Obtaining Ground-Truth Software Architectures

Joshua Garcia*, Ivo Krka*, Chris Mattmann*†, Nenad Medvidovic*

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

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

Abstract—**Undocumented evolution of a software system and its underlying architecture drives the need for the architecture's recovery from the system's implementation-level artifacts. While a number of recovery techniques have been proposed, they suffer from known inaccuracies. Furthermore, these techniques are difficult to evaluate due to a lack of "ground-truth" architectures that are known to be accurate. To address this problem, we argue for establishing a suite of ground-truth architectures, using a recovery framework proposed in our recent work. This framework considers domain-, application-, and context-specific information about a system, and addresses an inherent obstacle in establishing a ground-truth architecture — the limited availability of engineers who are closely familiar with the system in question. In this paper, we present our experience in recovering the ground-truth architectures of four open-source systems. We discuss the primary insights gained in the process, analyze the characteristics of the obtained ground-truth architectures, and reflect on the involvement of the systems' engineers in a limited but critical fashion. Our findings suggest the practical feasibility of obtaining ground-truth architectures for large systems and encourage future efforts directed at establishing a large scale repository of such architectures.**

I. Introduction

The effort and cost of software maintenance dominate the activities in a software system's lifecycle. Understanding and updating a system's software architecture is arguably the most critical activity within that lifecycle. However, the maintenance of an architecture is exacerbated by the related phenomena of architectural *drift* and *erosion* [31]. These phenomena are caused by careless, unintended addition, removal, and/or modification of architectural design decisions. To deal with drift and erosion, sooner or later engineers are forced to *recover* a system's architecture from its implementation. A number of techniques have been proposed to aid architecture recovery [19]. However, existing techniques are known to suffer from inaccuracies and typically return different results as "the architecture" for the same system. In turn, this can lead to (1) difficulties in assessing a recovery technique, (2) risks in relying on a given technique, and (3) flawed strategies for improving a technique.

This paper argues for establishing a set of "ground truths" that can be used to deal with these problems. In this context, a ground truth is the architecture of a software system that has been verified as accurate by the system's architects or developers who have intimate knowledge of the underlying application and problem domain. Such knowledge is often undocumented and thus less likely to be known to engineers who were not involved in constructing the system. There are examples in the literature of researchers who had a similar motivation to ours and who had extensively studied and documented the architectures of existing applications, but without the involvement of the applications' own engineers (e.g., [12], [27]). We refer to such architectures as *authoritative*. Authoritative architectures run the risk of (1) omitting certain architectural design decisions, (2) including spurious ones, and (3) recording yet others incorrectly. This is the very problem we try to remedy.

Obtaining ground-truth architectures must overcome two important obstacles, however. First, producing an accurate architectural design of a system is likely to be an arduous and time-consuming task; it is not reasonable to expect that the system's engineers will be able or willing to take considerable time away from their daily obligations to do this on our behalf. Second, a system's engineers are likely to have an incomplete, incorrect, and/or idealized picture of the system's architecture. To address these two obstacles, we have recently proposed an approach [20] that minimizes the burden on the system's engineers of producing a ground-truth architecture, while ensuring the architecture's accuracy. To deal with the latter problem, we first recover a preliminary version of a system's architecture from its implementation. Then, to deal with the former problem, we involve the system's engineers in a critical, but limited and carefully controlled, step in completing the recovery.

In this paper, we discuss our findings in obtaining the ground-truth architectures for four existing systems. The systems in our study come from several problem domains, including large-scale data-intensive computing; architectural modeling and analysis; and operating system command-line shells. These software systems have been used and maintained for years, are written in Java or C, and range from 70 KSLOC to 280 KSLOC. For each system, we had access to one or two of its architects or key developers. The variety of the systems allowed us to form some general insights about obtaining ground-truth architectures. We also discuss our experience and lessons learned in enlisting the help of the systems' engineers.

This work has resulted in several observations. Perhaps most surprisingly, our experience with the four ground-truth recovery efforts (as well as a fifth such effort currently under way involving Google's Chromium [3]) contradicts

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

the conventional wisdom, and our initial concern, that a system's developers will be unwilling to dedicate time to a recovery effort unless it is initiated and mandated by their organization. Another commonly stated view is that a system's architecture is reflected in its package and directory structure. In general, our data invalidated this premise. Despite the four systems' many differences—application domain, size, implementation language, etc.—we have identified several commonalities in their ground-truth architectures in terms of component sizes and types. These commonalities can help frame the expectations of a given architecture recovery technique and its accuracy. We have also found that focusing on a system's *semantics* yields a recovered architecture that matches the engineers' intuition more closely than the existing recovery techniques' predominant focus on system *structure*. Finally, certain discrepancies between a system's documented architecture and its ground-truth architecture recovered from the implementation were recognized as examples of drift and erosion. However, other such discrepancies seem to be considered acceptable and even expected by engineers. The difference in abstraction levels explains some but not all such cases. This has opened up some interesting questions about architecture recovery, and more broadly, about software architecture itself.

The paper is organized as follows. Section II presents the background of this work as well as related approaches. Section III discusses the details of the four systems whose ground-truth architectures we have recovered to date. Section IV presents the lessons learned. Section V concludes the paper.

II. Foundations

To position the work presented in this paper, we first discuss the relevant related work (Section II-A) and then overview our recently proposed framework for recovering ground-truth architectures [20] (Section II-B).

A. Related Work

The research literature in the area of architecture recovery contains several examples of manually extracted architectural information, authoritative architectures, and ground-truth architectures, all of which were used to to evaluate recovery techniques. Due to the scope of this paper, we do not discuss the existing (semi-)automated architecture recovery techniques, which are surveyed in [19], [23], [25].

Several studies have attempted to re-document and recover the high-level architectures of widely-used systems, including the Linux kernel [12], the Apache Web Server [21], and a portion of Hadoop [9]. In another set of studies [8], [10], [22], [33], [34], researchers evaluate their proposed recovery techniques using authoritative architecture recoveries of several additional systems. Since none of these were *ground-truth* architectures, it is likely that they suffered from inaccuracies. Furthermore, some of the above studies captured only very high-level architectural views, or erroneously conflated implementation packages with architectural components.

Ground-truth architectures have been used to evaluate four recently proposed recovery techniques [11], [14], [15], [24] that extend Murphy et al.'s reflexion models [29]. Christl and Koschke [14] use a ground-truth recovery of SHriMP [35], a tool for visualizing graphs comprising about 300 classes. An update to this technique [15] was evaluated on two additional ground-truth architectures of relatively small systems (up to 32 KSLOC). Bittencourt et al. [11] used ground-truth architectures of systems with comparable sizes: Design Wizard (7 KSLOC), a tool for extracting and querying designs, and Design Suite (24 KSLOC), a tool for abstracting and visualizing designs.

To our knowledge, only two ground-truth architectures of larger systems were obtained and used to evaluate recovery techniques. OpenOffice (6 MSLOC), an open-source productivity software suite, was used by Koschke [24]. TOBEY (250 KSLOC) [8], the backend for IBM's compiler products, was used to evaluate the ACDC [33] and LIMBO [8] recovery techniques. In the case of OpenOffice, the ground-truth was very coarse-grained, making the recovered architecture less informative. For TOBEY, a ground-truth was obtained by relying exclusively on a series of interviews with developers, calling into question its accuracy. Furthermore, TOBEY is a proprietary system, which limits its use in evaluating and improving recovery techniques.

Common to all authoritative and ground-truth architecture recoveries discussed in this section is that neither the process through which they were obtained nor the effort needed to produce them were documented.

B. Framework for Ground-Truth Architecture Recovery

Our ground-truth recovery framework [20] defines a set of principles and a process that result in a reliable ground-truth architecture. The framework's principles, referred to as *mapping principles*, serve as rules or guidelines for grouping code-level entities into architectural elements. The process involves a system's engineer in a limited but meaningful way.

1) Mapping Principles: We divide the mapping principles into generic, domain, application, and system-context principles. Figure 1 depicts how the principles relate to each other. Next, we describe each category of mapping principles.

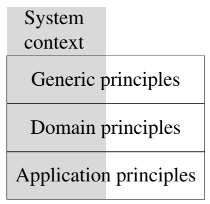


Fig. 1. Mapping principles used for ground-truth recovery.

Generic principles are long-standing software-engineering principles: separation of concerns, isolation of change, coupling, cohesion, etc. For example, a generic principle may state that code-level modules that depend on common code-level entities (e.g., interfaces) should be grouped together. These principles are typically used to automate existing architecture recovery techniques.

Domain principles are the mapping principles based on *domain information*. Domain information can be any data specific to the domain of the system in question, e.g., telecommunications, avionics, scientific computing, robotics, etc. The domain principles stem from (1) the knowledge about a domain documented in the research literature and industry standards, and (2) an engineer's experience working in a given domain. For example, a domain principle may state that the master/slave architectural pattern is commonly used in systems where massive amounts of data are stored and processed. If the reification of a domain principle and a generic principle results in a conflict in a given architecture recovery, the domain principle should override the generic principle.

Application principles are relevant specifically to the application whose architecture is undergoing recovery. Application principles may be obtained from available documentation, comments in the code, or feedback from the system's engineers. For example, one of the systems we studied is able to perform distributed upgrades. In that case, an application principle would state that any objects used primarily to perform distributed upgrades should be grouped together. If the reification of an application principle conflicts with a domain or generic principle, the application principle should override either of the other two principles.

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 analyzed. For example, the choice of operating system implies differences in the way components are configured, installed, deployed, run, or compiled, and thus may significantly affect the way code-level entities map to components.

2) Process for Ground-Truth Recovery: The ground-truth architecture recovery process aims 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 key engineer or architect of the system, and revised into a *ground-truth* based on the suggested modifications. Our process emphasizes the need for (1) the *recoverer* — an engineer producing the architecture to be certified, and (2) the *certifier* — the system's engineer or architect who approves the architecture or recommends changes to it. The ground-truth recovery process consists of eight steps, described next.

Step 1. The recoverers 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*.

Step 2. 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. Later steps in our process aim to eliminate any bias stemming from the choice of a particular technique.

Step 3. To recover an architecture, it is necessary to extract the implementation-level information as required by the selected generic recovery technique (e.g., structural dependencies between code-level entities).

Step 4. At this point, the recoverers can apply their chosen generic recovery technique to obtain an initial architecture. More specific information will be injected through domain, application, and system-context principles in later steps.

Step 5. Any mapping principles determined in Step 1 can be used to modify the architecture obtained in Step 4. In particular, any obtained groupings should be modified, if possible, to resemble expected components and connectors.

Step 6. The recoverer should identify any utility components (e.g., libraries, middleware packages, and application framework extensions) because their correct identification significantly impacts the quality of a recovered architecture [8], [25]. Steps 4–6 can be iteratively applied until the recoverer determines that the architecture has an appropriate granularity.

Step 7. At this point, the recoverer has produced an authoritative architecture recovery that has been enriched and modified with the help of different kinds of mapping principles. The recovered architecture is only now passed to a certifier. 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 principles.

Step 8. 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 and the ground-truth architecture is finalized.

III. Experience

To date, we have produced four ground-truth architectures using the above framework; a fifth recovery, of Google's Chromium [3], is currently underway. Figure 2 summarizes each *System* with which we have worked, its *Ver*sion, application *Dom*ain, primary implementation *Lang*uage, size in terms of *SLOC*, and the number of *R*ecoverers and *C*ertifiers involved in its architecture recovery. The variations in the systems' sizes, domains, and implementation languages allow us to extrapolate broader lessons about ground-truth recovery. We specifically selected the systems' versions with which the available certifiers were closely familiar. We selected Focus [28] as the architecture recovery technique; recall from Section II-B2 that our framework tries to minimize the bias of the chosen recovery technique on the recovered architecture.

In the remainder of the section, we discuss each of the four systems. For each, we illustrate the recovered ground-truth architectures, or certain portions thereof. The depicted ground-truth diagrams will be at a very low magnification by necessity. The point of showing these diagrams is not to explain

System	Ver	Dom	Lang	SLOC	R	C
Bash	1.14.4	OS Shell	C	70K	1	1
OODT	0.2	Data Middleware	Java	180K	1	1
Hadoop	0.19.0	Data Middleware	Java	200K	2	1
ArchStudio	4	Architectural IDE	Java	280K	1	2

Fig. 2. Summary information about systems recovered

their details, but instead to highlight the discrepancies between them and the systems' corresponding conceptual-architecture diagrams that were obtained from available documentation. These discrepancies will be revisited in Section IV-A. High-resolution ground-truth diagrams as well as other details from the four systems' recoveries can be found at [7].

A. Apache Hadoop

Hadoop is a widely used open-source framework for distributed processing of large datasets across clusters of computers [5]. Hadoop is about 200 KSLOC in size and contains over 1,700 OO classes. Two principal subsystems of Hadoop implement the Hadoop Distributed File System (HDFS) [30] and the Map/Reduce programming paradigm [18].

Map/Reduce is used for processing large-scale datasets in a parallel and distributed fashion. In this paradigm, a dataset is divided into parallelizable chunks of data. The dataset and the operations that can be performed on it are called *jobs*. A master, called *JobTracker* in Hadoop, runs on a single node. JobTracker takes a job and distributes it to workers, called *TaskTrackers*. Hadoop uses HDFS to store data. HDFS relies on a fault-tolerant master/slave architecture. A master node, called *NameNode*, manages the filesystem's namespace, access to files by clients, and the distribution of data. The other nodes in the cluster are slaves of the NameNode, called *DataNodes*. DataNodes store and manage blocks comprising files in HDFS.

Hadoop is accompanied with rich usage documentation, auto-generated API documentation, and high-level architectural documentation. Figure 3 depicts an architectural diagram taken from the Hadoop documentation. This diagram captures dependencies between certain system components and several operations that those components perform. The available documentation helped the recoverers get familiar with the application and to identify certain expected components (e.g., TaskTracker, NameNode). The documentation also revealed that Hadoop is divided into three major subsystems, corresponding to HDFS, Map/Reduce, and the system utilities.

Our certifier for Hadoop was a Yahoo! engineer who has been a long-time contributor to the system and has an intimate understanding of its architecture. We relied on two recoverers, both of whom are PhD students with previous experience in software architecture recovery. One of the recoverers had industry experience with building applications based on Map/Reduce; the other recoverer had industry experience with building distributed systems for large-scale data storage and processing. This experience aided the recoverers in obtaining mapping principles relevant to Hadoop's application domain. The recoverers relied on a combination of system dependencies obtained using IBM's Rational Software Architect (RSA) [4] and the Class Dependency Analyzer (CDA) [6].

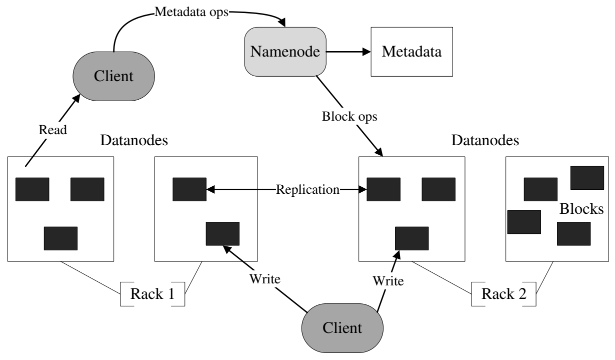
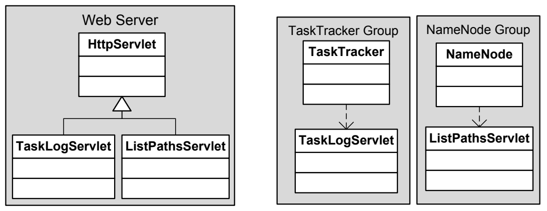


Fig. 3. An architectural diagram from Hadoop's documentation.

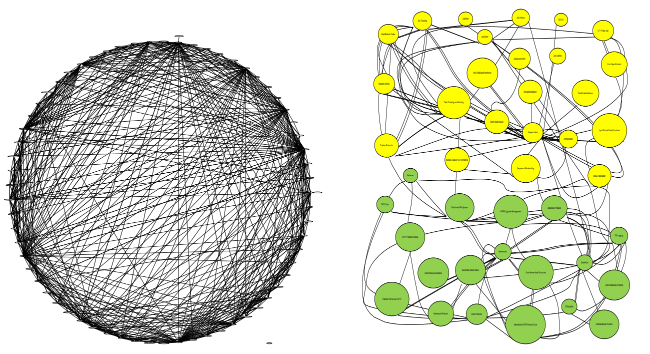


(a) Servlets merged into web server (b) Servlets separated according to use by domain components

Fig. 4. For Hadoop, an application principle overrides a domain principle.

To illustrate how different types of mapping principles are used in the ground-truth recovery process, Figure 4 shows how an application principle for Hadoop overrides a domain principle. A domain principle for Hadoop states that servlets should be grouped into a Web Server component (Figure 4a). Servlets are Java classes that respond to requests by using a request-response protocol and are often used as part of web servers. Therefore, the HttpServlet, TaskLogServlet, and ListPathServlet classes are all grouped together based on this domain principle. However, an application principle, stating that servlets should be grouped with the modules that depend on them, overrides that domain principle. Thus, the TaskTracker and TaskLogServlet classes are grouped into one component, while the NameNode and ListPathServlet classes are grouped into another component (Figure 4b).

Two views of the recovered ground-truth architecture of Hadoop are depicted in Figure 5. Again, at this magnification the diagrams are not intended to be readable, but rather to convey a sense of the size, structure, and complexity of the architecture. Figure 5a shows all of Hadoop's recovered components and their interdependencies in a circular layout. Figure 5b shows only the components from the Map/Reduce subsystem in yellow (lighter) ovals on top, and the components



(a) Ground-truth architecture of Hadoop in a circular layout (b) Ground-truth architecture of Hadoop with many utility components removed

Fig. 5. Two views of Hadoop's ground-truth architecture.

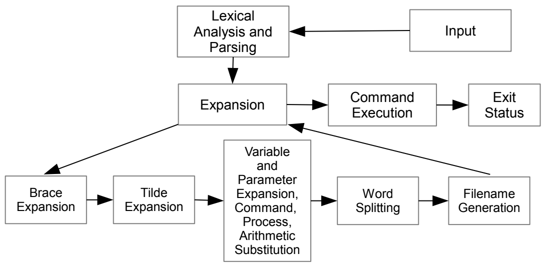


Fig. 6. Conceptual architecture of Bash.

from HDFS in green (darker) ovals on the bottom. The recovered ground-truth architecture of Hadoop is significantly different and more complex than what is depicted in Figure 3 or in any other available documentation.

B. Bash

The Bourne-Again SHell (Bash) is a command-line shell that provides a user interface to a GNU operating system (OS). Bash is included in popular OSs, such as GNU/Linux and Mac OS X. Bash is written in C, consists of about 70 KSLOC, and contains over 200 source files.

Bash's conceptual architecture, depicted in Figure 6, follows the data-flow style. Command-line input is parsed, processed, and executed by the different components in the architecture. Extensive documentation explaining the use and installation can be found on GNU's website. Some architectural information about Bash is also shipped with its code base. A chapter describing Bash's component architecture, written by Bash's primary developer/maintainer—who is also our certifier for the ground-truth architecture—can be found in [13]. Our certifier has been involved with the system for over 20 years and has been the primary developer of Bash for over 17 years.

The recovery of Bash was unique in that an incomplete recovery performed by another researcher [25] was available. Our single recoverer was a PhD student with experience in architecture recovery. Dependencies between the Bash files were obtained using the `mkdep` tool [1].

Figure 7 shows the ground-truth architecture of Bash, depicting its components and their interdependencies. Note that there are significantly more components and dependencies in the ground-truth architecture than in the conceptual architecture

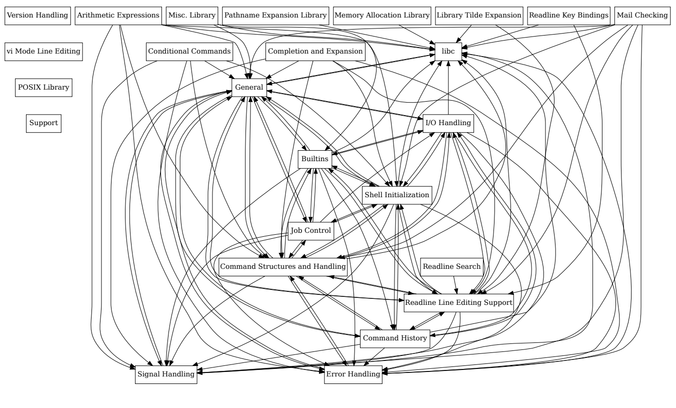


Fig. 7. Ground-truth architecture of Bash.

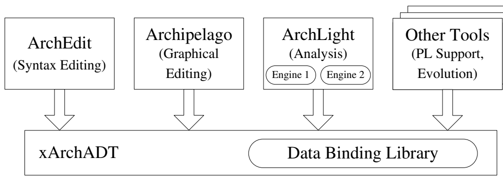


Fig. 8. Conceptual architecture of ArchStudio.

from Figure 6. Additionally, the data-flow from the conceptual architecture is not apparent in the recovered architecture. Despite these discrepancies, the conceptual and recovered views have certain similarities. For example, using the application principles resulted in the recovery of the *Readline Line Editing Support* component in Figure 7, which is a key functionality of the *Input* component from Figure 6.

C. ArchStudio

ArchStudio [16] is a development environment for modeling, analyzing, implementing, and visualizing systems and software architectures. ArchStudio is implemented in Java, and consists of about 280 KSLOC spread over 800 OO classes.

Conceptually, ArchStudio consists of a set of tools that all interact with a component called xArchADT, as depicted in Figure 8. ArchStudio uses an extensible architecture description language (ADL) called xADL [17] to represent all aspects of an architecture. ArchStudio tools interact via xADL descriptions through xArchADT. The components and connectors in ArchStudio's architecture follow the constraints of the Myx architectural style [2]. ArchStudio is implemented as a set of Eclipse plug-ins. This information is incorporated in the ground-truth recovery process in the form of system-context principles, and helped us to identify ArchStudio's components. Our two certifiers for ArchStudio were its primary architect and one of its developers. The recoverer was an MS student with significant industry experience, including prior experience with architecture recovery. CDA was used to extract dependencies from ArchStudio.

The ground-truth architecture of ArchStudio is depicted in Figure 9. It is important to notice that dependencies are sparse and that there are a significant number of disconnected components. This is consistent with the design decision to implement ArchStudio on top of the `myx.fw` architectural framework. The framework handles component interactions, which makes the

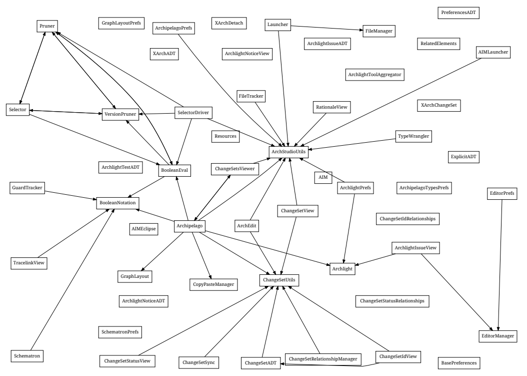


Fig. 9. Ground-truth architecture of ArchStudio.

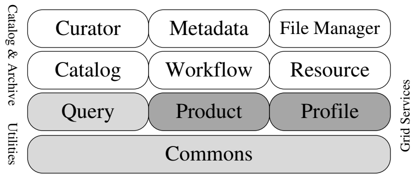


Fig. 10. Conceptual architecture of OODT.

component dependencies implicit. Our recovery process took into account the usage of myx.fw by defining a system context principle stating that components of ArchStudio must inherit from the class that represents components in myx.fw.

D. Apache OODT

Apache OODT [26] is an architecture-based framework for highly distributed, data-intensive systems. Created over the past decade, OODT consists of over 180 KSLOC. OODT provides services, tools, APIs, and user interfaces for information integration and large scale data-processing across a number of scientific domains (earth science, planetary science, bioinformatics, etc.). OODT is NASA's first project to be stewarded at the Apache Software Foundation.

The conceptual architecture of OODT is depicted in Figure 10. OODT provides components for sharing large-scale data and metadata deployed locally at a scientific institution or a data center. To enable this sharing of data, OODT has *product server* components that expose data files. Querying and retrieval of this data is enabled by *profile server* components that expose metadata associated with these data files, allow querying of metadata, and support retrieval of data files that have been queried. The data processing side of OODT provides components for file and metadata management, workflow management, and resource management. Complementary to these services are OODT's client frameworks for remote data acquisition; automatic file identification, metadata extraction, and ingestion; and scientific algorithm integration.

One of the key architects and primary developers of OODT served as the certifier for our recovery. The recoverer was a PhD student with architectural recovery experience. The recoverer had worked previously as a developer on the OODT project. However, he had very limited knowledge about the relationship of code-level entities to architectural elements. CDA [6] was used to extract dependencies from OODT.

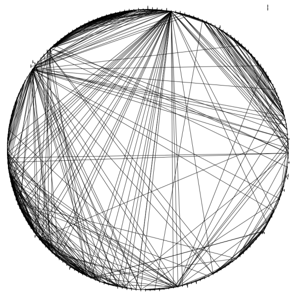


Fig. 11. Ground-truth architecture of OODT.

OODT's ground-truth architecture consists of 217 components and is depicted (at very low magnification) in Figure 11. Similarly to Hadoop, OODT's ground-truth architecture is considerably more complex than its conceptual architecture. OODT has about 20 subsystems, each of which is modularized into its own project. In turn, those subsystems were decomposed into sub-architectures at a much lower-level of abstraction than the conceptual architecture from Figure 10.

OODT's ground-truth recovery was unique in that a number of its application mapping principles involved the system's package structure. One such principle states that each component of a subsystem responsible for external system interfaces is located in packages ending with ".system".

IV. Lessons Learned

The overall aim of our work is to encourage obtaining more ground-truth architectures and to provide potential research directions for improving existing recovery techniques. To address that aim, we make several observations about the ground-truth architectural recoveries we have performed to date and the process used to obttain them. We discuss the discrepancies between the conceptual and ground-truth architectures (Section IV-A), the recovered components' sizes (Section IV-B), the role of utility components (Section IV-C), and the extent to which implementation packages and directories represent architectural components (Section IV-D). To improve understanding of the benefits of our ground-truth recovery framework, we discuss the relative importance of application and domain principles in comparison to generic principles (Section IV-E), and the effort required to obtain the ground-truth architectures (Section IV-F).

A. Discrepancies Between Architectures

The significantly higher numbers of dependencies in the recovered ground-truth architectures in comparison to the conceptual architectures of the studied systems suggest possible architectural drift and erosion (i.e., accidental, unplanned introduction of design decisions). While drift and erosion may explain some of the clutter in the ground-truth architectures, there are also more prominent reasons behind it: (1) implicit capture of architectural constraints (e.g., styles) in the implementation, (2) different levels of abstraction, and (3) different goals of architectural views.

The conceptual view of a system's architecture often considers the architectural style(s) used for the system's design, and consequently the specific types of software connectors employed by those styles. For example, according to its conceptual architecture from Figure 6, Bash follows the data-flow style and thus includes data-stream connectors. However, existing architecture recovery techniques are not able to directly recover connectors. In the case of Bash, this means that the data-stream connectors remain encapsulated inside implementation-level artifacts for I/O manipulation, such as buffers, and are thus mapped to the I/O utility component.

Besides style-based discrepancies, there are significantly more components in the ground-truth architectures than in their conceptual counterparts. In Bash, the application principles obtained from documentation and the modifications to the architecture recommended by the certifier resulted in architectural diagrams at a lower-level of abstraction than the conceptual architecture of Figure 6. For example, the *Job Control* component in Figure 7 can be considered part of the *Command Execution* component in Figure 6. Furthermore, application principles help to identify utility components, which are not typically depicted in conceptual architectural diagrams. For example, Hadoop's ground-truth architecture diagram that omits the utility components (Figure 5b) is indeed closer to the conceptual diagrams found in the documentation than the complete ground-truth diagram (Figure 5a).

Even though the conceptual and ground-truth architectures are at different levels of abstraction, all of them are considered valid by their certifiers. The documentation of Bash, ArchStudio, and OODT, which describes their respective conceptual architectures, was either written or recommended by the certifiers. In the case of Hadoop, our certifier's explanations indicated that the ground-truth was an architectural view that would help a developer understand the code from an architectural perspective. On the other hand, the intent behind the conceptual architecture may not be to strongly correlate with the implementation, but rather to partially and often informally represent certain important aspects of the system. For example, the conceptual architecture of Bash from Figure 6 represents a data element *Exit Status* in the same way it represents the processing components.

Our findings have potential impact on future research in the area of architecture recovery. First, since existing recovery techniques focus on components and are ill-suited for explicitly capturing architectural styles and connectors, further research should be done to identify them properly. Second, the recovery technique should be able to adapt to the goals of the recovery by presenting the appropriate, possibly partial, views. Third, further research should be conducted to determine how to identify those differences between the conceptual and the recovered architecture that are likely to suggest true architectural drift and erosion, as opposed to the differences that are simply a by-product of different abstraction levels and architectural views' objectives.

B. Recovered Component Size

Existing architecture recovery techniques can vary greatly in terms of the sizes of components they construct. Some techniques try to balance the numbers of clustered code-level entities (i.e., classes or files) across components [32]. Others allow sizes to vary depending on a system's structural dependencies, so that a single component may end up encompassing a large fraction of the system's entities [25].

We examined the extent to which components in our subject systems varied in number of constituent implementation entities. The goal of this analysis is to inform developers of new recovery techniques and future ground-truth recovery efforts about the ranges of component sizes one should expect. Figure 12 summarizes this information. For each recovered *System*, the table notes the total number of implementation *Entities*, the *Mean Size* of its components in the number of contained entities, and the ratio between the mean size and the total number of entities, expressed as a percentage (*% Mean Size*).

Across the four systems, the component sizes are not normally distributed and are mostly below the mean. The positive skewness of the component sizes, ranging from 1.93 to 4.88, confirms this fact. Over 84% of the groupings had

System	Entities	Mean Size	% Mean Size
Bash	214	9	4.00%
OODT	896	4	0.50%
Hadoop	1236	18	1.50%
ArchStudio	812	15	1.90%

Fig. 12. Data on the number of entities within components

component sizes that were smaller than the mean plus one standard deviation. Components that were beyond this size, which we refer to as *large* components, occurred infrequently. Over one half (52%-72%) of the components in Bash, Hadoop, and ArchStudio were *small*, i.e., less than half the mean size, while 42% of OODT's components were small. This suggests that recovery techniques are likely to obtain better results if the components are kept fairly small.

We also examined the extent to which components were *singletons*, i.e., consisted of only one implementation entity. Singleton components tended not to be explicitly called out in documentation and thus it is unlikely that they would be properly identified using traditional recovery techniques. Hadoop and ArchStudio had fewer than 5% singletons, while Bash had 16%. OODT was an outlier at both ends of the component-size spectrum: it had the greatest number of large components (16%) as well as singletons (24%).

C. Core and Utility Components

Previous research has shown that utility components are important for obtaining correct architecture recoveries [25], [8]. Therefore, we explicitly examined the extent to which utility components were found in each system. We discuss the roles recoverers and certifiers played in correctly identifying the utility components.

Figure 13 distinguishes utility components from core application components. The table depicts, for each *System*, the number of *Core* components, the number of *Util*ity components, the *Total* number of components, and the percentage of utility components in the system (*% Util*).

Utility components constituted significant portions of three of our four subject systems (ArchStudio being the exception). Bash contains the largest number of utility components since, as an OS shell, it utilizes extensive functionality for interfacing with the OS and handling command-line input. Hadoop also interfaces with the OS extensively, to perform its networking and filesystem functionality. One of Hadoop's three major subsystems is, in fact, dedicated to providing utility services (recall the discussion of Figure 5). One of OODT's major subsystems is also tasked with providing utility services. Additionally, a number of OODT subsystems contain their own specialized utility modules.

On the whole, the recoverers had little trouble distinguishing entities that implement core functionality from those implementing utility functionality. However, in each of the four

System	Core	Util	Total	% Util
Bash	16	9	25	36.00%
OODT	177	40	217	18.43%
Hadoop	47	21	68	30.88%
ArchStudio	52	2	54	3.70%

Fig. 13. Data on the number of core and utility components

System	Total	Span Pkgs	% Span Pkgs	Share Pkg	% Share Pkg	Not Pkg	% Not Pkg
Bash	25	6	24%	17	68%	23	92%
OODT	217	43	20%	85	39%	128	59%
Hadoop	68	18	26%	40	59%	58	85%
ArchStudio	54	18	33%	0	0%	18	33%

Fig. 14. The extent to which package or directory structures represent the architecture

systems, the certifiers made key recommendations regarding how the overall utility functionality should be divided into individual components based on application principles. For example, OODT has a largely standardized package structure per subsystem for housing utility components, of which the recoverer was unaware. For both Hadoop and Bash, in addition to supplying application principles for splitting the utility functionality into components, the certifiers also provided specific advice about moving to core components certain entities originally assigned to utility components by the recoverers.

D. Packages and Directory Structure

Conventional software engineering wisdom suggests that implementation-level package and directory structures should organize and group code elements according to the elements' conceptual functionalities. Hence, a straightforward architecture recovery method could be to simply consider packages and/or directories to be software components. In fact, this approach has been adopted by some existing work [10], [34]. To evaluate the validity of such a method, we examined the extent to which packages or directories represent the components in the four ground-truth architectures.

Figure 14 summarizes our findings. We calculated the numbers of components (1) whose constituent entities span multiple packages or directories, and the components (2) whose entities share a single package or directory with entities belonging to other components. For each *System*, the table depicts the *Total* number of components; the number of components whose constituent implementation entities span multiple packages and/or directories (*Span Pkgs*); the percentage of such components in the system (*% Span Pkgs*); the number of components whose entities share a package or directory with other components' entities (*Share Pkg*); the percentage of such components in the system (*% Share Pkg*); the total number of components for which the package or directory structure does not match the architecture (*Not Pkg*), which is the sum of Span Pkgs and Share Pkg; and the percentage of such components in the system (*% Not Pkg*).

Other than in ArchStudio, packages and directories were not generally representative of components in the studied systems. For Bash and Hadoop, the packages and directories were almost entirely different from architectural components (the value of % Not Pkg was 85% and 92%, respectively).

Bash has significantly more components than directories. Furthermore, Bash is a smaller system than the others, and is primarily maintained by a single person who is both the architect and lead developer. Therefore, the need to maintain clearly separated components is likely reduced and, thus, not mirrored in the directory structure.

Hadoop has relatively few packages scattered across many components and three major subsystems. An interesting aspect of the Hadoop recovery was that our certifier suggested many utility components that were split across packages. The certifier actually indicated that a later version of Hadoop had a package structure more representative of components, which we used as an aid in finalizing Hadoop's ground-truth recovery.

Although a majority of OODT's components were also not represented by packages, the mismatch was lower than in the case of Hadoop or Bash. This stems from OODT's package structure, which repeats across subsystems and identifies a recurring set of component types: utilities, domain data structures, command-line tools and actions, and external system interfaces. For this reason, after studying our first attempt at OODT's recovered architecture, the certifier informed us that we needed to follow the recurring package structure more closely in order to identify the appropriate component types.

ArchStudio's components resemble its packages and directory structure more closely than the other studied systems. The recovery process for ArchStudio did not explicitly include mapping principles indicating that packages and directories constitute components. However, ArchStudio's ground-truth architecture has relatively few components, each of which contains many classes. Furthermore, its implementation on top of the myx.fw framework requires explicit declaration of components in the implementation. Both of these factors may account for the 67% match of components to packages.

E. Application of Mapping Principles

For all systems except Bash, whose architecture was already partially recovered, the structural module-clustering rules of Focus [28] served as the generic principles that yielded the initial recovery. However, application and domain principles eventually dominated all the recoveries.

In the case of Hadoop, the principles for the domain of distributed systems included groupings regarding entry points of classes, threads, processes, and deployment onto different hosts. Keeping separate the expected components suggested in the Hadoop documentation (e.g., TaskTracker and JobTracker) helped further inform the recovery. Communication protocols and the code that implements them were isolated based on application principles, which were also key to producing the final ground-truth architecture. The certifier's assessment of the quality of our recoveries and suggestions for their modifications heavily focused on *semantic coherency*, i.e., whether a component captured cohesive functionality or related system concerns. Many application principles of Hadoop—e.g., "instrumentation classes should be grouped with metrics classes" or "the map-side join classes should not be grouped with join utilities"—were not obvious from either the documentation or

	Time Spent			Emails				
System	**RT (h)**	**CT (h)**	**CT/RT**	**Total**	**Recov**	**Intro**	**Sched**	**Misc**
Hadoop	120	8	7%	62	29	5	11	17
OODT	100	10	10%	25	5	5	10	5
Bash	80	2	3%	10	8	2	0	0
ArchStudio	100	7	7%	35	29	5	0	1
Mean	100	7	7%	33	18	4	5	6

Fig. 15. Time spent by recoverers and certifiers, and the number and purpose of exchanged email messages

the information about the domain. The certifier's role in the recovery was therefore critical.

OODT's recovery came about more as a result of generic principles and package structure than in the other systems. In particular, Focus's rules suggest grouping classes by inheritance, and this helped to identify components in OODT in several instances. Existing OODT documentation, in particular architectural diagrams and their explanations, provided information about application principles that identified many other OODT components. Furthermore, OODT contains a variety of connector types (e.g., data streams, RPCs, and arbitrators) and a combination of application and generic principles were used to identify the classes implementing these connector types.

ArchStudio's recovery was heavily based on application and domain principles derived from the myx.fw framework and the Myx architectural style. Those framework- and style-based principles, along with expected components obtained from documentation, dominated the recovery of ArchStudio. The available documentation similarly informed the recovery of Bash as it provided the application principles that were used to identify the files that implement different Bash components.

F. Effort of Recoverers and Certifiers

One of the major challenges we anticipated in obtaining the ground-truth architectures was having the required access to one or more certifiers to aid in the recovery. It is typically not expected that a system's architect will be willing to dedicate time to an effort whose purpose falls outside the architect's regular duties (e.g., to aid academic research). Unsurprisingly, having prior professional connections with certifiers helped to incorporate them into our studies. However, we found that, at least in the case of open-source projects, architects and developers were readily willing to expend the time and effort to help. In fact, for two of the four systems we studied, we did not have any prior connections with the eventual certifiers.

Another part of addressing this challenge involved trying to minimize the burden on the certifiers, while optimizing the use of their time and expertise. To examine whether it is realistic to recruit certifiers in aiding ground-truth recovery in general, we recorded the amount of time they spent on each of the four recoveries, as well as the amount and nature of communication between them and the recoverers. Without exception, the communication between certifiers and recoverers was performed via email. While potentially more time consuming than face-to-face meetings or phone conversations, we resorted to email because it lessens the burden on the certifiers and allows them to control when they spend the time on the recovery.

Figure 15 depicts the time recoverers and certifiers spent on the recovery effort and the number of email messages exchanged between them. For each *System*, the table shows the number of hours spent by the recoverers (*RT*); the number of hours spent by certifiers (*CT*); the ratio of the time spent by certifiers to the time spent by recoverers, expressed as a percentage (*CT/RT*); the *Total* number of emails exchanged; the number of messages involving discussions about the recovery itself (*Recov*); the number of emails involving introductions and initial requests for help (*Intro*); the number of emails about scheduling meetings (*Sched*); and the number of emails about any other issues, e.g., reminders to respond (*Misc*).

An overwhelming majority of the time spent to produce the ground-truth recoveries was invested by recoverers. On average, recoverers spent about 100 person-hours per system. By contrast, certifiers spent an average of seven hours on a recovery, which was only 3%-10% of the recoverers' time. This ratio suggests that certifiers need not spend a prohibitive amount of time verifying an intermediate recovery. Bash involved a particularly low amount of time spent by the certifier. This can be attributed to the availability of documentation that contained most of the domain and application principles. The availability of these principles, together with the smaller size of Bash relative to the other subject systems, resulted in fewer iterations with the certifier. On the other hand, the significantly higher number of components recovered in OODT than in the other systems was the likely reason why its certifier took longer to analyze the recovery results.

On average, about 30 email messages in total were exchanged to obtain a ground-truth recovery. Without exception, very few messages (2-5) were required to introduce the certifiers to the recovery project and to secure their help. One possible reason for this is that open-source developers and architects may see the usefulness of an improved architectural understanding of their systems more readily than their counterparts working, e.g., on commercial projects. Our on-going work on recovering the architecture of Google's Chromium will provide one test of that hypothesis.

A majority of email exchanges between recoverers and certifiers involved discussions of the systems' architectures and suggestions for modifying intermediate recoveries. Scheduling issues and reminders to respond unsurprisingly resulted in additional email exchanges. An outlier in terms of email traffic was OODT. The primary reason for its comparatively small amount of email discussion is that OODT's recoverer had previously worked as a developer of that system, although one with limited knowledge about OODT's overall architecture.

V. Conclusion

In this paper, we aimed (1) to help improve future architecture recovery techniques and (2) to encourage further construction of ground-truth architectures. The produced ground-truth architectures suggest three properties a recovered architecture is likely to have. First, our ground-truth architectures consisted of components with fairly limited numbers of entities, grouped predominantly based on their semantic similarities. Second, the components in a ground-truth architecture rarely have a direct correspondence to a system's package structure. Third, the perceived accuracy of a recovered architecture largely depends on the appropriate identification of utility components.

Our recovery experience appears to invalidate the prior intuition that constructing a ground-truth architecture for large systems is infeasible. In fact, we provide considerable evidence that engineers are not only willing to help recover the architectures of their systems, but also that their involvement can be made manageable by our ground-truth recovery framework. These results have encouraged us to continue this effort, with our on-going work focusing on Google Chromium, a very large system comprising 12 MSLOC.

More broadly, we believe that our findings can help improve the understanding of software architectures in general and of the ways architectures degrade. We observed that the differences between the conceptual and ground-truth architectures stem both from architectural degradation and from the different abstraction levels and purposes of those architectures. This raises further questions regarding the appropriate ways of constructing these different views, as well as understanding and reconciling their differences in support of software design, maintenance, and evolution tasks.

Finally, it is important to note that, although we recovered a single ground-truth architecture for each of the systems studied in this paper, a single system may have multiple ground-truth architectures. This will depend on the perspective from which the recoverers are approaching the architecture. Our recovery framework can, in fact, accommodate multiple ground-truths for a single system: once the mapping principles are verified by certifiers, then it is likely that more than one recovered architecture consistent with those mapping principles could be considered as *a* ground truth. This is an observation we are exploring in our on-going work.

Acknowledgment

The authors are grateful to Eric Dashofy, Chris Douglas, Bassel Haddad, Chet Ramey, and Yongjie Zheng for their assistance with recovering the ground-truth architectures of ArchStudio, Bash, and Hadoop. This work has been supported by the U.S. National Science Foundation under award numbers 1117593 and 1218115, and by Infosys Technologies, Ltd.

References

[1] (1993) mkdep. [Online]. Available: http://linuxcommand.org/man_pages/mkdep1.html

[2] (2012) Archstudio - Foundations - Myx and myx.fw:. [Online]. Available: http://www.isr.uci.edu/projects/archstudio/myx.html

[3] (2012) The chromium projects. [Online]. Available: http://www.chromium.org/

[4] (2012) IBM Rational Software Architect. [Online]. Available: http://www.ibm.com/developerworks/rational/products/rsa/

[5] (2012) Poweredby - Hadoop Wiki. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy

[6] (2012) Programmer's Friend - Class Dependency Analyzer. [Online]. Available: http://www.dependency-analyzer.org/

[7] (2012) recoveries:start [usc softarch wiki]. [Online]. Available: http://softarch.usc.edu/wiki/doku.php?id=recoveries:start

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

[9] L. Bass *et al.*, "Developing architectural documentation for the hadoop distributed file system," in *OSS*, 2011.

[10] F. Beck and S. Diehl, "Evaluating the Impact of Software Evolution on Software Clustering," in *WCRE*, 2010.

[11] R. Bittencourt *et al.*, "Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques," in *WCRE*, 2010.

[12] I. Bowman *et al.*, "Linux as a case study: its extracted software architecture," in *ICSE*, 1999.

[13] A. Brown and G. Wilson, "The architecture of open source applications," *Lulu. com*, 2011.

[14] A. Christl *et al.*, "Equipping the reflexion method with automated clustering," in *WCRE*, 2005.

[15] ——, "Automated clustering to support the reflexion method," *Information and Software Technology*, vol. 49, no. 3, 2007.

[16] E. Dashofy *et al.*, "Archstudio 4: An architecture-based meta-modeling environment," in *Companion to ICSE*, 2007.

[17] E. Dashofy and A. Van Der Hoek, "Representing product family architectures in an extensible architecture description language," *Software Product-Family Engineering*, pp. 124–144, 2002.

[18] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, 2008.

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

[20] J. Garcia *et al.*, "A framework for obtaining the ground-truth in architectural recovery," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on.* IEEE, 2012, pp. 292–296.

[21] B. Gröne *et al.*, "Architecture recovery of apache 1.3 – a case study," in *SERP*, 2002.

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

[23] ——, "Architecture reconstruction," *Software Engineering*, 2009.

[24] ——, "Incremental reflexion analysis," in *CSMR*, 2010.

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

[26] C. Mattmann *et al.*, "A software architecture-based framework for highly distributed and data-intensive scientific applications," in *ICSE*, 2006.

[27] ——, "The anatomy and physiology of the grid revisited," in *WICSA/ECSA*, 2009.

[28] N. Medvidovic and V. Jakobac, "Using software evolution to focus architectural recovery," *ASE Journal*, vol. 13, no. 2, 2006.

[29] G. C. Murphy *et al.*, "Software reflexion models: bridging the gap between source and high-level models," in *FSE*, 1995.

[30] K. Shvachko *et al.*, "The hadoop distributed file system," in *26th Symposium on Mass Storage Systems and Technologies*, 2010.

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

[32] V. Tzerpos, "Comprehension-driven software clustering," Ph.D. dissertation, University of Toronto, 2001.

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

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

[35] J. Wu and M. Storey, "A multi-perspective software visualization environment," in *CASCON*, 2000.