Uncovering Architectural Design Decisions

Arman Shahbazian, Youn Kyu Lee, Duc Le, and Nenad Medvidovic

Computer Science Department
University of Southern California
{armansha, younkyul, ducmle, neno}@usc.edu

Technical Report: USC-CSSE-17-701

***Abstract*—Over the past three decades, considerable effort has been devoted to the study of software architecture. A major portion of this effort has focused on the originally proposed view of four "C"s—components, connectors, configurations, and constraints—that are the building blocks of a system's architecture. Despite being simple and appealing, this view has proven to be incomplete and has required further elaboration. To that end, researchers have more recently tried to approach architectures from another important perspective—that of design decisions that yield a system's architecture. These more recent efforts have lacked a precise understanding of several key questions, however: (1) What is an architectural design decision (definition)? (2) How can architectural design decisions be found in existing systems (identification)? (3) What system decisions are and are not architectural (classification)? (4) How are architectural design decisions manifested in the code (reification)? (5) How can important architectural decisions be preserved and/or changed as desired (evolution)? This paper presents a technique targeted at answering these questions by analyzing information that is readily available about software systems. We applied our technique on over 100 different versions of two widely adopted open-source systems, and found that it can accurately uncover the architectural design decisions embodied in the systems.**

I. Introduction

Software architecture has become the centerpiece of modern software development, especially with the software development having its focus shifted from lines-of-code to coarser grained architectural elements such as software components and connectors [38]. Developers are increasingly relying on software architecture to lead them through the process of creating and implementing large and complex systems.

Considerable effort has been devoted to studying software architecture from different perspectives. Kruchten et al. proposed the 4+1 view model to describe software architecture using five concurrent views addressing specific concerns [24]. Another research thread has focused on architecture description languages (ADLs) [28] such as ACME [19] and π-ADL [29] for describing and modeling software systems' architectures. Architectures of the several now widely adopted systems such as mobile systems [8], and World Wide Web [16] has been extensively studied. Furthermore, researchers have recognized the importance of software architecture in the evolution and maintenance of software systems [17] which has led to the design of several architectural recovery techniques to help counteract the challenges brought about by architectural drift and erosion [14], [18], [17], [40].

These studies are typically based on some slant on the originally proposed view of software architecture as four "C"s—components, connectors, configurations, and constraints—that are the building blocks of a system's architecture. Despite being simple and appealing, this view has proven to be incomplete and has required further elaboration. To that end, researchers have more recently tried to approach architectures from another important perspective—that of design decisions that yield a system's architecture. These more recent efforts have lacked a precise understanding of several key questions, however: ① What is an architectural design decision (definition)? ② How can architectural design decisions be found in existing systems (identification)? ③ What system decisions are and are not architectural (classification)? ④ How are architectural design decisions manifested in the code (reification)? ⑤ How can important architectural decisions be preserved and/or changed as desired (evolution)? This paper presents a technique (*UnArch*) targeted at answering these questions by analyzing information that is readily available about software systems.

Our approach is guided and constrained by the following observations. The system already exists. We have access to the source code and issue repositories. We assume that the issue repository contains specific information, i.e., the list of issues and a means to obtain the pertaining code changes such as attached code commits or pull requests. Architectural documentation does not exist, is incomplete, or is unreliable. Access to the architects of some of the systems may be possible, but it is not likely. Furthermore, the architects' availability is highly constrained. Finally, the architects may not remember or be able to articulate their design decisions.

UnArch builds on top of the state-of-the-art architecture recovery techniques that recover a descriptive architecture of a system. A major shortcoming of these techniques is that they only depict "what" the architecture of a system looks like, and not "why" the architecture looks the way it does—which is the symptom of a phenomenon called knowledge vaporization in software systems [20]. To overcome this shortcoming, *UnArch* taps into the issue and code repositories. Using the information obtained from the code and issue repositories as its inputs, *UnArch* outputs a set of decisions that has been made during the system's evolution. We applied our technique on over 100 different versions of two widely adopted open- source systems, and found that it can accurately uncover the architectural design decisions embodied in the systems.

The contributions of this paper are defining and classifying different kinds of architectural design decisions that are made during the evolution of a software system; devising a technique to uncover and identify those design decisions in existing software systems; and empirically examining how they are manifested in software systems.

The remainder of this paper is organized as follows. Section II summarizes the fundamental research thrusts and concepts brought together to enable this work. Section III describes *UnArch* in detail. Sections IV describe our evaluation and key findings. Section V describes the threats to the validity of our approach and its mitigating factors. A discussion of related work (Section VI) and conclusions (Section VII) round out the paper.

II. Foundation

A. Architectural Design Decisions

For many years research community and industry alike has been focused on the result, the consequences of the design decisions made, trying to capture them in the "architecture" of the system under consideration, often using graphics. Representations of software architecture were and to a great extent still are centered on views, [12], [24], as captured by the ISO/IEC/IEEE 42010 standard [13], or usage of an architecture description language [28]. However, this approach to documenting software architectures can cause problems such as expensive system evolution, lack of stakeholders communication, and limited re-usability of architectures [36].

Architecture as a set of design decisions was proposed to address these shortcomings. This new paradigm focuses on capturing and documenting rationale, constraints, and alternatives of design decisions. More specifically Jansen et al. defined architectural design decisions as a description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, the design rules, and additional requirements that (partially) realize one or more requirements on a given architecture [20], [10]. The key element in their definition is rationale, i.e., the reasons behind an architectural design decision. Kruchten et al. proposed an ontology that classified architectural decisions into 3 categories: (1) existence decisions (ontocrises), (2) property decisions (diacrises), and (3) executive decisions (pericrises) [23]. Among the three categories, existence decisions – decisions that state some element/artifact will positively show up or disappear in a system – are the most prevalent and capture the most volatile aspects of a system [23], [20]. Property and executive decisions are enduring guidelines that are mostly driven by the business environment and affect the methodologies, and to a large extent the choices of technologies and tools.

The notion of design decisions used in this paper values the "rationales", and "consequences" as two equally important constituent part of design decisions. However, not all design decisions are created equal. Some design decisions are straight forward with clear singular rationale and consequence while some are crosscutting and intertwined [10], i.e., affect multiple components, connectors or both and often become intimately intertwined with other design decisions. To distinguish between different kinds of design decisions we classify them into three categories: (1) simple, (2) compound, and (3) crosscutting. Simple decisions have a singular rationale and consequence. Compound decisions include several closely related rationales, but their consequences are generally contained to one component. Finally, crosscutting decisions affect a wider range of components and their rationale follows a higher level concern such as architectural quality of the system.

B. Architecture Recovery, Change, and Decay Evaluator

In order to capture the consequence aspect of design decisions, we build on top of the existing work in architecture modeling and recovering. To obtain the static architectures of a system from its source code, we use our recent work which resulted in a novel approach, called *ARCADE* [9], [26]. *ARCADE* is a software workbench that employs (1) a suite of architecture-recovery techniques and (2) a set of metrics for measuring different aspects of architectural change. It constructs an expansive view showcasing the actual (as opposed to idealized) evolution of a software system's architecture.

ARCADE allows an architect (1) to extract multiple architectural views from a system's codebase and (2) to study the architectural changes during the system's evolution as reflected in those views. *ARCADE* currently provides access to ten recovery techniques. We use two of these techniques in this paper. These two techniques are *Algorithm for Comprehension-Driven Clustering* (*ACDC*) [40] and *Architecture Recovery using Concerns* (*ARC*) [18]. Our previous evaluation [17] showed that these two techniques exhibit the best accuracy and scalability of the ten. *ACDC*'s view is oriented toward components that are based on structural patterns (e.g., components consisting of entities that together form a particular subgraph). On the other hand, *ARC*'s view produces components that are semantically coherent due to sharing similar system-level concerns (e.g., a component whose main concern is handling distributed jobs).

To measure architectural changes across the development history of a software system, *ARCADE* provides several architecture similarity metrics: cvg [26] and a2a [9], MoJo [39], and MoJoFM [42]. These are system-level similarity metrics calculated based on the cost of transforming one architecture to another. Using similar principles, *UnArch* conducts the change analysis and extracts a system's architectural changes (refer to Section III-A).

III. Approach

Knowledge vaporization in software systems plays a major role in increasing maintenance costs, and exacerbates architectural drift and erosion [20]. The goal of *UnArch* is to uncover architectural design decisions in software systems, and thereby help reverse the course of knowledge vaporization by providing a crisper understanding of such decisions and their effects.

This section aims at providing answers to the five questions raised in the introduction. We elaborate on the *definition* and *classification* of design decisions. We describe how

architectural changes that are the *reifications* of architectural design decisions can be recovered from the source code of real software systems. We also describe a process whereby architectural decisions are *identified* in real systems. Finally, our approach enables engineers to continuously capture architectural decisions in software systems during their *evolution*.

In Section II-A, we identified two constituent parts of an architectural design decision, *rationale* and *consequence*. The static architecture of a system explicitly captures the system's components and possibly other architectural entities, but rationale is usually missing or, at best, implicit in the structural elements. For this reason, our approach focuses on the consequences of design decisions. We have developed a technique that leverages the combination of source code and issue repositories to obtain the design decision consequences. Issue repositories are used to keep of track of bugs, development tasks, and feature requests in a software development project. Code repositories contain historical data about the inception, evolution, and incremental code changes in a software system. Together, these repositories provide the most reliable and accurate pieces of information about a system.

UnArch automatically extracts the required information from a system's repositories and outputs the set of design decisions made during the system's lifecycle. In order to achieve this *UnArch* first recovers the static architecture of the target system. *UnArch* then cross-links the issues to their corresponding code-level changes. These links are in turn analyzed to identify candidate architectural changes and, subsequently, their rationales.

A high level overview of *UnArch*'s workflow is displayed in Figure 1. *UnArch* begins by recovering the static architecture of a system. This step is only required if an up-to-date, reliable, documented architecture is not available. After recovering or obtaining the architectures of different versions of its target system, *UnArch* follows through three distinct phases. In the first phase (**Change Analysis**) *UnArch* identifies how the architecture of the system has changed along its evolution path. The second phase (**Mapping**) mines the system's issue repository and creates a mapping (called *architectural impact list*) from issues to the architectural entities they have affected. Finally, the third phase (**Decision Extraction**) creates an overarching decision graph by putting together the architectural changes and the architectural impact list. This graph is in turn traversed to uncover the individual design decisions. In the remainder of this section we detail each of the three phases.

A. Change Analysis

Architectural change has been recognized as a critical phenomenon from the very beginnings of the study of software architecture [31]. However, only recently have researchers tried to empirically measure and analyze architectural change in software systems [26], [9]. These efforts rely on architectural change metrics that quantify the extent to which the architecture of a software system changes as the system evolves. This work has served as a motivation and useful foundation in obtaining a concrete view of architectural changes.

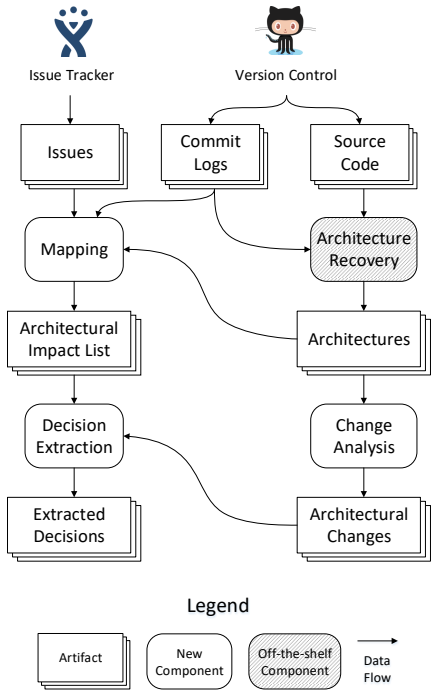


Fig. 1. Overview of *UnArch*. Using the existing source code, commit logs, and extracted issues obtained from code and issue repositories, our approach automatically extracts the underlying design decisions. Implementation of the new components spans over 4,000 Source Lines of Code (SLoC).

Specifically, we have designed *Change Analyzer* (*CA*), which is inspired by the manner in which existing metrics (e.g., a2a [9], MoJo [39], and MoJoFM [42]) measure architectural change. These metrics consider five operations used to transform architecture A into architecture B: addition, removal, and relocation of system entities (e.g., methods, classes, libraries) from one component to another, as well as addition and removal of components themselves [7], [27], [30]. We use a similar notion and define architectural change as a set of *architectural deltas*. An architectural delta is: (1) any modification to a component's internal entities including additions and removals (a relocation is treated as a combined addition to the destination component and removal from the source component), or (2) additions and removals of entire components. We then aggregate these deltas into architectural change instances. Algorithm 1 describes the details of the approach used to extract the architectural deltas and changes.

CA works in two passes. In the first pass, *CA* matches the most similar components in the given pair of architectures. In the second pass, *CA* compares the matched components, extracts the architectural delta(s), and clusters them into architectural change instances.

The objective of the matching pass is to find the most similar components in a way that minimizes the overall difference between the matched components. Since two architectures can have different numbers of components, *CA* first balances (Algorithm 1, line 5) the two architectures. To do so, *CA* adds "dummy" (i.e., empty) components to the architecture with fewer components until both architectures have the same

Algorithm 1: Change Analysis

Input: *ArchitectureA*, *ArchitectureB*
Output: *Changes* ⇐ a set of architectural changes
1 Let *ComponentsA* = *ArchitectureA*'s components
2 Let *ComponentsB* = *ArchitectureB*'s components
3 Let E_{all}, $E_{chosen} = \emptyset$
4 **if** $|ComponentsA| \neq |ComponentsB|$ **then**
5 $\quad$ *Balance*(*ComponentsA*, *ComponentsB*)
6 **foreach** $c_a \in$ ComponentsA **do**
7 $\quad$ **foreach** $c_b \in$ ComponentsB **do**
8 $\quad\quad$ $cost = CalculateChangeCost(c_a, c_b)$
9 $\quad\quad$ $e = \{c_a, c_b, cost\}$
10 $\quad\quad$ *add e to* E_{all}
11 $E_{chosen} = MinCostMatcher(ComponentsA, ComponentsB, E_{all})$
12 **foreach** $e \in E_{chosen}$ **do**
13 $\quad$ $Changes = GetChangeInstances(e.c_a, e.c_b) \cup Changes$
14 *return Changes*

number of components. After balancing the architectures, *CA* creates a weighted bipartite graph from architecture A to architecture B and calculates the cost of each edge. Existence of an edge denotes that component C_A has been transformed into component C_B. The cost of an edge is the total number of architectural deltas required to effect the transformation.

Algorithm 2: *GetChangeInstances* method

Input: *ComponentA*, *ComponentB*
Output: *Changes*
1 Let *EntitiesA* = *ComponentA*'s entities
2 Let *EntitiesB* = *ComponentB*'s entities
3 **if** $EntitiesA \cap EntitiesB = \emptyset$ **then**
4 $\quad$ *Change* ch_1, ch_2
5 $\quad$ $ch_1.deltas = EntitiesA$
6 $\quad$ $ch_2.deltas = EntitiesB$
7 $\quad$ *return* $\{ch_1, ch_2\}$
8 **else**
9 $\quad$ *Change ch*
10 $\quad$ $ch.deltas = (EntitiesA \cup EntitiesB) - (EntitiesA \cap EntitiesB)$
11 $\quad$ *return* $\{ch\}$

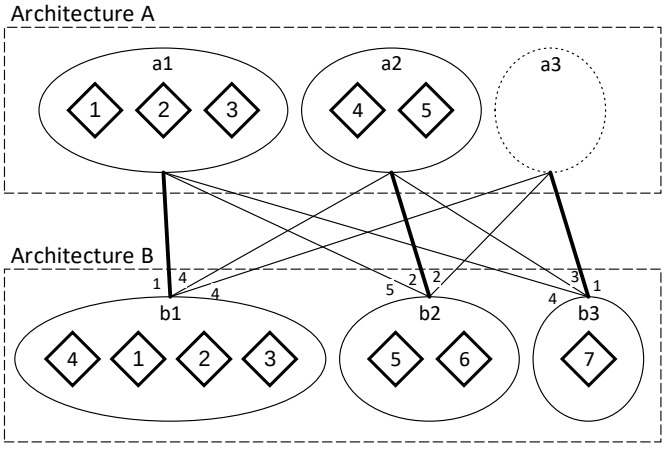


Fig. 2. Calculating the costs of the edges and finding the perfect matching. The bold connectors are the selected edges that lead to minimum overall cost.

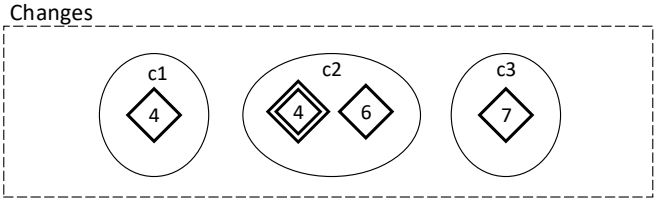


Fig. 3. Extracted changes between the architectures depicted in Fig. 2. Double-lined diamonds indicate removals while regular diamonds denote additions.

Figure 2 displays a simple example of two architectures and the corresponding bipartite graph with all possible edges. *MinCostMatcher* (Algorithm 1, line 11) takes the two architectures and the set of edges between them, and selects the edges in a way that ensures a bijective matching between the components of the two architectures with the minimum overall cost (sum of the costs of the selected edges). *MinCostMatcher* is based on linear programming; its details are omitted for brevity.

In the second pass, *CA* extracts the architectural deltas between the matched components. If there are no common architectural entities between two matched components, we create two change instances, one for the component that has been removed and one for the newly added component. The reason is to distinguish between transformations of components and their additions and removals. Figure 3 depicts the extracted changes of our example architectures.

B. Mapping

The output of *CA* is a set of architectural changes that is a superset of the consequences of design decisions. The goal of *Mapping* is to find all the issues that point to the rationale of the design decisions that yielded those consequences. To that end, *Mapping* first identifies the issues that satisfy two conditions: (1) they belong to the version of the system being analyzed and (2) they have been marked as resolved and their consequent code changes have been merged with the main code base of the system. *Mapping* then extracts the code-level entities affected by each issue. These code-level entities are identified by mining the issues' commit logs and pull requests. Using one or more architecture recovery methods available in *ARCADE*, the code-level entities are translated into corresponding architectural entities. The list of all issues, as well as the mapping between the issues and the architectural entities affected by them is called the *Architectural Impact List*.

A graph-based view of this list is displayed in Figure 4. It is possible for issues to have overlapping entities (e.g., i2 and i3 are both connected to entity 5). It is also important to note that the presence of an edge from an issue to an entity does not necessarily indicate architectural change (e.g., entities 1 and 5 are not part of any of the architectural changes displayed in Figure 3). This is intuitively expected, since a great many of issues do not incur substantial enough change in the source code and thereby the architecture of the system.

C. Decision Extraction

In its final phase, *UnArch* creates the overarching decision graph by putting together the architectural changes and their

Algorithm 3: *Decision Extraction*

Input: *ArchitecturalImpactList*, *Changes*
Output: *Decisions*

```
1 Let DecisionsGraph = bipartite graph of decisions
2 foreach (issue, entities) ∈ ArchitecturalImpactList do
3     foreach c ∈ Changes do
4         if c.deltas ∩ entities ≠ ∅ then
5             connect(issue, c) in DecisionsGraph
6 Decisions = FindDecisions(DecisionsGraph)
7 return Decisions
```

pertaining issues. This graph is traversed and individual design decisions are identified. Algorithm 3 details this phase.

Algorithm 3 traverses the architectural impact list generated in the *Mapping* phase and the list of changes. If there is an intersection between the entities matched to issues and the entities involved in changes, then it adds an edge connecting the issue with the change. The intuition behind this is that an issue contains the rationale for a decision if it affects the change(s), which are the consequences, of that decision. We note that, hypothetically, there can be situations in which an issue is the cause of a change without directly affecting any architectural deltas in that change. For example, if an issue leads to removing all the dependencies to an entity, that entity might get relocated out of its containing component by the architecture recovery technique. However, detecting these situations in a system's architecture is not possible with existing recovery techniques, because they abstract away the dependencies among internal entities of a component. Although such information could easily be incorporated, *UnArch* would be unable to deal with such scenarios as currently implemented.

The decisions graph for our running example is depicted in Figure 5. The *FindDecisions* method in Algorithm 3 removes all orphaned changes and issues, and in the remaining graph locates the largest disconnected subgraphs. Each disconnected

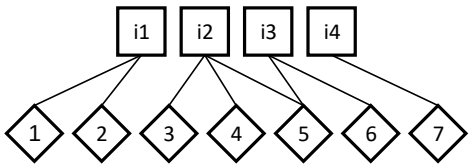


Fig. 4. Architectural impact list. Squares represents issues and diamonds represent entities. An edge from an issue to an entity means that resolving that issue resulted in modifying that entity.

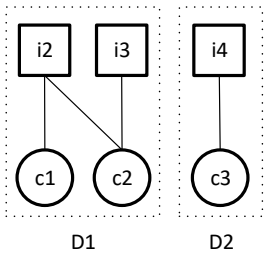


Fig. 5. The overarching decisions graph contains two decisions D1 and D2. Squares denote issues, and circles denote changes.

subgraph represents a decision. The reason is that these disconnected subgraphs are the largest sets of interrelated rationales and consequences that do not depend on other issues or changes. Intuitively, we expect that, in a real-world system, only a subset of issues will impose changes whose impact on the system can be considered architectural. Furthermore, each of those issues will reflect a specific, targeted objective. Therefore, in a typical system, the graph of changes and issues should contain disconnected subgraphs of reasonable sizes. This is discussed further in our evaluation in Section IV.

In Section II, we identified three different types of decisions: (1) Simple decisions are the decisions that consist of a single change and a single issue. These decisions have a clear rationale and consequence. (2) Compound decisions are the decisions that include multiple issues and a single change. These decisions are similar to simple decisions and the issues involved are closely related to an overarching rationale. Finally, (3) cross-cutting decisions are the decisions that include multiple changes and one or more issues. These decisions have a higher-level, "compound" rationale—e.g., improving the reliability or performance of a system—that requires multiple changes to be achieved. In Figure 5, D_1 is a cross-cutting decision while D_2 is a simple decision.

For illustration, Table I lists three real examples of decisions, one of each type, uncovered from Hadoop. Information in the *Issue(s)* column contains the summaries of the issues pertaining to that decision. Each boxed number indicates a separate issue or change. The data in the *Change(s)* column is a short description of the changes involved in a given decision. The simple decision in the top row is an update to satisfy a requirement by changing the *job tracking module*. The compound decision in the middle row describes the two sides of a problem that is resolved by changing the *compression module* of Hadoop. Finally the uncovered crosscutting decision in the bottom row is about a series of changes applied to increase the reliability of Hadoop's task execution by preventing data corruption, and providing two facilities that make it easier to check the status of the tasks that are being executed.

IV. Evaluation

We have empirically evaluated *UnArch* to verify its applicability and measure its accuracy in uncovering architectural design decisions. Section IV-A discusses the real-world systems on which *UnArch* was applied, demonstrating its applicability. Sections IV-B and IV-C discuss the precision and recall of our results, respectively, demonstrating *UnArch*'s accuracy.

A. Applicability

Table II contains information about the two subject systems we have used in our evaluation. These systems were selected from the catalogue of Apache open-source software systems [2]. We selected Hadoop [1] and Struts [6] because they are widely adopted and fit the target profile of candidate systems for our approach: they are open-source, and have accessible issue and code repositories. Furthermore, these systems are at the higher end of the Apache software systems' spectrum in terms of

TABLE I
EXAMPLES OF UNCOVERED DECISIONS IN HADOOP

Decision Type	Issue(s)	Change(s)
Simple	1 Job tracking module only kept track of the jobs executed in the past 24 hours. If and admin checked the history after a day of inactivity, e.g., on Monday, the list would be empty.	① `hadoop.mapred` component was modified
Compound	1 UTF8 compressor does not handle end of line correctly 2 Sequenced files should support 'custom compressors'	① `CompressionInputStream` was added and `CompressionCodec` was modified.
Crosscutting	1 Random seeks corrupts `InputStream` data 2 Streaming should send status signals every 10 seconds 3 Task status should include timestamp for job transitions	① `hadoop.streaming` was modified ② `hadoop.metrics` was modified ③ `hadoop.fs` was modified

TABLE II
SUBJECT SYSTEMS ANALYZED IN OUR STUDY

System	Domain	No. Ver.	No. Iss.	MSLoC
Hadoop	Distributed Processing	68	2969	30.0
Struts	Web Apps Framework	36	1351	6.7

size and lifespan. Both of these projects use GitHub [4] as their version control and source repository, and Jira [5] as their issue repository. We analyzed more than 100 versions of Hadoop and Struts in total. Our analyses spanned over 8 years of development, 35 million SLoC, and over 4,000 resolved issues.

An overview of the results of applying *UnArch* to the two subject systems is depicted in Table III. These results are grouped by (1) system (Hadoop vs. Struts) and (2) employed recovery technique (*ARC* vs. *ACDC*). In this table, *No. of Iss. in Decisions* represents the total number of issues that were identified to be part of an architectural design decision. On average, only about 18% of the issues for Hadoop and 6% of the issues for Struts have had architecturally significant effects, and hence have been considered parts of a design decision. This is in line with the intuition that only a subset of issues will impose changes whose impact on the system can be considered architectural. Morevoer, this observation bolsters the importance of *UnArch* for understanding the current state of a system and the decisions that have led to it. Without having access to *UnArch*, architects would have to analyze 5-to-15 times more issues and commits to uncover the rationales and root causes behind the architectural changes of their system. The remainder of Table III displays the total number of detected architectural changes (*No. of Changes*), the total number of uncovered architectural design decisions (*No. of Decisions*), and the average numbers of issues and changes per decision (*Avg. Issues/Decision* and *Avg. Changes/Decision*, respectively). It is worth mentioning that not all the detected changes were

TABLE III
OVERVIEW OF THE RESULTS

Systems	Hadoop		Struts	
	ACDC	ARC	ACDC	ARC
No. of Iss. in Decisions	427	674	70	94
No. of Changes	950	3935	220	1359
No. of Decisions	112	149	27	23
Avg. Issues/Decision	3.81	4.52	2.59	4.94
Avg. Changes/Decision	1.77	2.36	1.77	2.21

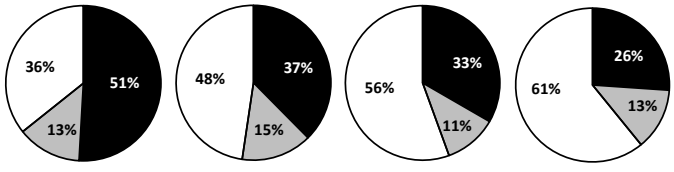
matched to design decisions. We will elaborate further on this in Section IV-C, which evaluates *UnArch*'s recall.

As displayed in Table III, depending on the technique used to recover the architecture, the number of uncovered design decisions varies. The reason is that *ACDC* and *ARC* approach architecture recovery from different perspectives: *ACDC* leverages a system's module dependencies; *ARC* derives a more semantic of a system's architecture using natural language processing and information retrieval techniques. Therefore, the nature of the recovered architectures and changes, and consequently the uncovered design decisions, are different. In our previous work, we showed that these recovery techniques provide complementary views of a system's architecture [26]. The propagation of these complementary views to our approach has yielded some tangible effects. For instance, *UnArch* running *ARC* was able to uncover a decision about refactoring the names of a set of classes and methods in Hadoop, while *UnArch* running *ACDC* could not uncover that decision. The reason is that *ARC* is sensitive to lexical changes by design. Depending on the context and objectives of using *UnArch*, architects can choose the most favorable view for their purposes.

UnArch aims to uncover three kinds of architectural design decisions (recall Section II). Our results confirmed the presence of all three kinds in our subject systems. Figure 6 depicts the distribution of different kinds of decisions detected for each pair of systems and recovery techniques. While the relative proportion of simple and cross-cutting decisions varies across systems and employed recovery techniques, the number of compound decisions is consistently the smallest. One possible explanation is that as a system matures resolution of its issues either become more isolated to a smaller scope that leads to detection of simple decisions, or more quality drivern that leads to detection of crosscutting decisions.

B. Precision

In order to assess the precision of *UnArch*, we need to determine whether the uncovered architectural design decisions



(a) Hadoop-ACDC (b) Hadoop-ARC (c) Struts-ACDC (d) Struts-ARC

Fig. 6. Distribution of different types of decision in the subject systems: solid black represents simple decisions; grey denotes compound decisions; white displays cross-cutting decisions.

are valid. As captured in the premise of *UnArch*, architectural design decisions are not generally documented, hence a "ground-truth" for our analyses was not readily available.

To overcome this hurdle, we devised a systematic plan to objectively assess the rationales and consequences of the uncovered design decisions. We defined a set of criteria targeting the two aspects of an architectural design decision—rationale and consequence—and used them as the basis of our assessment. Two PhD students carefully carried out the analysis and the results of their independent examinations were later aggregated. In the remainder of this section, we will elaborate on the details of the conducted analyses.

We use four criteria targeting different parts of an architectural design decision (two targeting rationales and two targeting consequences). Each criterion is rated using a three-level-scale, with the numeric values of 0, 0.5, and 1. In this scale, 0 means that the criterion is not satisfied; 0.5 means that the satisfaction of the criterion is confirmed after further investigation by examining the source-code, details of the issues, or commit logs; finally, 1 means that the criterion is evidently satisfied. The reason we use a three-level scale in our analysis is to measure the precision of the results of our approach from the viewpoint of non-experts, and to distinguish the decisions according to the effort required for understanding them. To that end, any criterion whose evaluation requires (1) in-depth system expertise, (2) inspection of information other than that captured in design decisions, and/or (3) having access to the original architects of the system, is given a rating of 0.

The criteria for assessing rationales are two-fold:

1) *Rationale Clarity* indicates whether the rationale and its constituent parts are clear and easily understandable. This is accomplished by looking at the summaries of the issues and pinpointing the problems or requirements driving the decision.
2) *Rationale Cohesion* indicates the degree to which there is a coherent relationship among the issues that make up a given rationale. *Rationale Cohesion* is only analyzed if the decision is shown to possess the *Rationale Clarity* criterion.

The criteria for assessing consequences are also two-fold:

1) *Consequence-Rationale Association* assesses whether the changes and their constituent architectural deltas are related to the listed rationale.
2) *Consequence Tractability* assesses whether the size of the changes is tractable. In other words, is the number of changes and their constituent deltas small enough to be understandable in a short amount of time?[1] *Consequence Tractability* is only analyzed if the decision is shown to possess the *Rationale Clarity* criterion.

The two PhD students independently scored every decision based the above criteria. The three-level scale allowed us to develop a finer grained understanding of the quality of the decisions.

[1]As a rule of thumb, decisions including more than five changes did not satisfy this criterion in our evaluation. This is, of course, adjustable.

```
Rationales:
 Issue 1:
   Desc: Seperating user logs from system
         logs in map reduce
   ID  : HADOOP-489
Consequences:
 Change 1:
   Added: org.apache.hadoop.mapred.TaskLog
```

Listing 1. A simple decision from Hadoop v. 0.9.0

```
Rationales:
 Issue 1:
   Desc: Implement the LzoCodec to support
         the lzo compression algorithms
   ID  : HADOOP-851
 Issue 2:
   Desc: Native libraries are not loaded
   ID  : HADOOP-873
Consequences:
 ...
```

Listing 2. Part of a crosscutting decision from Hadoop v. 0.10.1

As illustrative examples, we explain the scoring procedures for two decisions in Hadoop. Listing 1 displays a simple design decision as uncovered by *UnArch* in Hadoop version 0.9.0. The rationale consists of a single issue that explains the intent is to separate the user logs from system logs. However, the rationale summary does not explain why this needs to happen. Looking at the issue in Jira, the reason is that system logs are cluttering the user logs, and system logs need to be cleared out more frequently than user logs. Since we had to look at the issue to understand "why" this decision was made, the *Rationale Clarity* in this case was scored 0.5. Since we only have one issue, the *Rationale Cohesion* is not applicable. The consequence involves one change with a single architectural delta, i.e., adding the `TaskLog`. The relationship of this change to the issue is clear and the change size is tractable. Therefore, *Consequence-Rationale Association* and *Consequence Tractability* each received 1. In Listing 2 which is a crosscutting example from Hadoop 0.10.1, although the rationales seem unrelated, after inspecting the code and issue logs we realized that *LzoCodec* will be available only if the *Native Library* is loaded. Therefore, this decision received 0.5 for *Rationale-Cohesion*.

Table IV displays the average scores of the analyzed decisions, grouped by the decision type and the recovery technique used for uncovering the decisions. Figures 7 and 8 display the cumulative distributions of the decision scores for Hadoop and Struts, respectively. The right-leaning feature of these distributions indicates that the higher-quality decisions

TABLE IV
AVERAGE DECISION SCORES

Decisions	**Hadoop**		**Struts**	
	ACDC	***ARC***	***ACDC***	***ARC***
Simple	0.89	0.95	0.90	0.99
Compound	0.50	0.52	0.76	0.56
Crosscutting	0.61	0.76	0.78	0.77
Overall	0.72	0.72	0.81	0.71

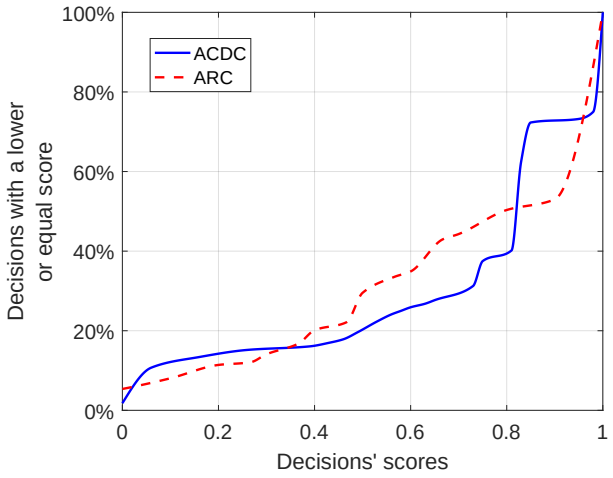


Fig. 7. Smoothed cumulative distribution of the decision scores for Hadoop.

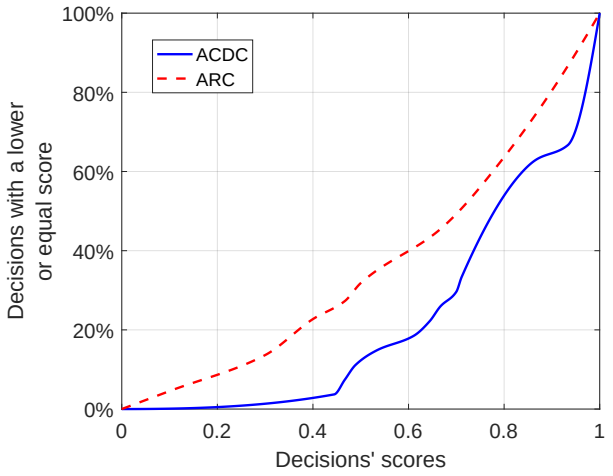


Fig. 8. Smoothed cumulative distribution of the decision scores for Struts.

are more prevalent than the lower-quality ones. The threshold of acceptability for measuring precision is adjustable, but in our evaluation we required that a decision scores at least 0.5 in the majority (i.e., at least three) of the criteria. In our analyses, on average (considering both *ARC* and *ACDC*) 76% of the decisions for Hadoop and 78% of the decision for Struts met this condition.

Most of the unacceptable decisions were made in the newly introduced major versions of the two systems. In our previous work, we observed that the number of architectural changes between a minor version (e.g., 0.20.2) and the immediately following version that is major (1.0.0) tends to be significantly higher than the architectural change between two consecutive minor versions. In these cases, the decision sizes (number of rationales and consequences) tend to be higher than our conservative thresholds, and these decisions tend to be rated as unacceptable. However, these decisions still provide valuable insight into why the architecture has changed.

The reason that the *ARC*-based decisions generally score lower (i.e., they are less right leaning) than the *ACDC*-based ones is due to the nature of changes extracted by *ARC*. While *ACDC* adopts primarily a structural approach to architecture, *ARC* follows a semantic approach and requires a higher level of system understanding. Therefore, attaining a conclusive rating for these decisions was not possible by only looking at the decision elements defined earlier. Our findings suggest that the uncovered decisions based on *ARC* are more suitable for experienced users.

TABLE V
ARCHITECTURAL CHANGE COVERAGE

	Hadoop		Struts	
	acdc	**arc**	**acdc**	**arc**
Before Cleanup	0.20	0.19	0.21	0.24
After Cleanup	0.85	0.67	0.80	0.63

C. Recall

Our objective in assessing the recall of our approach is to find out the extent to which *UnArch* manages to successfully capture the design decisions in our subject systems. Based on the definition of the architectural design decisions (recall Section II), every architectural change is a consequence of a design decision. Therefore, we use the coverage of architectural changes by the identified design decisions as a proxy indicator for measuring the recall of *UnArch*.

Our initial analyses reported an underwhelming recall: only a relatively small fraction of the extracted changes formed design decisions. The first row of Table V displays the results of our initial analyses. The recall of the extracted architectural changes was consistently around 20% in our subject systems regardless of the used recovery technique. To understand the root cause of this, we manually examined the detected architectural changes for which *UnArch* could not locate the rationale. We were able to identify two major reasons why an architectural change was not marked as part of a design decision by *UnArch*. The first was when architectural change was happening in the off-the-shelf components that are integrated with the system and evolve separately. These can be third-party libraries, integrations with the other Apache software projects, or even changes in the core Java libraries that are detected by the recovery techniques. Examples of this phenomenon for Struts includes changes to the Spring Framework's architecture [22], and for Hadoop changes to Jetty [3] and several non-core Apache Common projects. The second reason is what we call the "orphaned commit" phenomenon. Orphaned commits are the commits that conceptually belong to an issue, but (1) were not added to an issue, (2) have been merged with the code-base before their containing issues has been marked as resolved, or (3) a human error in the issue data rendered them useless for our approach (e.g., incorrectly specified affected version).

We consider orphaned commits a shortcoming of our approach that can affect the recall. However, the imposed changes on a system's architecture do not capture the original intentions of the developers and architects. Therefore, we carefully inspected the architectural changes to eliminate the ones caused by external factors. In our inspection, we created a list of namespaces whose elements should not be considered architectural changes caused by the developer

decisions. Truncated lists of these namespaces for Hadoop and Struts are displayed in Listings 3 and 4, respectively. We verified each entry by searching the system's code repository and confirming that the instances where imported and not developed internally by the developer teams.

```
com.facebook.*
java.lang.*
org.apache.commons.cli.*
javax.ws.rs.*
...
```

Listing 3. Imported namespaces for Hadoop

```
com.opensymphony.xwork2.util.*
java.io.*
org.apache.commons.*
org.springframework.*
...
```

Listing 4. Imported namespaces for Struts

We then reevaluated the recall of the changes for our approach. The results are displayed in the second row of Table V. The recall of our approach after eliminating externally caused changes is **73%** on average. This also reveals an interesting byproduct of *UnArch*, namely, by using *UnArch* or a specially modified version of it, we can detect the parts of a system that are not developed or maintained by the system's core team. This information can be used for automatic detection of external libraries and dependencies in software systems, and can help the recovery techniques in extracting a more accurate view of a system's "core" architecture.

V. Threats to validity

We identify several potential threats to the validity of our approach and results with their corresponding mitigating factors. The key threats to **external validity** involve our subject systems. Although we use two systems for our evaluations, these systems were chosen from the higher end of the Apache spectrum in terms of size and lifespan, each have a vibrant community, and are widely adopted. Another threat stems from the fact that both of our systems are using GitHub and Jira. However, *UnArch* only relies on the basic issue and commit information that can be found in any generic issue tracker or version control system. The different numbers of versions analyzed per system pose another potential threat to validity. This is unavoidable, however, since some systems simply undergo more evolution than others.

The **construct validity** of our study is mainly threatened by the accuracy of the recovered architectural views and of our detection of architectural decisions. To mitigate the first threat, we selected the two architecture recovery techniques, ACDC and ARC, that have demonstrated the greatest accuracy in our extensive comparative analysis of available techniques [17]. These techniques are developed independently of one another and use very different strategies for recovering an architecture. This, coupled with the fact that their results exhibit similar trends, helps to strengthen the confidence in our conclusions. The manual inspection of the accuracy of the design decisions uncovered by our approach is another threat. Human error in this process could affect the reported results. To alleviate this problem, two PhD students independently analyzed the results to limit the potential biases and mistakes. Moreover, the inspection procedure was designed to be very conservative.

VI. Related Work

We will briefly touch upon the most closely related approaches that have been proposed to justify, model, or recover architectural design decisions.

Jansen and Bosch et al. [20], [10] defined architectural design decisions and argued for the benefits of the invaluable information getting lost when architecture are modeled using purely structural elements. Kruchten et al. proposed an ontology that classified architectural decisions into 3 categories: (1) existence decisions (ontocrises), (2) property decisions (diacrises), and (3) executive decisions (pericrises) [23]. Taylor et al. [38] also defined software architectures in terms of the "principal design decisions" yielding them. Several researchers focused on studying the concrete benefits of using design decisions in improving software system's quality [37], and decision making under uncertainty [11]. Falessi et al. extensively studied design rationale and argued for the value of capturing and explicitly documenting this information [15]. A recent expert survey by Weinreich et al. [41] showed that knowledge vaporization is a problem in practice, even at the individual level.

Roeller et al. [32] proposed RAAM to support reconstruction of the assumptions picture of a system. Assumptions are the architectural design decisions that are made during the evolution of a software system. A serious problem with their approach is that the researchers need to acquire a deep understanding of the software system ADDRA [21] was designed to recover architectural design decisions in an after the fact documentation effort. It was built on the premise that in practice, software architectures are often documented after the fact, i.e. when a system is realized and architectural design decisions have been taken. Similar to RAAM, ADDRA also relies on the tacit knowledge of the architect.

VII. Conclusions and Future Work

Modeling software systems based on the original four "C"s—components, connectors, configurations, and constraints—that are considered the structural building blocks of a system's architecture is deemed responsible for a phenomenon called knowledge vaporization. Knowledge vaporization in software systems plays a major role in increasing maintenance costs, and also exacerbates architectural drift and erosion [20]. To that end, researchers have more recently tried to approach architectures from the perspective of architectural design decisions. These efforts, however, have lacked a precise understanding of architectural design decision's *definition*, *identification*, *classification*, *reification*, and *evolution*.

To address these questions, we have developed *UnArch*, a technique for automatically uncovering architectural design decisions in existing software systems. We built our definition of an architectural design decision on the constructs previously

identified by the research community. Our efforts have led to the first automated decision recovery technique that relies solely on the information in issue and code repositories of a software system which for many software systems are the only reliable source of information. Our empirical evaluation shows that *UnArch* exhibits high accuracy and recall and can successfully detect different types of design decisions.

There are a number of remaining research challenges that will guide our future work. There is a slew of information in software repositories that can help increase the accuracy of our approach. These include comments, commit messages, documentations, pull requests, etc. Our approach can be used in tandem with other techniques aiming to support better understanding of quality repercussion of architectural changes [25], [35], [34]. *UnArch* can also be extended with a neural abstractive summarization technique [33] to provide more accurate summaries of the rationales and consequences.

REFERENCES

[1] Apache Hadoop, http://hadoop.apache.org, 2017.

[2] Apache software foundation, http://apache.org, 2017.

[3] Eclipse jetty, https://eclipse.org/jetty/, 2017.

[4] Github, https://www.github.com, 2017.

[5] Jira, https://www.atlassian.com/software/jira, 2017.

[6] Struts, http://struts.apache.org, 2017.

[7] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: A reconfiguration language for distributed systems. *Distributed Systems Engineering*, 1(5):313, 1994.

[8] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software*, 119:31–44, 2016.

[9] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering*, pages 1–48, 2016.

[10] J. Bosch. Software architecture: The next step. In *European Workshop on Software Architecture*, pages 194–199. Springer, 2004.

[11] J. E. Burge. Design rationale: Researching under uncertainty. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22(04):311–324, 2008.

[12] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[13] I. S. O. E. Committee et al. Iso/iec 42010: 2011-systems and software engineering–recommended practice for architectural description of software-intensive systems. Technical report, Technical report, ISO, 2011.

[14] L. De Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[15] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, and P. Kruchten. The value of design rationale information. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):21, 2013.

[16] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[17] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496. IEEE, 2013.

[18] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 552–555. IEEE Computer Society, 2011.

[19] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. IBM Corp., 2010.

[20] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120. IEEE, 2005.

[21] A. Jansen, J. Bosch, and P. Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536–557, 2008.

[22] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, et al. The spring framework–reference documentation. *Interface*, 21, 2004.

[23] P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen workshop on software variability*, pages 54–61. Citeseer, 2004.

[24] P. B. Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

[25] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. Automated extraction of rich software models from limited system information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 99–108, April 2016.

[26] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *12th IEEE Working Conference on Mining Software Repositories*, pages 235–245, 2015.

[27] N. Medvidovic. Adls and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pages 24–27. ACM, 1996.

[28] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering*, 26(1):70–93, 2000.

[29] F. Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[30] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society, 1998.

[31] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[32] R. Roeller, P. Lago, and H. van Vliet. Recovering architectural assumptions. *Journal of Systems and Software*, 79(4):552–573, 2006.

[33] A. M. Rush, S. Chopra, and J. Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[34] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic. Detecting event anomalies in event-based systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 25–37, New York, NY, USA, 2015. ACM.

[35] A. Shahbazian, G. Edwards, and N. Medvidovic. An end-to-end domain specific modeling and analysis platform. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, pages 8–12. ACM, 2016.

[36] M. Shahin, P. Liang, and M. R. Khayyambashi. Architectural design decision: Existing models and tools. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 293–296. IEEE, 2009.

[37] A. Tang, M. H. Tran, J. Han, and H. Van Vliet. Design reasoning improves software design quality. In *International Conference on the Quality of Software Architectures*, pages 28–42. Springer, 2008.

[38] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. Software architecture: foundations, theory, and practice. 2009.

[39] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clusterings. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 187–193. IEEE, 1999.

[40] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *wcre*, pages 258–267, 2000.

[41] R. Weinreich, I. Groher, and C. Miesbauer. An expert survey on kinds, influence factors and documentation of design decisions in practice. *Future Generation Computer Systems*, 47:145–160, 2015.

[42] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 194–203. IEEE, 2004.