Enhancing Architectural Recovery Using Concerns

Joshua Garcia*, Daniel Popescu*, Chris Mattmann†*, Nenad Medvidovic*, and Yuanfang Cai‡

*Computer Science Department
University of Southern California,
Los Angeles, CA 90089, USA
{joshuaga,dpopescu,neno}@usc.edu

†Jet Propulsion Laboratory
California Inst. of Technology,
Pasadena, CA 91109, USA
mattmann@jpl.nasa.gov

‡Computer Science Department
Drexel University,
Philadelphia, PA 19104, USA
yfcai@cs.drexel.edu

***Abstract*—Architectures of implemented software systems tend to drift and erode as they are maintained and evolved. To properly understand such systems, their architectures must be recovered from implementation-level artifacts. Many techniques for architectural recovery have been proposed, but their degrees of automation and accuracy remain unsatisfactory. To alleviate these shortcomings, we present a machine learning-based technique for recovering an architectural view containing a system's components and connectors. Our approach differs from other architectural recovery work in that we rely on recovered software concerns to help identify components and connectors. A concern is a software system's role, responsibility, concept, or purpose. We posit that, by recovering concerns, we can improve the correctness of recovered components, increase the automation of connector recovery, and provide more comprehensible representations of architectures.**

I. Introduction

The effort and cost of software maintenance tends to dominate other activities in a software system's lifecycle. A critical aspect of maintenance is understanding and updating a software system's architecture. However, the maintenance of a system's architecture is exacerbated by the related phenomena of architectural *drift* and *erosion* [1], which are caused by careless, unintended addition, removal, and/or modification of architectural design decisions. These phenomena make the architecture more difficult to understand and maintain and, in more severe cases, can lead to errors that result in wasted effort or loss of time, money, even lives.

To deal with drift and erosion, a number of techniques have been proposed to help recover a system's architecture from its implementation [2]. These techniques mainly map implementation-level entities to high-level system *components* by clustering the entities and taking the resulting clusters to be components [3]–[8]. However, the prevailing coupling-and-cohesion-based clustering methods do not recover the *concerns* associated with the components, making it difficult to understand the meaning of a cluster or whether a cluster truly represents a component.

Moreover, the architecture of a software system also consists of *connectors*, which play a critical role in mediating component interactions [1]. Although a number of existing component recovery methods are automated, the connector recovery techniques uniformly depend on significant human involvement. In particular, existing techniques for connector recovery use patterns or queries to identify the connectors within a system [9]–[12]. These techniques require an architect to write a pattern or query for each implementation variant of every possible connector type. Creating such specifications is a manual task that can be time consuming and error prone.

This paper describes a novel technique that leverages system concerns to automate the recovery of both components and connectors. The objective of this work is to obtain automatically recovered software architectures that are more comprehensive and more accurate than those yielded by current methods. Different from existing techniques, which exploit only structural information to discover components (e.g., programming language-level dependencies and shared directories), our approach first recovers *concerns* from the implemented system using an information retrieval technique, and then combines the concerns with the structural information to automatically identify components as well as connectors. A concern is a role, responsibility, concept, or purpose of a software system. Components and connectors are, therefore, high-level elements containing data and computation that implement the concerns. The key distinction guiding this work is that concerns are application-specific in the case of components and application-independent in the case of connectors [1].

II. Approach

Figure 1 depicts the key aspects of our approach. First, we extract system concerns and structural information from source code. Then, we use this information to obtain system *bricks*, which are either components or connectors. In a parallel step, concerns are classified into *application-specific* (i.e., those related to components, their functionality, and their data) and *application-independent* (i.e., those related to connectors and the integration and interaction services they provide). Finally, we use these concerns to classify each recovered brick as a component or connector. We discuss each of these steps in the remainder of the section.

A. Obtaining Concerns through Probabilistic Topic Models

To obtain concerns, in this work we leverage a statistical language model used in information retrieval called Latent Dirichlet Allocation (LDA) [13]. Information retrieval techniques have been used by software engineering researchers [14]–[16], but not for the purpose of architectural recovery. LDA

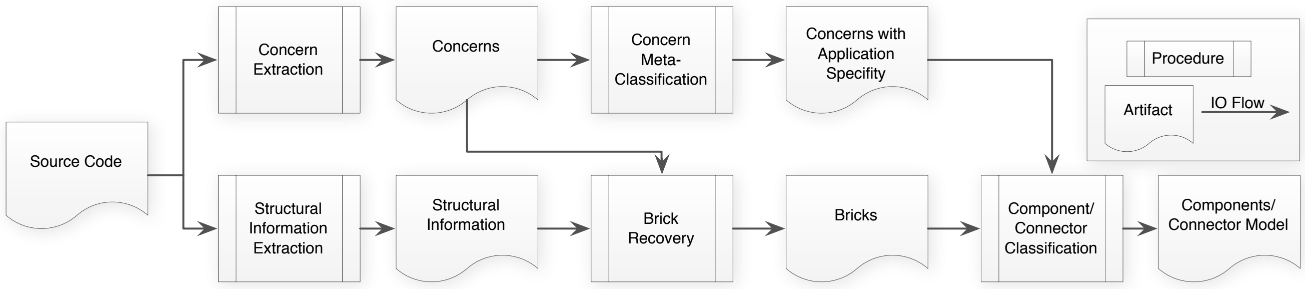
Source Code
Concern Extraction
Concerns
Concern Meta-Classification
Concerns with Application Specifity
Structural Information Extraction
Structural Information
Brick Recovery
Bricks
Component/ Connector Classification
Components/ Connector Model
Procedure
Artifact
IO Flow

Fig. 1. Overall approach for recovering components and connectors

TABLE I
A TOPIC ABOUT EVENTS FROM EVENT-BASED SYSTEMS AND ANOTHER TOPIC ABOUT WEATHER. BOTH OF THESE TOPICS APPEAR IN CLASS 1.

(a) A weather topic

Topic: Weather	
Word	Prob.
temperature	0.7
wind	0.1
humidity	0.2

(b) An event topic

Topic: Event	
Word	Prob.
send	0.2
receive	0.3
event	0.5

(c) A class document

Document: Class 1	
Topic	Prob.
Event	0.1
Weather	0.9

allows us to compute similarity measures between concerns and identify which concerns appear in a single software entity, such as a class, function, package, etc.

In LDA, we represent a software system as a set of *documents* called a *corpus*. A document is represented as a bag of words, which are identifiers and comments in the source code of a software entity. A document can have different topics, which are the concerns in our approach. A *topic* z is a multinomial probability distribution over words w drawn from a Dirichlet distribution with shape parameter β (further clarified below). Table I(a) shows an example of a topic labeled "Weather," in which words such as "temperature," "wind," and "humidity" appear with certain probabilities. LDA allows us to represent concerns in a human-readable form since a topic's meaning can be ascertained by examining its most probable words.

Consequently, a document d is represented as a multinomial probability distribution over topics z (called the document-topic distribution) drawn from a Dirichlet distribution with shape parameter α (further clarified below). A document in our work is a class. For example, "Class 1" in Table I(c) is a document that has two topics that occur with certain probabilities. These topics represent two concerns and the degree to which they are addressed in the class.

Topics are extracted using approximate iterative algorithms that maximize the likelihood estimates of the topic distribution and document-topic distribution [17]. To extract topics, the corpus and number of topics T to extract are required as input. The parameters to the distribution are set to $\alpha = 50/T$ and $\beta = .01$, because they have been shown to work well across different corpora [18].

TABLE II
EXAMPLE CLASSES WITH THEIR FEATURES.

	Structural		Concerns		
Name	Region	AbsractImpl	Strategy	Weather	Events
WAnalyzer	1	1	0	0.9	0.1
SAnalyzer	1	1	0.8	0.1	0.1
SMonitor	1	1	0.8	0.1	0.1

B. Brick Recovery

Next, we recover bricks using clustering techniques to group together implementation entities. We use both structural information (dependency relationships between software entities) and concerns (represented as topics) as *features* that determine whether or not a software entity belongs in a cluster. Features are distinctive properties of software entities. For example, Table II shows the set of structural and concern-oriented features for three implementation-level classes. The structural features are boolean: 1 indicates an existing dependency, and 0 indicates no dependency. In Table II, all three classes depend on the Region and AbstractImpl classes. The concern-oriented features each have a proportion value associated with each class.

A recovery technique based solely on structural information would cluster all three classes together or arbitrarily choose to create clusters out of the three classes. However, since the concerns reveal that the SAnalyzer and SMonitor are related to the Strategy concern, our approach would cluster those two classes together and avoid clustering the WAnalyzer with either of those two classes. This example illustrates how the correctness of clustering is dependent on the selection of features of the entities to be clustered.

The correctness of the clustering is also dependent on the similarity measure chosen to determine which entities belong in the same cluster [5]. We choose a similarity measure that takes both concerns and structural information into account. For structural information, we rely on state-of-the-art clustering techniques since each technique tends to utilize a different structural similarity measure. Similarity measures for concerns need to compare probability distributions, i.e., our representations of concerns. Therefore, we take similarity measures that compute the distance between probability distributions [19], and then combine that measure with the structural information measure.

TABLE III
EXAMPLE CONCERNS LABELED AS APPLICATION-SPECIFIC OR APPLICATION-INDEPENDENT AND ONE UNLABELED CONCERN.

Concern 1		Concern 2	
Label: app-indep		Label: app-indep	
word	prob.	word	prob.
event	.095	buffer	.054
port	.087	message	.053
request	.076	**listen**	.041
send	.055	**connection**	.015
receive	.054	session	.013

Concern 3		Concern 4	
Label: app-specific		Label: ?	
word	prob.	word	prob.
temperature	.025	**connection**	.056
wind	.021	**request**	.051
humidity	.013	pool	.050
pressure	.012	**send**	.047
request	.009	**listen**	.024

C. Concern Meta-Classification

In order to determine whether a brick implements a component or connector, we must determine whether each concern is application-specific or application-independent. We automate this classification, which we call *concern meta-classification*, through the use of supervised learning techniques because they allow us to create a function that can classify concerns.

In order to obtain correct classifications in supervised learning, it is important to choose the most relevant features to represent concerns. The key features of concerns are the different words of a concern, where some words denote application-independent concerns, while others denote application-specific concerns. For example, Table III shows four concerns: two application-independent concerns, one application-specific concern, and one concern that is not yet labeled as application-specific or application-independent. For each concern, we show the top 5 words and their probability values. Bolded words are words that appear in the unlabeled concern and one of the labeled concerns.

We choose the k-nearest neighbor algorithm for classifying concerns. First, a set of correctly labeled concerns must be provided as input to the k-nearest neighbor algorithm, along with a set of concerns that are not yet labeled. For every unlabeled concern, the algorithm computes the similarity between an unlabeled concern and labeled concern, and gives the unlabeled concern the label that is among the majority of the k most similar concerns to the unlabeled concern.

For example, consider Concern 4 in Table III. To determine similarity between concerns, we choose the similarity measure used in brick recovery. Using that measure, closer probabilities of words shared between concerns result in greater similarity between those concerns. Only the word "request" is shared between Concern 3 and Concern 1. Concerns 1 and 2 both share two high probability words with Concern 4. In this example, we let $k = 3$. Therefore, the similarity measure would determine that Concern 4 is most similar to Concerns 1 and 2, resulting in Concern 4 being labeled application-independent.

D. Brick Classification

We can determine whether a brick implements a component or connector using supervised learning. To enable correct brick classification, we must focus on the selection of relevant features for bricks; we select the following features: (a) labeled concerns of the brick, (b) usage of connector-implementing libraries, and (c) the brick's involvement in design patterns that provide interaction services.

Three different intuitions about distinguishing components from connectors inform our selection of features. First, since concerns of entities in a brick occur with certain probabilities, we can determine whether the concerns in a brick are primarily application-independent or application-specific. For example, Table IV depicts three bricks, two that are application-specific and one that is application-independent. Second, if a brick extensively uses a library that can be used to implement higher-order connectors, such as a socket or datagram library, then the brick likely implements a connector, which is an idea we have used before for individual classes rather than bricks [20]. In Table IV, Brick 1 uses a socket library 15 times, while Brick 3 uses a datagram library 20 times. Lastly, certain design patterns are more relevant to interaction needs than others. For example, the Adaptor/Wrapper, Proxy, Chain of Responsibility, Mediator and Observer design patterns provide interaction services between programming language-level objects, which we call *connector-oriented design patterns*. Therefore, their existence in a brick indicates that the brick is more likely to be implementing a connector. In Table IV, MediatorCount and ObserverCount count the number of classes in a brick that participate in any mediator or observer patterns, respectively. As the number of classes within a Brick that are involved in such design patterns grows, so does the probability that the Brick implements a connector. Note that thresholds for determining when a combination of features result in one labeling or another is determined by the supervised learning algorithms. We will need to determine the correctness of different supervised learning algorithms in our context empirically.

Table IV illustrates how a classifier produced by a learning algorithm may distinguish between components and connectors. Brick 1 is dominated by application-independent concerns, exhibits high usage of socket or datagram libraries, and has some classes that implement connector-oriented design patterns. Therefore, Brick 1 is labeled as a connector. Brick 2 is labeled a component because it is dominated by application-specific concerns, does not consist of any connector-oriented design patterns, and exhibits low usage of socket or datagram libraries. However, even though Brick 3 is dominated by application-specific concerns, it utilizes socket and datagram libraries extensively and consists of a significant number of classes that implement design patterns resulting in it being labeled as a connector. A remaining issue that requires additional experimentation is discovering less clear-cut cases for

TABLE IV
EXAMPLE BRICKS WITH THEIR FEATURES AND LABELS.

	Primary Concern Type	Socket Usage	Datagram Usage	MediatorCount	ObserverCount	Label
Brick 1	Application-Independent	15	0	2	0	Connector
Brick 2	Application-Specific	4	0	0	0	Component
Brick 3	Application-Specific	15	20	4	2	Connector

brick classification, such as classification of Brick 3, and how different learning algorithms deal with those cases.

III. CURRENT STATUS AND FUTURE WORK

We use the MALLET [21], Weka [22], and Soot [23] frameworks for concern extraction, supervised learning, and structural information extraction, respectively. For brick recovery, we are currently testing different similarity measures that can be used for combining concerns and structural information. We have tested concern extraction on multiple systems, including grid-based, event-based, and chat server systems. To assess the usefulness of supervised learning, we have performed component/connector classification using learning at the class-level. The results have been promising and have informed our selection of features in Section II-D. In the near future, we will empirically evaluate our approach on multiple systems and use our approach's resulting component/connector model to help automatically detect and pinpoint sources of architectural degradation.

REFERENCES

[1] R. Taylor, N. Medvidovic, and E. Dashofy, "Software Architecture: Foundations, Theory, and Practice," 2009.

[2] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE TSE*, vol. 35, no. 4, pp. 573–591, 2009.

[3] N. Anquetil and T. Lethbridge, "Recovering software architecture from the names of source files," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 3, pp. 201–221, 1999.

[4] R. Koschke, "Atomic architectural component recovery for understanding and evolution," Ph.D. dissertation, University of Stuttgart, 2000.

[5] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE TSE*, vol. 33, pp. 759–780, 2007.

[6] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE TSE*, vol. 31, pp. 150–165, 2005.

[7] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE TSE*, vol. 32, pp. 193–208, 2006.

[8] V. Tzerpos and R. Holt, "ACDC: an algorithm for comprehension-driven clustering," in *7th WCRE*, 2000, pp. 258 –267.

[9] R. Fiutem, P. Tonella, G. Anteniol, and E. Merlo, "A cliche-based environment to support architectural reverse engineering," in *3rd WCRE*, 2002, pp. 277–286.

[10] D. Harris, H. Reubenstein, and A. Yeh, "Recognizers for extracting architectural features from source code," *Reverse Engineering, Working Conference on*, vol. 0, p. 252, 1995.

[11] N. C. Mendonça and J. Kramer, "An approach for recovering distributed system architectures," *Automated Software Engineering*, vol. 8, no. 3, pp. 311–354, 2001.

[12] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE TSE*, pp. 454–466, 2006.

[13] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[14] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th ICSE*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[15] A. Kuhn, S. Ducasse, and T. Grba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[16] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ICSE*, 2010, pp. 95–104.

[17] T. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. Suppl 1, p. 5228, 2004.

[18] M. Steyvers and T. Griffiths, "Probabilistic topic models," *Handbook of latent semantic analysis*, vol. 427, 2007.

[19] J. Lin, "Divergence measures based on the Shannon entropy," *Information Theory, IEEE Transactions on*, vol. 37, no. 1, pp. 145–151, 1991.

[20] V. Jakobac, N. Medvidovic, and A. Egyed, "Separating architectural concerns to ease program understanding," in *Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*. ACM, 2005, pp. 1–5.

[21] A. McCallum, "Mallet: A machine learning for language toolkit," 2002.

[22] G. Holmes, A. Donkin, and I. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 1994, pp. 357–361.

[23] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, and E. Gagnon, "P. Co. Soot-a Java optimization framework," in *Proceedings of CASCON 1999*, 1999, pp. 125–135.