A Framework for Obtaining the Ground-Truth in Architectural Recovery

Joshua Garcia, Ivo Krka, and Nenad Medvidovic
Computer Science Department,
University of Southern California,
Los Angeles, CA 90089, USA
{*joshuaga,krka,neno*}*@usc.edu*

Chris Douglas
Yahoo! Labs, USA
chrisdo@yahoo-inc.com

***Abstract*—Architectural recovery techniques analyze a software system's implementation-level artifacts to suggest its likely architecture. However, different techniques will often suggest different architectures for the same system, making it difficult to interpret these results and determine the best technique without significant human intervention. Researchers have tried to assess the quality of recovery techniques by comparing their results with *authoritative* recoveries: meticulous, labor-intensive recoveries of existing well-known systems in which one or more engineers is integrally involved. However, these engineers are usually not a system's original architects or even developers. This carries the risk that the authoritative recoveries may miss domain-, application-, and system context-specific information. To deal with this problem, we propose a framework comprising a set of principles and a process for recovering a system's *ground-truth* architecture. The proposed recovery process ensures the accuracy of the obtained architecture by involving a given system's architect or engineer in a limited, but critical fashion. The application of our work has the potential to establish a set of "ground truths" for assessing existing and new architectural recovery techniques. We illustrate the framework on a case study involving Apache Hadoop.**

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* [12], which are caused by careless, unintended addition, removal, and/or modification of architectural design decisions. To deal with drift and erosion, a number of techniques have been proposed to help recover a system's architecture from its implementation [7]. However, despite the advances in the area, existing techniques often return different results as "the architecture" for the same system and are known to suffer from inaccuracies, which makes it difficult for a practitioner to assess a recovery technique.

Different objective criteria have previously been used for evaluating recovery techniques [14], [10]. The most comprehensive criterion is a comparison of the result of a recovery technique with an *authoritative recovery*. An authoritative recovery is the mapping, from architectural elements to code-level entities that implement those elements, that has been constructed or certified as correct by a human. Obtaining reliable authoritative recoveries remains challenging for non-trivial systems. Specifically, an authoritative recovery performed by an engineer who is not an expert for a particular system may not reflect the true architecture as viewed by the system's architect. This runs the risk of missing application-level or domain-level design decisions, which runs the risk of wasting significant effort.

A more reliable authoritative recovery is one that, after construction, is certified by an architect or engineer of the software system. More specifically, the engineer is a long-term contributor with intimate knowledge of the system's architecture, or *long-term contributor* for short. We refer to such an authoritative recovery as a *ground-truth recovery*. Typically, only a long-term contributor or architect of a system has sufficiently comprehensive domain knowledge, application-specific knowledge, and experience that may not be present in system documentation or implementation-level artifacts. However, there is a dearth of ground-truth recoveries due to the colossal effort required for an engineer to produce such a recovery. The ground-truth recoveries that are available are often of smaller systems, which reduce confidence that a recovery technique would work on larger systems, or only identify a few very coarse-grained components in larger systems, rendering the obtained recovery less informative.

In this paper, we propose a framework intended to aid the recovery of ground-truth architectures. The framework defines a set of principles and a process that results in a reliable ground-truth recovery. The process involves an architect or long-term contributor of the system in a limited yet meaningful way. The framework's principles, referred to as *mapping principles*, serve as rules or guidelines for grouping code-level entities into architectural elements and for identifying the elements' interfaces. The framework bases these principles on four types of information used to obtain ground truth: *generic information* (e.g., system-module dependencies), *domain information* (e.g., architectural-style rules), *application information* (e.g., the purpose of the source code elements), and information about the *system context* (e.g., the programming language used).

We focus our discussion around a case study in which we applied our framework to produce a ground-truth recovery of Apache Hadoop [1], a widely-used framework for distributed processing of large datasets across clusters of computers [3]. The final product of this case study is a ground-truth

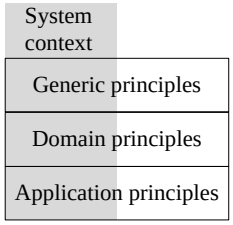


Figure 1. Classification of the principles used for ground-truth recovery.

recovery constructed with the aid of one of Hadoop's long-term contributors.

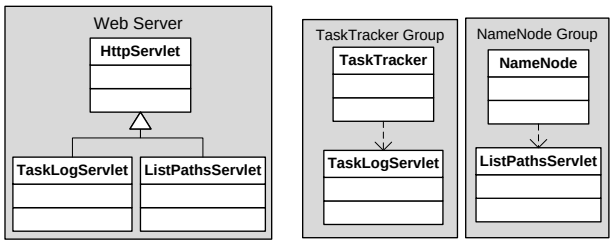
II. Mapping Principles

In this section, we present the types of mapping principles that we use in our framework for obtaining ground-truth recoveries. A key aspect of our framework is the identification of these mapping principles. They underlie the process for obtaining ground-truth recoveries we present in Section III. Figure 1 depicts how the different kinds of principles relate to each other. Next, we define the mapping principles and illustrate them with examples from the Hadoop case study.

Generic principles are independent of domain and application information. These principles include long-standing software-engineering principles often used to automate architectural recovery techniques, such as separation of concerns, isolation of change, and coupling and cohesion. For example, a generic principle may state that code-level modules that depend on common sets of code-level entities (e.g., interfaces or libraries) should be grouped together. In the case of Hadoop, this principle results in grouping code-level entities that involve sorting since they depend on interfaces involving reporting, comparing, and swapping. The generic principles are typically concerned only with the code-level entities and dependencies between them, and do not consider domain and application semantics. Such principles are typically implemented by and can be obtained from existing recovery techniques. In our case study, we used the rules from Focus [11], a recovery technique we have previously proposed, as the generic principles.

Domain principles are the mapping principles for recovering an architecture that are based on domain information. Domain information can be any data specific to the domain of the system in question, e.g., telecommunications, avionics, scientific systems, distributed data systems, robotics, etc. The domain principles stem from (1) the knowledge about a specific domain documented in the research literature and industry standards, and (2) an engineer's experience working in a particular domain. Figure 1 indicates that domain principles are at a level of abstraction below generic principles and above application principles. Architectural pattern rules are examples of domain principles.

The domain principles we applied on Hadoop were extracted from an existing reference architecture of grid systems that defines the typical components in such systems



(a) Servlets merged into web server (b) Servlets separated according to use by domain components

Figure 2. Different ways of applying mapping principles.

[8], and our knowledge of the domain of distributed data systems (master/slave architectural pattern, distributed communication). For example, the master/slave pattern, where the master manages jobs that are distributed among slaves, is commonly used in systems where massive amounts of data are stored and processed. Hadoop has a JobTracker that serves as a master that distributes jobs among slaves, called TaskTracker components in Hadoop. Hence, we introduced a domain principle dp_{tr} that groups together the code-level entities that primarily implement the master (JobTracker) and slave (TaskTracker) functionalities.

Application principles are mapping principles that are relevant specifically to the particular application being recovered. For example, Hadoop is able to perform distributed upgrades. Therefore, an application principle that utilizes this information is a rule stating that any objects that are used primarily for performing distributed upgrades should be grouped together. Application principles may be obtained from available documentation, comments in the code, or feedback from the system's long-term contributor or architect.

It is important to note that the different types of principles may suggest different groupings. Since application principles contain more specific information than either domain or generic principles, they have the highest priority, while the generic principles have the lowest priority. A case where application principles are applied is when an application utilizes components outside of the application's primary domain. For example, a distributed data system may choose to utilize web system components to perform certain functionalities. Specifically, Hadoop uses servlets to allow transfer of data over HTTP. Servlets are Java classes that respond to requests by using a request-response protocol and are often used as part of web servers. A domain principle dp_{ws} may indicate that web servlets should be grouped together to form a web server. Figure 2(a) shows a grouping of servlet classes based on dp_{ws}. However, for Hadoop, an application principle ap_{ws} representing this information states that servlets should be grouped with domain components, such as the JobTracker and TaskTracker. Thus, ap_{ws} supersedes dp_{ws}. For example, Figure 2(b) shows that the TaskTracker is dependent on a servlet for task logging,

which constitutes one group, while the NameNode, which is a domain component that manages the namespace of Hadoop's distributed filesystem, is dependent on a servlet that obtains meta-data about a file system, i.e., the ListPathsServlet, which constitutes another group.

Lastly, *system context principles*, represented in Figure 1 as the gray area spanning the other three categories of principles, are mapping principles that involve properties about the infrastructure used to build the application being recovered. For example, the choice of OS or programming language can impart useful recovery information. The choice of OS implies differences in the way components are configured, installed, deployed, run, or compiled. Therefore, the OS over which an application is run may significantly affect the way code-level entities map to components, or even what kinds of components may exist in the first place. A system context principle can be a generic, domain, or application principle.

III. Ground-Truth Recovery Process

In this section, we detail the process for deriving ground-truth recoveries based on the mapping principles described in Section II. In the case of Hadoop 0.19.0, this process resulted in the ground-truth architecture comprising 70 components, whose configuration forms an almost-complete graph. For reference, recovered diagrams of Hadoop can be viewed at full magnification at [5].

The idea behind the ground-truth recovery process is to utilize the different types of available information and the corresponding mapping principles to obtain an informed authoritative recovery. The authoritative recovery is then analyzed by a long-term contributor or architect of the system, and revised into a ground-truth based on the suggested modifications. Our process is unique in the way it (1) provides a set of well-defined steps that can help ensure consistency when the recoveries are performed by different researchers, (2) results in more informed intermediate recoveries due to the multiple types of utilized information and mapping principles, and (3) ultimately yields a ground-truth recovery. Few recovery techniques take human expertise explicitly into account [7]. Our process specifically emphasizes the need for a (1) long-term contributor or architect that certifies and recommends changes, referred to as the *certifier* and (2) an engineer producing the authoritative recovery to be certified, referred to as the *recoverer*. Therefore, our process minimizes the burden on the certifier while optimizing the use of the certifier's time and expertise.

The process we propose consists of eight steps depicted in Figure 3. The initial two steps deal with the identification and selection of the specific mapping principles that will be applied during the recovery. In Steps 3 and 4, the generic principles are applied on the extracted implementation-level dependency information. In Step 5, the available domain and application principles are used to refine the generic recovery. In Step 6, the utility components, whose primary purpose is to provide helper functions (e.g., sorting, parsing, GUI rendering) to other components, are identified. In the final two steps, the long-term contributor or architect validates the obtained authoritative recovery, which is then refined into a ground-truth recovery.

In our process, the certifier verifies and recommends changes (Steps 7–8) to an authoritative recovery performed by one or more recoverers (Steps 1–6). A certifier is not extensively involved in the recovery, but instead is asked only to comment on and verify the correctness of the authoritative recovery obtained by the recoverers. For example, a certifier can impart domain and application principles not evident in the documentation. Next, we describe the individual steps of our process and illustrate them using the Hadoop case study.

1) Gather domain and application information: The first step in our process is for a recoverer to utilize any available documentation to determine domain or application information that can be used to produce domain or application principles. From our experience, it is common for systems to contain some kind of documentation about components and connectors. We refer to these as *expected components and connectors*. The documents often include diagrams showing some components and their configurations. They also tend to contain explanations of each component's key functionalities. Therefore, domain and application principles, would indicate that some code-level entities should map to these components. Furthermore, documentation should provide information about system context information. Depending on the availability of a certifier, the domain and application principles may subsequently be refined according to the certifier's suggestions.

2) Select a generic recovery technique: The recoverers can select an existing recovery technique to aid in the production of an authoritative recovery. The use of a recovery technique injects generic principles into the recovery. For example, many recovery techniques will use different metrics or rules to represent generic principles, such as coupling, cohesion, or separation of concerns. Latter steps in this process work toward eliminating any bias that may remain from selecting a particular recovery technique ahead of time. In particular, use of domain and application information, manual modification to the output of the selected recovery technique, modifications made according to the certifier, and final verification of the authoritative recovery by the certifier will work toward eliminating any such bias. Therefore, many existing recovery techniques are appropriate for this step. However, for larger systems, automated techniques that work at larger granularities (e.g., that recover classes instead of functions, or files instead of classes) may be more appropriate than more manual techniques. Furthermore, the selected recovery technique may also depend on the recoverer's prior experience (e.g., a recoverer may prefer a recovery technique she helped to develop). In the Hadoop case study, we used

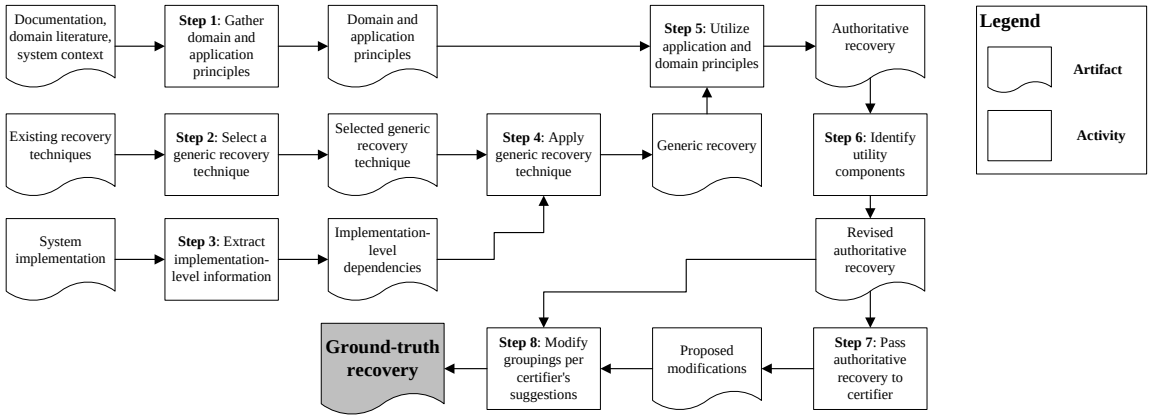


Figure 3. The process for obtaining ground-truth recoveries.

Focus [11] since we have developed it in-house and tested it extensively on systems from various domains.

3) Extract relevant implementation-level information: Architectural recovery often involves automatic extraction of implementation-level information. The information extracted will depend on the recovery technique selected by the recoverer and the available tool support. This extracted information typically includes at least the structural dependencies between the code-level entities. For Hadoop, we used IBM RSA [2] and the Class Dependency Analyzer [4] as tool support.

4) Apply generic recovery technique: At this point, the recoverers can apply their chosen generic recovery technique to obtain an initial recovery. This step is a straightforward application of the recovery technique since more specific information will be injected through domain, application, and system context principles in later steps.

5) Utilize domain and application principles: Any mapping principles determined in Step 1 can be used to modify the recovery obtained at the previous step. In particular, any groupings obtained should be modified, if possible, to resemble expected components or connectors. This is the first step in the process to make use of domain or application information in the recovery.

Domain and application principles had a significant effect on the components identified in Hadoop. For example, the rule of Focus that indicates that classes or groupings with two-way associations should be merged (gp_{tw}) would have affected 60 out of the 65 components that we identified for Hadoop 0.19.0. In particular, these 60 components would have been merged with one or more of the other identified components. Domain and application principles prevented these erroneous mergers. For example, the groupings for the NameNode and DataNode would be merged according to gp_{tw}, but a domain principle stated that domain components should never be grouped prevented this merger.

6) Identify utility components: The recoverer should identify any utility components (e.g., libraries, middleware packages, application framework classes). Previous empirical evaluations of recovery techniques that use authoritative recoveries [6], [10] have shown that proper identification of utility components has been a major factor in determining the correctness of recovery techniques. The generic principle for identifying utility components is to select groupings that have higher numbers of incoming than outgoing structural dependencies. Furthermore, utility components typically do not resemble the expected application components. Utility components may be split apart if the certifier in latter steps decides that code-level entities in utility components belong more appropriately to different groupings.

Our initial recovery of Hadoop consisted of 21 utility components. Most of the utility components originated from a particular directory of Hadoop called *core*. This directory contains about 280 classes that can be mapped to 14 components based on mapping principles involving the Java package structure (e.g., the main metrics-handling utilities comprise the classes and subpackages of the org.apache.hadoop.metrics package).

7) Pass authoritative recovery to certifier: At this point, the recoverer has produced an authoritative recovery that has been enriched and modified with the help of different kinds of mapping principles. Key to the authoritative recovery is the clear relationship between the proposed architectural elements and the code-level elements. The recovered architecture is only now passed to a certifier for verification. The certifier looks through the proposed groupings and suggests (1) addition of new groupings, (2) splitting of existing groupings, or (3) transfer of code-level entities from one grouping to another. These modifications should also include rationale behind the change in groupings in order to determine new domain or application information. In turn, this new information may reveal new mapping principles that can be used by the recoverer to further modify the obtained recovery. The certifier's rationale may also reveal the choice not to follow generic principles in certain cases.

8) Modify groupings according to certifier's suggestions: At this point, the recoverer modifies the groupings as suggested by the certifier. The recoverer and certifier may repeat

steps 7 and 8 until both are satisfied with the obtained recovery, at which point a ground-truth recovery is finalized.

Our certifier for Hadoop was intimately familiar with over 90% of the components (59 of 65 components) we identified in our authoritative recovery. Therefore, he was able to verify with confidence the existence and appropriate grouping of a large majority of the components. The certifier deemed 34 components to be appropriate as-is. However, he identified the need for close to 40% of the components (25 of 65 components) to be regrouped or modified in some fashion. This is only a single data point, but it is indicative of the risks of halting the architectural recovery process at the point at which an authoritative recovery is available. The recoverers carefully studied several sources of existing information, but still misinterpreted or completely missed certain key architectural details.

IV. Discussion and Future Work

We have highlighted several findings in the Hadoop case study that point to possible directions for future work. These findings relate to (1) groupings based on semantics as compared to structural dependencies, (2) utility components, (3) the extent to which package structure represents or indicates components, (4) the importance of domain and application principles, and (5) a repository of ground-truth architectures for assessment of architectural recovery techniques.

In many cases, our certifier prescribed groupings based on the semantics of classes while disregarding the structural dependencies between those classes. Previous work has hinted at the importance of focusing on semantics when recovering an architecture [9], [11], [13]. The Hadoop case study provides more evidence to back this intuition.

Hadoop's utility components contribute extensively to producing a structural view of the architecture that constitutes a near-complete graph. Further studies that examine the extent to which utility components result in these "noisy" architectures can help produce more understandable views of architectural recoveries. Specifically, researchers should consider ways to automatically identify utility components, and to support the creation of, both, utility-free and utility-rich views of the recovery.

For 64% of the components (45 of 70), implementation-level package structure did not match the final ground-truth groupings. While this is consistent with the intuition that packages often do not correspond to architectural elements, we did find that they help guide a recoverer in identifying architectural elements by providing a useful first approximation. Furthermore, in the case of Hadoop 0.19.0, the certifier suggested breaking up certain groupings in the authoritative recovery according to the refactorings performed on the implementation package structure in the subsequent versions of Hadoop. This suggests another avenue of future research: identification of components and their evolution based on package history across versions.

Despite our recoverers' domain experience that helped to obtain the authoritative recovery, application principles still had a significant impact on constructing the ground-truth recovery of Hadoop. Future studies should examine the extent to which domain principles and application principles affect architectural recovery. This is important because existing techniques are inaccurate at least to some extent due to a lack of consideration for domain and application principles. Obtaining such information in an automated fashion is a major challenge for future recovery techniques.

The main contributions of this paper are (1) identification of the different types of mapping principles, and (2) a process for reliably obtaining a ground-truth recovery. While these contributions should help researchers evaluate their recovery techniques in a more structured and rigorous manner, we believe that the benefits of this process may more broadly benefit the communities of researchers and practitioners. Specifically, we intend to use the proposed process as the basis of an open-source repository of ground-truth recoveries. This repository would (1) facilitate stronger and more informative evaluation of new recovery techniques, (2) serve as a basis for comparing the existing recovery techniques, and (3) provide a platform for researching the underlying principles of what constitutes a good architectural recovery.

References

[1] (2012) Apache™Hadoop™. [Online]. Available: http://hadoop.apache.org/

[2] (2012) IBM Rational Software Architect. [Online]. Available: http://www.ibm.com/developerworks/rational/products/rsa/

[3] (2012) Poweredby - Hadoop Wiki. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy

[4] (2012) Programmer's Friend - Class Dependency Analyzer. [Online]. Available: http://www.dependency-analyzer.org/

[5] (2012) recoveries:hadoop_0.19.0 [USC Softarch Wiki]. [Online]. Available: http://softarch.usc.edu/wiki/doku.php?id=recoveries:hadoop_0.19.0

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

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

[8] I. Foster *et al.*, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, 2001.

[9] R. Koschke, "What Architects Should Know About Reverse Engineering and Rengineering," in *WICSA*, 2005.

[10] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE TSE*, vol. 33, 2007.

[11] N. Medvidovic and V. Jakobac, "Using software evolution to focus architectural recovery," *Automated Software Engineering*, vol. 13, no. 2, 2006.

[12] R. Taylor *et al.*, "Software Architecture: Foundations, Theory, and Practice," 2009.

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

[14] J. Wu *et al.*, "Comparison of clustering algorithms in the context of software evolution," in *IEEE ICSM*, 2005.