

A Rationale-based Model for Architecture Design Reasoning

Antony Tang

A thesis
submitted in fulfillment of the requirement for the
degree of
Doctor of Philosophy
Faculty of ICT, Swinburne University of Technology

February 2007

Abstract

Large systems often have a long life-span and their system and software architecture design comprise many intricately related elements. The verification and maintenance of these architecture designs require an understanding of how and why the system are constructed. Design rationale is the reasoning behind a design and it provides an explanation of the design. However, the reasoning is often undocumented or unstructured in practice. This causes difficulties in the understanding of the original design, and makes it hard to detect inconsistencies, omissions and conflicts without any explanations to the intricacies of the design. Research into design rationale in the past has focused on argumentation-based design deliberations. Argumentation-based design rationale models provide an explicit representation of design rationale. However, these methods are ineffective in communicating design reasoning in practice because they do not support tracing to design elements and requirements in an effective manner.

In this thesis, we firstly report a survey of practising architects to understand their perception of the value of design rationale and how they use and document this knowledge. From the survey, we have discovered that practitioners recognize the importance of documenting design rationale and frequently use them to reason about their design choices. However, they have indicated certain barriers to the use and documentation of design rationale. The results have indicated that there is no systematic approach to using and capturing design rationale in current architecture design practice. Using these findings, we address the issues of representing and applying architecture design rationale.

We have constructed a rationale-based architecture model to represent design rationale, design objects and their relationships, which we call Architecture Rationale and Element Linkage (AREL). AREL captures both qualitative and quantitative rationale for architecture design. Quantitative rationale uses costs, benefits and risks to justify architecture decisions. Qualitative rationale documents the issues, arguments, alternatives and tradeoffs of a design decision. With the quantitative and qualitative rationale, the AREL model provides reasoning support to explain why architecture elements exist and what assumptions and constraints they depend on. Using a causal relationship in the AREL model, architecture decisions and architecture elements are linked together to explain the reasoning of the architecture design. Architecture Rationalisation Method (ARM) is a methodology that makes use of AREL to facilitate architecture design. ARM uses cost, benefit and risk as fundamental elements to rank and compare alternative solutions in the decision making process.

Using the AREL model, we have proposed traceability and probabilistic techniques based on Bayesian Belief Networks (BBN) to support architecture understanding and

maintenance. These techniques can help to carry out change impact analysis and root-cause analysis. The traceability techniques comprise of forward, backward and evolution tracings. Architects can trace the architecture design to discover the change impacts by analysing the qualitative reasons and the relationships in the architecture design. We have integrated BBN to AREL to provide an additional method where probability is used to evaluate and reason about the change impacts in the architecture design. This integration provides quantifiable support to AREL to perform predictive, diagnostic and combined reasoning.

In order to align closely with industry practices, we have chosen to represent the rationale-based architecture model in UML. In a case study, the AREL model is applied retrospectively to a real-life bank payment systems to demonstrate its features and applications. Practising architects who are experts in the electronic payment system domain have been invited to evaluate the case study. They have found that AREL is useful in helping them understand the system architecture when they compared AREL with traditional design specifications. They have commented that AREL can be useful to support the verification and maintenance of the architecture because architects do not need to reconstruct or second-guess the design reasoning.

We have implemented an AREL tool-set that is comprised of commercially available and custom-developed programs. It enables the capture of architecture design and its design rationale using a commercially available UML tool. It checks the well-formedness of an AREL model. It integrates a commercially available BBN tool to reason about the architecture design and to estimate its change impacts.

Acknowledgements

I would like to thank my supervisor, Professor Jun Han, for his exceptional support, patience and guidance to the completion of this thesis.

I am indebted to Professor T.Y. Chen who has introduced me to the wonderful journey of research and has supported me along the way. It is my pleasure to have the opportunities to work with and learn from Dr. Yan Jin, Associate Professor Ann Nicholson, Mr. Muhammad Ali Babar, Professor Ian Gorton and Dr. Pin Chen. Their suggestions, critical comments and encouragements have been invaluable.

I would like to thank Sparx Systems and Norsys for providing the software tools in this research. NCR and Guangzhou Electronic Banking Settlement Center have graciously allowed me to use the Electronic Fund Transfer System as a case study and I am thankful to them. I am grateful to Nikhil Alave who helped implement the AREL tool in VB.Net. Finally, I would like to thank the many practising architects and designers who have devoted their time to provide valuable inputs for the survey and the empirical study.

This work is dedicated to my wife Alice.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Signed:

Dated:

Contents

1	Introduction	1
1.1	Design rationale	3
1.2	Research motivations and research questions	4
1.2.1	Motivations	4
1.2.2	Research questions and research approach	6
1.3	Research outcomes	7
1.4	Structure of the thesis	8
I	Problem Analysis	10
2	An industry’s perspective of design rationale in software engineering	11
2.1	Industry practice of design rationale	12
2.2	Architecture frameworks and design rationale	14
2.2.1	Zachman Framework	17
2.2.2	4+1 View	17
2.2.3	Federal Enterprise Architecture Framework	18
2.2.4	Reference Model for Open Distributed Processing	19
2.2.5	The Open Group Architecture Framework	20

2.2.6	DoD Architecture Framework	21
2.2.7	Architecture framework summary	21
2.3	Summary	23
3	Related work in design rationale	24
3.1	What is design rationale?	25
3.2	Why do we need design rationale?	26
3.3	Existing methods for capturing and representing design rationale	29
3.3.1	Issue-Based Information System (IBIS) and its variants	30
3.3.2	Questions, Options and Criteria (QOC)	32
3.3.3	Design Rationale Language (DRL)	33
3.3.4	Software Engineering Using Design RATIONale (SeuRAT)	34
3.3.5	Architecture Decision Description Template (ADDT)	35
3.3.6	Views and Beyond (V&B)	37
3.3.7	Quantitative reasoning methods	38
3.4	Other related studies	38
3.4.1	Requirements engineering	39
3.4.2	Requirements traceability	39
3.5	How well design rationale methods work?	42
3.6	Summary	45
4	Research methodology and validation	46
4.1	Software engineering research methods	47
4.2	The chosen research and validation methods	49

4.3	Summary	51
II Architecture Design Rationale Practice in the Software Industry		52
5	A survey of the use of architecture rationale	53
5.1	Architecture rationale in the software industry	55
5.1.1	Design rationale approaches in software engineering	55
5.1.2	Generic design rationale	56
5.2	Survey methodology	57
5.3	Survey findings	59
5.3.1	Demographic data	59
5.3.2	Job nature of architects / designers	60
5.3.3	Designer's perception of the importance of design rationale	61
5.3.4	Using design rationale	64
5.3.5	Documenting design rationale	66
5.3.6	Comparing usage and documentation of design rationale	69
5.3.7	Design rationale and system maintenance	70
5.3.8	Risk as a design rationale	73
5.4	Discussion of findings	74
5.4.1	Different forms of design rationale	74
5.4.2	The role of an architect	75
5.4.3	Designers' attitude	76
5.4.4	Necessity for design rationale documentation	76
5.4.5	Design rationale to support impact analysis	76

5.4.6	Risk assessment in architecture design reasoning	77
5.4.7	Methodology support for design rationale	77
5.4.8	Tool support for design rationale	77
5.5	Limitations	78
5.6	Summary	79

III The Representation and Applications of Architecture Design Rationale **80**

6	Representing architecture design rationale	81
6.1	A conceptual model for design reasoning	82
6.2	Architecture Rationale and Elements Linkage (AREL)	84
6.3	Architecture elements	87
6.4	Architecture rationale	89
6.4.1	Qualitative rationale	91
6.4.2	Quantitative rationale	92
6.4.3	Alternative architecture rationale	93
6.4.4	Avoiding cyclic decisions in AREL models	93
6.5	The extended AREL	96
6.6	A UML representation of AREL and eAREL	97
6.6.1	The architecture element stereotype	98
6.6.2	The architecture rationale stereotype	99
6.6.3	The architecture trace stereotype	100
6.6.4	The AE and AR supersedence stereotypes	101

6.6.5	AREL well-formedness in UML	102
6.7	AREL usability	102
6.8	Summary	103
7	A case study of using the AREL model	105
7.1	The EFT system	106
7.1.1	The EFT architecture overview	106
7.1.2	Fault-resilient support	109
7.1.3	Payment messaging	111
7.1.4	Transaction integrity	114
7.1.5	Specialised message control process	117
7.1.6	Centralised control	118
7.1.7	Message sequencing	119
7.1.8	Error reporting	121
7.2	An empirical study to validate the AREL model	122
7.2.1	Objectives of the empirical study	122
7.2.2	About the empirical study	123
7.2.3	Selecting experts	124
7.2.4	Empirical study results	125
7.2.5	Limitations	131
7.3	Summary	132
8	The architecture rationalisation method	133
8.1	Background	134

8.2	The architecture rationalisation method	136
8.2.1	Qualitative rationale	137
8.2.2	Quantitative rationale	138
8.3	Other applications of ARM	143
8.3.1	Completeness of architecture design	143
8.3.2	Verifiability of architecture design	148
8.4	Summary	149
9	Architecture rationale and traceability	150
9.1	Background	152
9.1.1	Issues with design rationale	152
9.1.2	Requirements and design traceability	152
9.2	Traceability of architecture rationale	153
9.3	Traceability support	155
9.4	AREL and eAREL traceability applications in a case study	156
9.4.1	Design rationale representation	156
9.4.2	Forward and backward tracing	159
9.4.3	Tracing architecture design evolution	163
9.5	Discussion	165
9.6	Summary	165
10	Architecture decision dependency and causality	166
10.1	Background	166
10.1.1	Related work	167

10.1.2	Introduction to Bayesian Belief Networks	168
10.2	Building a BBN to represent an AREL model	169
10.2.1	Nodes: Representing architecture elements and decisions	170
10.2.2	Edges: Representing causal relationships	171
10.2.3	Probabilities: Quantifying the causal relationships	172
10.3	Reasoning about change impact with AREL	177
10.3.1	An example	177
10.3.2	Original beliefs modelled by AREL	181
10.3.3	Predictive reasoning	182
10.3.4	Diagnostic reasoning	184
10.3.5	Combining diagnostic and predictive reasoning	186
10.4	Discussions and limitations	189
10.5	Summary	190
11	Tool implementation	191
11.1	Capturing architecture design rationale	192
11.2	Checking AREL models	195
11.3	Tracing AREL models	197
11.4	Analysing AREL with BBN	199
11.5	Limitations	201
11.6	Summary	203
12	Conclusions	204
12.1	Summary	204

12.2 Contributions	206
12.2.1 Design rationale survey	206
12.2.2 Design rationale representation	207
12.2.3 Design rationale applications	208
12.2.4 Tool implementation	210
12.3 Future work	210
Bibliography	212
IV Appendices	227
A – AREL tool user manual	228
B – Creating stereotype package in Enterprise Architect	237
C – A survey questionnaire on architecture rationale	239
D – A questionnaire to validate the AREL model	247
List of related publications	255

List of Figures

2.1	Conceptual Model of Architecture Description [70]	15
3.1	IBIS Design Rationale Model	31
3.2	PHI Issues Hierarchy	31
3.3	REMAP Model	32
3.4	An Example of QOC Design Rationale	33
3.5	A DRL Decision Graph	34
3.6	A RATSpeak Design Rationale Model	35
3.7	A Design Rationale Traceability Model	41
6.1	An Architecture Rationale Conceptual Model	83
6.2	A Causal Relationship between AEs and an AR	85
6.3	An AREL Diagram of a reporting sub-system, in UML	86
6.4	A Composition of Architecture Elements	88
6.5	Components of Architecture Rationale	90
6.6	Illegitimate cyclic graph: (a) AE-cyclic case (b) AR-cyclic case	94
6.7	(a) Cyclic Design (b) Acyclic Design	95
6.8	AE and AR evolution support in eAREL	97
6.9	«AE» Stereotype to extend Architecture Drivers	99

6.10	«AR» and «AAR» Stereotypes	100
6.11	«ARtrace» Stereotype	101
6.12	«AEsupersede» and «ARsupersede» Stereotypes	101
7.1	A Use Case of the EFT System	107
7.2	Processing Services of the EFT System	108
7.3	Decisions that Support Fault-resilient architecture	109
7.4	Message Control Processing	111
7.5	Intertwined Issues in the Architecture of MCP	112
7.6	General Payment Message Processing Sequence Diagram	115
7.7	Decisions to Support Transaction Recovery	116
7.8	MCP Connection Design	117
7.9	Decisions to Support Centralised Control	119
7.10	Decisions to Identify a Payment Message	120
7.11	Decisions to Architect Error Catching and Reporting	121
8.1	Key Activities in Architecture Design	135
8.2	Risk Assessments of Alarm Services	146
9.1	Asynchronous message processing decision and its design impact	157
9.2	Details of the design rationale AR10	158
9.3	Forward tracing for impact analysis	160
9.4	Backward tracing for root-cause analysis	163
9.5	(a) MAC processing (b) Superseded architecture element	164

10.1	A medical example: (a) BBN nodes, arcs and CPTs; (b) BBN graph without evidence; (c) patient has a cough (diagnostic reasoning); (d) patient has a cough and a heavy smoker (both predictive and diagnostic reasoning). . . .	169
10.2	Basic Forms of AREL Relationship in the BBN	171
10.3	Payment Messaging Design: Asynchronous Message Processing	177
10.4	Payment Messaging Design: Security	179
10.5	A BBN Representation with prior probabilities/CPTs of a Payment Message Design: (a) Asynchronous Message Processing (b) Security	180
10.6	An Example BBN Shown with Beliefs Before Any Evidence is Entered . . .	182
10.7	Predictive Model	183
10.8	A Diagnostic Model	185
10.9	A BBN Model of Combined Reasoning	188
11.1	The AREL Tool-set	192
11.2	An Example of AE Tag Values	193
11.3	An Example of AR Tag Values	193
11.4	The AREL Constructs for Modelling	194
11.5	A hierarchy of elements in an AR	195
11.6	An Example of a Qualitative Rationale (QLR)	195
11.7	An Example of a Quantitative Rationale (QNR)	196
11.8	AREL Tool Menu Options	197
11.9	AREL Tool Consistency Check Results	197
11.10	Window for Specifying AREL Trace Criteria	198
11.11	An Example of AREL Trace Result	199
11.12	A Process to Extract AREL Model from UML into BBN	200

11.13A Process to Synchronise Change between UML and BBN	202
--	-----

List of Tables

2.1	The Zachman Framework	17
3.1	An Analysis of the Usability Features of Design Rationale Methods	44
4.1	Research Questions and Research Methods	49
4.2	Case Study Verification	50
5.1	Frequency of Reasoning about Design Choices	62
5.2	Importance of Design Rationale in Design Justification	62
5.3	Frequency of Considering Alternative Designs	62
5.4	Importance of Each Generic Rationale	63
5.5	Design Rationale Frequency of Use	65
5.6	Frequency of Documenting Discarded Decisions	66
5.7	Frequency of Documenting Generic Design Rationale	66
5.8	Reasons for Not Documenting Design Rationale	67
5.9	Design Rationale Usage	69
5.10	Frequency of Revisiting Design Documentation before Making Changes	70
5.11	Tendency of Forgetting the Reasons for Justifying Design Decisions	71
5.12	Do Not Understand Design without Design Rationale if Not Original Designer	71

5.13	Design Rationale Helps Evaluate Previous Design Decision	71
5.14	Frequency of Performing Impact Analysis	72
5.15	Importance of Each Impact Analysis Task	72
5.16	Correlation between Use of Design Rationale and Impact Analysis Tasks . .	73
6.1	An Analysis of the Usability Features in AREL	103
7.1	Empirical study - Question 1 Results	125
7.2	Empirical study - Question 2 Results	126
7.3	Empirical study - Question 3 Results	127
7.4	Empirical study - Question 4 Results	127
7.5	AREL Usefulness in Supporting Architecture Design Reasoning	128
7.6	Experts' Willingness to Use AREL	128
8.1	A Comparison of the Architecture Costs and Benefits	140
8.2	The Expected Architecture Cost and Benefit Ratio	142
10.1	Volatility of Architecture Elements and Validity of Architecture Rationales over a Sequence of Changes	187

Chapter 1

Introduction

Software engineering has been maturing over the years as increasingly sophisticated methods become available to guide developers. A general progress is the increasing level of abstraction to represent system models and implement them. The purpose is to handle complexity. There are many such examples: the transition from assembly programming to structured programming in high-level languages and then to object-oriented programming, and the transition from program design to software architecture design.

Software architecture design provides a high-level abstraction of a system. It is an important area of research in recent years because it lays the structural foundation of a system. It allows designers to visualise that a design is viable and that it would satisfy key requirements. When performing architecture design, architects consider both the technical and non-technical aspects of a system. Aspects such as requirements, project schedule, budget, information technology strategies and design trade-off are only some of the considerations in architecture design. Such diversity of considerations makes it challenging to balance the conflicting interests in an architecture design.

The question to address is how architects should organise and tradeoff between a wide range of considerations to provide a quality design. It has been suggested that this aspect is fundamental in architecture design for a system to succeed [47]. For instance, a software architect cannot consider security without ensuring that the performance of the system remains satisfactory. Similarly, an architect cannot design a system without considering the IT budget and the legacy systems that are in place. But different architects have different ideas on how to design based on individual experience and approach. Intuitively, we know that the results and the quality of the architecture design may differ depending on who the architect is. We may prefer to trust a more experienced architect with good track records rather than an inexperienced architect. This highlights a fundamental issue

that architecture design is highly dependent on the person who performs it. Therefore, the quality of the architecture design depends on an architect's experience, knowledge and decision making abilities [169].

Methodologies to support software architecture development have been proposed by many such as [64, 8, 23]. There are also published standards for architecture design and architecture frameworks [70, 164]. Although these methodologies support architecture design, very few prescribe on how architecture design decisions could be made and verified, and as such decision making is largely software architects dependent.

Software architects conveniently assume that they make the correct design decisions based on their intuition of the problems and their understanding of the potential solutions. The question of whether an architecture design is the most suitable one is vaguely addressed. For instance, a set of requirements are given to two designers, we cannot predict how similar or different their design and design qualities might be. The differences between two designs might be attributed to how contextual information are interpreted in decision making. Good decisions result in high quality system design and bad decisions result in poor quality design. Therefore, design decision is an important aspect in architecture design.

Argumentation-based design rationale methods have been proposed to capture design deliberations [93, 99, 97]. These methods have provided the initial research in design rationale. Despite their efforts, there is little evidence that they have been adopted by the software industry. Some have argued that these methods do not effectively capture and communicate design rationale for practical use [139, 132]. But mostly design rationale are not captured at all or they are not captured in a way that could be used effectively.

Without a systematic way to elicit and represent design rationale, the knowledge and the understanding of the architecture design can be eroded over time [119, 11]. Furthermore, the documentation of design rationale as evidence to support architecture design is usually not mandated, making it difficult to review and evaluate architecture design. Such erosion of architecture design rationale has a number of implications on software development. The understanding of a design can diminish over time and cannot be shared across the organisation. As a result, architecture design cannot be justified, validated and maintained easily. On the other hand, systematic capture of design rationale can provide much insights into the architecture design, thereby supporting evaluation, verification, traceability and maintenance of complex systems. Without design rationale, maintainers could find it difficult to understand the design when the underlying assumptions, constraints and tradeoffs are missing, and the design reasoning of the architecture cannot be traced.

In this thesis, we address the issue of the lack of design rationale in a number of ways. First, we investigate the use and documentation of architecture design rationale in the software industry by way of a survey. We conjecture and test that architecture design often relies on the experience and intuitions of architects instead of using systematic and objective methods for rational decision making. We then address the issue of an appropriate representation and applications of architecture design rationale. The main result of our work is a model, Architecture Design Rationale and Element Linkage (AREL), for capturing and representing architecture design rationale. This model is aligned with the industry architecture design practice to support its application. AREL facilitates design reasoning by allowing architects to trace requirements to related design decisions and design objects, and supporting the analysis of the underlying assumptions and constraints that affect a design. This analysis can be carried out in a qualitative way to explore the design justifications, or in a quantitative way to estimate the probability of change impact.

1.1 Design rationale

Design rationale is a reason or an intention in the act of designing. When design decisions are made, what is considered as a reason or an intention? and how could a decision be justified? There are different interpretations of design rationale. Moran and Carroll suggested that design rationale are reasons to express the purposes of the designed artefacts with their contextual constraints on realising the purposes [103]. According to Conklin and Burgess-Yakemovic [24], design rationale can have the characteristics of recording the history of how a design comes about through recording logical reasoning to support future reference. Carroll and Rosson [18] suggested that design rationale can be viewed as psychological claims that are embodied by an artefact whereby design deliberations through evolution can be assessed. Maclean et.al [97] claimed that design rationale can be a description of the design space and used to deliberate design decisions.

There is no lack of software architecture development methodologies in the industry, then why do we need to consider design rationale? Researchers in this area have argued that it can be used to improve the design decision making process by capturing, representing and reusing design knowledge lest that they might be lost. Perry and Wolf [119] suggested that as architecture design evolves, the system is increasingly brittle due to two problems: *architectural erosion* and *architectural drift*. Both problems may lead to the violations of the architecture design over time because the underlying rationale is not available to support the architecture design. Bosch suggested that architecture design decisions are crossing-cutting and inter-twined, so the design is complex and prone to erroneous interpretations without a first-class representation of design rationale [11]. As

such, the design could be violated and the cost of architectural design change could be very high and even prohibitive. Kruchten et al. suggested that architecture knowledge consists of the architecture design and its design rationale [86].

Despite the work by researchers such as [18, 24] and the standards organisations such as [70, 164], the adoption of design rationale is sporadic. Design decisions still rely heavily on the intuition and the experience of individual architects irrespective of the development methodology being employed. The quality of an architecture design therefore largely depends on the architecture decisions made by architects. Therefore the study of making rational and objective design decisions is worthwhile.

1.2 Research motivations and research questions

Architecture design is one of the early steps and arguably one of the most important steps in the software development life-cycle (SDLC). It provides a foundation to construct a system. Architecture design decisions thus become early design commitments in the SDLC. Activities of architecture design such as the procurement of platforms, determining interface standards, outlining system design structure and determining development standards are decided and executed at an early stage of the design process that can be difficult to change or reverse later. Hence it is important to ensure that the decisions behind the architecture design are correct. In this section, we describe the motivations and the research questions addressed by this thesis.

1.2.1 Motivations

The software engineering community has long been aware of the issue relating to the loss of design rationalisation knowledge. This phenomenon is evident in a lack of architecture design justification process that uses design rationale and a lack of design rationale documentation [11]. Researchers have put forward many suggestions to address this issue [14, 129, 33, 124, 94]. Argumentation-based design rationale methods address the issue of deliberation but they face challenges in their implementation (see Chapter 3). Methods such as ATAM, CBAM, and Views and Beyond to aid architecture design have also been suggested [8, 23]. However, the software development industry does not seem to have adopted such practices and it is unclear what methods are used instead, if any. We conjecture that the following factors contribute to the difficulties in design rationale implementation. We subsequently provide evidence through a survey to prove some of them:

- the argumentation-based design rationale systems do not effectively capture and communicate design reasoning;
- some of the design rationale methods are not practical in the industry setting because they do not work effectively with the development process;
- practitioners are not aware of the benefits to explicitly justify and capture design decisions, so they continue the traditional design practice without an explicit reasoning process;
- lack of methodology and tool support.

If design rationale is not used systematically, and architects continue to practise unstructured decision-making instead of an objective and verifiable reasoning process [156], a number of issues may arise:

- architects may not have considered all the factors that influence a design, resulting in a system with inferior quality;
- architecture decisions do not have to be justified in an objective way, therefore the argumentation, analysis and tradeoffs about a decision could be incomplete or biased;
- the verification of the architecture design remain a subjective exercise and its quality assurance ability questionable;
- the reasoning of the architecture decisions is not systematically captured and can “evaporate” over time;
- change impact analysis during architecture design maintenance requires reasoning support and the lack of such knowledge can lead to difficulties in understanding and maintaining a system;
- unstructured design rationalisation may cause difficulties in future system enhancements such that a minor change in requirement may cause a major architecture design change, therefore increases the risks and costs of a project inproportionally.

These are fundamental issues which have not been studied in detail and they relate primarily to architecture design rationale. Thus, we need to study the use and documentation of architecture design rationale. In the next section, we discuss the related research questions and the research approaches.

1.2.2 Research questions and research approach

Argumentation-based design rationale methods have mostly focused on design rationale deliberation but they have largely omitted the relationships with design elements. These methods are ineffective in the capture and communication of design rationale [139]. Given these shortcomings, we aim to investigate the following questions in this thesis to provide some improvements:

- In order to establish the usefulness of this research, the following background questions have been investigated:
 - Is design rationale important in architecture development?
 - Is design rationale important in system maintenance?
 - What is the current state of practice of design rationale in the software industry?
- How to improve the representation of design rationale for architecture development?
- How to implement traceability between requirements, architecture design elements and design rationale?
- How to quantify and estimate change impact analysis using architecture design elements and design rationale?

Since these research questions are inter-related, they need to be examined together in a holistic way. Our research approach is to first establish the importance of design rationale for this work to be meaningful. The perception that design rationale is useful is supported only by anecdotal evidence. Empirical evidence is thus required to establish that design rationale is indeed useful for architecture design and maintenance. We also need to determine what are the key elements in design rationale. To this end, we have conducted a survey involving architects and designers in the Asia Pacific region to study the usefulness and applications of design rationale.

Secondly, we analyse the implementation issues that exist in current design rationale methods. Using the analysis as a guideline, we designed the AREL representation scheme to overcome those issues. We then validate the AREL method using an empirical study to compare the effectiveness of AREL with traditional design specifications. Finally, we study the impact analysis and the traceability aspects of design rationale to propose methods for their applications. We use a case study based on an electronic payment system to demonstrate the design rationale applications.

1.3 Research outcomes

This section gives a summary of the research results. Firstly, the results of our survey have established the current practice of using and documenting design rationale. We found that most architects see the importance of design rationale. 85% of respondents told us that design rationale is important or very important, but only 43.3% of them document design rationale often. The ways they document design rationale vary and are not systematic. In the survey, we have investigated different types of design rationale that are useful to practising architects. The results indicate that there are no established methodologies for capturing and using design rationale.

Secondly, we have developed the AREL model to address the issue of design rationale representation. It represents architecture design rationale and their relationships with architecture elements. AREL has three objectives: (a) it must effectively capture design rationale to explain why design objects exist; (b) it must not interfere with the natural process of architecture design during design rationale capture; (c) the retrieval of design rationale is easy to achieve.

We use the Unified Modelling Language (UML) notation to represent the AREL model. Architects can capture and document design rationale whilst they are designing and documenting the system architecture in UML. Requirements, assumptions and constraints are used as the architecture drivers. These inputs or drivers motivate decisions to be made to create architecture design objects. Architecture design decisions are the results of the tradeoffs between factors such as costs, benefits, risks and weakness. The AREL model captures the design rationale during the decision process to provide a framework to explain the architecture design reasoning.

Finally, different applications of AREL have been developed to enhance the use of design rationale:

- Architecture development process - the Architecture Rationalisation Method (ARM) is a design rationale centric process. By applying ARM, architects can make use of AREL to facilitate decision making and exploring the design space until the architecture design is relatively risk free.
- Traceability to support design reasoning - we propose three traceability methods (forward, backward and evolution) to support the traversal of design rationale. This application allows architects to explain the dependency between requirements and design objects qualitatively for verification and maintenance purposes.
- Quantitative Analysis - we use Bayesian Belief Networks to quantitatively analyse

the change impacts of a system. This application provides a way to assess the likelihood of change, in terms of probabilities, in various parts of the system when requirements or design are changing.

We use a central bank electronic payment and settlement system as a case study to demonstrate AREL modelling and its applications. In an empirical study to evaluate the usefulness of AREL, experts were involved to compare the original design specifications and the AREL model for their effectiveness in explaining the design. The empirical study has shown that the majority of design rationale are recalled from architects' memory or are deduced by them. This is because design rationale is not completely documented in traditional design specifications. The empirical study has also shown that AREL serves well in capturing design rationale to help the experts understand the design reasoning.

A set of tools have been created to capture design rationale and support its applications, as currently existing design tools lack the abilities to link design objects to their rationale. A standard UML tool, Enterprise Architect, has been enhanced to support design rationale capture. We have developed the AREL Tool to check AREL model consistency and to support design reasoning analysis and tracing. We have made use of Netica to analyse impact analysis using Bayesian Belief Networks. The integrated tool-set demonstrates the applications of AREL in a practical industry setting.

1.4 Structure of the thesis

The thesis is organised as follows:

Chapter 1 gives an introduction and a context for the subsequent chapters of the thesis. In this chapter, the research background, research questions and the results are briefly described.

Chapter 2 describes the application of design rationale from an industry perspective. It provides a background on how the software industry in general deals with design rationale. Different architecture frameworks are examined in particular for their treatments of design rationale.

Chapter 3 explores the related work of design rationale published in the literature. In this chapter, different design rationale methods are described. A comparison is made to highlight the merits and limitations of these methods. Other work which is related to the application of design rationale are also cited and discussed.

Chapter 4 describes the research methodologies used in this thesis.

Chapter 5 describes a survey of practising software architects. The results of this survey has not only provided proofs to motivate the research in this area, it has also provided directions for formulating the architecture design rationale model and its applications.

Chapter 6 describes the Architecture Rationale and Elements Linkage Model (AREL) which is used to capture and represent design rationale. The elements and the structure of AREL are discussed in details.

Chapter 7 uses a case study to demonstrate how AREL is used in capturing and communicating architecture design rationale. The case study is about a central bank electronic payment and settlement system called Electronic Fund Transfer System (EFT). This system has been in production since 1998. AREL is used retrospectively to capture the design rationale of the system. In an empirical study, experts have evaluated the AREL representation using the case study as an example.

Chapter 8 describes the Architecture Rationalisation Method (ARM) which is a design rationale centric method to facilitate architecture development. This chapter describes how ARM helps architects construct an architecture design using qualitative and quantitative design rationale.

Chapter 9 describes the traceability application of the AREL model. Three types of traceability applications are shown. They describe how qualitative design rationale can be used to help understand the design reasoning. The case study is used to demonstrate design rationale traceability in AREL.

Chapter 10 describes change impact analysis using AREL and a quantitative method called Bayesian Belief Networks. A detailed discussion of the causality between architecture design rationale and architecture elements are made. Using this relationship, architects can estimate the likelihood of change across a system. An application of this method is demonstrated by using the case study.

Chapter 11 reports the implementation of the AREL tool-set to support AREL applications.

Chapter 12 concludes this thesis by outlining the major contributions and benefits of this work. It also discusses the areas where further research is required.

Part I

Problem Analysis

Chapter 2

An industry's perspective of design rationale in software engineering

Designers working in the software industry often use intuition and experience to rationalise their design decisions. Their design reasoning are generally carried out implicitly. The drawback of such an implicit approach is that the quality of decisions would heavily depend on the experience and expertise of the individuals. It has been established that design rationale play an important role in the design decision making process [33, 12], it would be useful to ascertain if the software industry adopts a structured and effective design rationalisation process and if they capture and manage design rationale.

There are a number of methods that offer design reasoning support, examples are ATAM [8], CBAM [4], View and Beyond [23], Use Case Driven Software Development [33] and SeURAT [14]. However, it seems that they have not been adopted by the software industry generally. The lack of guidance in design decision making means that personal experience plays a large role in design activities. Hence, designers with less experience could make less than optimal or even erroneous design decisions. This could be one of the reasons why the success of projects are related to the capabilities of technical personnel [74].

In this chapter, we examine the software industry practice of design. We discuss architecture design methodologies and how they make use of design rationale. Finally, we discuss the implications of lacking design rationale to enterprise systems.

2.1 Industry practice of design rationale

Software engineering research is often motivated by problems which arise in the production and the use of real-world software systems [138], therefore it is necessary to examine the real-world to provide a context of the problems being addressed in software engineering research. Architecture design plays a prominent role in software and system development. Its necessity is due to the highly complex and integrated systems which are typical of today's enterprise system environment. Such systems usually have sub-systems and components which are intricate and interdependent. They require continuous maintenance, enhancements and integration during their long life-span. Insufficient attention to architecture design may result in poor quality systems which are difficult and costly to maintain.

Projects or systems may fail because of poor architecture design. There have been a number of reports on such failures. The Standish Report [165] showed that there was an alarmingly high percentage of projects that either do not complete successfully, do not meet project objectives or have significant cost and schedule overrun. May at Crosstalk [98] reported that most of the project failure causes originate before the first line of code has been written. Poor user inputs, stakeholder conflicts, vague requirements and poor architecture are some of the failure reasons named in the report. In Jones's report [74], many ways were suggested to tackle project issues. An example is to reduce the costs and the risks of software development by producing quality and reusable software artefacts. Peter Neumann's column [106] on risks regularly reports IT system failures due to various reasons. Despite reporting of project and system failures over many years, and recommendations for remedying the problems, there is no shortage of such failures. Why?

Development methodologies are well adopted by the IT industry. Quality systems such as Integrated Capability Maturity Model (CMMI) for software and system development [16] and Six Sigma Total Quality Management [144] are receiving more acceptance by large organisations such as IBM, NCR General Electric and Lockheed. IEEE and ISO have issued standards such as Software Requirements Specification [69], Software Life-cycle Processes [67, 68], Architecture Development [70] and Reference Model for Open-Distributed Processing [71] to guide software development. Software and service providers such as IBM and Accenture also have their proprietary methodologies and tools to support software and system development. Most IT organisations employ some forms of development standards and methodologies. With so many methodologies and standards in place, why are there still quality issues, cost overrun and project failures? Are practitioners not following the methodologies and standards, or are the methodologies and standards lacking in any way, or simply the project failure measurements are inaccurate [35]? There may

be more than one answer and most likely the answers are non-trivial. However, there is something fundamental that underlies such project issues - the soundness of the design decisions.

Software engineering practices mostly focus on the design of concrete and visible artefacts and either omit or treat the decision making documentation as separate [43]. Design rationale is commonly ignored and undocumented. Most of the time they are not mandated as part of the software deliverable. Even though architecture design is thought to be a very important stage in the development process, the architecture design rationale are usually not rigorously tested in the design review process. The treatment of those factors which influence the design decisions are usually implicit and ad hoc. For instance, architecture decisions would make assumptions on factors such as delivery schedule, level of expertise in the development team and platform stability, but these assumptions are mostly implicit. Designers could be biased toward personal preferences and agenda that would affect the objectivity of the design decisions.

Given that design decisions are predominantly an implicit process, it is possible and quite common that a series of incorrect decisions can escape quality checks set out by development methodologies and standards, resulting in project failures. Thus we posit that improving the decision making process can help minimise mistakes and failures. Along with many other researchers [11, 12, 124], we argue that making quality decisions is an essential part of software development. High quality design requires that each decision is rational and that all relevant factors which influence the decision are well considered. As such, the system design process will improve if a methodology to enhance design decision making is available.

In a successful project, planning and architecture design are two of the crucial steps [154]. These two factors are interrelated in that the project plans set the constraints for the architecture design, and the architecture design dictates the resources required in a project. For instance, design decisions made during the architecture design dictate the resources required to develop the system. Even though the role of architecture design is very important in a project, its decisions are often not sufficiently justified. Less than optimal decisions could be made and risk could arise when decisions are not justified systematically. The author has numerous encounters of such incidents and some real-life examples are illustrated below:

- Design for the Future - arguing for reusability or flexibility, the architects make architecture design overly complex and costly without delivering additional benefits. The reusable or flexible features are actually never required or necessary [15].

- Schedule Constraints - management may have a budget and schedule constraint on the project. In view of such constraints, architecture designs are compromised to meet the schedule. The opportunity cost incurred is that the future enhancements are more costly [98].
- Platform Selection - selecting a software platform such as a database system is sometimes influenced by reasons that are not technical or financial in nature. It could be due to political reasons. As such, non-functional requirements such as performance, security and features that are essential to satisfy system requirements can be overlooked and compromised [98].
- Design Choices - the choice of a technical design may be influenced by architects' familiarity with certain technologies. It may be a subjective opinion based on what designers are comfortable with rather than an objective rationale to measure the effectiveness of a solution. On the other hand, some architects may choose to use a new technology not because it is a suitable solution but because the architects want to gain experience in such technologies.

Mystical software architectures without justified design decisions are quite common [170]. Creators of such architectures seem to want the project stakeholders to take many things on faith. There are many assumptions about the architecture that it could satisfy the business drivers, deliver the requirements and are implementable. Defects might be entrenched in the architecture design unknowingly. Existing quality and software engineering methodologies might not be able to detect these defects because design decisions are unjustified and undocumented. Design rationale are implicit in the design process and are probably not captured systematically, so they do not persist and can be lost over time. In order to address the issue of design knowledge evaporation, we study the process of architecture design rationalisation and the retention of design rationale.

2.2 Architecture frameworks and design rationale

Software systems are becoming more complex as more components are used in their construction, thus the organisation of the overall system - the software architecture - presents a new set of design problems [46]. With that complexity, changes in an architecture are more difficult because they would affect a large part of the system [105]. To manage the complexity in architecture design, a number of architecture frameworks [21, 31, 30] and research work [119, 46, 147, 90, 85] have been devoted to this subject.

A common approach to organising system components in an architecture is by using

important:

- The impact of the architecture design is high. Early architecture decisions are commitments which are difficult to reverse or change once they have been made. This is because a simple change of the architecture design later could involve major cost and schedule impact to the project [7]. For instance, a change in the technology platform (i.e. an architecture design change) at the time when the system is being deployed would have a large impact on the schedule and the cost of the project.
- The complexity of the architecture design is high. Architecture decisions are concerned with satisfying goals and requirements whilst dealing with technical and project issues. Compromises and tradeoffs between competing issues and requirements are common. The process to reach a set of balanced decisions which are acceptable by all stakeholders can also be difficult.
- Potential risks of architecture design is relatively high. Architecture design often involve integrating and utilising multiple system and software components. The behaviour of the final outcome before implementation is sometimes unclear. Therefore, careful considerations and rationalisation of the combined behaviour of system components are required to mitigate the risks.
- Architecture design requires communication and coordination. Large scale systems often involve multiple stakeholders who need to communicate and coordinate with each other [27]. Failure to communicate essential knowledge such as design reasoning could affect the quality of the design decisions.
- Knowledge retention is low. If careful decision making process in architecture design is important, then the retention of this reasoning knowledge is equally important [124]. This is because architects who have to maintain the system would require such design rationale to support architecture enhancements in the future.

Architecture frameworks provide a structured approach to designing system and software architecture. Some of them have provisions to support design rationale. These architecture frameworks provide features similar to the architecture description standard [70]. Notable examples are 4+1 View [84], The Open Group Architecture Framework (TOGAF, [164]), Federal Enterprise Architecture Framework (FEAF, [21]), Open Distributed Processing - Reference Model (RM-ODP, [71]), Department of Defence Architecture Framework (DoDAF, [31]) and the Zachman Architecture Framework [148]. The following sections describe these industry-based architecture frameworks.

2.2.1 Zachman Framework

The Zachman Framework (ZF) for Enterprise Architecture [176] is one of the earliest works in this area. It divides the architecture into a number of perspectives to represent different views of the system. ZF's key goals are for enterprise architecture analysis and modeling and it is also concerned with perspectives of constructing an information system. As shown in Table 2.1, a perspective is a row in a table representing how a stakeholder in a project team would view the system. The various stakeholders are *Planner*, *Owner*, *Designer*, *Builder* and *Subcontractor*. Each perspective would produce their respective outcomes such as *Scope Document*, *Enterprise or Business Model*, *System Model*, *Technology Model* and *Components*.

Table 2.1: The Zachman Framework

	What (Data)	How (Function)	Where (Network)	Who (People)	When (Time)	Why (Motivation)
Planner	Business Things	Business Process	Business Locations	Major Organisation	Major Business Event/Cycle	Major Goal Strategy
Owner	Semantic Model	Business Process Model	Business Logistic System	Workflow Model	Master Schedule	Business Plan
Designer	Logical Data Model	Application Architecture	Distribute System Arch.	Human Interface Architecture	Processing Structure	Business Rule Model
Builder	Physical Data Model	System Design	Technology Architecture	Presentation Architecture	Control Structure	Rule Design
Sub-contractor	Data Definition	Program	Network Architecture	Security Architecture	Timing Definition	Rule Specification
Functioning System	Data	Function	Network	Organisation	Schedule	Strategy

The framework specifies, for each perspective, different types of information that are characterized by (a) *what* - information and data; (b) *how* - function and process; (c) *where* - location of hardware / software; (d) *who* - people in terms of allocation of work and authority; (e) *when* - timing requirements of business process; (f) *why* - motivation. The *why* in the Zachman Framework provides the the context of the requirements, i.e. what motivates the requirements or the design. This type of reasoning is different to design rationale that explains why design decisions are made, i.e. why do I choose this design option but not another one.

2.2.2 4+1 View

The 4+1 View Model of Architecture is a framework for modelling software architecture [84]. It represents the *logical*, *process*, *development*, *physical* and *scenario* views of the system. The goals of 4+1 View Model is for architecture analysis and modelling of software systems. The framework uses four viewpoints to represent architecture models and a scenario view for discovery and verification.

- Logical View - represents the functional requirements of the system
- Process View - this view facilitates partitioning of software into independent software tasks that represent running processes and their inter-process communication in a distributed environment, taking into account non-functional requirements
- Development View - this view focuses on the organization of software modules
- Physical View - this view denotes mapping of software to hardware nodes
- Scenarios - scenarios or instances of use cases are used to discover and test the architecture design

The 4+1 View Model provides an iterative approach of architecture design through analysis and decomposition of design issues. It uses the UML notation for representing the models. Although the model mentions using design decisions, there is little details on how they should be documented and used.

2.2.3 Federal Enterprise Architecture Framework

The US Federal Government issued a standard Federal Enterprise Architecture Framework (FEAF) version 1.1 [21, 22] to guide architecture development in government agencies. FEAF is a framework issued by the US CIO Council to promote shared development for common US Federal processes, interoperability, and sharing of information among Federal Agencies and other Government entities. The framework is organized in 4 levels.

Level I is the highest level view which deals with architecture drivers or external stimulus and strategic direction of architecture. It facilitates the transformation of the current architecture to the target architecture through applying architecture standards and managing the architecture process. Level II provides more details by analysing the business drivers and design drivers of an architecture. The outcome of this process is the target business architecture and the target design architecture.

Level III expresses the architecture in more details by using business, data, applications and technology views to model the target architecture. Level IV uses a combination of ZF and Spewak's Enterprise Architecture Planning (EAP) methods [151]. ZF columns of data, functions and network are used to represent Data Architecture, Application Architecture and Technology Architecture. Again, architecture design is represented in the FEAF model but the decision making process and the design justifications are omitted.

2.2.4 Reference Model for Open Distributed Processing

The International Standards Organization (ISO) in conjunction with the International Telecommunication Union (ITU-T) issued a Reference Model for Open Distributed Computing (RM-ODP) for architecture development of distributed systems [71]. The ISO RM-ODP Standards are a set of international standards with four parts. Part 1 (ISO 10746-1/ITU-T X.901) provides an overview and a guide to the use of the reference model. Part 2 and Part 3 (ISO 10746-2/ITU-T X.902 and ISO 10746-3/ITU-T X.903) provide a foundation of concepts and they prescribe concepts, rules and functions for the modelling of ODP systems. Part 4 (ISO 10746-4/ITU-T X.904) is the architectural semantics which provide a formal description technique for Part 2 and Part 3. The primary objective is to allow the benefits of distribution of information processing services to be realized in an environment of heterogeneous IT resources and multiple organization domains.

RM-ODP uses five viewpoints to represent different aspects of a system. The Enterprise Viewpoint states high-level enterprise requirements such as (a) purpose and objectives of systems, (b) community or users of system and (c) business policies, guidelines, flows and constraints and (d) actions performed. The Information Viewpoint focuses on information semantics and information structures.

The Computational Viewpoint focuses on the decomposition of the system and on the constraints of the objects and their interactions. The objects specified and modelled can be computational, service support or infrastructure objects. Interactions between the objects are connected through interfaces. The Engineering Viewpoint focuses on the mechanisms and functions that support interactions between distributed objects. The Technology Viewpoint specifies the choice of technology, including products, standards and technology objects, selected to support the implementation.

RM-ODP provides the standards to define transparencies for the support of distributed processing. Transparencies are architecture patterns that are defined in the Engineering Viewpoint. An example of a transparent function is *access* transparency which masks differences in data representation and invocation mechanism to enable different heterogeneous objects to work together. RM-ODP primarily focuses on ODP architecture development. Architecture rationale and tradeoffs are not documented as part of the model. RM-ODP is formal and it provides a complete and consistent model for the specification of system architecture design.

2.2.5 The Open Group Architecture Framework

The Open Group, an industry standard organization, issued The Open Group Architectural Framework (TOGAF) version 8.1 in 2003 [164] to guide enterprise architecture development. TOGAF's goals are to provide a framework for the design, evaluation and building of architectures for enterprises. A key element of TOGAF is TOGAF Architecture Development Method (ADM) which specifies a process for developing enterprise architecture. The Enterprise Continuum is a virtual repository of all architecture assets that include models, patterns and architecture descriptions. The TOGAF Resource Base is a set of resources, guidelines, templates and background information to assist in the use of TOGAF. TOGAF ADM is a generic method which specifies an iterative approach for architecture development. ADM is not prescriptive on the breadth of coverage, the level of details, the extent of time horizon or the architectural assets to be leveraged. These can be determined by the architects to suit a particular project. The phases defined by ADM are the following.

- Preliminary Framework and Principles to define the baseline of the architecture within an enterprise
- ADM Cycle defines the architecture development cycle
- Requirements Management process is central to the ADM Cycle where it identifies, stores and interfaces requirements with all phases of the ADM Cycle.

The TOGAF Enterprise Continuum specifies a Technical Reference Model (TRM). TRM is a model that represents a system in terms of the Application, the Application Platform and the Communication Infrastructure and their inter-connectivity. TRM also describes Service Qualities provided by the system. TOGAF ADM is a comprehensive methodology that addresses architecture at the enterprise level as well as the individual system level. Its methodology supports the architecture evolution through using Enterprise Continuum as its knowledge base. Activities in each phase of the ADM framework are well defined but it leaves the implementation flexibility to practising architects to determine what is required for the system from a defined set of possible outcomes. TOGAF recommends the documentation of design rationale to trace design and architecture decisions. However, it does not elaborate as to what and how design rationale should be documented.

2.2.6 DoD Architecture Framework

The US Department of Defense released the DoD Architecture Framework (DoDAF) version 1.0 [31] for DoD architecture compliance. DoDAF Version 1.0 is developed specifically for the US DoD to support its war-fighting operations, business operations and processes. It was developed from and superseded the previous architecture framework C4ISR Architecture Framework Version 2.0.

Architecture development techniques have been provided in DoDAF to specify the processes for scope definition, data requirements definition, data collection, architecture objectives analysis and documentation. DoDAF uses Core Architecture Data Model (CADM) for architecture documentation. CADM is a standardized taxonomy to define views and their elements in a database. All Views provide an overview, summary and integrated dictionary of the architecture; Operational Views describe the business and the operations of the architecture, they describe the operation nodes, nodes connectivity, information exchange, organization relationship, operation rules, event-trace and logical data model; System Views describe the system and its components; Technical Views describes the current standard profile and the future technical standards forecast.

The DoDAF framework is specifically designed to support defense operations and therefore some of its processes and taxonomies are domain dependent. CADM is a well defined schema to support the documentation of architecture models in this domain. Using CADM and traceability matrix, operational requirements and design decisions can be traced in the architecture. DoDAF does not have provisions to record architecture rationale.

2.2.7 Architecture framework summary

Architecture frameworks use viewpoints to represent different perspectives. The interpretation of viewpoints are slightly different to suit the objectives of each framework. However, the viewpoints can be generalised by their common concerns:

- Business Viewpoint - represents the owners and stakeholders' requirements. Some frameworks broaden the interpretation to not only include functional requirements, but non-functional requirements and business environments.
- Data Viewpoint - represents the information and data that are kept within the system. Such information can include the data within the database and other information repositories; some frameworks extend their definitions to include all information that are kept and exchanged between systems.

- Applications Viewpoint - the software design of the system. It represents the high-level design of the system. Some frameworks have further classifications such as a software configuration model and a model representing process execution.
- Technology Viewpoint - the technologies which support the system. This represents the hardware and software platforms and their organisations in the architecture.

Although some of the frameworks specify the processes and the methodologies that are used in the construction of these viewpoints, for instance, DoDAF have suggested to use design tradeoffs [31] and TOGAF specifies the capture of design rationale [71, 164] in their models, there is still very little considerations on the decision making process which underpins and justifies the architecture. In an earlier study to compare architecture frameworks [160], it was found that these architecture frameworks do not mandate rationale capture nor do they specify how rationale should be represented or used. Without a proper guideline, architecture design rationale is probably not captured in these large enterprise systems. There are three implications:

- **Lack of Justifications** - A lack of design rationale documentation in large enterprise systems development implies that stakeholders would have to assume that the architecture design decisions that are made are comprehensive and correct.
- **Knowledge Loss** - The knowledge that is required to support maintenance in the long-term may be lost because the rationale behind the architecture design are gone when original designers are no longer available.
- **Quality Depends on Designer Experience** - Without a systematic design rationalisation process, architecture design and its quality would have a high dependency on the level of experience of the designer.

Although the IEEE 1471-2000 standard [70] and some of the architecture frameworks discuss the importance of design rationale, none of them provide much guidance in the use, the capture or the representation of design rationale. This does not necessarily mean that the architecture design are inferior, but the lack of an objective decision-making framework would place a higher reliance on the experience and the quality of the decision makers whereas a well-defined decision making approach can provide a systematic way to ensure architecture design qualities.

2.3 Summary

There are many software development standards and methodologies to improve the quality of the system and better the chance of success in IT projects. However, IT project failures are still frequently experienced. There are many reasons for such failures. One of those reasons is a lack of a structured approach to making and validating decisions in architecture design. The existing practice of loosely or not documenting design rationale hamper the decision making and validation process. Although architecture frameworks have acknowledged the need for design rationale capture, the practice of systematic design rationalisation is uncommon and an investigation into this area is required.

Chapter 3

Related work in design rationale

Information system design is a process of creating tangibles artefacts to meet the needs of business and organisations. During this process, designers create or design tangible solutions to meet intangible goals. Such creations involve creative thinking, design knowledge as well as the designer’s interpretation and preferences. The nature of the design process is not how things are, as in natural science, but is concerned with how things ought to be with devising artefacts to attain goals. Goals are satisfied by better or worse designs through searching and selecting from a set of alternatives which are not “given” but must be synthesised [141].

Rittel and Webber [134] viewed design as a process of negotiation and deliberation. They suggested that design is a “wicked problem” in which it does not have a well-defined set of potential solutions. Even though the act of design is a logical process, it is subject to how a designer handles the wicked problem. Singley and Carroll [143] suggested five distinct ways of bringing psychological constraints to bear on the design process. They analysed the positive psychological effects a design promotes and the negative effects it mitigates by using this taxonomy of design reasoning.

The choices a designer makes are a result of a design reasoning process. This design reasoning process may differ from designer to designer, and probably would change over time. It is often just intuitively used by designers. In order to understand this design reasoning process, one has to examine how synthesised artefacts satisfy their design goals. This in turn requires relevant design rationale to be made explicit for examination.

In this chapter, we examine the nature of design rationale and how it works for design and maintenance. We summarise the current design rationale methods and discuss their capabilities and shortcomings.

3.1 What is design rationale?

According to the Cambridge dictionary [127], a rationale is a reason or an intention for a particular set of thoughts or actions. When architects and designers make design decisions, what do they consider as a reason or an intention? Should a requirement or a constraint be considered a reason for a design? Or is it some generic justification that allows designers to judge that a design is better than its alternatives? There are many ways to interpret design rationale. A common understanding is using design rationale to explain “why” design artefacts exist.

An interpretation of design rationale depends on the perspectives of the researchers. Moran and Carroll suggested that design rationale are reasons to express the purposes of the designed artefacts with their contextual constraints on realising the purposes [103]. Design rationale are the logical reasons to justify a designed artefact. According to Conklin and Burgess-Yakemovic [24], design rationale can have the characteristics of recording the history of how a design comes about through recording logical reasoning to support future reference. Carroll and Rosson [18] suggested that design rationale can be viewed as psychological claims that are embodied by an artefact whereby design deliberations through evolution can be assessed. Maclean et al. [97] claimed that design rationale can be a description of the design space and used to deliberate design decisions.

Depending on the information need and how that need is to be satisfied, design rationale can exist in many different forms [33]. One way of looking at design rationale is by its level of use:

- No Explicit Rationale - the design reasoning process is not employed in a conscious and systematic way. Instead, it is intuitive to the designer. In this case, design rationale are not captured. When asked to explain the design, designers either remember the rationale from the past or reconstruct the design rationale by deduction [157, 53].
- Informal Rationale - this technique is used by designers to capture design reasoning in an unstructured way such as notes. The choice of which rationale to record is ad-hoc and the level of details of the documentation varies [33, 157].
- Template Based Rationale - design rationale is captured in a systematic manner using pre-defined templates incorporated in the design process. Examples are Architecture Decision Description Template [170] and Views and Beyond [23].
- Argumentation Based Rationale - design decisions are rationalised by using arguments for each alternative. Argumentation-based models focus on how the ideas and

their relations are to be represented [72]. Examples of such models are PHI [99], gIBIS [26] and DRL [94].

- Quantitative Rationale - another type of model is by way of quantifying costs and benefits of design alternatives in the decision making process. Examples of such techniques are CBAM [4] and AHP [152].

As the level of sophistication of design rationale changes, so are the amount of information captured, the complexity of its implementation and the associated costs. It would therefore be important to examine the reasons for capturing design rationale to assess their usefulness.

3.2 Why do we need design rationale?

Researchers in the area of design rationale have argued that there is a need to improve the design decision making process to capture, represent and reuse design rationale lest that they might be lost. Perry and Wolf [119] suggested that as architecture design evolves, the system is increasingly brittle due to two problems: *architectural erosion* and *architectural drift*. Both problems may lead to the violations of the architecture design over time because the underlying rationale is not available to support the architecture design. Bosch suggested that architecture design decisions are crossing-cutting and intertwined, so the design is complex and prone to erroneous interpretations without a first-class representation of design rationale [11]. As such, the design could be violated and the cost of architectural design change could be very high and even prohibitive. Kruchten et al. suggested that architecture knowledge consists of the architecture design and its design rationale [86]. They identified a number of use cases for design rationale in the architecture design process.

The reasons for capturing and using design rationale are twofold: using design rationale to support the design process; and using design rationale to support the maintenance activities. The following is a summary of why design rationale can be useful:

Supporting the Design Process

- Deliberating and Negotiating Design - by making explicit the main decision making elements, Dutoit and Peach suggest that design rationale facilitates negotiations among developers by systematically clarifying the issues and possible options, and evaluating them against a well-defined criteria [33]. Olson et al. have shown that

3.2. Why do we need design rationale?

descriptive categories based on general design rationale schemes can be useful in analysing activities in design meetings [112].

- **Justifying Design Decisions** - a rationale may explicate tacit assumptions, clarify dependencies and constraints, and justify a design decision by reasoning why a choice is made by selecting from amongst the alternatives [53]. There are many reasoning perspectives and hence many reasoning methods for justifying the design decisions, these methods are discussed in the next section.
- **Supporting Tradeoffs Analysis** - a design decision often involves resolving conflicting requirements where they cannot be fully satisfied simultaneously. Therefore, tradeoffs analysis method such as ATAM are used to provide ways to prioritise requirements and obtain a compromised decision [8].
- **Structured Design Process** - design rationale provides a design practice which is structured and accountable compared with the ad-hoc design practice. It provides a pertinent understanding of the context, the users, the tasks, the technologies and the situations in the project space [18]. With design reasoning, designer has a specific focus (so-called turtle-eye's view) to raise issues about a specific artefact in a design [122].
- **Design Verification & Review** - design rationale provides a record of why certain design decisions have been made. This provides a trail for design validation and review in which independent assessment of the design can be supported [159]. Reviewers can also validate the design through inspecting whether design alternatives are considered, design can be traced to requirements and sufficient reasons are provided in a design alternative [14].
- **Notational Support** - the semi-formal notations record the deliberation of design reasoning. They provide uniform notations to represent key rationale elements such as design issues (or questions), design alternatives (options or positions) and arguments. Methods such as gIBIS [26], QOC [97] and DRL [94] each have their own notational support.
- **Communication and Knowledge Transfer** - the knowledge about a system design needs to be shared amongst different interested parties. Information such as the current state of the design, any unresolved issues and what to do next can be captured by the design document and the design rationale. Conklin and Burgess-Yakemovic found that design rationale is most useful in knowledge transfer in an industrial case at NCR when some designers leave the design team and the knowledge needs to be transferred to new members of the team [24].

Supporting the Maintenance Activities

- **Retaining Knowledge** - in the design process, assumptions are made, constraints are considered and priority is assigned. These elements shape the design decisions. Often the only documentation available to the maintainers are the design specifications. If the designers who maintain and enhance a system are not the same person who created the system, which is often the case, the maintainer would have to second-guess these intangible rationales. Failure to retain such knowledge might cause violations and inconsistencies in a system [33]. It was found in a survey that most designers cannot remember the reasons of their design [158]. If the maintainers are not the original designer, then design rationale as a source of knowledge would be very useful [157]. Such retained knowledge could save costs because engineer's time to recover lost rationale is reduced [25].
- **Capturing Design Alternatives** - as Parnas and Clements pointed out that the ideal design document which consists of all the details is impractical [114]. Therefore, recording key design rationale information such as design alternatives and why they have been rejected can answer many questions about the design.
- **Understanding Decision Dependency** - design decisions sometimes are inter-twined and cut across a number of issues. Changing a decision may trigger a series of side effects to the other parts of the system through the ripple effect [59]. Designers could potentially omit necessary changes caused by the inter-dependent decisions in the design space. Some design rationale methods could address this issue by providing linkages between dependent decisions to facilitate traceability [97, 162].
- **Improving Design Understanding** - in one of the experiments, Brathall et al. showed that a group of designers equipped with the design rationale can work faster and identify more changes that are required in a system maintenance exercise than a control group without the design rationale [12]. Burge showed that using the design rationale, those designers with moderate and some experience in Java perform their work better than the designers without the design rationale [14].
- **Predicting Change Impact** - design rationale could be used to assist the maintainer to predict which part of the system is subject to change. This kind of predictive and diagnostic capabilities provide quantitative analysis to assist decision makers on predicting the change impact in a system [162].
- **Providing Traceability** - when the design rationale is associated with the design artefacts, maintainers would be able to trace how design artefacts satisfy requirements in a system with some reasoning support [129].

- Detecting Reasoning Inconsistencies of Design Decisions - in one study, a tool called SEURAT was built to verify the consistency and completeness of the design rationalisation [14]. This could help maintainers trace design rationale and identify any reasoning gaps in the design for maintenance purposes.

Even though design rationale have many uses in supporting the design and maintenance processes, capturing design rationale is still uncommon. It was suggested that the lack of short-term payoffs might be one of the causes [24]. Another reason was that the tools and the methods used in design rationale capture are not closely aligned with existing design processes therefore resulting in higher cost of design rationale capture [24]. In the following sections, we will examine some of the design rationale methods, the tools availability and their potential benefits and drawbacks.

3.3 Existing methods for capturing and representing design rationale

A simple explanation of design rationale is the explicit representation of the design reasoning. There are other different aspects of design reasoning. A design reason could be an intention to motivate the creation of a design artefact. It could also be a constraint or an assumption that influence a design artefact, a tradeoff between requirements, a judgement to select from a number of design options and an argument for or against a design proposition.

There are different approaches to design reasoning. One approach is by way of argumentation. The basic argumentation-based representation is to use nodes and links to represent knowledge and relationships. It dates back to Toulmin's argumentation representation using *datums*, *claims*, *warrants*, *backings* and *rebuttals* [168]. In Toulmin's schema, examples of a node are a *datum* or a *claim*, which represent an observation and a conclusion respectively. A link type such as "so" can be used to connect the nodes to show the inductive relationship. Since Toulmin, many similar argumentation-based approaches such as Issue-Based Information System (IBIS) [87] and Design Rationale Language (DRL) [92] have been invented. They fundamentally show the issue, the argument and the resolution of design argumentation.

A different approach to capturing design rationale is to use template-based methodologies. These methods make use of standard templates which is incorporated into the design process to facilitate design rationale capture. Contrary to argumentation-based methods, practitioners using the template-based methods do not construct argumentation diagrams

for deliberation but instead they capture the results of the reasoning. This approach is oriented towards the practical implementation of design rationale in the industry. Examples of this approach are Architecture Decision Description Template [170] and Views and Beyond [23].

Argumentation-based reasoning is sometimes insufficient to justify a decision from amongst the alternatives. A design decision could be influenced by many factors and requirements. Some of them have higher importance or priority than others. Quantitative methods were proposed to rank the priorities, the costs and the benefits in the decision making process. Examples of this approach are Cost Benefits Analysis Method (CBAM) [4] and ArchDesigner [2].

In this section, we will describe a number of design rationale methods, what they intend to achieve and how they work.

3.3.1 Issue-Based Information System (IBIS) and its variants

The Issue-Based Information System (IBIS) is a method for structuring and documenting design rationale [87]. Its approach is based on issue deliberation. Similar approach has been used in other design areas such as building architectural design and town planning. The key design aspect of IBIS is the articulation of questions. Each question or *issue* is followed by *positions* or *answers* that respond to the issue. A *position* can be supported or objected by *arguments*. A position derives or has ramifications to an artefact. Figure 3.1 is a diagram which shows such relationships. The various issue deliberations are connected by relationships such as “similar-to”, “replaces”, “temporal successor of” etc.

From 1970 to 1980, many attempts have been made to use IBIS but none of these systems got past the pilot project stage [43]. It was concluded that two types of information was omitted from IBIS: inter-related issues such as sub-issue cannot be represented by IBIS as dependency and hence it is difficult to model some of the deliberations; the pros and cons of alternative answers are not deliberated.

In order to overcome these limitations, McCall developed the Procedural Hierarchy of Issues (PHI) to document design rationale [100]. PHI uses a broader definition of the concept issues and it uses a new principle for linking issues together. In IBIS, issues denote a design question that is deliberated but PHI counts issue as every design questions, deliberated or not. PHI simplifies the inter-issue relationship in IBIS by a general *serve* relationship. Issue A *serves* issue B if and only if the resolution of A influences the resolution of B. As such, the dominant type of *serve* relationship in PHI is sub-issue. PHI is therefore a simple quasi-hierarchical structure connecting issues only by the *serve*

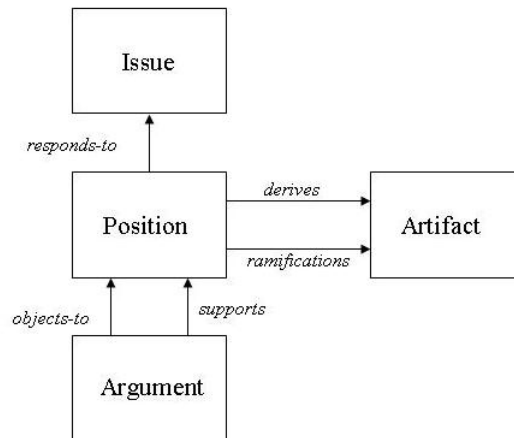


Figure 3.1: IBIS Design Rationale Model

relationships as shown in Figure 3.2.

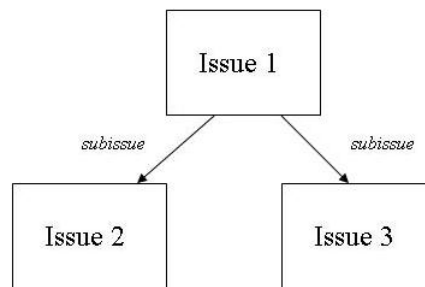


Figure 3.2: PHI Issues Hierarchy

Similar to IBIS, PHI provides the dependency relationships between issue resolutions and it records the pros and cons of alternative answers. PHI has only had limited success in its adoption for industrial use [132].

gIBIS enhances IBIS by allowing “other” node type for incorporating external materials [26]. It supports aggregation of the issue-position-argument (IPA) trees into a compound IPA node. A graphical representation of gIBIS was implemented at MCC. The graphical layout of gIBIS composes of a global network view, a local network view, an index view showing an hierarchy of issues and arguments, a node view showing the contents and attributes of the selected node and a control panel view.

The Representation and Maintenance of Process knowledge (REMAP) system is based on the IBIS method [128]. It extends the basic IBIS constructs of *issue*, *position* and *argument* with *requirements*, *constraints* and *design objects*. This extension provides a

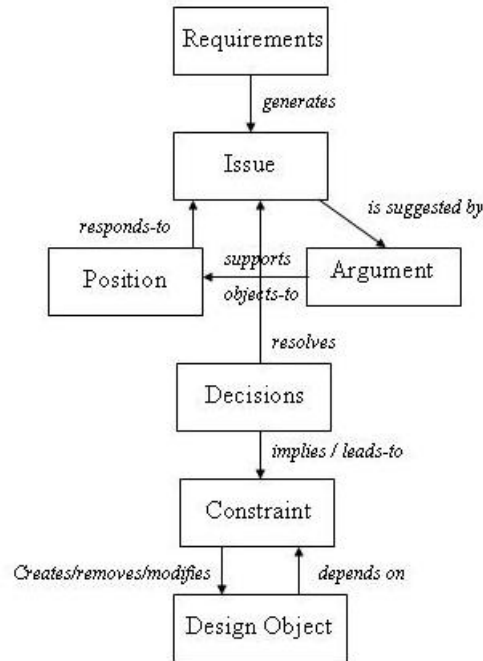


Figure 3.3: REMAP Model

linkage between requirements and design objects to design rationale (see Figure 3.3). It can capture the history about design decisions in the development-cycle as process knowledge. Issues are raised and resolved during the requirement phase when design objects are created. This issue deliberation process is supported by a graphical user interface tool.

3.3.2 Questions, Options and Criteria (QOC)

QOC is a semiformal notation which represents the design space analysis of an artefact. It explains why a design artefact is chosen from the space of possibilities. The main elements of QOC are the *questions* that identify the key design issues, the *options* that provide possible answers to the questions, and the *criteria* for assessing and comparing design options [97].

The QOC representation is accompanied by Design Space Analysis. MacLean et al. argue that designers are naturally capable of analysing and reasoning with QOC when designing artefacts in the design space [97]. Thus design rationale analysis are simply co-products of the design. Figure 3.4 is a QOC representation that shows an example of a screen object design. The *question* is whether the screen object should be wide or narrow. If the object is wide, it uses up screen real-estate but is easy to hit with a mouse. If it is

narrow, it saves screen real-estate but is difficult to hit with a mouse. Since the rationale is based on argument and not proof, MacLean et al. argue that the elements in QOC should be used to justify a design even though they may be subject to further arguments [97]. As such, QOC can be expanded into any arbitrary level of elaboration. Designers using QOC should select areas where these arguments serve the purpose of contentious design but not expand into every possible detail because it is not useful to do so.

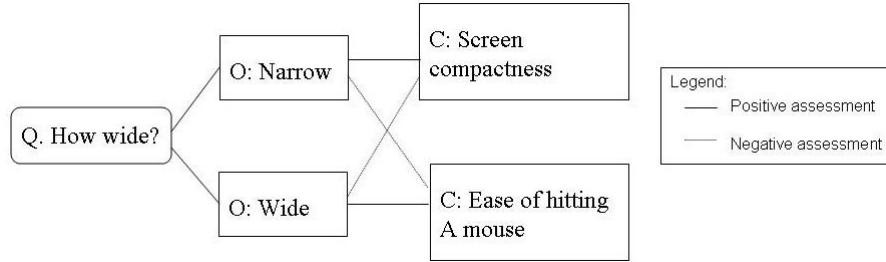


Figure 3.4: An Example of QOC Design Rationale

QOC supports the development of a space of alternatives. It enables designers to consider criteria which could dictate viability of options. Unlike IBIS, PHI and REMAP which capture the history of design deliberation, QOC focuses on design options.

Dutoit and Peach proposed the Rationale-based Use Case Specification method which combines use case specification to the QOC method of argumentation [34]. The argumentation for the use cases which comprised of functional and non-functional requirements are then captured in an enhanced QOC model. This method provides a better integration between requirement specification, design specification and the design rationale.

3.3.3 Design Rationale Language (DRL)

DRL records design rationale by describing how an artefact serves or satisfies expected functionalities. DRL is an expressive language which represents the qualitative elements in the reasoning spaces around decisions [94]. In DRL, the possible design options are contained in the *alternative space*, and the arguments to support or contradict a design is contained in the *argument space*, each design possibility is evaluated and the results are contained in the *evaluation space*, the evaluation is performed according to certain criteria which is contained in the *criteria space*, the issues which are made explicit and containing the alternatives, evaluations and criteria are contained in the *issue space*. The fundamental object types in DRL are *goal*, *question*, *claim* and *alternative*. The structure of a decision graph using these elements is shown in Figure 3.5.

A *goal* represents the *criteria* that needs to be satisfied. An *alternative* represents

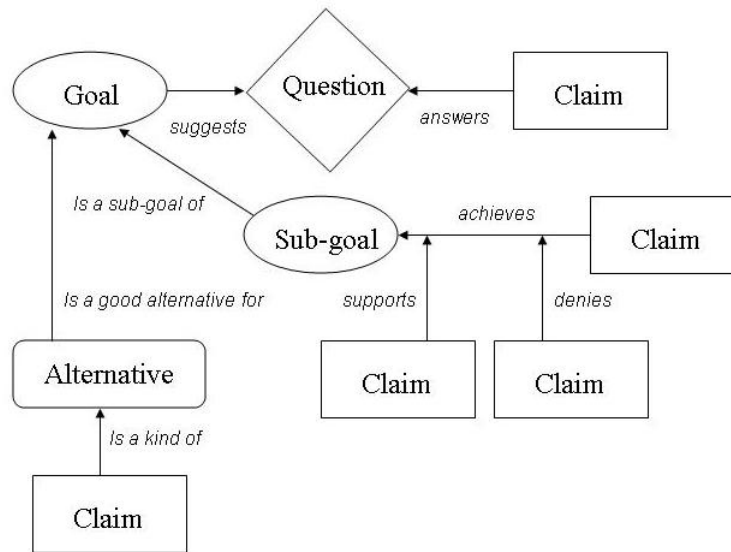


Figure 3.5: A DRL Decision Graph

an option that is being considered. Its relationship with the *goal* is whether it is a good alternative. A *question* is an issue which arises out of a *goal* to be answered. A *claim* is an answer to the *question*. A *claim* may or may not achieve a *goal*.

The argumentation of a design in DRL is constructed from these basic elements. They are in turn grouped into different design and argumentation spaces. DRL is implemented by a system called SIBYL [91]. User can edit and browse the argument space to explore the claims contained within.

3.3.4 Software Engineering Using Design RATIONale (SeuRAT)

Burge developed a system, Software Engineering Using Design RATIONale (SeuRAT), to support the use of design rationale during software maintenance by associating rationale with the code and by performing a series of inferences over the rationale to ensure design rationale consistency and completeness [14].

A design rationale representation system, RATSpeak, is created to support SeuRAT. RATSpeak design rationale representation is a modification of DRL. Figure 3.6 shows the elements represented in RATSpeak. A key difference between RATSpeak and DRL is that RATSpeak represents the concept of requirements instead of goals. Requirements compose of both functional and non-functional requirements. A decision problem is mapped to the requirements to support the argument for solution alternatives. The argument ontology is a hierarchy of common argument types that serve as types of claims. They provide a

common vocabulary required for inferencing.

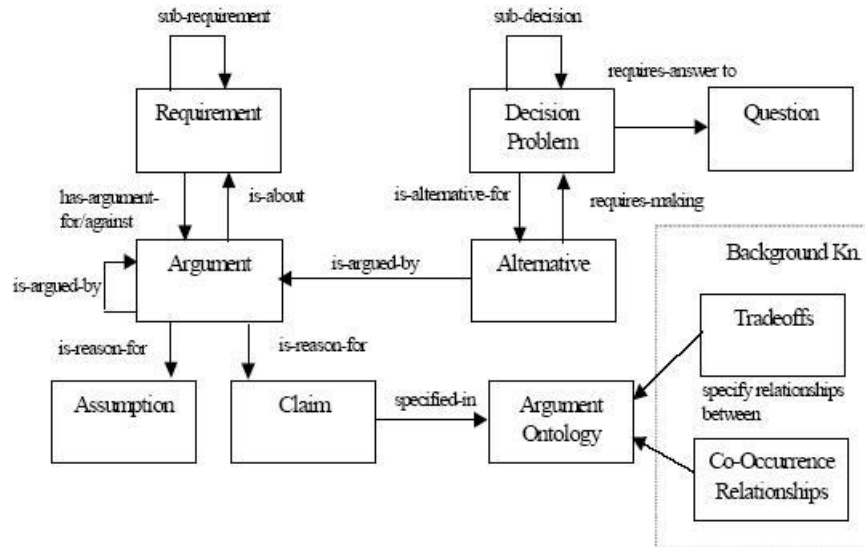


Figure 3.6: A RATSpeak Design Rationale Model

SeuRAT supports syntactic and semantic rationale inferences. Syntactic inferences are concerned with the validity of the structure of the rationale graph. For instance, if there is an alternative (i.e. potential solution) for a decision (i.e. problem), then the alternative must have a supporting argument (i.e. justification). Semantic inferences require examining the content of the rationale. For instance, identify selected alternative which is not as well supported as another alternative by comparing their rationales.

Using the captured rationale, SeuRAT enables maintainers to retrieve and check rationale for maintenance activities. Burge found that SeuRAT provided better results in test subjects who are not experts in Java than their respective control group in identifying problems and completing tasks [14].

3.3.5 Architecture Decision Description Template (ADDT)

Tyree and Akerman have proposed a pragmatic approach to capturing design rationale [170]. Instead of focusing on the deliberation of the design rationale and the representation model, they provide a template to capture decision rationale.

The decision description template contains a number of key elements, they are summarised below:

- Issue - describe the architecture issue being addressed

3.3. Existing methods for capturing and representing design rationale

- Decision - state the final architecture decision (i.e. position being selected)
- Status - decision status are pending, decided or approved
- Grouping - a grouping of architecture decisions by types such as integration, presentation and data. This ontology is used to help organise the set of decisions
- Assumptions - underlying assumptions which affect the decision
- Constraints - constraints that the decision might pose to the environment
- Positions - viable options to address the issue
- Argument - the reasons to support the position
- Implications - the implications of a decision might introduce a need to have other decisions, introduce new requirements, pose new constraints etc.
- Related Decisions - relationships which are interdependent
- Related Requirements - objectives or requirements that are related to a decision
- Related artefacts - related architecture, design and scope documents
- Related principles - agreed set of design principles which the decision is aligned with
- Notes - ideas that have been socialised by the team as additional information

The information captured in this template is comprehensive. They provide knowledge support during and after the architecture design process. Tyree and Akerman argue that this information is what needs to be socialised with the rest of the organisation [170]. There are no specific reasoning and deliberation methods in ADDT to guide the design reasoning process.

In the implementation of ADDT, they found that this method has several advantages [170]. It helped them to clearly identify systems which are affected by architectural change. It also conveyed risks and issues that arose out of the decisions. Team members could trace, read and understand the architecture decisions. By explicitly focusing on architecture decisions, they found that they could convey change more clearly. The reasons, options and implications of the solutions are more transparent and traceable.

3.3.6 Views and Beyond (V&B)

Views and Beyond is a collection of methods proposed by Clements et al. to document software architecture [23]. It proposes a number of view types to represent software architecture: Module Viewtype, Component-and-Connector Viewtype and the Allocation Viewtype. Each viewtype gives a different perspective of the structure of the system. For instance, the module viewtype has several representation styles such as decomposition of code and layering of modules.

As well as documenting what the architecture is, they argue that it is important to document why the architecture is what it is. Many decisions are made in a design and not all decisions need to be justified. A number of principles to guide decision documentation are suggested:

- Design team have spent significant time evaluating the options for the decision
- Decision is critical to the achievement of the requirement/goal
- Decision requires considerations of non-trivial background information
- Issues in a non trivial decision
- Decision has a widespread implication to the rest of the architecture design and will be difficult to undo
- It is more cost-effective to document the decision now rather than later

Similar to ADDT, they take a pragmatic approach to documenting design rationale. Instead of documenting the deliberation relationship, the following information, which is centric to a decision, is documented:

- Decision - a summary of the decision
- Constraints - key constraints that rule out possibilities
- Alternatives - options that have been considered and the reasons of ruling them out
- Effects - the implications and ramifications of the decision
- Evidence - any confirmation that the decision was a good one

This list of design rationales can be easily mapped to the ADDT template. ADDT provides a more comprehensive classification to assist designers to capture all the design rationale. The V&B template is more general and easier to implement.

3.3.7 Quantitative reasoning methods

Architecture design involves the selection of an optimal solution by eliminating inferior design alternatives. One way to facilitate such selection process is to quantify the design options in some ways. There are a number of methods which use quantitative analysis as a mean in the selection process. Architecture Tradeoff Analysis Method (ATAM) is a method to use tradeoffs to justify architecture design decisions [8]. ATAM method does the following:

- Identify the quality attributes as goals in a decision
- Assign relative priority to the goals
- Identify the architecture approaches (i.e. options)
- Create an utility tree to list the quality attributes, their attribute refinements and the scenarios
- Carry out an analysis of the alternative approaches by investigating the sensitivity points (i.e. which quality attributes are affected), the tradeoff points (i.e. compromise required in order to satisfy the quality attributes), risks and non-risks

Cost Benefit Analysis Method (CBAM) considers the costs and benefits which are associated with the decisions. The method calculates the expected return of an architecture strategy by weighing up the benefits and the costs. The benefits and the costs are computed by calculating the weighted benefits and the weighted costs of all related scenarios. This weighted method provides a quantifiable justification for making decisions [4].

ArchDesigner involves stakeholders to prioritise the quality attributes. Architects would elicit the impact of different architecture design options on the quality attributes. Then using the Analytical Hierarchy Process (AHP) to compute the value scores, they select the optimal design alternative to satisfy the goal [2].

3.4 Other related studies

A number of studies in the other areas of software engineering are relevant to design rationale. Design rationalisation is an integral part of the the design process which uses requirements to drive decision making. Therefore, some studies in the requirements engineering and non-functional requirements domains are relevant to this work. When design

rationale is captured, its relationship with the requirements and the design objects is important. It is not useful to the designers and maintainers if the knowledge cannot be effectively traced. The traceability of design rationale to requirements and design objects is essential in design rationale applications.

3.4.1 Requirements engineering

There were a number of studies on how to elicit requirements and decompose design [110, 56, 130]. Some of them combine rationale and scenarios to refine requirements during the elicitation process [126, 61].

Architecture Frame [130] uses problem frame for decomposing and recomposing design by using architecture styles in the solutions space to guide the analysis of the problem space. The KAOS [28] method uses a meta-model to analyse goals, actions, agents, entities and events for acquiring and modelling requirements. The elements considered in this method act as inputs in the decision process. Nuseibeh [109] argues that architectural positions are taken during the requirements specification process and the crosscutting of requirements therefore implicitly drives the architecture design. This is especially relevant in the case of non-functional requirements because of their interrelated nature. Chung and Mylopoulos represent non-functional requirements as soft-goals [20, 104]. The soft-goals are satisfied or operationalised in the model through a decomposition process.

Ali-Babar et al. provided a data model to capture design rationale and related artefacts called The Data Model for Software Architecture Knowledge (DAMASK) [3]. DAMASK captures architecture design patterns, scenarios, design options, design decisions and design rationale.

These methods generally use key inputs such as goals and soft-goals to systematically specify requirements and create design objects. They are relevant because they have identified different basic elements that are essential in the design process. Some of these elements are treated as motivational reasons and business drivers in our work.

3.4.2 Requirements traceability

The support for requirements traceability has long been recognised as important in the software development life-cycle. A survey of a number of systems by Ramesh and Jarke [129] indicates that requirements, design and implementation should be traceable. It is noted by Han [58] that traceability “provides critical support for system development and

evolution”. The IEEE standards recommend that requirements should be allocated, or traced, to software and hardware items [67, 68]. During the software development life-cycle, architects and designers typically have available to them business requirements, functional requirements, architecture design specifications, detailed design specifications, and traceability matrix. A means to relate these pieces of information together helps the designers maintain the system effectively and accurately. It can lead to better quality assurance, change management and software maintenance [149]. There are different aspects of traceability in the development life-cycle: (a) tracing requirements to design; (b) tracing requirements to source code and test cases; (c) tracing requirements and design to design rationale; (d) tracing evolution of requirements and design. Example methods to support requirements traceability are [120, 129], and an example of traceability automation is [38].

Requirements traceability is the ability to describe and follow the life of requirements, in both a forward and backward direction. Gotel and Finklestein [51] distinguish two types of traceability: *pre-requirements specification* (Pre-RS traceability) and *post-requirements specification* (Post-RS traceability). They argue that wider informational requirements are necessary to address the needs of the stakeholders. This is an argument for representing contextual information to explain requirements and design. An example of such contextual information is to support tracing requirements to stakeholders to support the analysis of knowledge contribution and use [52] .

Pinheiro and Goguen [120, 121] proposed TOOR, a tool to trace object-oriented requirement to design documents, specifications, code and other artefacts through user-definable relations. It supports the tracing of objects through their evolutions. The method uses regular expression to provide a context for the selected information. This allows pattern matching and restriction of the scope of the trace to certain parts of the project.

Huges and Martin [66] suggested that traceability needs to involve requirements and designs. They explained that considerations must be given to relevant project and design constraints for tracing *why* designs are created. They explained that information might be lost if the information is captured at the end of the project, therefore the traceability process must be an interactive part of the design process.

Domges and Pohl [32] suggested that the adaptation of trace capture should be tailored to project-specific needs for achieving a positive cost-benefit ratio. They provided an architecture framework to support this course. Using this framework, project manager would define project-specific trace definitions and application engineer would capture traces based on trace-definitions. Spanoudakis et al. [149] proposed a rule-based approach that supports automatic generation of traceability relations between textual documents

that specify uses cases and requirement statements. The approach employs rules to match keywords in the documents and the object models during the generation process. Another approach of traceability is to connect architecture decision to requirements using design decision tree [136]. Design decision tree reflects the dependency between the decisions to support traceability of the design to requirements.

Egyed [38, 39] has proposed an approach to generate trace dependencies between model elements and source code using test scenarios and execution foot-prints. This approach automates traceability at the source code level to discover common requirements and potential conflicts.

Some traceability approaches incorporate the use of design rationale in a limited way. Haumer et al. [60] suggested that the design process needs to be extended to capture and trace the decision making process through artefacts such as video, speech and graphics. Since such information is unstructured, making use of it can be challenging. A reference model for traceability was proposed by Ramesh and Jarke [129]. It adopts a model involving four traceability link types. The first group of two traceability links are product-related. They describe properties and relationships of design objects in which high-level objects are goals or constraints that are *satisfied* by low-level objects, and low-level objects are *dependent* on each other by way of links to depict relationships such as *part-of*, *supports* etc. The second group of two traceability links are process-related. A link type connects design object that *evolves* from one version to another and its evolution is supported by the link type *rationales*. The rationale and evolution link types are introduced to capture the rationale for evolving design elements. The rationale link type (see Figure 3.7) is intended to allow users to represent the rationale behind the objects or document the justifications behind evolutionary steps. The limitation of using rationale for design evolution in this approach is that other types of design reasoning are omitted.

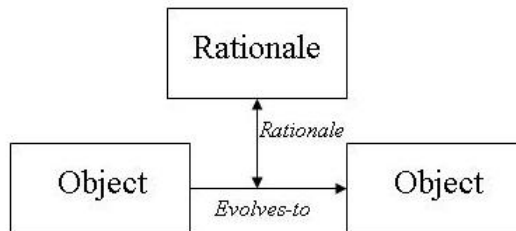


Figure 3.7: A Design Rationale Traceability Model

Some of the traceability methods such as [120] and [38] try to automate the traceability relationships between requirements, design and source code. A key perspective of these methods is to trace the implementation of related objects based on how a system is realised.

There is an implicit assumption that such traceability can provide the explanatory power to help designers understand the system. However, design reasoning is missing and needs to be reconstructed even though source code or design objects can be traced back to requirements. Additionally, implicit assumptions and constraints cannot be traced by such methods. As such, the traceability of design rationale requires further investigation. A more detailed discussion on design rationale traceability is in Chapter 9.

3.5 How well design rationale methods work?

Although researchers have different ideas of what and how to represent design rationale, they basically agree on the importance of retaining design rationale, and it is concurred by practising software architects [157]. With such common view, why then are design rationale methods not widely adopted by the software industry?

Shipman III and McCall suggested that neither the argumentation nor the communication perspective of argumentation-based design rationale has been generally successful in practice [139]. From the argumentation perspective, the issue has been the ineffective capture of rationale. From the communication perspective, design rationale cannot be retrieved effectively. Regli et al. [132] have a similar argument that design rationale must have three qualities: ease of input, effective view and activeness. That means the rationale capture process must least interfere with the natural progression of design activities.

Gruber and Russell [55] examined a set of empirical studies of people requesting, communicating and using information associated with design rationale. They found two basic uses for information explicitly related to decisions. First, decisions serve as loci for considering alternatives and linking dependent elements. Second, a designer would evaluate a design or a sub-design to predict the potential impacts on criteria. For design rationale to be successful, it must be captured as a by-product of the design process using software tools to make the effort worthwhile.

Buckingham Shum and Hammond [13] examined two claims made by argumentation-based design rationale approaches: (a) expressing design rationale as argumentation is useful and (b) designers can use such notations. However, they found no evidence to support argumentation-based schemas being useful as a record of previous argumentation. Also, they concluded that argumentation-based schemas assume that designers would be able to directly express rationale in terms of the argumentation-based structures, which could not be proven. They also concluded that there was a tendency for some of these semi-formal methods to move to an informal textual rationale. In another study, Shum [140] investigated the cognitive dimensions of design rationale. It was found that problem

arises when the design rationale system forces users to encode ideas before they are ready to make the decision thereby causing a premature commitment. Argumentation-based tools require users to either encode the deliberation when the ideas are still premature causing subsequent changes to be made. Otherwise, the users would have to recall the deliberation process and then record the full argumentation after a final decision has been made. Neither way is convenient to practitioners.

In an empirical study, it was suggested that the design rationale constructed using the QOC method is insufficient [76]. A number of design rationale issues were identified: (a) not all information was captured because analysts had not reported the facts; (b) QOC lacked weighted assessments since there is no quantitative analysis to indicate why a solution is better than its alternatives; (c) users of QOC have misconceptions of the captured information. In conclusion, it was found that some designers extensively use design rationale but less than half of the design rationale questions could be answered by the QOC-based documents.

Herbsleb and Kuwana [62] pointed out that the *why* question may often be used to establish the context of the design when it is unclear. However, in the current practice the *why* question are seldom asked because they generally cannot be answered using the current tools.

Although design rational methods have laid the foundation for design reasoning, most of them still have issues from a usability perspective. Learning from these issues, a set of criteria for a successful design rationale implementation are summarised below:

- Effective Capture of Design Rationale - the argumentation-based models capture both the reasoning and the design rationale argumentation structure. The argumentation structure is time-consuming and difficult to trace. Therefore, it should be simplified without losing key design rationale information.
- Effective Communication of Design Rationale - designers want to know the issues, the justifications, the potential alternatives and what design elements a decision affects. The necessity to replay the deliberation process as suggested by the argumentation-based models is thus reduced. The issues in argumentation-based models are the over representation of deliberation and under representation of its relationship with design artefacts.
- Design Artefact Focus - requirements and design objects specifications are used to support system evaluation and maintenance. Therefore, design rationale must explain design artefacts contained in these specifications in order to be useful.

3.5. How well design rationale methods work?

- Traceability and Impact Analysis - a primary use of design rationale is to help designers understand the justifications and the issues of a design in order to perform maintenance activities. One such activity is the change impact analysis. Design rationale methods must support analysis of the ripple effects using traceable design rationale to explain the design dependency.
- Comprehensive Design Rationale - design rationale is composed of many types of information. Reasoning can be based on argumentation or it can be based on quantitative analysis. Therefore, design rationale methods must be comprehensive and flexible to capture these different types of reasoning.
- Common Tool Support - software designers always face a very tight development schedule. Using a different tool to capture design rationale would increase the overhead of documentation and segregate the design knowledge. Design rationale should be captured as a by-product, and in the same way and by the same tool as requirements and design objects.

In this thesis, our objective is to capture design rationale in a convenient way to support software architecture design and maintenance. We aim to incorporate the design rationale method with current architecture design practices in the software industry. From this perspective, we analyse the usability of existing design rationale methods (see Table 3.1) using the criteria set out above. Each method is rated according to whether it satisfies the criteria in full, in part or not.

Table 3.1: An Analysis of the Usability Features of Design Rationale Methods

	gIBIS	PHI	REMAP	QOC	DRL	SeuRAT	ADDT	V&B
Effective capture of design rationale	No	No	No	No	No	No	Yes	Yes
Effective communication of design rationale	No	No	No	No	No	No	Yes	Yes
Design Artefact Focus	Part	Part	Yes	No	Part	Yes	Part	Yes
Traceability and Impact Analysis	No	No	Part	No	No	Part	Part	No
Comprehensive Design Rationale	No	No	No	No	No	No	Part	Part
Common Tool Support	No	No	No	No	No	No	Yes	Yes

ADDT and V&B's aim are to provide design rationale methods for practitioners. Therefore they have a more pragmatic approach than the other methods. In terms of the design rationale capture and communication, their structure is less formal than the other methods. Because of that, they could be implemented using different types of software tools and can be adopted by an organisation easily. However, the limited graphical representation in these two methods restricts their ability to provide rationale computation and traceability.

Other methods are research-oriented and there are issues in their effectiveness to capture and communicate design rationale. A design rationale method that supports comprehensive rationale computing (e.g. for rationale traceability and inference) might be in conflict with the ease of design rationale capture and communication. One of our objectives is to find a compromise where sufficient design rationale could be captured easily, and its payoffs outweigh the costs.

3.6 Summary

This chapter reports the state of the research in design rationale and its application in architecture design. We have reviewed various design rationale methods and how researchers propose to use them to deliberate design and support maintenance. We have examined past work on argumentation-based design rationale methods as well as other kinds of design rationale methods. We have discussed their purposes for deliberating design, capturing knowledge, supporting maintenance and inferencing design.

We have reviewed the applications of these design rationale methods and have found some issues. The key issues are the cost of capturing design rationale and the difficulties in communicating such knowledge. A number of other usability issues are also summarised in this chapter. They give us a guide to develop a new method.

Chapter 4

Research methodology and validation

The discipline of software engineering is concerned with all aspects of software production. It provides a systematic and organised approach to guide software development and implementation so that it is effective in delivering high-quality software [146].

There are two general approaches to systematically building up software engineering knowledge. Both of these approaches are useful and complementary. The first approach is described as the natural characterisation of maturing software technology [131]. Software technologies can be invented through a process of asking basic research problems, formulating ideas and concepts, developing and generalising those ideas through to their commercialisation [138]. In the initial stages of problem identification and concept formulation, it is relatively difficult to prove that a concept can eventually be practical and useful. This is because a technology depends heavily on the context of the environment where they are applied. Although some software technologies referred to as *research-in-the-small* can be tested in the laboratory, many software engineering techniques rely heavily on real-life situations which are not always easy to verify in an experimental setting [41].

A second approach is to apply software engineering processes through practice and refinement in real-life projects. In other words, the prime expectation is to improve quality and productivity at an organisation level. A typical example is the application of quality systems such as Integrated Capability Maturity Model (CMMI) [1, 16] or Six Sigma Total Quality Management [144] in an organisation. The CMMI process specifies that software development organisations can continuously monitor and improve their software development process by capturing process and product measures quantitatively, set performance objectives based on these measurements and optimise the performance. An organisation

employing such an approach would select methodologies and technologies which initially appear to be suitable for their environment, and continuously improve and fine tune them to improve their software engineering practices. Although this approach is scientific in its own way, most often than not the knowledge or insights gained would remain within the organisation and the results are not reported.

When new ideas are created, the arguments for their early stage development are usually supported by persuasion [138]. Examples of such models include [46, 119]. As time passes, continuous refinement and improvement will prove or disprove their validity. The following sub-sections describe some software engineering research methods. Based on these research methods, the appropriate validation techniques are selected for this research.

4.1 Software engineering research methods

Software engineering is still a maturing discipline. Therefore, it is not surprising that many researchers are still debating on how software engineering research should be conducted (e.g. [41, 166, 125, 172]). As reported by Adrion in [167] and by Glass in [49], software engineering research problems have different characteristics and thus different research methods are employed accordingly:

- Scientific Method - Scientists develop a theory to explain an observed phenomenon. They propose a model or a theory of behaviour, and validate the hypotheses of the model or theory through experimentation.
- Engineering Method - Engineers observe existing solutions and propose new improvements. They measure or analyse the improvements until no further improvements are possible.
- Empirical Method - Researchers observe a phenomenon and put forward some hypotheses. They then collect data to test the model using statistical methods or case studies. These data and their analysis are used to support or refute a hypothesis.
- Analytical Method - Researchers propose a formal theory or a set of axioms, then derive results from the theory and if possible compare the results with empirical observations.

Part of the problem with software engineering research methodology is that the boundaries of the field is not clearly defined. As a result, methodological conflicts arise when

there is a cross boundary situation. For instance, a situation where engineering and management issues are to be considered together, then what research method should be used?

Additionally, the maturation of software technology over time [138] adds another dimension to software engineering research. Different kinds of problems require different research paradigms. Different research settings also post different research questions. Hence, the methods and techniques used would vary accordingly. A summary of these research settings are listed:

- Feasibility - Is there a X, and what is it? Is it possible to accomplish X at all?
- Characterisation - What are the characteristics of X? What do we mean by X? What are the varieties of X and how are they related?
- Methods/Means - How to accomplish X? What is a better way to accomplish X? How can I automate doing X?
- Generalisation - Is X always true of Y? Given X, what will Y be?
- Selection - How do I decide between X and Y?

Given different research settings, the answers that are sought would be quite different in nature. The techniques that are required to validate those answers would also vary [138]. A summary of these techniques are listed below:

- Persuasion - use arguments to support the ideas that have been put forward by going through an example.
- Implementation - use a prototype or a case study to demonstrate the idea.
- Evaluation - compare different objects of research using a check-list of criteria. The comparison of measurements forms the basis of the evaluation. This approach can use expert-opinion as a basis for evaluation.
- Analysis - analyse the facts and the consequences through derivation and proof of a formal model, or use empirical methods such as statistical methods to carry out analysis in a controlled situation.
- Experience - a subjective evaluation based on personal experience and observations about the use of the result in actual practice.

Since case study research are often validated using anecdotal evidence which can be subjective, careful justifications of the evidence would be necessary. Yin described six

different sources of evidence that can be used in case studies [174]: *documentation, archival record, interviews, direct observations, participant observation* and *physical artifacts*. Each type of evidence represents some observations, the convergence of multiple evidences would lead to the conclusions and the facts. Other researchers use expert opinions to validate software engineering process [9], conduct review process [36], analyse accuracy of several methods of estimating project effort [81].

In summary, researchers should identify the nature of the problem so that it is founded on a well-defined research setting(s). The research settings help to determine what research methods to use. Appropriate validation techniques could then be selected to validate the results.

4.2 The chosen research and validation methods

This research has a number of objectives, each of them has one or more corresponding questions that need to be answered. Different research methods have been used to address the questions, and the research results have been validated by different validation techniques. Table 4.1 is a summary of the research problems and the corresponding research methods that have been used.

Table 4.1: Research Questions and Research Methods

Research Questions	Research Settings	Research Method	Validation Techniques
1. Is design rationale important in architecture development?	Feasibility	Empirical	Evaluation
2. Is design rationale important in system maintenance?			
3. What is the current state of practice of design rationale in the software industry?			
4. How to improve the representation of design rationale in architecture development?	Characterisation / Methods / Means	Analytical / Empirical	Evaluation
5. How to implement traceability between requirements, architecture design elements and design rationale?	Methods / Means	Analytical	Persuasion and Implementation
6. How to estimate change impacts using architecture design elements and design rationale?	Methods / Means	Analytical	Persuasion and Implementation

A summary of the evidence collected in this thesis to support the research questions are shown in Table 4.2:

Research questions 1 to 3 are used to ascertain that design rationale is important in the design and maintenance phases of development. This is carried out empirically and validated by collecting statistics from a survey.

Research question 4 investigates the representation of design rationale in architec-

Table 4.2: Case Study Verification

Research Questions	Evidence Collected
1. Is design rationale important in architecture development?	Statistical Results
2. Is design rationale important in system maintenance?	
3. What is the current state of practice of design rationale in the software industry?	
4. How to improve the representation of design rationale in architecture development?	Evaluation Results Expert Opinion
5. How to implement traceability between requirements, architecture design elements and design rationale?	Demonstration / Implementation
6. How to estimate change impacts using architecture design elements and design rationale?	Demonstration / Implementation

ture design. The resulting method characterises the relationship between design elements and design rationales. A proof-of-concept tool set has been developed to accompany the method and it overcomes the usability issues identified in Table 3.1. Using an empirical study, this new method is compared with using the traditional design specifications to examine their effectiveness in design reasoning. Domain experts are asked to evaluate the two methods to determine whether design rationale could help them understand and reason with the system design. The case study used in this research is an Electronic Fund Transfer System (EFT) which processes inter-bank fund transfers for The People’s Bank of China in Guangzhou (PBC-GZ). The system was in production between 1998 and May 2006. It was originally designed by the author who led a team of architects and designers using traditional waterfall software development methodology. A rationale-based model is reconstructed for the EFT system to capture the missing design rationale. Its validation is reported in Chapter 7.

For questions 5 and 6, we resolve the problems using traceability and Bayesian Belief Networks methods. Based on the case study examples, we demonstrate the applications in both research questions.

The architecture design rationale models presented in this thesis are early conceptual models to integrate design rationale into architecture development. As such, the research settings are to test the *feasibility* of the research concepts, *characterise* the design rationale problem and then *propose* methods to address the issues. The validity of the conceptual models is therefore limited to demonstrating that such an approach is feasible and likely to be useful. Like most software engineering research projects, the real success of this work can only be tested and validated through its continuous applications and improvements in industrial projects, which is beyond the scope of this thesis.

4.3 Summary

Design rationale research has been ongoing for over two decades but its adoption by the industry is still questionable. As such, software engineering research in this area is still in the early stages of problem identification and concept formulation. In this thesis, we have identified six research questions. The current state of the industry (questions 1,2 and 3) are validated by using a survey method and a statistical analysis.

The methodology to address the effectiveness of architecture design rationale representation (question 4) is validated in an empirical study by comparing it with traditional design specifications. The other two research questions on the application of architecture design rationale (questions 5 and 6) are implemented and demonstrated by using an industry example.

Part II

Architecture Design Rationale Practice in the Software Industry

Chapter 5

A survey of the use of architecture rationale

Design rationale captures the knowledge and reasoning that justify the resulting design. This knowledge describes how the design satisfies functional and quality requirements, why certain design choices are selected over alternatives and what type of system behaviour is expected under different environmental conditions [54, 93]. Despite the growing recognition of the need for documenting and using architecture design rationale by researchers and practitioners [8, 11, 27], there is a lack of appropriate support mechanisms and guidelines on what are the essential elements of design rationale, and how to document and reason with design rationale during decision making. The recently adopted IEEE standards (1471-2000) for describing architecture [70] and the architecture documentation methods like Views & Beyond (V&B) [23] raise the awareness of the necessity to documenting design rationale, limitations on their applications still exist.

The need to improve the capture and the use of design rationale in system design and maintenance has been reported by several researchers [11, 170, 14]. They have alluded to a perception that architects generally do not realize the critical role of explicitly documenting the contextual knowledge about their design decisions. Lack of empirical evidence makes it difficult to support or refute these claims. We believe that understanding the current industry practice of design rationale is one of the most important steps towards that goal. Presently, there is little empirical research that studies what practitioners think about design rationale, how they document and reason with design rationale, and what factors prevent them from documenting design rationale.

In order to improve our understanding of the state of reasoning practice in the software industry, we have gathered evidence from practising architects who design architectures

on a regular basis. Their inputs have provided insights into the practice of applying design rationale. This chapter reports the findings of a survey of practising architects in the Asia Pacific region. The findings of this survey have shed light on how design rationale are used, documented, and perceived by designers and architects working in this region.¹ The objectives of the study are:

- To understand the architects' perceptions about architecture design rationale and the importance of the different elements of design rationale (such as design constraints, design strengths and weaknesses).
- To determine the frequency of documenting and reasoning with different elements of design rationale, the main reasons for not documenting design rationale, and the common methods, techniques, and tools used to document design rationale.
- To identify the potential challenges and opportunities for improving the use and documentation of design rationale in practice.

During this study, we have encountered several interesting findings which enabled us to identify a set of research questions for further investigations. Since a theory explaining the attitude and behaviour toward the use of design rationale does not exist, this study employs an inductive approach (i.e., using facts to develop general conclusions) as an attempt to move toward such a theory.

The chapter makes three significant contributions to the Software Architecture (SA) discipline:

- It presents the design and results of the first survey-based empirical study in architecture design rationale practices.
- It provides information about how practitioners think about, reason with, document and use design rationale.
- It identifies the problems and contradictions of current design rationale practices. As a result, we propose a research agenda that aims to explore and enhance current architecture design rationale practices.

We discuss the current approaches to using design rationale in Section 5.1. We present our survey methodology in Section 5.2. Section 5.3 presents the results of the survey. A discussion of our findings and their limitations are in Sections 5.4 and 5.5 respectively.

¹This chapter is based on our work published in [157, 158, 6]. Han, Ali Babar and Gorton have suggested questions in the questionnaire and have contributed to the writing of the papers.

5.1 Architecture rationale in the software industry

Despite the growing recognition of the need for documenting and using architecture design rationale by practitioners [8, 11], there is a lack of understanding of how decisions are made, reasoned and justified in day-to-day architecture design. As described earlier, the software industry is not employing systematic rationalisation methods in any significant way. Hence, understanding the current industry practice of design rationale is one of the first steps towards providing a systematic approach to address the problems. However, there is little empirical research that studies what practitioners think about design rationale, how they reason with and document design rationale, and what factors prevent them from documenting design rationale.

5.1.1 Design rationale approaches in software engineering

Early work emphasizing the importance of design rationale in software design can be found in [114, 124]. Since then, the software engineering community has experimented with several design rationale approaches such as Issue Based Information Systems (IBIS) [87], Questions, Options, and Criteria (QOC) [97], Procedural Hierarchy of Issues (PHI) [99], and Design Rationale Language (DRL) [94]. Most of these methods have been adopted or modified to capture the rationale for software design decisions [124] and requirements specifications [34, 92, 128]. Other approaches (e.g. [123, 126]) combine rationale and scenarios to elicit and refine requirements.

Design rationale have been considered an important part of software architecture since Perry and Wolf [119] laid the foundation for the evolving community of software architecture. In the following years, researchers have emphasized the need for documenting design rationale to maintain and evolve architectural artifacts and to avoid violating design rules that underpin the original architecture [8, 11]. The growing recognition of the vital role of documenting and maintaining rationale for architectural decisions has resulted in several efforts to provide guidance for capturing and using design rationale such as the IEEE 1471-2000 standard [70] and the Views and Beyond (V&B) approach to document software architecture [23].

However, both of these are deficient in several ways. For example, the former provides a definition of design rationale without further elaborating on their nature and how they might be captured. The latter method provides a list of design rationales without justifying why they are important and how the information captured is beneficial in different design context. Moreover, it is unclear if the list of design rationales are complete.

Different approaches tend to characterize design rationale with different information. For example, [170] provides a template that captures certain types of information as design rationale; the V&B [23] approach considers other types of information such as information cross-cutting different views as design rationale. Thus, there is clearly a need for a common vocabulary or standard guidance so that practitioners understand the issues in reasoning with and consistently documenting design rationale.

5.1.2 Generic design rationale

The nature of design rationale is different for different design activities. The architecture decision process is different from the detailed design decision process. Brathall et al. [12] suggested that architectural level design addresses decisions with system-wide implications. Lassing et al. [89] suggested that architecture decisions have a large influence on the quality of the resulting system. Eden and Kazman [37] suggested that factors that separate architecture design activities from other design activities are: (a) architecture is concerned with the selection of architecture elements, their interactions and the constraints, whereas design is concerned with the modularisation and detailed interfaces of design element; (b) architecture is concerned with issues beyond algorithms and data structures of the computations; (c) architecture focuses on the externally visible properties of the software components. Since architecture design is at a higher level of abstraction and considers an array of different inputs, the complexity of the design reasoning is generally higher than the detailed design of a localised software component. Therefore, the design rationale for the two levels of design would be quite different. Given the complexity of architecture design, the types of design rationale and the way they influence architecture decisions are not very well understood, hence we are motivated to examine this area.

In this survey, we used nine types of generic design rationales selected from various references to test if and how our respondents perceive and use them. This set of generic rationale characterizes different aspects in which reasons can be portrayed and compared. Their selection is based on the templates or methods proposed by researchers to capture design rationale [23, 170, 8, 159]. A generic design rationale is an abstract grouping of the reasons for justifying decisions that are made in the design process. We used common terminologies so that practitioners could relate to them. Since this is an exploratory study, the list of generic design rationales (below) is comprehensive but not exhaustive.

1. *Design constraints* are the limitations placed on the design. They can be business or technical in nature [23].
2. *Design assumptions* are used to describe what are otherwise unknown factors that

affect the design [170, 88].

3. *Weakness* of a design describes what the design cannot achieve, which may be functional or technical in nature [8].
4. *Benefit* of a design describes what benefits the design can deliver to satisfy the technical or functional requirements [10, 159].
5. *Cost* of a design describes the explicit and implicit costs to the system and the business [10, 159].
6. *Complexity* of a design is a relative measure of the complexity of the design in terms of implementation and maintenance [42, 159].
7. *Certainty of design*, i.e. the design would work, is a measurement of risk that the design would meet its requirements [19, 159].
8. *Certainty of implementation*, i.e. the design is implementable, is a measurement of risk that the development team has the skill and the resources, in terms of schedule and cost, to implement the design [19, 159].
9. *Tradeoffs* between alternative designs is a mechanism to weigh and compare alternatives given each alternative design has its supporting design rationale and priorities [8].

Although the above list of design rationales have been suggested by various researchers and common sense tells us that they are useful, no empirical studies have been carried out to show that they are actually used in the software industry. In this survey, we asked our respondents to rank these rationales according to their usefulness and how often they are being used and documented. The results are reported in Section 5.3.

5.2 Survey methodology

Research method: Considering the objectives of our research and available resources, we decided to use the survey research method to understand architects' perception of, and current practices in architecture design rationale. A survey research method is considered suitable for gathering self-reported quantitative and qualitative data from a large number of respondents [80]. Our survey design was a cross-sectional, case control study. Survey research can use one or a combination of data gathering techniques such as interviews, self-administered questionnaires and others [95]. We decided to use a questionnaire as a

data collection instrument because we wanted to obtain the information from a relatively large number of practitioners, many of whom we would not be able to contact personally.

Survey instrument construction: Having reviewed the published literature on design rationale, we developed a survey instrument consisting of 30 questions on the understanding and practice of design rationale and 10 questions on demographics (e.g. age, experience, gender, education and others) of the respondents. Some of the demographic questions were designed for screening the respondents and identifying the data sets to be excluded from the final analysis. The questions were placed in different sections, namely design rationale importance, design rationale documentation, design rationale usage, and demographic information. Questions on demographics were put in the last section as it is considered good practice [80]. In addition, a coversheet explaining the purpose of the study and defining various terms was attached to the questionnaire (see Appendix C).

Instrument evaluation: The questionnaire underwent a rigorous review process for the format of the questions, suitability of the scales, understandability of the wording, the number of questions, and the length of time required to complete by experienced researchers and practitioners in the software architecture domain. We ran a formal pilot study to further test and refine the survey instrument. The pilot study was conducted with eight people who were considered strongly representative of the potential participants of our survey research (i.e. practitioners with more than three years software design experience). Data from the pilot study was not included in the analysis of the main survey.

Instrument deployment: We decided to use an online web-based survey, as this is usually less expensive and more efficient in data collection [142]. In order to implement the survey, we used the web-based tool Surveyor [111]. Participants accessed the survey through a URL.

Target population: The inclusion criteria was a software engineer with three or more years of experience in software development and who has worked or is working in a design or architect role. We did not have a formal justification for the amount of experience required of valid respondents. We based this on our extensive experience in designing architectures for large systems that made us believe that people with three or more years of experience in software development would be able to give reliable answers to the type of questions we had.

Sampling technique: The study needed responses from likely time-constrained software engineers, who we expected were less likely to respond to an invitation from unfamiliar sources. This made it hard to apply random sampling. Consequently, we decided to use non-probabilistic sampling techniques, availability sampling and snowball sampling.

Availability sampling operates by seeking responses from those people who meet the inclusion criteria and are available and willing to participate in the research. Snowballing requires asking the participants of the study to nominate other people who would be willing to participate. The major drawback of non-probabilistic sampling techniques is that the results cannot be considered statistically generalizable to the target population [80], in this case software designers/architects. However, considering the exploratory nature of our research, we believe that our sampling techniques were reasonable.

Invitation mechanics: We used two means of contacting potential respondents: personalized contact and professional referrals. The invitation letters were sent to a pool of software designers/architects drawn from the industry contacts of the four investigators and past and current students of the postgraduate information technology courses offered by the Swinburne University of Technology and the University of New South Wales. We requested the invitees to forward the invitation to others who were eligible for participation and provide us the contact for the forwarded invitation.

Data validation: For access control and data validation purposes, the survey URL was sent via email. Moreover, the responses gathered in the survey provided another mechanism of checking the validity of the respondents as genuine software engineering practitioners. For example, only one of the 81 respondents did not provide a job title and all other respondents had relevant job titles. A large number of respondents (55%) provided quite insightful and detailed comments to several open-ended optional questions.

5.3 Survey findings

The survey questionnaire was divided into eight main parts. Some of the key areas are the perception of the importance of design rationale; the use of design rationale; the documentation of design rationale. The profile of the respondents and the survey results are discussed and analyzed in this section.

5.3.1 Demographic data

We directly sent survey invitations to 171 practitioners. Our invitation was forwarded to 376 more people by the original invitees, meaning 547 invitations were sent. We received a total of 127 responses, which corresponds to 23% response rate. Anonymity and lack of resources did not allow us to contact non-respondents. Out of the total responses, we decided to exclude 46 responses from the analysis as they were incomplete or the respondents did not meet the work experience criteria (minimum 3 years software development

experience).

In summary, 80.2% of our respondents were male and 19.8% are female. The Office of Technology Policy uses Census figures to estimate that women represent 26.9% of computer systems analysts in the United States [101]. An Australian report found that 24% of undergraduate students in Information Technology in 2003 are women [17]. Although there is no statistics for the gender distribution of architects and designers for the Asia Pacific region, the reference information suggests that the ratio of gender distribution in this survey is reasonable. 67.9% of the respondents live in Australasia, 28.4% reside in Asia and 3.7% did not specify the region of their residence.

The respondents' experience in the information technology industry varies between 4 years and 37 years with an average of 17.12 years and a median of 15. The respondents have worked as a designer or architect for 9.75 years on average and a median of 8 years. These results show that the average respondents are experienced in design and architecture.

The average length of time an architect work with an organization (current or previous) is 7.65 years (a median of 6 years). This profile indicates that the respondents have had relatively stable jobs and they are likely to be familiar with the development standards within their organisations.

The average number of co-workers on the current (or last) project is 25 people (a median of 15 people). Although we cannot directly identify the size of the projects our respondents are involved in by the amount of software development such as source line of codes or development effort such as man months, this profile gives an indication of the relative size of the projects. 85.2% of the respondents have received an IT related tertiary qualification. This question is aimed to clarify the level of IT training received by the respondents.

The demographic data gives us confidence that our respondents are practitioners who are experienced in software architecture and design. Despite not being able to apply systematic random sampling because of the reasons described in Section 5.2, the results resemble the characteristics of architects and designers that we expect.

5.3.2 Job nature of architects / designers

In the survey, we asked respondents to tell us the primary tasks they perform as a designer/architect. A primary task is a task in which they spend at least 10% of their time on. The objective is to find out the scope of their job role. A summary of the percentages of respondents who perform those primary tasks are listed below:

- overall system design (86.4%)
- requirements or tender analysis (81.5%)
- non-functional requirements design (64.2%)
- software design and specification (58%)
- project management tasks (50.6%)
- IT planning and proposal preparation (49.4%)
- data modelling (44.4%)
- implementation design (42%)
- program design and specification (35.8%)
- test planning and design (29.6%)
- training (19.8%)

Our typical respondent's main efforts are spent in the early project phases including requirements and tender analysis, overall design, high level design, non-functional design and software design. Most of them also have management responsibilities such as project management and IT planning. To a lesser extent, they perform detailed design and implementation activities.

We asked our respondents if their projects or organisations recognise software architect roles. 43.2% of respondents said software architects are formally recognized across all projects in the organization, while 48.1% said only some projects in the organization recognized the use of architects. This may be due to the organization structure or the nature of the projects. It has been found that software architects would be involved in projects under the following circumstances: new projects (23.5%); mission critical projects (25.8%); high risk projects (27.2%); high cost projects (18.5%).

5.3.3 Designer's perception of the importance of design rationale

As there is little empirical evidence on how important design rationale are considered by designers, we posed a number of questions to this end. Respondents were asked to indicate how often they reason about their design choices and whether they think that design rationale are important to justify their design choices.

Table 5.1: Frequency of Reasoning about Design Choices

	Never				Always
	1	2	3	4	5
No of Respondents	0	1	8	34	38
Percentages	0	1.2	9.9	42	46.9

Table 5.2: Importance of Design Rationale in Design Justification

	Not Important			Very Important	
	1	2	3	4	5
No of Respondents	0	1	11	30	39
Percentages	0	1.2	13.6	37	48.1

The responses to those questions revealed (Table 5.1 and 5.2) that the majority of designers frequently apply reasoning to justify their architectural choices and they also consider that design rationale are important to justify their design choices.

We also asked the respondents about the frequency of considering alternative architecture designs (explanation for alternative architecture designs was provided) during their design process, as this is another indicator of the awareness of reasoning about design choices and the rigour that needs to be employed during this process. The responses to this question are provided in Table 5.3. The result indicates that the majority of respondents compare between alternative designs before selecting a particular architectural design among available alternatives.

Table 5.3: Frequency of Considering Alternative Designs

	Never				Always
	1	2	3	4	5
No of Respondents	0	1	15	31	34
Percentages	0	1.2	18.5	38.3	42

We asked the respondents to rank the importance of each of the nine generic design rationales listed in the survey. This ranking reflects the perception of respondents towards how useful a given design rationale is in design. Since decision making is something our respondents do on a regular basis, their perception of design rationale's importance should reflect the reasoning process that is performed intuitively. Table 5.4 presents the responses to this question. The majority of respondents considered that all nine design rationales are important.

The responses for all rationales are skewed towards the very important end. *Benefits of design*, *design constraints* and *certainty of design* received the highest support with combined level 4 and 5 percentages of 90.12%, 87.65% and 85.19% respectively. All other rationales are also considered important with the majority of respondents selecting level

Table 5.4: Importance of Each Generic Rationale

	Not				Very
	Important	2 (%)	3 (%)	4 (%)	Important
	1 (%)				5 (%)
Design Constraints	0.0	1.2	11.1	38.3	49.4
Design Assumptions	3.7	7.4	14.8	44.4	29.6
Weakness	2.5	7.4	28.4	43.2	18.5
Costs	0.0	7.4	14.8	43.2	34.6
Benefits	1.2	1.2	7.4	54.3	35.8
Complexity	0.0	2.5	25.9	46.9	24.7
Certainty of Design	0.0	3.7	11.1	29.6	55.6
Certainty of Implementation	2.5	4.9	16.1	32.1	44.4
Tradeoffs	0.0	4.9	30.9	44.4	19.8

4 or 5. This shows that most designers perceived that these rationales are important in reasoning about design decisions. Apart from the above-mentioned nine generic rationales, we also asked the respondents to add other rationales that they use for making architectural design choices. A significant number of the respondents (twenty eight), mentioned additional types of factors that influence their design choices. We have classified those factors into three broad categories. They are:

Business Goals Oriented

1. Enterprise strategies, technical directions and organizational standards
2. Management preferences and acceptance
3. Adherence to industry standards
4. Vendors relationship

Requirements Oriented (functional/non-functional)

5. Fulfill functional and non-functional requirements
6. Satisfy client business motivations
7. Buy vs. build decisions
8. Maintenance and expected life-cycle of products

Constraints and Concerns

9. Viability of solutions
10. Consideration of existing architecture constraints
11. Current IT architecture and capabilities
12. Compatibility with existing systems
13. Prior use of the design and how successful
14. Availability of technology and tools
15. Prototype and staged delivery
16. Time to market
17. Available time
18. Risk

These rationales show a variety of factors that influence the design decision making process. They also provide the context to enable architects and designers to trade-off conflicting goals by argumentation.

5.3.4 Using design rationale

Another important area of the survey was how frequently design rationale are used. An objective of the study is to discover whether respondents' perceived importance of design rationale (i.e. what they think) and their behaviour (i.e. what they do) are consistent. Therefore, the same set of design rationale we presented and discussed in the previous sections were used to query our respondents. In this section, we present the results of a

5.3. Survey findings

multi-item question on how often they use the generic rationales to reason about architectural decisions. Most respondents say that they frequently or always use the nine generic design rationale listed in the questionnaire. Table 5.5 summarizes the frequency of how respondents think they use the different types of rationales.

The results show that the *design constraint* rationale is used most frequently. The reason for the high usage could be that designers are usually expected to explore the solution space using certain business and technical constraints. These constraints are probably prominent in their minds and must be taken into account from the beginning of a project.

Table 5.5: Design Rationale Frequency of Use

	Never				Always
	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)
Design Constraints	0.0	0.0	12.3	42.0	45.7
Design Assumptions	2.5	2.5	30.9	33.3	30.8
Weakness	1.2	8.6	34.6	37.0	18.6
Costs	1.2	9.9	19.8	38.3	30.8
Benefits	1.2	1.2	12.3	49.4	35.9
Complexity	0.0	2.5	27.2	34.6	35.7
Certainty of Design	2.5	1.2	11.1	32.1	53.1
Certainty of Implementation	3.7	3.7	16.0	33.3	43.3
Tradeoffs	0.0	6.2	29.6	42.0	22.2

Other frequently used rationales are *benefits of design*, *certainty of design* and *certainty of implementation*. The combined usage frequencies (level 4 and 5) of these rationales are 85.3%, 85.2% and 76.6% respectively. We suspect that designers frequently use these types of rationales because they have to make a business case for their architectural choices to the management. They also have to justify their design choices using technical arguments to architecture reviewers and technical stakeholders such as programmers, implementers and maintainers. So they use those rationales more often to help them justify their architectural decisions.

On the other hand, respondents are less likely to use those rationales that can highlight the weaknesses of their design decisions. The combined usage frequencies (level 4 and 5) reported by respondents are relatively lower: *design weakness* (55.6%), *costs* (69.1%) and *complexity* (70.3%). This tendency of designers to pay relatively less attention to the weaknesses of their design decisions is similar to Lassing et al.’s warning against gathering scenarios to evaluate an architecture by the designers themselves, as it is highly likely that they would come up with the scenarios that have already been addressed by the proposed architecture [90]. Thus, we hypothesize that designers unknowingly look for those positive rationales to support the design decisions and pay less attention to those negative rationales.

5.3.5 Documenting design rationale

Several arguments have been made about the importance of documenting key architecture decisions along with the contextual information [114, 170]. It is important that design rationale are documented to a sufficient extent in order to support the subsequent implementation and maintenance of systems. With regards to design rationale documentation attitude and practice, we paid special attention to the frequency of documenting discarded design decisions, frequency of documenting each of the generic rationales, the reasons for not documenting design decisions (barriers to design rationale documentation), and methods and tools used for documenting design rationale. Table 5.6 presents the breakdown of the responses to the question on documenting discarded design decision.

Table 5.6: Frequency of Documenting Discarded Decisions

	Never				Always
	1	2	3	4	5
No of Respondents	11	18	17	19	16
Percentages	13.5	22.2	21	23.5	19.8

About 44% of the respondents (level 4 and 5) document discarded decision very often. About 36% of the respondents (level 1 and 2) do not document discarded decisions. This is likely because designers are under pressure to produce design specifications on schedule. At this stage, we are not aware of any software development or project management methodology that mandate the documentation of discarded decisions or methodically schedule time for such activities to take place. However, documenting the discarded decisions can help newcomers to the project understand the reasons for discarding design alternatives and expedite such understanding during the maintenance phase of the project.

Table 5.7: Frequency of Documenting Generic Design Rationale

	Never				Always
	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)
Design Constraints	1.2	2.5	13.6	19.7	63.0
Design Assumptions	3.7	3.7	13.6	25.9	53.1
Weakness	3.7	23.5	37.0	14.8	21.0
Costs	7.4	16.0	30.9	21.0	24.7
Benefits	2.5	9.9	18.5	32.1	37.0
Complexity	3.7	9.9	35.8	30.9	19.7
Certainty of Design	18.5	14.8	19.8	24.7	22.2
Certainty of Implementation	18.5	17.3	24.7	22.2	17.3
Tradeoffs	6.2	18.5	25.9	32.1	17.3

Respondents were also asked to indicate the overall frequency of documenting design rationale. 62.9% of the respondents replied that they completely document design rationale, which is an encouraging finding considering the common perception of design rationale not being widely documented.

We also investigated the frequency of documenting each of the generic rationales. Table 5.7 summarizes the frequency of documentation for each of the nine generic design rationales used in this research. The results show that *design constraints* and *design assumptions* are documented very frequently but the level of documentation is relatively lower for other types of rationale. 27.2% of the respondents replied that they never or seldom document *design weakness*. Similarly, 33.3% of respondents said they never or seldom document *certainty of design*. 35.8% of them said they never or seldom document *certainty of implementation*. These findings appear to agree with our previous assertion that negative rationales receive relatively less attention.

Based on these results, it appears that design rationale are commonly documented by software designers and architects. However, it also appears that the reasons about why a design alternative is chosen and why it is better than other alternatives are usually not documented. We do not have any theoretical grounds for explaining this phenomenon.

While the level of documentation is relatively high, the survey results give us no insight as to whether the rationales are sufficiently documented so that other designers can understand the architecture design without additional assistance. This raises two issues worthy of further investigation, namely:

- identify the rationales documented by architects and evaluate their effectiveness in explaining the design;
- identify how the documented rationales are used in the development life-cycle.

Barriers to documenting design rationale

We were also interested in identifying and understanding the reasons for not documenting design rationale. We believe that it is important to identify those factors that undermine efforts in documenting and maintaining design rationale. The respondents were given a list of reasons that are thought to be common causes of non-documentation in software engineering such as perceived usefulness, project budget and lack of time. The respondents were given a text box to provide additional reasons.

Table 5.8: Reasons for Not Documenting Design Rationale

Topic of Questions	Percent of Respondents	Number of Respondents
No standards	42	34
Not aware of	4.9	4
Not useful	9.9	8
No time/budget	60.5	49
No suitable tool	29.6	24

Table 5.8 summarizes the responses to the reasons for not documenting design rationale. These results reveal that lack of time/budget (60.5%) is considered the most common cause of not documenting design rationale. There is also a lack of appropriate standards (42%) and tools (29.6%) to support the documentation process. Only 4.9% of the respondents were not aware of the need to document design rationale, while 9.9% of the respondents said that documenting design rationale is not useful. A few respondents also provided several other reasons for not documenting design rationale. These reasons are:

- Lack of formal review process
- Not required for non-complex solutions
- Afraid of getting into a long cycle of design review
- Not required for low impact solution
- The dynamic nature of technology and solutions make it useless to document design rationale.
- It is not required for high level decision making

In summary, the reasons for not documenting design rationale can be classified into these groups: (a) the lack of standards and processes to guide why, how, what and when design rationale should be documented; (b) the time and budget constraints of projects; (c) the question of whether the cost and benefit of rationale documentation can be justified. These reasons are analogous to those concerning requirements traceability documentation in immature software development organizations [129]. Since the sample population is not specific to an industry or capability maturity level, the results may indeed reflect the general architecture design practice.

Methods and tools for documenting design rationale

An important part of any task in the software development life-cycle is the availability of process support and suitable tools to enhance productivity. It is important to identify what type of support is available to designers to improve design rationale practices. Hence the survey included a question on the methods and tools used for documenting design rationale. Twenty respondents provided comments to this question. We list the methods and tools used by the respondents to document design rationale below:

- Apply organization standards and templates to document using Word / Visio / Excel / Powerpoint
- UML tools
- IBM GS Methodology
- Document architecture decisions using formal method and notation
- Internally developed tools
- QMS Design Template document
- Requirements Traceability Matrix
- Architecture tool CORE

Our respondents used proprietary tools, proprietary templates, the Microsoft Office suite or UML design tools to document design rationale. As we suggested earlier, there is little awareness about the standards like IEEE 1417-2000 and methodologies like V&B. Design rationale tools like gIBIS [26] are not used. Although these results are anecdotal evidence, they point to the lack of industry standards as well as proper tools to capture, maintain and trace design rationale during the development life-cycle.

5.3.6 Comparing usage and documentation of design rationale

Given that design rationale are recognized by our respondents as important, it is revealing to compare the survey results concerning importance, use and documentation of each of the nine generic rationales. Table 5.9 presents the combined results from the last three sections. The scale is condensed by combining level 4 and level 5 (See the scale in the previous sections to interpret the results).

Table 5.9: Design Rationale Usage

	Level of Importance (%)	Frequency of Use (%)	Frequency to Document (%)
Benefits of Design	90.1	85.3	69.1
Design Constraints	87.6	87.6	82.7
Certainty that Design would work	85.2	85.2	46.9
Cost of Design	77.7	69.1	45.7
Certainty that design is implementable	76.5	76.5	39.5
Design Assumptions	74.0	64.1	79.0
Complexity of Design	71.6	70.3	50.6
Tradeoffs between alternatives	64.2	64.2	49.4
Weakness	61.7	55.6	35.8

We used Spearman’s Rank Order Correlation (ρ) to test the correlations between the *Level of Importance* and the *Frequency of Use* for the nine generic design rationale. This revealed that they are all correlated with r values all above 0.5 with the exception of *design complexity*, and all of them tested significant with $p < 0.01$. This indicates that there is a strong relationship between what respondents believe and what they practice. We also observe that across most design rationale, the usage frequency is less than the perception of importance, and the documentation frequency is less than the usage frequency. The lower frequency of documentation may be caused by the reasons put forwarded by the respondents (Section 5.3.5). This result may provide an explanation for the claims of design knowledge vaporization [11, 170].

5.3.7 Design rationale and system maintenance

In the survey, we asked the respondents about the use of design rationale in relation to carrying out impact analysis during system maintenance. To this end, we asked how often they revisit design documentation and specifications to help them understand the system before performing enhancements. Table 5.10 presents their responses, which indicate that the majority of the respondents consult design documentation frequently while performing maintenance or enhancement tasks. These results show that if design rationale are sufficiently documented and provided to the software maintainers, they are more likely to use the design rationale. These results are also consistent with the findings reported in [137], which conclude that system documentation, if available, is frequently used when performing maintenance tasks. It is also found that if the knowledge leading to design is available, a maintainer can effectively perform modification tasks by consulting that knowledge [57].

Table 5.10: Frequency of Revisiting Design Documentation before Making Changes

	Never				Always
	1	2	3	4	5
No of Respondents	0	8	9	20	30
Percentages	0.0	11.9	13.4	29.9	44.8

We asked our respondents how often they think they forget the reasons underpinning their design decision after a period of time. Table 5.11 shows the results to this question. The responses show that 40.9% of respondents (levels 4 and 5) forget the reasons concerning design decisions. This finding should provide a strong reason for regularly documenting and maintaining design rationale to support architecture maintenance and evolution. As mentioned earlier, some of the reasons could be reconstructed through inspecting available design specifications, but some reasons will inevitably be lost if the design is complex and the system was developed some time ago.

Table 5.11: Tendency of Forgetting the Reasons for Justifying Design Decisions

	Never				Always
	1	2	3	4	5
No of Respondents	2	15	22	22	5
Percentages	3.0	22.7	33.3	33.3	7.6

Often the architect who is responsible for maintenance is not the same person who originally designed the system. In such circumstances, we asked respondents whether they know why existing designs were created without documented design rationale. The results are shown in Table 5.12. Most respondents (80%) either agree or strongly agree with the statement that without design rationale, they may not understand why certain decisions are made in the design.

Table 5.12: Do Not Understand Design without Design Rationale if Not Original Designer

	Strongly Disagree				Strongly Agree
	1	2	3	4	5
No of Respondents	1	3	9	23	29
Percentages	1.5	4.6	13.8	35.4	44.6

When respondents were asked about the usefulness of design rationale to help understand past design decisions to assess potential modification, a majority of the respondents find design rationale helpful in this regards (see Table 5.13). Hence, we concluded that there is evidence to support the claims that design rationale are an important source of effective reasoning during architectural modification. It is also stressed that if the knowledge concerning the domain analysis, patterns used, design options evaluated, and decisions made is not documented, it is quickly lost and hence unavailable to support subsequent decisions in the development lifecycle [11]. Software maintenance experts also agree that many facts may be lost to the project, either because the developers may no longer be available to the project or the limitations on a human's ability to memorize detailed facts [75].

Table 5.13: Design Rationale Helps Evaluate Previous Design Decision

	Strongly Disagree				Strongly Agree
	1	2	3	4	5
No of Respondents	0	3	10	21	33
Percentages	0.0	4.5	14.9	31.3	49.3

Finally, when respondents were asked how often they do impact analysis during maintenance. Table 5.14 shows that a large number of respondents perform impact analysis before making any changes. Since impact analysis may consume a significant amount of resources and 80.6% of designers (levels 4 and 5) consider it to be useful in evaluating

5.3. Survey findings

previous design decisions (Table 5.13), it can be argued that the availability of design rationale can improve this process. Design rationale is considered an important input to impact analysis because if the knowledge about the factors that may have influenced the original design decisions is available, designers do not have to spend a large amount of time to understand the existing design and the potential ramifications of any changes [128].

Table 5.14: Frequency of Performing Impact Analysis

	Never				Always
	1	2	3	4	5
No of Respondents	0	4	13	29	19
Percentages	0.0	6.2	20.0	44.6	29.2

We were also interested in determining the importance of the tasks performed in impacts analysis, some of which require design rationale support. The importance of such tasks during impact analysis can be considered as an indicator of the need or usefulness of design rationale to improve this activity and subsequently the maintenance process. To investigate this issue, there was a multiple-items question on the importance of various tasks. Table 5.15 shows what the respondents think of the importance of the factors. In carrying out impact analysis, most respondents trace requirements. They analyse the design and design rationale based on the available design specifications. In the analysis, they are concerned about the feasibility of the implementation, its costs and risk. They are also concerned that the new design is consistent with the constraints and the assumptions of the existing system.

Table 5.15: Importance of Each Impact Analysis Task

	Not Important				Very Important
	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)
(a) Analyse and trace requirements	1.5	0.0	10.3	39.7	48.5
(b) Analyse specifications of previous design	1.5	10.3	19.1	35.3	33.8
(c) Analyse design rationale of previous design	1.5	11.9	32.8	32.8	21.0
(d) Analyse implementation feasibility	1.5	1.5	8.8	41.2	47.0
(e) Analyse violation of constraints and assumptions	1.5	8.8	25.0	41.2	23.5
(f) Analyse scenarios	1.5	7.6	16.7	36.4	37.8
(g) Analyse cost of implementation	0.0	10.3	13.2	30.9	45.6
(h) Analyse risk of implementation	0.0	4.4	5.9	32.4	57.3

Given the results, it is obvious that architects and designers often have the need to refer to design documentation for impact analysis. Some of information that they use require design rationale support such as understanding design assumptions, constraints, cost and risk. However, this information are not necessarily organised in a way that can be easily used for impact analysis. Additional factors that have been discovered in this survey can contribute to the difficulties in impact analysis: (a) architects can forget the reasons behind the design; (b) the person doing the maintenance may not be the original designers.

We further analyse the survey data and correlate the the importance of impact analysis tasks with the use and documentation of design rationale. This correlation is performed to check that those respondents who carry out impact analysis tasks also use and document design rationale. The correlation results are shown in Table 5.16. The correlation analysis found that items (a), (b), (c) and (e) are positively correlated with the overall level of design rationale documentation, they are also positively correlated to using design rationale to justify design decisions. The presence of correlation indicates that those respondents who analyse and trace requirements are also the people who prepare design rationale documentation thinking that they are important. Similarly, those who believe that design specifications, design rationale and constraints are important in impact analysis also value the importance of design rationale to justify their design decisions. That means for those designers who document design rationale, it seems likely that they will use this information during impact analysis. Moreover, it also indicates that the designers who trace requirements and design specifications value the importance of design rationale. These findings provide evidence to support the usefulness of design rationale for system maintenance.

Table 5.16: Correlation between Use of Design Rationale and Impact Analysis Tasks

Impact Analysis Task	Use of Design Rationale	Correlations
(a) Analyse and trace requirements	Level of documentation	$[r(67) = +.297, p < 0.05]$
	Use design rationale to justify decisions	$[r(67) = +.333, p < 0.01]$
(b) Analyse specifications of previous design	Level of documentation	$[r(67) = +.267, p < 0.05]$
	Use design rationale to justify decisions	$[r(67) = +.279, p < 0.05]$
(c) Analyse design rationale of previous design	Level of documentation	$[r(67) = +.251, p < 0.05]$
	Use design rationale to justify decisions	$[r(67) = +.351, p < 0.01]$
(e) Analyse violation of constraints and assumptions	Level of documentation	$[r(67) = +.295, p < 0.05]$
	Use design rationale to justify decisions	$[r(67) = +.306, p < 0.05]$

5.3.8 Risk as a design rationale

When an architecture design is committed, designers may not necessarily have full knowledge of whether the implementation could fully satisfy the requirements. For instance, the performance of a system cannot be easily determined. Additionally, it is difficult to estimate how the architecture would cope with future changing technologies and requirements, so architects would make assumptions and assessments to support their decisions [88]. Examples are “the likelihood of this technology not being supported in the next five years is low” or “the performance should be okay because we don’t expect any database contention”. In such cases, the architecture design cannot be fully tested until much later in implementation or even after it has been deployed. This aspect of the architecture design is quite different to designing a software module to satisfy a set of functional requirements. Detailed design which deals with localised issues can be unit-tested more easily.

Since there are uncertainties in architecture design, architects make risk assessments either implicitly or explicitly based on their beliefs. As such, ATAM suggested that risks and non-risks should be documented [8]. Two types of risk assessments are suggested for architecture design reasoning [159]: the architecture design can successfully deliver the required benefits; the development team is capable of delivering the design.

This survey asked the respondents to indicate if they do risk assessments and if so, what is an acceptable risk level. 24.7% of the respondents said they always explicitly quantify risks. 35.8% of the respondents said they sometimes explicitly quantify risks. 42% and 49.4% of the respondents indicated that they are *quite certain* and *most certain*, respectively, that their design can deliver the results and their team can implement the design. These results indicated that risk assessment is an important part of architecture design reasoning.

Given that risk assessment is frequently used in architecture design, the respondents were asked to identify an acceptable level of risk. 23.5% said that a 30% risk level was acceptable, 18.5% said that a 40% risk level was acceptable. However, 14.8% of respondents said that a 70% risk level was acceptable and 13.6% said that a 80% risk level was acceptable. There was no consensus on what is an acceptable level of risk.

5.4 Discussion of findings

Based on the survey, there is evidence to support that design rationale are an important part of the design documentation, and practitioners believe that design rationale should be documented. There is also a general perception that methodology and tool support for design rationale is lacking and there are barriers to design rationale documentation. These findings lead to a number of areas that require further investigation.

5.4.1 Different forms of design rationale

In addition to the nine generic design rationales reported in the survey, respondents had indicated that they consider other types of design rationale during design. Examples of the other design rationales they consider are *management preferences*, *vendor relationship* and *maintenance and expected life-cycle* (see Section 5.3.3 for a complete list). This list of rationales appeared to be different in nature from the ones that were proposed in the survey.

Except for *constraints* and *assumptions*, seven of the nine generic design rationales we

listed can be quantified in some ways. For example, costs (i.e. specific values) of design alternatives can be compared (i.e. high / low) and used in the selection.

Along with *constraints* and *assumptions*, the design rationales suggested by the respondents are contextual in nature. These rationales provide a context and a motivation to justify and reason about their design decisions. For example, the flexibility and user-friendliness requirements for the design of a system's user interface may present a conflict, i.e., a design cannot satisfy both simultaneously. As such, the architect might seek additional information and agreement on the priorities of these requirements from stakeholders in order to make a compromised decision, e.g. first user friendliness and later flexibility. The argumentation which is used to arrive at a decision would rely on a clear understanding of the underlying reasons (i.e. the context and the motivation) and the priorities of each of the requirements. Therefore, the context and the motivation of such reasoning play a major role in the decision reasoning.

Quantitative methods such as [4], [2] and argumentation methods such as [94], [97] have provided supporting technologies to capture design rationale in different ways. However, they have not addressed the fundamental issue of how design occurs and what the intrinsic reasoning process is. As such, further research to investigate how practitioners make use of various types of design rationales and the roles these rationales play in the decision making process would provide a foundation for design reasoning and design rationale capture.

5.4.2 The role of an architect

The survey results reported in Section 5.3.2 indicate that architects work on a variety of tasks such as requirements analysis, tender analysis, architecture design and software design. They also have management responsibilities. Program design and test planning is a much smaller part of their job. We interpret this result as an indication of the activities involved in the architecture design. As well as designing software, architecture design would also deal with: (a) high level issues such as planning and management; (b) constraints and assumptions arising from the information technology environment and the organisational environment; (c) the complexity of the decision making arising from the competing technical and non-technical aspects in a project. Of the architects surveyed, 48.1% said that not all projects require an architect's involvement. Projects that are either new, mission critical, high risk, high cost, complex or have high impacts on other systems or organisations require architects. The results clearly indicate that architects are involved in a broad array of tasks across the whole project scope. Given that the architecture decision making process is complex and there are many technical and non-technical considerations that influence architecture decision making, it is important to identify the design rationale

or considerations that influence architecture decision making, assess how they inter-relate in the decision process, and generalise their use into architecture design patterns [3].

5.4.3 Designers' attitude

Respondents frequently use design rationale to justify design choices. When we examine the list of design rationales they use, it appears that those design rationale that positively justify the design receive more attention than those negative rationales that explain why the design may have issues. That leads us to suspect that there might be a tendency to present *good news* rather than *bad news* during the design process. An analogous finding [79] may give us some insights to this behaviour. In many industry scenarios that we have encountered, some architects have a tendency to promote a design based on the benefits of new technologies. However they often do not explain the potential negative impacts of the new approach. Establishing if such a bias is commonly exhibited in architecture would be useful, because awareness of this phenomenon would help architects to be more objective in the assessment and selection of design choices.

5.4.4 Necessity for design rationale documentation

The survey found that there is a strong justification to document design rationale due to the tendency of the architects' forgetting their own design or the need for architects to modify systems designed by other architects. However, not every system requires comprehensive design rationale documentation because design rationale can be reconstructed for non-complex systems. The extent to which design rationale are documented also depends on the relative benefits it might deliver.

5.4.5 Design rationale to support impact analysis

Impact analysis requires the knowledge of dependency between design components [44]. The positive correlation between impact analysis tasks and the use and documentation of design rationale indicates that the respondents realize the connection between the two tasks at different stages of the development life-cycle. This finding indicates that there is a need to capture and maintain design rationale as a first-class entity to facilitate traceability during maintenance activities. The absence of design rationale may result in the violation of constraints or assumptions in a design. That is why we argue that the availability of design rationale can facilitate systematic reasoning to minimize the risks associated with system enhancements. Collectively these results indicate that there is a necessity to use

of design rationale in various aspects of impact analysis.

5.4.6 Risk assessment in architecture design reasoning

Two major differences exist between architecture design and detailed software design. Firstly, architecture design has a global impact, in terms of modifiability and correctness, to the system whilst the software design of a local module has a lower impact. Secondly, there are more certainty and testability in designing a software module than an architecture. The survey showed that architects often carry out risk assessments in architecture design. Over half of the respondents explicitly quantify risks. However, when they were asked what risk level would be acceptable, there was no consensus. This implies that there is no common understanding on how to measure risks. Is an 80% uncertainty in the architecture design acceptable? 13.6% of architects seemed to think so. Since architecture cannot be fully tested at the early stage of the development cycle, it is probably better to have a lower risk architecture design which is more likely to succeed. However, the risk assessment process in architecture design is not well understood and so there is much room for further investigations in this area.

5.4.7 Methodology support for design rationale

Some of the reasons for not documenting design rationale are budget constraints and lack of methodology. Given that most respondents consider design rationale important and their documentation useful, there needs to be guidelines under which the use and documentation of design rationale will provide greater benefits than the costs involved. This means that the need for design rationale documentation should be context dependent. For instance, a non-complex system may require little design rationale documentation since it can be reconstructed easily [27]. Our literature review shows that there is no comprehensive methodology to guide how we should use rationale-based techniques to design systems. Therefore, further studies of the use and documentation of design rationale to provide a methodology would be most beneficial.

5.4.8 Tool support for design rationale

At present, tool support for design rationale capture and retrieval is inadequate. The various tools that respondents reported using, including word processors and UML-based tools, do not have traceability features to support systematic design rationale description and retrieval. Therefore, it is important to understand how best to capture, represent

and use design rationale and then develop such tools to provide a design rationale enabled development environment.

5.5 Limitations

Our study has several shortcomings. Like most surveys in software engineering, our study faced reliability and validity threats. For example, contrary to the common perception that there is a lack of documentation of architecture design rationale, our finding shows that a large number of respondents document their design rationale. This result could be biased because there might be a tendency for the respondents to respond according to what they think is a good idea instead of what they actually practise. Following the guidelines provided in [80], we put certain measures in place to address the validity and reliability issues. For example, the research instrument underwent rigorous evaluation by experienced researchers and practitioners, all the questions were tested in a pilot study, and respondents were assured of anonymity and confidentiality. However, completely eliminating the possibility of bias error is difficult.

Most software engineering surveys also suffer from the problem of non-response error when members of the targeted sample do not respond to a survey. In our case it is possible that those who do not believe in the value of documenting architecture design rationale may have opted not to respond to the survey, which would have biased the results. Unfortunately, given the anonymous nature of the responses, we are unable to identify those participants who did not respond, hence, it is difficult to assess the representativeness of the sample.

Another limitation of the study is the non-existence of a proven theory of designers/architects' attitude towards documenting design rationale to guide our research. That is why we considered our research in understanding the practitioners' attitude and practices of design rationale documentation and usage as exploratory. Through the survey, we aimed at gathering facts in the hopes of drawing some general conclusions to help us and the software architecture community to identify research directions, and to facilitate the development of a theory on design rationale documentation and usage.

Geographical location of the respondents, mainly in the Asia Pacific region, is another limitation as the findings cannot be generalized globally. Though, Australian and Asian software engineering practitioner are internationally renowned for technical competency and high standards, the only way to test the findings for regional bias is to replicate the study in geographical regions such as the US and Europe.

5.6 Summary

The survey has gathered invaluable information about how designers use design rationale, through which we have gained important insights into the issues of design rationale use and documentation in the software industry. We found that practitioners see design rationale as important. When design rationale are available, they use design rationale to support maintenance and impact analysis. However, the level of documentation and the use of design rationale are lower than the perception of its importance. We found that architects have many roles and some of them are high-level planning and management. Architects who take on these roles might have other perspectives and considerations in their decision making in addition to technical considerations. Although risk assessment is something that architects view as important, there is no consensus as to what is an acceptable level of risk in architecture design. Respondents reported that there is a lack of methodology and tool support for the capture and application of design rationale.

Using the results of the survey, we have established the need for using design rationale in system design and maintenance activities. We have identified a number of areas for further investigations: (a) establish an architecture rationale representation that could effectively capture and communicate design reasoning; (b) provide the associated methods to support effective traceability of design reasoning in impact analysis; (c) provide a quantitative method to assess the risks or probabilities of change impacts in an architecture design. These works are described in the following chapters.

Part III

The Representation and Applications of Architecture Design Rationale

Chapter 6

Representing architecture design rationale

In the previous chapter, we have obtained the findings from a survey to support the capture and use of architecture design rationale. Similar arguments have been suggested by many others for different reasons. First, the design of a large system involves collaboration of designers who specialise in different aspects of the system. Design rationale can serve as a tool to clarify issues and assumptions that are tacit and implicit [88, 135]. Second, design rationale help to deliberate and justify system designs, such designs should be carefully considered, objective and unbiased [103]. Third, the tacit design knowledge of a system may be eroded when it is not captured [119]. Designers who understand the design may not be available for consultation. This can be very costly to system maintenance. Retaining tacit design knowledge could save time, reduce long-term cost, and help deliver software of better quality because designers would not have to second-guess the rationale behind the design [33].

Architecture design rationale is not very useful by itself. It is more useful when it relates requirements to design objects. This relationship is the basis to model architecture design rationale. Architecture Rationale and Elements Linkage (AREL) is created for this purpose. In Section 6.1, we look at the meaning of a design *reason*, which may have many interpretations. We discuss two types of design reasoning and their relevance in design decisions. We describe the concept of architecture design decision and its relationships with architecture elements in a conceptual model. We then develop the AREL representation for modelling architecture design rationale (in Section 6.2).¹

¹This chapter is based on our work published in [161].

AREL comprises two fundamental elements, Architecture Rationale (AR) and Architecture Elements (AE). *AR* represents the design justification at a decision point. *AE* represents the design artefact which participates in a decision as an input or an outcome. They are described in details in Sections 6.4 and 6.3, respectively.

6.1 A conceptual model for design reasoning

When architects and designers make design decisions, what do they consider as a reason or an intention? Should a requirement or a constraint be considered a reason for a design? Or is it some generic justification that allows designers to judge that a design is better than its alternatives? Design is a process of synthesising through alternative solutions in the design space [141]. Reasoning to support or reject a design solution is one of the fundamental steps in this process.

The concept of a reason has many dimensions. We argue that design reasoning come in two forms: motivational reasons and design rationale. *Motivational reasons* are the reasons which motivate the act of making a design and providing a context to the design. They are goals to be achieved by the architecture design, or they are factors that constrain the architecture design. An example is a requirement. Although a requirement by itself is not a reason, but the *need* of requirement is the reason for the design. There are a few aspects to a motivational reason:

- Causality - a motivational reason is an impetus to a design issue. As such, it is a cause of a design decision.
- Goal - a motivational reason can be a goal or a sub-goal to be achieved.
- Influence - a motivational reason can influence a decision by ways of supporting, rejecting or constraining a decision.
- Factuality - a motivational reason can represent information that is either a fact or an assumption.

A motivational reason can be a requirement, a goal, an assumption, a constraint or a design object exerting some influence on the architecture. It is important to represent motivational reasons explicitly as inputs to the decisions so that they are given proper attention in the decision making process. As suggested by [135], in order to have a deep understanding of software systems, undocumented assumptions have to be re-discovered.

Garlan et al. [48] found that conflicting assumptions which are implicit lead to poor-quality system architecture. As it is difficult to draw the line between requirements, assumptions and constraints [135], we take an all-inclusive approach and conjecture that missing motivational reasons (including assumptions, constraints and requirements) can affect the decision making process adversely and can result in inferior design solutions because of ill-informed decisions.

With motivational reasons come the design issues that need to be resolved to create a design solution. An architect would resolve the issues by evaluating the relative benefits and weaknesses of the available options to select the most suitable design. The arguments and the reasoning are captured as a result of the decision, i.e. the *design rationale*. To depict the relationship between motivational reasons, design rationale, and design objects, we present a conceptual model for design process in Figure 6.1, based on which we will develop the rationale-based architecture model AREL. The conceptual model capturing design reasoning relies on the distinction between motivational reasons and design rationale. There are two important aspects of such a conceptual model:

- Entity - it identifies the information that needs to be represented, namely architecture design rationale and architecture design elements.
- Relationship - it relates the architecture design rationale to the architecture elements in a structured way to explain how a decision is made and what the outcomes are.

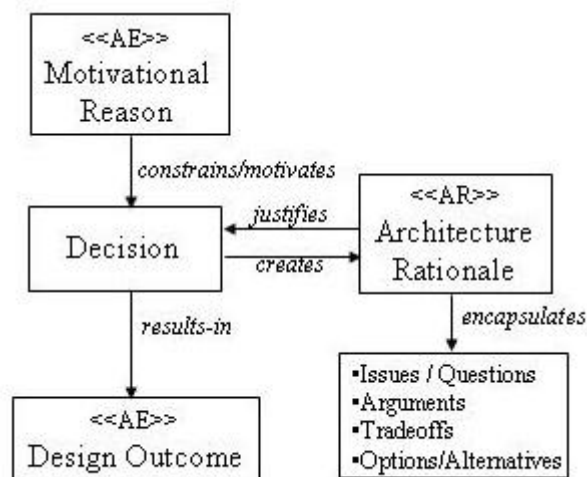


Figure 6.1: An Architecture Rationale Conceptual Model

A motivational reason acts as an input to a decision. It *motivates* and/or *constrains* a decision. A motivational reason should be explicitly represented in an architecture design,

as an architecture element, to show its influence on a decision. This is similar to a *goal* in DRL.

A decision *creates* and is *justified* by the architecture design rationale, similar to the *argument* in REMAP. The architecture rationale *encapsulates* the details of the justification. It contains a description of the issues addressed by the decision, the arguments for and against an option, and the alternative options that have been considered. Once a decision is made, the *result* of a decision is a design outcome or solution. A design outcome should be explicitly represented in the architecture design as an architecture element. Although this conceptual model has similarities, such as the representation of design rationale and design issues, to other design rationale models, it is unique because it simplifies the argumentation aspect and strengthens the linkage between motivational reasons, design rationale and design elements. This subject is further discussed in Section 6.2.

6.2 Architecture Rationale and Elements Linkage (AREL)

The AREL model is an implementation of the conceptual model using the UML notation [118, 113]. AREL is an acyclic graph which relates architecture elements *AEs* to architecture rationale *ARs* using directional links *ARtrace*. An *AE* is an architecture element that participates in a decision as an input (i.e. motivational reason) or an outcome (i.e. design outcome). They are stereotyped by $\ll AE \gg$. An *AR* encapsulates the architecture design justification of a decision. *ARs* are stereotyped by $\ll AR \gg$. Since *AR* has a one-to-one relationship to justify a decision, *AR* can therefore represent a decision point in AREL modelling. The relationships between an *AE* and an *AR* are connected by a directional association stereotyped $\ll ARtrace \gg$. It is also referred to as a *links* in the AREL definition below. The $\ll ARtrace \gg$ stereotype is used to represent the causal relationships between *AEs* and *ARs*. A causal relationship is a relationship between two entities where *AE* is a cause and *AR* is the effect. For instance, a motivational reason causes a decision to be made. Alternatively, *AR* is a cause and *AE* is an effect. For instance, a decision causes a design object to be created as an effect.

Definition 1 *An Architecture Rationale and Element Linkage (AREL) model is a tuple (AE, AR, PL) , where AE is a set of nodes representing architecture elements, AR is a set of nodes representing architecture rationales, and $PL \subseteq (AE \times AR) \cup (AR \times AE)$ is a set of directed links between the nodes, such that*

1. *all rationale nodes must be associated by links with at least one cause and one effect:*

$\forall r \in AR$, there exists a cause $e \in AE$ such that $(e, r) \in PL$ and an effect $e' \in AE$ such that $(r, e') \in PL$;

2. no subset of links in PL form a directed cycle.

According to Definition 1, an AR is connected to a minimum of two AE nodes through links, one from a cause AE and one to an effect AE . As such, the links to the AE s then represent the cause and the effect of the design decision represented by AR . Clause 2 of Definition 1 specifies that insertion of a link is not allowed if it would result in a directed cycle of links. Essentially, this means that the links maintain the integrity of the causal modelling whereby something cannot be the cause of itself, directly or indirectly.

Using Definition 1, a basic form of the model construct is $\{AE_1, AE_2, \dots\} \rightarrow AR_1 \rightarrow \{AE_a, AE_b, \dots\}$ where AE_1, AE_2 etc. are the inputs or the causes of a decision AR_1 , and AE_a, AE_b etc. are the outcomes or the effects of the decision. Figure 6.2 shows a UML representation of the AREL model of the relationship between a motivational input AE , a decision AR and a resulting AE . The cardinality in the relationship shows that the motivational AE and the resulting AE must be a non-empty set linked by the single decision AR . The *uniqueness* constraint in the diagram specifies that each instance of AE in the relationship cannot appear more than once.

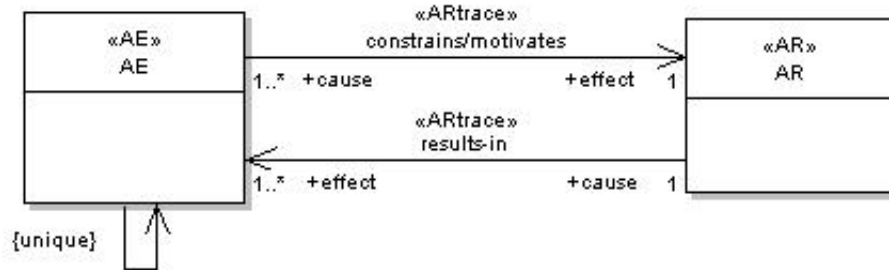


Figure 6.2: A Causal Relationship between AEs and an AR

The directional links of $\ll ARtrace \gg$ represents the causal relationships in Figure 6.2. An AE causes AR by *motivating* or *constraining* the decision, and the AR *results-in* an outcome AE by having a design rationale to justify the design. Conversely, an outcome AE depends on AR which in turn depends on an input AE . The causal relationship is the basis for forward tracing and the reverse is the implied dependency relationship which is the basis for backward tracing. Together they provide a means to navigate and explain the architecture design. This causal relationship simplifies and replaces similar relationships such as “*creates*” and “*achieves*” in argumentation-based methods (see Chapter 3). This is advantageous because the model becomes simpler and the complexity of traversal for knowledge retrieval is reduced.

An *AE* can be both an input and an outcome when it is involved in two decisions. As an input, it can be a requirement, a use case, a class or an implementation artefact. As an outcome, it can be a new or a refined design element. Since an *AR* contains the justifications of a design decision, designers can find the reasons of the decision and what alternatives have been considered.

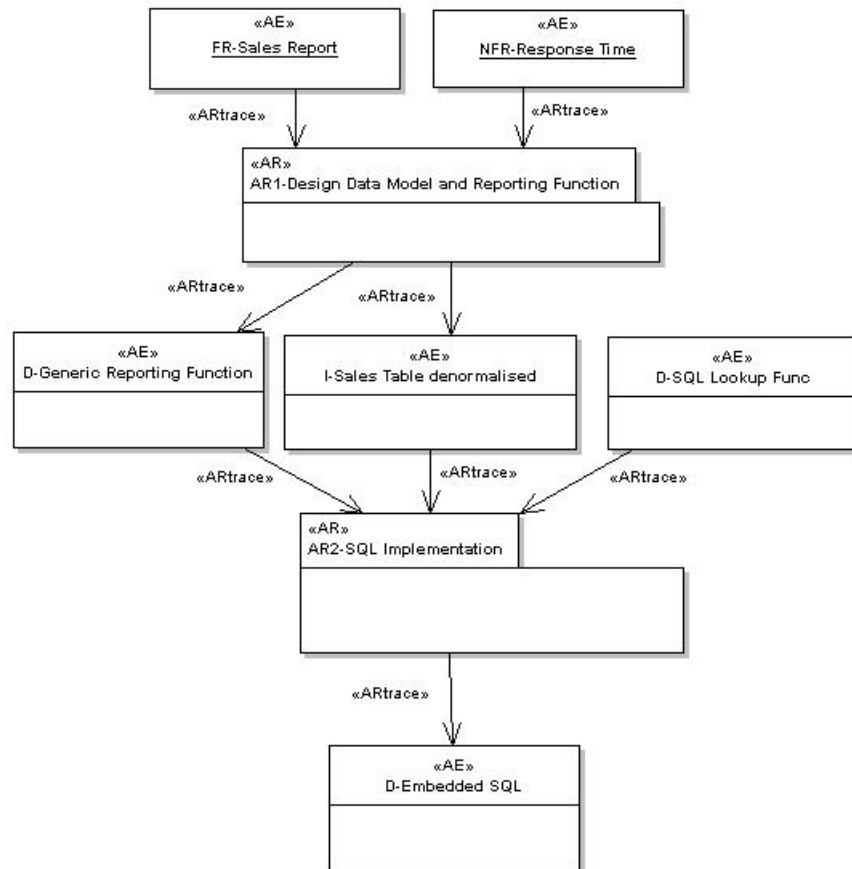


Figure 6.3: An AREL Diagram of a reporting sub-system, in UML

Figure 6.3 shows an example AREL diagram of a high-level design of a reporting sub-system in a typical sales and accounting system. The functional requirement *Sales Report* and the non-functional requirement *Response Time* are represented by two «AE» nodes. *AR1* justifies the design by de-normalising the Sales Table so that external joins can be eliminated to improve efficiency, and use a generic reporting function to produce standard report formats. As a result, two design objects *De-normalised Sales Table* and *Generic Reporting Function* are created. In the next part of the design, a decision is required to determine how the SQL statement is to be implemented. In the AREL decision graph, three input *AEs* are the required inputs to decision *AR2*:

1. *SQL Lookup* is a class to locate a SQL statement based on the required business function;
2. *Generic Reporting Function* specifies the format of report; and
3. *De-normalised Sales Table* supplies the data.

Together these *AEs* influence the decision *AR2* such that the optimal way to implement the design is to use embedded SQL statements in stored procedures. This design process continues as architects make design decisions by following design principles to satisfy requirements and create data models, design objects or implementation artefacts. The relationships between *AEs* and *AR* are connected by «ARtrace» links.

6.3 Architecture elements

In AREL, architecture elements *AEs* are artefacts that form part of the architecture design. They comprise the business requirements to be satisfied, the technical and organisational constraints on the architecture design, the assumptions that need to be observed and the design objects that are the results of the architecture design.

Architecture elements can be classified by a related set of concerns called viewpoints [70]. The purpose of such classification is to have a focus on the different perspectives of architecture design. Although there may be many ways to model viewpoints based on specific sets of perspectives [82, 63, 89], there are common viewpoints which are general to most software architectures. Using TOGAF as an example [164], we have selected four generic viewpoints (Business, Data, Applications and Technology) to classify *AEs*.

Figure 6.4 is a UML diagram which outlines the basic classification of the architecture elements and their viewpoints. The Business Viewpoint contains architecture elements such as functional and non-functional requirements, business, system and technology environments. These are the main drivers of the architecture design. Design objects are architecture elements classified by the Data, Application or Technology Viewpoints. An architecture element can be a motivational reason (i.e. an input) to a decision or an outcome of a decision or both.

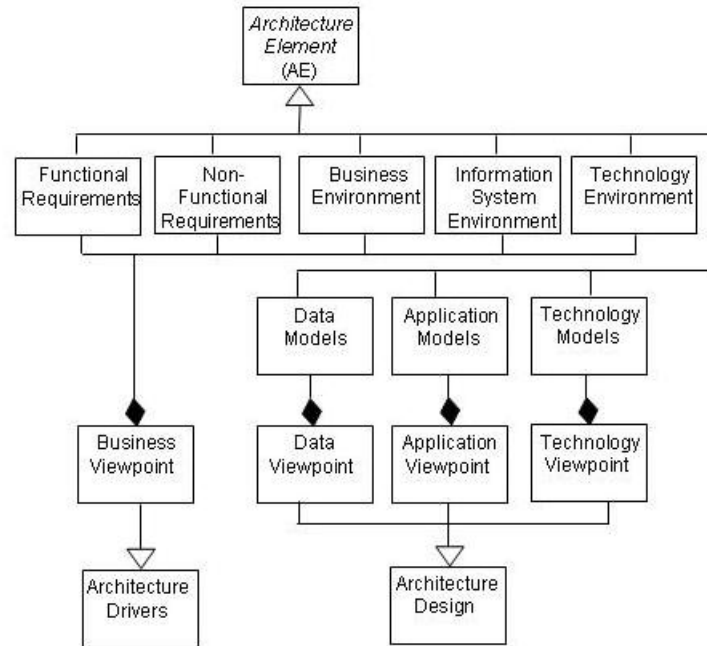


Figure 6.4: A Composition of Architecture Elements

Architecture Element as a Motivational Reason

1. Requirements - they are goals to motivate the design of the system. Examples are functional and non-functional requirements.
2. Assumptions - explicit documentation of the unknowns or the expectations provides a context to decision making.
3. Constraints - they are the limitations to what can be achieved. They may be of a technical, business, organisational or other nature.
4. Design Objects - the way one design object behaves may influence architecture design in other parts of a system by limiting the available design options.

Architecture Element as a Design Outcome

1. Design Objects - a design object is a result of an architecture decision to satisfy the motivational reasons.
2. Refined Design Objects - design objects can be refined as a result of a decision.

In the TOGAF framework, requirements are classified by the business viewpoint. We generalise the idea of the business viewpoint to include requirements and environmental factors. We call them *architecture design drivers*. We create five categories in the business

viewpoint to provide a logical sub-grouping based on how they influence the architecture design (see Figure 6.4). *System Requirement* is comprised of functional and non-functional requirements. *Environmental Factor* is comprised of business environments, information system environments and technology environments. Such classification allows architects to trace design reasoning to specific classes of root causes during analysis. For instance, an architect may want to analyse all non-functional requirements which affect a particular design object.

Architecture design elements are the results of the design process to realise and implement a system. They are classified by the following viewpoints: (a) Data Viewpoint - the data being captured and used by the applications; (b) Application Viewpoint - the processing logic and the structure of the application software; (c) Technology Viewpoint - the technology and the environment used in the system implementation and deployment. Their classification facilitates change impact analysis when architects want to focus on a specific aspect of a system. For instance, an architect may wish to find out how a change in requirement may affect the design objects in the application viewpoint.

6.4 Architecture rationale

In AREL, an *AR* comprises three types of justifications (Figure 6.5): qualitative rationale, quantitative rationale and alternative architecture rationale [159]. Qualitative design rationale (*QLR*) represents the reasoning and the arguments, in a textual form, for and against a design decision. These architectural arguments and assessments can only be represented in an informal and textual way. Quantitative rationale (*QNR*) uses the indices to indicate the relative costs, benefits and risks of the design options. This is an attempt to quantify design rationale for systematic comparisons and future analysis. These two types of rationale are described in sections 6.4.1 and 6.4.2 respectively. An *AR* also contains the Alternative Architecture Rationale (*AAR*) which documents the discarded design options (see Section 6.4.3). Often *AAR* provides an insight as to whether sufficient options have been considered when a decision is made. Some of these discarded design options may actually become useful as the context of the requirements and the project changes in the future.

The way *AR* is represented in AREL addresses a number of existing issues with the argumentation-based design rationale methods. Firstly, it is a cognitive burden to capture the complete explanations initially as designers who want to make use of this knowledge at a later stage most likely need not to replay the deliberation process as it was captured [53]. A second issue is that the design objects being discussed do not appear in the

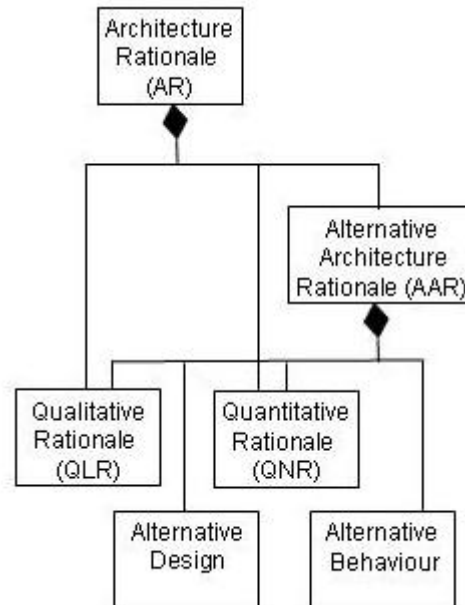


Figure 6.5: Components of Architecture Rationale

representation itself and are not linked to it in a defined way [122]. For designers who have to maintain a system, it is the design objects that are the focal point of investigation. For instance, a designer may ask “If a requirement is changed, which classes and data models might be affected and how?”. A third issue is that decisions are often inter-linked and inter-dependent but such relationships are implicit. A different approach to encapsulate and relate architecture rationale to design objects is adopted by AREL:

- Simplification - the *AR* entity records the key design issues, argumentation and design alternatives without explicitly capturing the design deliberation relationships. This approach simplifies the capture process by only capturing the results or justifications of the decisions without the overhead. It also provides a much simpler graphical representation when compared with methods such as gIBIS, PHI, QOC, DRL and SEURAT.
- Encapsulation - design rationale are encapsulated in an *AR*. In this way the decisions can be incorporated into the design process naturally to show the causal relationships between the design objects without over-complicating its representation. Should designers require more details about a decision, the details can be revealed by exploring the *AR* package. Our implementation in UML supports this feature (see Chapter 11 for details). This feature is more practical when compared with methods such as gIBIS, PHI, REMAP, QOC and DRL because the formal relationships offered by these methods are replaced by the qualitative rationale in a

textual form which is easier to enter and understand.

- Causal Relationship Chain - since *AR* acts as a connector between the cause architecture elements and the resulting architecture elements, we can construct a graphical representation showing direct dependencies between architecture elements and decisions in a chain. Such relationships can then be analysed to understand the design reasoning. This is an improvement over template-based design rationale methods because they do not link the design artefacts to design rationale in a chain of causal relationships. It is also an improvement over some of the argumentation-based methods such as gIBIS and QOC where design artefacts are not represented.

The encapsulation of architecture rationale in *AR* provides reasoning support to help architects understand a design decision and allow them to verify the decision. Relating *AR* to architecture elements *AE* provides the knowledge about the inter-dependency between architecture elements. It solves the problem of not being able to understand the design because of implicit assumptions and constraints.

6.4.1 Qualitative rationale

During the deliberation of design decisions, a number of factors have to be considered. Issues surrounding a decision need to be specified and alternative options to solving the issues are considered. Each alternative is assessed for its strengths and weaknesses. Some tradeoffs might be necessary in the decision making process. *QLR* is a template within *AR* to capture such qualitative design rationale. There has been substantial research on this subject by [94, 97, 23, 170, 3], we utilise some of their results as a basis for the qualitative design reasoning of *QLR*. Additionally, we have verified the design rationale elements through the survey reported in Chapter 5. The following information are contained in *QLR*:

- issue of the decision - the issue specifies the concern of this decision.
- design assumptions - they document the unknowns that are relevant to this decision.
- design constraints - the constraints that are specific to this decision.
- strengths and weaknesses of a design.
- tradeoffs - they document the analysis of what is a more appropriate alternative by using priorities and weightings.

- risks and non-risks - they are considerations about the uncertainties or certainties of a design option.
- assessment and decision - they summarise the decision and the key justifications behind the selection or exclusion of a design.
- supporting information - decision maker(s), stakeholder(s), date of decision, revision number and history.

The information contained in *QLR* provides the qualitative rationale to justify a decision. It helps architects and designers to understand the reasons and the justifications behind a decision.

6.4.2 Quantitative rationale

For most architecture design, the decision making process is based on the experience and the intuition of architects. Design tradeoffs are often not explicitly quantified. The lack of a quantifiable justification makes it very difficult to subsequently assess the quality and accuracy of design decisions using quality approaches such as CMMI. Quantitative rationale enables systematic estimates of the expected return of design alternatives. The expected return of a decision is represented by the architect's estimate of the Cost Benefit Ratio (CBR) at a decision point.

The *QNR* approach is based on three elements - cost, benefit and risk. It is measured by an index rating. The reasons for using an index instead of money terms are because (a) some of the assessment cannot be expressed in money terms for they may be intangible or difficult to estimate; (b) there may be multiple factors in which their benefits or costs values cannot be easily combined; (c) the comparison between design options can be made using the same scale; (d) it provides a uniform measure for reviews and verification. The quantitative measures comprises of the following items:

- Architecture Cost Index (ACI) - measures the relative cost of a decision with a rating between 1 and 10.
- Architecture Benefits Index (ABI) - measures the relative benefit of a decision with a rating between 1 and 10.
- Outcome Certainty Risk (OCR) - measures the risk that the outcome would not satisfy the intended goal(s) with a probability between 0 and 1.

- Implementation Certainty Risk (ICR) - measures the risk of not achieving the implementation with a probability between 0 and 1.
- Cost Benefit Ratio (CBR) - measures the ratio between expected benefits to the expected costs of the design decision

These estimates are used to assess the options at each decision point. They are the quantifiable justifications for selecting a design option. We have provided a brief introduction to *QNR* here. A detailed descriptions of their formulations and applications are described in Section 8.2.2.

6.4.3 Alternative architecture rationale

In terms of the argumentation-based design rationale approach, an alternative design is referred to as an option or a position. The association to support or refute an option is modelled explicitly. This leads to complicated decision models. The AREL model simplifies this by encapsulating the options and their arguments within the decision. In other words, zero or more alternative design options (i.e. *AAR*) are contained within an *AR*. *AAR* itself may contain entities such as arguments, design objects and behavioural diagrams (see Figure 6.5). This information allows architects to understand and verify the discarded design options, their relative merits and demerits, which have been considered in the decision process.

Both *AR* and *AAR* are implemented by the *package* entity in UML. An *AR* is a container of *AAR*. This multi-level structure hides the complexity of a decision and makes it easy for searching and retrieval. The multi-layered information is exposed only when they are required.

6.4.4 Avoiding cyclic decisions in AREL models

During the design process, architects may inadvertently cause a reasoning cycle to form as design decisions are incrementally made. For instance, this could happen when different teams of architects are designing simultaneously. Figures 6.6(a) and 6.6(b) show two cyclic examples which violate the AREL definition (Section 6.2). In Figure 6.6(a), we can see that if *AR3* was to be created and added to the AREL model, with *AE3* as an input and *AE1* as an outcome, this would create a directed cycle in the graph. The dotted arrows denote the illegitimate $\ll\text{ARtrace}\gg$ that would result in a cyclic graph, i.e. from *AE3* through to *AR3*, *AE1*, *AR1*, *AE2*, *AR2*, and back to *AE3*. Similarly, Figure 6.6(b)

depicts a cyclic graph created if *AR5* was modified to take *AE7* as an input, i.e. from *AR5* through to *AE6*, *AR6*, *AE7*, and back to *AR5*.

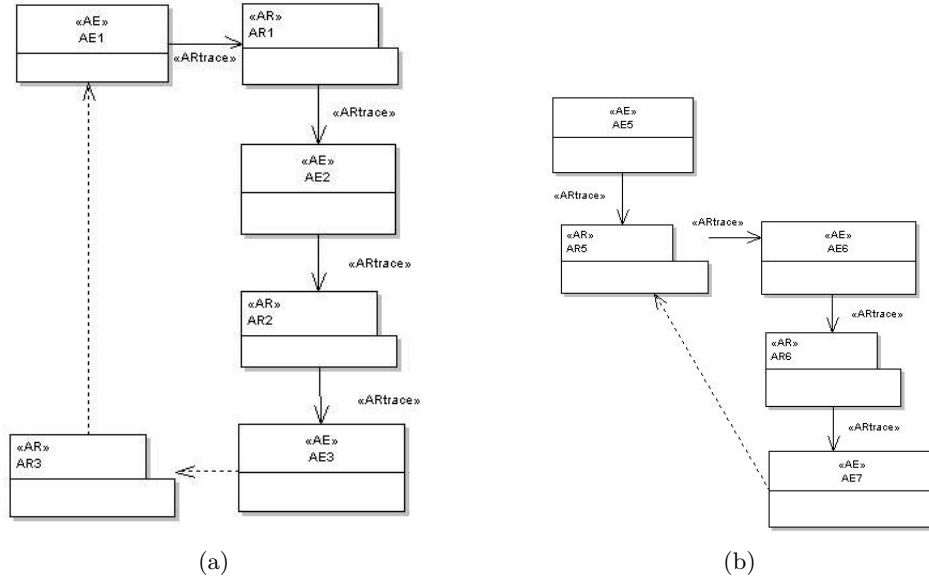


Figure 6.6: Illegitimate cyclic graph: (a) AE-cyclic case (b) AR-cyclic case

We do not allow cycles of design rationale links in AREL because they create ambiguity and inconsistency regarding the primary cause of decisions.² Using Figure 6.3 as an example, we consider a new requirement *FR-Support Custom Made Report Format* in Figure 6.7(a). We need to design a new element *D-Custom Report Design*. *D-Custom Report Design* would make use of the generic report template for customisation and some enhancements need to be made to *D-Generic Reporting Function* as well. As part of the decision, we use *D-Embedded SQL* as input to *AR3* because the design of *D-Custom Report Design* uses embedded SQL heavily. In this case, we have created a cyclic argument. Is *D-Generic Reporting Function* a primary cause or *D-Embedded SQL* a primary cause? The relationship is ambiguous.

Let us consider an alternative way to reason about the design. Figure 6.7(b) shows that the decision making process should take into account all the relevant functional requirements in *AR1*. It adds to the justification for sharing generic report templates between the two reporting functions. At this stage, the *AR3-SQL Implementation* is not the issue in focus. It is *AR2-SQL Implementation* which decides how *D-Embedded SQL* should cater for both requirements. There are two implications in the elimination of cyclic decisions:

²Furthermore, a cyclic model inhibits Bayesian Belief Networks (BBN) based change impact analysis. The application of BBN to AREL is discussed in chapter 10.

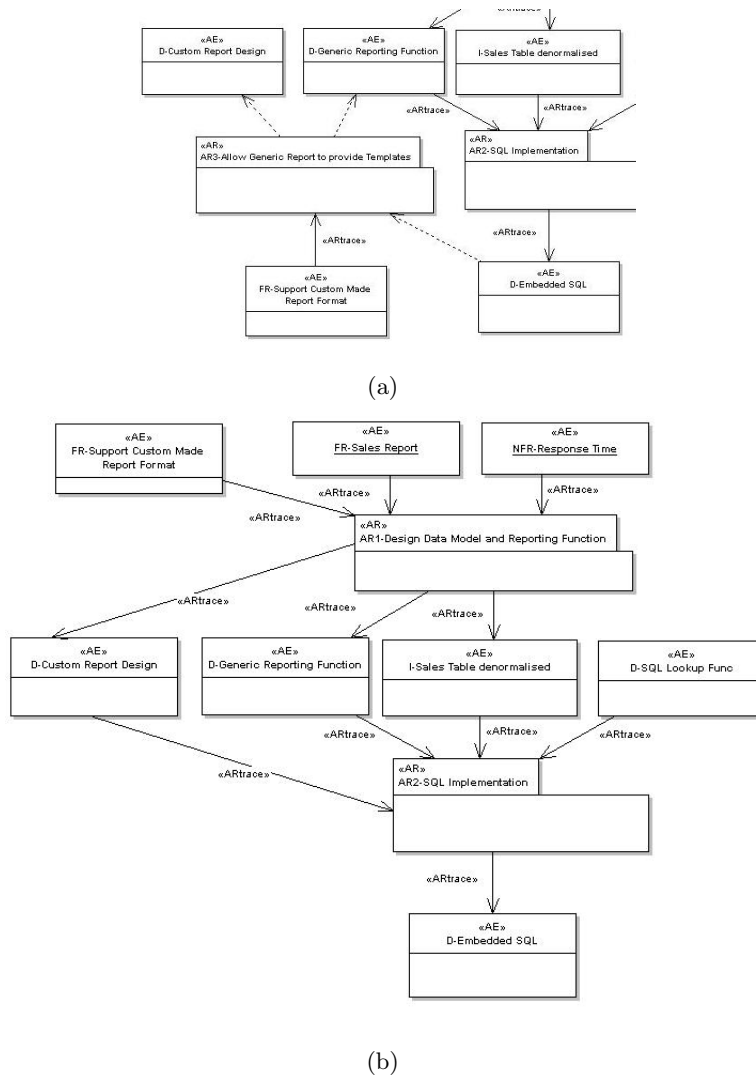


Figure 6.7: (a) Cyclic Design (b) Acyclic Design

- Identify the root goal of the design - the important goals behind a decision are identified. In this case, relevant requirements such as *FR-Sales Report* and *NFR-Response Time* should be considered (as shown in Figure 6.7(b)) because they affect the implementation of *D-Custom Report Design*. Figure 6.7(a) does not have these relationships. Should there be a change in the performance requirements, it would not be able to trace this change to *D-Custom Report Design*.
- Avoid ambiguity - if we were to modify *D-Embedded SQL* to support remote database access (in Figure 6.7(a)), it would be ambiguous and difficult to determine whether *D-Embedded SQL* or *D-Generic Reporting Function* is the cause of the decision relationship.

To prevent cycles from occurring in AREL, we provide an AREL Tool to carry out consistency checking to detect directed cycles (see Chapter 11). The AREL Tool warn architects about the directed cycles in a model. Human reasoning would be required to modify the structure of the architecture design reasoning to resolve such reasoning anomalies.

6.5 The extended AREL

Software architecture design evolves over time due to changes in the business requirements or in the business environment. As the architecture design evolves, the original design and their design rationale can be lost. An architect who is not exposed to the passage of events often cannot understand the convoluted architecture design due to past changes [157]. This may often prevent sound design decisions to be made during system maintenance and enhancements. To address this issue, we define an extended version of AREL called eAREL below to capture the evolution history.

Definition 2 *An extended AREL model eAREL is a tuple (AE, AR, PL, SP) , where AE , AR and PL are sets of architecture elements, architecture rationales and directed links, respectively, and SP is a bijective supersedence function between architecture elements or between architecture rationales, such that*

1. *let $AE_c \subseteq AE$, $AR_c \subseteq AR$ and $PL_c \subseteq PL \cap ((AE_c \times AR_c) \cup (AR_c \times AE_c))$ be the sets of architecture elements, rationales and directed links present in the current architecture design, respectively, then (AE_c, AR_c, PL_c) is the AREL model for the current architecture design;*
2. *let $AE_h = AE \setminus AE_c$ and $AR_h = AR \setminus AR_c$ be the sets of architecture elements and rationales stored for previous architecture designs, respectively, then the supersedence function satisfies $SP : (AE \rightarrow AE_h) \cup (AR \rightarrow AR_h)$, and it is a bijective function.*

In an eAREL model, both AR and AE have one current version and one or more historical versions. The current versions of AR and AE are denoted by AR_c and AE_c , respectively, and the historical versions of AR and AE are denoted by AR_h and AE_h , respectively. When an AR or an AE is superseded by a current version, the superseded elements are kept in superseded elements AR_h and AE_h . If multiple supersedence exists, they are linked in a linear list. This supersedence chain represents the direct supersedence relationship between different versions of architecture elements and rationales. The super-

sedence link of *AR* is stereotyped by `<<ARsupersede>>` and the supersedence link of *AE* is stereotyped by `<<AEsupersede>>`. Figure 6.8 shows the relationship in UML.

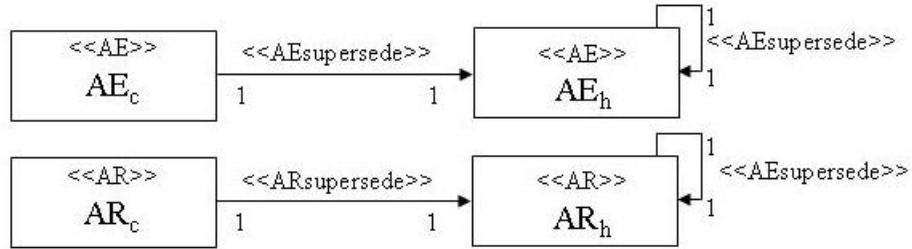


Figure 6.8: AE and AR evolution support in eAREL

In the eAREL model, the `<<ARtrace>>` links between superseded architecture elements and rationales are also kept to retain their relationships, i.e. $PL \setminus PL_c$. This supports a historical trace of previous architecture design decisions. The traceability of historical design and design decisions is discussed in Chapter 9.

6.6 A UML representation of AREL and eAREL

AREL and eAREL models are implemented using the UML notation [118]. They use UML *stereotype* to represent their semantics. We make use of *tagged values* in *stereotype* to capture information for AREL and eAREL. The choice of using UML for implementation is due to a number of reasons:

- UML is already very popular in software design. Architects can use the same representation and tool to carry out architecture design and record architecture rationales during the design process;
- UML has sufficient expressive power to support the rationale-based architecture modelling;
- UML is a standard graphical representation understood and used by many software development organisations;
- UML has a wide variety of tools which could be used to support the capture and the traversal of the model.

Our implementation is built on a UML tool Enterprise Architect [150]. It allows us to extend the semantics of the UML notation by using the *profile package* to define new

stereotypes and tagged values for capturing architecture elements and rationales as well as their relationships. This profile is downloadable from [153]. The details of the UML implementation to support AREL and eAREL are described in the following sections.

6.6.1 The architecture element stereotype

We define the AE stereotype as $\ll AE \gg$ to *extend* the UML constructs such as *object* and *class* to support architecture traceability. An extension of the UML construct means that additional attributes and characteristics can be added to the existing UML constructs to help design reasoning. The $\ll AE \gg$ attributes for traceability are implemented as *tagged values*:

- *elementID* - an unique identification of an *AE*.
- *elementVersion* - an integer number to identify a version of an element, the highest number is the most recent version. *elementID* and *elementVersion* together uniquely identify a version of an element. This is used in eAREL implementation.
- *elementType* - classification of the architecture viewpoints (e.g. business viewpoint). This is used for architecture tracing.
- *elementSubType* - sub-classification within an architecture viewpoint. For instance, the Functional Requirement within the Business Viewpoint. This is used for architecture tracing.
- *currFlag* - the latest version has the flag set to 'Y' and archived versions have the flag set to 'N'. This is used in eAREL implementation.
- *lastUpdateDateTime* - date and time of last modification.
- *author* - name(s) of the author(s).
- *documentLocation* - the location of any external documents that describe the AE, it could be an URL or a filename with document section references.

During architecture tracing, *elementType* and *elementSubType* can be used to limit trace results to those defined types that are of interest to architects. The *lastUpdateDateTime* and *author* tags are used for audit trail purposes. The *elementVersion* and *currFlag* tags support traceability of evolving *AEs*.

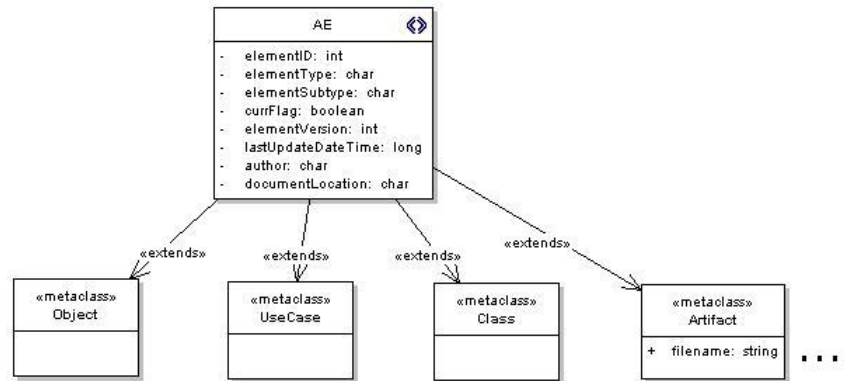


Figure 6.9: <<AE>> Stereotype to extend Architecture Drivers

To model the business viewpoint that contains requirements, environmental factors, constraints and assumptions, we use the <<AE>> stereotype to extend UML model elements *Class*, *Object*, *Artifacts* and *Use Case*. These extended UML model elements as shown in Figure 6.9 can create architecture models in different architecture viewpoints. For instance, a *Use Case* can capture system interaction, and *Artifacts* and *Objects* can capture requirements and constraints in the Business Viewpoint.

Architecture design elements that are represented by the Data Viewpoint, Application Viewpoint and Technology Viewpoint as shown in Figure 6.4 also use the <<AE>> stereotype to extend their UML meta classes. The UML design elements that have been extended include *Class*, *Object*, *Package*, *Component*, *Artifact*, *Table* and *Deployment*; and behavioural diagrams that have been extended are *Use Case*, *Communication Diagram*, *Analysis Diagram*, *Activity Diagram*, *State Chart* and *Sequence Diagram*. With the stereotyping, these UML constructs can capture the additional information as shown in Figure 6.9.

6.6.2 The architecture rationale stereotype

AR is implemented by a UML stereotype, denoted by <<AR>>, to extend the *Package* metaclass. It contains a number of attributes:

- *arID* is the unique identifier of an instance of *AR*.
- *currFlag* is a boolean flag to indicate whether the *AR* instance records a current or superseded rationale. Only the latest version of a decision has a truth value.
- *lastUpdateDateTime* - date and time of last modification.
- *author* - name(s) of the decision maker(s).

Figure 6.10 shows the `«AR»` and `«AAR»` stereotypes. Their implementation is built on the UML *Package* meta class. `«AR»` and `«AAR»` both contain a `«QLR»` and a `«QNR»`, where `«QLR»` and `«QNR»` contain the attributes that characterise the architecture rationale (see Section 6.4 for details of the attributes). An *AR* can contain zero or more *AAR* depending on the number of discarded design alternatives. If there is no design alternative, then there is no instance of an *AAR*. The containment of all relevant design rationale within an *AR* reduces the complexity in traceability without compromising the information it contains.

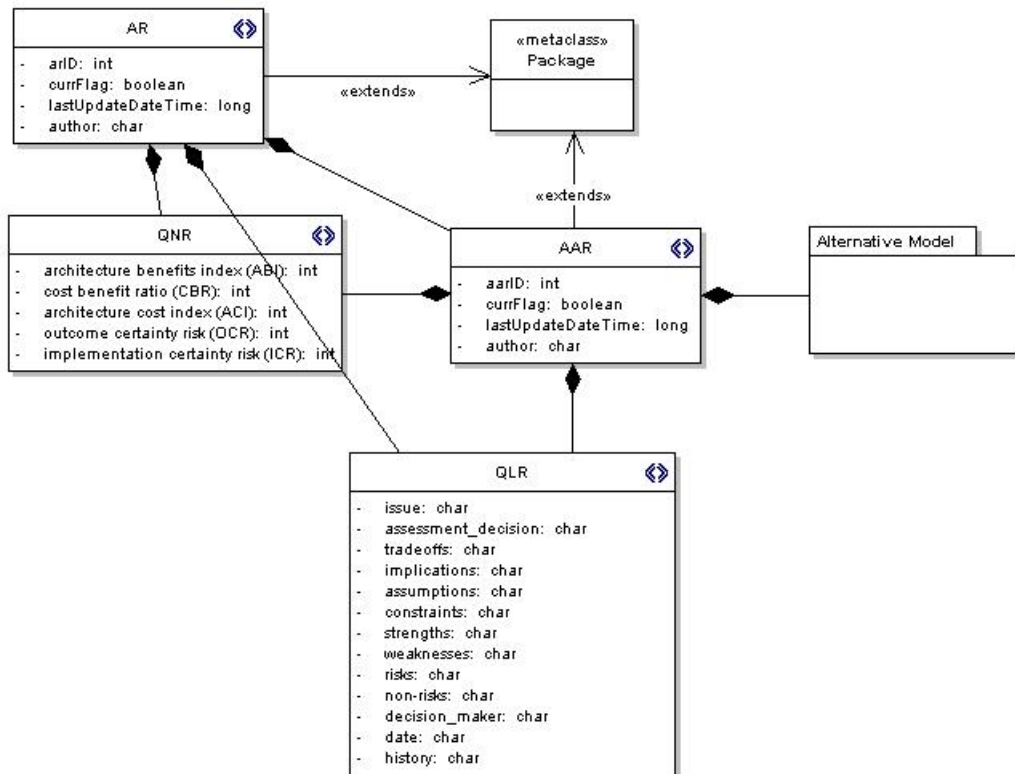


Figure 6.10: `«AR»` and `«AAR»` Stereotypes

The stereotype `«AAR»` provides a template to document alternative architecture designs that have been discarded. Since an *AAR* contains the element *Alternative Model*, which is implemented as a `«package»`, it can contain UML constructs such as use case, class diagrams, artifact, object or other UML diagrams. Architects can retain any documentation about the discarded design in the *Alternative Model* that might be useful.

6.6.3 The architecture trace stereotype

In Definition 1, we provide a directional causal dependency for tracing the links in AREL. We implement these links by a UML association stereotype `«ARtrace»`. The direction

of an `«ARtrace»` indicates causality. The reverse direction indicates the implied dependency. This relationship is used for traceability (described in Chapter 9) and impact analysis (described in Chapters 9 and 10).

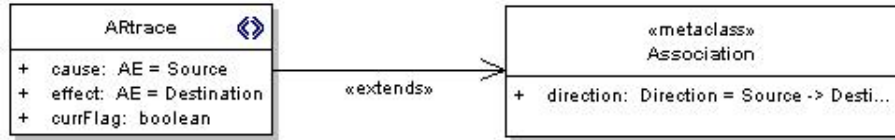


Figure 6.11: `«ARtrace»` Stereotype

Figure 6.11 shows the `«ARtrace»` stereotype which extends the UML *Association* relationship. Using a relationship *AE1-AR1-AE2*, for instance, there are two causal links or *ARtraces* depicted by *ARtrace1* and *ARtrace2*. *ARtrace1* has a *cause* of *AE1* and an *effect* of *AR1*. *ARtrace2* has a *cause* of *AR1* and an *effect* of *AE2*. The *currFlag* indicates whether this link is current or if it has been superseded.

6.6.4 The AE and AR supersedence stereotypes

Systems naturally evolve during the development and maintenance phases of their life-cycles. AREL provides the *as-is* view of the architecture design and decisions. A historical trace is provided by eAREL to capture evolving elements. Figure 6.12 shows the UML stereotypes `«AEsupersede»` and `«ARsupersede»`, that extend UML *Association* to link current *AE* and *AR* to historical *AE* and *AR*, respectively.

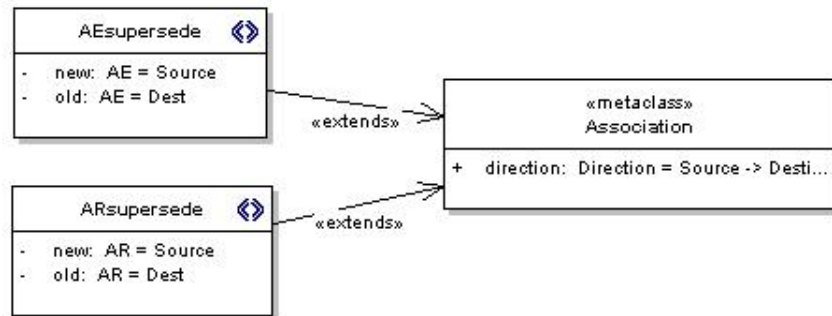


Figure 6.12: `«AEsupersede»` and `«ARsupersede»` Stereotypes

The `«AEsupersede»` stereotype provides an association between different versions of an *AE* over time. A new evolution of the architecture element is a new entity identified uniquely by the same *elementID* together with a new version number in *elementVersion*. An architecture element contains a history of all its previous versions through a chain of `«AEsupersede»` relationships. A current *AE* has a current `«ARtrace»` link with an *AR*. Superseded AEs are stored as historical records and they are linked to the current

AE by \ll AEsupersede \gg links. As mentioned in the last section, all \ll ARtrace \gg links in a superseded *AE* are made non-current.

Similarly, \ll ARsupersede \gg link connects current *AR* to superseded *AR*. Non-current or superseded *ARs* retain historical links (i.e. `ARtrace.currFlag=FALSE`) to *AEs* so that past \ll ARtrace \gg relationships are not lost. The replacement or the current *AR* would maintain the current \ll ARtrace \gg links (i.e. `ARtrace.currFlag=TRUE`) to all *AEs*.

The introduction of eAREL in the architecture model creates complexity in the model because multiple versions of *AR* and *AE* and their relationships are kept. However, it is possible to manage this complexity through proper tool implementation by selectively showing or hiding the information as required. When architects need to trace architecture evolution, the hidden relationships can be exposed through specific tool functions.

6.6.5 AREL well-formedness in UML

Given that AREL relies on UML extension, of which there is no support for checking model well-formedness, we require some methods to check the integrity of AREL so that any mistakes can be detected. In particular, the following well-formedness rules of AREL must be validated:

- Architecture decision - each architecture rationale *AR* must have an input *AE* and an outcome *AE*. A decision node *AR* can neither be a root node nor a leaf node in the AREL model.
- Architecture element - an architecture element *AE* cannot be linked to another architecture element *AE* without an architecture rationale *AR*.
- Acyclic graph - the graph must be acyclic.

The AREL Tool carries out consistency checking to ensure that the AREL model is well-formed. This is described in details in Chapter 11.

6.7 AREL usability

As discussed in Chapter 3, there are challenges in creating a design rationale method that can easily be used by practitioners. The existing design rationale methods that we have reviewed do not have all the required features (see Table 3.1). We intend to overcome these

6.8. Summary

issues with AREL. In Table 6.1, we recapitulate these usability features and describe how AREL has been implemented to fulfil them.

Table 6.1: An Analysis of the Usability Features in AREL

	AREL	Implementation Notes
Effective capture of design rationale	Yes	Design rationale can be captured in ARs during the design process, and AEs are captured as part of the architecture design.
Effective communication of design rationale	Yes	AREL uses a UML graphical representation to communicate the causal relationships between design rationale and design objects.
Design Artefact Focus	Yes	AE can represent requirements, assumptions, constraints, and design objects. They are classified by different architecture viewpoints.
Traceability and Impact Analysis	Yes	AREL provides tool support to carry out tracing and impact analysis.
Comprehensive Design Rationale	Yes	AR contains qualitative and quantitative design rationale as well as alternative design options.
Common Tool Support	Yes	A commercially available UML tool can be used to support AREL.

The way AREL is structured is simpler than the argumentation-based methods because design deliberations are not captured. Instead, design rationale are encapsulated in an AR node and the various kinds of design reasoning are captured with templates. Therefore, it is simpler for architects to record and use them. The usefulness of the design rationale captured by AREL is further explored in an empirical study in Chapter 7. Other usability features are discussed as follows: traceability in Chapter 9, impact analysis in Chapter 10 and tool implementation in Chapter 11.

6.8 Summary

In this chapter, we have introduced the AREL model to represent the architecture design rationale, architecture design elements and their causal relationships. We use a conceptual model to describe the high-level requirements of AREL. We distinguish between motivational reasons and design rationale. The AREL model is characterised as follows:

- Architecture element - an architecture element is an input and/or an output of an architecture decision. It can be a motivational reason to a decision, and it can also be an outcome of the architecture decision. Architecture elements are classified by viewpoints.
- Architecture rationale - an architecture rationale represents the reasons behind an architecture decision. It contains the qualitative rationale, the quantitative rationale and alternative designs.
- Directional link between architecture elements and rationale - architecture elements that are used as inputs into the decision and the architecture elements which are created as the outcome are linked to the architecture rationale

The AREL model is extended by eAREL to support the capture of design history. Both AREL and eAREL are implemented using UML. The AREL representation is an improvement over the argumentation-based design rationale methods because it simplifies the design rationale representation, design rationale therefore are easier to capture and communicate, and UML support is more readily available. Using the AREL model, applications such as design reasoning, traceability and impact analysis can be carried out.

Chapter 7

A case study of using the AREL model

The case study described in this chapter serves two purposes. It presents an industry case to demonstrate the application of AREL and it uses an empirical study to compare the reasoning capabilities of AREL with traditional design specifications. We choose to use an industry case because laboratory cases do not have the architecture complexity to demonstrate the AREL model. The system used in the case study is a central bank electronic payment system called Electronic Fund Transfer System (EFT). The EFT system processes high-value payment instructions between a central bank and the participating member banks to transfer and settle payment instructions. It is the backbone of the financial system in the South China region. As such, the EFT system has to be reliable, secure and efficient. The EFT system was operational from 1998 for 6 years. It is administered by the Guangzhou Electronic Banking Settlement Center, which is a division of the central bank in China.

The EFT System took about two years to design, develop and test, employing over thirty designers and developers. At the time of its development, China was in the process of developing a national payment system standard, so the EFT system architecture had to be adaptable to changes based on the national standards. The system had extensive design specifications to document how it had been designed. In this case study, we have selected a number of key requirements in the functional and non-functional areas to report. We have retrospectively implemented the AREL model to document the design decisions for the EFT system. This retrospective capture of design rationale is possible because the author was the lead architect of the system.

In order to validate if the AREL model could facilitate the understanding of design

reasoning, a comparison of the original specifications and the AREL model was made in consultation with payment system design experts. The experts who participated in this study are familiar with the design of electronic payment systems. The original specifications were presented and the experts were asked about the design of the system. Answers were obtained based on the design specifications and their prior knowledge of payment systems. These were compared with the answers obtained after the AREL model had been presented later to check if additional understanding could be gained.

In this chapter, we discuss the case study in Section 7.1. In Section 7.2, we use the empirical study to validate the AREL method.

7.1 The EFT system

The People's Bank of China Guangzhou branch (PBC-GZ) is a central bank branch which is responsible for the financial monitoring and inter-bank payment and settlement of the financial centre Guangzhou and its neighbouring cities. The Electronic Fund Transfer System (EFT) was built by the author and his team to transfer and settle high value payments between all the commercial and specialised banks in the provincial and neighbouring cities. It serves an area with a population of over ten million people in Southern China. The EFT system also acts as a gateway to connect these local banks to the national payment network. The EFT system comprises the High Value Payment System (HVPS) and Settlement Account Management System (SAM). The backbone of the system is the architecture design which provides the processing services infrastructure for the on-line fund transfers between the member banks and the central bank.

Being a mission critical system, the EFT system was carefully designed, thoroughly tested and documented. Although it comprises extensive specifications and documentation, the design rationale was either buried in the text of the specifications or they were not captured. Therefore it might be difficult to understand its intricate design when designers have to analyse the system to make changes.

7.1.1 The EFT architecture overview

In this case study, we present the system and software architecture of selected key areas of the EFT system. In particular, the message processing and exchange mechanism is described. This area of the architecture is selected because it involves intertwining non-functional requirements such as performance, reliability, recovery and security. Due to confidentiality and security reasons, detailed descriptions of the internal mechanisms of

how each component is implemented is not disclosed.

Figure 7.1 is the high-level use case of the system. The system is connected to all member banks in Guangzhou and its neighbouring cities. Local payment instructions which can be settled within the area are processed by the EFT System. Non-local payment instructions such as remittance of funds to other parts of China are forwarded to the Electronic Interbank System (EIS) system, which is a satellite link to the national payment centre in Beijing. The EFT system interface to the China National Automated Payment System (CNAPS) system was part of the plan but it had not been implemented.

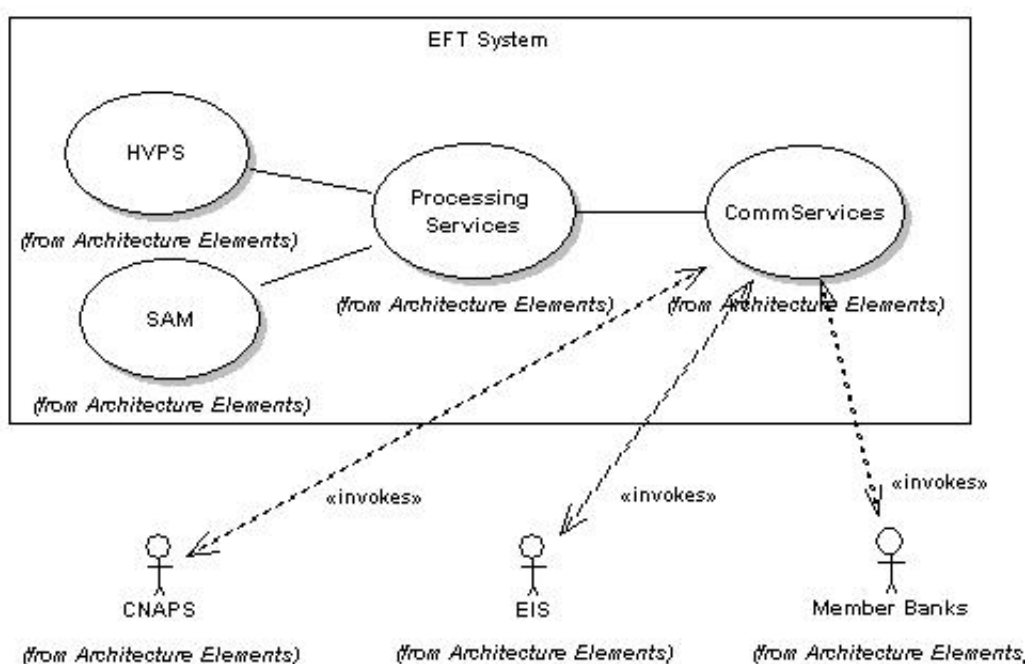


Figure 7.1: A Use Case of the EFT System

A payment instruction is either a credit instruction to pay or a debit instruction to request payment. Inter-bank payment instructions are settled by the central bank. It means that the central bank acts as an agent to ensure that participating member banks have enough funds to cover their financial positions. A key application of the EFT system is the High Value Payment System (HVPS) which is a real-time gross settlement application. It settles high value payment instructions in real-time to ensure banks could settle their financial obligations. This is crucial for the stability of any financial system by mitigating systemic risks.¹ The Settlement Account Management (SAM) is an accounting system which keeps track of the balances of member banks' settlement accounts and monitoring settlement activities.

¹Similar Real Time Gross Settlement (RTGS) systems are implemented in various countries such as Australia [133].

The processing services and the communication services are the backbone of the EFT system. They handle the transmission and the processing of payment messages. Since the EFT system was intended to act as a gateway to the EIS system and the CNAPS system, multiple messaging protocols support are required. In the case study, we illustrate the processing services and specifically the payment message processing layer to outline some of its architecture design and design rationale.

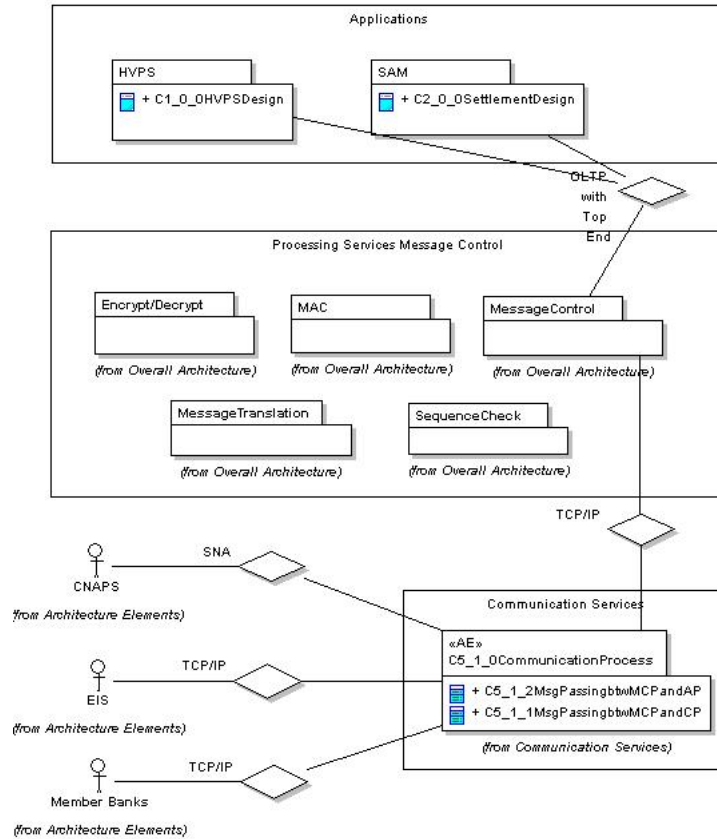


Figure 7.2: Processing Services of the EFT System

Figure 7.2 illustrates the key functionality of the processing services. External parties such as member banks and EIS systems connect to the EFT system using different communication and messaging protocols. The communication service layer is responsible for handling different communication protocols that are used by different external parties. Messages are passed onto the processing services layer where a payment message would be decrypted, checked for authenticity and processed. When a message has passed all the validation checks, it is passed to the central bank applications for payment queuing and settlement.

7.1.2 Fault-resilient support

The selection of the system and software architecture platform is one of the most fundamental architecture issues. A key driver of an electronic payment system is the *reliability* of the system. The risk of having an unreliable payment system would be damaging to the infrastructure of the financial system as well as the reputation of the central bank. This issue was paramount in the the architecture design of the EFT system.

Figure 7.3 shows the reasoning support for the selection of the architecture platforms to support the no single point of failure feature in the system (*R2.5.4*). There are four key issues to be addressed in this example.

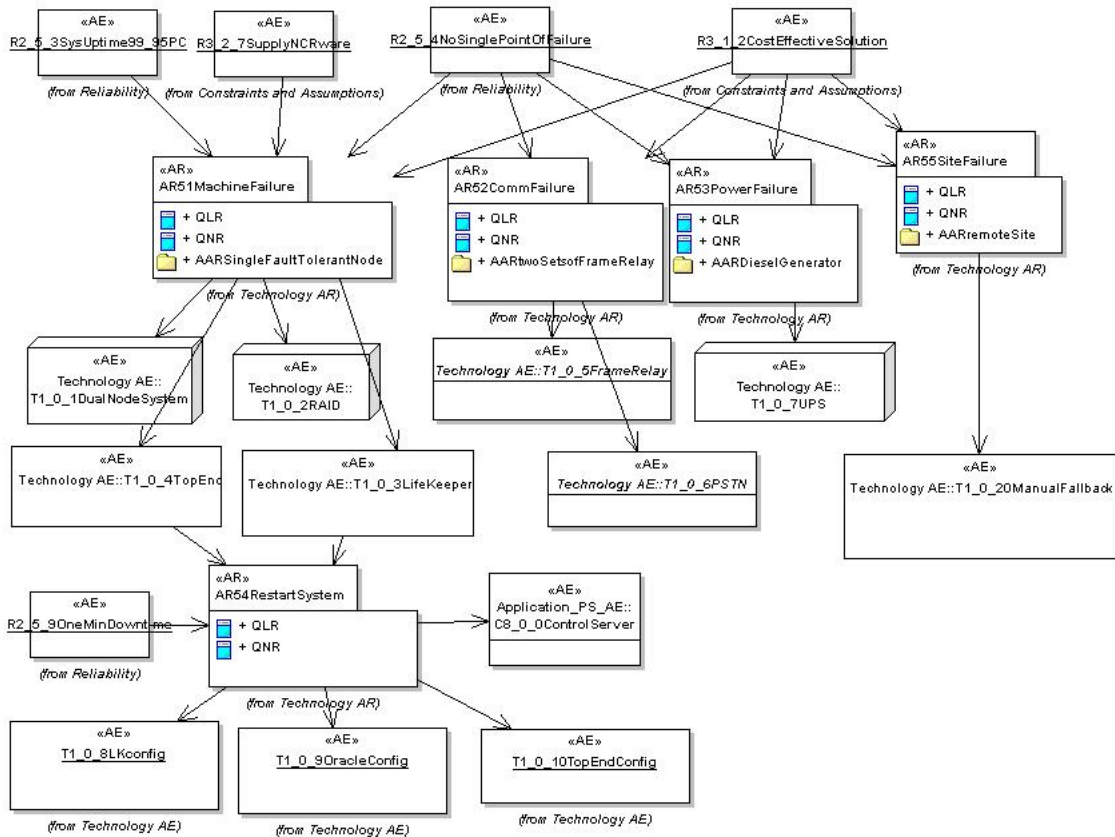


Figure 7.3: Decisions that Support Fault-resilient architecture

The first issue is the choice of a system that provides continuous processing with little chance of failing. There are two possible choices for a reliable machine. One of them is a fault-resilient system where there is always a system standing by to take over if a system node fails. *AR51* was a decision to choose this option because the proposed platform had fulfilled the reliability requirements of the central bank. Another option was to use a fault-tolerant system which had in-built backup processing modules. This alternative was

not chosen because of cost considerations. As a result of the decision, other associated platform products were required to maintain the 99.95% uptime. They included a dual-node UNIX configuration (*T1_0_1*), a set of disk array with backup controllers and RAID configuration (*T1_0_2*), middleware Top End² which load-balance services across multiple nodes (*T1_0_4*), and a node failure detection and switch over mechanism (*T1_0_3*).

Another reliability requirement states that the system, during market opening hours, cannot be down for more than a minute. Therefore the platform has to work coherently in order to provide such guarantee. The system needs to be designed and configured to recover from any failures within a short time. Decision *AR54* investigates a combination of recovery strategies using the platform environment. As a result, the topology of the servers monitored by Top End must distribute key processing services across both nodes to maintain services if one of the nodes fails. The LifeKeeper software checks the health of the system at a more frequent interval and would restart Oracle and key services if a system failure is detected. The EFT system provides the Control Server to carry out recovery at the payment transaction level.

A second issue is network reliability. At the time of development, broadband communication was not available in the city. The most reliable communication service was a frame-relay link. Therefore, this was chosen for network communication (*AR52*). A back-up communication service is to allow dial-in from member banks through the public switched telephone network (PSTN). An alternative option that was discarded was to use another frame-relay line as backup, but it was deemed uneconomical and the same means of communication is also more risky because the backup frame-relay line might fail as well.

A third issue is power failure. This possibility is almost imminent as power supply failures and power surges were quite common in Chinese cities. There are two alternatives to provide secondary power supply. One option was to use uninterruptible power supply (UPS) which was the chosen decision (*AR53*). Alternatively, power generator could be used but this would require a higher budget that could not be justified.

The final issue is to cater for natural disaster such as earthquake or fire which could damage the entire processing centre. One alternative was to have a remote site which could take over processing. At that time, there was no budget to allow for such an option, therefore the contingency was to have manual procedures to cater for such situation (*AR55*).

²Top End is a NCR middleware product that supports two-phase commit protocol in transaction processing and it monitors distributed processing

7.1.3 Payment messaging

When designing the architecture, one of the fundamental decisions was the scope of responsibility of a payment process. A single process (i.e. an UNIX process) could be responsible for all of the application processing, message processing and communication. It could also be divided into multiple processes which work together collaboratively.

Figure 7.4 depicts the decisions in the EFT system. It was decided that the Message Control Process (MCP) should be an independent process (*AR6*) because it has to support multiple message formats simultaneously (Requirements *R2.1.3* and *R2.8.1*). Most important of all, this is what the national payment standard had dictated (*R2.1.1* and *R5.1.0*). It was envisaged that the EFT system would evolve into the national payment system (assumption *R3.2.6*). This assumption formed a constraint that basically influenced the design philosophy of the system. As such, the architecture design of EFT was to cater for a software product that could be configured to suit different payment applications instead of a single customer solution.

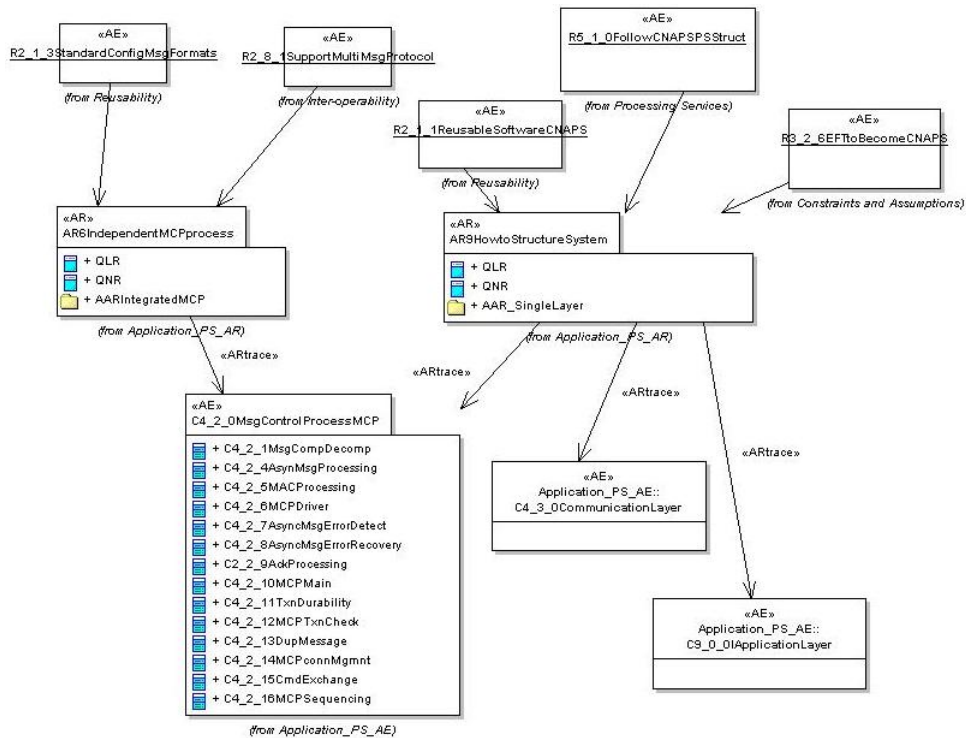


Figure 7.4: Message Control Processing

As a result of decision *AR9*, two additional layers of processing were required. They were the application layer (*C9.0.0*) and the communication layer (*C4.3.0*). These layers of software would exchange information with MCP (*C4.2.0*) through interfaces.

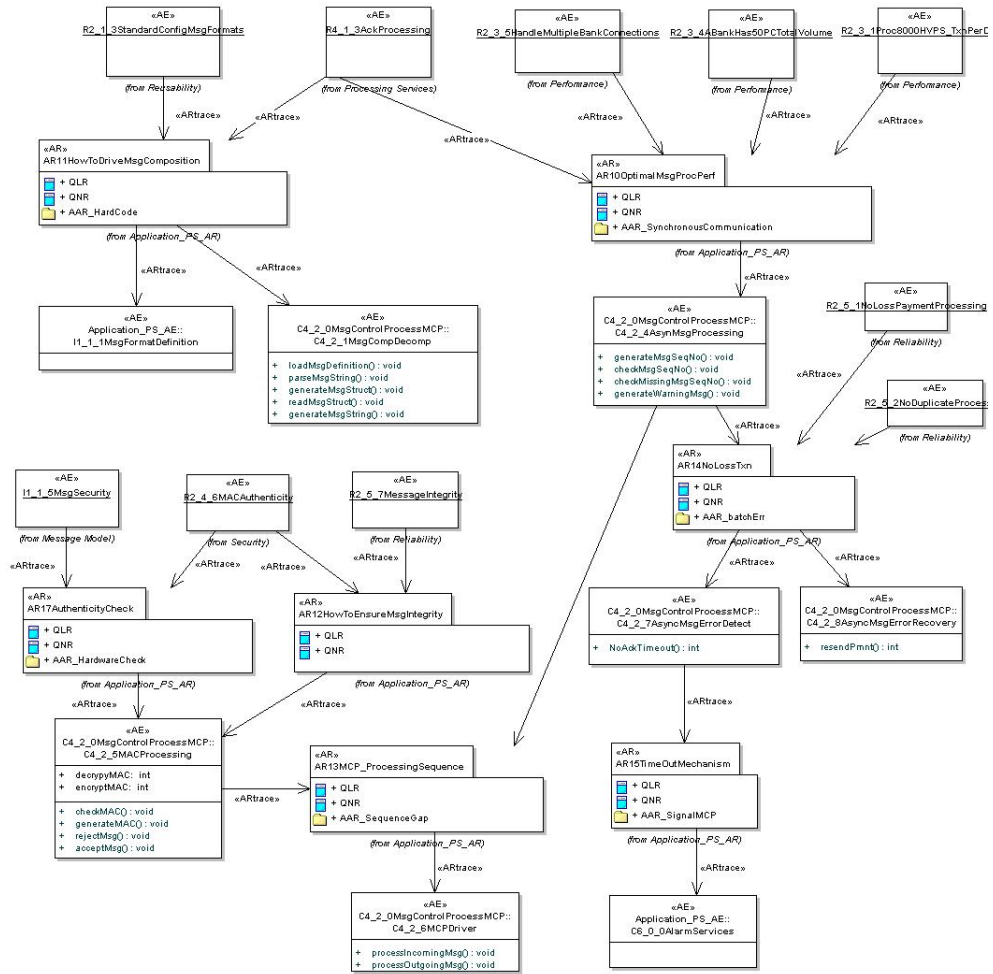


Figure 7.5: Intertwined Issues in the Architecture of MCP

The design of the MCP process involved many requirements. Figure 7.5 is a diagram which illustrates some of the key decisions. In the design process, one of the key issues was the way payment messages are to be composed and decomposed. At the time of the design, an initial draft of the national payment message standard was specified in the Request for Proposal (*R2.1.3*). This format was used as a basis for the EFT system since the EFT system was aimed at becoming a product which could be reused at the national level. As such, it was anticipated that the message format would be adaptable. As a result, the design of the message format was entirely definition-driven instead of hard-coding. This decision justifications are contained in *AR11*.

The message exchange protocol was another key decision which had to be made. Requirement *R2.3.5* specifies that a bank can have multiple connections to the payment processing centre. This is so that large volume of transactions can be sent through multiple bank branches settling through the same settlement account. There were two options

to address the issue: asynchronous message processing and synchronous message processing. Either model has technical and implementation constraints. An asynchronous model implies that payment messages can be out of sequence and the system needs to process and recover them even when the payment messages are out of sequence or missing. The disadvantage of such a design is that the processing logic would be more complicated but the upside of the design is its reliability and flexibility.

A synchronous model implies that a process would be blocked for incoming messages. Thus, dual-channels would be required to support bi-directional message exchange because both the member banks and the EFT system can initiate payment messages. This model is easier to implement but the performance was considered to be poorer. After weighing the pros and cons of the two models, the asynchronous model was chosen for its support of better performance and reliability (*AR10*).

The selection of the asynchronous messaging model means that a number additional design requirements to support this model must be in place. Therefore, new constraints had automatically been introduced to the architecture design. These constraints were (a) messages need to be sequenced to guarantee that none are missing during transmission or processing; (b) a legal acknowledgement mechanism needs to be in place to ensure that both parties of a payment transaction agree that a payment instruction has been sent and received properly. So in decisions *AR14* and *AR13*, message sequencing and missing payment message detection are decisions to guarantee message integrity.

A payment acknowledgement should arrive shortly after a payment has been sent. But acknowledgement may be missing due to various reasons such as a network error. So there ought to be a timer function to alert the processing unit if an acknowledgement does not come on time. *AR15* is a decision to implement an Alarm Service to notify any overdue acknowledgement.

AR12 is the justification for checking the integrity of a payment message, i.e. the payment message has not been tampered with. This is to ensure that no one has been able to modify the payment message during its transmission. *AR17* is a decision on how to check the authenticity of the message to assure the bank that a message is authentic. This mechanism forms the basis to prove the legal obligations of participating banks. Since the sequence number in a payment message is part of the authenticity check, it requires the design of the *C4.2.6* to check for duplicating and missing sequence numbers. The reasoning and implied constraints are contained in *AR13*.

In this example, security, performance and reliability concerns were intertwined in the EFT architecture design. The set of decisions are inter-related because they mutually affect each other. The performance consideration of the asynchronous model created a need for

sequence checking, which inter-relates with the security mechanism. It also created the need for a time-out mechanism, which inter-relates with the reliability of messaging.

7.1.4 Transaction integrity

One of the key issues in designing payment systems is to ensure that no payment transactions would be lost (*R2.5.1*) and no payment transactions would be processed more than once (*R2.5.2*). In order to achieve that, when payment messages are received, they would be captured in a durable storage and tracked during their processing life-cycle. As such, system failures could not cause any loss of payment transactions.

Figure 7.6 is a sequence diagram showing the life-cycle of a payment message from the sending bank to the receiving bank through the EFT system. When a message is passed between two entities such as a member bank or a process within the EFT system, there ought to be some guarantee that the message has been received properly when the communication takes place. For instance, when a member bank exchanges payment messages with the EFT system in either direction, acknowledgements would always be returned to confirm the receipt of messages.

As shown in Figure 7.6, a sending bank sends a payment message. The message is acknowledged by the Communication Process (CP) connected via TCP/IP to the sending bank. CP then forwards it to the Message Control Process (MCP) using the middleware Top End. Top End commits the message to the message queue as well as the Oracle database in a two-phase commit protocol so that the message is consistent and recoverable in case that the system fails. It guarantees the ACID state of the payment message. MCP takes over payment processing to authenticate and validate the payment message. Once the payment message checks out, it is passed to the HVPS application server for processing. When the payment is settled by HVPS, two notifications are sent, one to the sending bank and the other to the receiving bank.

Unless transaction processing or some reliable handshaking is used between processes, there is a possibility that a payment transaction can be lost when the system or communication fails and the payment message would end up in a “zombie” state that is not recoverable. Therefore, there is a need to architect the system to overcome this issue. Figure 7.7 is a diagram showing the design rationale to support payment transaction durability and recovery.

Two key requirements that specify the no loss of data (*R2.5.1*) and no duplicate of data (*R2.5.2*) are the main drivers of the architecture design. Together they are the common constraints on how payment messages could be processed. A number of design

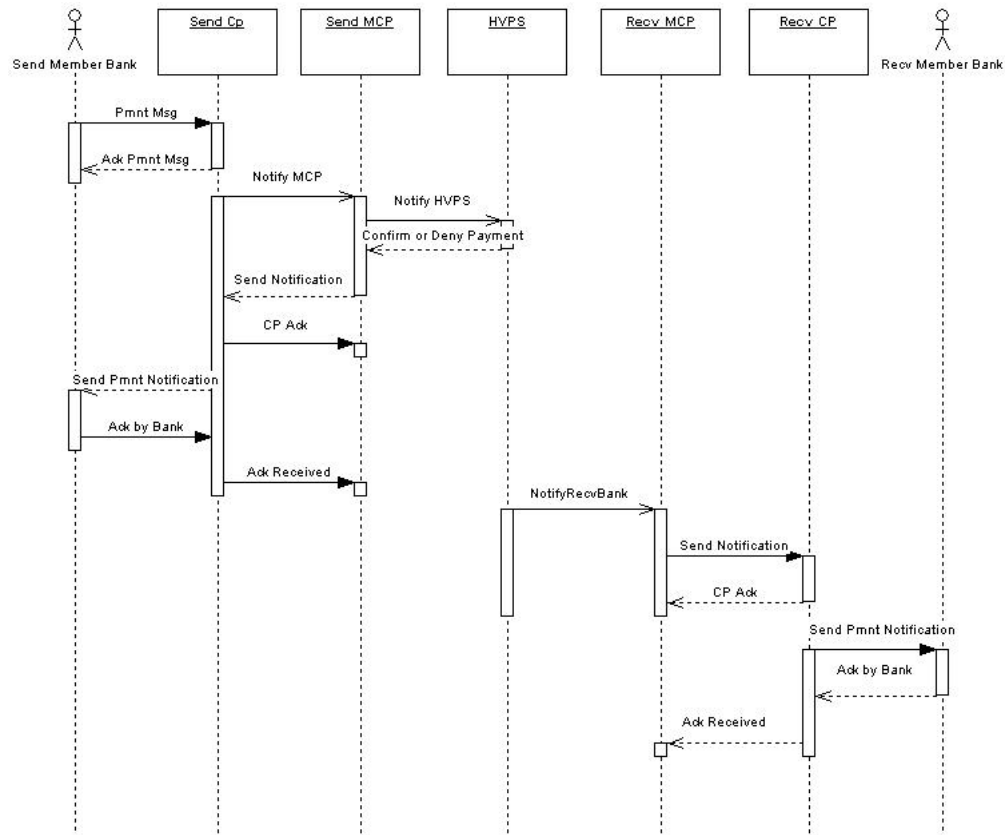


Figure 7.6: General Payment Message Processing Sequence Diagram

issues have arisen because of these requirements.

The first issue is to deal with missing messages between a member bank and the EFT system. The two requirements *R2-5-6* and *R2-5-8* specify that payment messages should be retransmitted when no acknowledgement has been received. In this case, we have to devise a mechanism to detect missing acknowledgements in real-time. In *AR19*, a decision was made to use an alarm clock for sending a timer event to the server to check for such scenario. This is the most efficient way to handle processing because the alternative would be to poll a remote system regularly. A delay of the acknowledgement might be due to a slow response of the remote system. However, it might also be a loss of the original transmission. We err on the side of duplicate transmission rather than missing payment by resending the original message. As a result, duplicate payment message would be ignored if it has been received more than once (*AR21*) and this mechanism uses a database table (i.e. *I2.0.1* MsgFlowLog table) to record the processing statuses of payment messages.

The second issue is the failure of the machine or the software service during processing. In this case, all servers must be able to recover from their previous states. Decision *AR20*

7.1. The EFT system

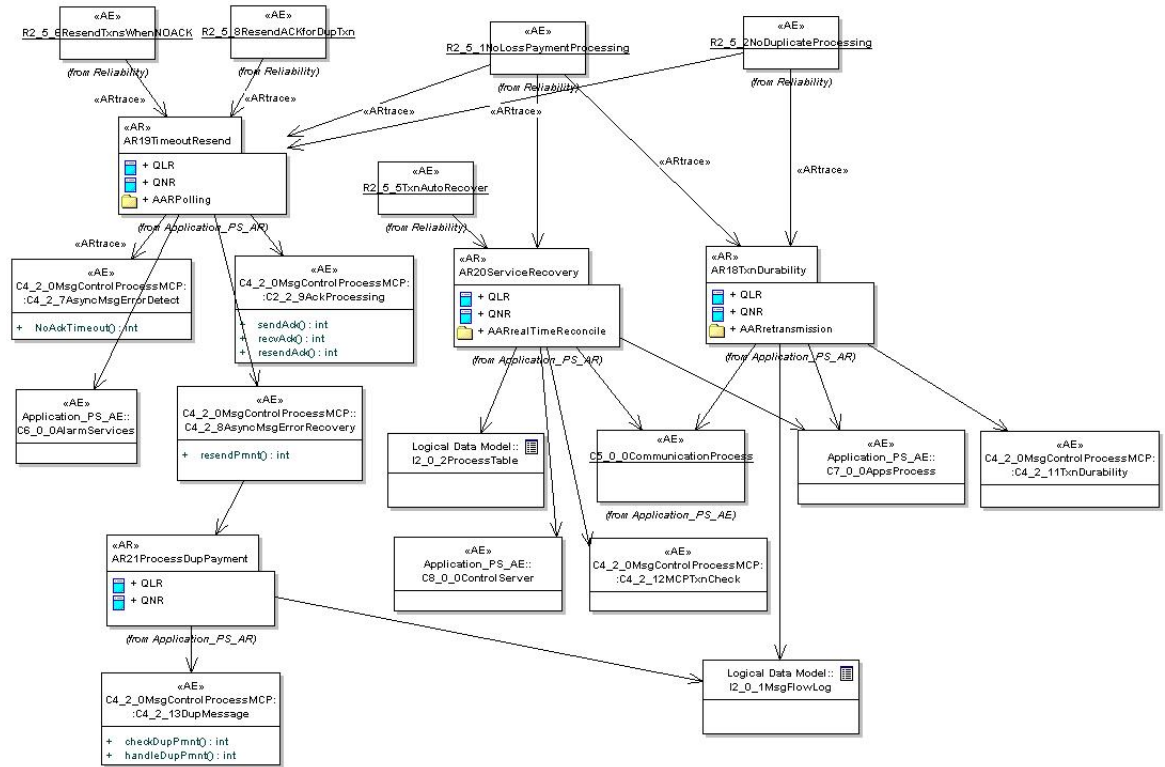


Figure 7.7: Decisions to Support Transaction Recovery

is used to justify a method which is most appropriate for such processing. In this decision, all services must have their status logged in Oracle in a Process Table *I2.0.2*. If any one of them terminates exceptionally for any reason, it could recover when brought back to service again. This is to ensure that any abnormality would trigger a system self-check to ensure that there are no left-behind payment transactions. As a result of the decision, the Control Server has to be created to coordinate such check and recovery activities. When the servers recover, they would investigate the incomplete units of work at the time of their failure and recover from that point. This decision is related to the next issue of how payment messages are kept in a consistent and durable state.

The third issue is to ensure the database ACID properties of a payment message. A server such as MCP could be in the middle of a transaction when it is interrupted. In order to ensure the atomicity of the transaction, a mechanism is required to support persistent inter-process communication (IPC). It means that no transaction could be lost during process-to-process communication. The decision was to use a two-phase commit where the transaction is logged in the MessageFlowLog Oracle table (*I2.0.1*) capturing the state of processing as well as putting the payment into a message queue for the next process in line. This will ensure that the payment transaction can never be in an inconsistent state. An example is the communication between MCP and HVPS.

In this example, we can see that the reliability requirements are realised by a number of related design decisions. The design rationale for these decisions are intertwined in that they influence each another in the implementation of design elements. For instance, the `MsgFlowLog` table (*I2.0.1*) is a result of two decisions (i.e. *AR18* and *AR21*), and those decisions are based on common requirements *R2.5.1* and *R2.5.2*.

7.1.5 Specialised message control process

In the architecture design of the EFT system, a consideration was given to the processing scope of MCP. Should a MCP process be a generic single-instance server to process payments for all the banks, or should there be multiple instances of MCP in which each instance is responsible for a single bank only? In the former case, a generic MCP process would take payment messages from any bank for processing. In the latter case, a MCP process is dedicated to a single bank connection.

Figure 7.8 shows the design reasoning of the decision. The inputs into the decision require that each message exchanged between the member bank and the system must be uniquely identifiable. The assumption is that the bank must have logged on successfully before payment messages can be exchanged. This assumption puts a constraint on the design such that payment messages cannot be exchanged if the bank has logged off, either voluntarily or by the EFT system.

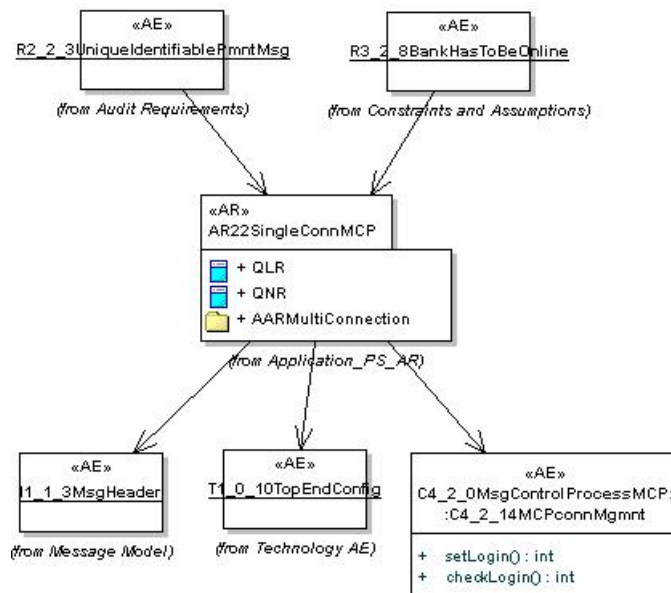


Figure 7.8: MCP Connection Design

Therefore, in assessing whether MCP could process all payment transactions, there is a need to check if the bank is still legitimately logged on to the EFT system when the transaction is received. *AR22* provides two alternative solutions. If there is a dedicated MCP server for each connection, as documented in the *QLR* in the *AR*, the server process can keep the login state in memory, hence perform much better because the login state is already kept. Alternatively, if MCP is generic, then every payment that is processed by it would need to be checked for bank login state. This is achieved either via the database or via a central authorisation server. This is documented in *AAR*. An individual MCP instance which is responsible for a dedicated bank connection gives a better performance for checking the message security, thus we choose this design.

As a result of the decision, the MCP design needs to employ a state machine. Messages passing through it would set its state according to the events such as the successful login of a bank. Top End has to be configured to support the start-up of multiple MCPs, one for each connection. The message header would need a data element, in particular the connection session identifier, to identify a connection.

7.1.6 Centralised control

The requirements *R5_2_0* and *R5_2_1* specify that the EFT system has to broadcast the settlement window timetable and the shutdown notification to member banks. These requirements would mean that special messages need to be sent through MCPs to all those banks that are on-line at the time.

There were two relevant issues in considering the architecture design. Firstly, what mechanism should be used to notify the banks and secondly which banks should be notified. Figure 7.9 shows the motivational reasons and the design reasoning of this design.

AR23 was a decision to use a centralised server to carry out the messages distribution. There was no alternative option in this decision because there is only one viable mechanism, i.e. a single process to coordinate its execution. A user would use an user interface *C9_0_1* to order a message distribution such as system shutdown. The order would be sent to the Control Server (*C8_0_0*) to execute. The Control Server would determine which banks are currently logged into the EFT system and then distribute the message to the banks. When designing the control server, a parallel issue was to determine how to detect bank login states. The decision in *AR24* was to make use of Session tables which logged the states of current bank connections (*I2_0_4* and *I2_0_5*).

It is common that a number of indirectly related decisions may exist in parallel in an architecture design. The decisions arising from the issues usually have common mo-

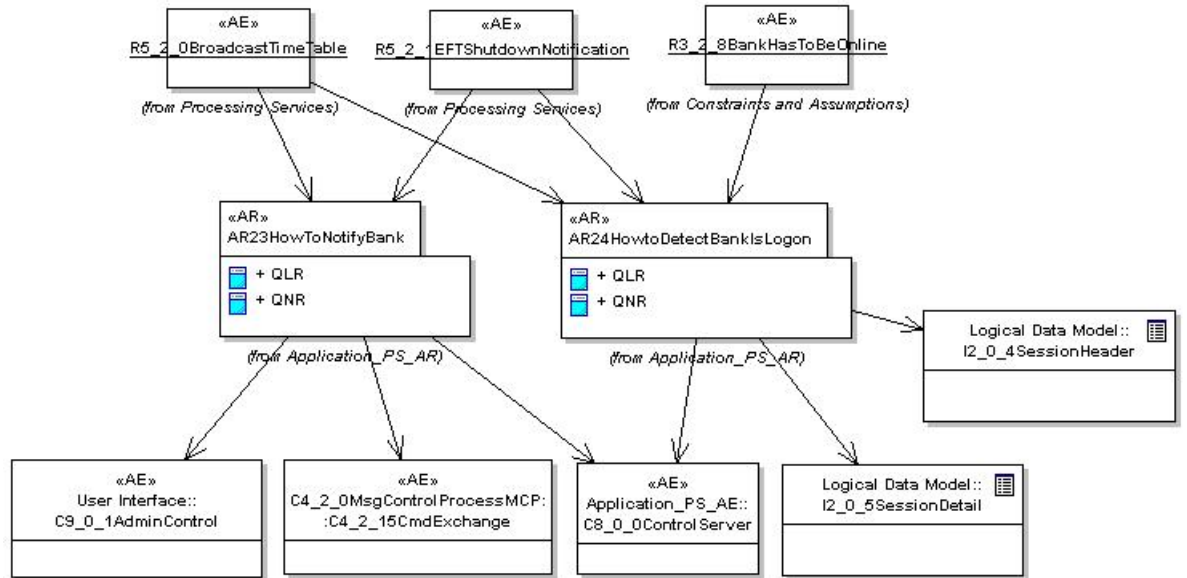


Figure 7.9: Decisions to Support Centralised Control

tivational reasons. In this case, the decisions are *AR23* and *AR24* and the common motivational reasons are *R5-2-0* and *R5-2-1*. The result of the inter-related issues in this case is *C8-0-0*. Although the two decisions themselves are independent, their causes and their effects can be common. This example shows that a requirement may cause multiple issues and decisions, and a design element may depend on multiple decisions. This multi-way relationships are often difficult to represent in textual specifications.

7.1.7 Message sequencing

One key requirement about payment messages is that every single message must be traceable and audit logged. This implies that each individual payment message must be uniquely identifiable (*R2-2-3*). There are many ways to assign an identifier to a payment message. The decision to be made is to decide which way is most appropriate to the EFT system. Two alternatives were considered. One alternative is to use a contiguous sequence number and the other way is to use a non-contiguous number. In either case, the messages from the EFT system and the messages from the bank must each have its own number sequence for their unique identification.

Figure 7.10 shows that two decisions are required. Decision *AR25* considers that the sequencing must be contiguous because a non-contiguous sequencing would make message reconciliation and auditing most difficult. If gaps were allowed, the system could not tell if a message is missing or if there was a gap in the sequence number. In this case, the only

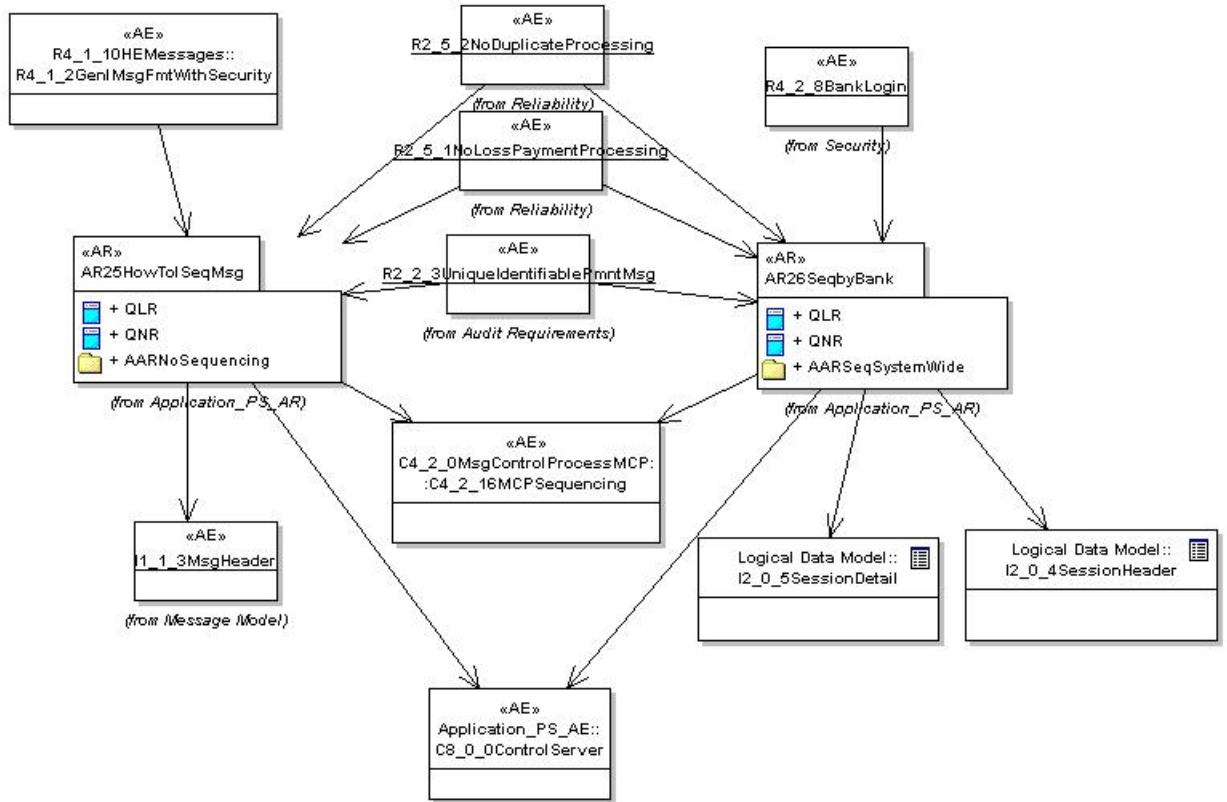


Figure 7.10: Decisions to Identify a Payment Message

way of reconciliation would be to include another mechanism acting as a reconciliation report to double-check what has been sent and received at each end.

A second decision *AR26* considers that the sequence number must be related to a bank connection. This is because each bank may send messages actively, and they have to assign their sequence number independently from the EFT system. Therefore, the architecture design is that a session identifier is created by the Control Server *C8_0_0* for each bank connection. A bank may have multiple connections to the EFT system simultaneously with different session identifiers. For each session, a unique sequence number with a defined range is used to identify a message in that session. Therefore, the composite key is a combination of session identifier and sequence number to uniquely identify a message.

Both the Control Server *C8_0_0* and MCP *C4_2_16* are the results of the two inter-related decisions. These decisions dictate their design through issues which are relevant to both of them. Hence, if the requirements or a decision has to change, the possible changes to these two inter-related components could be identified.

7.1.8 Error reporting

The catching and reporting of errors is one of the most important aspects in a system architecture. Exceptions and errors are imminent in any system. The timely reporting of errors enables remedial actions to correct the exception sooner rather than later. The requirement of the EFT system is to be able to log all errors and classify them by severity (requirements *R5_3_1* and *R5_3_2*). The classification of the errors define the severity level of the errors and hence the urgency of the responses to handle the error.

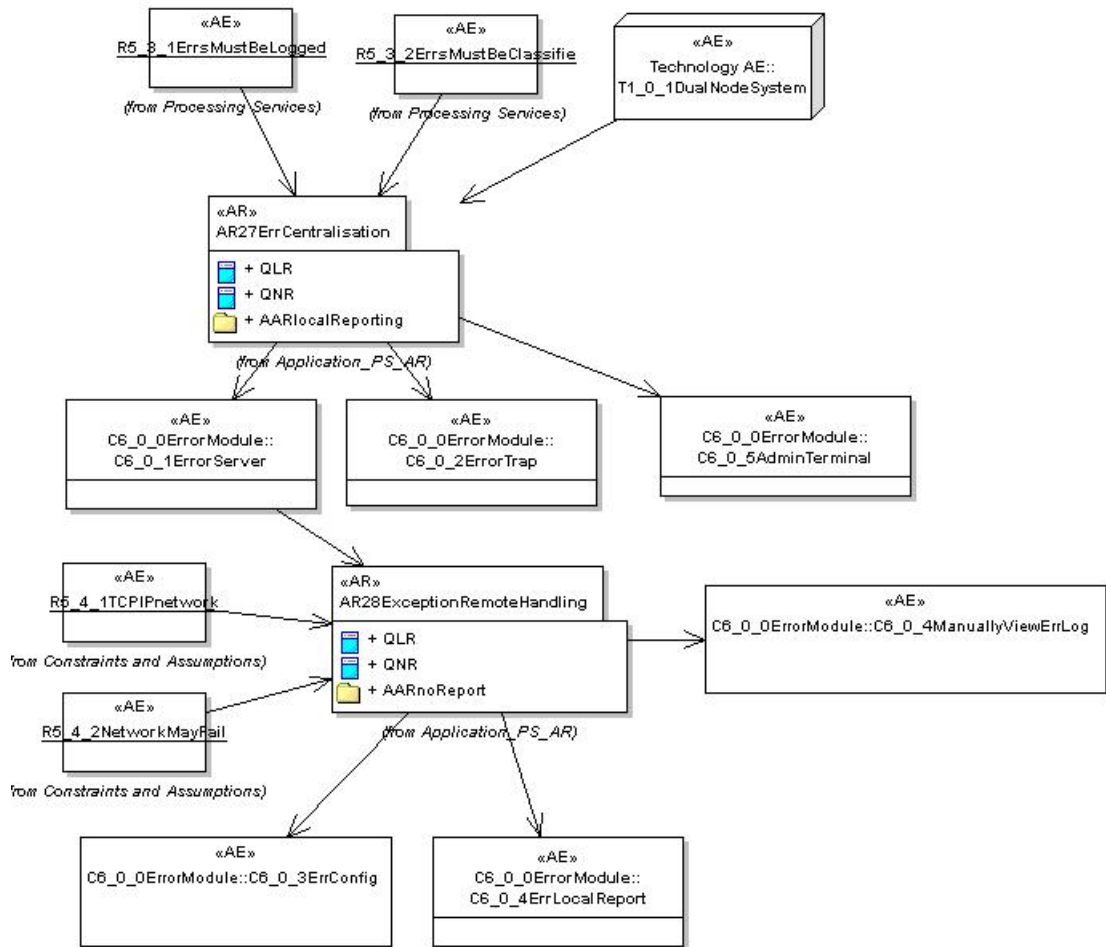


Figure 7.11: Decisions to Architect Error Catching and Reporting

Figure 7.11 shows that in order to cater for the errors, the design had to reason about where the errors would be reported. Since the system architecture is a dual-node system and it could expand to a quad-node system, logging errors in a local device would mean that administrators might not have a timely and centralised view of the error logs. Decision *AR27* considered that the error log should be centralised. It means that all software modules must be able to send errors to the Error Server (*C6.0.1*) through a standard

module *C6_0_2*. The error server would log errors from remote processes and report them through an administrative terminal to the administrator. The justification in *AR27* is to produce a single error log for ease of access. The alternative *AAR* in *AR27* suggests that errors can be logged in a computer where the error occurs. In a large distributed system environment, this design does not support diagnostic efficiently when inter-related errors span more than one computer.

There is, however, an assumption (*R5_4_2*) that the network might fail and the error message could not be sent to the Error Server. In this particular case, a decision has to be made to handle such potential failure. The creation of a design object *C6_0_1* raises a subsequent design issue that needs to be addressed. There are two possible design alternatives to address this issue, as documented in *AR28*. If the network communication is down, there are two design alternatives: no reporting or local reporting. The decision in *AR28* is to log the errors locally if the network has failed. We chose to use local reporting because it is mandatory to report errors. The alternative *AAR* is not to log the error, which is not acceptable.

7.2 An empirical study to validate the AREL model

The main objective of the AREL model is to capture and represent architecture design rationale. In the previous section, we have demonstrated this method by using a real-life system. In this section, we use an empirical study to validate the effectiveness of the AREL model.

Our intention is to validate that AREL facilitates the understanding of architecture rationale. The empirical study involves expert designers from the software industry to assess and validate the AREL method. We choose not to use student projects or small software development projects because the architecture complexity are often limited and the architecture rationale can easily be reconstructed. An industry example such as the EFT system is useful in the validation because it can demonstrate the architecture decision making process of a complex application.

7.2.1 Objectives of the empirical study

As stated in Chapter 4, one of the research questions to ask is “How to improve the representation of design rationale in architecture development?”. We need to ask this question because practising software architects have told us that design rationale representation is important (see Chapter 5). Therefore, we need to test the usefulness of the AREL method.

Using the EFT system, a comparison is made between the original design specifications that do not contain structured design rationale, and the AREL model. The comparison is to validate that the AREL representation is effective in facilitating the understanding of the architecture design of this case study. Although this validation cannot be generalised to all kinds of architecture design at this stage due to the single case limitation, it gives an indication of its potential.

7.2.2 About the empirical study

The validation of the AREL representation and its effectiveness is done by way of using expert opinions. Expert-opinion elicitation is defined as a heuristic process of gathering information and data or answering questions on issues or problems of concerns [5]. This type of validation has been applied to validating requirement process improvement model by a panel of experts [9]. We sought expert opinion instead of using students as test subjects because the experts have experiences in the domain area as well as architecture design. Their knowledge can provide a more reliable feedback to support our study. Furthermore, experts have a much wider exposure to different design methodologies and tools and can provide valuable insights towards this study.

We call on experts to mock carrying out design changes to the EFT system. Through this exercise, we gather the evidence to test whether or not AREL provides useful architecture design rationale to aid the understanding of system and software architecture. In a number of face-to-face interviews, the experts were asked about the design and design reasoning of the EFT system, with and without the AREL model. Then, the answers were analysed. The experts were also asked to comment on the usefulness of the the AREL method for capturing design rationale. The steps that were taken in the empirical study are listed below:

1. The design rationale for parts of the EFT system are retrospectively modelled using the AREL modelling method. The EFT design using AREL modelling is described in Section 7.1.
2. Original specifications are presented to the experts without the support of AREL.
3. Experts are asked four sets of design questions.
4. Answers from the experts are recorded. The experts would indicate whether the answers come from searching the specifications, memorising, deducing, guessing or any of the combinations.

5. Present the AREL model to the experts and discuss the design reasoning captured by AREL.
6. Compare the two sets of answers (with and without AREL support) to see if additional understanding are obtained with the use of the AREL model.
7. Interview the expert to obtain additional comments on advantages and disadvantages of AREL modelling.

7.2.3 Selecting experts

Experts who were associated with the EFT system were asked to participate in this research. Use of expert opinion is dependable because of the domain knowledge they possess [96]. We invited nine experts to participate (3 female and 6 male), all of whom have been working in this area for many years and can provide reliable assessment. On average, the experts have 16.6 years of IT experience (a maximum of 24 years and a minimum of 12 years), an average of 10.8 years of experience as an architect and designer (a maximum of 16 years and a minimum of 5 years), and an average of 7.7 years of experience in designing payment systems (a maximum of 18 years and a minimum of 1 year).

The expert panels are selected for their experience with architecture design and their working in the electronic payment system field for a number of years. They are characterised by the following:

- They have participated in the design, development and testing of the EFT systems or similar types of systems. Therefore, they were familiar with the functionality of such payment systems.
- They have had extensive design and development experience.
- They are willing to commit needed time and effort.
- They are willing to provide evaluations and interpretations of the new approach.

Owing to the specialised background knowledge required and the confidentiality of the system, there is a limitation as to who can be recruited. Consequently, we decided to use non-probabilistic sampling techniques and convenience sampling method instead of random sampling [80]. Availability sampling operates by seeking responses from those people who meet the inclusion criteria. They are available and willing to participate in the research. The major drawback of non-probabilistic sampling techniques is that the results cannot be considered statistically generalisable to the target population, in this case software architects.

7.2.4 Empirical study results

In this section, we describe the results of the study. The empirical study relies on a panel of experts to reason about the architecture with and without the aid of the AREL model. The experts may use the original design specifications, recall from memory, make deduction or make guesses to answer the questions. After the initial analysis of the design reasoning, a comparison is made to see if additional reasoning could be gained with the AREL model.

Experts were asked questions in four areas of the architecture design. These areas are described in the case study sections of this chapter. The following sections discuss each set of questions and the answers that have been provided by the experts.

Question 1 - Fault resilient support

This question asks about the fault-resilient feature of the EFT system and the reasons for having this design. The design reasoning is described in Section 7.1.2. Table 7.1 presents an analysis of how experts recall the design rationale. The column on *documentation* indicates that some of the information comes from a documented source such as specifications. *Memory* indicates that the experts remember the reasons. This source of information is not as reliable because the experts may remember wrongly or incompletely. *Deduction* is what the experts reconstruct as design rationale based on documented and memory sources. *Guessing* is what the experts suspect of the reasoning based on experience. There are no facts to support the reasoning of guesses. An expert may use more than one means to reason about the design.

Table 7.1: Empirical study - Question 1 Results

	Documentation	Memory	Deduction	Guess
Responses	6	3	4	2

In 7.1, 6 experts could find the answers from the specifications that describe what fault-resilient mechanisms are provided by the system. This is because the mechanisms are clearly documented in the design specifications. Based on which, most experts have deduced and guessed the design rationale without much trouble. The experts were able to describe application recovery, dual-node support for fault resiliency and communication backup mechanisms. The AREL model presented to the experts confirmed their answers. AREL provided additional items which have been missed by most of the experts. Examples are the power failure and site failure of the system.

Question 2 - Software layering

Question 2 asks why there are three layers of software in the EFT design and whether this is important to the architecture design. This design is described in Section 7.1.3 of the case study. Most experts could not find the relevant description in the architecture design specification (see Table 7.2). However, most experts could recall or deduce that the layering of software is due to the modularity and flexibility requirements of the software. Some experts mentioned the separation of concerns would help maintenance and produce a “cleaner” design.

Table 7.2: Empirical study - Question 2 Results

	Documentation	Memory	Deduction	Guess
Responses	1	5	6	3

When the experts were asked as to why there are three different UNIX processes with each process supporting a separate software layer, none of the experts could find a reason in the documentation. They deduced that it was due to the need to support different communication protocols. Some of the experts queried the legitimacy of this design. AREL was shown to the experts to explain that the original design was influenced by an external factor. The EFT system was meant to be a product to eventually support national payment systems, and an international panel of experts at the time dictated that the three layers of software have to be physically separate and hence the EFT design followed such architecture.

Using the architecture design reasoning approach, some experts have pointed out that this design is not necessarily the best approach. This is because having three physical UNIX processes means that two different communication protocols are required to support them. This necessarily increase the complexity of the EFT system without additional benefits. It would have been better if the messaging and communication layers are combined to form a single process. As such, both flexibility and the performance are maintained. The architecture design would be simplified. In this case, the AREL architecture rationale have facilitated discussions and arguments for the purpose of architecture design validation.

Question 3 - Asynchronous messaging

We ask the experts what the reasons are for selecting asynchronous communication instead of synchronous communication, in which the design is described in Figure 7.5. All experts suggested that this is somehow due to performance reason.

Table 7.3: Empirical study - Question 3 Results

	Documentation	Memory	Deduction	Guess
Responses	1	4	4	5

Table 7.3 shows that most experts could not find the answers from the documentation but instead they make guesses and deduction of why the system was designed this way. Most experts commented that this is a complex issue which involved a number of intertwined design. Some of the experts recalled the design features of this part of the system from memory and deduction.

When the AREL diagram was shown to them, some experts noted that this design might have been overly complicated. Some of them suggested that perhaps a synchronous approach should be reconsidered to see what the system architecture could look like and make comparisons with the the existing design to check their relative benefits. One expert noted that the complexity issues might arise with the synchronous design and further architecture explorations are required to assess the two alternatives.

The responses from the experts have highlighted the fact that although the system had been in production for many years, there are no consensus as to whether this design is the most appropriate one. This is because the reasons for choosing the asynchronous design instead of the synchronous design had not been explored or tested fully. It was suggested by the experts that AREL could provide added value to the architecture design process through its explicit reasoning representation in that comparisons could be made to check and verify the design.

Question 4 - MCP connection

We ask the expert why a single MCP process handles a single bank connection only. The design reasoning is described in Section 7.1.5. The experts found in the design specification some description of how this part of the system was designed, which gave some hints of the reason behind the design. The experts made guesses (see Table 7.4) based on the hints they gathered and what they thought might have been the reasons. The experts guessed that it was to do with the simplification of the design and security issues. None of the answers were specific.

Table 7.4: Empirical study - Question 4 Results

	Documentation	Memory	Deduction	Guess
Responses	3	4	1	6

When the AREL diagram was shown to the experts. It was immediately obvious that the online banking requirement is the driver of this decision and that the security checking is an important factor in this design decision. As such, AREL serves its purpose well by explicitly depicting the motivational reasons of the design and the design rationale.

Expert assessment of AREL

The last part of the interview is to have the experts assess the AREL model. The intention is to seek qualitative opinion on whether AREL would be useful in practice, which is one of the primary objectives of this thesis. Two questions were asked to see whether the experts would consider using the AREL for modelling architecture design decisions. The first question asks the experts whether AREL is useful in helping the experts reason with the EFT design. Table 7.5 shows the responses. Out of the 9 experts, 44.4% think that AREL is very useful (rating 5), another 44.4% think that it is useful (rating 4), 11.1% is neutral (rating 3). The result indicates that the experts find that AREL can help them understand the design reasoning.

Table 7.5: AREL Usefulness in Supporting Architecture Design Reasoning

	1	2	3	4	5
	Not Helpful				Very Helpful
Responses	0	0	1	4	4
Percentages	0	0	11.1	44.4	44.4

The next question asks the expert whether they would capture design rationale with AREL given reasonable project schedule and resources. The results are shown in Table 7.6. 55.6% of the experts indicated that they would capture design rationale with AREL. 44.4% of the experts indicated that they would possibly capture design rationale with AREL. The result shows that there is a very strong intention for the experts to capture design rationale. There are none who indicated that they wouldn't capture design rationale and there were none that do not know if they would capture design rationale. It means that all of the experts find AREL useful and are willing to use it in practice.

Table 7.6: Experts' Willingness to Use AREL

	No	Don't Know	Possibly	Yes
Responses	0	0	4	5
Percentages	0	0	44.4	55.6

The experts were then asked to provide comments, both positive and negative, about the AREL method. The experts commented on AREL with respect to designing systems and they also commented on the other issues that are design knowledge related. The following are the comments gathered in the interviews.

Traceable Design

1. Design specifications do not connect requirements to design, making it difficult to understand where the requirements come from initially. This lack of traceability in specification means that the designer has to search through the document in order to understand a design.
2. General grouping of requirements for traceability is required.
3. AREL can make explicit hidden relationship and allow architects to verify design with explicit reasoning.
4. AREL presentation is direct and easy to follow.
5. AREL can help architects to focus on a specific part of the design very quickly to start tracing interdependent requirements and design.

Design Rationale Support

6. AREL can facilitate the design thought process in a conscious way.
7. AREL provides a structure to apply common sense.
8. AREL enforces a design process in which architects have to justify why they design things in a certain way.
9. AREL can provide a high-level understanding of the system before designers start reading detailed specifications.
10. Architecture design should be at a reasonably high-level and AREL implementation should not go into too much details.
11. AREL provides a methodology to capture the knowledge from the design process through the requirement, design and implementation cycle.

Software Development Support

12. AREL provides a development “standard” which ensures that architecture rationale is captured.
13. AREL can highlight design complexity before implementation. This enables architects to have a better understanding of the problem, the associated costs and complexity of the design before committing to development.
14. AREL complements design specifications very nicely.
15. Graphical representation in AREL is useful for showing complex relationship which is hard to trace with textual specifications.
16. AREL facilitates verification by peer review and stakeholders review.
17. AREL is necessary to support maintenance activities in large projects.

From the above comments, there are three general areas where the experts have commented positively about AREL: (a) the traceability features in AREL help them understand the system; (b) the design rationale support helps decision making, knowledge

retention and comprehension; (c) the existence of a design reasoning process generally helps the development and maintenance of software systems. This is a very positive result because it shows that AREL can play an important role in software architecture.

On the other hand, the experts have some reservations about the implementation of AREL:

- with AREL, architects still need to be thorough with identifying assumptions and constraints or the design might still be inadequate.
- the completeness of the design rationale depends on the person doing it.
- the recording of the design reasoning would depend on individual author and this is a subjective matter.
- the graphical structure of AREL might be hard to manage when the design becomes very complex.
- the cost of building the AREL model is uncertain.
- cost of updating AREL may be high during the design phase.
- the practice needs to be standardised in an organisation so that it is repeatable.
- it may not be suitable for small project because the cost cannot be justified.
- certain decisions may not be documented deliberately due to politics.

Their reservations mainly surround how designers may use AREL and whether AREL would be cost effective. Since this pilot study is a small scale study on a specific system, and the tool used in the study is only a proof-of-concept, the real benefits of AREL's applications in real-life systems are yet to be proven.

Given what has been demonstrated to the experts during the interview sessions, experts were asked how AREL can be improved. A number of suggestions have been made to enhance AREL. The experts indicated that tool implementation is essential in AREL's application. Therefore, a lot of suggestions focus on how AREL should be implemented.

- because of the complexity of architecture design, tool support is critical in its successful implementation.
- if a template model for a certain type of design is available in AREL representation, architects can easily use it to adapt to a similar design

- tools should be available to generate traceability report based on designers' enquiry.
- in order to make full use of AREL, the requirements have to cross-reference design specifications.
- there should be keyword search in AREL to find the design rationale, requirements and design.
- enable cross-reference of multiple projects sharing similar architecture design.
- different types of search results should be shown in different colours to highlight search path.

In summary, the responses given by the experts clearly support the retention of design rationale knowledge using the AREL method. The experts see that AREL can help architecture design, verification and maintenance. The experts, however, are concerned with the cost and implementation of AREL. These two issues are related in that the cost can be minimised if suitable and usable tools are available to support the AREL applications.

7.2.5 Limitations

During the study, experts employ different techniques to analyse the problems when they were given the design specifications. Some experts would first read the specifications and then discuss the issues, especially for those who had involved in the related design of this system. Others would briefly read the specifications, then make deduction and guesses and seek discussions. The time it took each expert to reach their conclusions vary depending on the techniques as well as personal style. Additionally, the experts were not familiar with the AREL methodology and it took time to explain the AREL diagrams. Therefore, it was found that the time measured vary greatly depending on the familiarity of the expert on this particular aspect of the design. As such, the time cannot be used to gauge how quickly AREL could provide design reasoning compared with traditional design specifications.

There are differences in the level of experience between experts on certain subjects. Some experts are more familiar with the software design at the communication level, and some more familiar with the message composition design. In general, all of the experts had provided correct answers to the questions, the level of details in the answers given by the experts varied depending on their level of expertise in that area. Therefore, the usefulness of the AREL is inversely related to how familiar they are to the topic of discussion. The more familiar they are with the subject, the less they have to refer to design specifications or the AREL models.

The case study is specific to a particular industry which requires expertise of someone who understands electronic payment system. This criterion has placed a limitation on who can participate in the empirical study. Random sampling of architects does not apply in this case. Additionally, with a single case study, the claim that a method can be generalised is limited [174, 115]. So we cannot claim that the AREL method can generally benefit all types of architecture design because many more cases are required to reach such a conclusion. However, the validation by the experts does provide limited evidence to support that AREL is potentially beneficial to the understanding of architecture design rationale.

7.3 Summary

In this chapter, we have demonstrated how to apply AREL to capture and represent architecture design rationale in a case study. We have used a real-world electronic payment system, the EFT system, as our case study. We have presented various parts of the EFT system design to show how AREL captures architecture design rationale and how they are linked to the motivational reasons and design objects.

In order to validate that AREL can be useful in practice, we carried out an empirical study using 9 experts to study the design reasoning of the EFT system. All experts have extensive experience in system design and in payment systems. They were asked questions that required them to use design specifications and other means to reason with the EFT architecture design. Their answers were compared with the architecture rationale captured by AREL. They found that AREL provides a more explicit and direct way to document design rationale. The experts have indicated that AREL is useful in practice and they would use it if tools are available.

Chapter 8

The architecture rationalisation method

Methodologies such as the object-oriented analysis and design provide useful processes and methods for designing systems and software. However, their successful implementation relies on the designers who use them. Firstly, the designers interpret how a methodology is to be applied in a specific environment where many design decisions are made. As such, the correctness and appropriateness of these decisions become fundamental success factors. Secondly, when different designers are given identical requirements, they might come up with different design outcomes with varying qualities. This could be attributed to the differences in interpreting their design environment. Since reasoning and decision making have a direct bearing on the design outcome, therefore it is necessary to examine these processes in architecture design.

As shown in a survey [157], architects not only have to consider technical design but they also have to consider a broad range of issues such as management and planning. This expanded scope requires architects to make decisions based on a combination of factors such as business strategies, requirements, project constraints and technologies. The quality of the results may depend on how well the architects understand the situation surrounding the design and how well they make the decisions.

So far there is insufficient understanding on *how* architecture decisions affect the quality of architecture design. We assume that good decisions would lead to high quality architecture design. We further assume that good decisions are based on two conditions: (a) the information to support decision making is available, comprehensive and accurate; and (b) the analysis used in decision making is objective and sound.

Based on these assumptions, we seek to improve decision making by structuring the decision process. Such an approach can provide some discipline to the architecture design process where it currently relies heavily on individual experience and intuition. Such approach can also help less experienced architects to systematically make decisions to design better. Using the AREL model as a basis, we propose the Architecture Rationalisation Method (ARM) as a process to systematically reason about design decisions during architecture construction.¹ Section 8.2 describes how ARM works. Section 8.3 describes how ARM can be applied to various aspects of architecture design and system maintenance.

8.1 Background

The process of architecture design is a systematic decomposition of problems using requirements as a basis to drive design objects creation. The key focus of the architecture process is to create a design structure for the system. During this process, requirements analysis and clarifications may be performed. Although requirements elicitation and analysis are relevant, they are not the key outcomes of architecture design. There has not been a commonly agreed scope for architecture design process. Researchers [8, 119, 46, 84] and industry consortia [21, 31, 164] hold different views of the scope of architecture activities ranging from enterprise systems strategic planning to detailed software design.

Existing requirements engineering methods mainly focus on the elicitation, modelling and refinement of business goals. In particular, it articulates high-level business goals to become systems requirements. For example, graphical notations for requirements decomposition using *AND / OR* reduction links have been proposed [28, 61]. Mylopoulos et al. [104] present a framework for representing non-functional requirements and proposed decomposition methods to satisfy non-functional goals. Dijkman et al. [30] use a separation-of-concerns or multiple-viewpoint approach to relate the enterprise viewpoint (i.e. requirements) to the computational viewpoint (i.e. design). These methods provide different ways to transform requirements into design. The level of transformation required to reach the necessary design details is often not defined clearly.

In Figure 8.1, we outline a generic high-level system development process to define the scope for system and software architecture design activities.

- *Business requirements elicitation* stage - business analysts or designers gather business goals and produces system requirements. This activity is mainly concerned with requirements gathering and therefore is outside the scope of architecture design. Its

¹This chapter is based partially on our work published in [159].

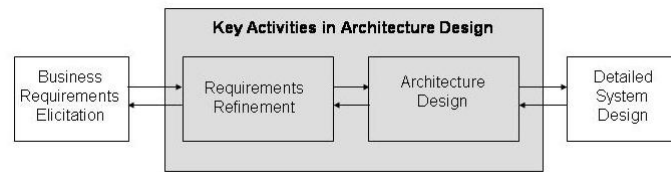


Figure 8.1: Key Activities in Architecture Design

outcomes, i.e. the elicited business and functional requirements, are inputs into the architecture design process.

- *Requirements refinement* stage - architects and designers gather architecture related information such as technical environment, information strategies and project environment. This information is analysed and refined with respect to the functional requirements. Any implicit assumptions about the requirements are clarified and documented at this stage.
- *Architecture design* stage - architects and designers create the high-level architecture design by using refined requirements as inputs.
- *Detailed design* stage - based on the architecture design, designers produce detailed design and program specifications. Since detailed designs usually are not concerned with the fundamental structure of the system, they are outside the scope of the architecture design process.

Depending on the development approach, the activities in an architecture life-cycle may take place in different orders. Examples are waterfall, iterative and agile methodologies. However, the relationship between these activities are still relevant. The arrows between the generic activities indicate the inherent relationships between the products within each stage. For instance, elicited requirements are inputs into the architecture design process and yet the architecture design process could change the requirements. Thus, the development process from requirements elicitation to detailed design has a feedback loop where the activities are influential in a bidirectional way (see Figure 8.1). Similar discussion can be found in software engineering texts such as [146].

Some researchers argue that early-phase requirements modelling and reasoning support can facilitate the requirements elicitation and refinement cycle [175, 108]. Some researchers propose methodologies to tightly relate requirements in the architecture process [56, 130]. But requirements might still change when design constraints or unclear requirements are uncovered in the downstream development process. Such modifications to requirements may be unavoidable because the issues are not obvious previously. The

scope of architecture design is thus required to cover requirements refinement as well as architecture design.

Using general software engineering design principles, we examine how architects and designers make design decisions. As discussed in [171], decision analysis deals with uncertainties. An architect deals with uncertainties by collecting evidence, analysing the possible outcomes and seeking ways to minimise such uncertainties in a design. During this process, an architect would attempt to find all viable design options, i.e. eliminating options that are not viable. Then the architect would try to select a design that maximises the benefits and minimises the risks and costs. In other words, the architect would try to get the best values out of an architecture design and at the same time to ascertain that the design is achievable. Such acts by designers are reported in our survey (see Chapter 5) where architects agree that costs, benefits and risks are important elements in design decision making. The ARM method is an attempt to make this intuitive mental process explicit in the architecture design process.

8.2 The architecture rationalisation method

The Architecture Rationalization Method (ARM) is a method to help architects make design decisions during the architecture process. ARM uses the AREL structure to represent the architecture design and the architecture rationale. The outcomes of ARM is an AREL model. The reason for having ARM is to provide a method to guide software architects in their approach to development. Using the rationale-based approach, requirements and architecture design may influence each other through causal effects. The ARM method provides three focal points to help assess these effects: cost, benefit and risk. These focal points can be integrated with a development process in reasoning with a design.

An architect can use ARM together with other development methods such as iterative, prototyping or agile. ARM supports them in the decision making realm by using qualitative and quantitative rationale. The two types of architecture design activities we focus on are:

- **Requirements Refinement** - an architecture design process often leads to the refinement of functional requirements due to conflict resolutions or requirements clarifications. Architecture design clarifies and defines non-functional requirements or quality of services such as system reliability and performance. Refining or defining functional and non-functional requirements takes place when contemplating an architecture design. A functional requirement may have implications on, say, the

usability of the system. With clarifications and tradeoffs, stakeholders may agree to change the requirement. Similarly, non-functional requirements are analysed and refined iteratively in the architecture process [20, 29, 102]. Constraints and assumptions can be uncovered and explicitly stated. ARM uses decision reasoning as a catalyst to refine and define functional requirements, non-functional requirements, assumptions and constraints.

- **Architecture Design** - the architecture design process is in general a top down approach where high level decisions are usually needed before the details of the design are considered. This is done so that the structure of the system is in place prior to addressing the design details. This process would continue until the increasing level of details provided by the architecture design are in place and satisfy all the major requirements.

ARM design rationalisation is an explicit process to support decision making. It provides a structure to support architecture design by ways of reasoning qualitatively and quantitatively. Design decisions are often interrelated in that one decision may have an impact on other decisions. This kind of inter-relationship amongst decisions require the architects to consider decisions in groups and tracing them to understand the design. Thus, the capturing of design rationale and reasoning relationships in AREL can be useful in supporting such analysis.

8.2.1 Qualitative rationale

The argumentation-based design rationale methods suggest that the design reasoning should be based on fundamental elements such as issues, arguments and positions (see Chapter 3). In AREL, the design reasoning is encapsulated in qualitative design reasoning or *QLR* to provide a template for investigating and capturing a decision (see Chapter 6). Architects first need to articulate the *issue* of a decision, this is done by drawing all the motivational reasons which contribute to the forming of this issue. For instance, an architect needs to decide on a programming language for the project. This is an issue. The motivational reasons to be considered might be the suitability of the language for the job, the skill set of the developers, the available compiler, the direction of the company etc.

A number of options might be available, say C++, Java and Visual Basic. Before reaching a conclusion, motivational reasons that influence the decision have to be clearly outlined and assessed. Each of them would exert certain constraints and influence to the decision in its own way. The architect would make a balanced assessment in order to reach

a decision.

This approach allows architects and designers to justify a decision based on the explicit representation of motivational reasons and design rationale. It provides a structure to support an objective decision making process. However, the assessment could still be biased if the architect is subjective and opinionated towards one solution. The qualitative reasoning mandated by ARM would encourage an architect to consider the options and their relative benefits. Additionally, the recording of such reasoning could be used for architecture verification. As such, it will reduce the chance of undesired biases.

8.2.2 Quantitative rationale

There are a number of methods to support quantitative analysis for decision making. Notable examples are CBAM [4] and AHP [2]. These methods commonly use priority, costs and benefits for assessing different options to make a decision. This may be justifiable for some of the key architecture decisions where a group of stakeholders are gathered together in the assessment process. It would be quite a costly process to always make decisions with a group of stakeholders. For most typical architecture decisions, however, only one architect or a small group of architects would be responsible after consultation with the stakeholders concerned.

In ARM, decision making requires the evaluation of the *costs*, *benefits* and *risks* of design options, and such evaluations are the quantitative rationale *QNR* for comparing and selecting design options. For instance, when two options are considered in a decision, their relative costs, benefits and risks can be compared in an explicit way to justify a decision. In a survey, we have asked practising software architects if they think these design rationale are useful. Their responses suggested that such design rationale are indeed very useful in justifying the design options (see Section 5.3.4). Furthermore, A quantitative approach could be more effective in characterising the decision, costing the design and improving the design process [42].

Expected costs and benefits

If there are different ways to design a system to achieve a set of goals, we can assume that an experienced and logical designer would attempt to select a viable design. To achieve this, an architect would first eliminate those options that would not work. Then the architect would find a design which maximises the benefits of the outcomes whilst minimises its cost. If more than one option is available, this exercise can be quite complex

since the architect has to decide on the trade-off between options.

ARM provides a quantitative method to characterise the cost and the benefit of a decision. The quantitative rationale is captured in *QNR*. The Architecture Cost Index (*ACI*) is assigned an index to weight the cost of implementing the decision. *ACI* is an index (between 1 and 10) where 1 is the least cost and 10 is the highest cost. *ACI* is an index that takes into consideration a multitude of cost factors to provide a relative cost index between alternative rationales. The considerations for *ACI* weighing are the following:

- Development costs - this cost takes into account the cost of development and training requirement.
- Platform costs - this cost takes into account the cost for platform support such as hardware and software.
- Maintenance costs - this cost takes into account routine operational maintenance and support, software maintenance, software modifiability and portability.
- Potential costs - security, legal and other costs that may arise from the design should also be considered.

The Architecture Benefits Index (*ABI*) is an index (between 1 and 10) where 1 is the least benefits and 10 is the highest benefits. It represents the relative benefits an architecture decision or design would deliver to satisfy the concerned requirements. If compromises need to be made between competing requirements, then the architect would make a judgement on the relative priority of requirements and the level of satisfaction the architectural design provides to meet the requirements. Similar assessments would be made for alternative designs for comparisons.

Example 1: A decision needs to be made on which database system to use for a Customer Relationship Management (CRM) system with 2 million customers, 30 million transactions per annum (data will be on-line for 5 years) and 100 on-line users. There are two options being contemplated: MySQL or Oracle.

Table 8.1 shows a list of factors being considered and the relative weightings that have been assigned to a database system with regards to this application. There are five factors to consider in each of the cost and benefit category. Given that each of the five factors have equal weighting, each can have a score of up to 2. The five factors together sum up to 10, which is the highest cost or benefit index. We use a five points scale (0, 0.5, 1, 1.5, 2) to weigh each factor. 0 being the lowest cost or benefit and 2 being the highest cost or

Table 8.1: A Comparison of the Architecture Costs and Benefits

Factors	MySQL	Oracle
Costs		
License Fees	0	2
Maintenance Fees	0.5	2
Training	1	1
Implementation	1	1
Integration	1	1
ACI	3.5	7
Benefits		
Performance	1	2
Flexibility	1	2
Reliability	0.5	2
Scalability	0.5	2
Security	0.5	1
ABI	3.5	9

benefit. In the above example, we see that the cost of MySQL license fee is 0 and the cost of Oracle is 2 because of the differences in licensing fee charges.

When we sum up the cost factors, the *ACI* of MySQL and Oracle are 3.5 and 7 respectively indicating that MySQL has a lower cost than Oracle. The *ABI* of MySQL and Oracle are 3.5 and 9 respectively indicating that Oracle provides a higher benefits than MySQL. This rating reflects the opinion of the architect and is contextual given the background of a specific project and organisation. The actual cost and benefits information such as the annual maintenance fees are gathered, analysed and documented in qualitative rationale *QLR* for future references.

Architecture risks

Charette [19] suggested that risk is an uncertainty about the future, and since the future involves changes, the person facing the future will have to make a choice to accommodate the uncertainties. He suggested that risk analysis and management in software engineering share this same characteristic about risk. In order to assess the uncertainties, subjective probabilities are often used to estimate the risk [10].

When architects make a decision in architecture design, he/she faces a similar kind of uncertainties about the design. The explicit representation of uncertainties will provide a focal point for architects to deal with them. For instance, if an architect is uncertain that a design is implementable, then a more detailed analysis and design is required to reduce or remove that uncertainty. In ARM, we apply the risk assessment principle to measure the uncertainty of a decision or a choice for a design decision.

ARM represents uncertainties with a ratio ranging between zero and one, where zero

indicates no risk and one indicate total risk. Outcome Certainty Risk (*OCR*) measures the risk or the uncertainty level of the architectural design meeting the desired outcome, represented by the Architecture Benefits Index (*ABI*). Basically it indicates the level of risk of not meeting the desired outcome by the architecture design at a decision point. In other words, it indicates if motivational reasons are satisfied by the architecture decision.

Implementation Certainty Risk (*ICR*) measures the risk or the uncertainty that there are no unexpected issues in the implementation of the architectural design. In other words, *ICR* represents the architect's assessment of the uncertainty that issues may occur during the design, development or implementation phases, thus affecting the Architecture Cost Index (*ACI*). For instances, does the team have the knowledge to carry out the implementation? Is the technology stable enough to support this implementation?

Using example 1 as an illustration, the architect feels that the *OCR* for MySQL is higher than Oracle, hence MySQL is given the *OCR* risk 0.4 and Oracle is given the *OCR* risk 0.2. This assessment basically is an estimation that the architect feels that there is a higher chance that there be some issues with MySQL. The *ICR* risk of MySQL is 0.3 and Oracle is 0.2. This indicates that when it comes to implementation, Oracle has less risk than MySQL because developers are more familiar with Oracle and there are more documentation provided by Oracle and various sources including Oracle local technical support to guide the developers.

Evaluation of risks is an important and useful method for architecture modelling because it provides a means to discover and identify uncertainties. We further argue that it should be done as an ongoing and incremental activity during the architecture construction process. Some research work suggests that software architecture risk is evaluated when a model is relatively complete [50], we suggest that risk analysis should be carried out during the entire decision making process. Designers can investigate the decision issue in more details if they feel that there are major uncertainties in the decision. Risk awareness helps designers to be mindful of assumptions and unknowns that might affect the design.

Architecture cost-benefit ratio

Although the costs, benefits and risks give us some measurements to understand the different perspectives of a design alternative, it is difficult to make comparisons between alternative design. As such, we require a utility function to allow us to compute an index to compare alternative designs. The cost-benefit ratio in ARM provides a value to each design alternative to help decision making. The calculation takes into account the risks that are involved in each design option.

Quantitative rationalization using these metrics may be used at each decision point during architecture design. It can be supported by qualitative arguments that are captured in *QLR*. The Expected Benefit (*EB*) is the Architecture Benefit Index (*ABI*) discounted by the potential impact of not meeting the benefits which is represented by *OCR*. As such, the benefit is discounted by its risk, i.e. $1 - OCR$. The expected Benefit (*EB*) is therefore:

$$EB = (1 - OCR) * ABI$$

Expected Cost (*EC*) is the Architecture Cost Index (*ACI*) amplified by the risk of design implementation which is represented by *ICR*. As such, the cost increases according to the risk, i.e. $1 + ICR$. The expected Cost (*EC*) is therefore:

$$EC = (1 + ICR) * ACI$$

Quantitative rationale evaluation uses Cost-Benefit Ratio (*CBR*) to measure the ratio of expected benefits to costs at a particular decision point. Cost-Benefit Ratio (*CBR*) is expressed as follows:

$$CBR = \frac{EB}{EC} = \frac{(1 - OCR) * ABI}{(1 + ICR) * ACI}$$

Given the *CBR* is represented by benefits over cost, thus the higher the *CBR* ratio, the better the expected benefit ratio. Given two alternative architectural designs that deal with the same set of requirements, the design that has a higher *CBR* ratio is necessarily the better option because the architect expect a higher return from choosing this option. *CBR* is an index to indicate expected returns of each design, architects need to assign the ratings using a uniform scale to make comparisons possible between options.

Table 8.2: The Expected Architecture Cost and Benefit Ratio

	ABI	OCR	EB	ACI	ICR	EC	CBR
MySQL	3.5	0.4	2.1	3.5	0.3	4.55	0.46
Oracle	9	0.2	7.2	7	0.2	8.4	0.85

Using example 1, the expected benefit rating of MySQL and Oracle are 2.1 and 7.2 respectively. Similarly, the expected cost of MySQL is lower than Oracle, 4.55 and 8.4 respectively (see table 8.2). Using the expected return formula, the *CBR* of MySQL and Oracle are 0.46 and 0.85 respectively. Since Oracle has a higher expected return, it can be chosen as the platform for implementing the CRM system.

The quantitative approach using *QNR* has three merits: (a) architecture rationale is quantified at each decision point; (b) *CBR* takes into account risks or uncertainties in architecture modelling; (c) the soundness of each architecture decision can be assessed and measured for evaluation and verification. Using this approach, a quantitative comparison

of architecture design options can be performed.

The estimations or the expectations given by architects are subjective in nature. *EC* and *EB* are subjectively expected utilities because the cost, benefit and risk measurements are all subjective assessment by an architect. CBAM [4] and ArchDesigner [2] provide a similar quantitative approach to rank important decisions. ARM differs to these approaches in that the quantitative rationale in ARM is risk-based. Although ARM's assessment is subjective in nature, we posit that experienced architects could improve risk estimation because analysis methodology such as CMMI can be performed to refine their estimations over many projects. This feedback loop could improve the decision making process and identify areas where design estimations are inferior.

8.3 Other applications of ARM

As discussed earlier, ARM can assist architects to reason with decisions by exploring, assessing and documenting design options. It can be used to supplement design methodologies such as object-oriented modelling and database modelling techniques.

With ARM, two other applications to support software development are now possible: (a) the scope of architecture design can be ambiguous because it is difficult to determine when architecture design is complete and when other design activities should take over. Section 8.3.1 describes how ARM is used to delineate architecture design activities from detailed design activities; (b) using architecture rationale, the architecture design can be verified by examining the design reasoning. This is described in 8.3.2.

8.3.1 Completeness of architecture design

Software architecture is concerned with the structure of a software system. Its objectives and activities are different to that of detailed software design. The need for such distinction lies in the role of the architects and the need for its verification. There is currently no agreed definition on the amount of details that are required in architecture modelling in order to satisfy the objectives of constructing an architecture. Instead of using *Intensional* and *Non-local* properties [37] to distinguish architecture activities, we propose a method for distinguishing architecture activities from detailed design activities based on the level of risks. The level of risks represents the uncertainty of the architecture model to meet its requirements [19]. The less uncertainty in an architecture design the more confident an architect is about the design.

As discussed in Section 8.2.2, architects provide estimates for Outcome Certainty Risk (*OCR*) and Implementation Certainty Risk (*ICR*) at each decision point. *OCR* represents the uncertainties of meeting the outcome objectives set out in the functional or non-functional requirements. This risk may arise because of a number of reasons:

- Functional or non-functional requirements may be ambiguous and need to be clarified or redefined.
- A design may have an impact on certain aspects of some non-functional requirements and the extent of that impact is unknown.
- Certain external environmental factors may have an impact on the implementation of the requirements and the extent of that impact is unknown.
- Assumptions need to be made about different aspects of the system such as business requirements, project environment or technologies.

ICR represents the uncertainties in implementing the design. This risk may arise for the following reasons:

- Technical feasibility of the implementation.
- Uncertainty due to the complexity of the design.
- Uncertainty due to a lack of experience, knowledge or skills of the development team.

These two types of risks may be resolved through iterations of refinement and architecture design. ARM records and analyses these risks by using quantitative design rationale *OCR* and *ICR*. The risks can be estimated and progressively reduced to an acceptable threshold as architecture design progresses and design details become available. The certainty that the architecture model could satisfy the requirements would increase. When this certainty level reaches a pre-defined threshold for all architectural requirements and designs, we regard that the architecture model is *complete*.

Different requirements in the system post different levels of risks. The level of design details required in various parts of an architecture model vary depending on the level of risk. For instance, a reusable item with a well defined behaviour and interface does not require additional in-depth architecture design. On the other hand, requirements which are complex in nature or have extensive non-functional requirement implications may need further investigation into its technical feasibility and explore more details of the design

to ascertain its feasibility and cost. Requirements that need architectural investigations are called architectural level requirements. Requirements that do not require architectural investigations because their risks are low, become the concern of detailed system design activities.

OCR and *ICR* depict the levels of uncertainties of a design in meeting its objectives. So they can be used to help determine whether further architecture design is required for the particular design area. Since the determination of *OCR* and *ICR* are semi-objective in nature, the acceptable level of risk becomes a semi-objective assessment and could be set to a defined acceptable level depending on projects. We choose to use 30% as a guide. Should either *OCR* or *ICR* be above this threshold, investigation or modelling at a more detailed level is required until the risk is reduced. Therefore, the following are the necessary conditions for the completeness of an architecture model:

- All known non-functional requirements have been identified through requirements refinement.
- All known architecture level requirements, functional and non-functional requirements, have corresponding linkages to design components through *AR*, directly or indirectly.
- All design components have corresponding linkages through *AR*.
- For each *AR*, *OCR* and *ICR* are below a defined threshold.

Example 2: This is an example from the EFT system to illustrate how to complete an architecture design using risk assessment. This example follows from the payment message design illustrated in Figure 7.5. When the Alarm Service *C6_0_0* is designed as a result of *AR15*, there is a hidden assumption that it would work for the peak loading scenario. This assumption has a risk associated with it. If the assumption is wrong, the design might not work. We quantify the risk factors using *OCR*. Also, there is also a gap in terms of how this mechanism could be implemented. In order to record multiple alarms, each alarm has to be identified using a unique key so that it could be associated with an expired payment message. The uncertainty of the design is a risk highlighted by *ICR*. Figure 8.2 shows this example and its design decision process based on risk assessments.

The UNIX system call *signal(2)* and UNIX function call *alarm(3)* support the setting of a single clock alarm only, but we need multiple alarms to be set simultaneously. Furthermore, we want to be able to identify a payment message when an alarm has sounded. These two design issues indicate implementation uncertainties that need to be investigated. We found that Top End middleware provides a timeout mechanism that would resolve these

8.3. Other applications of ARM

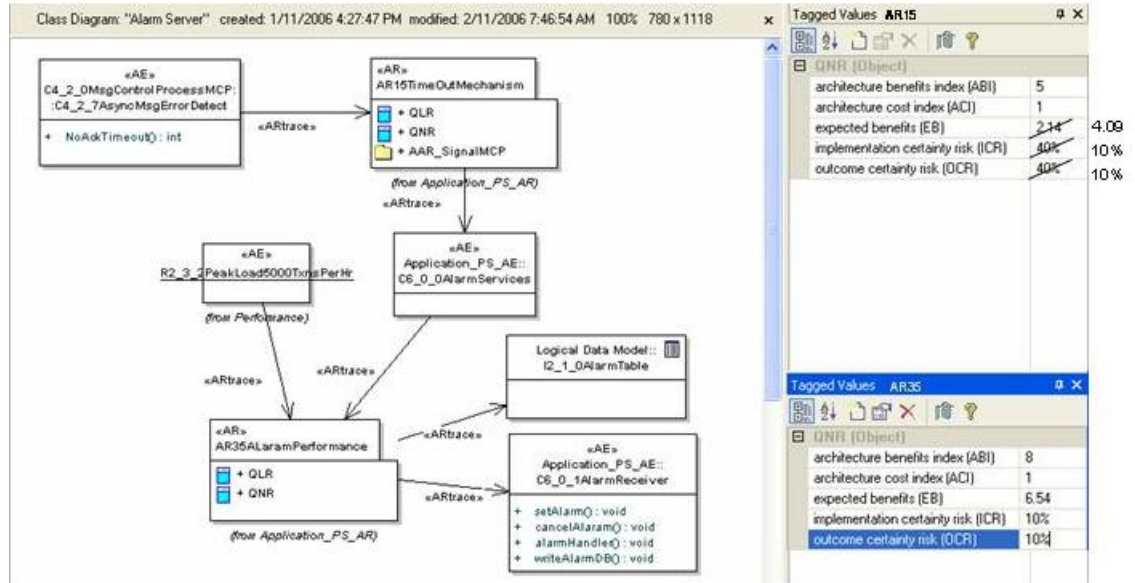


Figure 8.2: Risk Assessments of Alarm Services

two issues. Top End can handle multiple alarm settings and when a timeout is reached, Top End can return an identifier that is associated with the timeout message. Therefore in the architecture decision *AR35*, we create a Alarm Receiver design (i.e. *C6_0_1*). In this design object, we have a function to set the alarm through a Top End call, passing the process identifier, the object identifier and the event identifier as parameters. We also design a function to cancel the alarm, a function to handle the alarm when it has sounded, and a function to log the alarm into the database for persistent storage. At this stage, it is clear that the architecture design is implementable without much risks. *ICR* is therefore set to 10%.

The assumption that we initially make in *AR15* is that the Alarm Services is capable of handling the performance requirements. Since we don't know for sure, we have associated a risk factor in *AR15*. As shown in *AR15*, the outcome certainty risk is high (i.e. $OCR = 40\%$) because we expect each service call would involve setting a Top End timeout call as well as inserting a record into the Oracle database. At this stage, we do not know whether there would be any performance issue. In order to reduce the risk that the design outcome is acceptable, we need to investigate further into the design. In decision *AR35*, we use requirement *R2_3_2* peak load as an input to guide the decision. In order to ascertain that the use of a database table is a viable, a feasibility test was performed to ensure that 5000 records can be inserted and 5000 alarms can be set and reset by the Alarm Receiver within one hour. The feasibility test can simply be done by a prototype. The feasibility test demonstrated that such design should not create any performance problems. The *OCR* of *AR35* is therefore set to 10%.

We are now certain that the Alarm Services can be supported by table insertions and the Top End calls. As such, the *OCR* and *ICR* of *AR15* can also be reduced. This is reflected in the adjustments of *EB*, *OCR* and *ICR* in *AR15* in Figure 8.2. If the outcome certainty *OCR* at *AR15* cannot be reduced, it means that we don't know whether the design would work or not. If the risk of *AR35* is not at an acceptable level, we would not be certain that this design would need more investigation and design decomposition. Now that we are fairly certain that the design would work, i.e. the risk is low, then we do not need to go into further architecture design. The architecture design of this part of the system can be considered complete.

When architects design a system, *OCR* and *ICR* risk values are assigned to the *AR* nodes, usually along a top-down fashion. There are usually more uncertainties in the initial stage when architecture design is at the high-level because details of whether the decisions are valid are yet to be confirmed. As architecture design progresses, and more details become available, architects are more confident about the validity of the lower-level decisions. This confirmation from having more details requires that architects readjust the *OCR* and *ICR* at the higher-level decisions. Therefore, the risk level of a high-risk decision can now be updated to a lower risk level. This re-adjustment process serves to confirm the architecture design.

All requirements which involve high risk decisions, as indicated by *OCR* and *ICR*, should be part of architecture level requirements and need to be a part of the architecture design. Low risk functional requirements do not require architectural design because they do not have structural impact on architecture. Non-functional requirements usually have higher risk factors because (a) Non-functional requirements usually cut-across a large part of the system; (b) Non-functional requirements are usually competing with other non-functional requirements; (c) the impact of a design to satisfy a non-functional requirement requires more design attention. Therefore, all non-functional requirements should be considered in architecture modelling. This mechanism provides a way to define the scope of the architecture activities and allow measurement of completeness of the architecture outcomes.

We have chosen an arbitrary guide to establish a yardstick for determining an acceptable risk. In order to get a common consensus on an acceptable risk level, we undertook a survey [158]. Architects were asked as to what is an acceptable level of risk. 23.5% said that a 30% risk level was acceptable, 18.5% said that a 40% risk level was acceptable. However, 14.8% of respondents said that a 70% risk level was acceptable and 13.6% said that a 80% risk level was acceptable. It shows that either architects genuinely believe in a high risk architecture or they are inexperienced with risk assessment in architecture design and with the impact risk may have on design quality. More research is required in this

area.

8.3.2 Verifiability of architecture design

Architecture verification or evaluation is a quality assurance process to assess the soundness of an architecture design. A number of evaluation methods such as ATAM [78], SAAM [77] and ALMA [90] have been designed for such purposes. The need to verify architecture is recommended by IEEE standards [67]. They check different aspects of a software architecture to ensure its quality. During this process, architects are required to supply information, answer questions and justify the architecture design.

A common practice of verification in the industry is through peer review of design specification. Since design rationale is not systematically captured, architects who designed the system need to be present to support the verification process. However, it was found in our survey that 74% of architects could also forget the reasons behind the design [158]. Another issue is that architecture decisions should be verifiable during or after development, with or without the presence of the original architects. Architecture rationale is therefore vital in supporting such verification process. Some of the information that are captured in the AREL model and can be used to support architecture design verification are:

- Specifications of architecture drivers and motivational reasons.
- Specification of the architecture design objects.
- Documented architecture rationale to explain why the design can satisfy the requirement(s).
- Documentation of assumptions and constraints of architecture design.
- Traceable architecture rationale and elements.
- Documentation of design alternatives.

Architecture verification is supported by documented motivational reasons and design rationale. They can overcome the issue that architecture design cannot be verified easily because the design reasoning has not been captured. The documentation of design alternatives also support verification because it demonstrates that architects have explored different design options and assessed their appropriateness before selection.

8.4 Summary

In this chapter, we have presented the ARM method for architecture decision making. ARM complements other design methodologies by addressing the process to carry out design reasoning. This is a fundamental area that has often been omitted in architecture design. ARM requires architects to use qualitative and quantitative rationale to justify their decisions. Qualitative design rationale explores issues, arguments and options of a decision. Quantitative rationale quantifies the costs, benefits and risks of options. It enables architects to delineate architecture design from detailed design by way of obtaining acceptable risk levels in the architecture design. A number of advantages can be derived using ARM:

- Quantifies the choice of architecture decisions by explicitly compare the costs, benefits and risks of alternative design options.
- Define the scope of architecture design using risks as a guide.
- provide a means to support architecture verification.

As well as being useful to architecture development and verification, ARM facilitates the documentation of design rationale to support the maintenance process. Other uses of the AREL model are discussed in later chapters.

Chapter 9

Architecture rationale and traceability

In this chapter, we introduce the traceability methods to support change impact analysis and root-cause analysis for AREL. Through these traceability methods, we explore the reasoning support provided by AREL to explain why design objects exist and what assumptions and constraints they depend on. The results enable software architects to better understand and reason about an architecture design. Three traceability methods are discussed in this chapter and the EFT case study is used to demonstrate their applications¹.

System and software architecture design often involves many implicit assumptions [135] and convoluted decisions that cut across different parts of the system [109]. A change in one part of the architecture design could affect many different parts of the system. A simple shift of an implicit assumption might affect seemingly disparate design objects and such change impacts could not be identified easily. This intricacy is quite different from detailed software design where usually the design or program specifications are self-explanatory. At the system and software architecture level, there are a multitude of influences that can be implicit, complex and intractable. Without traceable design rationale, the implicit relationships between the design objects might be lost, therefore creating potential problems:

- The reconstruction of the design rationale through analysis might be expensive.
- Design criteria and environmental factors that influence the architecture might be unclear.

¹This chapter is based on our work published in [161].

-
- Business goals and constraints might be ignored.
 - Design integrity might be violated when intricately related assumptions and constraints are omitted.
 - Tradeoffs in decisions might be misunderstood or omitted.
 - The impact of the changing requirements and environmental factors on a system could not be accurately assessed.

Many of the argumentation-based design rationale methods represent the deliberations of design decisions [93, 26, 100] but they do not support effective design rationale retrieval and communication [139]. An effective design rationale model should support easy retrieval of the design rationale to explain why the associated design elements are created. It means that design rationale and design elements should be traceable.

Using the AREL model, two types of tracing are possible: (a) tracing an architecture design to understand the design dependency and reasoning; and (b) tracing the history of an evolving architecture design. Together they offer a number of advantages in system development and maintenance, which might otherwise be unattainable:

- It helps architects and designers to understand the reasoning of an architecture design.
- It allows architects and designers to analyse the change impacts of a design through forward tracing.
- It allows architects and designers to analyse the root-causes of a design through backward tracing.
- It supports design verification and maintenance.
- It retains design and decision history to help understand how and why a system has evolved.

Section 9.1 discusses relevant requirements and related design traceability work. Section 9.2 outlines the need for design rationale traceability and Section 9.3 proposes traceability techniques to address those needs. Section 9.4 provides examples to demonstrate how traceability techniques are applied.

9.1 Background

In this section, we examine the needs for a better way to recover design rationale to support design reasoning. We analyse existing design-rationale methods and requirement traceability methods to identify the challenges. From the perspective of a practising architect, we identify a number of uses cases for traceable design rationale.

9.1.1 Issues with design rationale

Researchers in the area of design rationale have argued that there is a need to improve the process to capture, represent and reuse design rationale. Perry and Wolf [119] suggested that as architecture design evolves, the system is increasingly brittle due to two problems: *architectural erosion* and *architectural drift*. Both problems may lead to architecture design problems over time if the underlying rationale is not available. Bosch suggested that as architecture design decisions are crossing-cutting and inter-twined, the design is complex and prone to erroneous interpretations without a first-class representation of design rationale [11]. As such, a design could be violated and the cost of architectural design change could be very high and even prohibitive. As discussed in earlier chapters, design rationale is important to support knowledge retention and facilitate the understanding of the architecture design. The form of design rationale representation is also important for it has to support structured and automated tracing.

9.1.2 Requirements and design traceability

Requirements traceability is the ability to describe and follow the life-cycle from requirements to their implementation, in both a forward and backward direction. For example, given a requirement, an architect might wish to find the design objects that realise the requirement in forward tracing. Gotel and Finklestein [51] distinguish two types of traceability: *pre-requirements specification* (Pre-RS traceability) and *post-requirements specification* (Post-RS traceability). They argue that wider range of informational requirements are necessary to address the needs of the stakeholders. This is an argument for representing contextual information to explain requirements and design. A survey of a number of systems by Ramesh and Jarke [129] indicates that requirements, design and implementation ought to be traceable. It is noted by Han [58] that traceability “provides critical support for system development and evolution”. The IEEE standards recommend that requirements should be allocated, or traced, to software and hardware items [67, 68].

In the development life-cycle, architects and designers typically have available to them

business requirements, functional requirements, architecture design specifications, detailed design specifications, and traceability matrix. A means to relate these pieces of information helps the designers maintain the system effectively and accurately. It can lead to better quality assurance, change management and software maintenance [149]. There are different aspects of traceability in the development life-cycle: (a) tracing requirements to design; (b) tracing requirements to source code and test cases; (c) tracing requirements and design to design rationale; (d) tracing evolution of requirements and design. Example methods and approaches to support requirements traceability are TOOR [120, 121], DOORS [145], Ramesh and Jarke [129], Huges and Martin [66], and Egyed [38].

Some traceability approaches incorporate the use of design rationale in a limited way. Haumer et al. [60] suggested that the design process needs to be extended to capture and trace the decision making process through artefacts such as video, speech and graphics. Since such information is unstructured, making use of it can be challenging. Ramesh and Jarke [129] proposed a reference model for traceability. It adopts a model involving four traceability link types, two of which are relevant here. The rationale and evolution link types are introduced to capture the rationale for evolving design elements. The rationale link type is intended to allow users to represent the rationale behind the objects or document the justifications behind evolutionary steps. Since its focus is on evolving design, other kinds of design reasoning such as the design tradeoffs are not considered.

The traceability methods and approaches described here make a fundamental assumption that traceable requirements could have the explanatory power to help designers understand the system. We argue that this is insufficient. Design rationale that are implicit to the decision process are absent. Thus, design rationale such as assumptions and constraints would need to be reconstructed even though source code or design objects can be traced back to requirements.

9.2 Traceability of architecture rationale

Being able to trace design and requirements to design rationale helps architects to understand, verify and maintain architecture design [173]. It supports the conscious reasoning of architecture design. There are many use-cases of traceable architecture design rationale in the software development life-cycle [86]. The following cases describe how traceable architecture design and design rationale support the software development life-cycle.

- Explain Architecture Design - the reasoning and the decisions behind a design can be traced because they are linked to the design objects through a causal relationship.

As such, the being of a design object can be explained by its associated design rationale.

- Identify Change Impacts - when a requirement or a decision is subject to change, the ripple effect of such a change should be traceable in order to analyse the change impacts to various parts of the system.
- Trace Root Causes - when there is a software defect in a system, it might be due to many reasons and the root causes have to be analysed and identified. Some of the causes might be to do with conflicting requirements, constraints or assumptions, it requires design rationale to help explain and identify them.
- Verify Architecture Design - the retention of traceable design knowledge supports independent verification of an architecture design. It supports the verification of the architecture design without the presence of the original designers.
- Trace Design Evolution - when decisions were made to enhance the system, the design rationale of each subsequent change could explain the evolution of the design object. Such a reasoning history is useful because design assumptions and constraints are explicitly represented and can be used as a context for previous and current decisions.
- Relate Architecture Design Objects - architecture design objects, which are seemingly disparate, may be related by a common requirement, constraint or assumption. For instance, the friendliness of a user interface design may be compromised because of a constraint on embedding a security feature. If the rationale of the compromise is not explicitly stated, an update to the security feature might not trigger a revisit to the design of the user interface.
- Analyse Cross-cutting Concerns - architecture deals with cross-cutting concerns especially in non-functional requirements. These concerns often require tradeoffs at multiple decision points. The reasons behind such tradeoffs explain a lot as to why and how the decisions have been made. A traceable architecture with design rationale could relate otherwise disparate requirements and design objects that are part of the cross-cutting concerns.

There is currently a lack of traceability methods to accomplish all these tasks. The “why such a design” question cannot be answered because the current methods do not relate design objects to design reasoning effectively [62]. On the other hand, common industry practice relies heavily on the reconstruction of design rationale and the manual tracing of design specifications. This situation can be improved by a traceable AREL model.

9.3 Traceability support

General traceability provides a means to retrieve related development artefacts such as system requirements and design objects. However, architecture verification and enhancement are better served if design rationale is traceable as well. Without design rationale, implicit assumptions, constraints and design reasoning can only be derived by second-guessing, or they might be overlooked altogether, and as such the risk that architects are not able to correctly verify or maintain the architecture design exists.

The AREL model provides a mean to represent the causal and the dependency relationships between design rationale and design objects. Such representation provides an additional perspective to understand the relationships between design objects and their design rationale. The tracing is therefore characterised by the inter-connections of design rationale and design objects using these causal relationships. Constraints, assumptions, decisions and tradeoffs can be traced to disparate parts of a system that they influence. There are three possible tracing techniques: forward tracing, backward tracing and evolutionary tracing. They rely on the AREL and eAREL links $\ll\text{ARtrace}\gg$, $\ll\text{AEsupersede}\gg$ and $\ll\text{ARsupersede}\gg$.

- Forward Trace - the purpose of this trace is to support the impact analysis of the architecture design. For example, an architect might use forward trace to find all the design objects that originate from a requirement, and the reasons behind their design. Given an architecture element AE_1 , all architecture rationale and architecture elements that are downwardly caused and impacted by AE_1 can be traversed.
- Backward Trace - the purpose of this trace is to support the root-cause analysis of the architecture design. For example, an architect may use it to analyse what requirements and factors influence a design object and the reasons why. For a given architecture element AE_1 , all decisions and other architecture elements that AE_1 depends on, directly or indirectly, are traced and retrieved. This enables architects and designers to retrieve the root causes such as requirements and assumptions of AE_1 , and to understand and analyse design justifications leading to AE_1 .
- Evolution Trace - the purpose of this trace is to support the analysis of the evolution of a decision or an architecture element. Given a AR_1 or AE_1 node, the history containing previous versions of the node is retrieved.

The traceability support based on the AREL model has two characteristics. Firstly, it provides an automated mechanism to support forward and backward tracing. An architect could specify the source of the trace (i.e. an AE), and the system would traverse all related

elements in the AREL model. Enterprise Architect [150] is the UML design tool we use to capture the AREL model, the results of a trace are created as UML diagrams.

Secondly, traceability can be selective based on architecture elements' classifications. Using the classification of the architecture elements by viewpoints and business drivers (see Section 6.3), architects could limit the trace results to the specified types of architecture elements. For instance, a trace could traverse only those *AEs* that are specific to the data viewpoint. The classification-based traceability could reduce information overload when analysing trace results. This is an initial attempt to implement the traceability scoping with classified architecture elements. It demonstrates that classification is useful in restricting the scope of the trace results.

9.4 AREL and eAREL traceability applications in a case study

Using the case study as a basis, we demonstrate the advantages of being able to explain and trace the architecture from the design reasoning perspective. It provides important information that are missing from most design specifications in which the focus is on design organisation, interface and behaviour. In the demonstration, we choose the processing services and specifically the payment message processing layer to illustrate the rationale-based architecture model and its traceability below. We demonstrate the traceability with the AREL tool-set. The tool-set comprises of an UML-based tool called Enterprise Architect and a tool for AREL tracing. A detailed description of the tool implementation is in Chapter 11.

9.4.1 Design rationale representation

This section recapitulates an example from the case study in Chapter 7 to demonstrate the traceability techniques. The EFT system had extensive design specifications to document what had been designed. However, they do not systematically document the assumptions, constraints and rationale of the design. Its specifications document the design of the software modules, their interfaces and behaviour. Designers who were not originally involved in the design would find it difficult to understand the design intentions. Thus, when system modifications are required, it would be hard to determine which parts of the system are affected. We use the payment messaging mechanism as an example to illustrate how a traceable AREL model captures this knowledge.

9.4. AREL and eAREL traceability applications in a case study

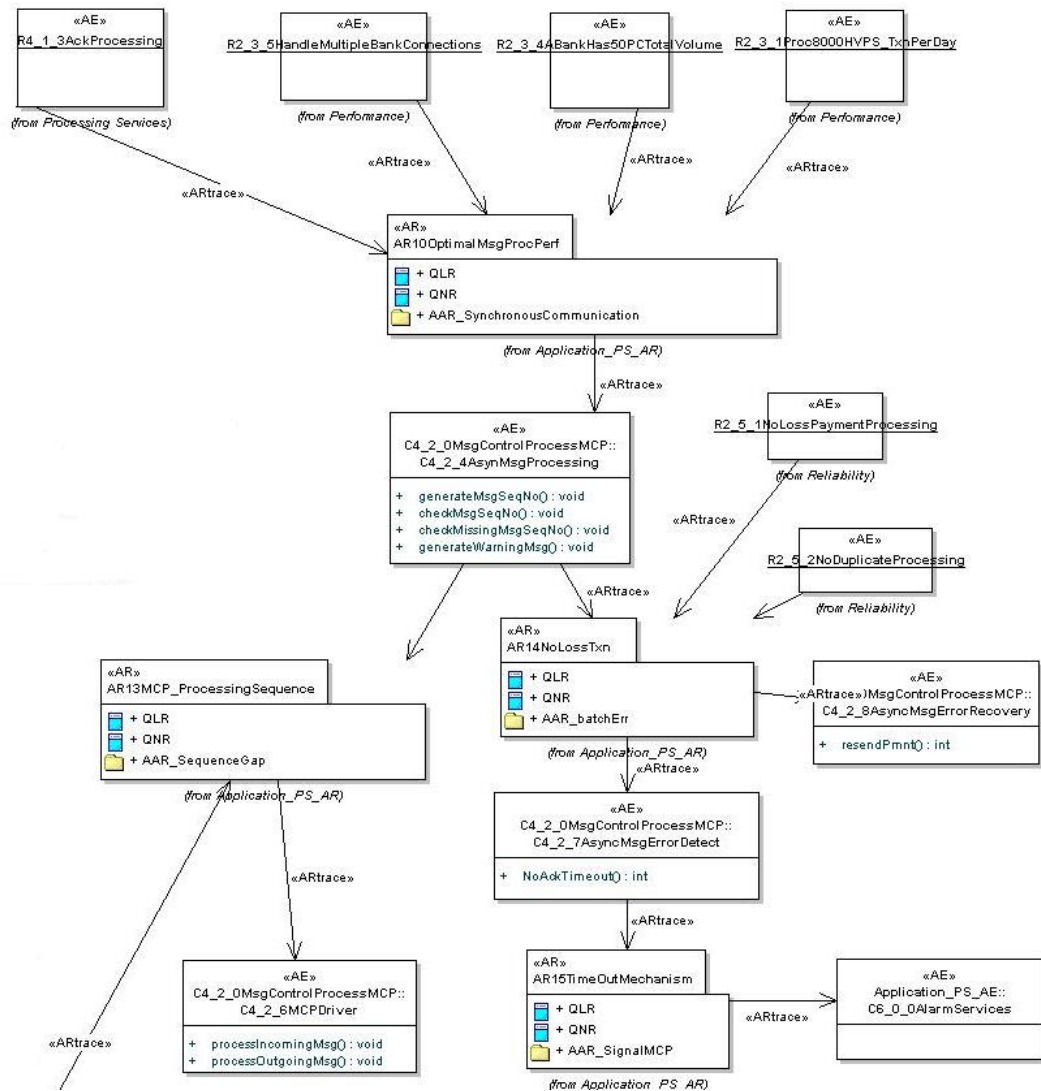


Figure 9.1: Asynchronous message processing decision and its design impact

Figure 9.1 is an example chain of decisions. In this case, the architect was to choose a payment messaging protocol to support payment message exchange between the central bank and the participating banks. At this decision point (i.e. *AR10*), the issue is to choose between synchronous and asynchronous messaging protocols. The motivational reasons specify that the decision has to take into considerations: (a) the handling of message acknowledgement; (b) the processing of at least 8000 payment messages per day, 50% of which would come from one bank; and (c) the system has to scale to process a much higher volume and so the processing unit must be able to handle multiple bank connections simultaneously.

The reasons or justifications to have chosen asynchronous over synchronous message processing are encapsulated in *AR10*. The *QLR* contained within *AR10* captures the

details of its design rationale as shown in Figure 9.2. When an asynchronous payment message is sent to the receiving party, an acknowledgement message from the receiving party is not expected immediately but eventually. The acknowledgement should come from the receiving party to indicate whether the message has been received correctly. Therefore, there is no blocking between the two communicating processes and they can continue with other work. Synchronous message processing requires the sender to block and wait for the remote party to respond. The concurrency of such a design is lower. This explanation is captured in *AAR* within *AR10*. By comparing the two methods, it was decided that asynchronous messaging would provide better processing efficiency than synchronous messaging. As a result of the decision *AR10*, the component *C4-2-4* was created.

QuR (Object)	
assessment_decision	Async mdg processing because a single process can't
assumptions	multiple connections in a single process
constraints	
date	Dec 1995
decision_maker	AT
history	
implications	check missing message by using seq number
issue	should async / sync messaging be implemented?
non-risks	no missing or duplicated messages
risks	complexity
strengths	fast turnaround in message processing
tradeoffs	performance against complexity
weaknesses	message matching requires DB support

Figure 9.2: Details of the design rationale AR10

The constraint or the implication of the *C4-2-4* design is that all messages must have a mandatory, unique and sequential message identifier. The design has to deal with a scenario where messages must not go missing even though they can be out of order. As a result, a number of decisions (i.e. *AR13*, *AR14* and *AR15*) have to be made to deal with this new constraint.

The choice of this design addresses the issue of payment processing efficiency and it has implications on different aspects of the system such as complexity and reliability. These intricate design relationships could be easily understood if design rationale are explicitly recorded and linked. Trying to reconstruct this relationship without captured design rationale can be difficult if the designer does not have in-depth knowledge of the system. AREL uses *AR* and *«ARtrace»* to capture the knowledge so that designers can understand the motivational reasons, the design rationale and the design outcome.

9.4.2 Forward and backward tracing

Architects often need to analyse change impacts and root-causes to understand the design during system enhancements. They rely on tracing the design to the requirements, assumptions and constraints to explain why and how the design is constructed. If this information is implicit, its influence could be omitted or misinterpreted because (a) the motivational reasons which influence disparate parts of the design would not be apparent; (b) the design rationale that justifies inter-related design objects would be missing. We resolve this issue by providing a tool to do forward tracing for analysing change impacts and backward tracing for analysing root causes.

Forward tracing

In this section, we discuss the methods for forward tracing in AREL and demonstrate how to carry out forward tracing. Forward tracing in AREL is the traversal of the AREL graph to produce a subset of the resulting AREL graph forward from a specified *AE* node. The following steps are involved in the forward tracing of AREL:

1. Specify a source *AE* node, say AE_1 , where tracing is to begin.
2. Specify the scope of the result set using the viewpoint classification (e.g. Application Viewpoint).
3. For all the descendants of the source node AE_1 , traverse the *AE* and *AR* nodes recursively to all the leaf nodes by following the *ARtrace* links. Since the AREL model is an acyclic graph, the traversal will terminate when all leaf nodes have been traversed.
4. All the *AE* nodes with the specified viewpoint classification(s) will be retained in the result set.
5. All the *AE* nodes that are not specified in the classification, but are located between AE_1 and its descendant *AE* nodes that are in the classification are retained. This is because if these unspecified nodes are not traversed, then the causal chain will be broken and some of the required *AE* nodes downstream from them would be omitted.

For example, we want to investigate what the change of requirement $R4_1_3$ would do to the architecture design and why. So we select the source of the trace (i.e. $R4_1_3$) and carry out tracing from the source. All the $\ll ARtrace \gg$ links that are connected to $R4_1_3$

9.4. AREL and eAREL traceability applications in a case study

are traversed and the connected *AR* and *AE* nodes downwards from it are retrieved. We use an automated tool to implement the traceability method and its results are created in a UML repository (see Chapter 11). Figure 9.3 is the result of the forward tracing based on the case study.

Two decisions that are directly affected by *R4-1-3* are *AR10* and *AR11*. *AR11* is a decision about how payment messages are to be composed and decomposed for transmission. *AR10* is a decision about which messaging protocol should be used with the key issue being performance. As discussed above, asynchronous messaging protocol was selected over synchronous messaging protocol.

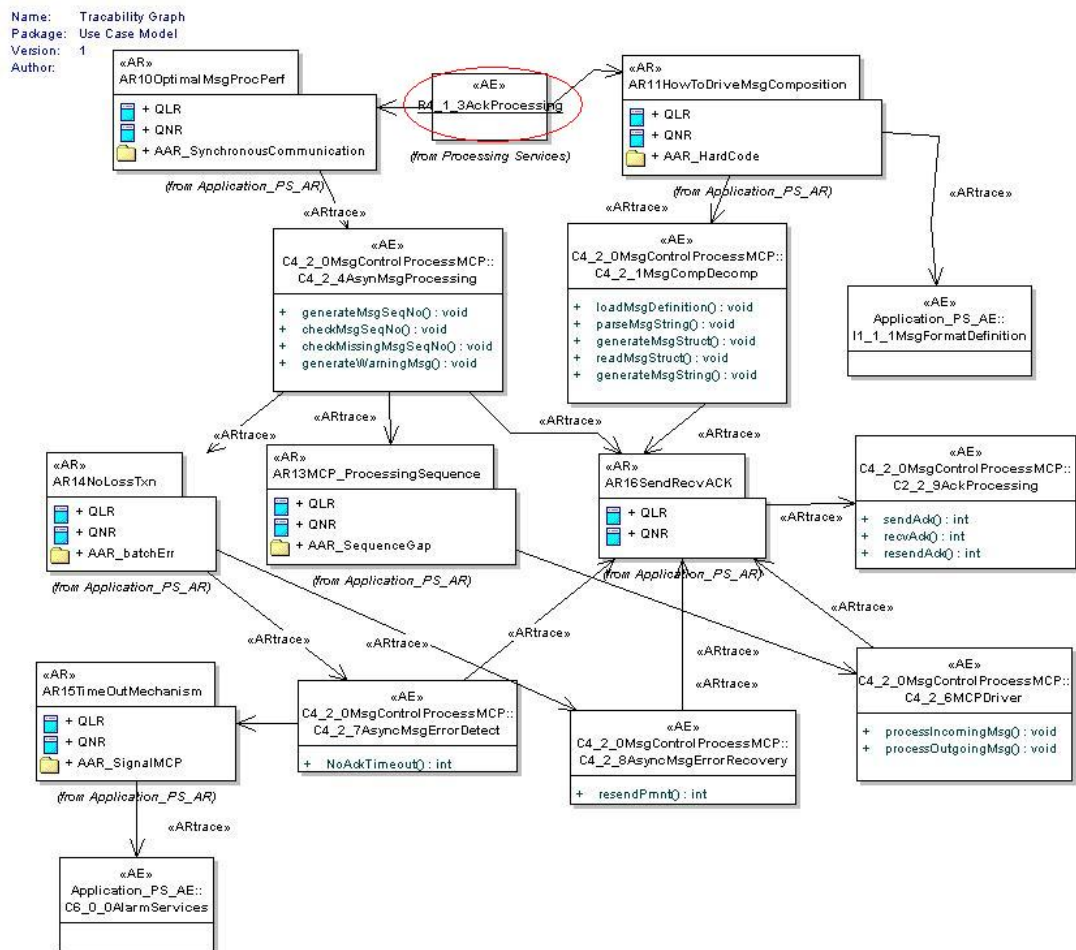


Figure 9.3: Forward tracing for impact analysis

The selection of the asynchronous mechanism means that a number of constraints and requirements to support the design must be in place. The criteria are (a) messages need to be sequenced and (b) a protective mechanism is available to guarantee that there is no loss of payment messages. Following the «ARtrace» link from *AR10*, we find *AR14*

and *AR13* which consider message sequencing and missing payment message detection respectively.

Normally a payment acknowledgement should arrive shortly after a payment has been sent, but if it does not arrive within the allowed elapsed time, there ought to be a time-out function to indicate that the acknowledgement is missing. *AR15* is a decision to implement such an Alarm Server to notify any overdue acknowledgement.

This example of forward tracing shows the impacts of a requirement to all the design objects and decisions that are affected by it. The design decision *AR10* creates a chain effect on the resulting design branching out into many decisions and design outcomes. AREL forward tracing has a number of characteristics:

- Impact analysis - when an architecture element AE_1 is specified, the architecture rationale (AR) and the architecture elements (AE) that depends on AE_1 are retrieved. Since AREL is a causal model, it implies that all resulting architecture decisions and elements are directly or indirectly impacted by AE_1 .
- Selection by viewpoints - the classification of architecture elements by architecture viewpoints allows architects to hone in on specific results by specifying what is required.
- Graphical representation - since AREL is implemented in UML, trace results are also represented in UML diagrams.

Backward tracing

In this section, we discuss the methods for backward tracing in AREL and demonstrate how to carry out backward tracing. Backward tracing in AREL is the traversal of the AREL graph to produce a subset of the resulting AREL graph backward from a specified AE node. The following steps are involved in the backward tracing of AREL:

1. Specify a source AE node, say AE_1 , where tracing is to begin.
2. Specify the scope of the result set using the viewpoint classification (e.g. Business Viewpoint).
3. For all the ancestors of the source node AE_1 , traverse the AE and AR nodes recursively to all the root nodes by following the reverse direction of the $ARtrace$ links. Since the AREL model is an acyclic graph, the traversal will terminate when all root nodes have been traversed.

4. All the *AE* nodes with the specified viewpoint classification(s) will be retained in the result set.
5. All the *AE* nodes that are not specified in the classification, but are located between AE_1 and its ancestor *AE* nodes that are in the classification are retained. Similar to forward tracing, if these unspecified nodes are not traversed, then the causal chain will be broken and some of the required *AE* nodes downstream from it would be omitted.

Using the example from the previous section, we demonstrate backward tracing of the root-causes of the alarm server. Consider the Alarm Server *C6_0_0*, if we want to understand why it is there, and what requirements have motivated the architects to create it, then we trace backwards from it. The resulting AREL graph is shown in Figure 9.4. It shows all the requirements, design elements and design decisions that lead to the creation of *C6_0_0*.

The reason for the creation of *C6_0_0* is due to the need of a timer mechanism (*AR15*) to support asynchronous error detection *C4_2_7*. The timer would time-out and send a notification if a payment message has not been acknowledged within a specified time. The justification in *AR15* specifies that this ought to be a separate server process because there is a technical constraint to implement time-out in the payment processing process itself. By tracing further backwards, we understand that the need for such implementation is due to the *No Loss of Payment Transaction* requirement. This requirement comes from *R2_5_1* as well as the implementation of *C4_2_4*. *C4_2_4* is in turn concerned with the performance issue of the payment system.

If an architect wants to assess the possibilities of enhancing the timer server, the first task is to understand the causes of such a design. These causes are intricate because there are underlying constraints and assumptions in a chain of dependency. In this case, the multitude of constraints are performance (*AR10*), reliability(*AR14*) and technical implementation(*AR15*). A change in the design must address all these inter-connected causes and constraints at the same time.

Similar to forward traceability, backward traceability supports results scoping by viewpoints. The resulting trace is a subset of the AREL model. Backward traceability supports root-cause analysis by retrieving architecture rationale and architecture elements for which the design element in question directly or indirectly depends on.

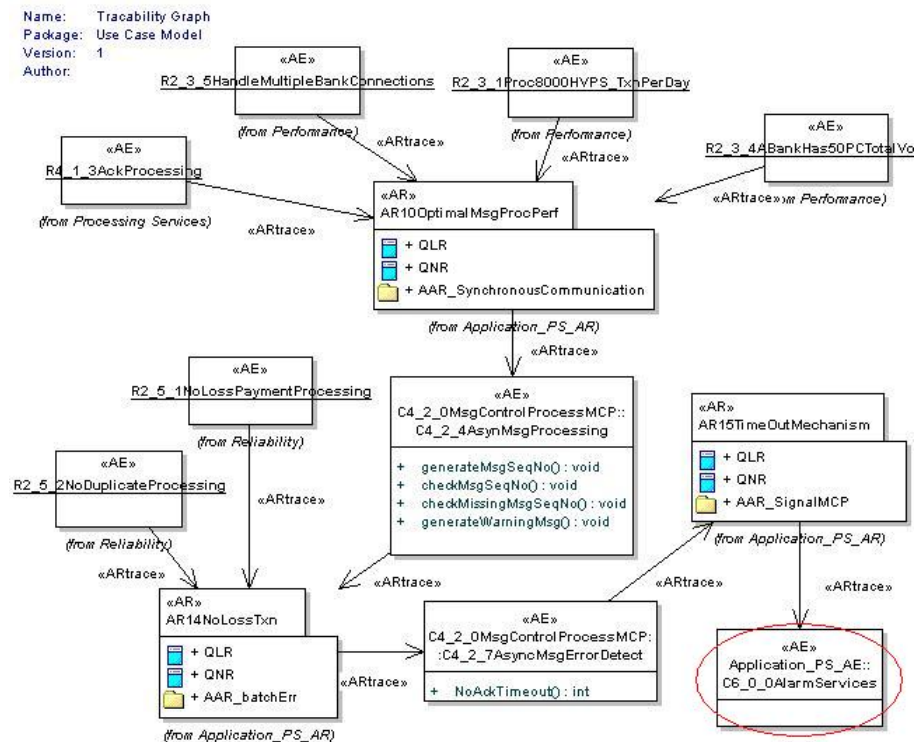


Figure 9.4: Backward tracing for root-cause analysis

9.4.3 Tracing architecture design evolution

Since architectures can have a long live-span and are subject to enhancements and adaptations over time, the view of the current architecture design does not necessarily provide all the information required for decision making. We use eAREL as a supporting mechanism for evolution tracking. It provides a means to track changes of the design and the decision making over time. Let us consider architecture element $C4_2_5$ for MAC Processing. In Figure 9.5(a), assuming $I1_1_5$ and $R2_4_6$ are the new requirements to support authenticity of the payment message using an encryption mechanism. The implementation requires the MAC code to be encrypted with a secret key to prove its authenticity. So the original design $C4_2_5$, which only supports clear-text MAC, would require some modifications. We decide to keep its original design history.

A copy of the original design, i.e. version 1 of $C4_2_5$ is archived and a relationship of the stereotype $\ll\text{AESupersede}\gg$ links it with version 2 of the $C4_2_5$, which is in the current AREL model. This is shown in Figure 9.5(b). Since version 1 of $C4_2_5$ MAC has now been replaced, its $\ll\text{ARtrace}\gg$ links to $AR12$ and $AR13$ are now obsolete, therefore these links are made non-current. New $\ll\text{ARtrace}\gg$ links are created to be related to

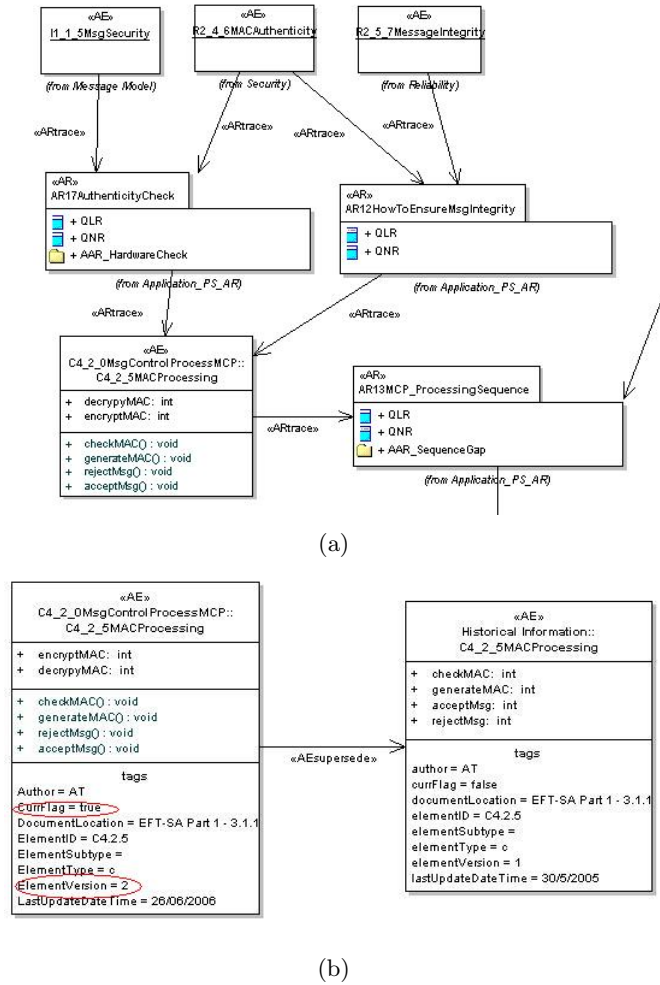


Figure 9.5: (a) MAC processing (b) Superseded architecture element

AR12 and AR13 for version 2 of the C4_2_5.

In the above example, we have demonstrated how a design can be superseded by its replacement. The supersession of AR works in the same way. eAREL can be useful in many cases. For instances, an architect may wish to investigate the extent of design changes in a particular release, or investigate the impact of a historical change, or simply understand the design history as background information. The architect can follow the «AESupersede» links to recover the previous designs. The tracing of historical changes to the architecture design begins from an initial AR or AE in AREL, and then trace through its history using «ARsupersede» or «AESupersede» links to the older versions of AR or AE. Finally, the inter-relationships between historical elements and rationale can be traced through non-current «ARtrace» links.

9.5 Discussion

In the empirical study (Section 7.2.4), it was found that the ability to trace architecture design and its design reasoning are important in software architecture. It enables an architect to reason with complex design and to support software maintenance if the knowledge cannot be remembered.

As commented by the experts who participated in the empirical study, the usefulness of this method depends largely on the cost-benefit of its implementation and how well an organisation adopts it. The cost of capturing the traceability relationships must be kept as low as possible, and the capture process itself must not impede the design process. This is yet to be tested in large-scale projects but it is anticipated that the benefit would exceed the cost in such projects. There is a major assumption in employing this method: the traceability relationships are captured perfectly and the design reasoning are documented thoroughly. This assumption depends on how well an organisation adopts the method and if the architects involved believe in design knowledge retention and use it diligently.

9.6 Summary

In this chapter, we have discussed the need for traceable design reasoning. We have identified three ways to trace an AREL model. Forward tracing supports impact analysis. Given a requirement, the design objects and design rationale that are directly and indirectly depended on and impacted by it can be traversed. Backward tracing supports root-cause analysis. Given a design element, its causes such as requirements, assumptions, constraints and design rationale can be traversed. Evolution tracing supports the traceability through the evolution of an architecture element or an architecture rationale. These methods facilitate the understanding of architecture design by allowing architects to trace the design with reasoning support.

AREL and its traceability are supported by the AREL Tool and the UML tool, Enterprise Architect (see Chapter 11 for details). The architecture viewpoints which classify the architecture elements in AREL can be used to scope the trace results. Using the EFT system as a case study, we have demonstrated how the architecture design can be traced.

Chapter 10

Architecture decision dependency and causality

Research into design rationale in the past has focused on argumentation-based design deliberations. These approaches cannot be used to support change impact analysis effectively because the dependency between design elements and decisions are not well represented and cannot be quantified. In earlier chapters, we have described change impact analysis using qualitative and traceability methods. In this chapter, we extend the AREL model to represent quantifiable causal relationships between architecture design elements and decisions. We apply Bayesian Belief Networks (BBN) to AREL, to capture the probabilistic causal relationships between design elements and decisions. We employ three different BBN-based reasoning methods to analyse design change impact: predictive reasoning, diagnostic reasoning and combined reasoning. These methods support the prediction and diagnosis of change impacts in the architecture design in a quantitative manner [83].¹ We illustrate the application of the BBN modelling and change impact analysis methods by using the case study illustrated in Chapter 7.

10.1 Background

As discussed in early chapters, design rationale has many benefits and it can be used to verify and trace the design of a system as well as to support its enhancements. The approach outlined in this chapter is to represent quantitatively the sensitivity to changes

¹This chapter is based on our work published in [162, 163]. Nicholson has contributed to applying BBN in AREL as well as the writing in the papers. Han and Jin have contributed to the development of the papers.

and the dependencies between architecture elements and architecture rationale in an AREL model.

10.1.1 Related work

Fenton and Neil [40] believe that management decision support tools must be able to handle causality, uncertainty and combining different (often subjective) evidence. Hence they suggest to use a solution based on BBNs. BBNs have been used in many applications (see [83] for a recent survey) including a causal model to detect user interactions with user interface in safety-critical systems which are prone to human errors [45]. A design reasoning approach has been proposed by Zhang and Jarzabek to use BBN to reason about architecture design decisions on quality attributes [177]. The purpose of this work is to use BBN as an aid to evaluate the architecture design for decision making.

BBNs [116, 73] are graphical models for probabilistic reasoning, which are now widely accepted in the AI community as practical and intuitively appealing representations for reasoning under uncertainty. Given the BBN representation of AREL, we show how BBN reasoning algorithms can be used to reason about change impacts. More specifically, architects can use BBN to undertake *what-if* analysis, to predict and diagnose impacts given complex combinations of requirement and design changes. This approach has the following advantages:

- We represent the AREL model as a BBN. The BBN graphical structure models and quantifies the causal relationship between architecture design decisions and architecture elements.
- Based on the AREL causal relationship in a BBN model, we employ three different probability-based reasoning methods to carry out change impact analysis:
 - Predictive Reasoning - predict what design elements and how likely they might be affected if one or more architecture elements are to change (e.g. requirements).
 - Diagnostic Reasoning - if one or more architecture elements (e.g. design object) are to change, diagnose what might be the causes of such change and the extent of their influence.
 - Combined Reasoning - by combining the use of predictive reasoning and diagnostic reasoning, reason about the ripple effect of the likely changes to the system through diagnosing the causes and predicting the effects.

10.1.2 Introduction to Bayesian Belief Networks

Bayesian Belief Networks (BBNs) are a well established method in the Artificial Intelligence community for reasoning under uncertainty, using (i) a graphical structure to represent causal relationships and (ii) probability calculus to quantify these relationships and update beliefs given new information.

A BBN is a directed graph with edges connecting nodes with no cycles (i.e. a directed acyclic graph). Its nodes represent random variables and its edges represent direct dependencies between variables. In theory, variables can be discrete or continuous, but in this work we construct models with discrete variables only. Nodes representing discrete variables have two or more discrete states, representing the possible values the variable may take. For instance, a discrete Boolean node may represent whether or not a patient has lung cancer with the two states *True* and *False*. Nodes without parents are called *root nodes* and have an associated *prior probability* distribution. Each node with parents has a *conditional probability table* (CPT), which, for each combination of the values of the parents, gives the conditional probability of its taking these values. Thus the CPT can be considered to quantify the strength of the causal relationships.

Let us illustrate these basic elements of a BBN – the nodes, arcs and prior and conditional probabilities – with a very simple example from the medical domain [83]. Suppose that a physician wants to reason about the chance that a patient presenting with a cough has lung cancer. The first relevant causal factor needing to be established would be whether the patient is a light, heavy or non-smoker. A possible diagnostic tool would be to take an Xray of the lungs. Figure 10.1(a) shows a BBN model for this example. The example illustrates that a heavy smoker has a higher probability of contracting lung cancer than a non-smoker. If a person has lung cancer, then there is a high probability that the person would cough, and there is a high probability that the lung cancer would be detected by the X-ray.

Once a BBN model has been constructed, posterior probabilities – often referred to as *beliefs* – can be computed. Figure 10.1(b) shows the physician’s prior beliefs before any observations or evidence are taken into account. The marginal probability distribution is represented by the bars in the diagram. Users can set the values of any combination of nodes in the network and this newly inserted evidence propagates through the network through computing their *marginal probabilities*, producing a new probability distribution over all the variables in the network. An explanation on the computation of marginal probabilities can be found in [83, 73, 117]. There are a number of efficient exact and approximate inference algorithms for performing this probabilistic updating [117], providing a powerful combination of predictive, diagnostic and explanatory reasoning. For example,

Figure 10.1(c) shows the physician’s beliefs after diagnostic reasoning given that the patient has a cough, while Figure 10.1(d) shows how the chance of the patient having lung cancer (and hence a positive lung x-ray) increases substantially if the patient is a heavy smoker (an example of predictive reasoning).

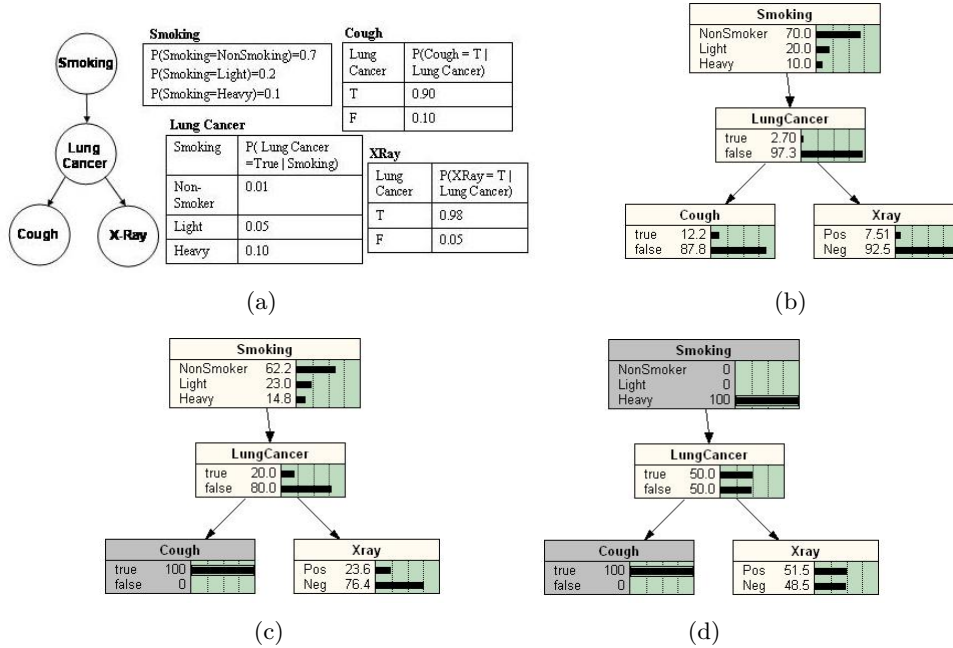


Figure 10.1: A medical example: (a) BBN nodes, arcs and CPTs; (b) BBN graph without evidence; (c) patient has a cough (diagnostic reasoning); (d) patient has a cough and a heavy smoker (both predictive and diagnostic reasoning).

The reader is referred to [83] for a more detailed introduction to the fundamentals of BBNs. The following sections explore how AREL models can be represented by a BBN (Section 10.2) to provide causal reasoning under uncertainty for change impact analysis (Section 10.3).

10.2 Building a BBN to represent an AREL model

Representing an AREL as a BBN enables prediction and diagnosis of change impact to the system when certain elements of the architecture have changed. For our modelling purposes, we are interested in the causal relationships between variables representing architecture design rationales and elements. It is generally accepted that building a BBN for a particular application domain – commonly called “knowledge engineering” – involves three tasks [83]:

1. identification of the important variables, and their values, which become the nodes and their states in the BBN;
2. identification and representation of the relationships between nodes, which means deciding which nodes should be connected by the directed edges; and
3. parameterization of the network, that is to determine
 - (a) the prior probabilities for each root node in the network and
 - (b) the conditional probability tables associated with each non-root node, which quantify the relationships between nodes.

We now show how to build a BBN to represent an AREL model by looking at each of these steps in turn.

10.2.1 Nodes: Representing architecture elements and decisions

Recall that an AREL model contains two types of components, Architecture Element (*AE*) and Architecture Rationale (*AR*). There is a direct mapping from these AREL components to the two types of nodes in the BBN, also called *AE* and *AR* nodes.

In an architecture design, we require the initial inputs into the design. These inputs are the motivational reasons such as the functional requirements, non-functional requirements, assumptions or constraints. These *AE* nodes are represented as root-nodes in AREL. The *AE* nodes which are the results of the design decisions are either the branches or the leaf nodes in AREL. The *AR* nodes represent the decisions made in the architecture design, and they cannot be root-nodes nor leaf-nodes.

The BBN implementation for AREL uses probability to represent the possible change impact to the architecture design. If a design decision is to change, then it is probable that those design elements which depend on the validity of this design decision are likely to become unstable and subject to change as well. In order to represent dependency, first we have to represent the *states* of the *AE* and *AR* nodes. Both the *AE* and *AR* nodes in the BBN have two states each. The two states of an *AE* node represent whether or not the *AE* node is stable or likely to change:

- *Stable* – *AE* is not subject to change
- *Volatile* – *AE* is subject to change

If an *AE* becomes volatile, then the decision which uses the *AE* as an input may not be as reliable because it is probable that a change in the underlying *AE* input could invalidate the design decision. As such, we need to represent the possible validity of the *AR* decision. The two states of an *AR* node are:

- *Valid* – the decision is valid
- *Invalid* – the decision is invalid

10.2.2 Edges: Representing causal relationships

Each $\llcorner\text{ARtrace}\gg$ link in an AREL model is represented by an edge in the BBN. As AREL links are defined to preclude directed cycles, this means that the corresponding BBN is valid, i.e. it is a directed acyclic graph. When an AREL model is formed, it is based on interconnected nodes with the fundamental structure of $AE_i \rightarrow AR_j \rightarrow AE_k$. This structure represents the directional causal relationship between the nodes starting from AE_i . *AEs* as motivational reasons drive decisions to create high-level design. The high-level design are refined and decomposed into finer-grained architecture design objects through making further decisions for the different aspects of the architecture such as data models, application models and technology models. This process of making decisions, creating and modifying architecture elements is carried out in a progressive way until the architecture is complete [159].

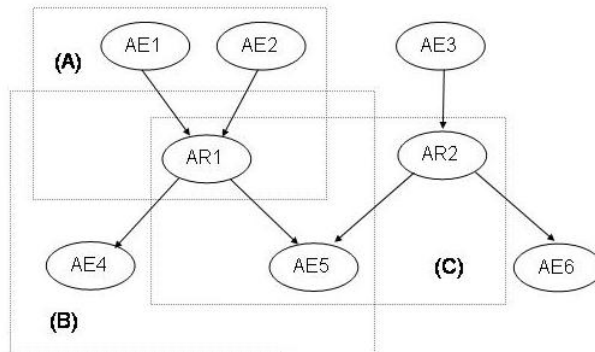


Figure 10.2: Basic Forms of AREL Relationship in the BBN

The three basic forms of relationships that are present in the BBN representation of the AREL model are shown in Figure 10.2:²

²In the BBN literature, (A) and (C) are referred to common effect structures, while (B) is a common cause structure. We distinguish between (A) and (C) here because of the difference for our purposes between the *AE* and *AR* nodes as a common effect.

- A. *AEs* as causes of an *AR* : *AE1* and *AE2* are converging inputs of *AR1*;
- B. *AEs* as effects of an *AR* : *AE4* and *AE5* are effects of *AR1*;
- C. Multiple *ARs* as causes of an *AE* : *AE5* depends on both *AR1* and *AR2*.

The fundamental $AE_i \rightarrow AR_j \rightarrow AE_k$ structure reflects the fact that a decision (represented by an *AR* node) depends on its input *AEs*, and in turn can affect other *AEs* as the outcomes of the decision. At this stage, we can see intuitively that the edges in the BBN can represent the important causal relationships between *AEs* and *ARs*, such as:

- while the input *AE* nodes are stable, a decision will probably remain valid;
- if the input *AE* nodes have changed, then it is likely that the conditions under which the decision was made are no longer valid and a reassessment of the inputs is necessary to validate the decision;
- the validity or otherwise of an *AR* node affects the stability of a consequent *AEs*.

These relationships are defined and quantified through the specification of the CPTs. We describe this aspect of the BBN knowledge engineering process next, first for the root nodes, and then for non-root nodes in relationships A, B and C above.

10.2.3 Probabilities: Quantifying the causal relationships

Root Nodes

Each root node in a BBN (i.e. nodes without parents) has an associated prior probability. As mentioned above, in a BBN representation of an AREL model, all root nodes must be of type *AE*, and represent either functional or non-functional requirements, basic designs or systems constraints. The architect must therefore estimate the prior probabilities for the different types of *AE* nodes. Prior probabilities assigned to the *AE* root nodes indicate the likelihood of change of these primary requirements, design or constraints.

If the *AE* is a requirement, one could estimate whether this requirement is volatile or stable by assessing whether the requirement is likely to change over time. For instance, an industry standard or a regulation is not negotiable and therefore the probability of its being stable is 100% and the probability of its being volatile is 0%. Another example is the requirement to produce data through different interface file formats such as XMI,

csv etc. The list of interface formats is likely to be extended as the system evolves, so the architect may estimate that the probability of this requirement being stable is 80% and the probability of its being volatile is 20%. Design elements which are root nodes in the BBN also need their volatility characteristics to be estimated by prior probabilities. For instance, general purpose library classes and routines are less dependent on changing requirements and so they tend to be more stable.

Obviously the volatility of different *AEs* will be highly domain dependent and hence the estimation of this volatility by providing prior probabilities for root nodes is part of the knowledge engineering task for the architect when building the AREL model.

Non-Root Nodes

Each non-root node in a BBN (i.e. nodes with incoming arcs) has an associated conditional probability table (CPT). These BBN conditional probabilities in the AREL context quantify the extent in which an architecture element influences the validity of a decision, or the extent in which a decision may change a resulting architecture element. All *AR* nodes and all non-root *AE* nodes in the BBN representation of AREL are assigned CPTs, which quantify the causal dependency on their parents. These probabilities are assigned by the architects using their past organisational experience and their assessment of the current situation.

Given the fundamental structure of $AE_i \rightarrow AR_j \rightarrow AE_k$, AE_i is either a root node with an assigned prior probability $P(AE_i)$, as described above, or a non-root node with a CPT. The probability of AR_j is conditionally dependent on the event that has occurred to AE_i , represented by $P(AR_j|AE_i)$. The probability of AE_k is conditionally dependent on the event that has occurred to AR_j , represented by $P(AE_k|AR_j)$. The *joint probability* is therefore the product of multiplying probability tables of AE_k , AR_j and AE_i . This calculation is called marginalisation and the resulting probability is the marginal probability [83].

An example of the simplest $AE1 \rightarrow AR1 \rightarrow AE2$ substructure might be when a decision ($AR1$) of using a .Net framework (i.e. output $AE2$) depends on the basis that the Microsoft Windows is the operating system platform (i.e. input $AE1$). If this decision is based *only* on the single input, this is represented by the CPT entries

- $P(AR1 = \textit{valid}|AE1 = \textit{stable})=1$
- $P(AR1 = \textit{invalid}|AE1 = \textit{volatile})=1$

- $P(AE2 = stable|AR1 = valid)=1$
- $P(AE2 = volatile|AR1 = invalid)=1$.

In this extreme case, a change in the input *AE1* to a Linux operating system would completely invalidate the *AR1* decision, which in turn renders the use of the .Net framework, represented by *AE2*, 100% volatile.

Of course, in real designs there are interactions between multiple *AEs* and *ARs*, and hence the BBN structures are more than chains. Also, the nature of the relationships may be more complex and less deterministic, which can be represented by probabilities other than just 0 and 1 in the CPTs. We will now look at how to quantify such more complex relationships, by considering in turn the three basic forms of relationships identified above in Figure 10.2.

Relationship A

If an *AE* is an input (i.e. a cause) to a decision and an *AR* is dependent upon it, any changes in the *AE* will affect the probability of the *AR*'s validity. Figure 10.2 shows an example of a type A relationship where both *AE1* and *AE2* are architecture elements that influence the decision *AR1*. The inputs in this relationship can be generalised to one or more causes (*AEs*). If any of the input *AEs* of a decision (*AR*) changes, then there is a possibility that the decision will no longer be valid. In this example, the conditional probability of *AR1* can be denoted by $P(AR1|AE1, AE2)$. As all nodes are binary in states, there are 4 possible combinations to consider for *AR1* being valid.

- $P(AR1 = valid|AE1 = stable, AE2 = stable)$.
- $P(AR1 = valid|AE1 = volatile, AE2 = stable)$.
- $P(AR1 = valid|AE1 = stable, AE2 = volatile)$.
- $P(AR1 = valid|AE1 = volatile, AE2 = volatile)$.

Note that in each case $P(AR1 = invalid)= 1-P(AR1 = valid)$. For the situation where all the *AE* causes are stable and not subject to change, the CPT entry for $P(AR1 = valid)$ reflects the confidence in the correctness of the original rationale for the architecture decision. We would expect in most cases it would be set to 1, or very close to 1. Conversely, the CPT entry for $P(AR1 = valid)$ would be lowest in the situation where all the *AE* causes are volatile.

For the intermediate situations where one or more, but not all, of the *AE* causes are subject to change, the CPT entries must be chosen carefully to reflect the relative weight of the individual causes on the architecture decision. For example, if there are two *AE* causes and both were of equal weight, say x , when making the decision, the CPT entries would be the same.

- $P(AR1 = invalid | AE1 = stable, AE2 = volatile) = x$
- $P(AR1 = invalid | AE1 = volatile, AE2 = stable) = x$

The actual value of x must then reflect the causal impact a single volatile *AE* is expected to have on the decision; x close to 1 means the decision will become invalid even with the single volatile cause, 0.5 means a fifty-fifty chance, while x close to zero means the decision is reasonably robust if only one *AE* cause is volatile. A more dominant (*AE2*) and less dominant (*AE1*) cause combination might be represented by:

- $P(AR1 = invalid | AE1 = stable, AE2 = volatile) = 0.8$
- $P(AR1 = invalid | AE1 = volatile, AE2 = stable) = 0.3$

In this example, if *AE1* is stable and *AE2* is volatile, the probability of *AR1* being invalid is higher than if *AE2* is stable and *AE1* is volatile. It indicates that *AE2* is more influential to *AR1* than *AE1* because when *AE2* becomes volatile, there is a higher chance that *AR1* will become invalid.

Relationship B

When an architecture decision is made, an *AR* is created to record the rationale and linked with one or more resulting *AEs*. During architecture design, if a requirement changes then it is possible that the *AR* depending on it will be invalidated and so all resulting designs from that *AR* are more likely to be volatile and subject to change. Note that there is no direct relationship between those *AEs* that are caused by the same *AR*, so we need only consider the effect of the *AR* on each *AE* individually. In Figure 10.2, for example, even though *AE4* and *AE5* are both dependent on *AR1*, they are independent of each other given *AR1*. So the CPT $P(AE_i | AR)$ for each AE_i is specified as follows:

- $P(AE_i = volatile | AR = valid)$
- $P(AE_i = volatile | AR = invalid)$

Note that in each case $P(AE_i = \textit{stable}) = 1 - P(AE_i = \textit{volatile})$. In most cases, while the *AR* is valid, we would not expect the *AE* to change, hence $P(AE_i = \textit{volatile} | AR = \textit{valid})$ would be close to 0. Conversely, if the *AR* is invalid, the *AE* is much more likely to change. The more an *AE* depends on an *AR*, the more likely it is subject to change when *AR* becomes invalid, and hence $P(AE_i = \textit{volatile} | AR = \textit{invalid})$ becomes closer to 1. If an *AE* depends only very loosely on an *AR*, then $P(AE_i = \textit{volatile} | AR = \textit{invalid})$ might be set much lower than 1. In Figure 10.2, $AR1 \rightarrow AE4$ and $AR2 \rightarrow AE6$ are examples of relationship B.

Relationship C

It is possible that an *AE* is the result of multiple separate decisions (*ARs*). For example, in Figure 10.2, *AE5* is the result of two separate decisions *AR1* and *AR2*. The conditional probability of *AE5* can then be defined as $P(AE5 | AR1, AR2)$. Since *AE5* is dependent upon both decisions *AR1* and *AR2* simultaneously, estimates for the CPT can be expressed as:

- $P(AE5 = \textit{volatile} | AR1 = \textit{valid}, AR2 = \textit{valid})$
- $P(AE5 = \textit{volatile} | AR1 = \textit{valid}, AR2 = \textit{invalid})$
- $P(AE5 = \textit{volatile} | AR1 = \textit{invalid}, AR2 = \textit{valid})$
- $P(AE5 = \textit{volatile} | AR1 = \textit{invalid}, AR2 = \textit{invalid})$

The causal relationship being modelled in this CPT is essentially an additive one: the more decisions that are invalid, the higher the probability that the common resultant *AE* will be volatile. While all of the decisions are still valid, the CPT entry for $P(AE5 = \textit{volatile})$ will be close to zero, and if all the decisions are invalid, it will be close to 1. For the intermediate situations where one or more, but not all, of the *AR* decisions are invalid, the CPT entries must be chosen carefully to reflect the relative weight of the individual decisions on the resultant *AE*. Assuming *AR1* has more influence on *AE5* than *AR2* in the example, the cause combination could be represented as:

- $P(AE5 = \textit{volatile} | AR1 = \textit{invalid}, AR2 = \textit{valid}) = 0.8$
- $P(AE5 = \textit{volatile} | AR1 = \textit{valid}, AR2 = \textit{invalid}) = 0.3$

These CPT entries mean that the combination where *AR1* is invalid and *AR2* is valid is quite likely to cause *AE5* to be volatile (probability 0.8). Conversely, when *AR1* is valid and *AR2* is invalid, the probability that *AE5* is volatile is lower (0.3), signifying less influence by *AR2*. In other words, *AR1* has a larger impact on *AE5*'s volatility than *AR2*.

10.3 Reasoning about change impact with AREL

10.3.1 An example

In this section, we build on the example of designing the message processing module to illustrate how to use quantifiable change impact analysis with BBN. This example is explained in the case study in Section 7.1.3. As discussed previously, one of the key issues in payment messaging design is to optimise its performance, and a key factor is whether payment messages are processed synchronously or asynchronously. In this case, asynchronous messaging was selected and a chain of architecture design decisions and design objects were formed (see Figure 10.3). When considering the payment messaging design, security issues such as authenticity and privacy must be considered as well. It is because security features could influence the software architecture, especially in the areas of exception handling and acknowledgement processing. In the following, we will consider the payment messaging design together with its security features.

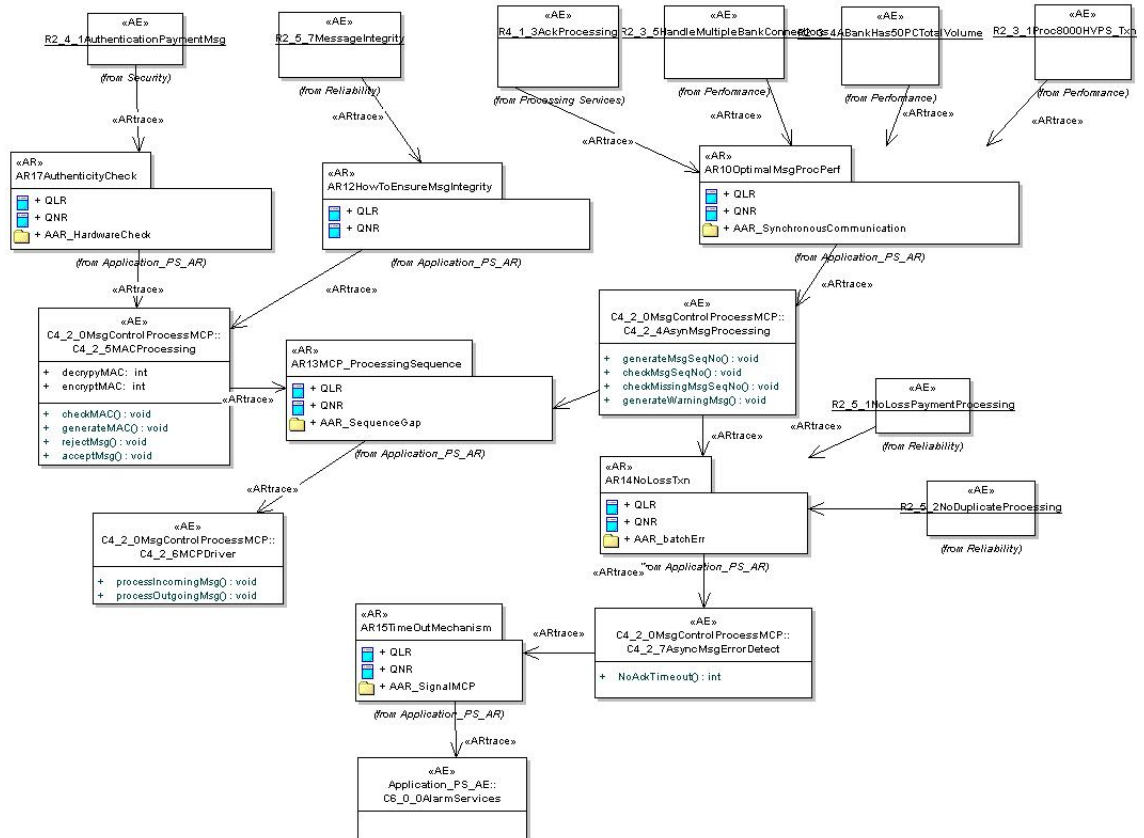


Figure 10.3: Payment Messaging Design: Asynchronous Message Processing

Payment messages are transmitted over different networks and some parts of those

network may be vulnerable to attack. It is possible to tamper with the payment messages by intruding the networks or the computers. So security protection of payment messages is a key issue in designing payment system. The following are three additional security requirements that need to be considered in message processing:

- *R2_4_3* Privacy of Payment Message - payment messages are encrypted to protect the privacy and confidentiality of the transaction.
- *R2_4_1* Payment Message Authenticity - when a payment message is authenticated, it means that the system recognises that this payment message can only be sent by the authorised person or organisation. The non-repudiation characteristic of payments thus forms the legal basis of a payment system.
- *R2_5_7* Payment Message Integrity - this mechanism ensures that the payment message has not been modified in any way during its transmission. It is used to protect the payment message from hackers and transmission errors.

In Figures 10.4, we show the security design decisions using the AREL representation. Decision *AR29* is about the implementation for the privacy requirement, it was decided that a software algorithm would be used to implement this feature. Similarly, *AR17* and *AR12* are decisions to implement design objects to cater for both the authenticity requirement and the message integrity requirement using a software module *C4_2_5*. This module basically calculates a Message Authentication Code (MAC) which is based on a special key shared between the sender and the receiver only. If the code checks out, then it proves that the message is originated from a specific sender and no one else. It also proves that the message has not been tampered with during its transmission.

Now that a software algorithm to guarantee payment messages' privacy and authenticity is used, one must be certain that the security keys used in the calculation cannot be compromised. Thus, decision *AR30* is made to manage the security keys for such purposes. At the time of the decision, PKI was not commercially available in the chosen platform and so this alternative was not implementable.

As part of the validation, we need to decide on how to manage the processing sequence (i.e. *AR13*). The sequence number ensures that the payment message to guarantee that there are no gaps in a series of messages. If the check fails, then the payment message is rejected, this is achieved by *C4_2_9*.

Figures 10.5(a) and 10.5(b) show the BBN representation of an AREL model for the system with probability tables showing the prior probabilities and CPTs of the payment

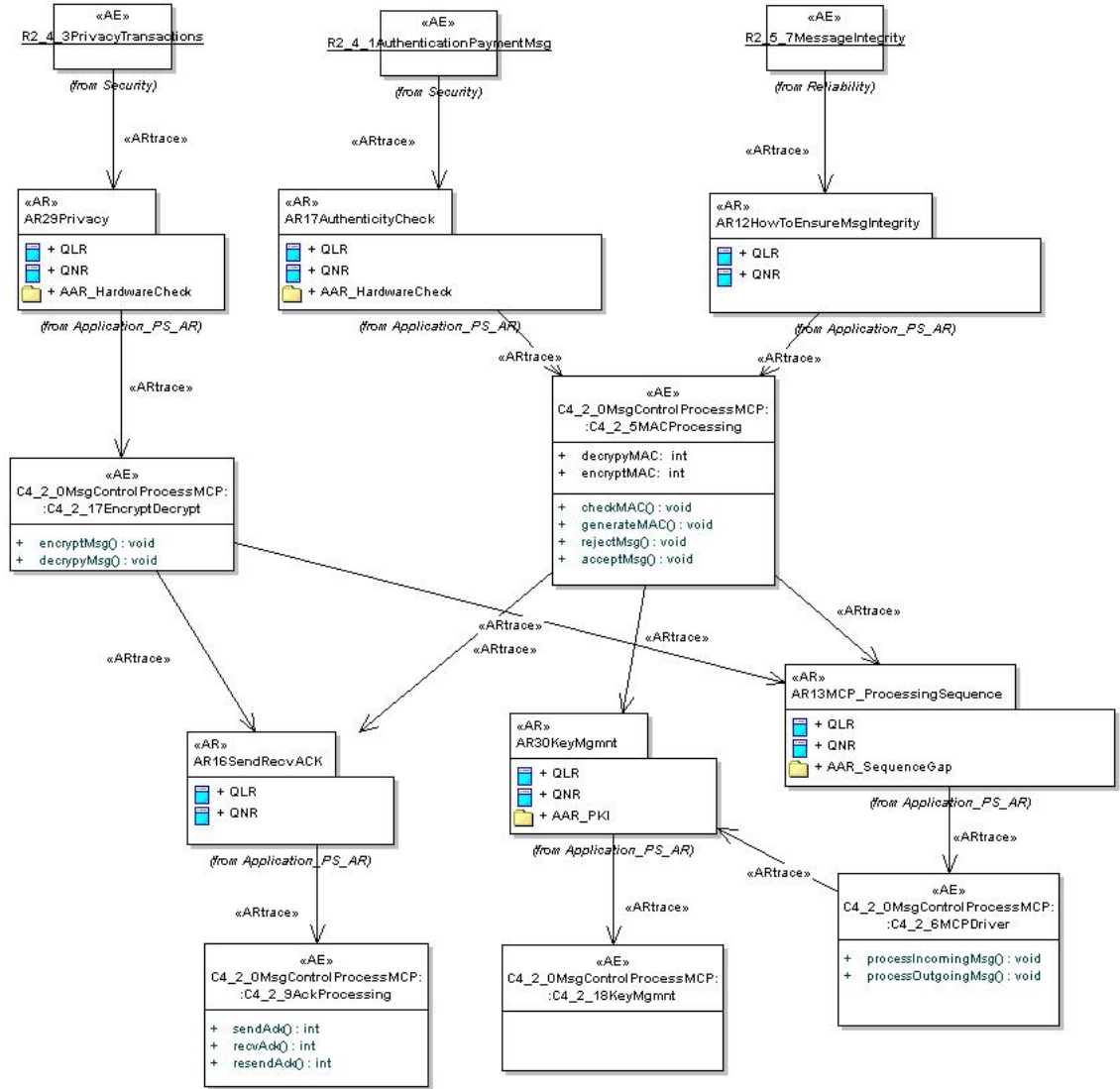


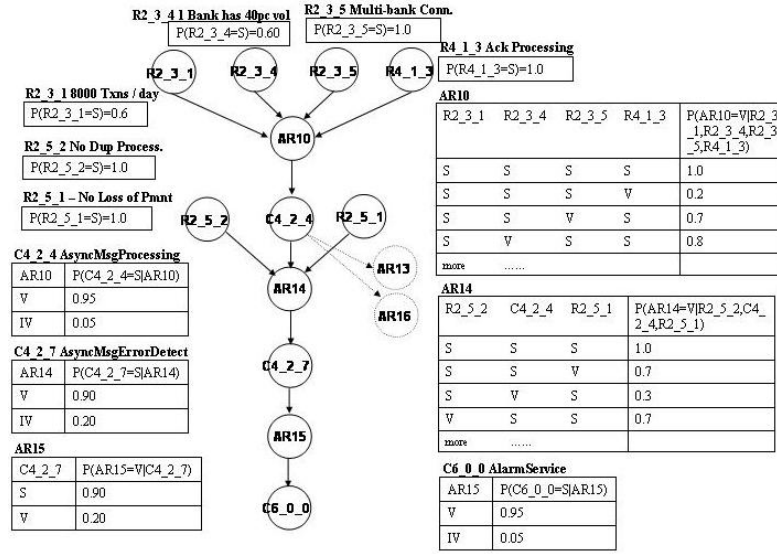
Figure 10.4: Payment Messaging Design: Security

message design. In Figure 10.5(a), requirement *R2.3.5* specifies that the system has to handle multiple bank connections and requirement *R4.1.3* specifies that payment messages must be properly acknowledged. These requirements cannot be compromised or changed and so their prior probabilities are set to 1 (or 100%)³. Similarly, requirements *2.5.1* and *R2.5.2* specify no loss and no duplicate processing is allowed, their prior probabilities are set to 1. Requirement *R2.3.1* specifies that the daily transaction is 8000 high-value payment messages per day, this can change over time and its prior probability is set to 0.6.

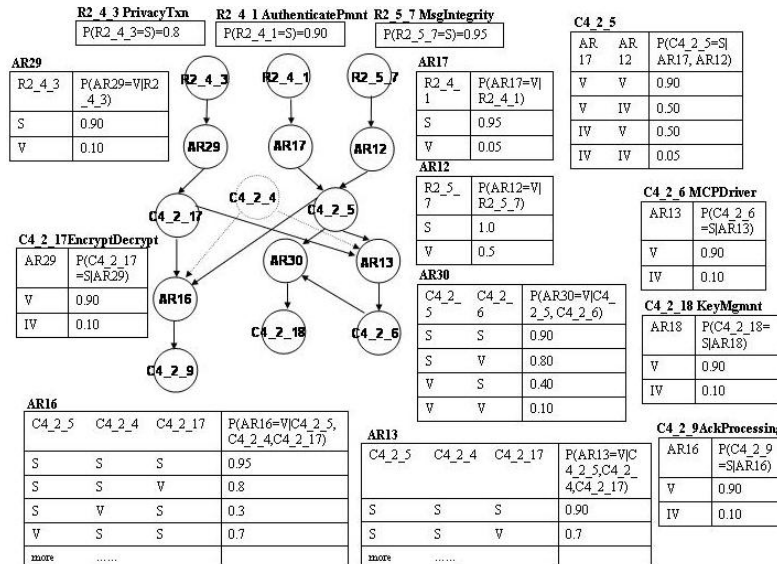
³The probability can be represented by a number between 0 to 1 or its equivalent in percentage point between 0% to 100%

10.3. Reasoning about change impact with AREL

Decision $AR10$ depends on four requirements (i.e. $R2.3.1$, $R2.3.4$, $R2.3.5$ and $R4.1.3$). From the conditional probability table (CPT), $R4.1.3$ is more influential to $AR10$ than the other three requirements because when it becomes *volatile*, the probability that $AR10$ is *valid* drops to 0.2. (i.e. $P(AR10 = V | R2.3.1 = S, R2.3.4 = S, R2.3.5 = S, R4.1.3 = V) = 0.2$) The influence is due mainly to whether acknowledgement message should be processed asynchronously or not.



(a)



(b)

Figure 10.5: A BBN Representation with prior probabilities/CPTs of a Payment Message Design: (a) Asynchronous Message Processing (b) Security

$AR14$ depends on requirements $R2.5.1$, $R2.5.2$ and design $C4.2.4$. However, $C4.2.4$

is most influential because if we change the fundamental design of asynchronous processing, say to synchronous, then the assumption of how to detect loss of acknowledgement messages would change and the subsequent design would also be different. This is reflected in the CPT of *AR14* where the probability of *C4-2-4* being *volatile* whilst the other requirements being *stable* is 0.3. *C4-2-4* also affects decisions *AR13* and *AR16*, their CPTs are shown in Figure 10.5(b).

Figure 10.5(b) shows the prior probabilities and the CPTs of the security design for payment message processing. There are a number of key decisions, *AR13* is a decision on the processing steps of MCP. It is influenced by *C4-2-17* because it must be decrypted before the message can be processed. It is influenced by *C4-2-5* because the message must then be validated. If validation fails, the message is sent back to the originator with a negative acknowledgement immediately. The decision is influenced by *C4-2-4* because this decision is based on the assumption that the treatment of acknowledgement is asynchronous, therefore MCP must be stateful, i.e. MCP has to remember the status of a payment message so that when the acknowledgement is returned at some point in the future, it could be matched to the original message.

AR16 is a decision on processing acknowledgement. It depends on a number of factors including *C4-2-4*, *C4-2-5* and *C4-2-17*. The decision relies heavily on *C4-2-4* in that if the acknowledgement processing model changes, it is highly likely that the decision will become *invalid*.

10.3.2 Original beliefs modelled by AREL

The AREL model in Figure 10.6 is a representation combining the illustrations in Figure 10.5(a) and 10.5(b) without the details of the prior probabilities and CPTs. It shows the marginal probability distribution when no evidence (i.e. change) has been added. The specific visualisation format is provided by Netica [107] BBN software. The visualisation includes the name of each node across the top, the state names are on the left hand side of the nodes (*valid/invalid* for *AR* nodes and *stable/volatile* for *AE* nodes). Marginal probabilities are shown as percentages (e.g. 85.6% and 14.4% for *AR10*) by the two horizontal bars. The marginal probabilities, for instance, of a leaf-node such as *C4-2-9*, are calculated based on its own CPT and the CPTs and prior probabilities distributions of its ancestors. For instance, the marginal probability of *C4-2-9* indicates that based on the current requirements and design, there is a probability of 0.70 (shown as 70% in Figure 10.6 by Netica) that the design is stable.

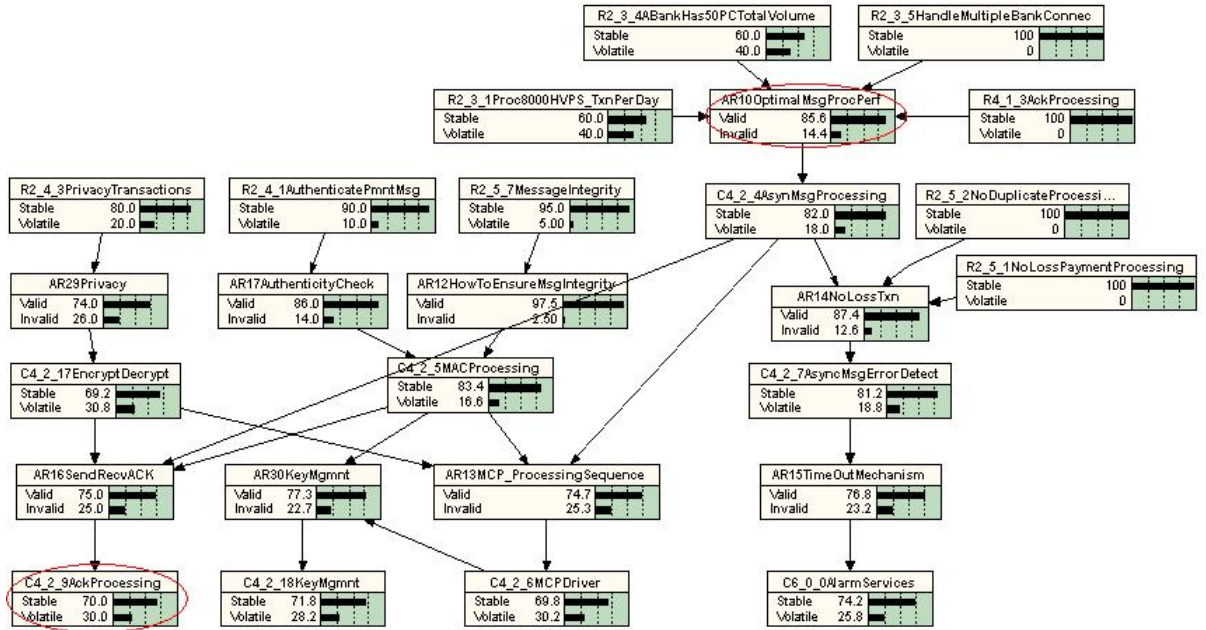


Figure 10.6: An Example BBN Shown with Beliefs Before Any Evidence is Entered

10.3.3 Predictive reasoning

As the requirement or the environment in the system changes, we want to predict their effect in terms of possible architecture design changes. Being able to identify the areas that are likely to change would improve both the accuracy and efficiency of architecture enhancements. Recall from Section 10.2 that reasoning in a BBN means users first set the values of certain nodes in the network to indicate evidence that has been gathered about the system. For the AREL BBN model, setting values means that change takes place in the system. These changes are then propagated through the network, producing a new probability distribution over the remaining variables in the network [83], showing the what-if scenarios of the impact of change. Multiple changes could be introduced simultaneously. Let us consider how we can do this type of predictive reasoning in the BBN representation of AREL.

To add one or more likely changes as evidence to the AREL model, an architect would instantiate the BBN with evidence that an *AE* is “volatile”, i.e. *volatile* state is set to 100%. The BBN belief updating algorithms would then compute the posterior probabilities for *AR* and *AE* nodes which are affected by the additional evidence in the network. The architect can assess the change impacts based on the posterior probabilities. In particular, which decisions (*AR* nodes) are more likely to be invalid and which architecture elements (*AE* nodes) are more likely to be volatile.

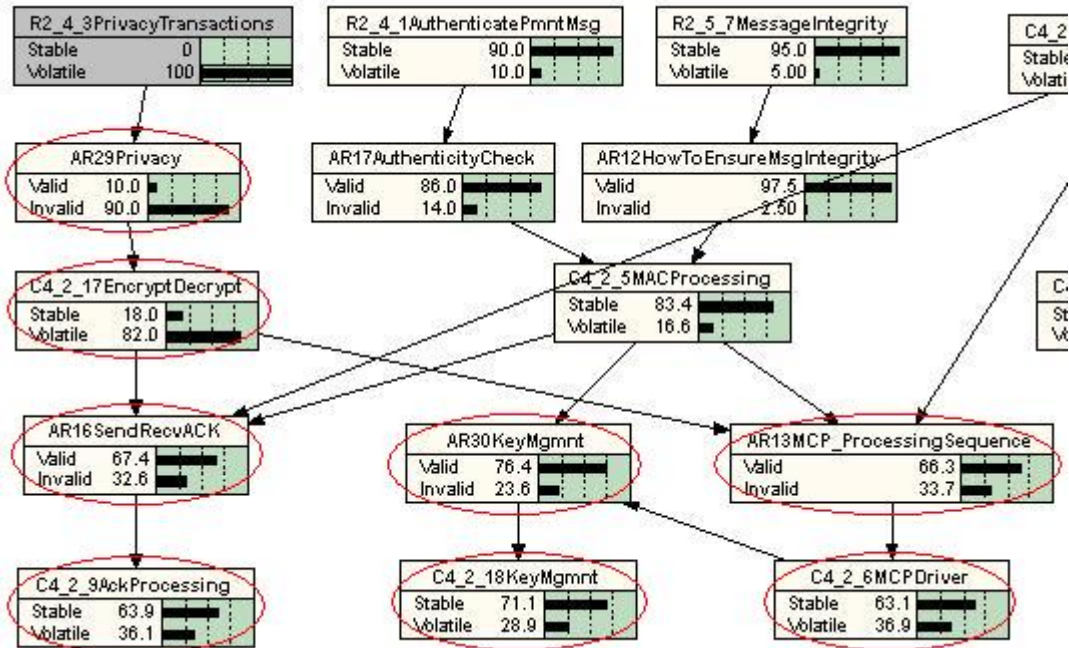


Figure 10.7: Predictive Model

Let us assume that the bank wants to enforce a stronger privacy policy. It means that the requirement $R2_4_3$ would change and become volatile. This change is inserted as evidence to set the volatile state of $R2_4_3$ to 100% (see highlighted node in Figure 10.7). Given this evidence, we can do a what-if analysis and predict that there is a 90% chance that the decision $AR29$ will be invalid leading to a 82% chance that design decision $C4_2_17$ will be volatile. This prediction is based on the strength of the relationship, represented by CPT, between architecture design reasoning and its motivational reason, as well as the strength of the relationship between the design outcomes and the design reasoning. We predict that if $R2_4_3$ is to change, there is a 82% chance that $C4_2_17$ would be affected and subject to change.

This volatility also ripples through $AR16$, $C4_2_9$, $AR13$, $C4_2_6$, $AR30$ and $C4_2_18$. This is because all these nodes are indirectly and conditionally dependent on $R2_4_3$. For the ARs , the probabilities of their validity have decreased. For the AEs , the probabilities of their volatility have increased. Note that apart from these nodes, the rest of the BBN network remain unchanged. This is because those nodes are conditionally *independent* of $R2_4_3$ and its descendants. That is, a change in $R2_4_3$ has no effects on them. This phenomenon is called *direction-dependent separation* or simply *d-separation* (see [73, 83] for details).

BBN enables architects to predict the likelihood of change impact in the related parts

of the architecture design when a modification(s) is introduced to the system architecture. The quantification provides a means by which architects can assess how likely the system are to be affected.

10.3.4 Diagnostic reasoning

Another use of BBN reasoning in AREL is to diagnose possible causes and influences in architecture design. As shown in the preceding section on predictive reasoning, changes in the high-level requirements or design objects might affect the decisions and architecture elements which depend on them. Conversely, given an architecture element, an architect might want to discover what influences this architecture element, and the reasoning is based on the requirements and decisions that it depends on and the strength of that dependency. When a dependent design object is subject to change, the reasons of the change might be due to changes in its ancestor objects such as requirements.⁴ The BBN can be used to diagnose these possible causes or influences by identifying changes in the posterior probabilities of the ancestor nodes. Note that such diagnostic reasoning propagates “backwards” against the direction of the edges, which represent the causal relationships.

In AREL modelling, such diagnostic reasoning might take place if evidence is inserted into a non-root *AE*. For instance, the current alarm service requires that for every payment message that is entered into the system, an alarm needs to be set using the UNIX signalling method (i.e. `signal(2)`) and the alarm is recorded in the database for recovery purpose. If an acknowledgement does not arrive in time, the alarm will sound to trigger the follow-up actions. This mechanism is processing intensive and complex.

Let us analyse the scenario of changing the Alarm Service design *C6_0_0*. The reason for this analysis is so that we can discover which *AEs* the design object is depended on. First, we set *C6_0_0*'s volatile state to 100% (see Figure 10.8), we observe that the posterior probabilities of decisions *AR15*, *AR14*, *AR10*, *AR13*, *AR30* and *AR16* will change to become less valid. Also, the posterior probabilities of the root nodes that *C6_0_0* depends on have had their volatility increased. As such, we diagnose that the requirements *R2_3_1* and *R2_3_4* influence the design. Requirements *R2_3_5*, *R4_1_3*, *R2_5_1* and *R2_5_2* are also root nodes of *C6_0_0*, indirectly. However, they have their *stable* state set to 100% because these requirements cannot be altered, thus their volatility have shown no change.

The posterior probabilities of design objects *C4_2_4*, *C4_2_7*, *C4_2_6*, *C4_2_18* and

⁴Changes might also be caused by new external elements, i.e. external causal intervention. As such, new root-nodes can be used to model the change.

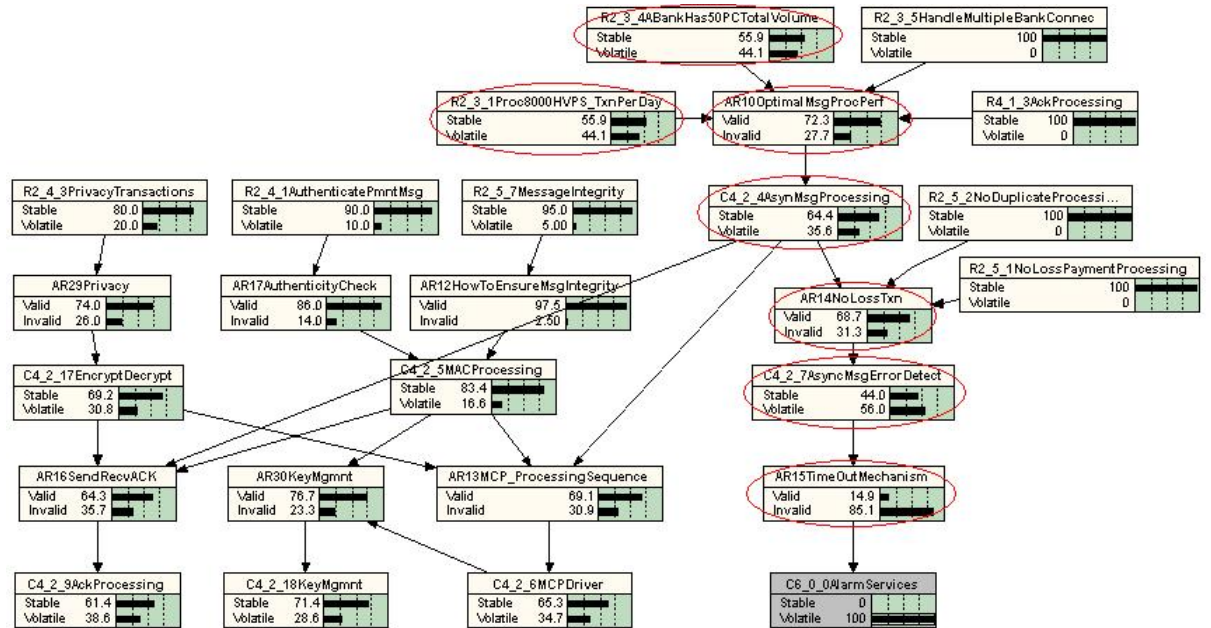


Figure 10.8: A Diagnostic Model

$C4_2_9$ have had their volatility increased. In particular, $C6_0_0$ depends indirectly on $C4_2_7$ and $C4_2_4$. The volatility of $C4_2_7$ has increased from 18.8% to 56%. $C6_0_0$ is affected by $C4_2_7$ because the purpose of the alarm service is to notify the error processing unit that an error has occurred. Tracing backwards, this design is in turn caused by $C4_2_4$ where asynchronous acknowledgement mechanism has been chosen. The volatility of $C4_2_4$ has increased from 18.0% to 35.6%. The increase in the volatility indicates the level of dependency between the AE node in question and the nodes in which that AE node depends on.

Out of the eight AE nodes and six AR nodes that are affected by a change in $C6_0_0$, only four AE nodes and three AR nodes are ancestor nodes of $C6_0_0$. They are circled in Figure 10.8. The rest of the affected nodes are conditionally dependent on $C4_2_4$ but $C6_0_0$ does not depend on them. When evidence is inserted into $C6_0_0$, the reasoning process diagnoses the dependency on its ancestor nodes. This chain of reasoning is performed in a backward direction. Notice that a change of posterior probabilities in $C4_2_4$ has triggered an update of its dependent nodes (i.e. $AR13$, $AR16$ and their descendant nodes). This mechanism highlight possible changes in these nodes if $C4_2_4$ is to change. As such, the ripple effect of architecture changes can be high-lighted through their dependent relationships. This is an enhancement to the manual traceability methods described in the last chapter.

Such diagnostic reasoning is useful because, before any changes are introduced to the

system, architects can investigate the possible causes (i.e. both architecture elements and decisions) which contribute to the being of the design object. As such, considerations of these causes could be made to ensure the design consistency.

10.3.5 Combining diagnostic and predictive reasoning

When an architecture is subject to impact analysis in a real-life project, it is often necessary to understand the relationships between the design object which is undergoing modification and other design objects which might be affected by such modification. The side-effects of a modification may come from a few areas: (a) design objects which are directly affected by the modification; (b) requirements, constraints or assumptions which are in conflict with the modification; (c) other design objects that may be indirectly affected when the requirements, constraints or assumptions are compromised.

Using BBN's diagnostic and predictive reasoning, architects may use posterior probabilities to guide the traversal of the AREL network to quantify likely changes within the network. Based on the message processing example shown in Figure 10.6, we investigate the changes that are likely to happen to the design of the system if we are to modify *C4_2_9*.

C4_2_9 is responsible for sending acknowledgement messages to a bank that originates a payment message. It is also responsible for receiving acknowledgement from a bank after sending it payment messages. In either case, only when an acknowledgement is received or sent would a payment message be considered legally binding. With the current design of the system, the system has to reconcile what acknowledgements have not been sent or received using a signalling mechanism. The design seems complicated and resource intensive. We are to investigate what-if this mechanism is to change and which parts of the system would be impacted.

Using the BBN network, we set the evidence and quantitatively analyse the likelihood of change. Table 10.1 illustrates a sequence of reasoning steps to find out how setting the evidences for *C4_2_9* and other architecture elements could affect the validity of architecture rationales and the volatility of architecture elements. This combined reasoning process involves using both the BBN diagnostic reasoning and predictive reasoning. The leftmost column in Table 10.1 shows the name of the BBN node (grouped by the *AE* nodes in the upper set, and the *AR* nodes in the lower set). Each of the remaining columns shows the change that is being investigated at each step, corresponding to the evidence being entered into the BBN, by setting the appropriate *AE* volatility to 100%.

The following is a description of the reasoning steps that are illustrated in 10.1:

Table 10.1: Volatility of Architecture Elements and Validity of Architecture Rationales over a Sequence of Changes

BBN Node	Step 1 Modify C4.2.9	Step 2 Modify C4.2.4	Step 3 Modify C4.2.7	Step 4 Modify C6.0.0
AE nodes – Volatile State				
C4.2.9 Ack Processing	100%			
C4.2.4 Asyn Msg Processing	41.9%	100%		
R2.3.1 8000 HVPS	45.6%	59.2%		
R2.3.4 1 Bank =50% vol	45.6%	59.2%		
C4.2.5 Mac Processing	25.4%	18.4%		
R4.2.6 MCP Driver	38.9%	51.5%		
R4.2.18 Key Management	33.1%	31.2%		
C4.2.7 Error Detect	30.5%	59%	100%	
R6.0.0 Alarm Service	33.2%	51.2%	77%	100%
AR nodes – Invalid State				
AR16	75%	96.4%		
AR30	28.9%	26.5%		
AR13	36.1%	51.8%		
AR10	32.4%	76.2%		
AR14	29.3%	70%	94.9%	
AR15	31.4%	51.3%	80%	98.7%

- Step1 – evidence is inserted in *C4.2.9* (i.e. its volatility is set to 100%) to reflect that the module has to change. The change triggers probabilities to be updated in the network as shown in the column (step 1) of table 10.1. At this juncture, we try to diagnose the requirements and design which are closely related to the acknowledgement processing. Traversing backwards from *C4.2.9*, we can see that two requirements *R2.3.1* and *R2.3.4* are shown to be the possible causes of this change(i.e. inserting evidence in *C4.2.9* increases *R2.3.1* and *R2.3.4*'s volatility from 40.0% to 45.6%). The increased volatility helps us to reason that the performance requirements play a part in the design. The change in the posterior probability of *C4.2.4* from 18.0% to 41.9% indicates that it is sensitive to a change in *C4.2.9*, so this architecture element is subject to a major impact in the design change.
- Step 2 – We insert the evidence in *C4.2.4* (i.e. its volatility is set to 100%) to contemplate that it is being changed, and we examine what change impacts it may have on the rest of the system. At this point, we reason that the complexity of the acknowledgement process is due to the choice of asynchronous processing. This design requires to store the status of each payment message so that the acknowledgements can match up with the payment messages. This is a cumbersome and inefficient mechanism. Assuming we change it to a synchronous processing mechanism and set the volatility of *C4.2.4* to 100%, there is an impact on the node *C4.2.7* that depend on it. Its volatility increase from 30.5% to 59%. Another part of the system affected by this change is *C4.2.6* and its volatility has increased from 38.9% to 51.5%, indicating that the processing sequence might have to change.

- Step 3 – evidence is inserted in $C4_2_7$ (i.e. its volatility is set to 100%) to reflect that the design object has to change. This is because when the asynchronous messaging design changes, its error detection mechanism would have to change as well. The MCP process will handle error processing immediately. In this way, the error detection in $C4_2_7$ must change.⁵ The change in $C4_2_7$ has an impact on $C6_0_0$ further down the network in that alarms would no longer be needed to be set for each payment message. The volatility of $C6_0_0$ thus rises from 51.2% to 77%.
- Step 4 – evidence is inserted in $C6_0_0$ to indicate changes in the alarm setting processing. Since there are no other architecture design that depends on it, there is no further change impact analysis that is required. As a result of this change, the posterior probability of the invalidity of $AR15$ has increased from 80% to 98.7%.

Figure 10.9 shows the posterior probabilities of the full network after all predictive and diagnostic reasoning have been performed, i.e. combining all the evidence for nodes $C4_2_9$, $C4_2_4$, $C4_2_7$ and $C6_0_0$. Note that the posterior probabilities would be the same regardless of the order in which the evidence was added, or if evidence for all five nodes were added together. The change in the probabilities over the *sequence* of reasoning steps is clearly crucial to supporting the human reasoning by the architect who is performing the what-if analysis.

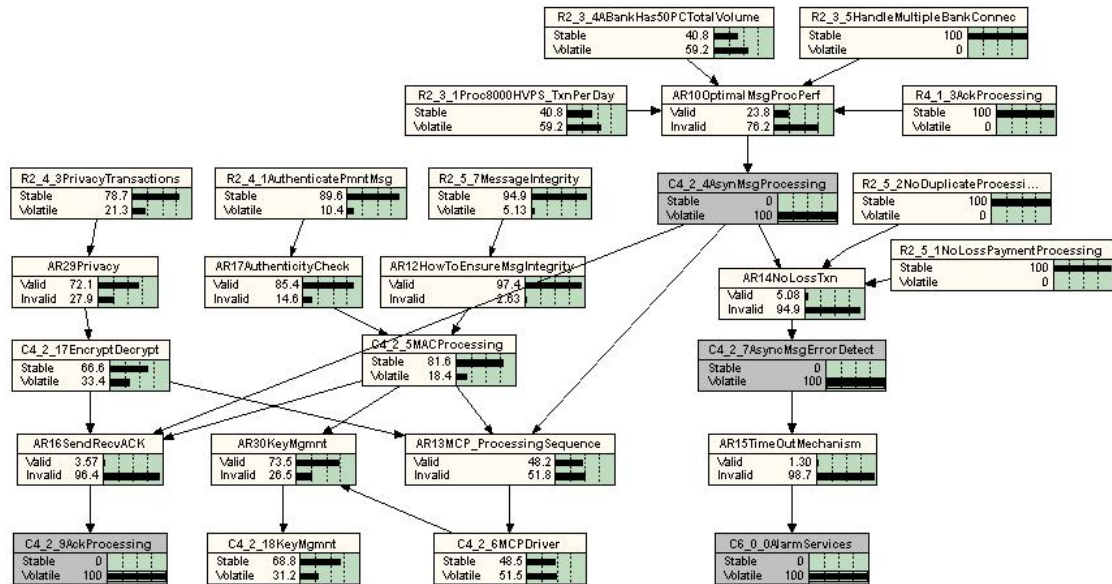


Figure 10.9: A BBN Model of Combined Reasoning

⁵ $C4_2_8$ error recovery and all its associated architecture decisions and design are part of the original architecture design. A change in $C4_2_9$ will affect this part of the system. The change impacts are not shown here to simplify the illustration.

This example demonstrates that in impact analysis, it is necessary to use both diagnostic and predictive reasoning in combination to help determine the change impacts. One modification to the architecture often triggers a ripple effect where multiple modifications are required.

10.4 Discussions and limitations

The BBN representation of AREL requires architects to supply the probabilities of the AREL nodes. This relies on the architects' experience and intuition. These estimates are semi-objective in nature where different architects may provide different estimates given the same design scenario. However, in the longer-term architects should be able to supply consistent and accurate estimates because feedbacks of any inaccuracies would help architects to refine their estimates. Having to provide an accurate probability value to a fine degree of granularity, say one percentage point, would be difficult and probably meaningless. One possible way to overcome this granularity issue is to specify the probabilities with a qualitative verbal scale. For instance, the volatility of an *AE* node could be specified as very high, probable, fifty-fifty, improbable and impossible. These ordinal categories can then be mapped into probabilities, and tests can be undertaken to determine the calibration of that mapping to obtain the desired outcomes of such modelling. We argue that although the approximation may not be as accurate, it provides some useful information to assist architects in their analysis. A similar risk assessment categorisation has been suggested by [19]. A Netica-compatible software tool [65] for such qualitative elicitation of probabilities and verbal maps already exists. Interested readers are referred to [83] for a comprehensive discussion of the subject.

BBN provides an estimation for architects to quantify change impacts, which is an evidence to investigate into certain parts of the architecture design that might be subject to change. The usefulness of this method will be enhanced if the costs and the risks of the possible changes are quantified. This is the goal of our future research.

A survey has indicated that architects carry out impact analysis by tracing requirements, analysing the design rationale and considering constraints and assumptions of the design (see Section 5.3.7). Such activities can be supported by the traceability approach described in Chapter 9. However, there are two challenges if this approach is used alone. Firstly, the strength of the relationships between architecture decisions and elements cannot be quantified. BBN overcomes this issue by using quantitative reasoning to estimate the likelihood of the change impacts to an architecture design. Secondly, a change in the architecture design may create a ripple effect that cannot be traced easily. A change in a

design object may cause a change in a requirement, and the impact on the requirement may have an impact on other disparate design elements. In this instance, forward and backward tracings may not be sufficient to uncover all change impacts. On the other hand, BBN can compute the likelihood of change that ripple through the network. Therefore, these two approaches are complementary to each other in supporting design reasoning and system maintenance activities.

It is obvious that not all system designs need to apply BBN to AREL, smaller systems that are easy to trace and understand will gain very little from it. We suggest that large, complex and long life-span systems that contain intricate design decisions to cater for extensive non-functional requirements are candidates to use AREL with BBN.

The assignment of probabilities to AREL nodes individually can be laborious, it might be a cost-saving measure to automatically assign CPTs. To make this possible, architecture design patterns with pre-assigned probabilities are required. Architecture design patterns can represent common ways in which decisions are made to deal with certain aspects of a design. An example would be a security design in a system. Such architecture design patterns would capture the essence of a design and the probabilities of change impacts. When architecture design is carried out using design patterns, its repeated use would help improve the accuracy of the probability estimation for change impact analysis.

10.5 Summary

In this chapter, we enhance the AREL model to provide a quantifiable reasoning structure using the Bayesian Belief Networks. It enables architects to quantify change impacts to architecture decisions and elements using probabilities. We have identified three different reasoning methods in which change impacts can be analysed: (a) predictive reasoning; (b) diagnostic reasoning; and (c) combined reasoning. With these methods, architects can carry out quantitative analysis to predict the probability of change in an architecture design. These methods are complementary to the traceability and qualitative analysis methods presented in earlier chapters.

Chapter 11

Tool implementation

In the previous chapters, we have described the AREL representation and its applications to support architecture design and maintenance. This chapter describes the tools used supporting these activities. The AREL tool-set captures *AE*, *AR*, *AAR* and their relationships described in Chapter 6. It can be used to support the ARM process since it captures the *QLR* and *QNR* (see Chapter 8). It supports the traceability methods (see Chapter 9) and change impact analysis using BBN (see Chapter 10).

In an empirical study described in Section 7.2.4, expert designers have indicated that the success of AREL would depend on its tool implementation. First of all, they think that a graphical tool is important to represent such relationships. They suggest that without a graphical tool, it would be ineffective to use AREL because of the complex relationships that might exist in a large system architecture. They further suggested that the tool must be able to retrieve the information by allowing architects to specify the selection criteria for information retrieval.

In addressing these needs, we have established four objectives when building the tool-set to support AREL implementation: (a) the design tool should use commercially supported software so that it can integrate with architecture design processes in the software industry; (b) the graphical notation must be an industry standard and as such we have chosen to use UML; (c) a tight integration of the tool-set for easy learning and application; (d) the tools must support the key features of the AREL applications.

The tool-set to support the AREL applications comprises of three components, Enterprise Architect [150], Netica [107], and our custom-built AREL Tool. Together they form the AREL tool-set. Their relationship is shown in Figure 11.1. The three components of the AREL tool-set exchange information via the Enterprise Architect repositories and

Netica repositories.

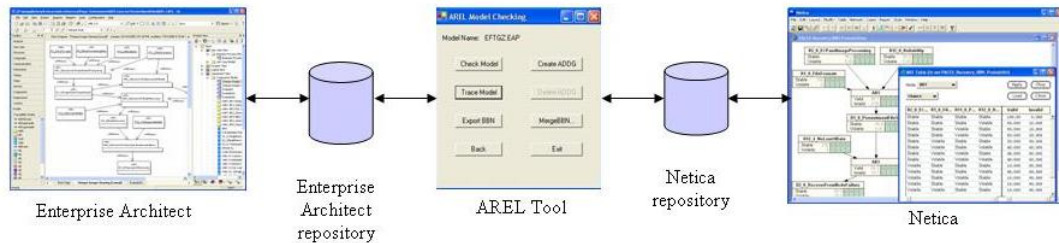


Figure 11.1: The AREL Tool-set

The key features of the tool components are as follows:

- Enterprise Architect - a UML tool that supports the design of a system. We have enhanced it to support architecture rationale capture and to relate architecture rationale with architecture design.
- AREL Tool - this is our custom-developed program to support consistency checking of the AREL models and to support AREL model tracing. It also converts UML models into Netica models.
- Netica - a BBN tool to support the capture of prior probabilities and CPTs. It computes the posterior probabilities when evidence is inserted into the BBN.

A detailed description of the tool implementation to support AREL and its applications are provided in the following sections.

11.1 Capturing architecture design rationale

Previous research in the area of design rationale has indicated that there are two usability criteria for design rationale systems. Capturing design rationale must have minimal interference with the design process [132] and the users of design rationale must be able to retrieve the reasons to answer their “*why such-a design*” questions [62]. Therefore, the tool set should be able to support both the architecture design process and the design reasoning process. We address these issues in AREL by integrating commercially available tools to our custom-developed programs to provide the functionality.

The tool to capture AREL models is a UML tool, Enterprise Architect (version 5.00.767). Since AREL is meant to be a part of the architecture development process

where UML modelling is done, therefore using the same notation to represent architecture rationale provides consistency for the architecture design representation.

Since standard UML does not support the AREL extension, we have to enhance it with *stereotypes* (see Section 6.6). In Enterprise Architect, a facility using *Stereotype Package* is available to enable us to custom-define new stereotypes. The new stereotype is an extension of the standard UML constructs with added definitions. In AREL, the additional information that we need to capture are contained in architecture elements, architecture rationale and their trace relationships. For instance, we create a stereotype `<<AE>>` for a standard UML *class*. Additional information can be captured in an *AE* with tagged values (see Figure 11.2). These values are encapsulated in an architecture element.

The screenshot shows a 'Tagged Values' window for a class element named 'C4_2_1MsgCompDecomp (Class)'. The window contains a table of key-value pairs representing the tagged values for this element.

C4_2_1MsgCompDecomp (Class)	
Author	AT
CurrFlag	true
DocumentLocation	EFT Architecture Design Specification - Part 1
ElementID	C4_2_1MsgCompDecomp
ElementSubtype	
ElementType	c
ElementVersion	1
LastUpdateDateTime	10/11/2005

Figure 11.2: An Example of AE Tag Values

Similarly, the architecture rationale stereotype `<<AR>>` encapsulates additional information. This is shown in Figure 11.3.

The screenshot shows a 'Tagged Values' window for a package element named 'AR3PaymentMsgGeneralFormats (Package)'. The window contains a table of key-value pairs representing the tagged values for this element.

AR3PaymentMsgGeneralFormats (Package)	
ARid	AR3PaymentMsgGeneralFormats
Author	AT
CurrFlag	true
LastUpdateDateTime	12/12/2005

Figure 11.3: An Example of AR Tag Values

In order to make use of these stereotypes extension, the AREL stereotype package has to be imported into Enterprise Architect. This package can be downloaded from [153]. The installation instructions are contained within the downloaded package. Once the package has been successfully installed, the user will see the AREL model elements that can be used for capturing design rationale. This is shown in Figure 11.4. On the left hand side is a list of AREL model elements that can be used for creating an AREL model.

Users can drag an AREL model element such as an *AEclass* icon from the list and then drop them on the design canvas to the right hand side to create an instance of it.

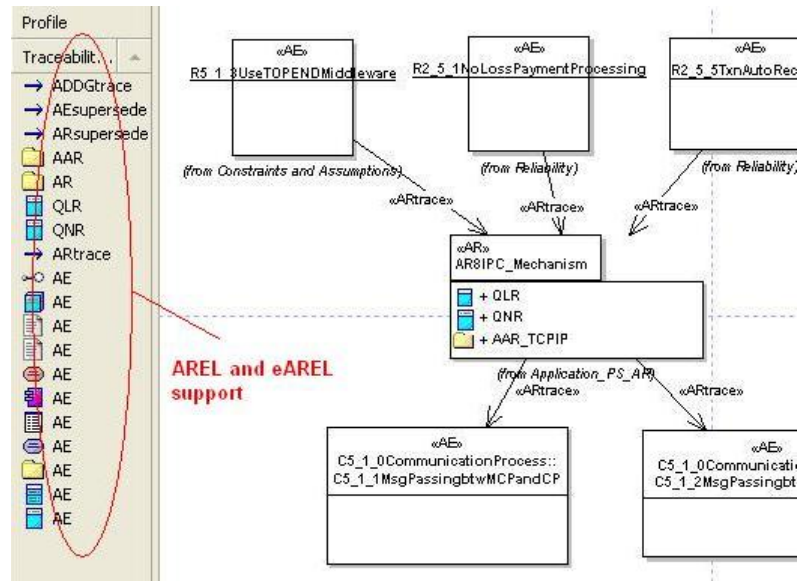


Figure 11.4: The AREL Constructs for Modelling

This works similarly for design rationale *AR*. When the user needs to relate the *AE* to the *AR*, the user has to click on *ARtrace* icon and drag the arrow from *AE* to *AR*. This action creates the causal link between them to indicate that the *AE* is the motivational reason for the *AR* decision. Similarly, *AEsupersede* and *ARsupersede* icons can be used to connect superseded architecture elements and architecture rationale, respectively.

A key feature of AREL is capturing design rationale in *AR* and *AAR*. By using the stereotype extension in Enterprise Architect, we provide a convenient way to input design rationale using the design rationale capture templates. After creating an *AR* on the design canvas, users need to drop *QNR*, *QLR* and *AAR* to manually create them within an *AR*.¹

An *AR* contains information in a hierarchy as shown in Figure 11.5. Since *AR* is implemented as a UML *package*, it can contain *AAR*, *QLR* and *QNR*. *AAR* is also implemented as a UML *package* and it can contain *QLR*, *QNR* and the alternative design.

The design rationale captured in *QLR* and *QNR* are implemented using *tagged values*. Users can enter the information through the pre-defined templates, one for each type of design rationale. A sample of qualitative rationale *QLR* is shown in Figure 11.6 and a sample of quantitative rationale *QNR* is shown in Figure 11.7.

¹Unfortunately, we cannot automatically create the hierarchical structure in *AR* because we cannot modify Enterprise Architect to integrate this feature. Please see Section 11.5 on the limitations of the tool.

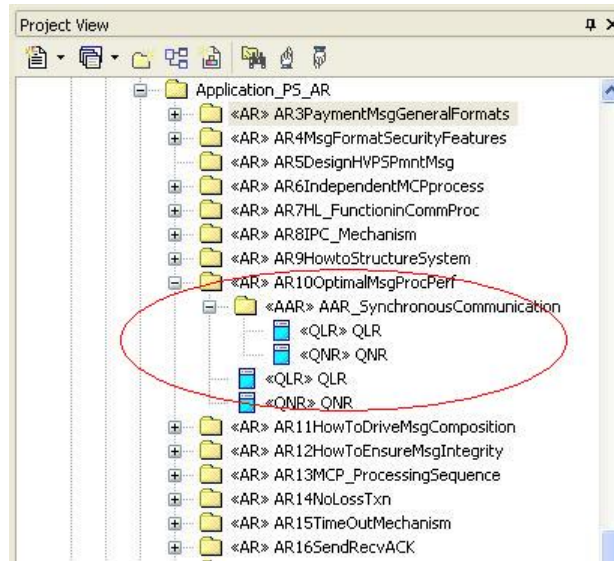


Figure 11.5: A hierarchy of elements in an AR

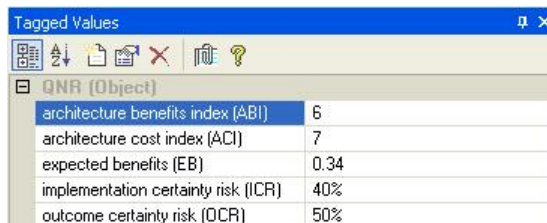
The screenshot shows a 'Tagged Values' window with a table of qualitative rationale (QLR) for an object. The table has two columns: the attribute name and its value.

QLR (Object)	
assessment_decision	Async mdg processing because a single process can handle multiple connections
assumptions	multiple connections in a single process
constraints	
date	Dec 1995
decision_maker	AT
history	
implications	check missing message by using seq number
issue	should async / sync messaging be implemented?
non-risks	no missing or duplicated messages
risks	complexity
strengths	fast turnaround in message processing
tradeoffs	performance against complexity
weaknesses	message matching requires DB support

Figure 11.6: An Example of a Qualitative Rationale (QLR)

11.2 Checking AREL models

The structure of the AREL model has been defined in a particular way so that architecture rationale can be related to the architecture elements in a causal relationship. Its definition is contained in Section 6.2. We had to implement a custom checking program called the AREL Tool for a number of reasons: (a) we are not able to modify Enterprise Architect to verify the AREL structure, we have developed the AREL Tool to read the Enterprise Architect repository and to check the AREL model directly; (b) OCL is not implemented in Enterprise Architect to support model checking; (c) although AREL model requires an acyclic graph, users can still create a UML model that is recursive, as such the model checking program must be a trigger to traverse the entire model and it must terminate,



QNR (Object)	
architecture benefits index (ABI)	6
architecture cost index (ACI)	7
expected benefits (EB)	0.34
implementation certainty risk (ICR)	40%
outcome certainty risk (OCR)	50%

Figure 11.7: An Example of a Quantitative Rationale (QNR)

hence a separate program is required. The AREL Tool is developed in Microsoft .Net and runs on the Windows XP platform.

The consistency checking of the AREL model follows a number of rules:

- Detect any directed cycles in the AREL model. An error is shown when it happens.
- Detect any improper model construct which violates the AREL definition. The anomalies are: two *AEs* are linked directly by `<<ARtrace>>` without an *AR*; two *ARs* are linked directly by `<<ARtrace>>` without an *AE*; the leave node is an *AR*; the root node is an *AR*. The tool reports these as errors.
- Detect any isolated items such as an *AE* or an *AR* which are not connected to any other elements. The tool reports these as warnings.

The AREL Tool is contained in a program (i.e. *EAModelProjectUI.exe*). It is a stand-alone program and can be downloaded from [155]. The installation of the program is done by running *Setup.exe*. When the program starts up, the user has to specify the Enterprise Architect repository that contains the AREL model. Then the AREL Tool would open the repository and a number of options are displayed in a window for the user to select. This is shown in Figure 11.8.

The user can click the *Check Model* button to start the consistency checking. When checking is complete, the tool displays a screen to show any errors or warnings which are present in the AREL model in a pop-up window. If any error is reported, then the result would indicate that the AREL model is *inconsistent*. An example of the window is shown in Figure 11.9. A detailed error report is also produced and it is located in the same directory as the Enterprise Architect repository. If errors are present, users will need to update the AREL model to correct them and re-run the AREL Tool to check that they have been fixed. The user manual of the AREL Tool is contained in Appendix A and so its operations are not described in details here.

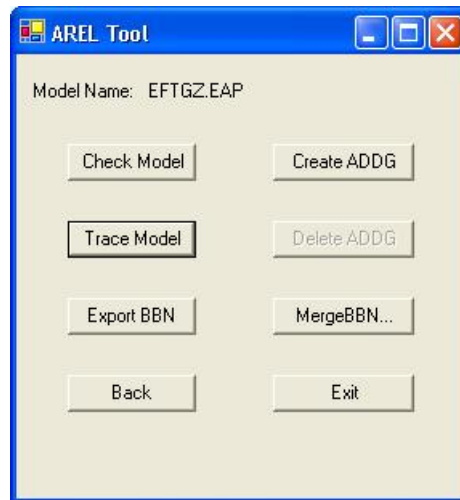


Figure 11.8: AREL Tool Menu Options

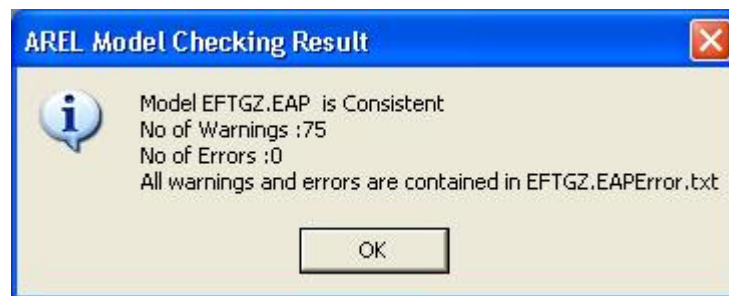


Figure 11.9: AREL Tool Consistency Check Results

11.3 Tracing AREL models

As discussed in Chapter 9, the tracing of architecture rationale and architecture elements is important to help architects understand architecture design reasoning. The AREL Tool enables architects to specify and trace an AREL model. Architects can select the *Trace Model* button to activate tracing (see Figure 11.8) and a new window would be displayed (see Figure 11.10).

Architects then input a valid architecture element *AE* which is where tracing will begin. Architects then select the Element Type(s) and the Element Subtype(s) to specify those elements that are required. There are four different viewpoint types classified in AREL: Business, Information, Application and Technology. Each of them provides a perspective to represent the architecture. There are five sub-viewpoints in the Business Viewpoint, they specify different types of drivers which motivate the creation of an architecture. These sub-viewpoints are functional requirements, non-functional requirements, information system environment, business environment and technical environment. Ar-

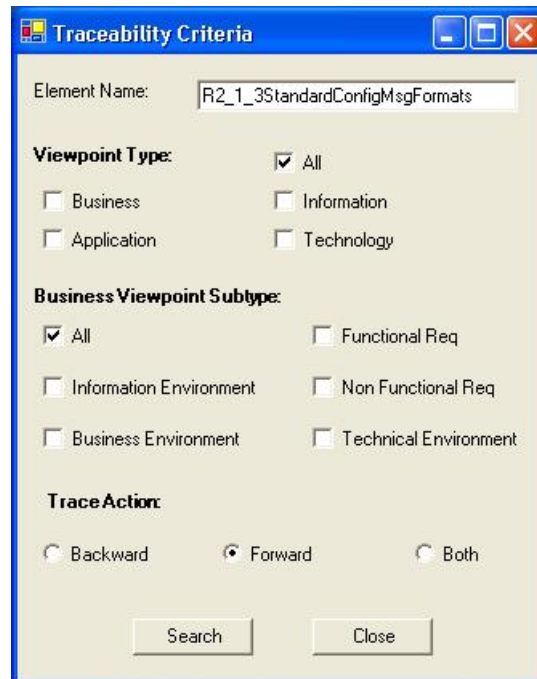


Figure 11.10: Window for Specifying AREL Trace Criteria

Architects may select all of the viewpoints or only those viewpoints that are interesting to them .

Architects also need to specify the trace actions. The trace actions dictate whether the traceability is to be forward tracing, backward tracing or both. In forward tracing, only those *AEs* and *ARs* which are forwards from the specified *AE* are retrieved (i.e. same direction as *ARtrace* links). In the backward tracing, only those *AEs* and *ARs* which are backwards from the specified *AE* are retrieved (i.e. opposite direction as *ARtrace* links). When both actions are specified, then all forwards and backwards *AEs* and *ARs* are retrieved.

The AREL Tool starts tracing from the specified *AE* element to the root nodes in a backward tracing, and to the leaf nodes in a forward tracing. If the architecture elements do not belong to the specified element type or element sub-type in the scope of tracing, then normally these unwanted elements will not be retrieved in the result. However, there is an exception. If the unwanted elements (i.e. those elements which types have not been specified) are part of a causal relationship between an *AE* where tracing starts and *AEs* which are part of the desired results, then the unwanted elements are retrieved. This is because if these unwanted elements are not retrieved, the chain of causal relationship would be broken.

The results of the trace is created within the Enterprise Architect repository in a

11.4. Analysing AREL with BBN

separate diagram called *Traceability Graph*. A sample of a result is shown in Figure 11.11. Through Enterprise Architect, architects can access the results of the tracing, explore the design rationale contained in the ARs and so on.

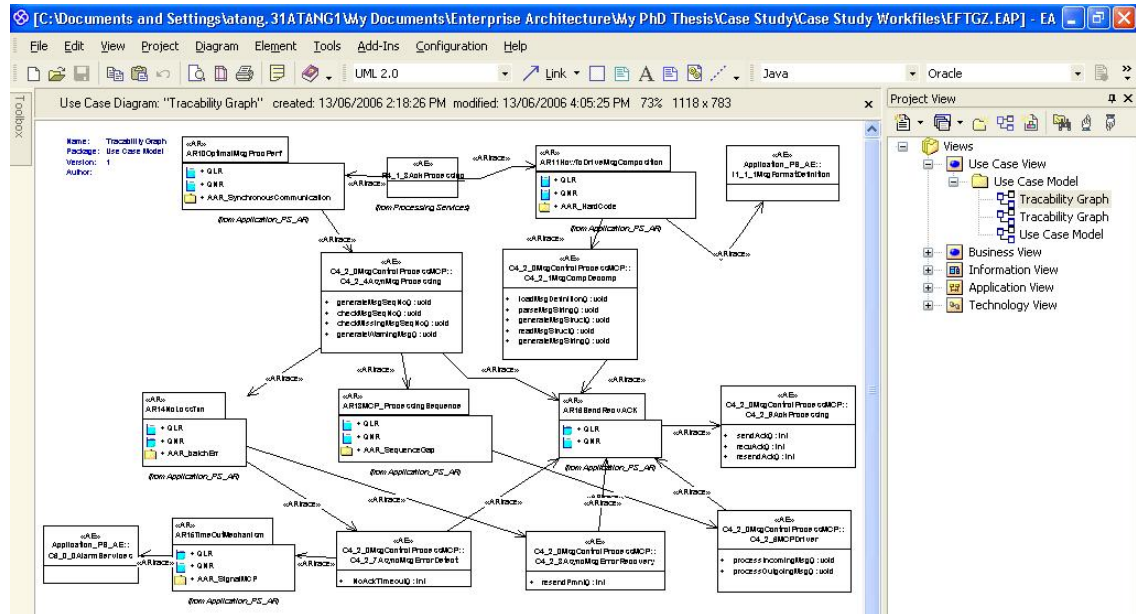


Figure 11.11: An Example of AREL Trace Result

11.4 Analysing AREL with BBN

As noted earlier, Enterprise Architect does not contain any functionality to capture or compute the probabilities required in a BBN. The reason for it is because we cannot build this facility into Enterprise Architect without implementing all the BBN functionalities. It would be a highly complex exercise. Therefore, we rely on another program to perform the BBN functionalities. Netica (version 2.17) [107] provides the BBN functionalities that allow us to implement the AREL-BBN modelling and it has the flexibility to interface with other packages as well.

The nodes and links in an AREL model is constructed with Enterprise Architect, and it has to be an acyclic graph to be used for BBN computation. The probability capture is performed by Netica. As such, the information which are captured by the two packages need to be merged together.

The steps to create and merge this information are described below (also see Figure 11.12):

11.4. Analysing AREL with BBN

- Step 1 - construct the design objects and capture the design decisions using Enterprise Architect.
- Step 2 - use the AREL Tool to convert the AREL model in the UML notation into the Netica repository format (i.e. Netica *.dne* extension). This is achieved by clicking the *Export BBN* button to export the AREL model to Netica.
- Step 3 - Netica reads the exported file and designers can assign prior and conditional probabilities to *AEs* and *ARs* for BBN computation.

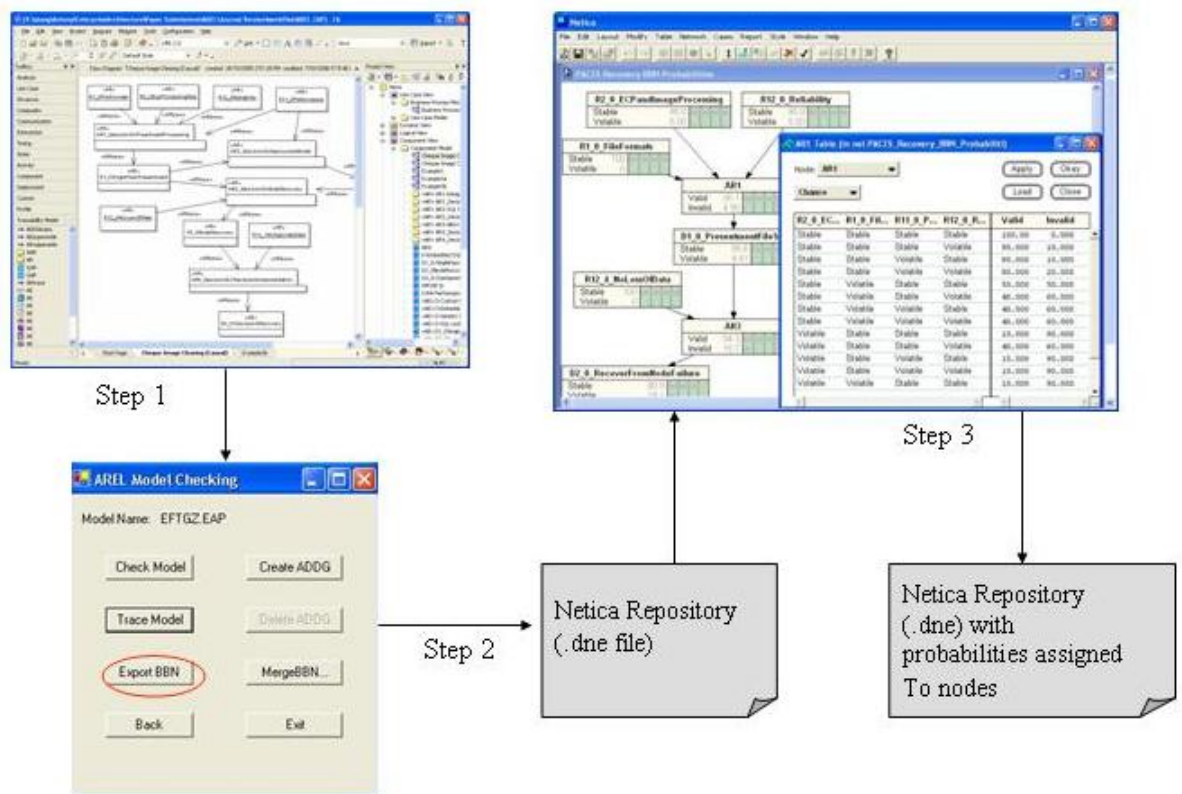


Figure 11.12: A Process to Extract AREL Model from UML into BBN

As designers continue to make decisions and modify the decision structure over the life-span of a system, the AREL model (in UML) would change accordingly. These changes need to be reflected in Netica without having to re-enter all the probabilities previously assigned. This would happen when the following design changes take place:

- *AE* and *AR* nodes are added - prior or conditional probabilities need to be captured for the new nodes.
- *AE* and *AR* nodes are deleted - those nodes which depend on the deleted nodes should have their CPTs adjusted.

- new links are added or existing links deleted - the CPTs of affected nodes need to be adjusted.

Given that the UML repository contains the most current version of the AREL model, and the BBN model already contains some assigned probabilities, the BBN model needs to be updated and synchronised with the current AREL model. Figure 11.13 shows an example of how to synchronise them.

- Step 1 - the AREL model has been updated.
- Step 2 - the AREL model in UML is converted to the Netica format using the AREL Tool. This is achieved by clicking the *Export BBN* button to export the AREL model to Netica.
- Step 3 - since there already exists a BBN representation of the AREL model prior to the update in Step 1 and we want to retain the probabilities that have already been entered, so we merge the probabilities from the original BBN representation with the newly exported model. This is achieved by using the *Merge BBN* button to merge the two files. The result of the merge is to create a new BBN model with the latest AREL structure and the probabilities from the previous BBN model merged with the new model.
- Step 4 - after merging the two sources, the BBN model now reflects the new AREL structure. However, the nodes in this model which have been affected by any changes require probabilities assignment and/or adjustments. The probabilities can be entered using Netica.

Using a combination of Enterprise Architect, AREL Tool and Netica, we provide the necessary tools to automate the design rationale capture, retrieval, traceability, change impact analysis and BBN reasoning.

11.5 Limitations

As noted at the beginning of this chapter, we have set ourselves a number of objectives when constructing the AREL tool-set and most of them have been achieved. However, there are a number of limitations. First, AREL operations cannot be tightly integrated with Enterprise Architect. For instance, the AREL operations cannot be directly activated from the Enterprise Architect menu options to fully integrate AREL functionalities into

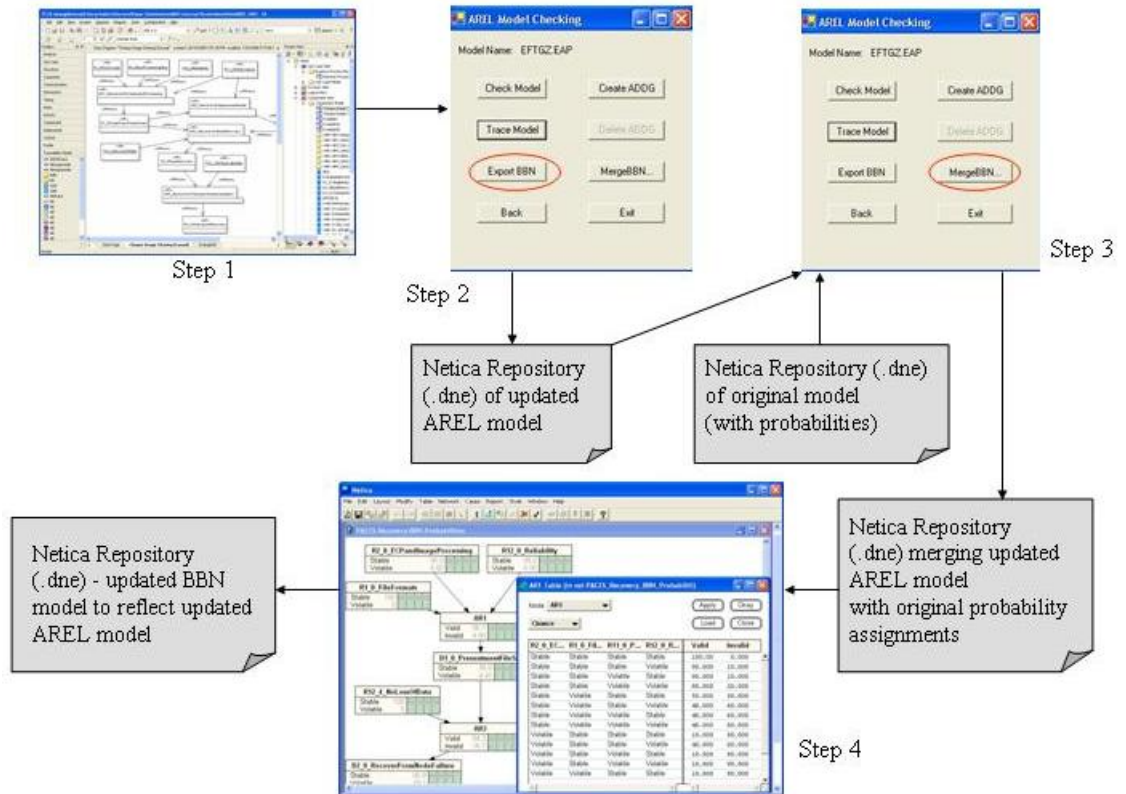


Figure 11.13: A Process to Synchronise Change between UML and BBN

Enterprise Architect. This is because of a software incompatibility between Enterprise Architect and the Microsoft SOE at the university. The only option that we had at the time was to develop a program to access the Enterprise Architect repository to provide a similar functionality. The usability is compromised because of this restriction. As a prototype to demonstrate the concept, the tool provides sufficient functionality to carry out the key tasks.

Second, we have no access to the source code of either Enterprise Architect or Netica, therefore there cannot be a seamless integration where prior probabilities and conditional probability tables can be captured and computed within the UML tool. The integration of the two tools can only be achieved by using our custom-developed programs to access their respective repositories and creating intermediate files for their integration.

Finally, although most of the AREL features described in this thesis have been implemented, the AREL tool-set is a proof-of-concept and it is immature for real-life applications. This is because a number of usability features must be implemented if it is to be widely used in a commercial setting. For examples, automatic creation of an *AR* package; the ability of the UML tool to provide multi-perspective views. Users want to be

able to show, hide and search architecture decisions, elements and relationships based on the perspectives that are sought. Also, the automation and usability features to support eAREL are not implemented. However, such features can be realised in a commercial implementation quite easily.

11.6 Summary

In this chapter, we have described the tool-set to support the capture and applications of the AREL model. The tool-set comprises three elements: Enterprise Architect is a commercially available tool for drawing UML diagrams and performing object oriented design; Netica is a commercially available tool for computing BBN; the AREL Tool integrates them together.

We have enhanced Enterprise Architect to enable architects to capture quantitative and qualitative design rationale. The AREL Tool supports the traceability of the AREL model. It creates trace results in UML for analysis by architects. The AREL Tool checks AREL models to ensure that they are consistent and error free. Since the application of BBN requires data inputs from both Enterprise Architect and Netica, the two sources of information have to be merged. This function is performed by the AREL Tool.

Chapter 12

Conclusions

12.1 Summary

A fundamental issue of architecture design presently is a lack of systematic approach to design reasoning. The quality of system and software architecture design can be highly dependent on the person who designs it. How architecture is designed depends on an architect's experience, knowledge and decision making abilities. As such, the reasoning of design decisions directly affects the architecture design and its quality. Design reasoning is an intuitive process that is performed by all designers, but little attention is paid to its explicit deliberation, verification or documentation. As a result, it affects architecture design in three ways: first, the reasoning behind an architecture design might be incorrect or incomplete but there is no explicit information or design rationale for its verification; second, once the system development has been completed, the architecture design can be costly and difficult to change at that stage if it is incorrect or not optimal; finally, it is sometimes difficult to understand the architecture design for maintenance purposes if the design rationale is not documented.

In this thesis, we have investigated the use and documentation of architecture design rationale in the software industry to understand the current practice. Based on the results, we have proposed ways to improve the representation and the use of architecture design rationale. In summary, we have addressed the following research questions in this thesis:

- Explore the use and documentation of architecture design rationale in the software industry. The objective of this investigation is to establish a basis for the follow-on research. A number of questions have been explored:

- Is design rationale important in architecture development?
- Is design rationale important in system maintenance?
- What is the current state of practice of design rationale in the software industry?
- How to improve the representation of design rationale for architecture development?
- How to implement the traceability between requirements, architecture design elements and design rationale?
- How to quantify and estimate change impacts using architecture design elements and design rationale?

As a result of addressing these research questions, we have achieved the following. Firstly, we have conducted a survey to establish the current practice of using and documenting design rationale. We have found that most architects see the importance of design rationale but the ways they document design rationale vary and are not systematic. Secondly, we have developed the Architecture Rationale and Element Linkage (AREL) model to represent design rationale. The AREL model represents a causal relationship between architecture rationale and architecture elements, which provides an explanation of how requirements, assumptions and constraints can be related to architecture design objects. Architecture rationale captures both qualitative and quantitative rationale. Together they explain the architecture decision by recording the design issues, arguments, assumptions, design alternatives, costs, benefits and risks. Thirdly, we have developed the Architecture Rationalisation Method (ARM) to guide the architecture design process in decision making. Finally, we have developed different applications of AREL for traceability and change impact analysis. Their purposes are to support architecture development and maintenance.

We have conducted an empirical study to test the viability of AREL based on interviewing experts who develop electronic payment systems. Experts who participated in the empirical study indicated that AREL can be very useful in helping architects understand the design of a system. Based on an operational Electronic Fund Transfer System, we have demonstrated how to capture and represent architecture design rationale using AREL. We have also demonstrated how to apply traceability methods and BBN methods to support change impact analysis.

We acknowledge that we have not quantified the cost effectiveness of using AREL, which is context dependent on factors such as the system life-span, system complexity and developers' familiarity with the system. Despite all that, we have demonstrated and, through the empirical study, obtained positive confirmation on the viability of this design

reasoning methodology. We believe that the AREL model and the ARM process have provided a new and systematic approach for architects to design systems.

12.2 Contributions

This thesis has presented a new approach, called Architecture Rationale and Elements Linkage (AREL), to capturing and encapsulating the design reasoning of system and software architecture. This is an improvement over the current software industry practice. It is also an improvement over the argumentation-based design rationale methods. There are a number of areas where this thesis has contributed to the knowledge of software engineering.

12.2.1 Design rationale survey

We have conducted a survey on the use and documentation of design rationale. This survey is important because it has established the need for having design rationale in architecture design. From the survey, we have obtained evidence to support that design rationale is an important part of the design documentation. Practitioners believe that design rationale should be documented. There is also a general perception that methodology and tool support for design rationale is lacking and they are barriers to design rationale documentation.

We have identified nine generic design rationales and the respondents have indicated additional types of design rationale that are useful for design reasoning. Together they form a basis for grouping and capturing design rationale. The survey results have indicated that architects work on a variety of tasks such as requirements analysis, tender analysis, architecture design and software design, as well as having management responsibilities. Program design and test planning is a much smaller part of their job. In the survey, we have found indications that there might be a tendency to present “*good news*” rather than “*bad news*” during the design process. It is because of this likely bias, there is a need to use design rationale to support the verification of the architecture design. The survey has found that there is a strong justification to document design rationale due to the architects’ tendency to forget their own design. It is also useful when architects have to understand systems that are designed by the others.

The survey has shown that architects often carry out risk assessments in architecture design. Over half of the respondents explicitly quantify risks. However, when they were asked what risk level would be acceptable, there was no consensus. This implies that there

is no common understanding on how to measure risks, and the risk assessment process in architecture design is not well understood. It suggests that further investigation in this area is required.

It has been found that the methodology and tool support for design rationale capture and retrieval is inadequate. The various tools that have been reported, including word processors and UML-based tools, do not have traceability features to support systematic design rationale description and retrieval. Therefore, it is important to understand how best to capture, represent and use design rationale and then develop tools to provide a design rationale enabled development environment.

In summary, the survey has allowed us to gain an in-depth understanding of the application of design rationale in the software industry. It has confirmed that even architects consider them to be useful, there is little in terms of methodology and tools to support their applications. The survey has established the need to further study architecture design rationale, and clarified a number of important research questions that we address in this thesis.

12.2.2 Design rationale representation

In this thesis, we have introduced a rationale-based architecture model (AREL) to represent architecture design rationale. The AREL model uses two types of reasoning support: motivational reasons and design rationale. Motivational reasons induce architecture issues that need to be resolved. Architecture design decisions are justified by architecture design rationale which is comprised of qualitative rationale, quantitative rationale and alternative design options. In an architecture design, intricately inter-dependent design objects often have common requirements, assumptions, constraints and decisions. They can be explicitly related and reasoned in AREL models. This reasoning support complements the documentation of design structures and interfaces commonly appeared in design specifications.

AREL captures two types of design rationale: qualitative design rationale *QLR* and quantitative design rationale *QNR*. *QLR* represents the reasoning and the arguments, in a textual form, for and against a design decision. *QNR* uses indices to indicate the relative costs, benefits and risks of design options. Together they can help architects justify architecture decisions. This representation is an improvement over the argumentation-based methods because it simplifies the representation of qualitative argumentation, and provides quantification to justify why a design is chosen over its alternatives. The AREL model is extended by eAREL to support design evolution. Both AREL and eAREL

implementations use UML for easy adoption by the software industry.

The AREL model representation is an improvement over existing types of design rationale methods described in Chapter 3 because it: (a) provides a structured approach to capturing architecture rationale; (b) provides comprehensive reasoning that encompasses both qualitative and quantitative design rationale; (c) associates architecture rationale to architecture elements in reasoning. We have demonstrated the application of the AREL model in a real-world system using an electronic payment system. In an empirical study, expert architects in the area of electronic payment system have been asked to compare traditional design specifications and the AREL model. They have found that the AREL model complements the design specifications to provide a better explanation of the design reasoning of the system.

12.2.3 Design rationale applications

Architecture design completeness

One of the issues in software architecture is the the difficulty of articulating what is a complete architecture design. This issue directly affects how much design details architects have to carry out to be certain that the architecture is viable and complete. Using AREL as a basis, we have introduced Architecture Rationalisation Method (ARM) to facilitate the architecture design process. ARM uses the risk indices in *QNR* to guide architecture decomposition. By considering the implementation risks (*ICR*) and outcome certainty risks (*OCR*), architects can determine that an architecture is complete when these risks are at an acceptable level and all major requirements have been addressed. This process of using risk levels to guide architecture decisions has the benefit of assessing whether the requirements can be fulfilled and the architecture design can be implemented. It allows the architects to focus on these two aspects of architecture design until such time that they are achievable.

Using the Cost-benefit Ratio (*CBR*) that is defined in *QNR*, architects can compare the relative benefits of design alternatives quantitatively. Together with *QLR*, they can help architects make and justify their decisions during architecture design, and these decisions can be verified independently by assessing the architecture rationale.

Architecture rationale and design tracing

Understanding an architecture design often requires knowing the motivations and the reasoning of the design. This is important if architects have to verify or maintain the system. The AREL model can capture the necessary knowledge to attain such goals. However, the retrieval of such knowledge becomes an issue if the design is complex. Therefore, we have applied different traceability methods to support the traversal of architecture design rationale, requirements (as motivational reasons) and design objects.

Three type of tracing can be applied to AREL and eAREL. Forward tracing supports impact analysis. Given a requirement, the design objects and the design rationale that are impacted can be traversed. Backward tracing supports root-cause analysis. Given a design element, its causes such as requirements, assumptions, constraints and design rationale can be traversed. Evolution tracing supports the traceability through the design evolution of an architecture element or an architecture rationale. The implementation of AREL and its traceability are supported by the AREL Tool and Enterprise Architect.

We have used a partial architecture design of the Electronic Fund Transfer system as a case study. During this exercise, we have recovered and recaptured their architecture rationale. With the case study, we have demonstrated the application of AREL traceability in a number of examples to trace the architecture rationale in supporting maintenance activities.

Architecture design change impact analysis

AREL overcomes the issue of relating design decisions to design elements by joining them in causal relationships. This is an improvement over argumentation-based design rationale systems. However, when an architecture is subject to a change, the change can *ripple* through the architecture design and cause other changes. This is because inter-related requirements, constraints and assumptions might have been affected by the initial change. Another issue in managing change is that the change impact is often difficult to quantify, making it difficult for the architect to make an initial assessment of where in the design and how much change is involved.

We have addressed these two issues by applying Bayesian Belief Networks (BBN) to AREL. BBN is used to quantify the strength of the causal relationship between architecture elements and architecture rationale. This strength is represented by the probability that an architecture element is *stable*, and the probability that an architecture decision is *valid*. They quantify the likelihood of change impact based on the dependency between

architecture decisions and elements. We have identified three different reasoning methods in which change impact can be analysed: (a) predictive reasoning; (b) diagnostic reasoning; and (c) combined reasoning. These methods allow the architects to trace the ripple effect using quantifiable probability values.

12.2.4 Tool implementation

The experts who have participated in the empirical study tell us that tool support is key to the successful implementation of AREL. As such, we have created a tool-set which is aimed at demonstrating the viability and usefulness for the software industry.

We choose to use an industry standard notation UML supported by a commercial tool, Enterprise Architect. This means that the designers can design the architecture and capture the design rationale using the same tool. We have implemented the AREL Tool to support consistency checking and to integrate the Enterprise Architect repository with the BBN tool. We have made use of Netica to support our BBN implementation and analysis of AREL. Netica is integrated with the Enterprise Architect without redundant data re-entry. It reduces the efforts required to capture design rationale for analysing change impacts.

12.3 Future work

In this thesis, we have proposed to use risk assessments for evaluating the completeness of an architecture design. In the BBN analysis, we also use probabilities, as a kind of uncertainties, to measure the validity of decisions. Although architects are aware that risk assessment is something that is important, there does not seem to be a commonly acceptable method to apply them. This has been demonstrated by the survey that we have conducted. Therefore, even when we suggest that AREL is useful, its success can depend on the way architects assess risk. An objective assessment would consider all the available facts, but it is easy to be biased when an architect interprets the facts differently. So does it come down to the abilities of an individual? We think not. First, a design reasoning approach such as AREL will make architects more aware of the necessity to assess risk. Second, an architect's ability and objectivity to assess risk should be measured over a longer term, and not by a few decisions. Therefore, we think that further research could help us understand the psychology of architects when designing a system, and thereby provide a better way to minimise risks and improve the chances of a successful architecture.

The survey has given us many insights into how design rationale is used and doc-

umented, but the survey does not allow us to observe the actual capture process that practising architects go through. We plan to design and execute a large scale field study into the software design process. This study will consist of multiple case studies, as described by [174] and successfully demonstrated by [27]. Some of the techniques we plan to use include studying practitioners' attitudes towards design rationale use and documentation. We will conduct in-depth interviews and examination of design specifications. We expect these experimental techniques will enable us to discover the answers to many questions regarding the behaviour of architects during design.

Although we can show AREL's explanatory and reasoning power to help understand an existing system with the empirical study, we cannot demonstrate AREL's reasoning capabilities to support a new system's development in this thesis. It is our intention to test this aspect of AREL in the future. In a new experiment, we will apply AREL to the architecture design process to assess its performance on the effectiveness of capture and its usefulness in supporting architecture design. The results will allow us to further refine AREL.

The experts from the empirical study have reacted positively towards the AREL approach, but they have also questioned the cost-effectiveness of the method. A cost-benefit analysis on AREL's applications would be necessary. Different perspectives could be taken in this analysis: (a) the cost and benefit of using AREL and ARM in different phases of the software development life-cycle; (b) the cost and benefit of applying each of the techniques such as traceability, risk analysis using *QNR*, and impact analysis using BBN; (c) the cost and benefit of applying these techniques to specific domains or different project scales.

The AREL tool-set is only a proof-of-concept, its effectiveness in practice requires a formal evaluation. The tool-set itself also requires better integration between UML modelling and BBN computation. Design rationale capture using AREL and ARM relies very much on architects recording the information diligently. Any automated design rationale capture to enhance this process would be an improvement. Ideas such as language processing, white board capture tool can be explored.

In this thesis, we have provided a high-level classification of the architecture elements by viewpoints. This classification supports the scoping of traceability results. We think that further classifications are required because it will provide further scoping of the trace results when using the AREL traceability methods.

The application of BBN to AREL has provided a foundation for change impact analysis. Some further research opportunities in architecture design and software engineering are possible: (a) to predict the cost of change impact to architecture enhancements; (b)

to examine the complexity of the system through analysing the dependency between architecture elements and decisions; (c) to consider using Dynamic Bayesian Networks for temporal analysis as systems evolve over time; (d) to derive from AREL a more abstract and concise model that represents the associations between decisions.

Bibliography

- [1] D. Ahern, A. Clouse, and R. Turner, *CMMI distilled : a practical introduction to integrated process improvementn.* Boston, USA: Addison-Wesley, 2004.
- [2] T. Al-Naeem, I. Gorton, M. A. Babar, F. A. Rabhi, and B. Benatallah, "A quality-driven systematic approach for architecting distributed software applications." in *Proceedings 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 244–253.
- [3] M. Ali-Babar, I. Gorton, and B. Kitchenham, "A Framework for Supporting Architecture Knowledge and Rationale Management," in *Rationale Management in Software Engineering.* Springer, 2006, pp. 237–254.
- [4] J. Asundi, R. Kazman, and M. Klein, "Using Economic Considerations to Choose Amongst Architecture Design Alternatives," Carnegie Mellon University, Tech. Rep. CMU/SEI-2001-TR-035, ESC-TR-2001-035, December 2001.
- [5] B. Ayyub, "A Practical Guide on Conducting Expert-Opinion Elicitation of Probabilities and Consequences for Corps Facilities," Technical Report IWR Report 01-R-01, Institute for Water Resources, Alexandria, VA, USA, Tech. Rep., 2001.
- [6] M. Babar, A. Tang, I. Gorton, and J. Han, "Industrial Perspective on the Usefulness of Design Rationale for Software Maintenance: A Survey," in *Proceedings 6th International Conference on Quality Software (QSIC 2006).* IEEE, 2006.
- [7] R. Balzer, T. E. C. Jr., and C. C. Green, "Software technology in the 1990's: Using a new paradigm." *IEEE Computer*, vol. 16, no. 11, pp. 39–45, 1983.
- [8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice.* Boston: Addison Wesley, 2003.
- [9] S. Beecham, T. Hall, C. Britton, M. Cottee, and A. Rainer, "Using an expert panel to validate a requirements process improvement model." *Journal of Systems and Software*, vol. 76, no. 3, pp. 251–275, 2005.

- [10] B. W. Boehm, *Software Engineering Economics*. New Jersey: Prentice Hall PTR, 1981.
- [11] J. Bosch, "Software Architecture: The Next Step," in *Proceedings 1st European Workshop on Software Architecture (EWSA)*, St Andrews, UK., 2004, pp. 194–199.
- [12] L. Bratthall, E. Johansson, and B. Regnell, "Is a Design Rationale Vital when Predicting Change Impact? A Controlled Experiment on Software Architecture Evolution," in *Second International Conference on Product Focused Software Process Improvement*, 2000, pp. 126–139.
- [13] S. Buckingham Shum and N. Hammond, "Argumentation-Based Design Rationale: What Use at What Cost?" *International Journal of Human-Computer Studies*, vol. 40, no. 4, pp. 603–652, 1994.
- [14] J. Burge, "Software Engineering Using design RATIONale," Ph.D. dissertation, Worcester Polytechnic Institute, 2005.
- [15] S. Bushell, "It is the Business, Stupid." [http : //www.cio.com.au/index.php/id;719608738;pp;1;fp;16;fpid;0](http://www.cio.com.au/index.php/id;719608738;pp;1;fp;16;fpid;0), 2006.
- [16] Carnegie Mellon Software Engineering Institute, "CMMI SE/SW Version 1.1," <http://www.sei.cmu.edu/cmmi/models/model-components-word.html>, 2002.
- [17] Carrington, K. and Pratt, A., "How Far Have We Come? Gender Disparities in the Australian Higher Education System," 2003.
- [18] J. Carroll and M. Rosson, "Deliberated Evolution: Stalking the View Matcher in Design Space," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 4, pp. 107–146.
- [19] R. Charette, *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill Book Company, 1989.
- [20] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Boston: Kluwer Academic, 2000.
- [21] CIO-Council, "Federal Enterprise Architecture Framework version 1.1," <http://www.cio.gov/archive/fedarch1.pdf>, 1999, last accessed: May 21, 2004.
- [22] —, "A Practical Guide to Federal Enterprise Architecture version 1.0," <http://www.cio.gov/archive/bpeaguide.pdf>, 2001, last accessed: May 21, 2004.
- [23] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures : Views and Beyond*, 1st ed. Addison Wesley, 2002.

- [24] E. Conklin and B.-Y. K.C., “A Process-Oriented Approach to Design Rationale,” in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 14, pp. 393–427.
- [25] J. Conklin, “Design Rationale and Maintainability,” in *Proceedings of the 22nd International Conference on System Sciences*, 1989, pp. 533–539.
- [26] J. Conklin and M. Begeman, “gIBIS: A hypertext tool for exploratory policy discussion,” in *Proceedings ACM Conference on Computer-Supported Cooperative Work*, 1988, pp. 140–152.
- [27] B. Curtis, H. Krasner, and N. Iscoe, “A field study of the software design process for large systems.” *Commun. ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.
- [28] A. Dardenne, A. Van-Lamsweerde, and S. Fickas, “Goal-directed Requirements Acquisition,” *Science of Computer Programming*, vol. 20, pp. 1–36, 1993.
- [29] M. Denford, J. Leaney, and T. O’Neill, “Non-Functional Refinement of Computer Based Systems Architecture,” in *The 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, 2004, pp. 168–177.
- [30] R. Dijkman, D. A. Quartel, L. F. Pires, and M. J. van Sinderen, “A Rigorous Approach to Relate Enterprise and Computational Viewpoints,” in *Proceedings 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 2004.
- [31] DoD, “Department of Defense Architecture Framework version 1.0 - Volumn 1 definition and guideline and Volumn 2 product descriptions,” <http://www.aitcnet.org/dodfw>, 2003, last accessed: April 1, 2004.
- [32] R. Domges and K. Pohl, “Adapting traceability environments to project-specific needs,” *Communications of the ACM*, vol. 41, no. 12, pp. 54–62, 1998.
- [33] A. Dutoit and B. Paech, “Rationale Management in Software Engineering,” in *Handbook of Software Engineering and Knowledge*, Nov. 30 2000.
- [34] A. H. Dutoit and B. Paech, “Rationale-Based Use Case Specification,” *Requirements Engineering*, vol. 7, no. 1, pp. 3–19, 2002.
- [35] T. Dyba, B. Kitchenham, and M. Jorgensen, “Evidence-based software engineering for practitioners,” *IEEE Software*, vol. 22, no. 1, pp. 58–65, 2005.
- [36] T. Dybå, “An instrument for measuring the key factors of success in software process improvement.” *Empirical Software Engineering*, vol. 5, no. 4, pp. 357–390, 2000.

- [37] A. Eden and R. Kazman, “Architecture, Design, Implementation,” in *International Conference for Software Engineering*, 2003, pp. pp 149 – 159.
- [38] A. Egyed, “A Scenario-Driven Approach to Traceability,” in *Proceedings 23rd International Conference on Software Engineering (ICSE)*, 2001, pp. 123–132.
- [39] —, “A Scenario-Driven Approach to Trace Dependency Analysis,” *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 116–132, 2003.
- [40] N. Fenton and M. Neil, “Software Metrics: Roadmap,” in *Proceedings 22nd International Conference on Software Engineering (ICSE)*, may 2000, pp. 357–370.
- [41] N. E. Fenton, S. L. Pfleeger, and R. L. Glass, “Science and Substance: A Challenge to Software Engineers,” *IEEE Software*, vol. 11, no. 4, pp. 86–95, July 1994.
- [42] Fenton, Norman and Pfleeger, Shari Lawrence, *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, 1996.
- [43] G. ‘Fischer, A. Lemke, and R. McCall, “Making Argumentation Serve Design,” in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 9, pp. 267–294.
- [44] M. Fyson and C. Boldyreff, “Using Application Understanding to Support Impact Analysis,” *Journal of Software Maintenance: Research and Practice*, vol. 10, pp. 93–110, 1998.
- [45] J. Galliers, A. Sutcliffe, and S. Minocha, “An Impact Analysis Method for Safety-Critical User Interface Design,” *ACM Transactions on Computer-Human Interaction*, vol. 6, no. 4, pp. 341–369, 1999.
- [46] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” in *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora, Eds. World Scientific Publishing Co., 1992, pp. 1–40.
- [47] D. Garlan, “Software architecture evaluation and analysis session report.” in *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture*, 2005, pp. 227–228.
- [48] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural mismatch or why it’s hard to build systems out of existing parts.” in *Proceedings 17th International Conference on Software Engineering (ICSE)*, 1995, pp. 179–185.
- [49] R. L. Glass, “The Software-Research Crisis,” *IEEE Software*, vol. 11, no. 6, pp. 42–47, 1994.

- [50] K. Goseva-Popstojanova, A. E. Hassan, A. Guedem, W. Abdelmoez, D. E. M. Nassar, H. H. Ammar, and A. Mili, "Architectural-Level Risk Analysis Using UML," *IEEE Trans. Software Eng.*, vol. 29, no. 10, pp. 946–960, 2003.
- [51] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proceedings International Conference on Requirements Engineering (RE)*, 1994, pp. 94–101.
- [52] O. Gotel and A. Finkelstein, "Extended Requirements Traceability: Results of an Industrial Case Study," in *Proceedings 3rd IEEE International Symposium on Requirements Engineering (RE)*, Annapolis, MD, USA, 1997, pp. 169–178.
- [53] T. Gruber and D. Russell, "Generative Design Rationale: Beyond the Record and Replay Paradigm," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 11, pp. 323–350.
- [54] T. R. Gruber and D. M. Russell, "Design Knowledge and Design Rationale: A Framework for Representing, Capture, and Use," Knowledge Systems Laboratory, Stanford University, California, USA, Tech. Rep., 1991.
- [55] T. Gruber and D. Russell, "Derivation and Use of Design Rationale Information as Expressed by Designers," Stanford University, Tech. Rep. KSL-92-64, 1992.
- [56] J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures Using Problem Frames," in *IEEE Joint International Conference on Requirements Engineering*, 2002, pp. 137–144.
- [57] M. Hamada and H. Adachi, "Recording Software Design Processes for Maintaining the Software," in *Proceedings of the 17th Computer Software and Applications Conference*, 1993.
- [58] J. Han, "TRAM: A Tool for Requirements and Architecture Management," in *Proceedings 24th Australasian Computer Science Conference*. Gold Coast, Australia: IEEE Computer Society Press, 2001, pp. 60–68.
- [59] —, "Designing for Increased Software Maintainability," in *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press, 1997, pp. 278–286.
- [60] P. Haumer, K. Pohl, K. Weidenhaupt, and M. Jarke, "Improving Reviews by Extended Traceability," in *Proceedings 32nd Hawaii International Conference on System Sciences*, 1999.
- [61] W. Heaven and A. Finkelstein, "UML profile to support requirements engineering with KAOS," *IEE Proceedings - Software*, vol. 151, no. 01, 2004.

- [62] J. D. Herbsleb and E. Kuwana, "Preserving knowledge in design projects: What designers need to know," in *Proceedings of the Conference on Human Factors in computing systems*. New York: ACM Press, Apr. 24–29 1993, pp. 7–14.
- [63] R. Hilliard, "Viewpoint modeling," in *Proceedings of 1st ICSE Workshop on Describing Software Architecture with UML*, 2001.
- [64] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*. Reading, Massachusetts: Addison Wesley, 2000.
- [65] L. Hope, A. Nicholson, and K. Korb, "Knowledge engineering tools for probability elicitation," Technical report, School of Computer Science and Software Engineering, Monash University, Tech. Rep., 2002.
- [66] T. Hughes and C. Martin, "Design traceability of complex systems," in *Proceedings 4th Annual Symposium on Human Interaction with Complex Systems*, 1998, pp. 37–41.
- [67] IEEE, "IEEE/EIA Standard - Industry Implementation of ISO/IEC 12207:1995, Information Technology - Software life cycle processes (IEEE/EIA Std 12207.0-1996)," 1996.
- [68] —, "IEEE/EIA Guide - Industry Implementation of ISO/IEC 12207:1995, Standard for Information Technology - Software life cycle processes - Life cycle data (IEEE/EIA Std 12207.1-1997)," 1997.
- [69] —, "IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)," 1998.
- [70] —, "IEEE Recommended Practice for Architecture Description of Software-Intensive System (IEEE Std 1471-2000)," 2000.
- [71] ISO/ITU-T, "Reference Model for Open Distributed Processing (ISO/ITU-T 10746 Part 1 - 4)," 1997.
- [72] A. Jarczyk, P. Loffler, and F. Shipman III, "Design Rationale for Software Engineering: A Survey," in *Proceedings of the 25th Hawaii International Conference on System Sciences*, 1992, pp. 577–586.
- [73] F. Jensen, *Bayesian Networks and Decision Graphs*. Springer-Verlag, 2001.
- [74] C. Jones, "Software Engineering: The State of the Art in 2005," <http://www.spr.com>, Software Productivity Research LLC, Tech. Rep., 2005.
- [75] M. Kajko-Mattsson, "A Survey of Documentation Practice within Corrective Maintenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

- [76] L. Karsenty, “An Empirical Evaluation of Design Rationale Documents,” in *CHI*, 1996, pp. 150–156.
- [77] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd, “SAAM: A Method for Analyzing the Properties of Software Architectures,” in *Proceedings 16th International Conference on Software Engineering (ICSE)*, 1994, pp. 81–90.
- [78] R. Kazman, M. H. Klein, M. Barbacci, T. A. Longstaff, H. F. Lipson, and S. J. Carrière, “The architecture tradeoff analysis method,” in *ICECCS*, 1998, pp. 68–78.
- [79] M. Keil, H. J. Smith, S. Pawlowski, and L. Jin, “‘Why didn’t somebody tell me?’: climate, information asymmetry, and bad news about troubled projects,” *SIGMIS Database*, vol. 35, no. 2, pp. 65–84, 2004.
- [80] B. Kitchenham and S. L. Pfleeger, “Principles of Survey Research, Parts 1 to 6,” *Software Engineering Notes*, 2001-2002.
- [81] B. Kitchenham, S. L. Pfleeger, B. McColl, and S. Eagan, “An empirical study of maintenance and development estimation accuracy.” *Journal of Systems and Software*, vol. 64, no. 1, pp. 57–77, 2002.
- [82] H. Koning and H. van Vliet, “A method for defining IEEE Std 1471 viewpoints,” *The Journal of Systems and Software*, vol. 79, pp. 120–131, 2005.
- [83] K. B. Korb and A. E. Nicholson, *Bayesian Artificial Intelligence*. Chapman and Hall / CRC, 2004.
- [84] P. Kruchten, “The 4+1 View Model of Architecture,” *IEEE Software*, vol. 12, no. 6, pp. pp 42–50, 1995.
- [85] —, “Software Architecture-A Rational Metamodel,” in *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints ’96) on SIGSOFT ’96 workshops*, 1996.
- [86] P. Kruchten, P. Lago, H. v. Vliet, and T. Wolf, “Building up and Exploiting Architectural Knowledge,” in *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture*, 2005.
- [87] W. Kunz and H. Rittel, *Issues as Elements of Information Systems*. Center for Planning and Development Research, University of California at Berkeley, 1970.
- [88] P. Lago and H. van Vliet, “Explicit assumptions enrich architectural models,” in *Proceedings 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 206–214.

- [89] N. Lassing, D. Rijsenbrij, and H. V. Vliet, "Viewpoints on Modifiability," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 4, pp. 453–478, 2001.
- [90] N. H. Lassing, P. Bengtsson, H. van Vliet, and J. Bosch, "Experiences with ALMA: Architecture-Level Modifiability Analysis," *Journal of Systems and Software*, vol. 61, no. 1, pp. 47–57, 2002.
- [91] J. Lee, "SIBYL: A Tool for Managing Group Decision Rationale," in *Proceedings of the Conference on on Computer-Supported Cooperative Work*, 1990, pp. 77–92.
- [92] —, "Extending the Potts and Bruns Model for Recording Design Rationale," in *13th International Conference on Software Engineering*, 1991, pp. 114–125.
- [93] —, "Design Rationale Systems: Understanding the Issues," *IEEE Expert*, vol. 12, no. 3, pp. 78–85, 1997.
- [94] J. Lee and K. Lai, "What's in Design Rationale," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 2, pp. 21–52.
- [95] T. C. Lethbridge, "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Engineering*, vol. 10, no. 1, pp. 311–341, 2005.
- [96] M. Li and C. Smidts, "A ranking of software engineering measures based on expert opinion." *IEEE Trans. Software Eng.*, vol. 29, no. 9, pp. 811–824, 2003.
- [97] A. Maclean, R. Young, V. Bellotti, and T. Moran, "Questions, Options and Criteria: Elements of Design Space Analysis," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 3, pp. 53–106.
- [98] L. May, "Major Causes of Software Project Failures," [http : //www.stsc.hill.af.mil/crosstalk/1998/07/causes.asp](http://www.stsc.hill.af.mil/crosstalk/1998/07/causes.asp), 1998.
- [99] R. McCall, "PHIBIS: Procedural Hierarchical Issue-Based Information Systems," in *Proceedings Int'l. Congress on Planning and Design Theory*, 1987.
- [100] —, "PHI: A conceptual foundation for design hypermedia," *Design Studies*, vol. 12, no. 1, pp. 30–41, 1991.
- [101] C. Meares and J. Sargent, "The digital work force: building infotech skills at the speed of innovation," [http : //www.technology.gov/reports/techpolicy/digital.pdf](http://www.technology.gov/reports/techpolicy/digital.pdf), 1999.

BIBLIOGRAPHY

- [102] M. Moore, R. Kazman, M. Klein, and J. Asundi, "Quantifying the value of architecture design decisions: lessons from the field," in *Proceedings 25th International Conference on Software Engineering (ICSE)*. Piscataway, NJ: IEEE Computer Society, May 3–10 2003, pp. 557–563.
- [103] T. Moran and J. Carroll, "Overview of Design Rationale," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 1, pp. 1–19.
- [104] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Non-functional Requirements: A Process-Oriented Approach," *IEEE Transactions on Software Engineering*, vol. 18, no. 6, pp. 483–497, 1992.
- [105] J. Nedstam, E.-A. Karlsson, and M. Höst, "The architectural change process." in *ISESE*, 2004, pp. 27–36.
- [106] P. Neumann, "Risks to the Public in Computers and Related Systems," [http :
//www.csl.sri.com/neumann](http://www.csl.sri.com/neumann).
- [107] Norsys Software Corp, "Netica - Application for BBN and Influence Diagrams User's Guide," <http://www.norsys.com/dl/NeticaMan.Win.pef.zip>, 1997.
- [108] B. Nuseibeh, "Weaving Together Requirements and Architecture," *IEEE Computer*, vol. 34, no. 3, pp. 115–119, March 2001.
- [109] —, "Crosscutting requirements," in *Proceedings 3rd international conference on Aspect-oriented software development*, 2004, pp. 3–4.
- [110] B. Nuseibeh, J. Kramer, and A. Finkelstein, "ViewPoints: meaningful relationships are difficult!" in *The 25th International Conference on Software Engineering*, 2003, pp. 676–681.
- [111] ObjectPlanet Inc., "Surveyor: Web-based Survey Application," <http://www.sharewareconnection.com/surveyor.htm>, 2002.
- [112] G. Olson, J. Olson, M. Storosten, M. Carter, J. Herbsleb, and H. Rueter, "The Structure of Activity During Design Meetings," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 7, pp. 217–239.
- [113] OMG, "UML 2 Infrastructure Final Adopted Specification," <http://www.omg.org/cgi-bin/doc?ptc/2003-09-15>, 2003.
- [114] D. Parnas and P. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, vol. 12, pp. 251–257, 1985.

- [115] D. L. Parnas, “The limits of empirical studies of software engineering.” in *ISESE*, 2003, pp. 2–7.
- [116] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*. San Mateo, CA: Morgan Kaufmann, 1988.
- [117] —, *Causality: models, reasoning, and inference*. Cambridge University Press, 2000.
- [118] P. Pender, *UML Bible*. Wiley Publishing Inc., 2003.
- [119] D. E. Perry and A. L. Wolf, “Foundations for the Study of Software,” *ACM SIGSOFT*, vol. 17, no. 4, pp. 40–52, 1992.
- [120] F. A. C. Pinheiro, “Formal and Informal Aspects of Requirements Tracing,” in *Workshop em Engenharia de Requisitos*, Brazil, 2000, pp. 1–21.
- [121] F. A. C. Pinheiro and J. A. Goguen, “An object-oriented tool for tracing requirements,” *IEEE Software*, vol. 13, no. 2, pp. 52–64, 1996.
- [122] C. Potts, “Supporting Software Design: Integrating Design Methods and Design Rationale,” in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 10, pp. 295–322.
- [123] —, “ScenIC: A strategy for inquiry-driven requirements determination,” in *Proceedings: 4th IEEE International Symposium on Requirements Engineering*. IEEE Computer Society Press, 1999, pp. 58–65.
- [124] C. Potts and G. Burns, “Recording the Reasons for Design Decisions,” in *10th International Conference on Software Engineering*, 1988, pp. 418–427.
- [125] C. Potts, “Software-engineering Research Revisited,” *IEEE Software*, vol. 10, no. 5, pp. 19–28, Sept. 1993.
- [126] C. Potts, K. Takahashi, and A. I. Antón, “Inquiry-Based Requirements Analysis,” *IEEE Software*, vol. 11, no. 2, pp. 21–32, 1994.
- [127] C. U. Press, “The Cambridge Dictionary Online,” [http : //dictionary.cambridge.org/](http://dictionary.cambridge.org/), 2006.
- [128] B. Ramesh and V. Dhar, “Supporting Systems Development by Capturing Deliberations During Requirements Engineering,” *IEEE Transactions on Software Engineering*, vol. 18, no. 6, pp. 498–510, 1992.
- [129] B. Ramesh and M. Jarke, “Towards Reference Models for Requirements Traceability,” *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 58–93, 2001.

- [130] L. Rapanotti, J. Hall, M. Jackson, and B. Nuseibeh, "Architecture-driven problem decomposition," in *12th IEEE International Conference on Requirements Engineering*, 2004, pp. 73–82.
- [131] S. Redwine and W. Riddle, "Software Technology Maturation," in *Proceedings of the 8th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1985, pp. 189–200.
- [132] W. C. Regli, X. Hu, M. Atwood, and W. Sun, "A Survey of Design Rationale Systems: Approaches, Representation, Capture and Retrieval," *Engineering with Computers*, no. 16, p. 209235, 2000.
- [133] Reserve Bank of Australia, "The Australian High Value Payment System," [http :
//www.rba.gov.au/PublicationsAndResearch/FinancialStabilityReview/Mar2004/Pdf/fsr02004](http://www.rba.gov.au/PublicationsAndResearch/FinancialStabilityReview/Mar2004/Pdf/fsr02004).
- [134] H. W. J. Rittel and M. M. Webber, "Dilemmas in a general theory of planning," *Policy Sciences*, vol. 4, no. 2, pp. 155–169, 1973.
- [135] R. Roeller, P. Lago, and H. van Vliet, "Recovering architectural assumptions," *The Journal of Systems and Software*, vol. 79, pp. 552–573, 2005.
- [136] J. Savolainen and J. Kuusela, "Framework for Goal Driven System Design," in *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, 2002.
- [137] C. B. Seaman, "The Information Gathering Strategies of Software Maintainers," in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 141–.
- [138] M. Shaw, "The Coming-of-Age of Software Architecture Research," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*. Los Alamitos, California: IEEE Computer Society, May12–19 2001, pp. 656–665.
- [139] F. Shipman III and R. McCall, "Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication," *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, vol. 11, no. 2, 1997.
- [140] S. Shum, "Cognitive Dimensions of Design Rationale," in *People and Computers VI: Proceedings of HCI'91*, 1991, pp. 331–344.
- [141] H. Simon, *The Sciences of the Artificial*. MIT Press, 1981.

- [142] Z. Simsek and J. Veiga, “A Primer on Internet Organizational Survey,” *Organizational Research Methods*, vol. 4, no. 3, pp. 218–235, 2001.
- [143] M. Singley and J. Carroll, “Synthesis by Analysis: Five Modes of Reasoning That Guide Design,” in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 8, pp. 241–265.
- [144] Six Sigma Academy, “Six Sigma Process Improvement Methodology,” <http://www.6-sigma.com/>, 2005.
- [145] W. Smith, *Best Practices: Application of DOORS to System Integration*. 1999 So. Bascom Av., Suite 700, Cambell, CA 95008, USA: QSS Quality Systems and Software, 1998.
- [146] I. Sommerville, *Software Engineering*, 7th ed. England: Addison-Wesley, 2004.
- [147] D. Soni, R. L. Nord, and C. Hofmeister, “Software architecture in industrial applications,” in *The 17th international conference on Software Engineering*, 1995, pp. 196–207.
- [148] J. Sowa and J. Zachman, “Extending and formalising the framework for Information Systems Architecture,” *IBM Systems Journal*, vol. 31, no. 3, 1992.
- [149] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause, “Rule-based generation of requirements traceability relations,” *The Journal of Systems and Software*, vol. 72, no. 2, pp. 105–127, July 2004.
- [150] Sparx Systems, “Enterprise Architect V5.00.767,” <http://www.sparxsystems.com/>, 2005.
- [151] S. H. Spewak, *Enterprise Architecture Planning*. John Wiley & Sons, 1992.
- [152] M. Svahnberg, C. Wohlin, L. Lundberg, and M. Mattsson, “A quality-driven decision-support method for identifying software architecture candidates.” *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, no. 5, pp. 547–573, 2003.
- [153] A. Tang, “An UML Profile Extension to Support Architecture Decision Traceability using Enterprise Architect,” <http://www.ict.swin.edu.au/personal/atang/ArelStereotypePackage.zip>, 2005.
- [154] —, “Managing project risks with architecture modelling,” *Australian Project Manager - Journal of the Australian Institute of Project Management*, vol. 15, no. 2, pp. 13–14, 2005.

- [155] —, “An AREL tool for traceability and validation,” <http://www.ict.swin.edu.au/personal/atang/AREL-Tool.zip>, 2006.
- [156] A. Tang, M. Babar, I. Gorton, and J. Han, “A Survey on Architecture Design Rationale,” Swinburne University of Technology, Tech. Rep. SUTICT-TR2005.02, 2005.
- [157] —, “A Survey of the Use and Documentation of Architecture Design Rationale,” in *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture*. U.S.A.: IEEE, 2006, pp. 89–98.
- [158] —, “A Survey of Architecture Design Rationale,” *Journal of Systems and Software*, 2006, doi:10.1016/j.jss.2006.04.029.
- [159] A. Tang and J. Han, “Architecture Rationalization: A Methodology for Architecture Verifiability, Traceability and Completeness,” in *Proceedings 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '05)*. U.S.A.: IEEE, 2005, pp. 135–144.
- [160] A. Tang, J. Han, and P. Chen, “A Comparative Analysis of Architecture Frameworks,” in *1st Asia-Pacific Workshop on Software Architectures and Component Technologies, APSEC 2004*, Korea, 2004.
- [161] A. Tang, Y. Jin, and J. Han, “A rationale-based architecture model for design traceability and reasoning,” *Journal of Systems and Software*, 2006, doi:10.1016/j.jss.2006.08.040.
- [162] A. Tang, Y. Jin, J. Han, and A. Nicholson, “Predicting Change Impact in Architecture Design with Bayesian Belief Networks,” in *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture*. U.S.A.: IEEE, 2006, pp. 67–76.
- [163] A. Tang, A. Nicholson, Y. Jin, and J. Han, “Using Bayesian Belief Networks for Change Impact Analysis in Architecture Design,” *Journal of Systems and Software*, 2006, doi:10.1016/j.jss.2006.04.004.
- [164] The Open Group, “The Open Group Architecture Framework (v8.1 enterprise edition),” <http://www.opengroup.org/architecture/togaf/#download>, 2003.
- [165] The Standish Group, “2004 Third Quarter Research Report,” [http : //www.standishgroup.com/sample_research/PDFpages/q3 – spotlight.pdf](http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf), 2004.
- [166] W. F. Tichy, “Should Computer Scientists Experiment More?” *Computer*, vol. 31, no. 5, pp. 32–40, 1998.

- [167] W. F. Tichy, N. Habermann, and L. Prechelt, “Summary of the Dagstuhl workshop on future directions in software engineering,” *SIGSOFT Software Engineering Notes*, vol. 18, no. 1, pp. 35–48, 1993.
- [168] S. Toulmin, *The Uses of Argument*, 2nd ed. Cambridge University Press, 2003.
- [169] J. Tyree, “Architectural design decisions session report.” in *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture*, 2005, pp. 285–286.
- [170] J. Tyree and A. Akerman, “Architecture Decisions: Demystifying Architecture,” *IEEE Software*, vol. 22, no. 2, pp. 19–27, 2005.
- [171] von Winterfeldt, Detlof and Edwards, Ward, *Decision Analysis and Behavioral Research*. Cambridge University Press, 1986.
- [172] R. J. Walker, L. C. Briand, D. Notkin, C. B. Seaman, and W. F. Tichy, “Panel: empirical validation: what, why, when, and how,” in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. Piscataway, NJ: IEEE Computer Society, 2003, pp. 721–722.
- [173] R. Watkins and M. Neal, “Why and how of requirements tracing.” *IEEE Software*, vol. 11, no. 4, pp. 104–106, 1994.
- [174] R. Yin, “The Abridged Version of Case Study Research,” in *Handbook of Applied Research Methods*. Sage Publications, 1998, ch. 8, pp. 229–259.
- [175] E. Yu, “Towards modelling and reasoning support for early-phase requirements engineering,” in *Proceedings 3rd IEEE International Symposium on Requirements Engineering (RE)*, Washington D.C., USA, 1997.
- [176] J. Zachman, “A framework for Information Architecture,” *IBM Systems Journal*, vol. 38, no. 2 & 3, 1987.
- [177] H. Zhang and S. Jarzabek, “A bayesian network approach to rational architectural design.” *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, pp. 695–718, 2005.

Part IV

Appendices

Appendix A

AREL Tool User Manual

1 Introduction

The capture and the representation of architecture design rationale has often been omitted by practising software architects. As part of the research on this subject, a model to support design rationale representation has been proposed. It is called the Architecture Rationale and Elements Linkage (AREL). This is a user manual of the AREL tool to support the AREL implementation.

This tool is part of a tool-set which comprises two other pieces of software. They are Enterprise Architect (version 5.00.767) and Netica (version 2.17). Enterprise Architect is an UML tool for capturing UML diagrams. Netica is a Bayesian Belief Networks (BBN) tool for capturing BBN diagrams and computing Bayesian probabilities.

The AREL tool supports five key functionalities:

- Perform AREL model checking for
 - Ensuring acyclic graph
 - Detecting erroneous AREL constructs
- Perform traceability of AREL models
- Export Enterprise Architect UML models to Netica
- Merge Enterprise Architect UML models with Netica Models
- Create and delete Architecture Design Decision Graphs (ADDG)

2 Installing the AREL Tool

To install the AREL tool, first download it from <http://www.ict.swin.edu.au/personal/atang/AREL-Tool.zip>. Unzip the installation files in the directory where you save the zip file. Run Setup.exe to install the tool into Program Files.

Please ensure that you follow the instructions provided in the README.txt file to ensure that you setup the Windows environment correctly.

3 Starting the AREL Tool

Once the tool has been installed, you can execute it like any other Windows programs. Double click the executable file (i.e EAPModelProjectUI) to start the program.

As soon as the software starts up following screen would appear (Figure 1).

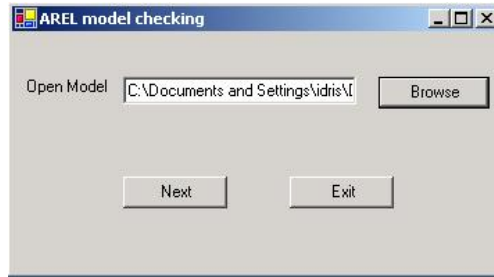


Figure 1 – Opening an Enterprise Architect model

To open an AREL Model enter path of desired Enterprise Architect repository file (i.e. EAP file) or click the **Browse** button to navigate to the file. Click **Next** when the file has been specified. Click **Exit** to exit the application.

4 AREL Tool Options

Upon opening an EAP file, the following screen (Figure 2) will be displayed. It has a number of buttons for supporting the AREL functionalities.

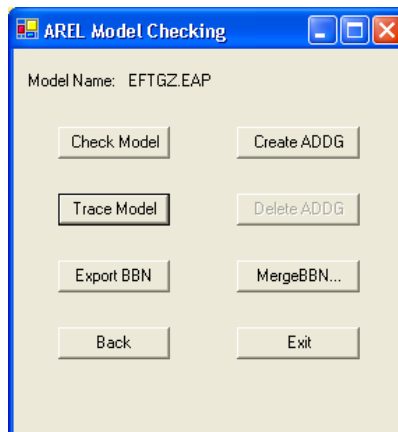


Figure 2 – AREL Tool Main Menu

4.1 AREL Check Model

When the **Check Model** button is clicked, the tool would progress to check the AREL model that is currently opened for any errors. The model checking is to verify the AREL rules:

- Every root elements must be an architecture element (AE)
- Every leaf elements must be an architecture element (AE)
- Every non leaf architectural element (AE) must be connected to another architectural rationale element (AR).
- No two same types of AREL elements must be connected. Therefore, no AR-AR or AE-AE connections are allowed.
- There are no acyclic graph presents in the model.

Some correct models are shown below.

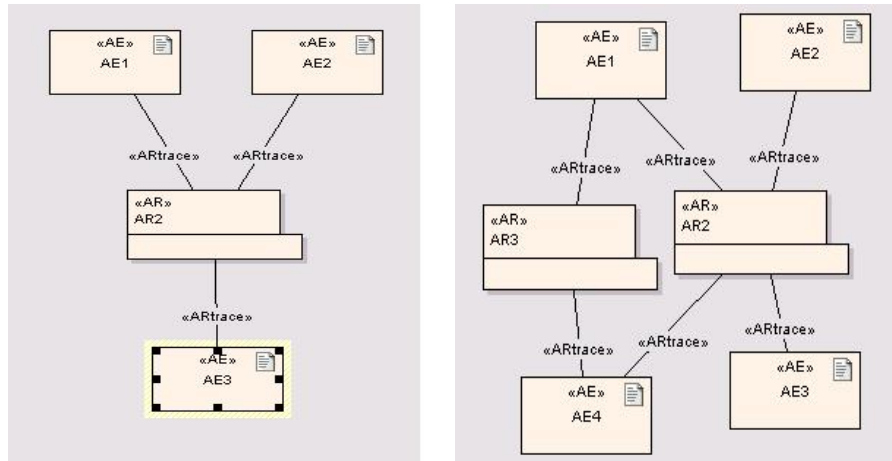


Figure 3a and 3b – examples of correct AREL models

Some incorrect models are shown bellow.

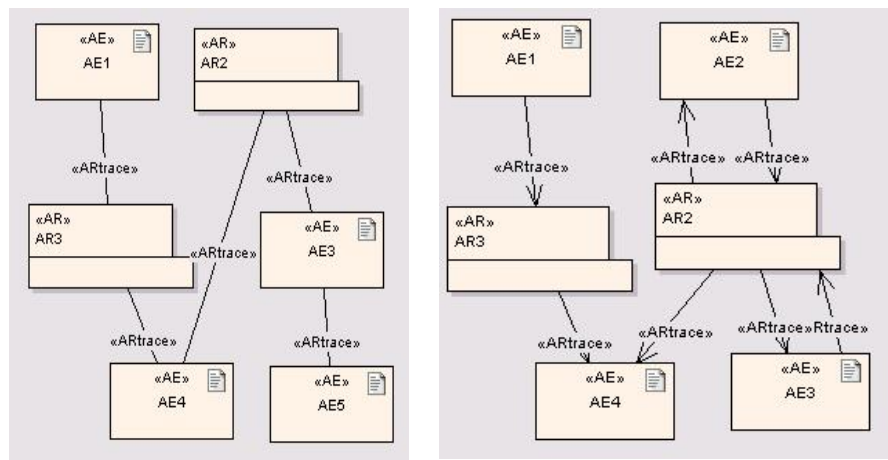


Figure 4a and 4b – examples of incorrect AREL models

When the model checking operation is complete, the following screen (Figure 5) is displayed to show if any errors or warnings have been detected. Errors indicate that

the AREL model is incorrect and has violated the rules described above. A warning indicates that there is an element within the EAP file that is not part of the AREL model. For instance, all stand-alone architecture elements are highlighted as warnings. When the user has finished with the error window, clicking the **OK** button would close this window and transfer control to the AREL Model Checking screen.

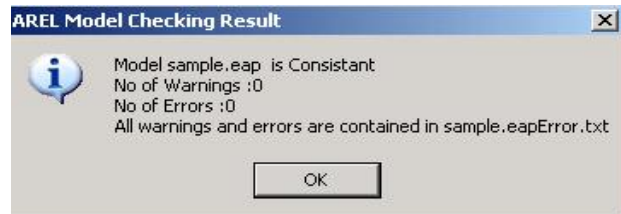


Figure 5 – AREL checking result screen

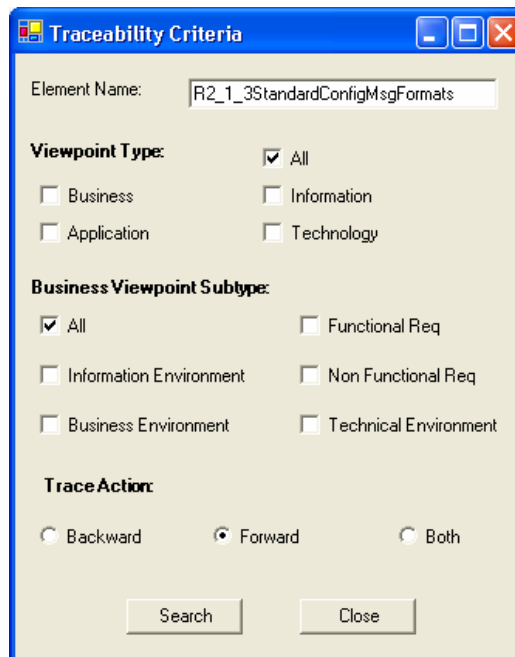
If there are any errors or warnings as shown above, they are logged in the same directory as the EAP file. The name of the log file is the same as the EAP file name with a “.eapError.txt” extension.

If there are any errors, the user **cannot** perform any other functions like **Trace Model**, **Create ADDG Graph**, **Export BBN graph** or **Merge BBN graph**.

4.2 AREL Model Tracing

AREL model tracing can be used to help architects retrieve specific parts of the AREL model for traceability purposes. When this option is selected, the AREL tool would first check that the AREL model is correct before the trace menu is displayed. A message is shown at the bottom of the menu window to indicate that the checking is in progress (i.e. The message is *Processing...*). A window showing a summary of the errors and warnings would be displayed. Users should click **OK** to close the window after viewing the information.

The model checking ensures that the trace would not end up in an infinite cycle should the AREL model had an error such as a cyclic model. If the model is consistent, then the traceability screen would be displayed (Figure 6) to specify trace criteria.



The screenshot shows a dialog box titled "Traceability Criteria". It has a standard Windows-style title bar with minimize, maximize, and close buttons. The main content area is light beige. At the top, there is a text box labeled "Element Name:" containing the text "R2_1_3StandardConfigMsgFormats". Below this, there are three sections of options:

- Viewpoint Type:** Includes a checked checkbox for "All", and unchecked checkboxes for "Business", "Information", "Application", and "Technology".
- Business Viewpoint Subtype:** Includes a checked checkbox for "All", and unchecked checkboxes for "Functional Req", "Information Environment", "Non Functional Req", "Business Environment", and "Technical Environment".
- Trace Action:** Includes radio buttons for "Backward", "Forward" (which is selected), and "Both".

At the bottom of the dialog, there are two buttons: "Search" and "Close".

Figure 6 – AREL trace specification screen

This screen is used for specifying the trace criteria. The user has to input a valid architecture element (AE) name. As shown in the diagram above, this name must be identical to what is stored in the EAP file.

Next, the user has to specify the Element Type and the Element Subtype. These two categories specify what types of AEs to be retrieved.

There are four different viewpoint types classified in AREL: Business, Information, Application and Technology. Each of them specifies a perspective of the architecture. There are five sub-viewpoints in the Business Viewpoint, they specify different types of drivers which motivate the creation of an architecture. These sub-

viewpoints are functional requirements, non-functional requirements, information system environment, business environment and technical environment.

Users may select all of the viewpoints or selected viewpoints. The AREL tool would retrieve only those elements which viewpoints have been included.

The trace actions dictate whether the traceability is to be forward tracing, backward tracing or both. In forward tracing, only those AEs and ARs which are downwards from the specified AE element are retrieved. In the backward tracing, only those AEs and ARs which are upwards from the specified AE are retrieved. When **Both** are specified, then all upwards and downwards AEs and ARs are retrieved.

On completion of the trace the following screen is displayed (Figure 7).

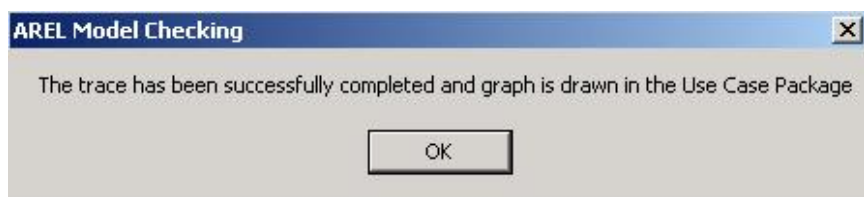


Figure 7 – message screen after tracing is complete

The result of the tracing is to create an UML graph which is stored in the same EAP repository as the AREL model. The trace results are located in the **Use Case Package** under **Use Case Model** with the name of **Traceability Graph**.

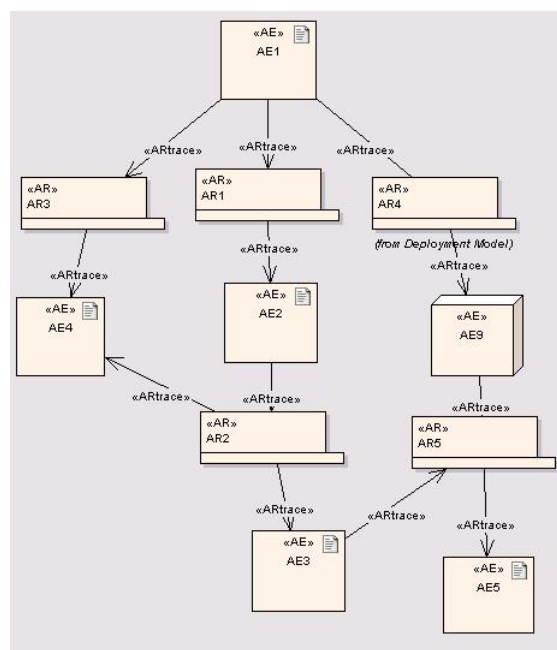


Figure 8 – An example of an AREL model

If Figure 8 is the original AREL model, a downward trace of AE2 would result in a graph such as the one in Figure 9. An upward trace of AE2 would result in a graph such as the one in Figure 10. Traceability of both directions would result in a graph such as the one in Figure 11.

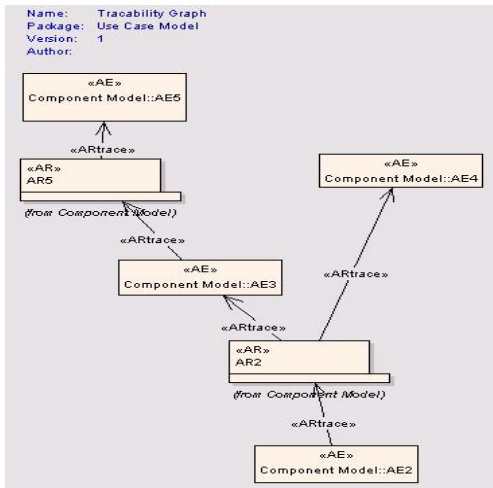


Figure 9 – result of downward trace

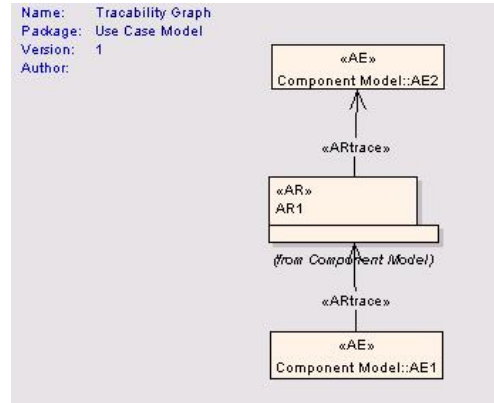


Figure 10 – result of upward trace

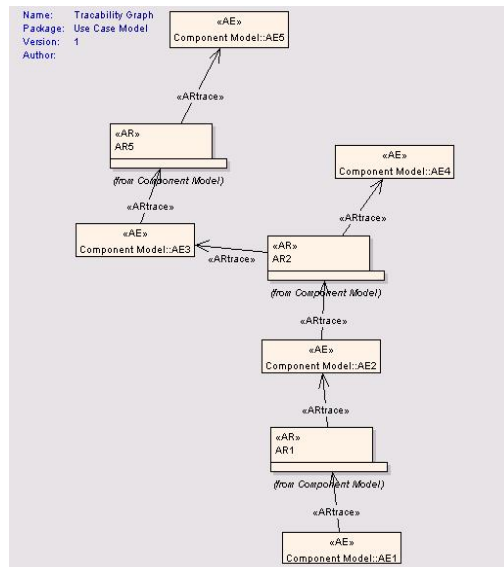


Figure 11 – result of both upward and downward trace

4.3 Exporting Graph to Netica

The AREL tool supports the integration between Enterprise Architect and Netica. This is done by using the AREL tool extracting the UML model and then transforming the model in a format which the Netica software can read.

Since the AREL model is an acyclic model, it can be imported by Netica. When the **Export BBN** button is clicked, the AREL tool would first check that there are no errors in the model. It will then create the Netica file with file extension **dne**. The name of the file would be the name of the UML model plus the date and time when the dne file was generated. For instance, an Enterprise Architect model called *ABC* would have the filename *ABC (20 July 2006(12-37 PM)).dne*.

Once this file is generated, Netica can open it directly. A user would enhance the data in this file. The following data entry and graphics manipulation are probably necessary:

- Reorganise the graph and position the AEs and ARs for better visibility
- Assign prior and conditional probabilities to the nodes

4.4 Merging Enterprise Architect Graph with Netica

After exporting the AREL model from the Enterprise Architect repository (in UML) into the Netica format, it is possible for the models in both notations to change. The design in the UML model might change because of design changes. For the Netica model, the probabilities and the positioning of the Netica model elements might be enhanced. As such, there is a need to merge the two models together to synchronise the changes. This can be achieved by using the **Merge BBN** button.

The end result is an updated Netica file which has all the elements and relationships of the UML model, but preserves the positioning and the probabilities of the Netica model. The basic assumption of this function is that the elements and the relationships in the AREL model are defined by the Enterprise Architect repository. So if there are additions, deletions or modifications in the repository, it would be reflected in the merged Netica form.

This function will prompt the user to enter the Netica file name which is to be merged with the UML model, this is shown in Figure 12.

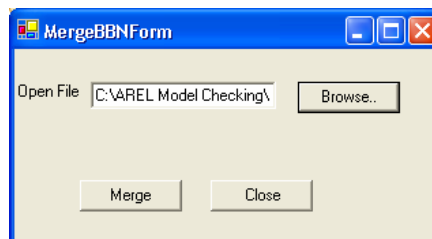


Figure 12 – Specify the Netica file to merge with

When the filename is specified, user can use the **Merge** button to start the merging process. The end result will be stored in a file called *ABC merge (20 July 2006(3-11 PM))*.

4.5 Create ADDG Graph

Based on the AREL model which consists of AEs and ARs, an Architecture Design Decision Graph (ADDG) can be created. This is an UML graph created in the Enterprise Architect which only shows the relationships between the ARs and filter out all the AEs in the graph. The purpose of this tool is to highlight the relationships between architecture decisions.

To initiate the graph's creation, click on the **Create ADDG** button. When the ADDG graph is successfully created in the Enterprise Architect repository, the following pop-up window will be displayed.

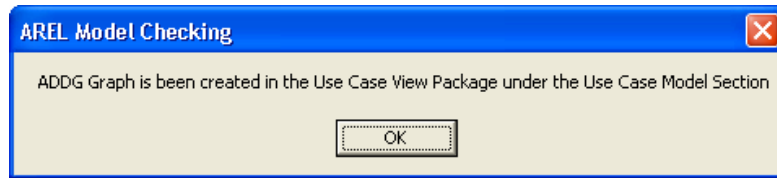


Figure 13 – confirmation of ADDG creation

The ADDG graph is created in the same repository under Use Case View, Use Case Model and ADDG Graph.

4.6 Delete ADDG Graph

Since there can only be one ADDG graph at one time in the repository, the AREL tool provides the facility for an user to remove the graph. This can be done by using the **Delete ADDG** button.

4.7 Use another Enterprise Architect Repository

If the user wants to specify another Enterprise Architect repository, one could use the **Back** button to navigate to the open model screen (see Figure 1) to open another UML model.

4.8 Exit the AREL tool

This button ends the execution of the AREL tool.

Appendix B

Creating Stereotype Package in Enterprise Architect

This section describes the step to create the stereotype package that can be used in Enterprise Architect version 5.00.767.

Step	Procedure	Comments and Questions
	<i>Creating Stereotypes</i>	Note: You do not need to carry out steps 1 – 8 because they have been performed and the results are contained in the XML file
1	Drag <i>Profile</i> icon from the <i>Toolbox</i> menu onto a diagram to create a profile package called StereotypePackage .	I specified UML2.0 in <i>Workspace Perspective</i>
2	Open StereotypePackage	
3	Drag <i>Stereotype</i> from the <i>Toolbox</i> menu to create stereotype class called EA	All default values are used except changing the class name to EA
4	Highlight EA and press F9 to create attributes	Attributes created are Version and ElementID
5	Drag <i>Metaclass</i> from <i>Toolbox</i> onto the <i>Profile</i> diagram and choose types from the dialogue.	Chosen types are class, objects, package etc. See figure 1
6	Use the <i>Extension</i> to point from EA to the <i>Metaclasses</i>	
7	Save the project	
8	From the <i>Project View</i> , right-click on <<profile>> StereotypePackage , do <i>Save Package as UML profile</i>	Name of profile is StereotypePackage .
	<i>Importing AREL Stereotypes into Enterprise Architect</i>	
9	Select Resource View	
10	Select UML Profiles	
11	Right-click to import UML profiles	Select the exported UML profile StereotypePackage . Package appears under UML Profiles.
12	Right-click StereotypePackage to select <i>Show Profile in UML Toolbox</i>	StereotypePackage appears in the <i>Toolbox</i> menu.
	<i>Using the Stereotype to support AREL</i>	
13	Drag a stereotype EA from Resource View	

	onto the diagram	
14	Assign values to tags according to stereotype <<AE >> or <<AR >>	
15	Click Stereotype <<ARtrace >> and connect <<AE >> node to <<AR >> node	

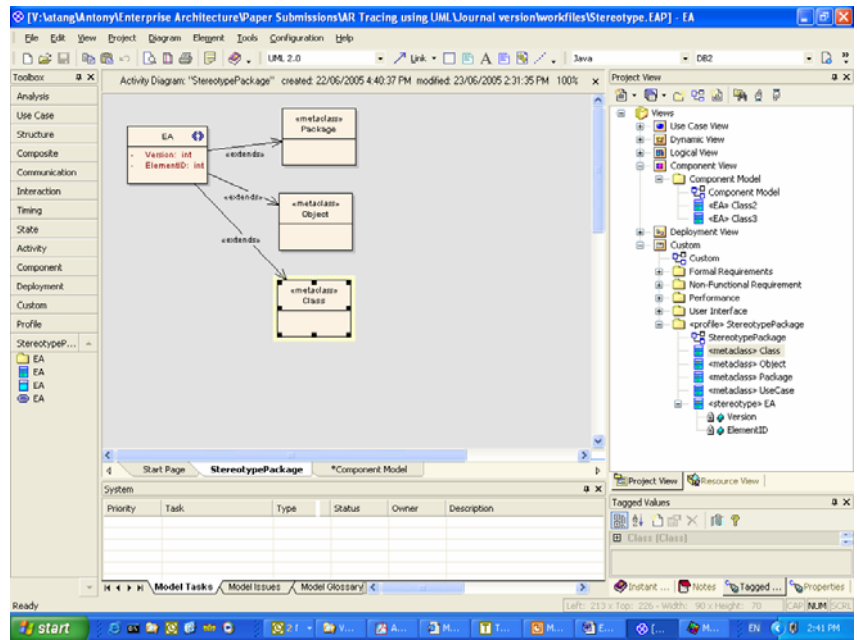


Figure 1 – Defining Stereotypes

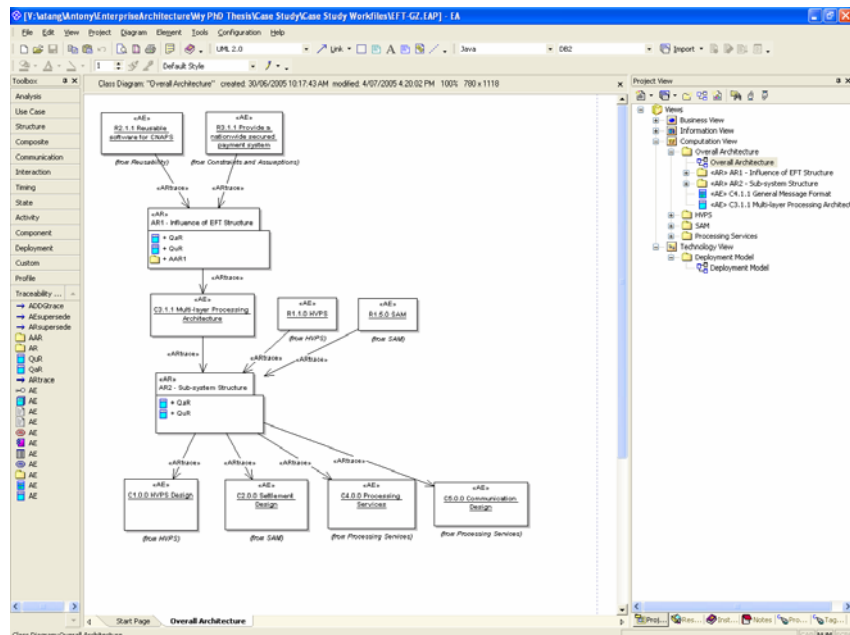


Figure 2 – Using Extended Profiles in AREL

Appendix C

A Survey Questionnaire on Architecture Rationale

The following is the questionnaire used in the architecture design rationale survey. It has been implemented on a web site using three static web pages.

An Investigation into Architecture Design Rationale (A Joint Research Project by Swinburne University of Technology and NICTA)

Disclosure Statement

The purpose of this survey is to study the practices of the architecture design rationale. In order to collect the appropriate data for the research, the participants should be system and software architects and designers with at least 3 years of experience. This survey would provide insights on what industry practitioners see as important in reasoning about architecture decisions. The data will enable us to verify the theory on architecture rationalisation. This survey contains 42 questions, mostly multiple choice, and will take 30 minutes to complete.

Your participation in this research is not anonymous due to the recording of your e-mail address. This survey will also collect your personal information towards the end. They include your e-mail address, gender, city of residence and information relating to your position. You may opt not to answer these questions. Should you choose to receive the final research report, I will e-mail you the research report when all the statistical information are analysed and conclusions are drawn. All data collected in this survey will be kept for 5 years in accordance with the Australian government regulations.

Thank you for your support.

Privacy Protection

Only investigators listed below will have access to your responses. Your e-mail address is collected for the sole purpose of sending you the final research report. Your e-mail address will be kept confidential and will not be disclosed. We will not use your e-mail address in any way other than sending you the research report. Information collected in this survey, except your e-mail address, may be disclosed in research reports and will be used for research purposes only.

Useful Definitions

The following are some useful definitions that are used consistently in the survey and will help you answer the questions in the survey:

- Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution (IEEE Std 1471-2000).
- Architecture design is the process of constructing the architecture of a system.
- Architecture rationales are reasons to justify an architecture design.
- Architecture rationalisation is the process of reasoning about architecture design decisions.

How to fill in the questionnaire

- Please answer all questions. You may skip optional questions if they are not applicable to you.
- Please complete this questionnaire in a single attempt. This tool does not have the intelligence to recognise individual users to allow subsequent updates.
- Please do not use the browser BACK button because it could confuse the application server. If you like to revisit the definitions during the survey, just start another window to view this page again.
- You may withdraw from the survey by scrolling to the end of the questionnaire and check the withdrawal checkbox.
- According to university policies, you are advised to print a copy of the complaint form after you complete the questionnaire.

Investigators

This research is carried out jointly by the Centre of Component Software and Enterprise Systems, Swinburne University of Technology and National ICT Australia. The following are the investigators:

- Antony Tang, PhD Candidate, Faculty of ICT, Swinburne University of Technology. E-mail: atang@it.swin.edu.au
- Ian Gorton, Senior Principal Researcher, Empirical Software Engineering Group, NICTA. E-mail: Ian.Gorton@nicta.com.au
- Muhammad Ali Babar, Research Scientist, Empirical Software Engineering Group, NICTA. E-mail: [Muhammad.Alibabar@nicta.com.au](mailto:Mohammad.Alibabar@nicta.com.au)
- Jun Han, Professor, Faculty of ICT, Swinburne University of Technology. E-mail: jhan@it.swin.edu.au

Consent Form

If you agree to participate in this research, please tick the following boxes and press **Start** on the web page. Enjoy the questionnaire.

- I agree to participate in this activity, realising that I may withdraw at any time. I agree that the research data collected for this study may be published or used by the investigators for research purposes.
- I would like to receive a copy of the research report when it becomes available.

[Continue]

Questions

1. As a designer/architect, the following are my job's primary tasks. (Tick any task if you spend at least 10% of your time on that task in a project)
 - a) Project Management Tasks
 - b) IT Planning or Proposal Preparation
 - c) Requirements Analysis or Tender Analysis
 - d) Overall Design of System
 - e) Software Design and Specification
 - f) Data Modelling
 - g) Program Design and Specification
 - h) Test Planning and Design
 - i) Design of Non-functional Requirements (security, performance, interoperability, flexibility, standards, usability etc.)
 - j) Implementation Design (capacity planning, system environment, platforms etc.)
 - k) Training
2. The role of software architect is formally recognised in my organisation for: (Please tick the appropriate choice)
 - a) All projects across the organisation
 - b) Some projects only
 - c) Not at all
3. If your answer to the last question is for **some projects only**, the criteria that dictate whether the project needs an architect are: (Please tick all appropriate choices)
 - a) New systems
 - b) Mission or business critical systems
 - c) Systems which are considered high-risk
 - d) Systems which are over certain budget
 - e) Other criteria, please specify: _____(text 256 char)
4. The organisation that I work with carries out software architecture reviews by architects external to the project for : (Please tick the appropriate choice)
 - a) All projects across the organisation

- b) Some projects only
 - c) Not at all
5. If your answer to the last question is for **some projects only**, the criteria that dictate whether the project requires external architect review are: (Please tick all appropriate choices)
- a) New systems
 - b) Mission or business critical systems
 - c) Systems which are considered high-risk
 - d) Systems which are over certain budget
 - e) Other criteria, please specify: _____ (text 256 char)

6. I consider the appropriateness of alternative architecture designs during the design process before I make a decision (Note: an alternative design is a design that you have considered.): (Frequency of occurrence)
7. I document discarded alternative designs : (Frequency of occurrence)
8. When making architecture design decisions, the importance of each of the following design rationales play in my decision making process is : (Note: design rationales are reasons to justify the design.) (Level of Importance)
- a) Design constraints
 - b) Design assumptions
 - c) Weakness of a design
 - d) Cost of a design
 - e) Benefit of a design
 - f) Complexity of a design
 - g) Am I certain that this design would work
 - h) Am I certain that I or the team could implement it
 - i) Tradeoffs between design alternatives
9. This is an optional question. The other design rationales I also consider but are not listed above are _____
text(256)
10. I use the following design rationales to reason about my architecture design: (Frequency of occurrence)
- a) Design constraints
 - b) Design Assumptions
 - c) Weakness of design
 - d) Cost of design
 - e) Benefit of design
 - f) Complexity of design
 - g) Certainty that design would work
 - h) Certainty that you could implement it

i) Tradeoffs between alternatives

11. I document these types of architecture design rationales: (Frequency of occurrence)

- a) Design constraints
- b) Design Assumptions
- c) Weakness of design
- d) Cost of design
- e) Benefit of design
- f) Complexity of design
- g) Certainty that design would work
- h) Certainty that you could implement it
- i) Tradeoffs between alternatives

12. On an overall scale, the level of documentation of architecture design rationales that I do is: (Level of Documentation)

13. If I do not document architecture design rationale, the reasons are as follows. (Please tick applicable reasons.)

- a) There are no standards or requirements in the project or organisation to do so
- b) I am not aware of the need to document it
- c) Documenting design rationale is not useful
- d) Time / budget constraints
- e) Absence of appropriate tools for documenting design rationale
- f) Other reasons, please specify : _____(text 256 chars)

14. This is an optional question. If and when I document architecture design rationale, these are the tools, procedures or methods that I use
_____(text 256 chars)

15. When I design a system or software architecture, I reason about why I make certain design choices. (Frequency of occurrence)

16. I think it is important to use design rationales to justify design choices.. (Level of agreement)

17. In my education or my professional training, I have been trained to make use of design rationales explicitly to justify design choices. (Level of Training)

18. I sometimes design architectures to enhance existing systems. Yes / No (if No, go to question 28)

19. I revisit architecture design documents and design specifications to help me understand the design of the system for making system enhancements. (Frequency of occurrence)
 20. Design rationales of existing systems are important to help me understand previous designs and assess my options in system enhancements and integration. (Level of Agreement)
 21. I forget the reasons that justify my designs after a period of time. (Level of forgetfulness)
 22. If I am not the designer of an architecture, I may not know why existing designs are created in a certain way without documented design rationale or someone who can explain the design. (Level of Agreement)
 23. I do architectural impact analysis during system enhancements and integration to assess how new changes might affect the existing system. (Note: architectural impact analysis is used to analyse the extent and impact of changes to the structure of the system.) (Frequency of occurrence)
 24. The following items are important when I carry out architectural impact analysis. (Level of Agreement)
 - a) Analyse and Trace Requirements
 - b) Analyse Specifications of Previous Design
 - c) Analyse Design Rationale of Previous Design
 - d) Analyse Feasibility of Implementation
 - e) Analyse Violation of Constraints or Assumptions of Previous Design
 - f) Analyse Scenarios
 - g) Analyse Cost of Implementation
 - h) Analyse Risk of Implementation
 25. This is an optional input. Other additional steps that I will take when carrying out impact analysis are
_____ text(256)
-

26. When I design, I am relatively certain (i.e. I consider the risk factor) that the resulting design will work and I or my team are capable of implementing it. (Level of Certainty)
27. I explicitly quantify the risk of implementation when I design. (Frequency of occurrence)
28. Different potential architecture designs have different degrees of uncertainty, or risk, to achieve the desired business outcomes. (Level of Agreement)

29. There might be different degrees of uncertainty, or risk, in implementation depending on the capability of the design/development team. (Note: The capability refers to the experience and knowledge of particular technology used in the implementation.) (Level of Agreement)
30. The level of risk of an architecture design, i.e. before detailed design and implementation, that I consider acceptable for an important project is: (Level of Risk Scale)
-

31. My E-mail address : _____ (text 128)
32. Sex : Male (M) / Female (F)
33. City of residence : _____ (text 40)
34. No of years I have been in the IT industry : XX
35. No of years I have been a designer / architect : XX
36. No of years I have been with the current (or last) organisation : XX
37. My current (or last) job title : _____ (text 40)
38. I have used at least one Software Development Methodology or Standard in a project in the past. Yes / No.
39. The number of co-workers in my current or last project team, including project managers, architects, designers, programmers and testers, is: XXXX
40. I have obtained a tertiary qualification in an IT related field such as Computer Science, Information Technology etc. Yes / No
41. For your privacy protection, you may withdraw from this survey any time. If you have decided not to participate in this survey anymore, you can withdraw by checking the following box. <Withdraw participation>

< END OF SURVEY >

We will send you the research report if you have chosen to receive it. Should you have any queries, please do not hesitate to contact us. Thank you for your participation.

- Antony Tang (atang@it.swin.edu.au)
- Ian Gorton (Ian.Gorton@nicta.com.au)
- Muhammad Ali Babar (Muhammad.Alibabar@nicta.com.au)
- Jun Han (jhan@it.swin.edu.au)

Complaint Procedure

If you have any complaint about this survey and the way it is conducted, please contact:

The Chair
Human Research Ethics Committee
Swinburne University of Technology
PO Box 218
Hawthorn Vic 3122
Phone: +613 92145223

Please make a copy of this page for your reference.

Appendix D

A Comparison of Design Specifications and Architecture Rationale and Elements Linkage (AREL) Using Expert Opinions

Project Title

**A Comparison of Design Specifications and Architecture Rationale and Elements Linkage (AREL) Using Expert Opinions
(A Research Project by Swinburne University of Technology)**

Investigators

This research is carried out by Antony Tang (E-mail: atang@it.swin.edu.au) and Prof. Jun Han (jhan@ict.swin.edu.au), Swinburne University of Technology.

Disclosure Statement

The purpose of this study is to investigate the effectiveness of the AREL model representation of design rationale in comparison with traditional design specifications.

This empirical study is carried out by way of interviewing experts. It is expected that the interview session would take up to 90 minutes. The interview questions presented in this questionnaire may require referencing the Electronic Fund Transfer System (EFT) design specifications and the AREL model. Participants are invited to use them to answer the questions as if engaging in a real design session.

Although participants are asked to provide answers, the answers could be sourced from past memory of the design, reconstructing design reasoning, deducing or the documentation supplied. Time would be kept to compare the effectiveness of different documentation formats, *time 1* for using design specifications and *time 2* for using the AREL model. The accuracies and completeness of the answers are also compared qualitatively.

Your participation in this research is not anonymous due to the recording of your e-mail address. This survey will collect some personal information. They include your e-mail address, gender and information relating to your position. You may opt not to answer these questions. All data collected in this survey will be kept for 5 years in accordance with the Australian government regulations.

Thank you for your support.

Privacy Protection

Only investigators listed above will have access to your responses. Your e-mail address is collected if any further correspondence or clarifications are required. Your e-mail address will be kept confidential and it will not be disclosed. Information collected in this survey, except your e-mail address, may be disclosed in research reports and will be used for research purposes only.

Who are the Participants

The participants are experts who know how to design electronic payment systems, their participation is by invitation from the investigators. It is expected that about seven experts will be invited to participate.

Information provided to experts during the interview

- The EFT System Design – System Architecture Overview
- The EFT System - Network Architecture
- An introduction to AREL modelling
- An AREL model of the EFT System (selected design)

Questions

A. About the participant.

1. E-mail address : _____
2. Gender : Male (M) / Female (F)
3. No of years I have been in the IT industry : _____
4. No of years I have been a designer / architect : _____
5. No of years I have spent designing / developing electronic payment systems :

B. Exploring design reasoning.

1. What are the key fault resilient features supported by the EFT system? Why do we need it?

Time 1 : _____
Answer 1 :

Document Deduction
Memory Guess

Time 2 : _____
Additional Answer 2 :

2. Why are there three layers of software (application, messaging and communication) in the design? Do you think the reasons are important to the architecture design?

Time 1 : _____
Answer 1 :

Document Deduction
Memory Guess

Time 2 : _____
Additional Answer 2 :

3. Why does the system use asynchronous messaging and what are the implications of such design?

Time 1 : _____
Answer 1 :

Document Deduction
Memory Guess

Time 2 : _____
Additional Answer 2 :

4. Why does a MCP process handle one bank connection only? Can I change it to handle multiple bank connections simultaneously? What are the implications if I do?

Time 1 : _____
Answer 1 :

Document Deduction
Memory Guess

Time 2 : _____
Additional Answer 2 :

C. General Comments on AREL Modelling.

1. Do you think that the AREL model is useful to help you reason with the EFT design?

(Not Helpful) 1 2 3 4 5 (Very Helpful)

2. Given you have reasonable project schedule and resources, would you capture design rationale with AREL?

Yes / Possibly / Don't Know / No

3. What are your comments and observations of AREL (both positive and negative feedbacks)?

4. Is there anything in AREL which can be improved?

< END OF QUESTIONNAIRE >

List of related publications

M. Ali Babar, A. Tang, I. Gorton and J. Han. Industrial Perspective on the Usefulness of Design Rationale for Software Maintenance: A Survey. *Quality Software International Conference 2006 (QSIC 2006)*, pp. 201-208. IEEE Computer Society Press.

A. Tang, Y. Jin and J. Han. A Rationale-based Architecture Model for Design Traceability and Reasoning. *Journal of Systems and Software*, accepted for publication on 21 August 2006, 17 pages. Elsevier.

A. Tang, M. Ali Babar, I. Gorton and J. Han. A Survey of Architecture Design Rationale. *Journal of Systems and Software*, vol. 79, no.12, pp. 1792-1804. Elsevier.

A. Tang, A. Nicholson, Y. Jin and J. Han. Using Bayesian Belief Networks for Change Impact Analysis in Architecture Design. *Journal of Systems and Software*, vol. 80, no.1, pp. 127-148. Elsevier.

A. Tang, Y. Jin, J. Han and A. Nicholson. Predicting Change Impact in Architecture Design with Bayesian Belief Networks. In *Proceedings of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 5)*, pp. 67-76. IEEE Computer Society Press.

A. Tang, M. Ali Babar, I. Gorton and J. Han. A Survey of the Use and Documentation of Architecture Design Rationale. In *Proceedings of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 5)*, pp. 89-98. IEEE Computer Society Press.

A. Tang and J. Han. A Methodology for Architecture Verifiability, Traceability and Completeness. In *Proceedings of the 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS05)*, pp. 135-144. IEEE Computer Society Press.

A. Tang and J. Han. A Comparative Analysis of Architecture Frameworks. In *Proceedings of the 2004 Asia-Pacific Software Engineering Conference*, pp. 640-647.

IEEE Computer Society Press.

A. Tang, Managing project risks with architecture modelling. In *Australian Project Manager - Journal of the Australian Institute of Project Management*, June 2005, Vol 25(2). pp. 13-14. Australian Institute of Project Management.