ——, "A generic architecture for data flow analysis to support reverse engineering," in *Proceedings of the 2nd Int'l Conf. on Theory and Practice of Algebraic Specifications*, ser. Algebraic'97. British Computer Society, 1997, p. 10.

[50] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and Implementation of a Hybrid Parallel Performance Measurement System," in *2010 39th Int'l Conf. on Parallel Processing.* IEEE, Sep. 2010, pp. 492–501.

[51] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "BabelRef: Detection and renaming tool for cross-language program entities in dynamic web applications," in *Software Engineering (ICSE), 2012 34th Int'l Conf. on*, 2012, pp. 1391–1394.

[52] F. Perin, "Enabling the Evolution of J2EE Applications through Reverse Engineering and Quality Assurance," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conf. on*, 2009, pp. 291–294.

[53] F. Perin, T. Girba, and O. Nierstrasz, "Recovery and analysis of transaction scope from scattered information in Java Enterprise Applications," in *Software Maintenance (ICSM), 2010 IEEE Int'l Conf. on*, 2010, pp. 1–10.

[54] R.-H. Pfeiffer and A. Wasowski, "Taming the confusion of languages," in *Proceedings of the 7th European Conf. on Modelling foundations and applications*, ser. ECMFA'11. Springer-Verlag, 2011, pp. 312–328.

[55] R.-H. Pfeiffer and A. Wakasowski, "TexMo: a multi-language development environment," in *Proceedings of the 8th European Conf. on Modelling Foundations and Applications*, ser. ECMFA'12. Springer-Verlag, 2012, pp. 178–193.

[56] R.-H. Pfeiffer and A. Wasowski, "Cross-language support mechanisms significantly aid software development," in *Proceedings of the 15th Int'l Conf. on Model Driven Engineering Languages and Systems*, ser. MODELS'12. Springer-Verlag, 2012, pp. 168–184.

[57] T. Polychniatis, J. Hage, S. Jansen, E. Bouwers, and J. Visser, "Detecting Cross-Language

Dependencies Generically," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conf. on*, Mar. 2013, pp. 349–352.

[58] F. Ricca and P. Tonella, "Building a Tool for the Analysis and Testing of Web Applications: Problems and Solutions," in *The 7th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001.* Springer-Verlag, 2001, pp. 373—-388.

[59] M. Salah and S. Mancoridis, "Toward an environment for comprehending distributed systems," in *10th Working Conf. on Reverse Engineering, Proceedings.* Reengn Forum; IEEE Comp Soc, Tech Council Sofware Engn; Univ Victoria; Univ Alberta, 2003, pp. 238–247.

[60] H. Schink, "sql-schema-comparer: Support of multi-language refactoring with relational databases," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th Int'l Working Conf. on*, Sep. 2013, pp. 173–178.

[61] H. Schink, M. Kuhlemann, G. Saake, and L. Ralf, "Hurdles in multi-language refactoring of hibernate applications," in *Proceedings of the 6th Int'l Conf. on Software and Database Technologies.* SciTePress - Science and and Technology Publications, 2011, pp. 129–134.

[62] S. Sidler, S. Reinhard, and P. Sommerlad, "Cross Language Refactoring for Groovy and Java in Eclipse," in *3rd wprkshop on refactoring tools (WRT'09)*, 2009, pp. 1–2.

[63] D. Strein, H. Kratz, and W. Lowe, "Cross-Language Program Analysis and Refactoring," in *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE Int'l Ws. on*, 2006, pp. 207–216.

[64] N. Synytskyy, J. R. Cordy, and T. Dean, "Resolution of static clones in dynamic Web pages," in *Web Site Evolution, 2003. Theme: Architecture. Proceedings. Fifth IEEE Int'l Ws. on*, 2003, pp. 49–56.

[65] N. Synytskyy, J. R. Cordy, and T. R. Dean, "Robust multilingual parsing using island grammars," in *Proceedings of the 2003 Conf. of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '03. IBM Press, 2003, pp. 266–278.

[66] F. Tomassetti, M. Torchiano, and A. Vetro, "Classification of Language Interactions," in *2013 ACM / IEEE Int'l Symposium on Empirical Software Engineering and Measurement.* IEEE, Oct. 2013, pp. 287–290.

[67] F. Tomassetti, A. Vetro, M. Torchiano, M. Voelter, and B. Kolb, "A model-based approach to language integration," in *Modeling in Software Engineering (MiSE), 2013 5th Int'l Ws. on*, May 2013, pp. 76–81.

[68] F. Tomassetti, G. Rizzo, and M. Torchiano, "Spotting automatically cross-language relations," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conf. on*, Feb. 2014, pp. 338–342.

[69] P. Tramontana, "Reverse engineering Web applications," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE Int'l Conf. on.* IEEE Computer Society, 2005, pp. 705–708.

[70] A. Vetro', F. Tomassetti, M. Torchiano, and M. Morisio, "Language interaction and quality issues," in *Proceedings of the ACM-IEEE Int'l symposium on Empirical software engineering and measurement - ESEM '12.* ACM Press, 2012, p. 319.

[71] S. Weijun, L. Shixian, and L. Xianming, "An Approach for Reverse Engineering of Web

Applications," in *2008 Int'l Symposium on Information Science and Engineering.* IEEE, Dec. 2008, pp. 98–102.

[72] A. R. Yazdanshenas and L. Moonen, "Crossing the boundaries while analyzing heterogeneous component-based software systems," in *Proceedings of the 2011 27th IEEE Int'l Conf. on Software Maintenance*, ser. ICSM '11. IEEE Computer Society, 2011, pp. 193–202.

[73] ——, "Tracking and visualizing information flow in component-based systems," in *2012 20th IEEE Int'l Conf. on Program Comprehension (ICPC).* IEEE, Jun. 2012, pp. 143–152.

[74] ——, "Fine-grained change impact analysis for component-based product families," in *2012 28th IEEE Int'l Conf. on Software Maintenance (ICSM)*, no. 5. IEEE, Sep. 2012, pp. 119–128.

[75] X. Zheng and M.-H. Chen, "Maintaining Multi-Tier Web Applications," in *Software Maintenance, 2007. ICSM 2007. IEEE Int'l Conf. on*, 2007, pp. 355–364.

[76] C. Jones, *Estimating Software Costs : Bringing Realism to Estimating: Bringing Realism to Estimating*, 2nd ed., ser. McGraw-Hill's AccessEngineering. Mcgraw-hill, 2007.

[77] R.-H. Pfeiffer and A. Wasowski, "Cross-language support mechanisms significantly aid software development," in *Proceedings of the 15th Int'l Conf. on Model Driven Engineering Languages and Systems*, ser. MODELS'12. Springer-Verlag, 2012, pp. 168–184.

[78] M. Fowler, *Domain-Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[79] P. Mayer and A. Schroeder, "Cross-Language Code Analysis and Refactoring," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th Int'l Working Conf. on*, 2012, pp. 94–103.

[80] D. Binkley, "Source Code Analysis: A Road Map," in *Future of Software Engineering (FoSE).* IEEE, May 2007, pp. 104–119.

[81] B. A. Kitchenham, "Guidelines for performing Systematic Literature Reviews in Software Engineering," Keel University & University of Durham, Tech. Rep., 2007.

[82] B. A. Kitchenham, T. Dybå, and M. Jorgensen, "Evidence-based Software Engineering," in *Int'l Conf. on Software Engineering (ICSE)*, 2004, pp. 273–281.

[83] M. Harman, "Why Source Code Analysis and Manipulation Will Always be Important," in *2010 10th IEEE Working Conf. on Source Code Analysis and Manipulation.* IEEE, Sep. 2010, pp. 7–19.

[84] T. Rout, "Consistency and conflict in terminology in software engineering standards," in *Proceedings 4th IEEE Int'l Software Engineering Standards Symposium and Forum (ISESS'99). 'Best Software Practices for the Internet Age'.* IEEE Comput. Soc, 1998, pp. 67–74.

[85] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 5, pp. 684–702, Sep. 2009.

[86] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain." *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, 2007.

[87] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, Aug. 2002.

[88] T. Hendrix, J. Cross, L. Barowski, and K. Mathias, "Tool support for reverse engineering multi-lingual software," in *Proceedings of the Fourth Working Conf. on Reverse Engineering*. IEEE Comput. Soc, 1997, pp. 136–143.

[89] F. Chow, "Intermediate representation," *Communications of the ACM*, vol. 56, no. 12, pp. 57–62, Dec. 2013.

[90] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *2011 IEEE 11th Int'l Working Conf. on Source Code Analysis and Manipulation*. IEEE, Sep. 2011, pp. 173–184.

[91] D. Jin, J. R. Cordy, and T. R. Dean, "Where is the Schema? A Taxonomy of Patterns for Software Exchange," in *Int'l Ws. on Program Comprehension (IWPC' 2002)*, 2002, pp. 65–74.

[92] J. Hayes, W. G. Griswold, and S. Moskovics, "Component design of retargetable program analysis tools that reuse intermediate representations," in *Proceedings of the 22nd Int'l Conf. on Software Engineering*, ser. ICSE '00. ACM, 2000, pp. 356–365.

[93] A. R. Yazdanshenas and L. Moonen, "Crossing the boundaries while analyzing heterogeneous component-based software systems," *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 193–202, 2011.

[94] H. Lochmann and A. Hessellund, "An integrated view on modeling with multiple domain-specific languages," in *The IASTED Int'l Conf. Software Engineering (SE 2009)*, 2009, pp. 1–10.

[95] J. Oberleitner, F. Rosenberg, and S. Dustdar, "A lightweight model-driven orchestration engine for e-services," in *Proceedings of the 6th Int'l Conf. on Technologies for E-Services*, ser. TES'05. Springer-Verlag, 2006, pp. 48–57.

[96] F. Ricca, P. Tonella, E. Pianta, and C. Girardi, "Experimental results on the alignment of multilingual Web sites," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conf. on*, 2004, pp. 288–295.

[97] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, no. 6, pp. 139–143, Jun. 2010.

[98] R. Rahimi and R. Khosravi, "Architecture conformance checking of multi-language applications," in *Proceedings of the ACS/IEEE Int'l Conf. on Computer Systems and Applications - AICCSA 2010*, ser. AICCSA '10. IEEE Computer Society, 2010, pp. 1–8.

[99] C. Ackermann, M. Lindvall, and R. Cleaveland, "Recovering Views of Inter-System Interaction Behaviors," in *Working Conf. on Reverse Engineering (WCRE)*. IEEE, Oct. 2009, pp. 53–61.

[100] I. Å𝑑ora, "A meta-model for representing language-independent primary dependency structures," in *ENASE 2012 - Proceedings of the 7th Int'l Conf. on Evaluation of Novel Approaches to Software Engineering*, 2012, pp. 65–74.

[101] A. Puder, "An XML-based cross-language framework," *Lecture Notes in Computer*

Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 3762 LNCS, pp. 20–21, 2005.

[102] N. Asif, F. Shahzad, N. Saher, and W. Nazar, "Clustering the source code," *WSEAS Transactions on Computers*, vol. 8, no. 12, pp. 1835–1844, 2009.

[103] D. Strein, R. Lincke, J. Lundberg, and W. Löwe, "An Extensible Meta-Model for Program Analysis," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 592–607, Sep. 2007.

[104] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski, "Behind the Scenes in SANTE: A Combination of Static and Dynamic Analyses," *Automated Software Engg.*, vol. 21, no. 1, pp. 107–143, Mar. 2014.

[105] H. Kim, K. G. Doh, and D. A. Schmidt, "Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7935 LNCS. Springer, 2013, pp. 194–214.

[106] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, pp. 14–es, Sep. 2007.

[107] B. Kitchenham and A. Burn, "Validating Search Processes in Systematic Literature Reviews," in *The 1st Int' Ws. on Evidential Assessment of Software Technologies, In conjunction with ENASE 2011.* SciTePress, 2011, pp. 3–9.

[108] T. MAREW, J. KIM, and D. H. BAE, "SYSTEMATIC FUNCTIONAL DECOMPOSITION IN A PRODUCT LINE USING ASPECT-ORIENTED SOFTWARE DEVELOPMENT: A CASE STUDY," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 01, pp. 33–55, 2007.

[109] B. Kitchenham, P. Brereton, M. Turner, M. Niazi, S. Linkman, R. Pretorius, and D. Budgen, "The impact of limited search procedures for systematic literature reviews âĂŤ A participant-observer case study," in *2009 3rd Int'l Symposium on Empirical Software Engineering and Measurement.* IEEE, Oct. 2009, pp. 336–345.

[110] T. Greenhalgh and R. Peacock, "Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources." *BMJ (Clinical research ed.)*, vol. 331, no. 7524, pp. 1064–5, Nov. 2005.

[111] F. Boughanmi, "Change Impact analysis of Multi-Language and Heterogeneously-licensed Software," Ph.D. dissertation, École Polytechnique de Montréal, 2010.

[112] F. Perin, "Reverse Engineering Heterogeneous Applications," Ph.D. dissertation, University of Berne, 2012.

[113] J. Cohen, "Weighted kappa: nominal scale agreement with provision for scaled disagreement or partial credit." *Psychological bulletin*, vol. 70, no. 4, pp. 213–220, 1968.

[114] E. Mendes, "A systematic review of Web engineering research," in *2005 Int'l Symposium on Empirical Software Engineering, 2005.* IEEE, 2005, pp. 481–490.

[115] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* SAGE, 1998.

[116] G. W. Noblit and R. D. Hare, *Meta-Ethnography: Synthesizing Qualitative Studies.* Sage Publications, 1988.

[117] N. Britten, R. Campbell, C. Pope, J. Donovan, M. Morgan, and R. Pill, “Using meta ethnography to synthesise qualitative research: a worked example.” *Journal of health services research & policy*, vol. 7, no. 4, pp. 209–15, Oct. 2002.

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

[119] C. Seaman, “Qualitative methods in empirical studies of software engineering,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[120] S. Carpendale, “Evaluating information visualizations,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4950 LNCS, 2008, pp. 19–45.

[121] B. Meyer, “Incremental research vs. paradigm-shift mania,” *Communications of the ACM*, vol. 55, no. 9, p. 8, Sep. 2012.