2011 27th IEEE International Conference on Software Maintenance (ICSM)

Crossing the Boundaries while Analyzing Heterogeneous Component-Based Software Systems

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

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

***Abstract*—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**

I. 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.

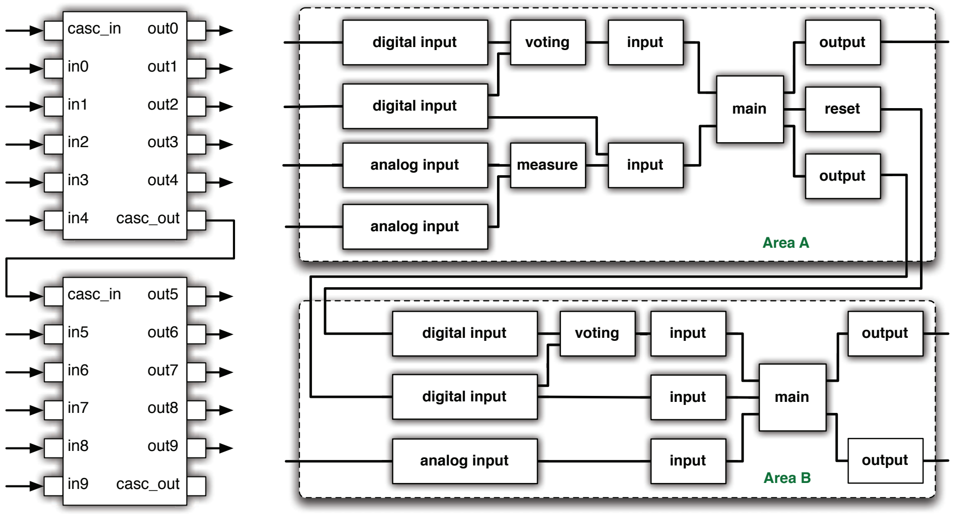
Analyzing the properties of programs *within* closed code boundaries is a well-established area that has been studied for some decades [1], and techniques have successfully been implemented in professional program analysis tools [2], [3]. However, most of these tools have strict limitations on the programming 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 II describes the background of this study. We describe our approach in Section III, and report on our prototype implementation in Section IV. We evaluate our approach and prototype implementation in Section V, discuss the related work in Section VI, and conclude in Section VII.

II. 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



(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

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.

III. Approach

A. 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

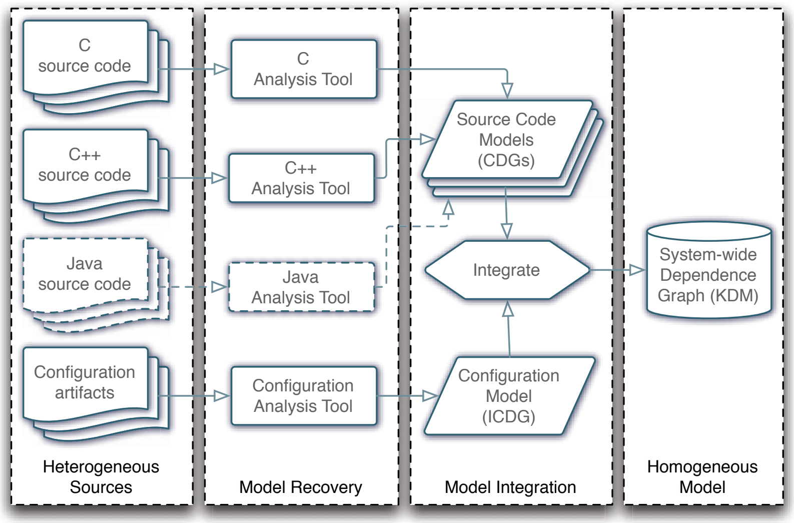


Figure 2. Integrating models derived from heterogeneous sources into a homogeneous model

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.

B. 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 IV):

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 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

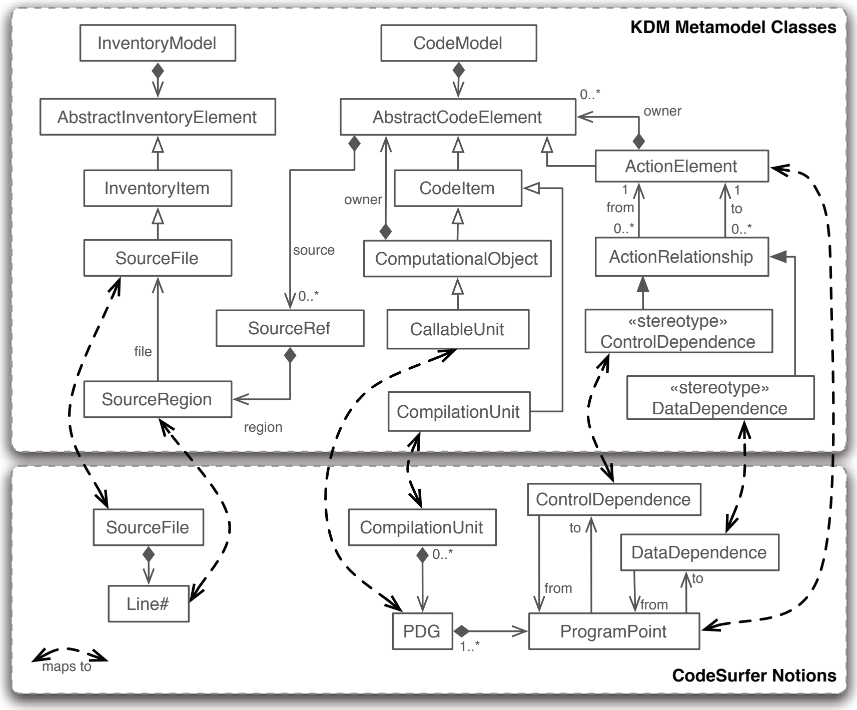


Figure 3. Target KDM metamodel classes and their mapping to CodeSurfer constructs

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.

IV. Prototype Implementation

In this section we discuss a prototype implementation of the approach that was sketched in Section III. 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.

A. 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.

B. 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

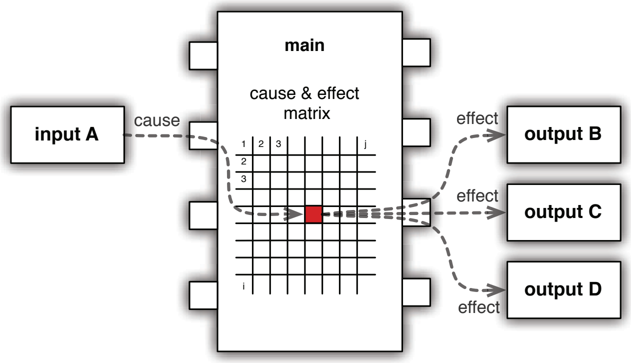


Figure 4. Cause&Effect matrix: a domain-specific inter-component communication mechanism based on shared memory.

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 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 ensuring that a cause could trigger multiple effects. The cells to which the

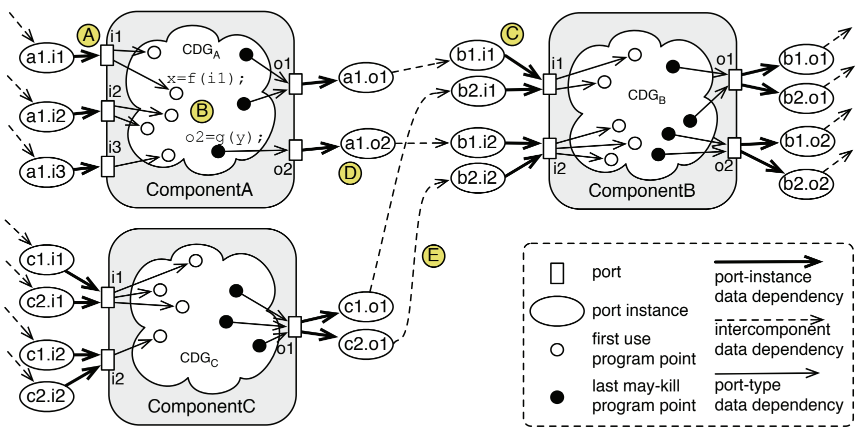


Figure 5. Assembling the SDG from the individual CDGs and the ICDG (note that markers A–E are explained in the text).

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.

C. 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).

D. 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.

V. 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.

A. 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 II. 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 I.

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 IV-A&B and slice it for a given set of slicing criteria using the tool described in Section IV-D.

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.

B. 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 I. 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.

Table I
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

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 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.

C. 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

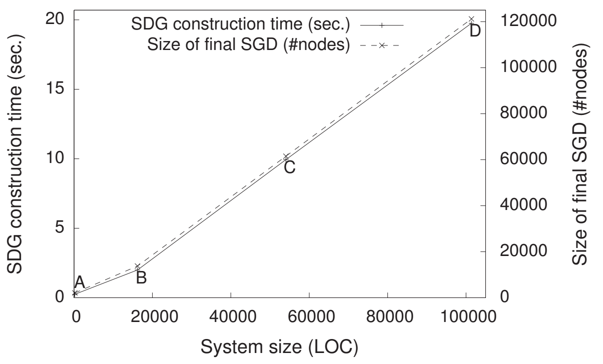


Figure 6. Transformation time and SDG size vs. system size.

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 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.

VI. 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], [11], [12], [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.

VII. 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.

REFERENCES

[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 *8th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation*. IEEE, 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 *16th Working Conf. on Reverse Eng.*, 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 *25th 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 *2nd IEEE Int'l Ws. on Source Code Analysis and Manipulation*, 2002.

[18] J. Zhao, "A slicing-based approach to extracting reusable software architectures," in *4th 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 *Proceedings of the 30th Int'l Conf. on Software Engineering*. ACM, 2008, pp. 391–400.