An Iterative Framework for Software Architecture Recovery: An Experience Report

Banani Roy and T.C. Nicholas Graham

Queen's University, Kingston, Ontario, Canada K7L 3N6
{broy,graham}@cs.queensu.ca

Abstract. Both architecture recovery and architecture evaluation play an important role in the area of software reverse-engineering. In this paper, we propose and evaluate a framework for incremental and iterative application of automated architecture recovery (using SWAG Kit) and architecture analysis (using SAAM.) We conclude that SWAG Kit helps in generating a low-level architecture that forms the basis of analysis, while SAAM helps in deriving from this a deeply understood conceptual architecture. The process is iterative, where SAAM analysis helps refine the parameters fed to SWAG Kit, in turn leading to a superior architecture for further analysis. We have applied this process to the extraction of the architectures of three open source compression tools, and we report on the strengths and weaknesses of the approach that this case study exposed. Over all, we conclude that the framework allowed us to understand the software architectures more deeply than would have been possible with the software architecture recovery process alone.

Keywords: Software Architecture Recovery, SAAM, SWAG Kit, Iterative Framework, Evaluation.

1 Introduction

Legacy software systems often lack adequate architectural documentation. When present at all, architectural documentation is often inconsistent with the current state of the system [22,6]. Lack of architectural documentation can make it difficult to bring new developers into the project or to methodically analyze the effect of proposed architectural changes. To address this problem, numerous researchers have proposed the use of automated tools to recover the architecture of a system from its source code [6,19,14]. Architecture recovery tools such as Rigi [27], Shrimp [29], SWAG Kit [30] and Dali [17] automate parts of the process, requiring human guidance to create documentation of the architecture of software systems.

Despite decades of research and considerable progress in the development of such tools, they have yet to obtain wide-spread industrial adoption. In this paper, we argue that the quality of software architectural recovery can be improved by applying systematic analysis to the architectures generated by recovery tools. Architectural analysis helps identify the questions that we wish our description of

R. Morrison, D. Balasubramaniam, and K. Falkner (Eds.): ECSA 2008, LNCS 5292, pp. 210–224, 2008.
© Springer-Verlag Berlin Heidelberg 2008

an architecture to answer, in terms of non-functional requirements such as modifiability, security, usability and availability [1,3,7]. In this paper, we present a framework for iteratively applying architectural recovery and architectural analysis. We present our experience in applying this method to the recovery of the architectures of three open-source compression toolkits. To our knowledge, this is the first experience report directly reporting on the benefits and limitations of combining these two techniques.

In our study, we examine whether value is added to the process of architectural recovery by using an architecture evaluation method in addition to a recovery tool. I.e., we address the question of wheher an architectural analysis helps provide a deeper understanding of the architecture in the recovery process, and whether it helps obtain a more accurate view of the architecture.

To help evaluate these questions, we have developed a framework that iteratively combines the SWAG Kit architecture recovery tool [30] with the SAAM architecture evaluation method SAAM [7,16]. In this framework, a recovery tool is first used to obtain a low-level architectural representation from the system's source. An architecture evaluation method is then applied to the extracted system representation. The automated recovery and analysis steps are iteratively applied until an acceptable architectural description is obtained.

We have applied this method to three open source compression applications/libraries: *ZLib* [32], *ZDelta* [31] and *GZip* [15]. We applied SWAG Kit and SAAM to each application using the iterative framework. Through this, we are able to evaluate the benefits and weaknesses of the approach.

From the case study, we have learned several interesting lessons. We found that software architecture recovery is weak at identifying subsystem structures. In our experience, automatic decomposition of an architecture into subsystems did not contain the information a programmer needs to answer maintenance questions. E.g., the subsystem structure generated using SWAG Kit for *ZDelta* failed to identify its encryption/decryption units. SAAM helps in refining subsystem structure by helping to identify the questions that the architecture must answer. For example, maintenance scenarios quickly identified the importance of encryption/decryption in *ZDelta*, identifying the need to refactor the subsystem decomposition.

We found that SWAG Kit is limited to producing a static architectural view comprising components and connectors. SAAM evaluation requires a deeper understanding of how architectural components collaborate to accomplish a specific task suggested by SAAM's scenarios. Both of these approaches benefited from the iterative application of SWAG Kit and SAAM; as understanding of the architecture increased via analysis, it was possible to improve the architecture generated by SWAG Kit, which in turn resulted in improvements in the analysis.

Our experience shows that this iterative method is tractable for modestly sized systems. The three compression applications to which we applied the method were each approximately 10,000 lines of code in length. It took about one person-week per application to apply the method and extract an architecture. When scaling the approach to larger systems, we believe that the required time will vary

based on the architecture recovery tool and its browsing facilities, the domain of the target application, the source code structure, the quality of comments in the source code and the availability of good documentation.

Our experience shows that the architecture recovery team and the architecture evaluation team should work closely together (or even be the same team), since there is a tight and iterative interaction between the architecture recovery tool and the evaluation process.

There are some limitations to this combined approach. For example, it was difficult to define a terminating point for the iterative process. Also, it was challenging to define appropriate scenarios for the evaluation process; if wrong scenarios are chosen, the final architecture may remain unsuitable for future analysis tasks. On balance, we conclude that despite these limitations, by iteratively applying architectural recovery and analysis, it is possible to gain a strong understanding of a software architecture with modest time investment.

The organization of the paper is as follows. In section 2, we review other techniques combining architectural recovery and evaluation. Section 3 explains how SWAG Kit and SAAM can be combined to extract a software architecture. In section 4, we illustrate the lessons that we learned from our case study, in which we applied our method to extract the architectures of *ZLib*, *ZDelta* and *GZip*. Section 5 concludes the paper.

2 Related Work

While there is significant literature on software architectural evaluation [16,10,8] (for a comprehensive summary of all architectural evaluation methods see our technical report [28]), little attention has been paid to its methodical application to architecture recovery. Some methods propose the combination of architecture recovery and architectural evaluation, but these approaches are purely sequential.

Lutz and Gannod [19], for example, have discussed the architectural analysis of a software product-line using a three-phase approach. The phases are software architecture recovery, scenario-based assessment of the extracted architecture and model checking of safety-critical behaviors. In contrast to our iterative approach, Lutz and Gannod use a purely forward approach. The software architecture is manually recovered from the available information and code base, and is compared to an existing software architecture using a scenario-based method. In this approach, the evaluation method plays no role in the recovery process.

A similar approach has been proposed by Bowman *et al.* [6]. This technique is based on dividing the software into subsystems based on its *ownership architecture*. The ownership architectures are then compared with existing conceptual architectures. Their study shows that ownership architecture is a good predictor of concrete architecture and is closely correlated to the conceptual architecture.

In addition to scenario-based approaches, some metrics-based evaluation methods have been applied to evaluating extracted software architectures. Again, in contrast to our iterative approach, these approaches are incremental. For example,

Medvidovic *et al.* [21] have quantitatively and qualitatively evaluated the Focus architectural recovery approach by extracting and validating the two middleware intensive systems: OODT [18] and the Globus Toolkit [12].

Guo *et al.* [14] have proposed a semi-automated architecture reconstruction method. This method uses patterns to guide users in achitectural recovery. This work is similar to ours in that it iteratively applies an architecture recovery tool (the Dali Workbench [17]) and an architecture evaluation technique. Little feedback is provided on the success of this iterative approach; the focus of Guo *et al.*'s approach is on evaluating the pattern matching approach.

We conclude, therefore, that there is room for study of the effectiveness of iterative application of automated architectural extraction and architecture analysis to the task of software architectural recovery.

3 Framework

In this section, we describe our framework for recovering the software architecture of legacy systems. We first explain how SWAG Kit supports automated extraction of software architectures from source code, and then describe how SAAM is used to evaluate the resulting software architecture. We then explain how these two can be applied together to recover meaningful architecture of legacy systems. In section 4, we report on our experience applying this framework to the recovery of the architectures of three open-source compression libraries.

3.1 Automated Software Architecture Extraction

Two types of software architectures are useful for understanding a complex software system: *conceptual* and *concrete*. A *conceptual architecture* provides an abstract view of the system by hiding its implementation details [3]. A *concrete architecture* shows the system as implemented. In this paper, we focus on recovering the conceptual architecture, as it serves our purpose of understanding the components and their relationships of the implemented system. In order to obtain a conceptual architecture, we use an architecture recovery tool to obtain a concrete architecture. This concrete architecture is then evaluated, abstracted and further refined into a conceptual architecture.

Architecture Recovery Steps: The general approach of recovering a software architecture consists of the following steps [26]:

1. Determine the low-level system representation (concrete architecture) by applying the architecture recovery tool on the source code of the target applications.
2. Identify the architectural elements/components by combining domain knowledge, design documents and the extracted low-level system representation.
3. Identify the relationships between the architectural elements to obtain a high level architectural representation of the system.

In the first step of the architectural recovery, we use *SWAG Kit* [30] to automatically extract the low-level system representation from the source code. We chose SWAG Kit because it is a mature toolkit which can be used for extracting, abstracting and exploring software architectures. This tool automatically extracts architectures from calls information in C or C++ source. SWAG Kit provides the *LSEdit* editor for visualizing and refining the architecture.

3.2 Software Architecture Analysis

The second major part of our framework is software architectural analysis. We use the *Software Architecture Analysis Method* (SAAM), as it is a widely-studied scenario-based method, and has been applied to numerous industrial problems.

Users of SAAM first identify the quality attributes of most importance to their application domain. They then elicit scenarios identifying plausible tasks involving the architecture (e.g., modification scenarios, or security attack scenarios.) After that, SAAM analysts determine the degree to which the software architecture has support for those scenarios. Analysts identify a list of changes that the scenarios require and provide necessary guidelines for addressing those changes in the software architecture.

3.3 The Combined Approach

Our combination of automated software architecture extraction and architectural analysis is incremental and iterative. The output of the architecture recovery tool is used as the input to the analysis method, and the analysis results are used to improve the extracted software architecture. Our combined framework is shown in Fig. 1. In the following, we discuss how SWAG Kit can be combined with SAAM using an incremental and iterative approach.

We automatically obtain a low-level system representation using SWAG Kit. The identification of architectural elements and derivation of a conceptual architecture is done manually. This manual analysis draws from domain knowledge, design documents, source code and source code comments; together, these form *architecturally significant concepts*. The overhead of this manual analysis can be significantly reduced if a reference architecture is readily available for the domain of interest [26,13,9].

Architecturally significant concepts, low-level system representation (from the recovery tool), and the reference architecture (if available) are then analyzed to obtain the subsystem structure and eventually, an initial version of the extracted architecture.

We use SAAM to identify shortcomings in the extracted architecture by examining the impact of scenarios on the architecture. SAAM helps find solutions to these shortcomings. This information is fed back to the recovery process. Concretely with SWAG Kit, this means manually modifying the inputs to the *LSEdit* tool that is used to view and refine archtictures.

Since the initial version of the software architecture is extracted using a tool, the architecture extraction team might focus too heavily on the tool output

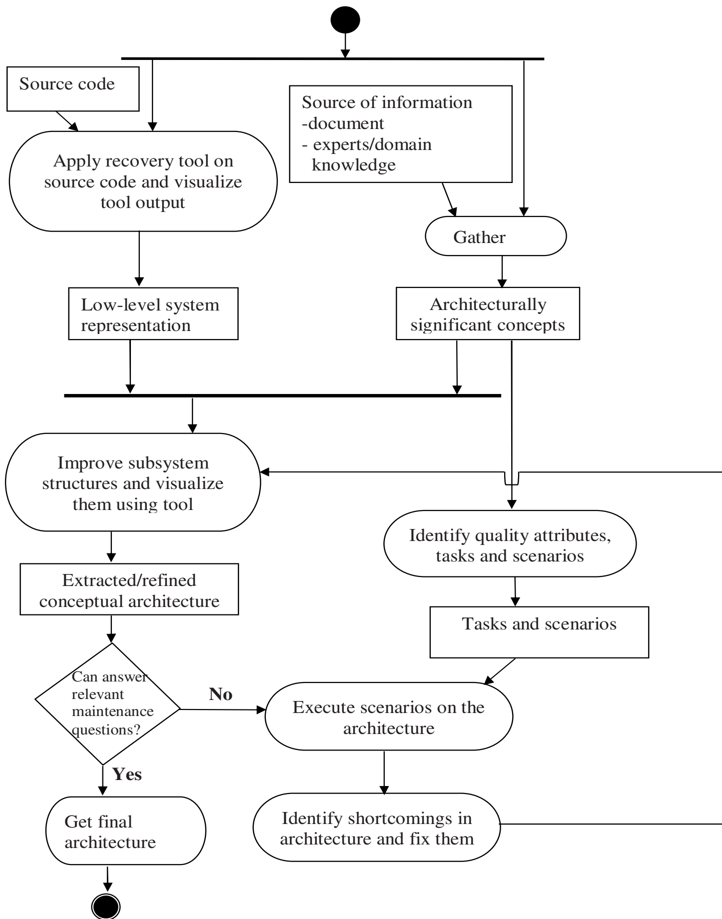


Fig. 1. The framework for combining architecture recovery tool and SAAM

when deriving a high level system representation. For example, the extraction team may fail to correctly identify the attributes that are most important to the application domain, the mechanisms by which these quality attributes are satisfied, and what protocols are used for the components' interaction. As a result, the initially extracted architecture might fail to capture important information required by maintenance programmers. SAAM can help improve this initial architecture by identifying scenarios that address relevant quality attributes.

On the other hand, the tool-supported analysis and visualization facilities of the recovery process can help SAAM analysts. SWAG Kit provides utilities to view the software architecture at a finer granularity, which SAAM can use to fine tune the analysis. For example, the *LSEdit* editor in SWAG Kit provides facilities for browsing internal elements of components/subsystems and their interfaces. These can help the evaluation team understand the functionality of abstract components.

Table 1. Some Relevant Information of *ZLib*, *ZDelta* and *GZip*

Application Name	Version	Number of lines	Number of files	Nature and language	Conceptual SA exists?
ZLib [32]	1.2.3	8.5KLOC	22	Library (C)	No
ZDelta [31]	2.0	7.0KLOC	27	Library(C)	No
GZip [15]	1.2.4	7.3KLOC	16	Application(C)	No

In this way, the architecture recovery process and evaluation method can be combined to enhance the correctness and suitability of an extracted software architecture.

4 Case Study and Lessons Learned

Our study had two aims. First, we wanted to investigate the practicality of using automatically recovered architectures as the basis for analysis of legacy systems for which no architectural documentation is available. Second, we wanted to examine whether architectural analysis plays a helpful role in the architecture recovery process. In particular, we wanted to see whether the use of an architectural analysis method (such as SAAM) can improve the quality of the architecture recovered by a tool (such as SWAG Kit).

In order to study these issues, we used our framework to recover the architectures of three open source compression/decompression systems: *ZLib*, *Zdleta* and *GZip*. *ZLib* is a general purpose lossless data-compression library. *ZDelta* is based on *ZLib*, but has been significantly modified; it provides new interfaces for streaming the target data and extensive runtime parameterizations. *GZip* is a compression utility that uses the same compression algorithm as *ZLib* and *ZDelta*. Information on these systems is listed in table 1.

With the case study we learned several interesting lessons as listed and explained below.

1. SWAG Kit is weak at identifying subsystem structure. On the other hand, architecture analysis is effective in identifying subsystem structure once the recovery process identifies low-level components and connectors.
2. The iterative application of SWAG Kit and SAAM helps to identify and resolve errors in the architecture, and leads to a deeper understanding of the architecture than that obtained with SWAG Kit alone.
3. SWAG Kit emphasizes architectural structure (components + connectors, and eventually subsystems). To get a dynamic view of a system (protocols used by components to collaborate; how specific tasks are carried out; data and control flow), the architecture analysis step is helpful.
4. Our approach can help evaluate which of a set of candidate libraries best suit a project's needs.
5. When applied to moderate applications ($\sim$10KLOC), only modest time investment (about one person week) was required to perform a complete analysis.

6. The iteration between architectural extraction and architecture analysis requires close collaboration between the people performing the tasks.

4.1 Improved Subsystem Structure

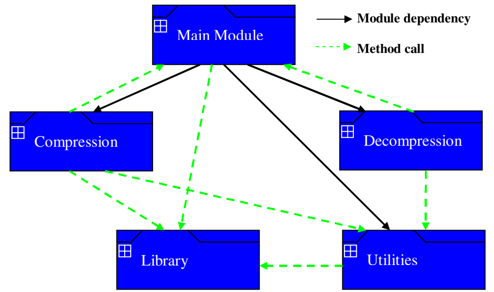
In order to aggregate low-level source code information into a higher level of abstraction, we were required to derive the subsystem structure. SWAG Kit is unable to provide subsystem structures automatically. Therefore, as a part of the recovery process, we manually analyzed the source code, the comments of the source code, and used domain knowledge to obtain an initial subsystem structure. However, we were unsure how well this substructure decomposition matched the system's true conceptual architecture. To verify the subsystem structures and to understand their dynamics, we evaluated them using SAAM.

First, we used SWAG Kit to derive a low-level architecture. Then, we derived the subsystem structures for each of the three applications; these are shown in Figs. 2(a), 2(b) and 2(c). A brief description of each subsystem or module follows:

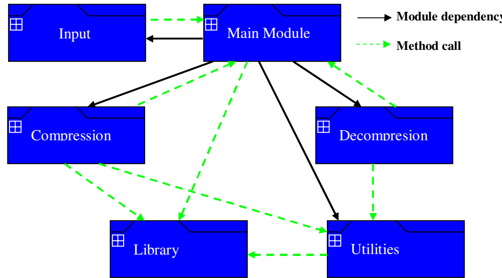
- The **Input Module** is used to specify the compression/decompression algorithm and to specify compression levels. Runtime parametrization of the library is required to dynamically select between multiple compression/decompression algorithms and different files sizes. Separation of the *Input Module* from the *Main Module* localizes the changes that are necessary to adapt to a new compression algorithm.
- The **Main Module** coordinates the rest of the components. It consists of functions required to invoke and terminate the application, manage the session, specify the input file (to be compressed/ decompressed), output the result, and deal with errors.
- The **Compression Module** carries out the actual compression. The separation of the *Input Module* ensures that this module does not depend on any hard-coded compression or input algorithms.
- The **Decompression Module** provides the function of decompressing data streams.
- The **Utilities Module** provides useful functions to the rest of the application, such as memory management and graphical display.

However, this recovery process did not provide conceptual architecture adequate for answering maintenance questions. Therefore, we proceeded through the remaining steps of SAAM. At this stage, we first identified three quality attributes: *modifiability*, *integrability* and *security*. Scenarios illustrate the importance of these important quality attributes in the compression/decompression domain.

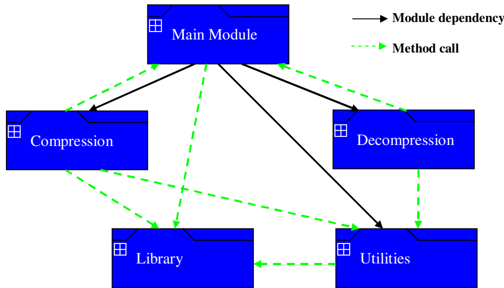
- Modifiability of compression libraries is important. Two examples of changes that might be required are: 1) *add a new compression algorithm to the toolkit* and 2) *modify the toolkit to run under a different operating system.*



(a) *ZLib*



(b) *ZDelta*



(c) *GZip*

Fig. 2. Initial subsystem structures of *ZLib*, *ZDelta* and *GZip*

- Library applications such as *ZLib* and *ZDelta* should be easy to integrate with other applications. Two plausible scenarios are 1) *add compression functionality to a file transfer program* and 2) *provide a graphical user interface for a standalone file compression program*.
- Maintaining confidentiality in electronic documents is vitally important. Example scenarios might be 1) *encrypt a document before saving it on a USB key* and 2) *encrypt a document before emailing it*.

While analyzing the architectures with respect to the scenarios, we found that the subsystem structures obtained from SWAG Kit were insufficient to analyze how easily the scenarios could be enacted.

We successfully used SAAM to improve the subsystem structures. For example, we considered the following *integrability* scenario: *Rather than using the standard Bluetooth Device Discovery model to detect nearby mobile services, developers wish to implement a system that relies on machine-readable visual tags for out of band device and service selection. While implementing the visual tag application, the developers want to use an easily adaptable built-in compression library to store the image of the visual tag in order to save memory space.* The automatically extracted subsystem structures for *ZLib* and *ZDelta* lacked the information necessary to analyze how well their architectures could support this task. We were unable to find a component/subsystem specification that illustrated how to use the library application. Both architectures provide separate *Main* and *Utility* modules. This indicates that if a developer wants to adapt the entire compression/decompression library to another environment, changes can be localized to the *Main Module* only.

However, the separation between these two modules is far from sufficient to determine how easily this scenario could be enacted. We therefore further analyzed the architecture using the *LSEdit* visualization facilities. We browsed the tool output and searched for files and interfaces that might be related to this task. We found that both *ZDelta* and *ZLib* have files illustrating the use of the library applications. So we modified the subsystem structures of *ZLib* and *ZDelta* and displayed the improved structural views using *LSEdit*. These versions of the architectures are shown in Figs. 3(a) and 3(b).

In our experience, the automatically extracted architectures did not support analysis of the system with respect to our scenario. SAAM analysis can help identify and fix the shortcomings of the automatically extracted architecture, and can help improve the subsystem decomposition. However, the use of the recovery tool can significantly help in carrying out the SAAM analysis.

4.2 Better Understanding of the Architecture

In the remaining SAAM evaluation process, we again used the updated architectural views to execute the remaining scenarios. To analyze the architecture for the *security* quality attribute, we used the scenario: "*a developer wishes to incorporate encryption in the compression feature*". To map the scenario onto the architectures, we looked for the components in the architectures that support encryption/decryption. As we had not considered security issues during the recovery process, none of the extracted architectures contained the information that was required to address the security scenario.

To explore the scenario, we investigated the libraries' source code and comments and browsed the tool output to find interfaces that handle data streaming and security issues. Interestingly, we found that *ZDelta* handles data streaming in the target file and has a provision for incorporating data encryption in a modularized manner.

Therefore, we again refined the architectural view of *ZDelta* and introduced a new *Security Module* subsystem in *ZDelta*'s architecture (see Fig. 3(b)) in order to explicitly address the *security* quality attribute.

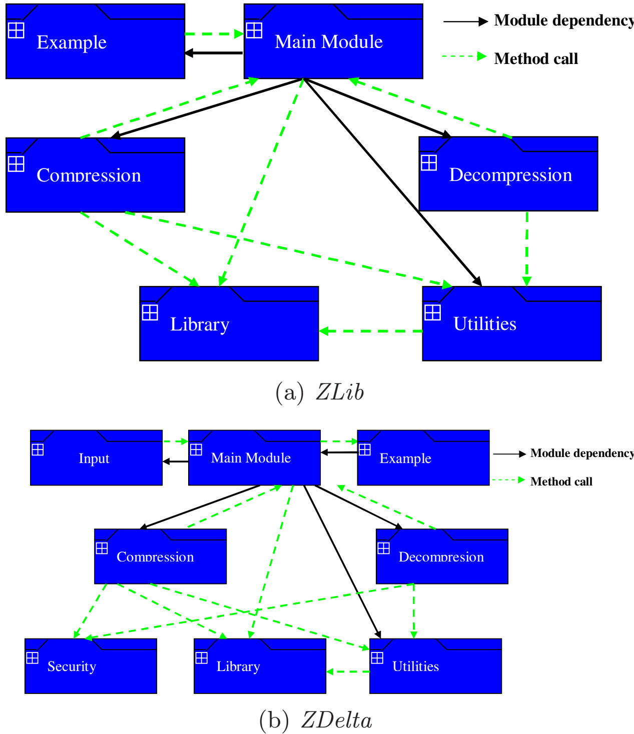


Fig. 3. Refined subsystem structures of *ZLib* and *ZDelta*

The investigation of the security scenario helped us learn more about what kinds of quality attributes a compression application can support and what kind of mechanisms it provides to address these quality attributes. Additionally, the repeated iterations of the *LSEdit* browser with further SAAM analysis guided our understanding in the compression area.

In summary, SAAM analysis using scenarios helped refine the conceptual architecture, while the SWAG *LSEdit* tool helped carry out this analysis on an imperfect view of the architecture. Incremental and iterative analysis helped move to a superior system decomposition.

4.3 Understanding the Dynamics of the Architecture

Architecture recovery using SWAG Kit helped us extract the applications' static architectures in terms of components and connectors. However, these static architectures did not help us understand the architectural dynamics, and did not support analysis of the strengths and weaknesses of the three software architectures. SAAM evaluation helped us in this regard by guiding us in understanding how components collaborate to perform a task. The browsing facility of SWAG Kit helped us find the appropriate component interfaces, which were used to accomplish the tasks quickly and easily.

As discussed in section 4.1, we determined two tasks to assess the *modifiability* quality attribute in this compression/decompression domain: *add a new compression algorithm to the toolkit* and *modify the toolkit to run under a different operating system.* A plausible scenario for the first task is: *A developer wants to add a new lossy compression algorithm for use with media files.*

When we analyzed how well the architectures support this scenario, we found that both *ZLib* and *GZip* use the *Deflating algorithm* along with *Huffman coding*, both directly encoded in the *Compression Module*. So, if the developers were to add a new compression algorithm, they would have to modify both the *Main* and *Compression* modules.

ZDelta provides better support for this task, as the *Input Module* is separated. The developers would only have to substitute the old algorithm with the new one, without making further modifications. The change is localized and does not affect other components.

For the second task, we found that as *ZLib*, *ZDelta* and *GZip* all have separate *Utility Modules*. They can be easily adapted to a new operating system, since the changes are localized to this one module only.

Scenario mapping helped us understand the interaction among components for executing the two tasks. By means of the interactions, we came to know that *ZDelta* has a more cohesive modular structure than that of *ZLib* and *GZip*. Thus, the combined approach helped us in understanding the dynamics of the extracted architectures.

4.4 Provision for Comparing Architectures

When architectural analysis is used during architecture recovery, developers can use the evaluation results to compare different candidate architectures. In our case, as all the applications/libraries are in the domain of compression/decompression, we used the result of the SAAM evaluation to compare them with respect to the identified quality attributes.

Based on the evaluation results of SAAM, we determined that they match up to different levels of conformance to the quality attributes, such as *modifiability*, *integrability* and *security*. The comparison results are summarized in Table 2.

Out of the three applications, *ZDelta* is the best in terms of *modifiability*, since the compression algorithm can be explicitly specified and the application can easily be adapted to a new operating system. The architectures of *ZLib* and *GZip* do not satisfy the modifiability requirement properly, since they do not have an explicitly defined *Input Module*. Both *ZLib* and *GZip* provide support for deploying the application on different platforms.

ZLib and *ZDelta* provide ease of adaptation, addressing the integrability quality attribute.

ZDelta provides better support for *Security* by separating the *Decompression* and *Security* modules. Both *ZLib* and *GZip* fail to handle data encryption in a modularized manner.

These examples show that combining architectural analysis with architectural extraction can help analyze which set of possible systems best meet a project's

Table 2. Comparison between Three Compression/ Decompression Applications

Application/ Quality Attribute	**Modifiability**		**Integrability**	**Security**
	Different Techniques	Different Platforms	Ease of Adaptation	
ZLib	X	√	√	X
ZDelta	√	√	√	√
GZip	X	√	X	X

√=Supports the task or QA; X=Does not support the task or QA

needs. This ability is interesting, for example, for open source projects evaluating choices of what third party code to reuse.

4.5 Reasonable Tractability

Using our iterative approach, it took us about seven days to extract a conceptual architecture for each of the three applications (or a total of three staff weeks.) The applications were of modest size, each consisting of about 10KLOC.

This time benefited from SWAG Kit's detailed initial architecture and powerful browsing facilities. Extraction time also varies depending on the availability of the proper source code documentation, the quality of comments in the source code, and the structure of the source code. Further study is required to see how this time scales to large architectures. However, at least for a system of about 10KLOC, seven days seems a modest effort for the considerable payback in architectural knowledge.

4.6 Team Interactions

Our iterative approach requires considerable interaction between the architecture extractors and evaluators. In our case, the extractors and the evaluators were in the same group. While extracting the architectures using SWAG Kit and SAAM iteratively, we found that the close interaction between extractors and evaluators saved collaboration time and effort. We believe that this might be beneficial for larger systems.

4.7 Feedback

We contacted the authors of the three libraries to ask them how well the resulting architectures represented their system. Two of the three responded. The first respondent reported that the conceptual architecture seemed correct. The second reported that the architecture was incorrect, as it grouped the underlying source files differently from his understanding of the system. The latter result is interesting, as it shows that factoring the architecture around quality-driven scenarios can lead to different decompositions than intended by the original author. It is not clear whether the new decompositions are superior to the author's intuition of what belongs together for analysis and maintenance tasks. Considerable further research is required to address this question.

5 Conclusion

In this paper, we have detailed our experience applying a framework for extracting and evaluating architectures of legacy systems. Our case study applied SWAG Kit for architectural extraction and SAAM for architecture analysis. To our knowledge, no one has reported such a case study combining these approaches in an iterative and incremental manner.

In our case study, we extracted the architectures of three open source compression applications. We found that the combined approach was tractable (at least for modestly-sized applications). The use of SAAM can significantly improve the subsystem structures obtained using SWAG Kit. The combined approach helped us understand the dynamics of a software architecture in a better way than the architecture recovery process alone.

The primary limitation of our approach is that our data is largely subjective. A next step would be to perform a study in the combined approach with third party developers. Nonetheless, this study allowed us to demonstrate clear benefits and weaknesses of the incremental and iterative framework.

Acknowledgement

This work was supported in part by the Natural Science and Engineering Research Council of Canada. We thank Rob Fletcher for his help in an earlier version of this paper.

References

1. Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., Zaremski, A.: Recommended Best Industrial Practice for Software Architecture Evaluation (CMU/SEI-96-TR-025) (1996)
2. Babar, M.A., Zhu, L., Jefery, R.: A Framework for Classifying and Comparing Software Architecture Evaluation Methods. In: Australian Software engineering, pp. 309–318. IEEE CS, Washington (2004)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley, Reading (1998)
4. Bergner, K., Rausch, A., Sihling, M., Ternité, T.: DoSAM - Domain-Specific Software Architecture Comparison Model. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712, pp. 4–20. Springer, Heidelberg (2005)
5. Bosch, J., Molin, P.: Software architecture design: Evaluation and transformation. In: Engineering of Computer Based Systems Symposium, pp. 4–10. IEEE CS, Los Alamitos (1999)
6. Ivan, T.B., Holt, R.C.: Software Architecture Recovery Using Conway's Law. In: Centre for Advanced Studies Conference, pp. 123–133. IBM Press, Toronto (1998)
7. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional, Reading (2002)
8. Dobrica, L., Niemela, E.: A Survey on Software Architecture Analysis Methods. IEEE Transactions on Software Engineering 28, 638–653 (2002)

9. Eixelsberger, W.: Recovery of a Reference Architecture: A case study. In: 3rd International Software Architecture Workshop, pp. 105–108. ACM, New York (1998)
10. Graaf, B., Dijk, H.v.: Evaluating an Embedded Software Reference Architecture. In: 9th European Conference on Software Maintenance and Reengineering, pp. 354–363. IEEE CS, Washington (2005)
11. Garlan, D.: Software Architecture: A Roadmap. In: The Future of Software Engineering, pp. 93–101. ACM, New York (2000)
12. Globus, http://www.globus.org/
13. Gronbaek, K., Wiil, U.K.: Towards a Reference Architecture for Open Hypermedia, http://www.aue.aau.dk/~kock/OHS-HT97/Papers/gronbak.html
14. Guo, G.Y., Atlee, J.M., Kazman, R.: A Software Architecture Reconstruction Method. In: Working IFIP Conference on Software Architecture, pp. 15–34. Kluwer B.V., Deventer (1998)
15. GZip, http://www.gzip.org/
16. Kazman, R., Abowd, G., Webb, M.: SAAM: A Method for Analyzing the Properties of Software Architectures. In: 16th International Conference on Software Engineering, pp. 81–90. IEEE CS, Los Alamitos (1994)
17. Kazman, R., Carriére, S.J.: Playing Detective: Reconstructing Software Architecture from Available Evidence. Automated Software Engineering 6, 107–138 (1999)
18. OODT, http://oodt.jpl.nasa.gov/oodt-site/
19. Lutz, R., Gannod, G.C.: Analysis of a software product line architecture: an experience report. The Journal of Systems and Software 66, 253–267 (2003)
20. Matinlassi, M.: Evaluating the Portability and Maintainability of Software Product Family Architecture: Terminal Software Case Study. In: 4th Working IEEE/IFIP Conference on Software Architecture, pp. 295–298. IEEE CS, Wasington (2004)
21. Medvidovic, N., Jakobac, V.: Using Software Evolution to Focus Architectural Recovery. Automated Software Engineering 13, 225–256 (2006)
22. Mendonca, N.C., Kramer, J.: An Approach for Recovering Distributed System Architectures. Automated Software Engineering Journal 8, 311–354 (2001)
23. Monroe, R.T., Kompanek, A., Melton, R., Garlan, D.: Architectural Styles, Design Patterns, and Objects. IEEE Software 15, 43–52 (1997)
24. Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S.: An empirical study of static call graph extractors. In: 18th International Conference on Software Engineering, pp. 158–191. ACM, New York (1996)
25. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. In: Software Engineering Notes. ACM Sigsoft, vol. 17, pp. 40–52. ACM, New York (1992)
26. Pinzger, M., Gall, H., Girard, J.F., Knodel, J., Riva, C., Pasman, W., Broerse, C., Wijnstra, J.G.: Architecture Recovery for Product Families. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 332–351. Springer, Heidelberg (2004)
27. Rigi, http://www.rigi.csc.uvic.ca/
28. Roy, B., Graham, T.C.N.: Methods for Evaluating Software Architecture: A Survey, p. 82, School of Computing TR 2008-545, Queen's University (2008), http://www.cs.queensu.ca/TechReports/reports2008.html
29. Shrimp, http://www.thechiselgroup.org/shrimp
30. SWAG Kit: Software Architecture Group, http://www.swag.uwaterloo.ca/SWAGKit/
31. ZDelta, http://cis.poly.edu/ZDelta/
32. ZLib, http://www.zlib.net/