Constructing a Shared Infrastructure for Software Architecture Analysis and Maintenance

Joshua Garcia[⋆], Mehdi Mirakhorli[†], Lu Xiao[α], Yutong Zhao[α], Ibrahim Mujhid[†], Khoi Pham[β], Ahmet Okutan[†], Sam Malek[⋆], Rick Kazman[γ], Yuanfang Cai[δ], and Nenad Medvidović[β]

[⋆]University of California Irvine, {joshug4,malek}@uci.edu
[†]Rochester Institute of Technology, {mxmvse,ijm9654,axoeec}@rit.edu
[α]Stevens Institute of Technology, {lxiao6,yzhao102}@stevens.edu
[β]University of Southern California, {khoipam,neno}@usc.edu
[γ]University of Hawaii, kazman@hawaii.edu
[δ]Drexel University, yfcai@cs.drexel.edu

Abstract*—Over the past three decades software engineering researchers have produced a wide range of techniques and tools for understanding the architectures of large, complex systems. However, these have tended to be one-off research projects, and their idiosyncratic natures have hampered research collaboration, extension and combination of the tools, and technology transfer. The area of software architecture is rich with disjoint research and development infrastructures, and datasets that are either proprietary or captured in proprietary formats. This paper describes a concerted effort to reverse these trends. We have designed and implemented a flexible and extensible infrastructure (*SAIN*) with the goal of sharing, replicating, and advancing software architecture research. We have demonstrated that *SAIN* is capable of incorporating the constituent tools extracted from three independently developed, large, long-lived software architecture research environments. We discuss *SAIN*'s ambitious goals, the challenges we have faced in achieving those goals, the key decisions made in *SAIN*'s design and implementation, the lessons learned from our experience to date, and our ongoing and future work.

***Index Terms*—architecture analysis, maintenance, interoperability, reproducibility, reusability**

I. Introduction

A software system's architecture comprises the principal design decisions employed in the system's construction and evolution [1]–[3]. Architecture is a key determinant of the system's properties. While it is possible, for example, to make low-level design decisions for a system (e.g., the choice of a specific data structure), to implement the system carefully, and to test it thoroughly, none of those activities can mitigate inadequate architectural choices. Put simply, software systems "live and die" [4] by their architectures.

Despite this critical importance, the architectures of many systems are not explicitly documented. Instead, those architectures are reflected—actually, hidden—in the myriad system implementation details, posing significant challenges to the development, maintenance, and evolution of long-lived systems. In particular, the effort and cost of *software maintenance* dominate activities in a software system's lifecycle [5]–[8]. Understanding and updating a system's architecture is a critical facet of maintenance. The engineers of such a system must regularly 1 *analyze* the system to understand it, its architecture, and the implications of their planned changes; to do so, the engineers must somehow 2 *recover* the architecture from the system's implementation in order enable the analysis, and determine how to best 3 *represent* the obtained architectural knowledge. Software engineering practice has shown this to be an exceptionally challenging problem, and engineers are forced to guess—and they very often actually ignore—the architectural implications of their choices and decisions.

To overcome this problem, for over the past two decades, software architecture research has yielded many different tools and techniques [9]. However, empirical studies and technology transfer are impeded by disjoint research and development environments, lack of a shared infrastructure, high initial costs associated with developing and/or integrating robust tools, and a dearth of datasets. The resulting one-off solutions inhibit further advances in software architecture research, delaying or preventing systematic synthesis and empirical validation of new or existing techniques. As a result, researchers and practitioners in need of cutting-edge tools tend to re-invent, re-implement research infrastructure, or ignore particular research avenues altogether. In doing so, they repeat each other's efforts as well as mistakes, so that opportunities for potential breakthroughs are often missed and the field is replete with solutions that do not work as advertised and/or are not interoperable.

To address these challenges, we propose **S**oftware **A**rchitecture **IN**strument (*SAIN*), a first-of-its-kind framework for assembling tools in support of *architecture-based software maintenance*. *SAIN*'s capabilities have been motivated by directly engaging the software researcher and practitioner communities, in the form of three workshops as well as a survey conducted by the authors. *SAIN* is delivered as a web-based platform consisting of three principal components: 1 a *catalogued library* of cutting-edge tools for reverse engineering and analyzing software systems' architectures; these tools are either provided by their original authors or reproduced from literature; 2 a *plug-and-play instrument* for integrating the tools and techniques to facilitate empirical studies of software architectures; and 3 *reproducibility wizards* to set up experiment templates, produce replication packages, and release them in easy-to-run and modify formats.

SAIN aims to facilitate empirical studies as well as development of new architecture analysis and maintenance solutions. By providing an extensible repository of architectural artifacts for non-trivial software systems, a major goal of *SAIN* is to enable researchers to establish a shared understanding of the relative accuracy of different techniques, to identify the gaps and sources of inaccuracy, and to develop new solutions to continually improve results. *SAIN* provides researchers with commonly needed data structures to represent architectural artifacts and algorithms for conducting a wide range of analyses, thereby enabling our community to build on each others' work and to reduce the re-development of commonly needed capabilities.

SAIN also has the potential to impact the *practice*. Over time, it will provide practitioners with an authoritative source where they can obtain and try out various tools, provide feedback, contribute to the repository of architectural artifacts, and influence the research in this area. Similarly, the benchmark results, made available through *SAIN*'s portal, will help the practitioners determine which tools are suitable for obtaining architectural information for their systems.

The key contributions of this paper are as follows:

- We introduce a *SAIN*, a framework that comprises a library of cutting-edge tools for architecture recovery and analysis, a plug-and-play instrument for integrating tools, and reproducibility wizards to support replication of architecture-based research studies.
- We discuss our experience and our users' experiences of *SAIN* in terms of the three tool suites currently contributed to *SAIN*; 13 architecture recovery components, 8 components for computing architectural metrics or analyses, 2 fact extractors, and 9 utility components from those tool suites; one compact case study of *SAIN* run on a game engine project called Mage and another detailed case study of *SAIN* run on Hadoop 2.5.0; and the empirical results of the detailed case study, which analyzes the relationships between architectural smells, architectural tactics, and error-proneness.
- We discuss experimental results from our detailed case study that are summarized into 5 major findings that can aid architects with maintainability by focusing on a small set of architectural elements that involve error-prone modules, architectural tactics, and architectural smells.
- We make *SAIN* publicly available for researchers and practitioners at [10].

Section II covers *SAIN*'s foundational concepts. Section III discusses the requirements elicitation process for *SAIN* and the key challenges it aims to overcome. Section IV discusses *SAIN*'s key design principles and alternatives considered. Section V details our experience to date; Section VI summarizes our lessons learned; and then our paper concludes.

II. Background and Foundation

To set the stage for subsequent discussion, we introduce key concepts framing software architecture, recovery, and analysis.

A. Architectural Decay

As software evolves, a major challenge impeding its successful maintenance is *architectural decay* [2], [11], where changes made to a system in the course of maintenance and evolution actually violate the system's intended architecture. The effects of decay include increased time and effort required to perform maintenance tasks and introduction of architectural defects (e.g., a system unable to interface with outside agents due to conflicting assumptions about network protocols).

As an example of architectural decay, consider Bash [12], a widely used Unix shell. Bash's conceptual architecture [13] is depicted in Figure 1a. Its as-implemented architecture [14], shown in Figure 1b, shows noticeable decay: not only do the components differ, but there are many dependencies that are unaccounted for in the conceptual architecture.

Decay has been reported in the architectures of a number of widely-used software systems [14], [15]. Recent studies have increasingly showcased the urgent need to address architectural decay. A study surveying over 1,800 software engineers and architects found architectural decay to be the greatest source of *technical debt* [16], and to be highly correlated with bugs and additional maintenance effort [17], [18].

B. Software Architecture Recovery

Reverse-engineering an architecture from implementation artifacts is referred to as *architecture recovery* [19]–[21]. Multiple *architectural views* [2], [22] of a system may be desirable, depending on the objective of recovery. For instance, a runtime view may be appropriate for reasoning about a system's security, performance, and availability [23], [24], while different structural and/or behavioral views, obtained either automatically [20], [21], [21], [21], [25]–[33], [33], [34], [34], [34]–[36] or with the aid of analysis tools [37]–[48], may allow reasoning about the implications of a range of system changes. Thus, different recovery techniques may be needed for different architectural analyses [49]–[51]. Having ready access to multiple recovery techniques directly motivated *SAIN*.

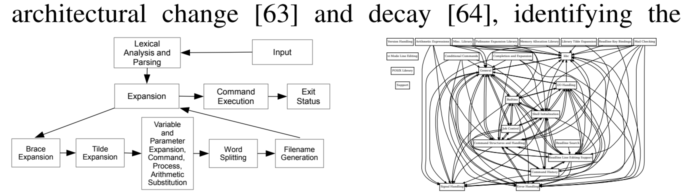
For illustration, consider the four architectural views of Bash in Figure 2. Figure 2a is the as-implemented architecture from Figure 1b, redrawn in a circular layout. The other three views were each obtained from a different automated recovery technique. Figure 2b uses information retrieval to create a semantic architectural view, while Figures 2c and 2d depict different structural views. Each of the four views may be useful for different maintenance tasks. For example, a structural view may be more effective when considering system reconfiguration; a semantic view may be better suited for understanding the system concerns. Prior work has suggested a way of integrating multiple architectural views [52].

C. Architectural Analyses

Once an architecture is recovered from code-level artifacts, a variety of analyses and subsequent activities are made possible: identifying or predicting instances of architectural decay; repairing the architecture to eliminate decay; optimizing it to achieve desired quality attributes; and so on. We highlight a body of analyses that has inspired *SAIN* most directly.

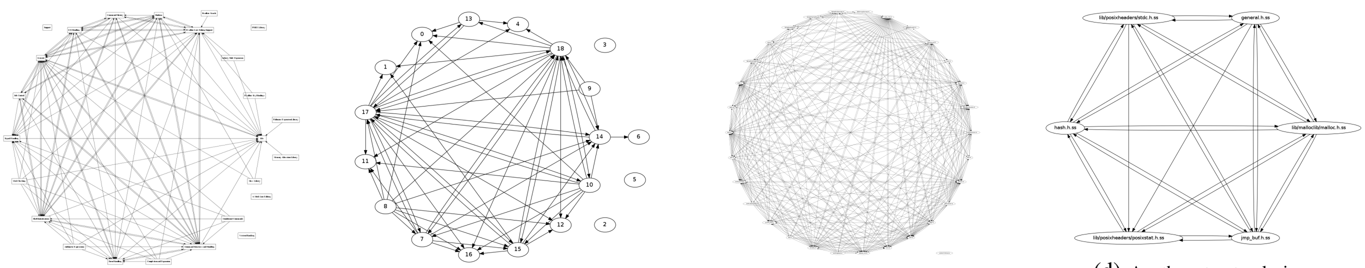
A prominent activity for tracking architectural decay in software systems is *architectural smell detection* [53], [54]. An architectural "smell" is a design decision that negatively impacts a system's maintenance and evolution. Potential adoption of existing techniques for detecting [17], [55]–[57] and, subsequently, repairing [58]–[62] architectural smells is hampered by the lack of 1 readily reusable recovery techniques and 2 architectural benchmarks (e.g., architectural models that can serve as "ground truths") on which their efficacy can be evaluated and subsequent improvements measured.

Recently, evolutionary architectural analyses have been performed across multiple versions of existing software systems. These studies include assessing the nature and extent of architectural change [63] and decay [64], identifying the



(a) Conceptual arch. (b) As-implemented arch.

Fig. 1: Architectures of Bash. The architectures are depicted at this magnification only as a way of visually comparing them; the reader is not expected to understand their details.



(a) As-implemented architecture. (b) Semantic view. (c) Structural view. (d) Another structural view.

Fig. 2: Architectural views of Bash from different recovery techniques. The architectural views are depicted at this magnification only for visual comparison; the reader is not expected to study their details.

correlation between co-occurring changes across architectural modules and implementation defects [65], and attempting to predict architectural decay [66]. However, as before, the lack of available architectural benchmarks and "out of the box" recovery techniques restricts the scope and architectural phenomena studied, and renders each study one-of-a-kind.

III. *SAIN*'s Requirements and Challenges

SAIN is motivated by challenges that are faced by the software engineering research community, and frequently discussed at conferences, workshops, and in the research literature. More specifically, the requirements for *SAIN* were directly elicited from the software architecture and software engineering research communities.

To elicit requirements for *SAIN*, we utilized two complementary techniques: requirements elicitation at brainstorming workshops, and an online survey. We organized three invitation-only, focused workshops that involved around 50 researchers and practitioners from the software architecture and empirical software engineering areas. The workshop attendees were guided through discussions of opportunities, challenges, and community needs for the area of software architecture. Furthermore, the participants brainstormed features and use cases of *SAIN* as well as ways to address the challenges and community needs. Through these workshops, we solicited and specified 17 requirements for *SAIN*.

We further asked the community to help us prioritize these requirements through an online survey—which was filled out by 60 members of the research community. These requirements involved creating a repository of benchmarks and datasets (e.g., machine-readable architectural models) and tools (e.g., tools that extract implementation-level information or architectural metrics), and the kinds of user interfaces and utilities *SAIN* would provide to the research community (e.g., reusable experiment templates or visualization capabilities). Ultimately, these various requirements involved re-occurring and time-consuming research prototyping challenges that can be potentially automated or outsourced as engineering tasks. Satisfying these requirements would facilitate and speed up research breakthroughs and productivity for many research groups working in the areas of software architecture, maintenance, and empirical software engineering.

The resulting requirements obtained from these workshops and the survey fall under five key challenges faced by the software engineering community, when conducting architecture-oriented research centered on software maintenance and evolution. In this paper, we focus on the three challenges that we have prioritized for the current version of *SAIN*.

C1 – Research Tool Accessibility and Reusability. Implementations of research techniques are often unavailable, defective, not easily accessible, or no longer supported by their original creators. For tools that do work, it is common for them to not operate as advertised, requiring major effort to adapt these tools for further research.

C2 – Interoperability of Tools. Software architecture research and technology transfer is hampered by dispersed research environments and stove-piped solutions emerging from different research groups. This, in turn, inhibits research advances, makes it difficult to synthesize techniques and tools in novel ways, and complicates comparisons of research solutions. Researchers and practitioners in need of cutting-edge architectural analyses must often recreate tools or their major elements, including basic code analysis, reverse-engineering functions, and frameworks. Furthermore, different assumptions that these tools make (e.g., about the execution environments, formats used, implementation languages, etc.) prevent their combined use, further inhibiting breakthroughs.

C3 – Reproducibility of Experiments and Analyses. Due to inaccessible, non-reusable, or defective tools, datasets, and case studies, and incompatible underlying tool assumptions, it is difficult to reproduce the results of many previous software architecture-oriented research studies [67], [68]. For software architecture-oriented research, it is often necessary to construct previous tools and datasets entirely from scratch to that end [14], [15], [49]–[51], [69]. Several repositories for collecting and sharing research artifacts have been established within the software engineering community, including PROMISE [70], Eclipse Bug Data [71], Bug Prediction Dataset [72], SIR [73], and TraceLab [74]. These repositories have played a major role in fostering research in various sub-fields of software engineering, such as software testing and analysis, requirements traceability, and software maintenance. However, none of these repositories are aimed at providing and sharing artifacts related to software architecture research, nor can they be easily modified and adapted to host such artifacts.

IV. Design and Features of *SAIN*

In addressing the three major challenges the current version of *SAIN* targets, we describe the design principles, major design decisions, and the key features of *SAIN*.

A. *SAIN's Design Principles*

Various design principles were considered and architectural alternatives analyzed to identify a design that could adequately address the needs and challenges identified through *SAIN*'s requirements elicitation effort. *SAIN*'s core design principle is based on a *plug-and-play architecture* to enable tool accessibility and reusability (C1), interoperability of tools

(C2), and reproducibility of experimental templates (C3). Specifically, components added to *SAIN* that respect a standard interface can easily interoperate with other *SAIN* components for novel experiments; entire experiment workflows can be saved, modified, and shared; and *SAIN* allows for search and navigation of tools of interest for researchers who wish to reuse or access tools or their constituent components.

To facilitate ease of composing a new experimental pipeline using existing *SAIN* tools or their constituent components (C3), *SAIN* incorporates a plug-and-play solution based on components that respect a standard interface expected by *SAIN* and provision of wrappers or converters to address disparate languages or data formats. This solution allows users to upload an executable format of an existing tool or its constituent components into *SAIN* and have it ready for integration with other tools or components which, in turn, addresses C2. Upon importing a tool or component, an *SAIN* user needs to use *SAIN* to specify the tool's or component's interface, parameters, and specific execution commands. By adhering to such a standard, *SAIN* can execute the tool or component. This integration solution relies on interface compatibility, however, since each tool or component may be developed by different researchers using different languages and formats, especially for novel research prototypes, *SAIN* allows users to upload and incorporate components that act as wrappers or converters, enabling integration of novel tools and components with existing *SAIN* components (i.e., C2).

SAIN experimental pipelines utilize a *pipe-and-filter* architectural style that helps combine components in some experiments that involve sequential processing of the information. Furthermore, *SAIN* uses a *blackboard* architectural style in cases where a sequential order cannot be defined. This architectural style allows components to communicate through a shared data model. The use of these two styles enable flexible experiment workflows to be designed, saved, reused, and shared—which aids in addressing C3.

SAIN's design also enables a *drag-and-drop* mechanism that users can leverage to easily compose new experiment pipelines by dragging a component from *SAIN*'s component catalog and dropping it onto the integration environment's canvas. This simplicity of use and access directly supports overcoming C1. The *SAIN* UI relies on a graphical programming language that allows creation of workflows which can be used by *SAIN* to compose components and generate a fully executable pipeline in the back end, which helps to address C3. This graphical programming language-based UI is depicted in Figure 3.

B. Prototyped SAIN Design Alternatives

To evaluate various alternatives brainstormed by the team during joint application design sessions, we implemented a prototype of the architecture to examine five design alternatives early on and assess the risks. The first two solutions we assessed but did not adopt are based on *Google's Blockly* [75], a library that represents coding concepts as interlocking blocks and generates syntactically correct code in the programming language, and a publish subscribe-based architectural style. The three remaining solutions are ones we collectively adopted for *SAIN*: a *custom visual programming language* which allows a user to run the tools from the browser without needing to write a single line of code; a microservice-based design; and a hybrid pipe-and-filter and blackboard architectural style-based solution. In the following paragraphs, we discuss these five design alternatives in more detail.

While Blockly was effective at forming a program using low-level coding elements, this design alternative was less practical and more complex for integration of disparate tools compared to the alternatives. Needing to specify low-level programming elements increases complexity of experiment template or workflow creation without a worthwhile increase in experiment expressibility.

For the visual programming language-based solution, each tool contributed to *SAIN* is represented as a *graphical block* or node in *SAIN*'s front end. On the back end, each tool is represented as a Node.js API service. The prototype of this solution was successful at addressing requirements related to all three major challenges C1, C2, and C3. The visual aspect of the approach, which is similar to end-user programming solutions, simplified quick experimentation with ease of tool reuse and access (C1), while still allowing complex component integration (i.e., C2) and sophisticated experiments (i.e., C3). Therefore, this design alternative was chosen and *SAIN* is delivered as a web-based platform that can be used for quick experimentation by even novice users and new researchers.

The prototype of the microservice-based design included using typical microservice solutions, i.e., containers and exposure of tool interfaces using HTTP. This solution enables standalone reuse of tools and their constituent components, allowing researchers to easily run each tool or component within a Docker container on their local machines. To enable integration of contributed tools in *SAIN* as microservices, our visual programming language is used to allow users to compose experiments without dealing with technical difficulties.

The final alternative we considered was a hybrid pipe-and-filter and blackboard architectural style compared to a simpler publish-subscribe style. Although the publish-subscribe style would enable a highly flexible architecture for experiment templates, the highly general interfaces of such a style were unsuitable for the more specific and controlled interfaces needed to contribute tools to *SAIN*. Additionally, a pipe-and-filter style more naturally modeled the kinds of pipeline-like workflows used in empirical software engineering-oriented experiments. On the other hand, the blackboard style enabled a user to have flexible integration of partial solutions to form an experiment in which components could communicate or independently act by reading and writing data in a global shared store. This design is particularly suitable for *SAIN* as complex experiments may not necessarily have a deterministic pipeline and might be composed of various experimental fragments. The hybrid pipe-and-filter and blackboard style is well-suited for enabling various forms of interactions needed to create complex experiments in which tool integration can be process-centric or data-centric.

C. SAIN's Library of Architecture Recovery and Analysis Tools

Through the three workshops and online survey discussed in Section III, *SAIN*'s requirements focused on four different types of tools it must support to enable tool reuse and accessibility (C1) and tool interoperability (C2): tools for architecture recovery; architectural analysis and metrics; fact extractors; and utilities. These types of tool are selectable in the *SAIN* visual programming language-based UI depicted in Figure 3. Specifically, the pane on the left side of Figure 3 shows four groups of tools selectable by a user that can be dragged-and-dropped onto the canvas of the window to produce experiment

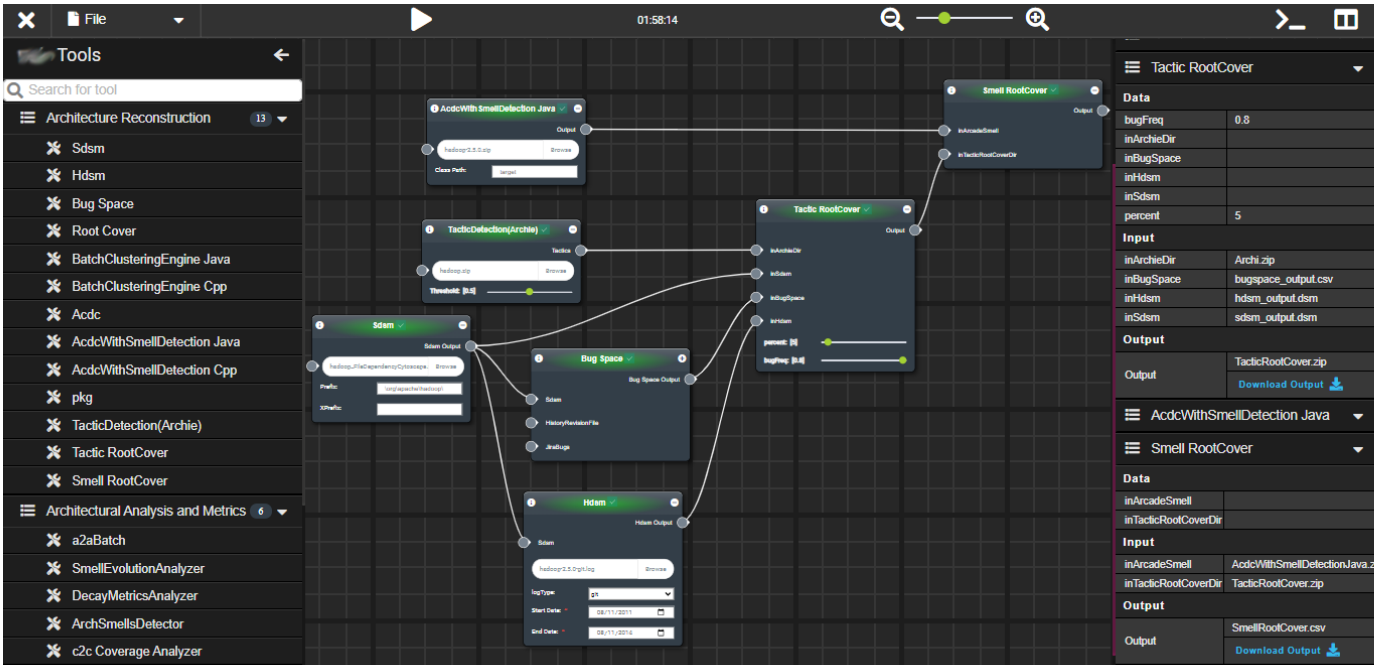


Fig. 3: An experiment integrating three tools to study error-prone modules, architectural tactics, and architectural smells. The left pane has components that can be drag-and-dropped onto the canvas in the middle, where they can be integrated. Intermediate data for each component is accessible from the right pane.

workflows. We discuss each of these tools and their importance further in the following paragraphs.

Architecture recovery tools obtain architectural abstractions of a system based on implementation-level entities. Given that such tools aim to directly determine an architecture to overcome the pervasive problem of architectural decay, having such tools were critical for *SAIN* in addressing C1.

Tools for computing metrics related to architecture recovery and analysis were deemed highly important and discussed extensively in *SAIN* workshops and the online survey. Participants of the workshops and survey pointed to the need to use standard metrics and easily reuse tools to measure architectures (e.g., compute metrics about architectural smells) and compare architectures of implemented systems from various domains (e.g., metrics for comparing a recovered architecture against a ground-truth architecture [28]).

Fact extractors are used to obtain *raw facts* about a software system. Examples of such raw facts include dependencies between software modules, system and package dependency graphs, change requests from issue-tracking repositories, architectural metrics, etc. There was extensive discussion in workshops about how simply having fact extractors that are accessible and reusable would facilitate and speed up empirical research in software architecture on its own—especially since many fact extractors often need to be re-implemented to serve as raw materials for creating novel experiments.

Utilities are tools that provide "helper" functionality, such as data-format conversion and statistical analysis that may not be architectural in nature on their own but are critical for interoperability of tools, i.e., C2. For example, different architecture recovery techniques can sometimes use different data formats as input for representing raw facts. In *SAIN*, an example utility is a tool for uploading projects from different sources (e.g., a GitHub repo or a program directory) or a generic data-mining tool like Mallet [76] that might be reused in some studies.

D. *SAIN features: Reproducibility Wizards*

To address the key challenge of realizing reproducible experiments and analysis (C3), three key reproducibility wizard features have been implemented in *SAIN*: experimental workflow composition, reusable experimental templates, and easy assembly of replication packages.

To enable novel and reproducible experiment templates in *SAIN*, we provided features for construction of workflows involving *SAIN* artifacts and datasets. Combining artifacts into workflows facilitates running new experiments or reproducing previous ones. For example, Figure 3 shows the workflow of the seven components from three tool suites Titan, Archie, and ARCADE: `Sdsm`, `Hdsm`, `Bug Space`, `Tactic Detection`, `ACDCWithSmellDetection`, `Tactic RootCover`, and `Smell RootCover`. This new experiment template enables the integration of architectural roots of error-proneness, architectural tactic implementation, and architectural smell analysis, leading to new and valuable findings which otherwise are not available. This template intuitively illustrates the workflow of using the seven components to identify the architectural roots of error-proneness and their association with architectural tactics in a software project. The experiment rationale and details will be introduced in Section V. The point here is that, following the flow in this template, an analyst can easily reproduce this experiment, by first executing `Sdsm`, `TacticDetection`, and `ACDCWithSmellDetection`. The intermediate output of `Sdsm`, `Bug Space`, `Hdsm`, and `Tactic Detection` are used as input to `Tactic RootCover`; and the intermediate output of `ACDCWithSmellDetection` and `Tactic RootCover` are used as input to `Smell RootCover`.

Beside the ability to specify new experiment workflows, we have included a number of predefined, commonly employed workflows to serve as templates. Users can easily reuse and revise these experimental pipelines. Currently, we have released six templates focusing on reverse engineering different architectural views, detecting architectural smells, and investigating the relationships between smells and software quality issues.

One of the key features of the instrument is to allow researchers to easily assemble their experiment setup using *SAIN* and a save menu, export it as a self-contained replication package (available on the top-left *File* menu in Figure 3). Storage of experiment templates or workflows using this feature allows for sharing the exact experiment structure used by a researcher—enabling researchers to easily understand and modify existing experiments to produce novel experiments to achieve new breakthroughs in software architecture research.

V. Experience with *SAIN*

To convey our experience of constructing *SAIN*, we discuss the tool suites and components it currently contains, the experience of the initial users of *SAIN*, and some *SAIN* experiments conducted so far. We further present a compact case study of architectural smell detection using *SAIN* on the game engine project, Mage [77], and a detailed case study of *SAIN* on Hadoop 2.5.0 [78], a large and widely used framework for distributed processing of large datasets across cluster computers. The latter case study combines various components from three different tool suites incorporated into *SAIN*, the benefits and challenges provided by *SAIN* in that context, and novel empirical results obtained from it.

A. Current SAIN Tool Suites

SAIN has been populated with components from three separate tool suites that support architecture recovery and analysis, have been used in a variety of empirical studies, and are publicly available: *Titan*, a tool suite that extracts representations called Design Rule Spaces (*DRSpaces*) that bridges the gap between architecture and defect prediction [52], [57], [79], [80]; *Archie*, a tool suite that automates the creation and maintenance of architecturally-relevant trace links between code, architectural decisions, architectural tactics, and related requirements [23], [81]–[84]; and *Architecture Recovery, Change, And Decay Evaluator (ARCADE)*, a tool suite that employs a collection of architecture-recovery techniques and a set of metrics for measuring different aspects of architectural change [28], [55], [63], [85].

Archie: Tactic Detection. Archie [23], [86] is a reverse engineering method that detects architectural tactics. It detects security tactics, such as audit, authenticate, HMAC, Secure Session Management, and RBAC; reliability tactics, such as heartbeat and CheckPoint; and performance tactics, such as Resource Pooling, Resource Scheduling, and Asynchronous Invocation [84]. Archie leverages machine learning and structural analysis techniques to identify tactics and map them to code snippets, classes, or source files.

Titan is a tool suite for bridging the gap between software architecture and maintenance quality [79], built upon the design rule theory proposed by Baldwin and Clark [87]. It captures the architecture of a software system as multiple, overlapping design spaces, called *Design Rule Spaces* (*DRSpaces*). Each *DRSpace* is composed of a leading file, which is the design rule of the space, and a set of member files that structurally depend on the leading file, directly or indirectly. In addition, Titan also models the history coupling between source files—how frequently they change together in revision commits—as an additional layer of architectural connections. Titan can identify and rank the *DRSpaces* in a project which aggregate the error-prone files—thus these *DRSpaces* are called the *Architectural Roots* (*ArchRoots*), which deserve attention from practitioners interested in addressing the long-term maintenance quality of a project.

ARCADE: Smell Detection. ARCADE's smell-detection component can identify architectural smells that contribute to maintenance difficulties in a project [55], [64], [88]. The definition of a subset of those architectural smells, which are focused on later in this section, and their potential negative impacts are described below: 1) **Dependency Cycle** occurs when a set of components (e.g. classes or source files) whose links form a circular chain, causing changes to propagate from one component to another on the chain. Such high coupling between components violates design principles for modularity. 2) **Link Overload** manifests when a component has interfaces involved in an excessive number of links (e.g. procedure-call dependencies), affecting the system's separation of concerns and isolation of change. 3) **Concern Overload** occurs when a component implements an excessive number of concerns, violating the principle of separation of concerns, potentially increasing the size of a component and reducing its maintainability.

B. Current SAIN Components

SAIN's components are divided into the four types described in Section IV-C: architecture recovery, architectural analysis and metrics, fact extractors, and utilities. These components may be part of the web-based integration capability or reproducibility wizards of *SAIN*, available in the form of microservices, or as their original source or binaries. As of the writing of this paper, *SAIN* contains 13 components for architecture recovery, 5 components for architectural analysis and metrics, 2 fact extraction components, and 4 utility components available as part of *SAIN*'s web-based integration environment—from three different tool suites. 6 architecture recovery techniques, 8 architectural analyses and metrics, 2 fact extractors, and 9 utility components are available as microservices. These components allow for recovery of other components, architectural tactics, and DRSpaces; architectural analysis and measurement of architectural tactics, architectural smells, defects, change-proneness, etc.; fact extractors for structural dependencies, natural language processing-based information, issue repository extraction, etc.; and utilities for data-format manipulation, visualization, etc.

SAIN includes extensive documentation describing the purpose of each individual component from a tool suite, its inputs, outputs, and links to publications describing the tool suite further. Our users have so far found the documents ease the burden of understanding each individual component, as opposed to the tool suite as a whole, or even standalone tools of each tool suite. *SAIN*'s design that forces tool authors to utilize a standardized form of documentation requiring descriptions of inputs and outputs eased user understanding of *SAIN* components.

The variety of components available in the form of visual integration mechanisms, microservices, or individually downloadable component source or binaries has also allowed multiple students to use tools from outside their research group with greater ease and a shorter learning curve. More specifically, several research groups have re-used *SAIN* components that originate outside of their own research group for novel experiments. We elaborate on two of these experiments in Sections V-C and V-D.

C. An Experiment to Identify Architectural Smells

To conduct research to identify architectural smells in open-source projects that use or implement AI/ML, we leveraged the tool integration module of *SAIN*. Figure 4 shows the *SAIN* experiment template for this study. We leveraged components that were already deployed in *SAIN* to design an experiment which takes a project jar file as input and produces a CSV file that lists architectural smells in each module of the project. First, we used the Dependency Builder component that is part of ARCADE to extract the dependencies in a given project. Then, we used ACDC [29], a widely used architecture recovery technique available in *SAIN*, to discover clusters that follow patterns commonly observed in decompositions of software systems and recover module views of a software system's architecture. We included the Smell Detection component of ARCADE, which takes the outputs of the Dependency Builder and ACDC as input and generates an XML file that lists the identified architectural smells. Finally, the Smell Analyzer component is used to deserialize the output of the smell detector and generate a CSV file that lists class- and component-level smells. These aforementioned four components were connected using the simple and user-friendly drag-and-drop tool integration interface of *SAIN* and the designed experiment was saved as a JSON file for later use.

The tool integration module of *SAIN* allows users to export and import JSON files created for experiments. The smell detection experiment was imported to identify the architectural smells in Mage [77], a game engine project. Mage was downloaded as a zip file and provided as input to the first component in the experiment. It took us around two minutes to extract various types of component- and class-level architectural smells in the Mage project. The output of the components at each step of the experiment was visualized in the component panel and generated log files were accessible through a back-end terminal output window to trace and debug any issues throughout the experiment process—which is accessible through the >_ UI element in the upper-right region of Figures 3 and 4. Using the import and export features of *SAIN*, it was possible to (1) repeat a previous experiment with the same input or re-run it for additional projects and (2) share the experiment JSON file with other members of the research team to reproduce our experiment results at any time. The rich and user-friendly experimentation environment of *SAIN* helps make the results of scientific experiments reproducible and supports research transparency.

D. Integrating SAIN Components

To further showcase our experience with *SAIN*, we elaborate on a preliminary case study accomplished by integrating *SAIN* components, which were originally produced by different research teams. This case study shows how *SAIN* can help researchers achieve a result whose sum is greater than its parts in software architecture analysis with flexible and versatile functions while obtaining interesting, novel research findings.

Our case study subject is an open-source project, Hadoop, which is actively and widely used and maintained. We analyze Hadoop by integrating the insights from the three different tool suites, (i.e. ARCADE, Archie, and Titan) currently available in *SAIN*. We investigate Hadoop version 2.5.0, since it is a major release, and has been previously analyzed by all three tool suites. Figure 3 depicts this experiment as realized in *SAIN*.

1) Integrating ArchRoots and Architectural Tactics: We began our experiment by integrating Titan's architectural root detection and Archie's architectural tactic detection (*TacticDetection* in Figure 3) to enable more advance analyses.

Integration Motivation and Rationale: We aim to integrate the insights of *ArchRoots* and architectural tactics. To this end, we can achieve a multi-perspective view of *ArchRoots* from their architecture design structure, error-proneness, and involvement in tactic implementation. This view helps us answer questions such as the following: How much are the architectural tactics associated with error-proneness? How are tactic files and error-prone files architecturally connected to each other?

To answer these questions, we aim to identify *DRSpaces* that are led by tactic files using the *ArchRoots* detection component. This component exhaustively searches for all the *DRSpaces* that are led by each and every source file in a system as the leading file. This tends to identify large spaces that have the largest coverage on error-prone files. However, these spaces do not necessarily have a focus on tactics. While in this integration case study, Titan only searches for the *DRSpaces* that are led by tactic files. Therefore, the key aim of our study guides us as to how and to what extent the architectural tactics are associated with error-proneness by focusing on source files that are impacted by the tactic files as their "design rules". We refer to the *ArchRoots* associated with tactics from the integration study as *Tactic-ArchRoots*, extracted by the *Tactic RootCover* component in Figure 3.

Our study's results are illustrated in Figure 5. The x-axis shows the ranking of the top x *ArchRoots* detected by Titan. The y-axis shows the coverage of the top x *ArchRoots* to the $Error_5$ space (i.e. the set of error-prone files with at least 5 bug fixes in the revision history for Hadoop-2.5.0). The rectangular data points represent the original *ArchRoots*; while the diamond-shaped data points represent the *Tactic-ArchRoots*. We make the following observations from this result:

Finding 1: The tactic implementation is non-trivially associated with the error-proneness in Hadoop-2.5.0—38% of the error-prone files are aggregated in design spaces that are led by the tactic files. Therefore, it is important for practitioners to investigate the error-proneness of a project from the perspective of tactic implementation.

The top five *ArchRoots*, considering *DRSpaces* led by any source file, can cover 80% of files in $Error_5$. The implication is that error-prone files are significantly linked to each other through their architectural connections. This is consistent with previous findings [57]. While the maximal coverage of the $Error_5$ by the *Tactic-ArchRoots* reaches up to 38% with a total of 30 *Tactic-ArchRoots*. This large coverage of error-prone files by *Tactic-ArchRoots* indicates that error-proneness of Hadoop-2.5.0 is non-trivially associated with the tactic files. Practitioners should examine this association through *Tactic-ArchRoots* when trying to improve the maintenance quality of the 38% of files in $Error_5$.

The top five *Tactic-ArchRoots* already cover 31% (out of the maximal 38%) of all *Tactic-ArchRoots*. Therefore, we suggest that developers prioritize the top five *Tactic-ArchRoots* when examining the relationship between error-proneness and tactic

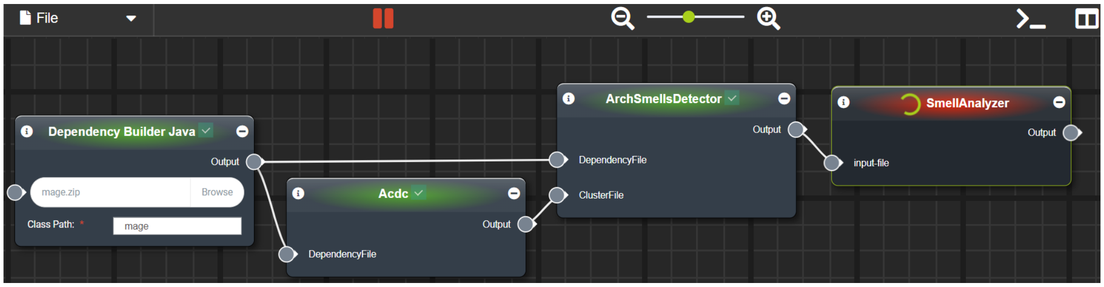
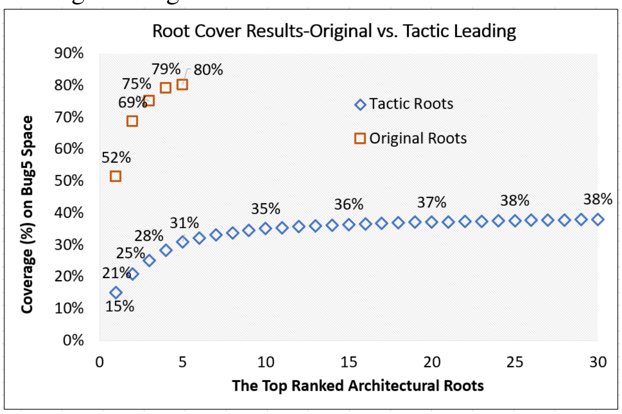


Fig. 4: An experiment designed to identify architectural smells from architectures recovered by ACDC. The first three components have finished executing, while *Smell Analyzer* is currently running.

Fig. 5: Original ArchRoots vs. Tactic ArchRoots



implementation. In Table I, we list the characteristics of the top five *Tactic-ArchRoots*. The first column shows the tactic leading files of the identified *Tactic-ArchRoots*. The second and the third columns show the frequency and ranking of each tactic leading file for fixing bugs. The last two columns describe the characteristics of the *Tactic-ArchRoots*, in terms of *BSC* and *DSB* measurements. *BSC* is the percentage of files in $Error_5$ that are covered in a root; *DSB* is the percentage of files in a root that are from $Error_5$. Table I shows that the top five tactic leading files could be very error-prone: *UserGroupInformation*, *CommonConfigurationKeysPublic*, and *FileContext* are highly ranked for their bug fixing frequency. These *Tactic-ArchRoots* have a high concentration of error-prone files in Hadoop-2.5.0. For example, the *DSB* of the ArchRoot led by *FileContext* reaches up to 54%, indicating every one in two files in this root contain more than five bug fixes. Such insights would not be available without the integrated analysis of Titan and Archie.

Leading Tactic File Info			Root Info	
Tactic File	B. Freq.	B. Rank	BSC	DSB
UserGroupInformation	31	3	15%	33%
CommonConfigurationKeysPublic	14	12	11%	45%
MiniDFSCluster	3	128	7%	19%
FileContext	13	15	6%	54%
Token	2	206	8%	35%

TABLE I: Top Five Tactic ArchRoots

Finding 2: The top five *Tactic-ArchRoots* deserves special attention from the developers, since they strongly relate to the error-prone files—i.e., 19% to 54% of files in each *Tactic-ArchRoot* is error-prone as measured using DSB. The top five *Tactic-ArchRoots* can provide a useful perspective for examining the association between tactic implementation and error-proneness in a project.

2) Integrating Tactic-ArchRoots and ARCADE Smell Detection: For the next step of our case study, we integrated ARCADE's architectural smell detection with *Tactic-ArchRoots*. In other words, we aim to investigate whether and to what extent the most error-prone *Tactic-ArchRoots* from the above integration also suffer from architectural smells. This integration of architectural smells and *Tactic-ArchRoots* is realized by *Smell RootCover*, which takes *ACDCWithSmellDetection*, the smell detector of ARCADE based on ACDC, and *Tactic RootCover* as input—all of which are depicted in Figure 3. In Hadoop-2.5.0, we identified three types of smells: *Dependency Cycle* (1 instance), *Link Overload* (5 instances), and *Concern Overload* (1 instance). The architectural smells detected by ARCADE can have negative impacts on the maintainability and software quality, which in turn can increase the error-rate of the components involved in the smells. For instance, if one of the components involved in a *Dependency Cycle* contains an error, a change fixing the error can propagate changes to other components in the smell.

Integration Motivation and Rationale: Practitioners can gain valuable insights by viewing *Tactic-ArchRoots* and the architectural smells in combination. In particular, for our case study, we are interested in answering the following question: How are the source files in the top five *Tactic-ArchRoots* involved in the architectural smells? This can potentially help developers to reveal the underlying architectural design flaws that lead to high error-proneness of *Tactic-ArchRoots*.

The integration rationale is that, for *Tactic-ArchRoots*, we investigate how each instance of an architectural smell, detected by ARCADE, is contained in the roots. Note that an architectural smell instance is usually composed of a group of source files. For example, we identified a *Dependency Cycle* formed by 78 source files in Hadoop-2.5.0. A *Tactic-Root* alone may not contain all the files of a smell. Thus, we calculate the percentage of files in each architectural smell instance that are also contained in a *Tactic-ArchRoots*. In addition, we calculate the percentage of files in each architectural smell instance that are contained and aggregated in the top x *Tactic-ArchRoots*.

Table II presents the overview of the integration analysis of combining *Tactic-ArchRoots* and architectural smells.

Finding 3: Overall, among a total of *seven* instances of architectural smells in Hadoop-2.5.0, *five* instances are involved in the *Tactic-ArchRoots*. This indicates that reviewing the *Tactic-ArchRoots* is important to investigate most (5/7) architectural smells in Hadoop-2.5.0.

The detailed analysis of each involved architectural instance is shown as a column in Table II. When considering all the

Top Five Tactic-ArchRoots	ARCADE Smell Instance (# Files)									
Tactic Leading File	Dp. Cycle_1 (# 78)		Link OL._1 (# 33)		Link OL._2 (# 3)		Link OL._3 (# 18)		Link OL._4 (# 1)	
	Single	Accu	Single	Accu	Single	Accu	Single	Accu	Single	Accu
UserGroupInformation	29%	29%	3%	3%	100%	100%	11%	11%	0%	0%
CommonConfigurationKeysPublic	3%	31%	3%	6%	0%	100%	0%	11%	100%	100%
MiniDFSCluster	0%	31%	0%	6%	0%	100%	0%	11%	0%	100%
FileContext	1%	32%	0%	6%	0%	100%	6%	17%	0%	100%
Token	32%	38%	0%	6%	100%	100%	6%	17%	0%	100%
All Tactic-ArchRoots	38%		6%		100%		17%		100%	

TABLE II: Smells Identified by ARCADE in *Tactic-ArchRoots*

identified *Tactic-ArchRoots* (a total of 30 roots as shown in Figure 5), they contain 6% to 100% of the files in different architectural smell instances, as shown in the last row. This indicates that different architectural smells have different level of associations with *Tactic-ArchRoots*. In addition, if we focus on the top five *Tactic-ArchRoots*, we notice that the maximal percentage (6% to 100%) of files have already been covered for each smell instance.

Finding 4: These results indicate that developers only need to focus on the top five *Tactic-ArchRoots* for understanding how the architectural smells overlap with the tactic implementation and their error-proneness.

To illustrate in greater depth the interesting results that can be obtained by using *SAIN* to integrate various architectural tools, we present a qualitative example and visualization to show how combining *Tactic-ArchRoots* and smell analysis helps developers understand the root causes of error-proneness of the top ranked *Tactic-ArchRoots*. Figure 6 is a part of the Design Structure Matrix (DSM) visualization of the top ranked *Tactic-ArchRoot* led by tactic file *UserGroupInformation*. Due to space limitations, we only illustrated part of the space that focuses on the tactic implementation for *Authenticate*, which ensures that a user or a remote system is who it claims to be.

A DSM is an $n \times n$ square matrix, which represents the relationship among source files in a system. As shown in Figure 6, the rows and the columns represent the source files from the *Tactic-ArchRoot* led by *UserGroupInformation* (this leading file is listed in row 1). The relationship among files is captured in the $n \times n$ square matrix—found in the outer rectangular box on the right of Figure 6. Titan captures two types of relationship between files in the DSM: (1) structural dependencies, including "ext" and "dp"—where "ext" indicates that the file on the row extends the file on the column, while "dp" represents all other general types of reference relationships, such as method call and variable declarations; (2) the historical coupling, captured as a numeric value, indicating the number of times the file on the row changes together with the file on the column in the same commits. For example, cell[3,2] says "dp,36", indicating the file on row 3 *ipc.Cline* depends on the file on row 2 *ipc.Server*, and they change together 36 times in the same commits. This indicates strong coupling between *ipc.Server* and *ipc.Client*. To make reading Figure 6 more intuitive, we color-coded the cells based on the weight of the historical coupling between files, where darker shades of red indicate a larger number of co-occurring commits. The DSM visualization helps us to gain insights regarding both the structural and historical coupling among files in a system.

Using the basic DSM, we integrated three additional aspects of information for each involved source file in the space: (1) the involved tactic(s) (labelled as column "Tactic"); (2) the involved smell(s) (labelled as column "Smell"); and finally, (3) the error change frequency (labelled as column "E-Freq"), which is color-coded in a heat-map based on the value—darker shades of red indicate a higher error change frequency. This additional information helps us to investigate how different tactic files are coupled with each other both structurally and historically, and how they are involved in architectural smells which, in turn, provide insights regarding the root cause of the error-proneness of *Tactic-ArchRoots*.

For example, through this integrated DSM visualization in Figure 6, we have the following overall finding:

Finding 5: The complicated structural and historical coupling among the tactic files tend contribute to the error-proneness of *Tactic-ArchRoot-1* in Hadoop 2.5.0. There appears to be little-to-no relationship between error-proneness and architectural smells for *Tactic-ArchRoot-1* in Hadoop 2.5.0.

VI. Discussion and Lessons Learned

To realize the current version of *SAIN*, our team of several developers and research groups faced major development challenges. More specifically, we have a geographically distributed team across three different continents and 13 different time zones. We found that agile methods with two-weeks sprints, joint application design (JAD) session [89], and exploratory prototyping of design alternatives worked effectively to develop *SAIN* under these circumstances.

For many tools or components in *SAIN*, a variety of data types are used, from general types such as XML and JSON to specific types for sub-domains of architecture research, such as the Rigi Standard Format [29], [88], [90]–[92] for clustering-based architecture recovery. The long-standing problem of data conversion has required the construction of new utility components that act as adaptors or wrappers. Nevertheless, we have found that building these conversion tools has not been a major pain point for researchers using *SAIN* compared to the high variability of formats. Once these utility components or connectors are built, they can be easily contributed back to *SAIN*.

As a result, we aim to include support mechanisms to aid documentation of datasets, benchmarks, and their metadata—in a similar manner as we have done for tools and their components—which itself has already eased the burden of interoperability. We believe that this challenge further emphasizes the need for architecture researchers to address problems of disparate data types, possibly through flexible languages that can describe current architectural phenomena with mechanisms allowing for incorporation of future phenomena. To that end, extensible architectural languages such as ACME [93], [94] or xADL [95], [96] may be a promising starting point.

Each *SAIN* component can take a wide variety of input options or complex configuration files. Incorporating these components into our plug-and-play integration tool or creating microservices out of them aided in determining the best default options or the key options for components of a tool suite.

ID	File Name	Tactic	Smell	E-Freq	1	2	3	4	5	6	7	8	9	10	11
1	UserGroupInformation	Authenticate, RBAC		31	(1)	,13	,9				dp,11	dp		dp	
2	ipc.Server	Authenticate, RBAC		51	dp, 13	(2)	,36			dp,11	dp,	dp	dp	dp, 14	,11
3	ipc.Client	Authenticate, RBAC		40	dp, 9	dp,36	(3)			,6	dp,7			dp, 6	dp, 12
4	FSNamesystem	Authenticate, RBAC		3	dp	dp		(4)	,7			dp	dp		
5	JspHelper	Authenticate, RBAC		0	dp			dp, 7	(5)		dp,				
6	ServiceAuthorizationManager	Authenticate, RBAC		3	dp	,11	,6			(6)	dp,6				
7	security.SecurityUtil	Authenticate	cycle	6	dp, 11		,7			,6	(7)				
8	security.token.TokenIdentifier	Authenticate	cycle	0	dp							(8)			
9	security.token.SecretManager	Authenticate		0								ext, dp	(9)		
10	security.SaslRpcServer	Authenticate		12	dp	dp,14	,6					ext, dp	dp	(10)	,6
11	security.SaslRpcClient	Authenticate		14	dp	dp,11	,12				dp,	dp		dp, 6	(11)

Fig. 6: *Tactic-ArchRoot-1* led by *UserGroupInformation* with Smells. Numbers along the diagonal refer to the ID of each file.

The visual nature of our plug-and-play integration mechanism made it easy to identify the key input options that users must supply (e.g., a zipped directory), without even having to look extensively at existing documentation, which could be ample for some of the tool suites. As a result, our experience encourages wider use of visual or block-based paradigms for creating novel experiments, re-using them, or sharing them.

Moreover, we found that students, developers, and researchers could easily try out and combine various components from tool suites they never tried before. Although this did not completely eliminate integration challenges (e.g., the need to create new data conversion components or modify existing components), *SAIN* made it easier to identify these issues for novel experiments that users wished to run.

The plug-and-play integration panel provided by *SAIN* allows a novice researcher to quickly become familiar with the workflow of different tool suites. For example, the integration experiment was driven by a third year Ph.D. student who had no prior experience with Archie or ARCADE, and only had very limited experience with Titan components. He was able to accomplish the integration case study in a time frame of two weeks. He finally ended up contributing two new analysis components, built upon existing components. This would not be possible without the support of *SAIN*.

Due to the experience described above with our plug-and-play mechanisms, composition of experiments was significantly eased. Other challenges remained, however. For instance, debugging an error in experiment can be more challenging due to a *SAIN* user being unable to set breakpoints and step through a program to diagnose or fix a bug. Running on a remote machine (e.g., *SAIN* server), as opposed to a local machine can create unexpected delays or slowdowns. Nevertheless, *SAIN* developers have managed to overcome many of these initial issues by providing research prototype interoperability mechanisms, allowing various users across several research groups to more easily and quickly learn and use architecture-oriented tools and components from outside their respective groups.

The *SAIN* platform provided a comprehensive view of different architectural instruments that are available. It allows the researchers to think out-of-the-box about the potential connections and integration opportunities among different components that were initially developed by independent teams. These connections and opportunities only became explicit and available when different architectural instruments were organized and reviewed together.

VII. Conclusion

Over three decades of software engineering research aimed at tackling the problem of architectural decay has resulted in a plethora of techniques and tools to address the problem. Researchers attempting to address this long-standing architecture problem face enormous challenges behind tool reuse and accessibility, tool interoperability, and reproducibility of experiments and analyses using these tools. To address these three major challenges, we have constructed *SAIN*, a first-of-its-kind framework for assembling tools to support architecture-based software maintenance. *SAIN* comprises a library of cutting-edge tools for architecture recovery and analysis, a plug-and-play instrument for integrating tools, and reproducibility wizards to support replication of architecture-based research studies. We make *SAIN* publicly available for researchers and practitioners at [10].

We have discussed our experience of *SAIN* and our users' experiences of *SAIN* in terms of the three tool suites; 13 architecture recovery components, 8 components for computing architectural metrics or analyses, 2 fact extractors, and 9 utility components; one compact case study and a detailed case study of our users running novel experiments using *SAIN* and how it eased the process for them; and the results of the detailed case study, which analyzes the relationships between architectural smells, architectural tactics, and error-proneness. This detailed case study resulted in 5 major findings that can aid architects interested in improving maintainability of their systems by simply focusing on a small set of *Tactic-ArchRoots*.

It is an open challenge to determine how to provide mechanisms that (1) ease dataset and benchmark inclusion and integration into an experiment and (2) microservice creation for research prototypes or their components. We, therefore, aim to study mechanisms for specifying and integrating datasets and benchmarks into our plug-and-play mechanisms and reproducibility wizard. Although full automation of microservice or containerization is desirable, a highly valuable first step is to design interfaces and supporting software mechanisms that reduces the manual labor needed to create a container for a research prototype or one of its components. We aim for our future work to overcome this challenge.

VIII. Acknowledgements

This work was supported in part by awards CNS-1823262, CNS-1823246, CNS-1823074, CNS-1823354, CNS-1823214, CNS-1823177, CNS-1823074, CCF-1823177, OAC-1835292, CCF-1816594, and CCF-1717963 from the National Science Foundation and the U.S. Office of Naval Research under grant N00014-17-1-2896. We would also like to thank the anonymous reviewers for their valuable feedback, which helped us to improve this work.

REFERENCES

[1] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996, vol. 1.

[2] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2013.

[4] "Analysis: It experts question architecture of obamacare website," http://www.reuters.com/article/us-usa-healthcare-technology-analysis-idUSBRE99407T20131005, 2013.

[5] A. Telea and L. Voinea, "Visual software analytics for the build optimization of large-scale software systems," *Computational Statistics*, vol. 26, no. 4, pp. 635–654, Dec. 2011. [Online]. Available: http://link.springer.com/10.1007/s00180-011-0248-2

[6] T. A. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 494–497, Sep. 1984, conference Name: IEEE Transactions on Software Engineering.

[7] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989, conference Name: IBM Systems Journal.

[8] S. Yau and J. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 545–552, Nov. 1980, conference Name: IEEE Transactions on Software Engineering.

[9] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 573–591, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1109/TSE.2009.19

[10] "Software Architecture INstrument (SAIN)." [Online]. Available: http://sain.usc.edu

[11] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[12] "Bash - GNU Project - Free Software Foundation," https://www.gnu.org/software/bash/, 2016.

[13] A. Brown and G. Wilson, *The Architecture of Open Source Applications*. Lulu. com, 2011, vol. 1.

[14] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 901–910. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486911

[15] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 69–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819022

[16] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 50–60. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786848

[17] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, May 2015, pp. 51–60.

[18] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 179–188.

[19] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE TSE*, 2009.

[20] R. Koschke, "Architecture reconstruction," *Software Engineering*, 2009.

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

[22] P. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, pp. 42–50, November 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=624610.625529

[23] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic centric approach for automating traceability of quality concerns," in *International Conference on Software Engineering, ICSE (1)*, 2012.

[24] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "Automated detection and classification of non-functional requirements," *Requir. Eng.*, vol. 12, no. 2, pp. 103–120, 2007.

[25] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Working Conference on Reverse Engineering (WCRE)*, 1997.

[26] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE TSE*, 2005.

[27] R. Naseem, O. Maqbool, and S. Muhammad, "Improved similarity measures for software clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011.

[28] 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*, 2013, pp. 486–496.

[29] V. Tzerpos and R. Holt, "ACDC: an algorithm for comprehension-driven clustering," in *Working Conference on Reverse Engineering (WCRE)*, 2000.

[30] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE TSE*, 2006.

[31] N. Anquetil and T. Lethbridge, "File clustering using naming conventions for legacy systems," in *Conference of the Centre for Advanced Studies on Collaborative Research*, 1997.

[32] ——, "Recovering software architecture from the names of source files," *Journal of Software Maintenance: Research and Practice*, 1999.

[33] J. Misra, K. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus, "Software clustering: Unifying syntactic and semantic features," in *Working Conference on Reverse Engineering (WCRE)*, 2012.

[34] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *ASE*, 2011.

[35] A. Corazza, S. Di Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[36] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

[37] K. Wong, S. Tilley, H. Muller, and M. Storey, "Structural redocumentation: A case study," *Software, IEEE*, vol. 12, no. 1, pp. 46–54, 1995.

[38] R. Kazman and S. Carriere, "View extraction and view fusion in architectural understanding," in *Software Reuse, 1998. Proceedings. Fifth International Conference on*, Jun. 1998, pp. 290 –299.

[39] R. Kazman and S. J. Carrière, "Playing detective: Reconstructing software architecture from available evidence," *Automated Software Engineering*, vol. 6, pp. 107–138, 1999, 10.1023/A:1008781513258. [Online]. Available: http://dx.doi.org/10.1023/A:1008781513258

[40] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *ICSE*, 1991.

[41] R. C. Holt, "Structural manipulations of software architecture using tarski relational algebra," *Reverse Engineering, Working Conference on*, vol. 0, p. 210, 1998.

[42] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, vol. 81, no. 4, pp. 536 – 557, 2008, ¡ce:title¿Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006)¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412120700194X

[43] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 109–120. [Online]. Available: http://portal.acm.org/citation.cfm?id=1130239.1130657

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

[45] G. Guo, J. Atlee, and R. Kazman, "A software architecture reconstruction method," in *WICSA*. Springer Netherlands, 1999, p. 15.

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

[47] H. M
"uller, M. Orgun, S. Tilley, and J. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[48] N. Medvidovic and V. Jakobac, "Using software evolution to focus architectural recovery," *Automated Software Engineering*, vol. 13, pp. 225–256, 2006, 10.1007/s10515-006-7737-5. [Online]. Available: http://dx.doi.org/10.1007/s10515-006-7737-5

[49] M. Shtern and V. Tzerpos, "Evaluating software clustering using multiple simulated authoritative decompositions," in *ICSM*. IEEE, 2011, pp. 353–361.

[50] ——, "On the comparability of software clustering algorithms," in *ICPC*. IEEE, 2010, pp. 64–67.

[51] ——, "A framework for the comparison of nested software decompositions," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2004.

[52] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 967–977.

[53] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *QoSA '09: Proc. 5th Int'l Conf. on Quality of Software Architectures*, 2009.
[54] J. Garcia, D. Popescu, G. Edwards, and M. Nenad, "Identifying Architectural Bad Smells," in *13th European Conference on Software Maintenance and Reengineering*, 2009.
[55] J. Garcia, "A unified framework for studying architectural decay of software systems," Ph.D. dissertation, University of Southern California, 2014.
[56] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.
[57] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: A new model for representing and analyzing software architecture," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 657–682, 2018.
[58] J. B. Tran and R. C. Holt, "Forward and Reverse Repair of Software Architecture," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 12–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782007
[59] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *Automated Software Engineering*, vol. 8, no. 1, pp. 89–120, 2001. [Online]. Available: http://link.springer.com/article/10.1023/A:1008715808855
[60] J. Philipps and B. Rumpe, "Refinement of information flow architectures," in *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*. IEEE, 1997, pp. 203–212.
[61] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.
[62] I. Ivkovic and K. Kontogiannis, "A Framework for Software Architecture Refactoring Using Model Transformations and Semantic Annotations," in *Proceedings of the Conference on Software Maintenance and Reengineering*, ser. CSMR '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135–144. [Online]. Available: http://dl.acm.org/citation.cfm?id=1116163.1116398
[63] D. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," *To appear in the 12th Working Conference on Mining Software Repositories*, 2015.
[64] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 176–17 609.
[65] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality," in *ACM Student Research Competition, 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013, pp. 1450–1452.
[66] (2016) Architectural decay prediction. https://seal.ics.uci.edu/projects/decayprediction/.
[67] M. Galster and D. Weyns, "Empirical research in software architecture: How far have we come?" in *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. IEEE, 2016, pp. 11–20.
[68] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *Empirical Software Engineering*, vol. 17, no. 1, pp. 75–89, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9181-9
[69] R. Koschke, "Incremental reflexion analysis," in *CSMR*, 2010.
[70] (2016) Promise software engineering repository. http://promise.site.uottawa.ca/SERepository/.
[71] (2016) Eclipse bug data. https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/.
[72] (2016) Bug prediction dataset. http://bug.inf.usi.ch/.
[73] (2016) Software-aritfact infrastructure repository. http://sir.unl.edu/portal/index.php.
[74] (2016) Tracelab. http://www.coest.org/index.php/tracelab.
[75] "Blockly." [Online]. Available: https://developers.google.com/blockly
[76] A. McCallum, "Mallet: A machine learning for language toolkit," 2002.
[77] "magefree/mage," Dec. 2020, original-date: 2012-04-27T13:18:34Z. [Online]. Available: https://github.com/magefree/mage
[78] "Apache Hadoop." [Online]. Available: https://hadoop.apache.org/
[79] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 763–766.
[80] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 488–498.
[81] M. Mirakhorli, "Tracing architecturally significant requirements: a decision-centric approach," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1126–1127. [Online]. Available: http://doi.acm.org/10.1145/1985793.1986014
[82] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic-centric approach for automating traceability of quality concerns," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 639–649. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337298
[83] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai, "A study on the role of software architecture in the evolution and quality of software," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 246–257. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820548
[84] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitoring architectural tactics in code," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.
[85] 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*, vol. 22, no. 3, pp. 1146–1193, 2017.
[86] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang, "Archie: A tool for detecting, monitoring, and preserving architecturally significant code," in *CM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.
[87] C. Y. Baldwin, K. B. Clark, K. B. Clark *et al.*, *Design rules: The power of modularity*. MIT press, 2000, vol. 1.
[88] M. Schmitt Laser, N. Medvidovic, D. M. Le, and J. Garcia, "Arcade: an extensible workbench for architecture recovery, change, and decay evaluation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1546–1550.
[89] E. Carmel, R. D. Whitaker, and J. F. George, "Pd and joint application design: A transatlantic comparison," *Commun. ACM*, vol. 36, no. 6, p. 40–48, Jun. 1993. [Online]. Available: https://doi.org/10.1145/153571.163265
[90] "CCVisu Tutorial." [Online]. Available: https://ccvisu.sosy-lab.org/manual/main003.html
[91] K. Wong, "Rigi user's manual," *Department of Computer Science, University of Victoria*, 1998.
[92] H. A. Muller, "RIGI: A MODEL FOR SOFTWARE SYSTEM CONSTRUCTION, INTEGRATION, AND EVOLUTION BASED ON MODULE INTERFACE SPECIFICATIONS," Thesis, 1986, accepted: 2007-05-09T19:46:02Z. [Online]. Available: https://scholarship.rice.edu/handle/1911/16096
[93] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," *Foundations of component-based systems*, vol. 68, pp. 47–68, 2000.
[94] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *CASCON First Decade High Impact Papers*, 2010, pp. 159–173.
[95] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "A highly-extensible, xml-based architecture description language," in *Proceedings Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2001, pp. 103–112.
[96] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 2, pp. 199–245, 2005.