An Exploratory Case Study Using CBSP and Archium

Charles L. Chen
The University of Texas at Austin, Empirical Software Engineering Laboratory,
Austin, Texas
U.S.A
clchen@ece.utexas.edu

Danhua Shao
The University of Texas at Austin, Empirical Software Engineering Laboratory,
Austin, Texas
U.S.A
dshao@ece.utexas.edu

Dewayne E. Perry
The University of Texas at Austin, Empirical Software Engineering Laboratory,
Austin, Texas
U.S.A
perry@ece.utexas.edu

ABSTRACT

The need for architectural rationale has long been recognized, but unfortunately it has remained a relatively unexplored area of research in software architecture. However, there is growing interest in methods for capturing the rationale behind software architectures. We summarize two of these methods (CBSP and Archium), present an exploratory case study using both methods to evolve a real software system, and then use the results from this case study to analyze and compare these methods. From this analysis, we conclude that the CBSP and Archium methods are complementary rather than competing because of their respective strengths and weaknesses.

Categories and Subject Descriptors

D.2 Software Engineering; D.2.1 Requirements; D.2.10 Design; D.2.11 Software Architecture

General Terms

Software Architecture, Exploratory Case Study, Experience Report.

Keywords

Architecture evolution, System maintenance, Architecture rationale, case study, accessibility framework, design trade-offs

1. INTRODUCTION

Going from requirements to architecture is the first and hardest step in engineering software systems. The choices made during this step shape the project and may restrict the choices to be made in evolutionary development. Unfortunately, this wealth of choices and complex tradeoffs is often captured inadequately, if at all. Having a better understanding of the rationale for why these choices were made can bring significant benefits for both the initial design phase of the system as well the evolution and maintenance phase.

The importance of rationale in software architecture has been long recognized. Perry and Wolf proposed their model of software architecture composed of elements, form, and rationale [4]. However, as seen in Garlan and Perry [8], most of the research in software architecture has focused on ADLs, and there has been little work done on using or providing rationale.

More recently, there has been some interest in capturing rationale. Duenas and Capilla advocate using a Design Decision view of software architecture [5]. Wolf and Dutoit focus on rationale for their Rationale-based Analysis Tool for object-oriented requirements analysis [6]. Perry and Grisham [9] explore the issues with using rationale, specifically in the context of COTS. CBSP and Archium are methods designed to document the rationale as part of the architecture [1,2,3].

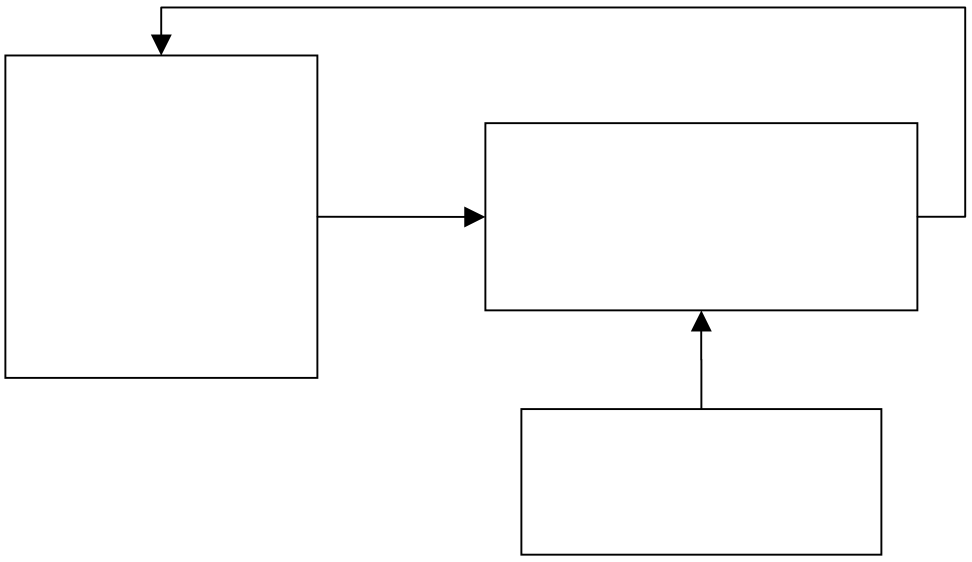
Exploring the capture and use of rationale in software architecture is one of the research areas of our lab. Recently, we had an opportunity to explore these issues firsthand. One of the authors created the Core Library Components for Text-To-Speech (CLC-4-TTS) Suite, a set of extensions for Firefox that enables the visually impaired to browse web pages [7]. Because of the need to stay up to date with Firefox and because of new features, the CLC-4-TTS Suite is constantly evolving. In a recent iteration, the ability to process CSS speech properties was added; this is a feature that is not available in even the leading commercial screen readers. In the process of evolving this system, we applied the CBSP and Archium methods in designing a new component vital for having CSS speech property support. The CBSP and Archium methods were selected because they are the more recent attempts at capturing and using rationale in architecture. This paper summarizes the CBSP and Archium methods for capturing architecture rationale and presents an exploratory case study that uses and evaluates them in the context of this evolutionary step.

2. THE CBSP METHOD

The CBSP method was developed by Grunbacher et al. [1,2]. CBSP is designed to help architects bridge the gap between requirements and architecture. The name CBSP comes from the idea of breaking down an architecture into components, buses, systems, and their respective properties. The steps of the CBSP method are as follows:

1. Selection of requirements for the next iteration
2. Architectural classification of requirements
3. Identification and resolution of classification mismatches
4. Architectural refinement of requirements
5. Trade-off choices of architectural elements and styles with CBSP

The first step of this method simplifies the problem by eliminating and refining requirements using stakeholder prioritization. During the second step, a team of architects classifies the requirements in terms of their relevance along the Component, Bus, System, Component Property, Bus Property, and System Property dimensions. In the third step, the architects



compare their classifications and discuss any mismatches. By having this discussion, misunderstandings about the requirements can be uncovered and resolved. After the architects have agreed on the classification of the requirements, they refine these requirements into architecturally friendly CBSP artifacts in the fourth step. An example of this refinement process can be seen below in section 4.1 where we apply the CBSP approach to developing a component for a real software system.

In the last step, the architects use these CBSP artifacts to choose an appropriate architectural style for the system by comparing the amount of support each style has for the relevant properties. Grunbacher et al. present an example of doing this evaluation in [1]. For the Data Component dimension, they list possible properties as aggregated, persistent, streamed, and cached. They also show the amount of support that is provided for each property by the Client-Server, C2, Event-Based, Layered, and Pipe-and-Filter architectural styles. For instance, pipe-and-Filter is good at supporting streamed data, but bad at supporting cached data; the reverse is true of the Client-Server style. They repeat this for the properties related to the Processing Component, Bus/Connector, and System dimensions. However, it is important to note that the evaluation that they did reflected their specific case study and is not necessarily appropriate for other projects. Ultimately, choosing the various properties, delineating the various styles, and evaluating the amount of support that a particular style has for a given property is a task that is left up to the architects for their particular domains and specific projects.

3. THE ARCHIUM METHOD

Jansen and Bosch proposed a new approach called Archium [3] for documenting the rationale as part of the design and evolution of software architecture. In the traditional view of software architecture, the architecture is composed of components and connectors. The problem with this traditional model is that the design rationale is lost, making it difficult to evolve the system. To overcome this problem, Jansen and Bosch proposed viewing software architecture as a series of design decisions; this information about design decisions can help developers evolve their current architecture. In the Archium view, a design decision includes the following parts:

Rationale: Why the change is being made

Design rules: What rules should be followed

Design constraints: What should not be allowed

Additional requirements: New requirements resulting from the change

The information that Archium provides about design decisions can be helpful in evolving the system, checking for violations of design rules and constraints, pruning obsolete design decisions, preserving the integrity of the concepts, defining the design space clearly, analyzing both the software architecture and design, and tracing changes in the architecture. Archium supports first class architectural design decisions, explicit architectural changes, documentation of modifications/subtractions/additions, and clear relationships between architectures and their implementations.

The problem to be solved is the core of this model [3]. Motivation and Cause elements describe where the problem comes from. Solution elements specify possible approaches to solving the problem. Decision elements show which solution is selected as the final solution for the problem as the result of making trade-offs. Applying this decision leads to a modification on the software architecture. All of this happens within the context of requirements and other architectural decisions.

Each solution contains the following elements: Description, Design Rules, Design Constraints, Consequences, Pros, and Cons. The Description delineates the modifications that will result from this solution. The Design Rules are the specifications to which the implementation must conform as found, for example, in architectural styles and/or patterns. The Design Constraints define the limitations on what the architectural entities are allowed to do. The Consequences capture the consequences of using this solution. The Pros are the expected benefits of using this solution. The Cons are the possible problems of using this solution.

With this design decision model, Jansen and Bosch propose a new meta-model for software architecture. In this meta-model, the design decision is specified as a first class entity, and the software architecture is specified as a series of design decisions.

The meta-model consists of the Architecture Model, the Design Decision Model, and the Composition Model. The Architecture Model defines the elements in the same way as in common software architectures. It includes the following: Component Entity, Delta, Interface, Port, Connector, and Abstract Connector. The Design Decision Model defines the design decision and includes Design Fragments and Design Decisions. The Composition Model applies the Design Decision Model to the Architecture Model. It contains the Composition Technique, the Composition Configuration, and the Design Fragment Composition. The Archium meta-model can be seen in Figure 1.

4. AN EXPLORATORY CASE STUDY USING CLC-4-TTS

The CLC-4-TTS Suite is composed of two libraries of functions designed for use by Firefox extension developers and an application layer designed to provide an interface for end users. The first of these two libraries provides text-to-speech functions, and the second provides DOM traversal and manipulation functions. The application layer uses both of these libraries to transform Firefox into a self-voicing browser which enables the visually impaired to access web pages. The architecture of the CLC-4-TTS Suite is constrained by Firefox and what its extension system allows; these constraints are felt acutely at the lower levels of the architecture. The CLC-4-TTS Suite is cross OS compatible and supports Mac and Linux in addition to Windows. In order to provide text-to-speech services for non-Windows platforms, it relies on Java FreeTTS.

The CLC-4-TTS Suite was initially developed at the start of 2005; since then it has been constantly evolving and has undergone several major iterations. The latest iteration involved adding support for CSS speech properties that allow web developers to specify the style in which text is read by screen readers the same way they would specify the style in which text is displayed on the screen. Instead of adjusting the font family, color, size, etc., web developers can set the pitch, speaking rate, volume, etc. Adding support for adjustable speech properties in Java FreeTTS was a difficult problem, and the first attempt to do so was plagued with faults resulting from race conditions. Speech properties were incorrectly associated with messages, and sometimes a speech property would be applied to all messages with no way for the user to revert to the default speech property. This first attempt was done with the idea of adding a custom queue system but without any formal capture of the design rationale.

This exploration was motivated by a desire to evaluate the CBSP and Archium methods as well a practical attempt to step back and re-evaluate the requirements in order to design a good solution for adding adjustable speech properties to Java FreeTTS. The CBSP method was used first to come up with a design for accomplishing this. Then the Archium method was used to consider alternative designs. These designs were compared, and the experiences of using these methods were recorded and analyzed. There are validity issues with possible bias from previous experience with the system and the learning effect of using Archium after having used CBSP. However, this is an exploratory case study and provides a much needed replication of the case studies presented in the CBSP and Archium papers since there have been no attempts to replicate those case studies in the literature.

4.1 Using CBSP

4.1.1 High Level Design

The problem was broken up into the following six requirements:

R1: Speech properties must be configurable for FreeTTS.

R2: Users must be able to interact with the system at all times.

R3: Speech should not have any long pauses.

R4: Equal priority messages should be spoken in the order they were received.

R5: High priority messages must be able preempt lower priority messages; preempted messages do not have to be saved.

R6: Speech properties from one message may not interfere with those from another.

These requirements were then refined into the following CBSP elements as seen in Figure 2:

R1 – R1_C: There needs to be a SetProperties function for FreeTTS.

R2 – R2_SP: The system needs to be multi-threaded in order to handle constant user interaction.

R3 – Eliminated: The definition of “long pause” was ambiguous and a correctly functioning queue system should not have this problem.

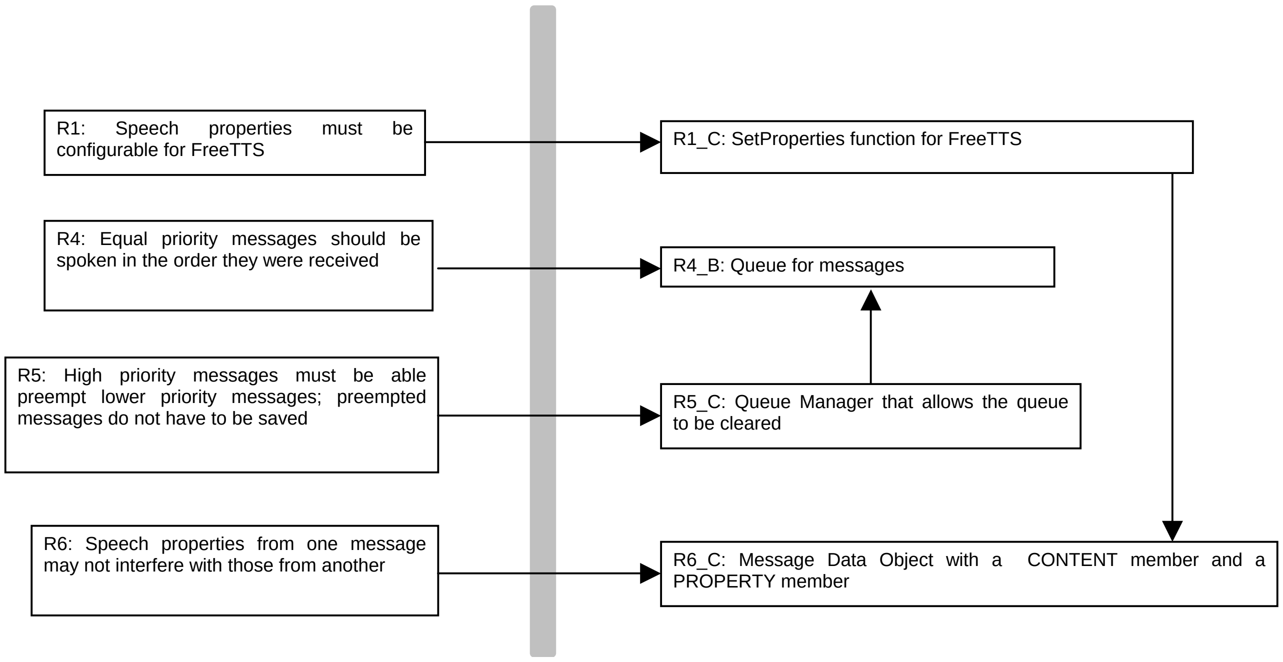
R4 – R4_B: There needs to be a queue for messages.

R4_BP: The queue for this system should be first in, first out (as opposed to a double-ended queue).

R5 – R5_C: There needs to be a Queue Manager that allows operations on the queue. One of these operations has to clear the queue.

R6 – R6_C: There needs to be a Message Data Object that has a CONTENT member and a PROPERTY member.

Deriving these CBSP elements was a fairly straightforward process. CBSP makes it easy to trace from these elements back to their requirements. However, where there was room for variation, the CBSP method did not allow for the capture of these alternative solutions. For example, this design reflects the use of a queue system created as a component within the CLC-4-TTS system, but an alternative design of leveraging queuing capabilities that already exist within FreeTTS is not captured and has no place to be documented within the CBSP method.


R1: Speech properties must be configurable for FreeTTS
R4: Equal priority messages should be spoken in the order they were received
R5: High priority messages must be able preempt lower priority messages; preempted messages do not have to be saved
R6: Speech properties from one message may not interfere with those from another
R1_C: SetProperties function for FreeTTS
R4_B: Queue for messages
R5_C: Queue Manager that allows the queue to be cleared
R6_C: Message Data Object with a CONTENT member and a PROPERTY member

Since the goal was to add support for speech properties in Java FreeTTS under the existing CLC-4-TTS Suite and not to create a brand new architecture, the final step of selecting an architectural style was omitted. Had we needed to perform this step, we would have had to analyze the amount of support that various architectural styles had for the properties that we are interested in and which were not necessarily covered in the CBSP to style mappings as laid out by Gruenbacher et al. in [2]. For instance, the data component should be queued, but there is not a queued/buffered property in their style mappings. The processing component needs to be dynamically reconfigurable, but dynamically reconfigurable is only a property for the system in their style mappings.

4.1.2 Low Level Design

There are two implementation options for creating the queue system. The queue system could be fully implemented as an explicit data object, or simply continue to use the existing system (with some minor modifications) and accept minor buggy behavior since FreeTTS will queue messages, but not properties. Because properties could not be queued, the last property would be applied to all messages. Thus it is possible for all the messages in the queue to be spoken with an incorrect property until the last message. The worst possible scenario would be if new messages were constantly being added to the queue and the properties were always different; in such a case, the speech property would always be different from the one intended for the current message. The first approach is far more correct as it is a true queue system; the second approach is much easier to implement. Thus the first approach should be chosen if the bugginess of the second approach is unacceptable; otherwise, the second approach should be used.

The rationale behind the requirements was re-examined; tracing back from the CBSP elements, the most relevant requirement for the queue system was requirement R6, “Speech properties from one message may not interfere with those from another.” In the CLC-4-TTS Suite, the only time there could be such interference is when the user is trying to read an HTML element. Reading an HTML element generates three equal priority messages: 1. the type of the element if it is special (for example, a heading, link, check box, etc.), 2. the content of the element, and 3. the status of the element if applicable (whether the check box is checked or not, etc.). The desired behavior in such a case is to read the content with the speech properties specified on the page, and to read the type and the status with the default speech properties. This can only be accomplished with the first method; if the second method were to be used, then the property of the last element wins.

A careful risk assessment was performed by analyzing the possible scenarios. The most common usage of speech properties is in the middle of a paragraph (usually to add emphasis to a particular word); thus there would be no special type to identify and no status. Furthermore, no new messages would be put on the queue until after the queue was cleared, either because everything in the queue had already been spoken or because the user interrupted the queue by performing an action that generated a high priority message that removed all of its predecessors from the queue. This means that at worst, there would never be more than three messages in a queue; after the third message, the queue would be reset. Because of these behavioral properties, we can enumerate the scenarios that involve speech properties as follows:

1. There is only the content. – There is only one message in the queue, and the last property used is its property. There is no error. This is also the most frequent case where speech properties are used.

2. There is type and content. – There are two messages in the queue. The property that will be used for both messages belongs to the last message. The type will be read with the property for the content; this may make the type sound strange, but it will still be understandable.

3. There is content and status. – There are two messages in the queue. The property that will be used for both messages belongs to the last message. The content will be read with the property for the status; this means the content will be read with the default property, making it sound as if speech properties were not implemented.

4. There is type, content, and status. – There are three messages in the queue. The property that will be used for all the messages belongs to the last message. The type and the content will be read with the property for the status. However, since the type and the status are both system generated messages and both will be using the default property, the type will be read with the correct property. The only error in this case is reading the content with the default property, making it sound as if speech properties were not implemented.

Since the effects of the errors are minor and transient, and the errors would only occur rarely, the second implementation option is acceptable.

4.2 Using Archium

4.2.1 High Level Design

The problem was stated in Archium format as follows:

Problem

The current interface to FreeTTS does not allow speech properties to be set.

Motivation

The result of this is that Linux and Mac users are unable to experience the CSS speech property support being introduced

Cause

Speech property support has not been implemented yet

Context

Evolving the existing CLC-4-TTS Suite

Two potential solutions were explored: 1.) Using a JSML Generator, and 2.) Using a Queue System. The pros and cons analysis is shown in the following subsections.

4.2.1.1 Potential Solution #1: JSML Generator

•Description:

Use JSML to encode the properties into a string along with the message. Pass the entire thing into FreeTTS.

•Design rules:

All generated strings must be well formed JSML strings.

•Design constraints:

Message needs to be put within tags that contain the properties; therefore messages and associated properties should be delivered at the same time.

•Consequences:

CLC-4-TTS Suite is dependent on FreeTTS supporting JSML.

•Pros:

1. Easy to code (similar system exists for SAPI 5 already)
2. FreeTTS manages the queue
3. Easy to force FreeTTS to empty queue (for prioritization)

•Cons:

1. FreeTTS does not yet support JSML; significant wait time expected as the FreeTTS project appears to be in hiatus (last update was in February 2005).

4.2.1.2 Potential Solution #2: Queue System

•Description:

Create a queue system that will set the speech properties for FreeTTS, pass FreeTTS a message to be spoken, and then wait until it is ready for a new message with a different set of speech properties.

•Design rules:

Must keep track of messages and associated speech properties

•Design constraints:

Queue must not interfere with users' ability to interact with the system as a whole; blocking is only to block the speech portion but nothing else.

•Consequences:

CLC-4-TTS Suite is dependent on Java FreeTTS allowing the setting of speech properties.

•Pros:

1. Can be implemented immediately as Java FreeTTS already allows for the setting of speech properties.

•Cons:

1. Far more difficult than using a JSML generator.

Because the JSML Generator approach was infeasible, the Queue System approach was selected as the design.

4.2.2 Low Level Design

The same two implementation options exist for the Archium version. Making the choice between these two options can be considered as simply another Archium model.

Problem

The current implementation of the CLC-4-TTS system does not have an explicit queue.

Motivation

The result of this is that equal priority messages cannot be queued and spoken in order such that their speech properties do not interfere with each other.

Cause

An explicit queue has not been implemented yet

Context

Evolving the existing CLC-4-TTS Suite

4.2.2.1 *Potential Solution #1: Explicit Queue System*

•Description:

Create a queue object. Use this to explicitly queue the messages.

•Design rules:

All messages to FreeTTS must be sent through this new queue system to ensure that all messages are queued.

•Design constraints:

Queue system must accommodate messages and their speech properties. Queue must interface with FreeTTS.

•Consequences:

CLC-4-TTS Suite must be heavily modified to account for this new queue system.

•Pros:

1. Messages and their properties will be handled correctly all the time

•Cons:

1. A great deal of effort is required to overhaul the existing system to use this new queue system
2. Risky approach – potential to introduce a large number of bugs
3. Restricts future change – switching to a different system (such as a JSML generator) in the future would be difficult as the queue system will need to be undone first.

4.2.2.2 *Potential Solution #2: Implicit Queue System*

•Description:

Rely on the existing system (after minor modifications) to handle queuing.

•Design rules:

Messages should continue to be sent the way they are. Speech properties for messages may be sent with the messages if there are any.

•Design constraints:

Only minor changes will be made to the system; for the most part, things should stay the same.

•Consequences:

CLC-4-TTS Suite is slightly modified to accommodate this system. Some bugginess will have to be accepted.

•Pros:

1. Can be implemented quickly without disrupting the current system
2. Easy to switch to a different system in the future

•Cons:

1. Not all cases will be handled correctly

Because of the rare occurrence of the conditions where there will be a problem and because having the flexibility to use a different, better system in the future is important for a constantly evolving project like the CLC-4-TTS Suite, the second solution is chosen.

4.3 Comparison of CBSP and Archium

Although the methods were different, they both resulted in the same design. The likely explanation for this result is that this design is the most direct solution to the problem. There is also the possibility that this result is an artifact of both methods having been used by the same person; however, he tried to use these methods as independently as possible by following the steps for one method, then restarting from scratch and following the steps for the other method.

The high level design was implemented according to the low-level design decisions. After modifying the code of the text-to-speech component to take advantage of the existing queue system within FreeTTS, the system performed correctly (except for the known buggy cases), thus validating the design. Interestingly enough, capturing the possibility of using a JSML generator and leaving the flexibility to use it in the future may soon have some practical value since there has been some recent news that the FreeTTS project will incorporate code that will allow it to support JSML in the near future. Using a JSML generator would be a better solution than the current queue system; capturing the rationale for why it was not selected will be helpful during refactoring in deciding whether or not these reasons still exist given the current situation.

We found that the CBSP method was useful in providing a structured process for going from requirements to architecturally friendly CBSP artifacts and in tracing these artifacts back to the original requirements. However, it was difficult to evaluate trade-off choices made in the creation of the CBSP artifacts; CBSP provides no support for capturing and reasoning about alternatives in deriving the CBSP artifacts from the requirements. Although there was no attempt to choose an architectural style using the CBSP artifacts given the nature of the project, had it been necessary, we would probably have found that step to be difficult since there is no guidance given in the CBSP methodology for determining the amount of support that an architectural style provides for a particular property.

Interestingly, we found that the strengths and weaknesses of the Archium method were exactly the opposite of those of the CBSP method. Not only are alternative designs captured explicitly as part of the Archium process, the pros and cons of those designs are documented as well. This provides a strong basis for reviewing design choices and determining if past choices are still valid in the current context of the system. However, creating these potential solutions was difficult; Archium has no process or guidance for deriving the architectural solutions from the requirements and leaves that step completely up to the designers.

The most interesting result of this exploratory case study is that the CBSP and Archium methods appear to be complementary rather than competing. Thus using CBSP to go from requirements to possible architectural solutions and then using Archium to document and evaluate the alternatives may be better, both in terms of the quality of the resulting architecture and the documentation of the underlying rationale, than either method alone. Both of these methods can improve by learning from each other. CBSP could benefit from having better tradeoff analysis and capturing alternative solutions. Archium could benefit by having a clearer path between requirements and architectural design. Choosing an appropriate architectural style is an area where further research is needed as neither method adequately addresses this issue.

5. CONCLUSIONS AND FUTURE WORK

We have summarized the CBSP and Archium methods. Our exploratory case study applied both methods to designing a solution for a real-life software system; the results of this study indicate that the CBSP and Archium methods may be complementary rather than competing. CBSP is useful in helping architects go from requirements to architecturally friendly elements; Archium is useful for capturing the rationale for picking one solution over other alternatives. Using both methods can result in an easier transition from requirements in the problem space to architectural elements in the solution space and in better documentation of why certain choices were made – this increased traceability is invaluable in software evolution. The selection of an appropriate architectural style is an area that requires further research since neither of these methods currently addresses this problem.

6. ACKNOWLEDGMENTS

This research is supported in part by NSF CISE Grant CCR-0306613.

7. REFERENCES

[1] Grunbacher, P., Egyed, A., Medvidovic, N. *Reconciling Software Requirements and Architectures: The CBSP Approach*. Proceedings of the 5th IEEE International Symposium on Requirements Engineering, 2001, 202-211.

[2] Grunbacher, P., Egyed, A., Medvidovic, N. *Reconciling Software requirements and Architectures with Intent Modeling.* Software and Systems Modeling, Vol. 3 No. 3, 2004, 235-253.

[3] Bosch, J. *Software Architecture: The Next Step*. Proceedings of the First European Workshop on Software Architecture (EWSA 2004), 2004, 194-199.

[4] Perry, D., Wolf, A. *Software Architecture*. 1989. http://www.ece.utexas.edu/~perry/work/papers/swa89.pdf.

[5] Duenas, J., Capilla, R. *The Decision View of Software Architecture*. EWSA, 2005, 222-230.

[6] Wolf, T., Dutoit, A. *A Rationale-based Analysis Tool.* 13th International Conference on Intelligent & Adaptive Systems and Software Engineering, 2004.

[7] Chen, C. *CLC-4-TTS and Fire Vox: Enabling the Visually Impaired to Surf the Internet*. Undergraduate Research Journal, Vol. 5 No. 1, 2006, 32-42.

[8] David Garlan and Dewayne E. Perry, Special Issue on Software Architecture, IEEE Transactions on Software Engineering, 21:4 (April 1995)

[9] Dewayne E. Perry, and Paul Grisham. *Architecture and Design Intent in Component and COTS Based Systems*, International Conference on COTS Based Software Systems, Februrary 2006, Orlando FL.