Eliciting Architectural Decisions from Requirements using a Rule-based Framework

WenQian Liu Steve Easterbrook
Department of Computer Science
University of Toronto
Toronto, ON, M5S 3G4, Canada
{*wl,sme*}*@cs.utoronto.ca*

Abstract

Making architectural decisions based on requirements, analyzing cost-benefit trade-offs, and keeping design options open is a difficult task. Existing work on classification of architectural styles and features of reusable components, and derivation of relevant architectural styles provides useful heuristics to the task, but it remains to be largely a labor-intensive activity.

In this paper, we propose a rule-based framework with automated reasoning for eliciting architectural decisions from requirements. Our goal is to gain a deeper understanding of the relationships between requirements and architectural decisions, define generic mappings based on these relationships, and use these mappings to guide architectural design with a higher degree of automation.

Keywords

Architecture, requirements, mapping, decision elicitation, design guidance, rule-base

1 Introduction

It has been recognized by the research community that building a systematic bridge between requirements and software architecture plays an important role in software engineering [1]. In particular, making architectural decisions based on requirements, analyzing cost-benefit trade-offs, and keeping design options open remains to be a labor-intensive task. A number of approaches have made progress towards providing assistance to software architects.

Early works include Shaw and Clements' classification of architectural styles that had appeared in the published literature [8]. Each style is categorized according to its characteristics with respect to constituent parts (components and connectors), control issues, data issues, control and data interaction issues, and reasoning. Moreover, intuition and rules of thumb on choosing styles to fit the problem are discussed as a preliminary step to design guidance.

Around the same time, Kazman, Clements, and Bass provided a classification on architectural elements in terms of features, which can be used to identify reusable elements that match required feature criteria [5]. In their approach, temporal and static features are defined for classifying architectural elements and describing the matching criteria of requirements.

More recently, Egyed et al. addressed this problem using the CBSP (Component-Bus-System, and Properties) approach [4]. In this work, the WinWin negotiation model [2] is adapted to classify the requirements according to the CBSP properties in the architectural context. Based on these properties, a CBSP model can be built to derive and validate architectural styles.

There are five characteristics in common in these approaches:

1. classification of requirements and architectural properties
2. definition of a partial mapping from requirements properties to architectural elements or decisions using a common language
3. provision of design alternatives and trade-off analysis
4. abstraction of information
5. reuse through styles by condition matching

Despite these advances, a number of key issues in bridging the gap between requirements and software architecture are not well addressed to date.

Unified description language In order to bridge the gap between requirements and architecture, we need to define mappings between them. To establish these mappings, requirement specifications and architectural descriptions must be formulated in a common language. This motivates the development of a unified language. We have seen that this is done implicitly in the above approaches. But we do not yet know the following.

- How feasible is it to use a unified language?
- How to express requirements and architectural descriptions effectively using a unified language?
- What are the key properties of such a language?

Relationship between requirements and architectural decisions

- What kind of architectural decisions are frequently made in building large systems?
- Clearly, the architectural decisions made are related to the benefits and risks that are induced. Are we able to define relationships between them in assisting the trade-off analysis?
- How do architectural decisions relate to the system's requirements?
- Are we able to classify relations between requirements and architectural decisions that are generic and reusable?
- How do we abstract key architectural decisions made in existing systems?

Studying decision making processes in existing systems may provide insight into general relationship between requirements and architectural decisions.

Architectural decisions deferral and trade-off In practice, it is often necessary to defer an architectural decision until further information is acquired and to keep design options open. Therefore, it is undesirable to make every decision up front and have little flexibility in making changes. However, having too many open ends will make decision making difficult and prevent the development progress. This leads to the questions below. Answers to these questions can help analyzing the trade-offs between different architectural decisions, and project architectural evolutions with changing requirements.

- At what stage must these decisions be made before proceeding further? How much can they be deferred?
- To what extent does architectural evaluation help in choosing the best solutions for deferred decisions?
- To what extent do architectural decisions precede and shape identification of requirements?
- Are there any common factors for deferring decisions? Do they relate to specific classes of requirements?

The earlier work has provided insights to the questions posed above, but answers to many of them remain unknown. In particular, answers to the question of what are the generic and reusable mappings between requirements, architectural properties, and decisions can lead to significant progress. Our research is mostly motivated by these questions. In answering these questions, we could gain a deeper understanding of the relationships between requirements and architectural properties, define generic mappings based on the relationships, and use the mappings to guide the architectural design with a higher degree of automation.

In this paper, we propose a framework that can be used to elicit architectural decisions from requirements, and describe a potential rule-based implementation with automated reasoning capability. Although user interaction is required in this framework, we believe it is a worthwhile experiment. Our reasons are the following. Firstly, the framework can be customized for any application domain, and the rule-base can be easily updated as new mappings are required. Secondly, existing architectural decision making knowledge can be evaluated using this framework. Thirdly, the evaluation of knowledge can help us define the relationships between requirements and architectural properties. Lastly, this framework can be extended with higher degree of automation once the reasoning system covers enough decision making strategies.

We plan to build a rule-based tool to capture the mappings, and in the process of doing so, to study how decisions are made, what is the essential knowledge required, and the structure of the knowledge. Our prior experience [6, 7] shows that attempting to develop a rule-based (or production) system raises useful questions about what knowledge and heuristics to apply and how they interrelate.

In section 2, we describe our proposal of a general design guidance framework for eliciting architectural decisions. In section 3, we outline a rule-based implementation of the framework. In section 4, we present concluding remarks.

2 Architectural Decision Elicitation Framework

Requirements need to be obtained from stakeholders. Likewise, architectural decisions need to be elicited from requirements. Even though a large body of research results and practical heuristics is available for making architectural

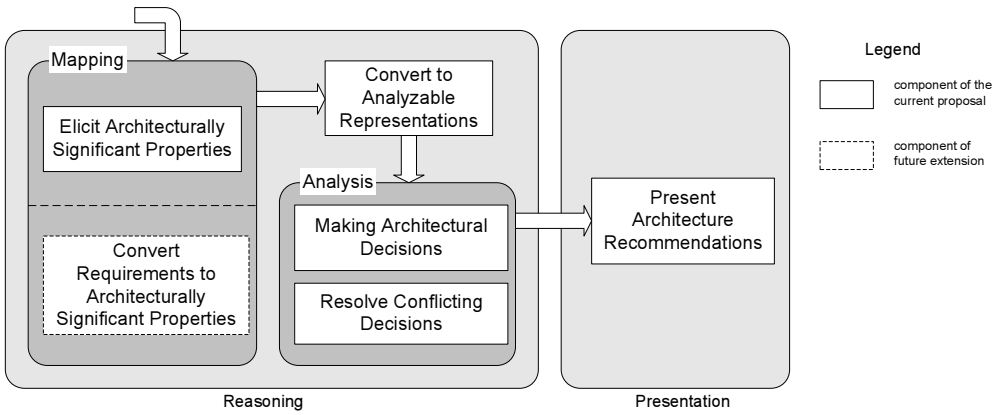

Figure 1. The Architectural Decision Elicitation Framework

decisions, architects still need to carefully go through their knowledge-base (usually their experience) to identify relevant information, and analyze cost-benefit trade-offs, before making a decision. We propose an architectural decision elicitation framework (ADEF) that encapsulates the knowledge required of making architectural decisions, and provides automated mapping from architecturally significant properties to architectural decisions. This framework adapts a general Waterfall model.

There are two main modules, *Reasoning* and *Presentation* in ADEF, as shown in figure 1.

The **Reasoning** module encapsulates the decision making knowledge, and reasons about the requirements to elicit relevant architectural decisions. This module consists three parts: *mapping*, *conversion*, and *analysis*.

The ***Mapping*** submodule uses built-in decision trees (directed acyclic graphs) to provide guidance to the user in manually mapping each requirement specification to one or more architecturally significant properties. Figure 2 illustrates an example of a partial decision tree with only the properties that are significant in choosing architectural styles (the ideas in this example are adapted from [4, 8]). We use *decision node* to refer to both interior node and leaf node in the decision tree, and *property node* to refer to leaf node only.

Here is how the mapping is achieved. For each requirement specified, starting at the root of the decision tree, present the user with the choices represented by the decision nodes associated (i.e. immediately below and connected) to the root, and ask the user to decide whether each choice is relevant to the current requirement. For each relevant decision node chosen by the user, its associated decision nodes are then presented to the user in a depth-first fashion until no more nodes are available as a choice. The descriptions of the *property nodes* chosen by the user are the architecturally significant properties. Such a property and its decision making history, including the nodes along the branch and the source requirement, are described as a *decision unit* and sent to the conversion submodule.

A future extension to this module is the automated mapping from requirements to architecturally significant properties shown as the dotted box in figure 1. This extension dictates that the requirements be stated in a formal language.

The ***Convert to Analyzable Representation*** (conversion) submodule converts the *decision units* created in the mapping module to a form that can be interpreted by the analysis module. The converted decision units are then sent to the analysis module as new facts.

The ***Analysis*** submodule provides ongoing automated reasoning of the following types using a predefined knowledge-base:

1. *making architectural decisions*: based on change in the fact base (either a newly converted decision unit, or a newly made decision), make appropriate architectural decisions using heuristics defined in the knowledge-base, then store the decisions as new facts

2. *resolve conflicting decisions*: provide resolution and explanation when multiple conflicting decisions are made for the same part of the system

The **Presentation** module presents the resulting architectural decisions to the user and updates changes made to previous results. The process then is repeated for another requirement specification.

Next, we describe a rule-based implementation proposal for this framework.

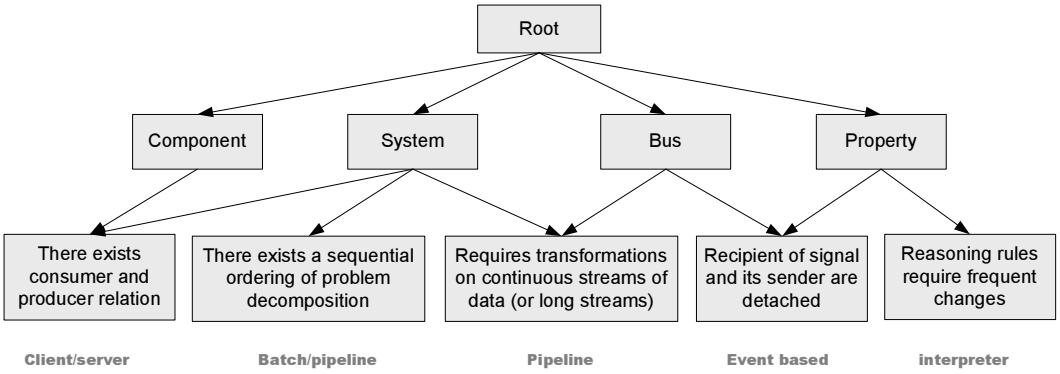

Figure 2. An Example Decision Tree

3 A Rule-based Implementation

In section 2, we have seen that the knowledge-base for the *mapping submodule* is implemented in a decision tree. In this section, we describe a rule-based approach to achieve the automated reasoning capability provided by the *analysis submodule* using a production system.

A production system keeps the fact knowledge-base (or fact base) separate from the rule base. Rules can be defined a priori and maintained independently in the rule base. The fact base is updated during the system execution. Not only can new facts be added, but also the results made by each reasoning cycle are fed back to the fact base as updates. Ongoing reasoning is performed whenever updates to the fact base are received. These characteristics of production system help to achieve dynamic analysis required of the framework. In addition, our prior experience has shown positive results in applying the rule-based approach for automated reasoning [6, 7]. Thus, we believe it is viable to use a rule-based implementation for the *analysis submodule*.

We first give a brief overview of production systems, then discuss the rule-based implementation for the *analysis submodule*.

3.1 Production System Overview

A *production system* is a reasoning system that uses forward-chaining derivation techniques. It uses rules, called *production rules* or *productions* in short, to represent its general knowledge, and keeps an active memory, known as the *working memory* (WM), of facts (or assertions) which are called *working memory elements* (WMEs) [3].

A *production rule* is usually written as:

IF *conditions* **THEN** *actions*

The *conditions*, also known as *patterns*, are partial descriptions of working memory elements, which will be tested against the current state of the working memory. For example, the following rule debits a bank account.

```
IF    (transaction
         (type debit)
         (amt ?x)
         (account ?a))
      (account
         (id ?a)
         (balance ?y∧{≥?x}))
THEN  REMOVE 1
      MODIFY 2 (balance [?y-?x])
```

where *?a*, *?x* and *?y* are variables; {≥*?x*} is a test for balance≥x; `REMOVE 1` deletes the first (i.e. `transaction`) WME from the working memory; and `MODIFY 2` selects the second WME and assigns the value of `y-x` to `balance`.

Each condition can be either positive or negative. A negative condition is of the form *-cond*, where *cond* represents a positive condition. A rule is applicable if all of the variables can be evaluated using the WMEs in the current WM such that the conditions are met. A positive condition is satisfied if there is a matching WME in the WM; a negative condition is satisfied if there is no matching WME in the WM.

A *working memory element* has the following form,

(type (attribute$_1$ value$_1$) ... (attribute$_n$ value$_n$))

where type and attribute$_i$ are atoms, i.e. a string, a word, or a numeral; and value$_i$ is an atom or a list.

The basic operation of a production system is a cyclic application of three steps until no more rules can be applied:

1. *recognize*: identify applicable rules whose conditions are satisfied by the WM;
2. *resolve conflict*: among all applicable rules (or *conflict set*), choose one to execute;
3. *act*: apply the action given in the consequent of the executed rule.

3.2 Rule-based Analysis

In the *analysis submodule*, there are two goals to achieve: architectural decision making, and conflicting decision resolution. We use production rules to capture the knowledge and strategies for these goals. The challenges here are to identify commonly used architectural decisions and the architectural properties required for making these decisions, choose an effective and concise representation scheme for facts, identify conflict conditions and resolutions, and design rules ro reflect these properties.

To illustrate how a decision making rule is defined and executed, we use the example decision tree shown in figure 2. In this decision tree, the property nodes are closely related to some well known architectural styles. We use this information to design the rules. For example, taking the first property node from the left, we can characterize it with a rule of the following form:

```
IF          (there exists consumer and producer
             relation)
THEN ADD    (use client/server style)
```

However, this rule does not capture key decision nodes along the branch and the source requirement. In addition, the representation used in the rule is not concise. We refine the rule to be the following:

```
IF          (property
             (uid ?id)
             (relation consumer-producer)
             (source ?reqID)
             (concern ?part))
THEN ADD    (style
             (name client-server)
             (property (uid ?id)))
```

where the variable `?id` is matched to the unique identifier of the decision unit (denoted as `property`); `?reqID` is matched to the unique source requirement identifier; and `?part` is matched to the part that the decision unit is concerned about at the top level of the decision tree: *component, system, bus,* or *property*.

When this rule matches a WME and is executed by the production system, the client-server style WME is then added to the fact base as a recommended architectural style to use for the part concerned.

Note that this rule is defined in close relation to what is in the sample decision tree. This is indeed the case when designing rules. They must provide means to match the decision units to architectural decisions.

When two decisions made by the rules are associated with the same part of concern, and cannot be used together in the architecture consistently, a conflict is identified. The conflicting decision resolution rules are designed to capture such conditions and provide solutions thereto.

4 Conclusions

In this paper, we have posed many open questions for bridging the gap between requirements and software architecture. Our research is motivated by these questions. In particular, we are interested in exploring the applicability of a unified description language for requirement specifications and architecturally significant properties, classifying architectural knowledge for building decision trees and production rules for requirement mapping and analysis, identifying the relationships between requirements, architectural decisions and properties.

We have proposed a framework to provide design guidance in eliciting architectural decisions from requirements, and a rule-based implementation. Although human interaction is required to map requirements to architecturally significant properties in the framework, we believe that using a tool implementing the framework can help us evaluate existing architectural decision-making knowledge, and define the relationships between requirements, architectural decisions and properties. Understanding of such relationships can help us to provide higher degree of automation and minimize human involvement.

References

[1] In J. Castro and J. Kramer, editors, *"The First International Workshop on From Software Requirements to Architectures (STRAW'01)"*. At the 23rd International Conference on Software Engineering, Toronto, 2001.

[2] B. W. Boehm and R. Ross. "Theory W Software Project Management: Principles and Examples". *IEEE Transactions on Software Engineering*, pages 902–916, 1989.

[3] R. J. Brachman and H. J. Levesque. "Knowledge Representation and Reasoning". In preparation, 2001.

[4] A. Egyed, P. Grünbacher, and N. Medvidovic. "Refinement and Evolution Issues in Bridging Requirements and Architecture - The CBSP Approach". In J. Castro and J. Kramer, editors, *"The First International Workshop on From Software Requirements to Architectures (STRAW'01)"*, pages 42–47. At the 23rd International Conference on Software Engineering, Toronto, 2001.

[5] R. Kazman, P. Clements, G. Abowd, and L. Bass. "Classifying Architectural Elements as a Foundation for Mechanism Matching". In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[6] W. Liu. "Rule-based Detection of Inconsistency in Software Design". Master's thesis, University of Toronto, Department of Computer Science, July 2002.

[7] W. Liu, S. M. Easterbrook, and J. Mylopoulos. "Rule-Based Detection of Inconsistency in UML Models". In L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar, editors, *Workshop on Consistency Problems in UML-Based Software Development*, pages 106–123. At the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, 2002.

[8] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems, 1996.