ABSTRACT
Scientists today conduct new research via software-based experimentation and validation in a host of disciplines, including materials science, life sciences, astronomy, and physics. Scientific software represents a significant investment due to its complexity and longevity, its life spanning sometimes decades. Unfortunately, there is little reuse of scientific software beyond small libraries, increasing development and maintenance costs. Modern workflow and grid technologies offer a promising medium for reuse in this domain, however reuse at the workflow level is quite different than that of software libraries. 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 traditional state of the art general cluster techniques, helping to promote component-based reuse of scientific kernels 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 researchers to run powerful climate models [?], fold proteins for drug discovery [?], and model the effects of earthquakes in order to better understand our natural environment [?]. The complex scientific software systems that facilitate this type of in silico experimentation represent a significant investment not only in time to develop, but also in time to maintain. As an example, software developed for Earth science missions at NASA’s Jet Propulsion Laboratory must survive years and, in some cases, 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 This software must be updated over time to deal with machine failures, outdated software and changes to software/hardware support, as well as to take advantage of new technologies (where possible) to reduce operations and maintenance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. Furthermore, maintenance may be necessary in order to update the scientific codes for new experimentation, or instrument calibration.

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 [?]. Often these two groups are the same people, leading to the end-user developer problem [?].

To save development efforts, scientists reuse successful code by compiling it into software libraries and referencing these libraries in later software systems. We have observed that as scientists evolve their experiments or increase the fidelity of their simulations, library reuse is limited to small, well-defined functions, and the scientist spends much of her time understanding and adapting the control flow of the software system in order to introduce new features in a largely haphazard fashion. This observation is confirmed in literature [?].

Unlike the haphazard and often opportunistic adaptation of existing scientific software, the codification of the scientific processes and experiments in the formalization of workflow-based composition [?, ?] offers the scientist a promising advancement in in silico development. In this paradigm a scientist builds a workflow model of the software system, relying the experiment in a highly-modifiable, flexible model rather than embedding it in the control flow of a monolithic software system, affording great flexibility of orchestration.
placed in the system with new modules, leading to more coarse-grained software reuse and providing the scientists a more domain-oriented, less software-centric model to achieve reuse. Additionally, workflows can support the scientific process in a number of vital ways, including experiment replication, scaling, and third-party validation, each of which is currently limited by language-specific and platform-specific library-based software reuse.

Unfortunately, the state of practice points to a significant disconnect between (1) developers of scientific software systems and (2) the software engineers and computer scientists attempting to support this domain through advances in workflow technology on Grid [?], Cloud and Web computing infrastructure. Indeed, in a recent survey [?], 81% of nearly 2000 computational scientists surveyed did not utilize these advanced computational infrastructures in their in silico experimentation, despite the advantages [?]. A means of automatically extracting relevant workflow models from existing software is key to facilitating this migration, enabling software reuse at the level a scientist desires, closely coupling the scientist’s expertise with her ability to maintain and evolve the software, and ultimately reducing the need for her to become a software expert in addition to a domain expert.

A key insight in the course of our study into scientific software [?] is that scientific codes often follow an implicit dataflow architecture. By automatically recovering this dataflow architecture from code, we can provide scientists with a method of identifying the workflow stages, or software kernels, in their existing software systems. In this paper, we 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 kernels in scientific software, not only to build componentized versions of existing, monolithic software orchestrated via workflow, but also to enable component-based reuse. Scientific kernels can be composed into new scientific software, evolving legacy software systems more strategically than was feasible through low-level, library-based reuse and adaptation.

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 will introduce software kernels in greater detail, describe our methodology for recovering these kernels from software, and present our clustering algorithms for kernel discovery and their implementation in KADRE. In Section 4, we will show how we have developed a representative set of domain applications and measure compare KADRE’s automatic clustering to manual decomposition. Section 5 presents our evaluation, including representations of the search space, training results, and the benefits to the scientist. Finally, we will present conclusions and 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) [?]. 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 [?]. Hayes-Roth et al., extended this definition to include not only a reference architecture, but other reusable artifacts including reference components capturing domain knowledge and configuration methods for selecting these components and configuring the reference architecture [?]. 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) [?, ?].

It is within these definitions of domain-specific software architecture and architectural recovery that we will define domain-specific architectural recovery—an architectural recovery process that utilizes domain knowledge to extract system architectures compliant to a domain-specific reference architecture. With KADRE, we are concerned with identifying 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). We refer to this type of software component as a kernel.

There are several different automatic approaches to general architectural recovery, most of which result in partial architectural descriptions [?]. These partial descriptions are developed utilizing feature extraction from both static analysis and runtime analysis, and agglomeration techniques such as clustering [?]. The fundamental notion in software clustering is discerning a similarity between elements of the input source code. 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.

Approaches to automatic clustering have leveraged genetic algorithms [?], latent semantic analysis of source code [?], and software evolution and change [?], while the semi-automatic techniques have focused on metrics [?] and development of a means of assessing a cluster’s quality [?].

Maqbool and Babri recently reviewed hierarchical clustering techniques for software architecture recovery and provided an analysis of the behavior of various similarity and distance measures [?]. As part of their conclusion, the authors emphasize that the quality of clustering depends on the characteristics of the analyzed software systems. KADRE utilizes the common characteristics of software in the scientific computing domain to recover workflow components.

Related to our proposed automatic domain-specific technique are domain-specific recovery methodologies [?], [?] that utilize human interpretation to cluster source code into more highly-refined architectural models [?, ?]. These methodologies recover domain-specific architectures utilizing domain knowledge such as source code, domain specific terms, system documentation, and experience of architects who have built systems within the particular domain.

Once identified and isolated by KADRE, kernels can be integrated into scientific workflow systems responsible for modeling, executing, and managing the discovery and production phase of in-silico science [?]. Composition of scientific workflows is being studied extensively by the Grid [?] community. A representative project is Wings (Workflow INstance Generation and Selection) [?] which enables scien-
3. APPROACH

In order to move from the low-level reuse to workflow-based orchestration (and component-based software reuse), scientists must re-architect existing monolithic scientific software systems into workflow component-based systems. As workflow systems are constructed to support dataflows, scientists are, in essence, recovering the dataflow architecture of existing software systems.

Illustrated in Section 2, clustering techniques have shown to be useful for this problem by agglomerating software elements into system components, although a majority of these techniques have used general heuristics for tuning parameters and feature weights [?]. To improve upon these heuristics, we can leverage domain specificity in a significant way. Because we have constrained our domain to scientific code (e.g., mathematical and physical simulations), we posit that it is possible to construct a representative training set that can be used to improve upon the results of general clustering algorithms by training via supervised clustering.

In the rest of this section, we will present our hypothesis regarding the process of scientific software decomposition based on our collective experience in this domain, and our codification of this process into a clustering algorithm. Additionally, we will further illustrate our clustering process with an example scientific application.

3.1 Hypothesis

Over the last five years the authors of this paper have worked to study and refactor existing software used by JPL into workflow-based systems as part of our work with OODT [?], a data and computational grid platform. Based on our experience, 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 [?]: 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 the elements in the source code (i.e., Are two functions in the same class? Are they in the same subsystem?).
- The call distance between elements (i.e., Are two functions called by the same parent function? Are they executed far from one-another temporally?).
- The 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. In the rest of this section, we will present just such a clustering process, which we call KADRE.

3.2 Clustering Algorithm

In organizational terminology, a cadre is a group of key personnel or entities in an organization that form its core. Likewise, 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, given as Algorithm 1, 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.

**Algorithm 1: Iterative Element Clustering Algorithm.**

```
input : program P
output: set C of element clusters
set C ← all elements ∈ P
while |C| > 1 and ∃(a, b) ∈ C × C : Sim(a, b) ≥ δ do
  find max_{a,b ∈ C} Sim(a, b)
  remove a and b from C
  add (a, b) to C
```

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 δ, which is a tunable parameter (see Section 4 for more information about how the threshold is set) or when a single cluster is produced. The latter condition is added to the algorithm to ensure termination.

Algorithm 1 is compact, codifying much of the complexity in Sim(a, b), which we discuss in the ensuing section and in the remainder of the paper. The work to determine Sim(a, b) is a result of our study of the scientific software domain and our own realization of the utility and power of understandable, compact measures and heuristics.
In the next subsection, we will describe how we measure similarity and further illustrate its use via an example scientific software package.

### 3.3 Formal Definition of Similarity

During the affinity clustering process given in Subsection 3.2, the key metric employed to judge “distance” (and clustering potential) is the similarity between any two elements in the program. During our study of the manual clustering process for discovering scientific kernels, we utilized a number of sources of information in order to determine if two program elements should be clustered together.

We have formalized this information into our similarity metric (see in Algorithm 1), \( Sim(a, b) \), given as Equation 1. As with the manual process we studied, similarity in our clustering algorithm is affected by three sources of information: (1) the proximity measure between \( a \) and \( b \), given as \( prox(a, b) \), (2) the scaled distance between the entities in the program’s call tree, given as \( call(a, b) \), and (3) the data dependency measure between \( a \) and \( b \), given as \( data(a, b) \).

\[
Sim(a, b) = \frac{\alpha \cdot prox(a, b) + \beta \cdot call(a, b) + \gamma \cdot data(a, b)}{\alpha + \beta + \gamma}
\]  

In Equation 1, \( \alpha \), \( \beta \), and \( \gamma \) are weighting parameters that adjust the influence of these measures on the overall similarity of two software features. In the rest of this subsection, we will describe each of the measures in greater detail. In Section 4, we will show how we train the weighting parameters to be tuned specifically for the domain of scientific software.

#### 3.3.1 Proximity

The code-level proximity between any two software elements is best understood by visualizing 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. This entity relationship is shown in Figure 1.

![Figure 1: A general program element model and hierarchical tree view.](image)

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 sub-tree 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.

If each of the elements’ parents are contained by the same parent container, then their proximity is language dependent because various languages support different degrees of encapsulation. For languages that do not support objects, the element hierarchy tree is more shallow than the general model presented in Figure 1. In Fortran77, for example, programs contain modules, modules contain functions/subroutines, and functions and subroutines contain lines of code.

In order to accommodate different source languages, we assign the proximity of elements with this relationship the value \( \lambda \), a parameter with a value such that \( 1 > \lambda > 0 \). \( \lambda \) is an adjustable parameter used to adapt the proximity metric \( a \ priori \) to different element hierarchies. For languages with shallower entity trees, entities are more likely to share a common parent regardless of whether they should be clustered, so we lower the value of \( \lambda \) to compensate.

The equation for \( prox(a, b) \) is given in Equation 2.

\[
prox(A, B) = \begin{cases} 
1 & \text{if } \text{par}(A) = \text{par}(B) \\
\lambda & \text{if } \text{par}(\text{par}(A)) = \text{par}(\text{par}(B)) \\
0 & \text{if } \text{par}(\text{par}(A)) \neq \text{par}(\text{par}(B)) 
\end{cases}
\]

where \( 1 > \lambda > 0 \) and \( \text{par}(\Gamma) = \text{the parent entity of } \Gamma \)

#### 3.3.2 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. If two functions in a program are executed in sequence, then they are good candidates for clustering, whereas, for example, a function that is called only at the beginning of execution is not a good candidate to cluster with a function that is called only at the program’s end.

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. The equation for calculating \( call(a, b) \) is given as Equation 3.

\[
call(a, b) = 1 - \frac{\text{minimal path from } a \text{ to } b \text{ in graph } P'}{\text{longest path in } P'}
\]

where \( P' \) is the undirected call graph of program \( P \)

#### 3.3.3 Data Dependency

The final measure that affects the similarity metric of two elements, \( a \) and \( b \), is \( data(a, b) \), given in Equation 4. If two program elements share a large amount of data, then they are good candidates for clustering. If we revisit the definition of scientific kernels given in Section 3.1, it is important that we minimize the data dependencies between the resultant clusters in order to allow the kernels to be stateless and exchange all data dependencies with control flow.

Our measure of \( data(a, b) \) is similar to Girard, et. al.’s indirect relationship metric in [?], although we analyze this re-
lationship using a resource tree of data dependencies rather than a flow graph. Put another way, we do not capture 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 $call(a, b)$).

$$\text{data}(a, b) = \frac{\text{common}(a, b)}{\text{common}(a, b) + \text{distinct}(a, b)}$$

(4)

As shown in Equations 5 and 6, $\text{common}(a, b)$ is the set of data dependencies, or resources, common to both element $a$ and element $b$ and $\text{distinct}(a, b)$ is the set of resources which are used by either $a$ or $b$ but not both. As with [1], the operator $\oplus$ is the symmetric difference of the sets.

$$\text{common}(a, b) = W(\text{resources}(a) \cap \text{resources}(b))$$

(5)

$$\text{distinct}(a, b) = W(\text{resources}(a) \oplus \text{resources}(b))$$

(6)

Not all shared resources should be weighted equally, however. 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 clusters. In Equations 5 and 6, $W$ is a weighting parameter given by multiplying the size (memory footprint) of the shared resource normalized to overall program footprint by its Shannon information content [1] – a measure of the resource’s frequency of use. Equation 7 shows how the weights of the parameters determined to be both $\text{common}$ and $\text{distinct}$ are applied.

$$W(X) = \frac{\sum_{x \in X} w(x)}{\sum_{e \in X} w(e)}$$

$$\text{where } w(x) = -\log(\text{prob}(x)) \cdot \text{size}(x)$$

(7)

In Equation 7, $\text{prob}(x)$ is the probability that, given an element in program P, it will utilize $x$ as a data resource. Additionally, the set $E$ is the set of all elements in program P. From these equations, we can see that if a particular resource is shared by all elements of a program (its $\text{prob}(x) = 1$), then the log term in Equation 7 zeroes out: it provides no information to us, so its Shannon information content is zero. Additionally the size of $x$ is a multiplier to give more weight to variables with larger memory footprints.

### 3.4 An Example

In order to illustrate the clustering process, and specifically, to show how similarity between program elements is utilized, we will use an example common to the scientific domain: a Lower-Upper Matrix Decomposition package called LUD that is used to solve systems of linear equations.

#### 3.4.1 Conceptual Overview

The LUD software solves a system of equations, $a \bar{x} = b$ for $x$, given in the form:

$$a = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

by performing four conceptual steps:

1. Populate matrices $a[][]$ and $b[]$
2. Perform a partial pivot operation on $a[][]$
3. Solve the system $a[][] \times x[] = b[]$
4. Compute $b[]$ by multiplying out $a[][]$ with the result to step 3 and computing the error

Because each kernel represents a conceptual step in the scientific process, our clustering algorithm should produce four clusters, each corresponding to one of the conceptual steps outlined above.

```java
1 public class LUD {
2   double a[][], b[], x[];
3   int ipvt[], n;
4
5   public void init(int size){
6       ... /*populate a[n][n], b[n]*/
7       21 }
8
9   public void run(){
10      dgefa(a,ipvt,n); //perform partial pivot
11      dgesl(a,ipvt,b,n) //solve a[n][n]*x[n]=b[n]
12      26 }
13
14   public void validate(){
15       ... /*estimate roundoff error*/
16      dmxpy(a,b,x,n); //calculate a[n][n]*b[n]
17      ... /*compare residual*/
18      57 }
19
20   private void dmxpy(double a[], double b[], double x[], int n){
21      ... /*calculate a[n][n]*x[n]=b[n]*/
22      63 }
23
24   private void dgefa(double a[], int ipvt[], int n){
25       ... /*perform partial pivot on a[]*/
26      109 }
27
28   private void dgesl(double a[], int ipvt[], int n){
29       ... /*solve for b[]*/
30      135 }
31
32   public static void main(String[] args){
33      LUD lud = new LUD();
34      lud.init(500);
35      lud.run();
36      lud.validate();
37      143 }
```

*Figure 2: Elided source code for LUD.*

In order to properly illustrate the clustering process, we show elided source code for LUD in Figure 2; we show method
signatures, methods calls, and variables shared by the methods, although we have omitted the actual method logic for clarity.

### 3.4.2 Clustering Process

From Figure 2, we can see that there are seven methods, or program features, we are clustering, namely main, init, run, validate, dmxpy, dgefa, and dgesl. In order to determine if any of the elements are to be clustered together, we need to calculate the pairwise similarity of each element and each of its constituent measures: proximity, call relationship, and data dependency.

Due to space constraints, we will illustrate the calculation of our similarity metric on two methods, run and dmxpy; the reader should be able to verify the full affinity matrix for LUD given below. In order to calculate Sim(run, dmxpy), we must calculate the three individual features: prox(run, dmxpy), call(run, dmxpy), and data(run, dmxpy).

In order to calculate proximity, we arrived at a value of \(\lambda = 0.4\) for Java-based software after some experimentation. Because LUD is a small example comprising a single class, all methods are highly proximate to one another, and so \(\text{prox}(\text{run}, \text{dmxpy}) = 1\).

![Figure 3: LUD’s undirected call tree.](image)

In order to calculate the call relationship between each of the methods in LUD using Equation 3, we must first recover the call graph of LUD. The undirected version of this call graph is shown in Figure 3.

From the undirected graph in Figure 3, we can calculate the longest path in LUD’s undirected call tree to be 4 (dgefa() \(\rightarrow\) run() \(\rightarrow\) main() \(\rightarrow\) validate() \(\rightarrow\) dmxpy()). Using this information and Figure 3, we can calculate \(\text{call}(\text{run}, \text{dmxpy}) = 0.25\).

To calculate the data dependency relationship between run and dmxpy, we first must calculate the weighting for each of the data resources in LUD. From Figure 2, we can see that there are five shared resources, namely a[], x[], b[], ipvt[], and n. Table 1 shows our calculations for \(\text{prob}(x)\), \(\text{size}(x)\), and \(w(x)\) for each resource in LUD.

![Table 1: Weights associated with LUDcomp resources.](image)

<table>
<thead>
<tr>
<th>Resource</th>
<th>(\text{prob}(x))</th>
<th>(\text{size}(x))</th>
<th>(w(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[]</td>
<td>1.0</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>x[]</td>
<td>0.83</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>b[]</td>
<td>0.67</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>ipvt[]</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>n</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

We can calculate \(\text{data}(\text{run}, \text{dmxpy})\) in the equation as follows:

\[
\text{data}(\text{run}, \text{dmxpy}) = \frac{W((\text{a}[], \text{b}[], \text{n}))}{W((\text{a}[], \text{b}[], \text{n})) + W((\text{x}[], \text{ipvt}[]))}
\]

Solving this equation, we see that \(\text{data}(\text{run}, \text{dmxpy}) = 0.4\).

We can combine these features to produce the following calculation for \(\text{Sim}(\text{run}, \text{dmxpy})\):

\[
\text{Sim}(\text{run}, \text{dmxpy}) = \frac{\alpha(1) + \beta(0.25) + \gamma(0.4)}{\alpha + \beta + \gamma}
\]

In the next subsection, we will show how the tuning of these weighting parameters, as well as the threshold parameter \(\delta\), creates different clusterings, some of which more accurately reflect our notion of scientific kernels.

### 3.4.3 The Role of Tuning Parameters

Assuming an even distribution of weights to each of the relationships above (e.g., \(\alpha = 1\), \(\beta = 1\), and \(\gamma = 1\)), we would arrive at the affinity matrix shown as Table 2.

![Table 2: Affinity matrix for LUD assuming even weighting.](image)

<table>
<thead>
<tr>
<th></th>
<th>init</th>
<th>run</th>
<th>validate</th>
<th>dmxpy</th>
<th>dgefa</th>
<th>dgesl</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>0.7</td>
<td>0.767</td>
<td>0.683</td>
<td>0.48</td>
<td>0.617</td>
<td></td>
</tr>
<tr>
<td>run</td>
<td>0.633</td>
<td>0.55</td>
<td>0.494</td>
<td>0.55</td>
<td>0.917</td>
<td></td>
</tr>
<tr>
<td>validate</td>
<td>0.917</td>
<td>0.42</td>
<td>0.55</td>
<td>0.33</td>
<td>0.467</td>
<td>0.667</td>
</tr>
<tr>
<td>dmxpy</td>
<td>0.683</td>
<td>0.55</td>
<td>0.494</td>
<td>0.33</td>
<td>0.467</td>
<td></td>
</tr>
<tr>
<td>dgefa</td>
<td>0.48</td>
<td>0.69</td>
<td>0.917</td>
<td>0.667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dgesl</td>
<td>0.617</td>
<td>0.917</td>
<td>0.467</td>
<td>0.667</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Table 2, we can see that the pair of entities to be clustered first is either dmxpy and validate, or run and dgesl. Assuming that we cluster the former pair first and the latter pair second, we arrive at a reduced affinity matrix in which the next pair of entities to agglomerate is init and the cluster formed by validate and dmxpy. We can see from Table 3, however, that the similarity of this pair has fallen much lower than the similarities of the perviously combined program elements.

![Table 3: Reduced affinity matrix for LUD after two clusterings.](image)

<table>
<thead>
<tr>
<th></th>
<th>init</th>
<th>valid/dmxpy</th>
<th>dgefa</th>
<th>run/dgesl</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>0.683</td>
<td>0.4833</td>
<td>0.617</td>
<td></td>
</tr>
<tr>
<td>valid/dmxpy</td>
<td>0.333</td>
<td>0.467</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dgefa</td>
<td>0.333</td>
<td>0.467</td>
<td></td>
<td></td>
</tr>
<tr>
<td>run/dgesl</td>
<td>0.667</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recalling Algorithm 1, program elements are only combined if their similarity is greater than the threshold parameter, \(\delta\). Like \(\alpha\), \(\beta\), and \(\gamma\), \(\delta\) is a trainable parameter. In this example, a \(\delta\) greater than the distance between init and the cluster consisting of valid and dmxpy (see Table 3), stops KADRE’s clustering process at a point in which the clusters are meaningful scientific kernels (described in Subsection 3.4).

\(\delta\) is not the only tuning parameter that plays a role in our clustering process. Had the weightings \(\alpha\), \(\beta\), and \(\gamma\) not been set equal, but instead favored call relationships more heavily...
(α = 0.2, β = 0.5, and γ = 0.3, for example), then the first two clusterings would once again take place, but it would be dgefa and the cluster formed of run and dges1 that would be the third potential clustering. If these two methods are clustered, then we have combined the conceptual steps 2 and 3 of LUD (recalling Section 3.4.1), violating our definition of scientific kernels.

In fact, an affinity-based clustering tool called Bauhaus, which implements a widely-cited component recovery technique [7, 10], gave us just this clustering (see Figure 4). Since this is an untrained clustering tool, the user of the tool must somehow determine the weighting of 19 inter-related parameters in order to arrive at a more meaningful clustering.

Figure 4: Initial clustering produced by Bauhaus.

In the next section, we will show how we leverage domain specificity to create a training set of sample scientific software systems and also how we utilize this training set to incrementally improve the clusters produced by KADRE without the manual intervention of the user.

4. PARAMETER TRAINING

From the LUDecomp example in the last section, we saw that different values of α, β, γ, and δ effect a weighting mechanism that produces different clusterings of program elements. In order to refine the weighting of individual measures in the Sim(a, b) metric, we can compare the clusterings produced by KADRE to expertly decomposed scientific kernels, adjusting α, β, γ, and δ in order to more closely align the clusters with scientific kernels.

Figure 5: Three different clusterings of LUD.

In Figure 5, we see three different clusterings of LUD, each produced with different weights associated with the features calculated by Sim(a, b) and also different thresholds. Clustering A matches our manually decomposed scientific kernels for LUD. Clustering B is a clustering that would be produced if all features (prox, call, and data) were equally weighted, and 0.34 < δ < 0.683. This example was discussed above. As another example, Clustering C is produced if α = 0.2, β = 0.5, γ = 0.3, and 0.565 < δ < 0.6.

In the rest of this section, we will describe how we tune these parameters to increase the quality of KADRE’s clusterings. We will detail how we developed a training set and how we measure the distance between two different clusters.

4.1 Developing an Evaluation Suite

In order to train KADRE to accurately cluster program elements into scientific kernels, we have developed an evaluation suite of scientific software systems. To be a training set, the programs must have matching decompositions performed by experts. Furthermore, the systems themselves must be representative of the domain. We chose seven systems from the Java Grande Benchmarking Suite [11] to build our suite (see Table 4 for descriptions of the systems).

The Java Grande Benchmark is a set of scientific and engineering applications originally produced to test the performance of the Java programming language for use in the computational science domain. Each program was selected as representative of the domain by computational science experts from academia, industry and government [7]. They are also representative of a variety of coding practices and code structures (object-orientation, hierarchical composition, and class encapsulation are used to varying degrees) as they were initially written by different developers. Based on a recent survey, these programs share a number of characteristics with computational science’s average code, including size; Wilson found that more than two-thirds of the computational scientists he surveyed worked with scientific software with less than 5 KSLOC [7]. Despite the small size of these programs as measured in SLOC, they are are still very complex (200 lines of byte-level data manipulations in the case of crypt, for example).

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. During manual decomposition, we not only studied the static and dynamic properties of the software systems, but also used our knowledge of the domain and specifically our understanding of the scientific and mathematical processes implemented by each of the systems to arrive at a decomposition representative of the high-level steps in the systems’ respective scientific processes. For descriptions of the decomposed systems, please see [7].

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.

As further validation for our selection of decompositions, we redeveloped each program as a workflow-based orchestration utilizing the selected kernels. In each case, we validated the equivalence of the scientific results produced by
4.2 Measuring Quality

For KADRE to discover not just software clusters but true scientific kernels, we trained KADRE to produce clusterings of necessary quality. In or to define quality, we first 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. Using Clusterings A and C from Figure 5, we can see that if dgefa was moved to the cluster containing run and dgesl, we could transform Clustering A to Clustering C. Similarly, if we where to move init to the cluster containing valid and dmxy, we can transform Clustering C to Clustering B.

In [7], Tzerpos and Holt defined the MoJo metric as the number of Move and Join 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. Quality, or \(Q(a, b)\), is defined as: \(Q(a, b) = (1 - \frac{\text{MoJo}(a, b)}{n}) \times 100\%\) where \(n\) is the number of elements clustered [7].

5. EVALUATION

In order to evaluate KADRE, we have performed four analyses: Initially, for each program in the evaluation suite, we performed a coarse-grained exhaustive search in order to generate search space maps. These maps allow us to form general conclusions about successful trends in the values of our weighting parameters. Second, we performed a side-by-side comparison of KADRE and a general software clustering technique implemented in Bauhaus (as described in Section 3). Additionally, we performed a leave-one-out cross validation analysis in order to show KADRE’s predictive abilities over unseen data. Finally, we present an initial case study analysis in which we estimate the time saved in developing workflow orchestrations of the evaluation suite using KADRE and the increase to the quality of resulting clusters.

5.1 Mapping the Search Space

KADRE’s analysis is inexpensive; the mean execution time of KADRE on the evaluation suite using a mid-range PC (CPU: Intel Dual-core 3.06GHz; RAM: 4 Gb) is 2.3 seconds. It is therefore possible to do a coarse-grained exhaustive search. For each program in the evaluation suite, we produced 1000 clusterings, initially allowing KADRE to halt after producing a single cluster (e.g., by setting \(\delta=0\)). In each of these runs, we varied incrementally by 0.1 the weighting values of \(\alpha\), \(\beta\), and \(\gamma\) between 0.0 and 1.0 inclusive.

For each clustering iteration (i.e., after each clustering decision is made – see Algorithm 1), we measured the MoJo between the clustering produced thus far by KADRE and our expert decompositions. We used these measures to form maps of the search space for each of the programs in the evaluation suite. In Figure 6, we show an example map that we produced for LUB, showing the relationship between \(\beta\), the weighting parameter for prox(a, b), and \(\gamma\), the weighting parameter for data(a, b).

In our analysis, the 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, \(data(a, b)\), weighted by the parameter \(\gamma\), was more influential (and therefore weighted more heavily) than \(call(a, b)\). The best results in Figure 6 were obtained with low \(\beta\). 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). Due to space limitations, we are unable to show every map we created; they can be found in [?].

5.2 Comparison to Other Techniques

Using the most successful weighting generated during our exhaustive search (α = 0.1, β = 0.3, γ = 0.6, and δ = 0.4), we evaluated the distance between KADRE’s clustering of each of the evaluation suite programs and the “best of breed” scientific kernels using Mojo. Further, we compared these clusters to the clusters produced by an untuned version of Bauhaus as described in Koschke’s work [?]. Results of this comparison using the Q metric—described above—are shown in Table 5.

Table 5: The measured quality of KADRE’s and Bauhaus’ clusterings.

<table>
<thead>
<tr>
<th></th>
<th>KADRE</th>
<th>Bauhaus</th>
<th>Euler</th>
<th>KADRE</th>
<th>Bauhaus</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUD</td>
<td>83.3%</td>
<td>83.3%</td>
<td>50%</td>
<td>62.5%</td>
<td></td>
</tr>
<tr>
<td>Crypt</td>
<td>83.3%</td>
<td>66.7%</td>
<td>MD</td>
<td>71.4%</td>
<td>64.3%</td>
</tr>
<tr>
<td>Sparse</td>
<td>100%</td>
<td>50.0%</td>
<td>Search</td>
<td>75.0%</td>
<td>50.0%</td>
</tr>
<tr>
<td>FFT</td>
<td>100%</td>
<td>25.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Overall, KADRE was highly successful at automatically creating clusters that match our expert-created scientific kernels. We should note that we would expect Bauhaus to perform better had we experimentally tuned its affinity clustering algorithms. Nonetheless, we feel that the comparison presented in Table 5 is fair considering that any tuning of Bauhaus would be left to the scientist [?], and may be arbitrary without significant knowledge of the algorithm.

5.3 Cross Validation

In order measure the predictive value of KADRE on unseen data, we performed a leave-one-out cross-validation analysis—an analysis common in statistical approaches to model fitting. We separated the evaluation suite into seven training sets, each leaving a single program as a target set. We selected values for α, β, γ, and δ that maximized the quality of clusterings for each training set program and then measured the quality of clusterings produced using these parameters for the target program.

We measured an error of 19.57% with a mean squared error of 6.43% in this analysis. This result indicates that KADRE performs well for unseen data, producing clusters that match scientific kernels with greater than 80% accuracy.

5.4 Case Study Analysis

As part of an advanced graduate-level course in software architecture, twenty-three teams of computer science students—both full time students and working engineers—were each asked to decompose one of the programs in the evaluation suite and orchestrate the program via workflow. The manual decomposition took an average of nine developer weeks to accomplish based on self-reporting in required status reports whereas, the orchestration took an average time of one developer week.

While most teams generally agreed with one another on the decompositions as well as with our expert decompositions, we found that the teams could potentially fall into two traps during manual decomposition: under-decomposition as a result of the lack of understanding of the overall dataflow in the program, and over-decomposition that correlated with lack of understanding of the scientific concepts being reified. We saw an overall variance of 22% in Q(a,b) for manual decomposition, suggesting that it is more prone to error than KADRE-based decomposition.

We have drawn two conclusions from this case study: (1) the error rate of manual decomposition is greater than that of KADRE’s automatically produced decompositions, and (2) the time to produce a workflow orchestration of a scientific program can be greatly reduced when KADRE is used (several seconds using KADRE vs. several weeks using the manual approach).

5.5 Threats to Validity

There are two primary threats to the validity of this research that we should note, including the threat of overtraining on selected scientific systems and the threat that the sampling of these systems was biased. In order to address these threats, we have selected scientific applications that were judged by a forum of domain experts as representative of the domain and were created by individual contributors utilizing a variety of development methodologies over a long period of time.

Additionally, we have produced target decompositions ourselves (owing to the lack of a commonly accepted training dataset) and these decompositions are treated as “ideal” solutions. Expert decomposition is common practice in the area of software clustering evaluation [?]. To mitigate our bias, we produced these decompositions utilizing not only our own expertise in the domain but also through selection of “best of breed” decompositions produced by over eighty software engineers working in small teams and vetted for accurate representation of the scientific steps implemented in software.

6. 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 that can now 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 [?] in order to automatically generate workflows orchestrations.

Acknowledgments

The authors thank Joshua Garcia for his assistance. This research was partially conducted at the Jet Propulsion Laboratory managed by the California Institute of Technology under contract with NASA.