KADRE: Domain-Specific Architectural Recovery for Scientific Software Systems

David Woollard[†][◇] Chris A. Mattmann[†][◇] Daniel Popescu[◇] Nenad Medvidovic[◇]

[†]Jet Propulsion Laboratory
California Inst. of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
{woollard,mattmann}@jpl.nasa.gov

[◇]Computer Science Dept.
Univ. of Southern California
Los Angeles, CA 90089, USA
{dpopescu,neno}@usc.edu

ABSTRACT

Scientists today conduct new research via software-based experimentation and validation in a host of disciplines. Scientific software represents a significant investment due to its complexity and longevity yet there is little reuse of scientific software beyond small libraries which increases development and maintenance costs. To alleviate this disconnect, we have developed KADRE, a domain-specific architecture recovery approach and toolset to aid automatic and accurate identification of workflow components in existing scientific software. KADRE improves upon state of the art general cluster techniques, helping to promote component-based reuse within the domain.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

Keywords

Scientific Computing, Workflows, Software Architecture

1. INTRODUCTION

Software plays a vital role today in conducting science, allowing experimentation in everything from climate science to drug discovery. The complex scientific software systems that facilitate this type of *in silico* experimentation represent a significant investment in time and maintenance. For example, Earth science mission software at NASA's Jet Propulsion Laboratory must survive years and sometimes decades of operations, outlasting not only technology but also the software developers who engendered it.

Because of its longevity and its complexity, much effort goes into the maintenance of scientific software. This software must be updated over time to deal with machine failures, outdated software and changes to software/hardware support, as well as to take advantage of new technologies (where possible) to reduce operations and maintenance costs, improve performance, etc.

There are a number of challenges to scientific software maintenance not the least of which is the propensity for scientific codes to have been designed and written by domain experts; experts who are not software engineers by training and therefore have built largely monolithic systems. Additionally, this software is often very complex and is only well understood by developers and small user base [18]. Often these two groups are the same people, leading to the end-user developer problem [13].

The codification of the scientific processes and experiments in the formalization of workflow-based composition [22] offers the scientist a promising advancement in *in silico* development. In this paradigm a scientist builds a workflow model of the software system, reifying the experiment in flexible model rather than embedding it in the control flow of a monolithic software system. Often though, the process of translating the workflow into reusable code is a human-centric, time-costly process. However, if the process was automated it would enable software modification and evolution at the level a scientist desires, decoupling the scientist's expertise from the requirement of having to become a software expert.

A key insight in the course of our study into scientific software [19] is that scientific codes often follow an implicit dataflow architecture in which the components reify steps in the scientific process. In this paper, we exploit that insight and present KADRE (Kernel Analysis, Decomposition, and Re-Engineering), a novel technique for automated discovery of workflow components from monolithic scientific codes. KADRE is a domain specific software architectural recovery tool that uses clustering techniques to aid the scientist in automatically and accurately locating components in scientific software.

In the next section, we will discuss related work including software architectural recovery techniques, specifically focusing on software clustering techniques as a means of identifying software components. In Section 3, we introduce software kernels in greater detail, and describe how we recover kernels. In Section 4, we present our evaluation of KADRE's performance and we conclude with future work.

2. BACKGROUND AND RELATED WORK

Software architecture captures the principal design decisions behind a software system and is often represented in terms of components (units of computation), connectors (interactions amongst components) and configurations (arrangements of components and connectors and the rules that guide their composition) [15]. A domain-specific software architecture is a codification of domain knowledge in a software architecture that is applicable to software systems developed for a given application domain [16]. Hayes-Roth et. al., extended this definition to include other reusable artifacts including reference components capturing domain knowledge and configuration methods for selecting the components and for configuring the reference architecture [6]. Software architectural recovery is the process of elucidating a software system's architecture from its source code, available documentation and other artifacts (e.g., runtime behavior) [3, 11].

We will define *domain-specific architectural recovery* to be an architectural recovery process that utilizes domain knowledge to extract system architectures compliant to a domain-specific reference architecture. Our technique, KADRE, identifies a specific type of component: a workflow component which is functional in nature (free of side-effects) and represents a single step in the scientific process (a single domain concept), which we call a *kernel*.

Several automatic approaches to general architectural recovery exist, most of which result in partial architectural descriptions [3]. These descriptions utilize feature extraction from both static analysis and runtime analysis, and agglomeration techniques such as clustering [9]. Similarity-based software clustering techniques provide a means of grouping related program entities (procedures/methods, source code, variables, and other programming language level constructs) based on their relationship to one another. Siliarity metrics have included latent semantic analysis of source code [8], software evolution and change [1], and features gathered from static and dynamic analysis of the code [4, 7]. In [9], the authors posit that the quality of clustering results depends heavily on the characteristics of the software to be clustered, suggesting that constraining KADRE's clustering process to a specific software domain will improve quality.

Several domain-specific recovery methodologies [5, 10] have been proposed that utilize human interpretation to cluster source code into more highly-refined architectural models [11, 12]. These methodologies recover domain-specific architectures utilizing *domain knowledge* such as source code, domain specific terms, system documentation, and an architects' experience.

In the next section, we will present our approach for domain-specific architectural recovery and show how domain knowledge can be leveraged to create component decompositions of existing monolithic software systems meaningful to the domain of computational science.

3. APPROACH

Based on our experience refactoring scientific software used at JPL, we have seen that recovering a dataflow architecture from an existing monolithic scientific software system is a task in which software components are identified and used as stages in a workflow process; in turn, the workflow process implements the actual scientific experiment being attempted by the software's authors. We call these workflow components *scientific kernels*. A scientific kernel is a snippet of source code that implements a single step in the scientific process being reified in software. A kernel is similar to a filter in dataflow architectures [15]: it is stateless and exchanges all necessary data as part of the transfer of control flow.

Existing scientific software systems are written in a variety of languages (Fortran, C/C++, and increasingly Java) and also contain varying degrees of encapsulation (subsystems, modules, classes, objects, functions, etc.) that form *elements* of the software system. During past manual decompositions of different scientific software systems into kernels, we have repeatedly utilized a number of sources of information about these elements, in addition to our conceptual understanding of the scientific process being implemented. These sources of information include:

- Proximity between elements in source code (i.e., Are two functions in the same class or same subsystem?).
- Call distance between elements (i.e., Are two functions called by the same parent function? Are they executed far from one-another temporally?).
- Data dependencies between elements (i.e., Do two functions share a large amount of data? Do they manipulate completely separate resources?)

These experiences and observations have led us to the hypothesis that underlies this work: Automatically agglomerating low-level software elements, namely functions, into clusters that are meaningful scientific kernels is possible if (1) *a clustering process is used that incorporates these forms of information about the elements*, and (2) *the clustering process is further tailored to the domain of scientific software by use of an appropriately representative training set of sample decompositions.*

3.1 Clustering Process

We have codified our clustering process in a tool we call KADRE. As with organizational terminology, in which a *cadre* is a group of key personnel or entities in an organization that form its core, KADRE is a tool that aids the scientist in the decomposition of monolithic code into a workflow-based system by automatically identifying scientific kernels. In order to identify these kernels, KADRE uses an affinity clustering algorithm that implements a clustering technique, originally inspired by the manual approach we have used over time to decompose scientific software systems. Elements of the program, in this case functions, are iteratively combined until the resulting clusters exhibit maximum internal cohesion and are minimally similar to one-another.

Like other clustering techniques, we employ a similarity metric in order to measure the "distance" between code elements. The similarity of clusters is measured using full linkage chaining – the similarity of cluster a and cluster b is taken as the minimum pairwise similarity of all elements in a and all elements in b. The clustering process terminates when the similarity between all clusters is below a threshold value that is a tunable parameter. As with the manual process, similarity in our clustering algorithm is affected by three sources of information: (1) proximity, (2) call distance, and (3) data dependency, each of which is explored notionally below. For a formal definition of our similarity measure, see [21].

Proximity

The code-level proximity between any two software elements is best understood in terms of the hierarchical composition of the elements in the program as a tree. For an object-oriented language like Java, the base node in the tree represents the program, its children the modules (in Java, these are packages), the modules' children the classes, the classes' children the functions (or methods, in the case of Java), and the functions' children the lines of code.

If the two elements contained in program P have the same parent, they are highly proximate to one another. On the other hand, if they are not contained in the same 2-level subtree of the containment tree (i.e., their parents' parents are not identical), they are considered unrelated. For example, if two functions do not exist in the same module, they are considered proximally unrelated.

Call Relationship

The second measure affecting similarity between two program elements to be considered is the call relationship, or $call(a, b)$. Notionally, the call relationship between two program elements is the elements' relationship in the dynamic execution of the program. We measure $call(a, b)$ as the normalized distance between nodes representing the program elements in the undirected version of the program's call graph. Specifically, we can determine if an element a is similar to an element b in terms of call relationships by calculating the minimal path in the undirected call graph, call it P', between the nodes a and b. This measure is normalized by dividing by the longest path in P'. Since this is a distance measure, we take the complement to get a similarity metric.

Data Dependency

The final measure that affects the similarity metric of two elements, a and b, is $data(a, b)$, or the data dependencies of the two elements. If two program elements share a large amount of data, then they are good candidates for clustering. Our measure of $data(a, b)$ is similar to Girard, et. al.'s indirect relationship metric in [4], although we analyze this relationship using a resource tree of data dependencies rather than a flow graph in order to eliminate transitive data dependencies. This distinction is important so as to not indirectly influence the $data(a, b)$ feature with call structure (since call structure is already accounted for in $Sim(a, b)$ via the parameter $call(a, b)$).

When measuring data dependencies, it is important to consider that not all shared resources should be weighted equally. Two functions that both manipulate a large array of variables should be judged more similar for clustering purposes than two functions that exchange a single variable, for example. Likewise, if a resource is shared by all program elements, then it should not be used as rationale for clustering any two elements over all possible clusterings. For each identified data dependency, we apply a weighting parameter given by multiplying the size (memory footprint) of the shared resource normalized to overall program footprint by its Shannon information content [14] – a measure of the resource's frequency of use.

3.2 Parameter Tuning

In order to train KADRE to produce clusters that are meaningful scientific kernels, we need to weight each of the features as well as the threshold parameter. We chose to train KADRE using supervised learning on a set of representative scientific applications. To accomplish this learning process, we require not only a representative set of scientific programs with which to train, but also a measure of the closeness, or quality, of the produced clusterings.

3.2.1 Developing an Training Suite

In order to train KADRE to accurately cluster program elements into scientific kernels, we have developed a suite of representative scientific software systems. We chose seven systems from the Java Grande Benchmarking Suite [2] to build our suite: **LUD** – a lower-upper decomposition program, **Crypt** – a cryptography application, **Sparse** – a sparse matrix multiplication program, **FFT** – a fast fourier transform program, **Euler** – a program that simulates fluid dynamics in a channel, **MD** – a molecular dynamics simulation, and **Search** – an A.I. application that uses alpha-beta pruning to explore a complex space.

For each of the seven evaluation systems, we have clustered the system manually in order to generate a variant of the given software system that is decomposed into scientific kernels. In addition to our own decompositions, we conducted a study in which we taught twenty-three teams of computer science graduate students (eighty-two students in all), to decompose scientific applications. The students were enrolled in an advanced graduate-level software architecture course and were given extensive guidance in order to apply our manual decomposition strategy. Three to four teams decomposed each system in the evaluation suite; these decompositions were combined by the authors to create a "best of breed" decomposition – a decomposition in which the software clusters are true representations of the scientific steps being conducted.

3.2.2 Measuring Cluster Quality

For KADRE to discover not just software clusters but true *scientific kernels*, we trained KADRE to produce clusterings of necessary quality. In order to judge quality, we must measure the distance between the automatically determined clustering and an expert decomposition of the software system into scientific kernels. Notionally, we can measure the distance between two clusterings as the effort required to transform one clustering to another. In [17], Tzerpos and Holt defined the **MoJo** metric as the number of **Mo**ve and **Jo**in operations required to convert one set of clusters to another set. We have leveraged this metric to judge the distance between KADRE's clusters and expert decompositions of the systems detailed above, and therefore the quality of the clustering produced by KADRE. Cluster quality, or $Q(a, b)$, is defined as: $Q(a, b) = (1 - \frac{\mathbf{MoJo}(a,b)}{n}) \times 100\%$ where n is the number of elements clustered [17].

4. EVALUATION

We preformed a leave-one-out cross validation analysis in order to show KADRE's predictive abilities over unseen data. We separated the evaluation suite into seven training sets, each leaving a single program as a target set. We trained the weighting parameters to maximize the quality of clusterings for each training set program and then measured the quality of clusterings produced using these parameters for the target program.

Overall, KADRE was highly successful at automatically creating clusters that match our expertly-created scientific kernels (see Figure 1). We measured an average error of

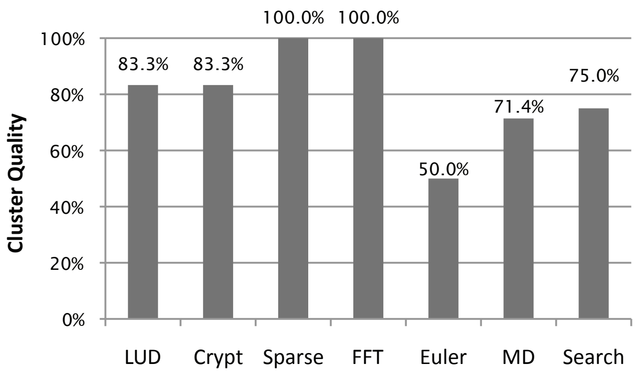

Figure 1: Cross-validation analysis results.

19.57%. This result indicates that KADRE performs well for unseen data, producing clusters that match scientific kernels with greater than 80% accuracy.

In our analysis, we found that data dependencies and call relationships between program elements were more significant than the proximity of the elements. This is not surprising considering the ill-formed encapsulations of the scientific software systems in our evaluation suite. Additionally, data dependencies were more influential (and therefore weighted more heavily) than call relationships. Again, this outcome of training is logical in that an important goal of our componentization is to minimize the amount of data transferred in the dataflow connectors (e.g., between workflow stages).

5. CONCLUSION & FUTURE WORK

In this paper, we have defined domain-specific software architecture recovery and shown how domain knowledge can be leveraged to recover components conformant to a domain-specific architecture. Specifically, we have shown that much of the recovery effort can be automated. As orchestration of scientific kernels is key to the development of scientific software supporting replication, scaling, and third-party validation, we have helped further the scientific goals of "in silico" experimentation.

We have provided scientists with a means of identifying kernels, not only to build a componentized version of the original software to be orchestrated via workflow, but also to provide the scientist with an effective basis for component-based reuse. Scientific kernels can be composed into new scientific software, and can be used in evolving legacy software systems more rapidly than was possible through platform- and language-dependent library-based reuse.

In our future work, we will integrate KADRE's kernel identification with our previous work on architectural wrappers for workflow components [20] in order to automatically generate workflow orchestrations.

Acknowledgments

The authors thank Joshua Garcia. Effort partially conducted at the Jet Propulsion Laboratory managed by the California Institute of Technology under contract with NASA.

6. REFERENCES

[1] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *IWPC*, pages 259–268, May 2005.

[2] J. M. Bull et al. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, pages 375–388, 2000.

[3] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE TSE*, 35(4):573–591, 2009.

[4] J.-F. Girard et al. A metric-based approach to detect abstract data types and state encapsulations. *JASE*, 1997.

[5] A. E. Hassan and R. C. Holt. A reference architecture for web servers. In *Proc. of WCRE*, 2000.

[6] B. Hayes-Roth et al. A domain-specific software architecture for adaptive intelligent systems. *IEEE TSE.*, 21:288–301, 1995.

[7] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *IWPC*, pages 201–210, 2000.

[8] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *JASE*, page 251, 1999.

[9] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE TSE*, 33(11):759–780, 2007.

[10] C. A. Mattmann et al. The anatomy and physiology of the grid revisited. In *WICSA/ECSA*, 2009.

[11] N. Medvidovic and V. Jakobac. Using software evolution to focus architectural recovery. *JASE*, 13(2):225–256, 2006.

[12] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *ICSM*, page 744, 2001.

[13] J. Segal. Some challenges facing software engineers developing software for scientists. In *ICSE SECSE09'*, pages 9–14, 2009.

[14] C. E. Shannon. A mathematical theory of communication. *Bell System Tech. Journal*, 27, 1948.

[15] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 2009.

[16] W. Tracz. Dssa (domain-specific software architecture): Pedagogical example. *ACM SEN*, 20, 1995.

[17] V. Tzerpos and R. Holt. Mojo: A distance metric for software clusterings. *WCRE*, page 187, 1999.

[18] G. Wilson. How do scientists really use computers? *American Scientist*, 97(5), September/October 2009.

[19] D. Woollard et al. Case studies in science data systems: Meeting software challenges in competitive environments. In *Int. Conf. on Space Operations*, 2008.

[20] D. Woollard et al. Injecting software architectural constraints into legacy scientific applications. In *ICSE SECSE09'*, pages 65–71, 2009.

[21] D. Woollard et al. Kadre:domain-specific architectural recovery for scientific software systems. Technical report, USC-CSSE-2010-514, 2010.

[22] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Record*, 34(3), 2005.