Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 918–934
The Journal of Systems and Software
www.elsevier.com/locate/jss

A rationale-based architecture model for design traceability and reasoning

Antony Tang *, Yan Jin, Jun Han

Swinburne University of Technology, Faculty of ICT, Melbourne Vic 3122, Australia

Received 19 October 2005; received in revised form 4 August 2006; accepted 19 August 2006
Available online 27 October 2006

Abstract

Large systems often have a long life-span and comprise many intricately related elements. The verification and maintenance of these systems require a good understanding of their architecture design. Design rationale can support such understanding but it is often undocumented or unstructured. The absence of design rationale makes it much more difficult to detect inconsistencies, omissions and conflicts in an architecture design. We address these issues by introducing a rationale-based architecture model that incorporates design rationale, design objects and their relationships. This model provides reasoning support to explain why design objects exist and what assumptions and constraints they depend on. Based on this model, we apply traceability techniques for change impact analysis and root-cause analysis, thereby allowing software architects to better understand and reason about an architecture design. In order to align closely with industry practices, we choose to represent the rationale-based architecture model in UML. We have implemented a tool-set to support the capture and the automated tracing of the model. As a case study, we apply this approach to an real-world electronic payment system.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Design rationale; Architecture design; Traceability

1. Introduction

Highly complex and integrated systems are typical of today's large system environment. Such systems usually involve system and software architectures that are intricate and highly inter-dependent. They also require continuous maintenance, enhancements and integration. Architects and designers who are not the original developers often have to quality-control and maintain the system. These people require a good understanding of the architecture in order to carry out their work effectively. Generally in the software industry, design rationale cannot be obtained from design specifications because there is no systematic practice to capture them. Even when some design rationale are captured, they are not organised in such a way that they can be retrieved and tracked easily. Remedying this situation becomes critical and challenging when system requirements and operating environments continue to evolve.

Design rationale capture the reasons behind design decisions. They show how the system design satisfies the requirements, why certain design choices are selected over alternatives and how environmental conditions influence the system architecture. During the design process, decisions are made and justified but the justifications are often unrecorded and are lost over time (Bosch, 2004; Perry and Wolf, 1992; Tyree and Akerman, 2005; Parnas and Clements, 1985). When asked to explain the design, designers either have to remember the rationale or reconstruct them (Gruber and Russell, 1996).

In a recent survey on architecture design rationale (Tang et al., 2006a), we found that 85.1% of architects agreed that the use of design rationale is important in justifying design and 80% of the respondents said they fail to understand the reasons of a design decision without design rationale support. Furthermore, 74.2% of respondents forget their own design decisions half the time or more often. These

* Corresponding author. Tel.: +61 3 92145198; fax: +61 3 98190823.
E-mail addresses: atang@ict.swin.edu.au (A. Tang), yjin@ict.swin.edu.au (Y. Jin), jhan@ict.swin.edu.au (J. Han).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.08.040

results indicate the need to capture the design rationale for system maintenance. The erosion of architecture design rationale can result in ill-informed decisions because the original design reasoning was missing. As a result, it may lead to inconsistent design and violations of design constraints. The impacts can be serious because architecture design is fundamental to a system. Consequently, the rectification of errors can be very costly.

System and software architecture design often involves many implicit assumptions (Roeller et al., 2005) and convoluted decisions that cut across different parts of the system (Nuseibeh, 2004). 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.
- 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 (Lee, 1997; Conklin and Begeman, 1988; McCall, 1991) but they do not support effective design rationale retrieval and communication (Shipman and McCall, 1997). An effective design reasoning model should enable the capture of design rationale and the clear explanation of design objects. The three major challenges in this exercise are (a) to identify the information required for design reasoning; (b) to create a model to retain design rationale and their relationships with design objects; and (c) to use the model to support effective traceability of design reasoning to help understand an architecture design.

In view of these objectives, we introduce the rationale-based model, Architecture Rationale and Elements Linkage (AREL), to support design rationale capture and traversal. We capture architecture design rationale qualitatively and quantitatively. Qualitative design rationale provides the arguments for and against a design alternative whilst quantitative design rationale uses cost, benefit and risk to quantify the merits of a design alternative. Using architecture design rationale as connectors, we relate requirements, constraints and assumptions to design objects to give an explanation of an architecture design. The AREL model uses a viewpoint-based architecture framework to structure the architecture design. The viewpoints represent different perspectives of an architecture such as business, information and software design. Such classifications provide a structure for the tracing and traversal of architecture elements.

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.

The remainder of this article is organised as follows. We discuss the related work in the areas of design rationale and requirements traceability in Section 2. We define the AREL model in Section 3. In Section 4, we discuss the tracing capabilities of the AREL model. In Section 5, we present a case study about a real-world electronic payment system. AREL uses the UML graphical notation for its representation. Its implementation, described in Section 6, requires a tool-set which is comprised of Enterprise Architect and our customised tool.

2. Background

In this section, we examine the needs for a better design rationale representation to support design reasoning. We analyse existing design-rationale methods and traceability methods to identify the challenges. From the perspective of a practicing architect, we highlight the importance of having traceability and design rationale through a number of use cases.

2.1. 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 (Perry and Wolf, 1992) 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 can be complex and prone to erroneous interpretations without a first-class representation of design rationale (Bosch, 2004). When changes are introduced, the architecture design could be violated and the cost of change could be very high and even prohibitive.

Design rationale is important in many ways. It retains design knowledge such as design assumptions, constraints and design reasoning that are often not captured. It captures design alternatives to help understand why some designs have been rejected. It helps architects to understand a design since design decisions are often inter-twined and cut across a number of issues. A change to a decision may trigger a series of ripple effects to the other parts of the system (Han, 1997). Therefore, when there is a system defect or a need for design modification, design rationale can help maintainers analyse root causes and diagnose change impacts.

There are different approaches to representing design rationale. The argumentation-based design rationale representation uses nodes and links to represent knowledge and relationships. It dates back to Toulmin's argumentation representation (Toulmin, 1958). Since Toulmin, many similar argumentation-based approaches such as Issue-Based Information System (IBIS) (Kunz and Rittel, 1970) and Design Rationale Language (DRL) (Lee, 1991) have been developed. They fundamentally show the issue, the argument and the resolution of design argumentation. Shipman III and McCall argued that neither the argumentation nor the communication perspective of argumentation-based design rationale has been generally successful in practice (Shipman and McCall, 1997). From the argumentation perspective, the problem has been in the ineffectiveness of capturing rationale because of the extensive documentation. From the communication perspective, design rationale cannot explain the reasoning for design objects effectively.

There are three key issues that impede the application of 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 (Gruber and Russell, 1996). A second issue of the argumentation-based approach is that the design objects being discussed do not appear in the representation itself and are not linked to it in a defined way (Potts, 1996). 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 capturing design rationale is to use template-based methodologies. These methods make use of standard templates which are 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 software industry. Examples of this approach are the method proposed by Tyree and Akerman (Tyree and Akerman, 2005) and Views and Beyond (Clements et al., 2002). However, neither of the methods explicitly models the relationships between design rationale and design objects. As such, it is difficult to trace through a series of inter-related design objects which are affected by a design change.

Savolainen and Kuusela (Savolainen and Kuusela, 2002) proposed an approach for structuring goals and decisions. The Design Decision Tree (DDT) provides a means to connect requirements to architecture decision and design elements. Their framework is similar to this work but there is little detail on the composition of design rationale and how the framework may support design rationale traceability.

2.2. Requirements and design traceability

Requirements traceability is the ability to describe and follow the life of requirements, in both a forward and backward direction. Gotel and Finkelstein (1994) 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. A survey of a number of systems by Ramesh and Jarke (Ramesh and Jarke, 2001) indicates that requirements, design and implementation ought to be traceable. It is noted by Han (Han, 2001) 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 (IEEE, 1996; IEEE, 1997).

During 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 (Spanoudakis et al., 2004). 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 TOOR (Pinheiro, 2000; Pinheiro and Goguen, 1996), DOORS

(Smith, 1998), Ramesh and Jarke (2001), Hughes and Martin (1998) and Egyed (2001).

Some traceability approaches incorporate the use of design rationale in a limited way. Haumer et al. (1999) 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 (2001) 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 are not considered.

An implicit assumption of these traceability methods is that they can provide the explanatory power to help designers understand the system. However, design reasoning still needs to be reconstructed even though source code or design objects can be traced back to requirements. Besides, assumptions and constraints that are implicitly stated are not traceable by such methods.

2.3. Why have traceable design rationale?

Being able to trace design and requirements to design rationale helps architects to understand, verify and maintain architecture design (Watkins and Neal, 1994). It supports the conscious reasoning of architecture design. There are many use-cases of traceable architecture design rationale in the software development life-cycle (Kruchten et al., 2005). 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, they require 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 software. If the rationale of the compromise is not explicitly stated, an update to the security software 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 effective architecture model and 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 (Herbsleb and Kuwana, 1993). 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 introducing a design rationale-based architecture model and associated methods to support traceability.

3. An architecture model for design rationale capture and tracing

Design *reason* may have many meanings and interpretations. In this section, we first discuss two types of design reasoning and their relevance to design decisions, and introduce a conceptual model for representing design reasoning. Using the conceptual model as a basis, we then implement the Architecture Rationale and Elements Linkage (AREL) for architecture design rationale modelling. The AREL model captures the relationship between two entities: Architecture Rationale (AR) and Architecture Elements (AE).

3.1. 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 (Simon, 1981). 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 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. 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 Roeller et al. (2005), in order to have a deep understanding of software systems, undocumented assumptions have to be re-discovered. Garlan et al. (1995) 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 (Roeller et al., 2005), 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 rational, and design objects, we present a conceptual model for design process in Fig. 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:

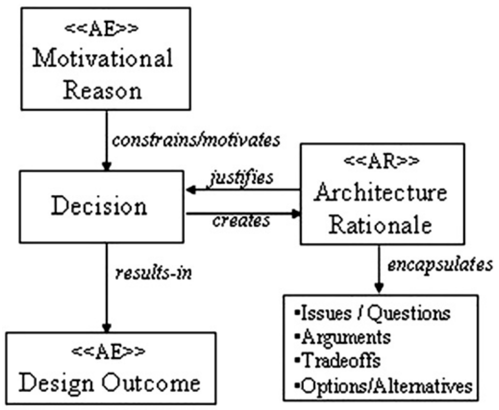


Fig. 1. An architecture rationale conceptual model.

- 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.

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.

A decision *creates* and is *justified* by the architecture design rationale. 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 different 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.

3.2. Architecture rationale and elements linkage

The AREL model is an implementation of the conceptual model using the UML notation. AREL is an acyclic graph which relates architecture elements AEs to architecture rationale ARs using directional link ARtrace.[1] An AE is an architecture element which participates in a decision as an input (i.e. motivational reason) or an outcome (i.e. design outcome). An AR encapsulates the architecture design justification of a decision. Since AR has a one-to-one relationship to justify a decision, AR can therefore represent a decision point in AREL modelling. In this section, we focus on the relationships between AEs and ARs. In the

[1] A detailed discussion on the acyclic nature of the graph is contained in Tang et al. (2006b).

next two sections, we discuss in detail what constitute an AE and an AR, respectively.

The relationships between AEs and ARs are represented by the UML stereotyped association ⟨⟨ARtrace⟩⟩. Instead of explicitly modelling specific purposes of a relationship such as *motivates* and *constrains*, the AREL model represents them by using a generic *causal* relationship. This simplifies the relationship between AE and AR.

A basic form of the model construct is $\{AE_1, AE_2, \ldots\} \rightarrow AR_1 \rightarrow \{AE_a, AE_b, \ldots\}$ 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. Fig. 2 shows a UML representation of 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 *unique* constraint in the diagram specifies that each instance of AE in the relation must be unique.

The causal relationships shown in Fig. 2 are the directional links using ⟨⟨ARtrace⟩⟩. 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 basis for understanding the architecture design. A cause-effect relationship to generalise specific relationships such as "*motivates*" is advantageous because it simplifies the complexity of traversal.

An AE can be an input or an outcome of a decision. 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 out the reasons of the decision and what alternatives have been considered.

UML is widely adopted in the industry for analysis and design specification. It is often used for requirements specification, use case analysis and design specification. We have selected UML to model AREL for a number of reasons. Firstly, designers can use the UML elements that they already document to be part of the AREL model, hence saving efforts in repeated data entry. Secondly, the data entry and update of architecture rationale can be performed with the same UML tool (see Section 6). Finally, UML graphical representations can effectively depict complex relationships.

3.3. Architecture elements

In AREL, the architecture element AE is an artefact that forms part of the architecture design. They comprise the business requirements to be satisfied, the technical and organisational constraints on the architecture design, the assumptions which need to be observed and the design objects which are the results of the architecture design.

Architecture elements can be classified by a related set of concerns called viewpoints (IEEE, 2000). 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 (Koning and van Vliet, 2005; Hilliard, 2001; Lassing et al., 2001), there are common viewpoints which are general to most software architectures. Using TOGAF as an example (The Open Group, 2003), we have selected four generic viewpoints (business, data, applications and technology) to classify AEs.

Fig. 3 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 (Tang et al., 2006a). Design objects are architecture elements classified by the data, application or technology viewpoints.

3.3.1. 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 to provide a context to decision making.

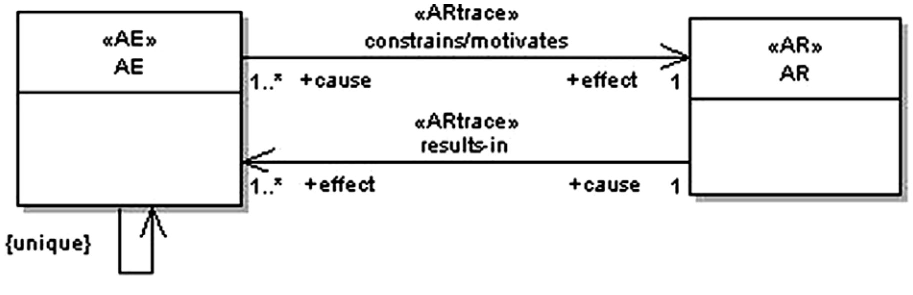


Fig. 2. A causal relationship between AE and AR.

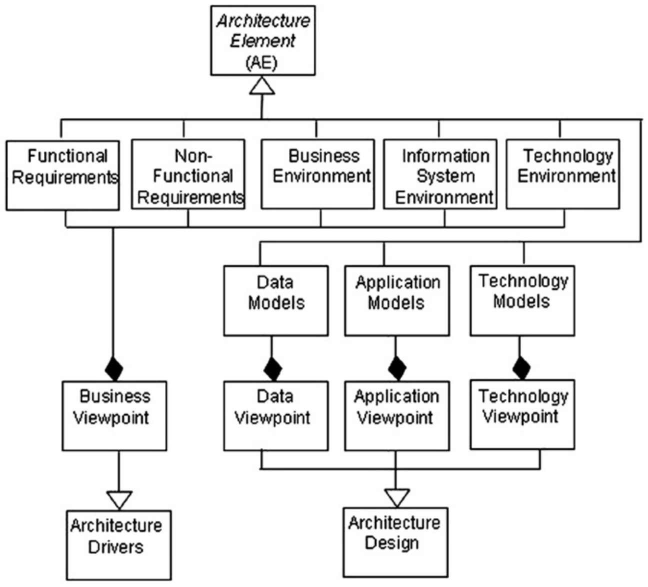


Fig. 3. A classification of architecture elements.

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.

3.3.2. 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 Fig. 3). *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; and (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 a requirement may affect the design objects in the application viewpoint.

3.4. Architecture rationale

In AREL, an AR comprises three types of justifications (Fig. 4): qualitative rationale, quantitative rationale and alternative architecture rationale (Tang and Han, 2005). Qualitative design rationale (QLR) is the reasoning and the arguments, in a textual form, for or against a design decision. Quantitative rationale (QNR) uses the indices to indicate the relative costs, benefits and risks of the design options. These two types of rationale are described in Sections 3.4.1 and 3.4.2 respectively. An AR also contains the Alternative Architecture Rationale (AAR) which documents the discarded design options (see Section 3.4.3).

Using AR to encapsulate and relate architecture rationale to design objects, the AREL approach is different to the other argumentation-based methods:

- Simplification – the AR entity records the key design issues, argumentation and design alternatives without explicitly capturing the design deliberation relationships. We argue that in most cases the design deliberation relationships are not required by the designers and they are time-consuming to capture. This approach simplifies the capture process by only capturing the results or justifications of the decisions without the overhead.
- Encapsulation – design rationale are encapsulated in an AR. This way the decisions can be incorporated into the design process naturally to show the causal relationships

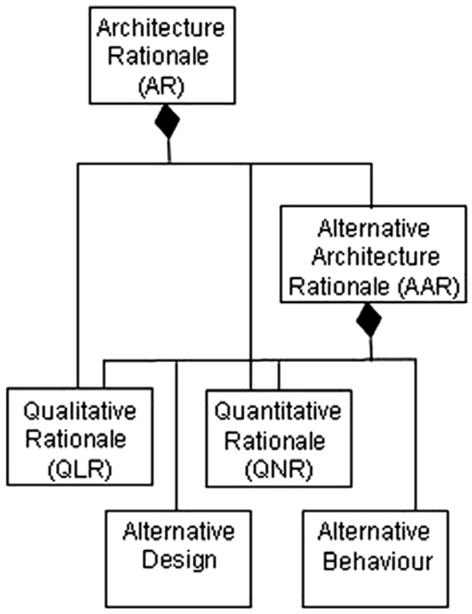

Architecture Rationale (AR)
Alternative Architecture Rationale (AAR)
Qualitative Rationale (QLR)
Quantitative Rationale (QNR)
Alternative Design
Alternative Behaviour

Fig. 4. Components of architecture rationale.

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.
- 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.

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.

3.4.1. 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. QLR is defined using a combination of reasoning methods (Lee and Lai, 1996; Maclean et al., 1996; Clements et al., 2002; Tyree and Akerman, 2005; Ali-Babar et al., 2006). 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 which 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.

3.4.2. 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 accuracy of design decisions using quality systems such as CMMI.

Quantitative rationale enables the systematic estimates of the cost, benefit and risk of design options. Using these three basic elements, we can compute the expected return of each design option. As such, architects are able to justify why certain design options are better than the others. This article does not explore the use of quantitative design rationale. Interested readers are referred to Tang and Han (2005).

3.4.3. 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 Fig. 4). This information allows architects to understand and verify the discarded design options, their relative merits and demerits, that have been considered in the decision process.

The implementation of AAR relies on a UML tool. Both AR and AAR are implemented by the *package* entity in UML. Therefore, 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.

3.5. Extending AREL to support evolution

Architecture systems evolve over time. The evolution signifies changes in the requirements or in the environments that dictate the design. 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 (Tang et al., 2006a). 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 to capture the evolution history.

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

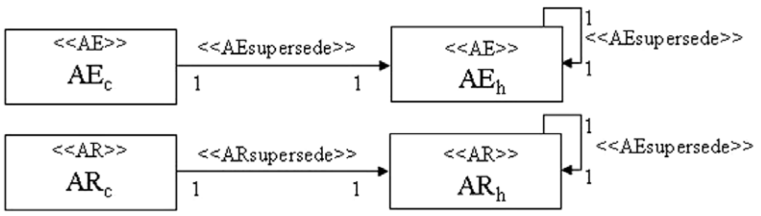


Fig. 5. AE and AR evolution support in eAREL.

elements are kept in superseded elements AR_h and AE_h. If multiple supersedence exists, they are linked in a linear linked list. This supersedence chain represents the direct supersedence relationship between different versions of architecture elements and rationales. The supersedence link of AR is stereotyped by ⟨⟨ARsupersede⟩⟩ and the supersedence link of AE is stereotyped by ⟨⟨AEsupersede⟩⟩. Fig. 5 shows the relationship in UML.

In the eAREL model, the ⟨⟨ARtrace⟩⟩ links between superseded architecture elements and rationales are also kept to retain their relationships. The *version* attribute of ⟨⟨ARtrace⟩⟩ links is reset to indicate that it is a replaced version. In this way it supports the retension of historical architecture design decisions.

4. 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 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-connection 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 ⟨⟨ARtrace⟩⟩, ⟨⟨AEsupersede⟩⟩ and ⟨⟨ARsupersede⟩⟩.

- Forward trace – the purpose of this trace is to support the impact analysis of the architecture design. Given an architecture element AE_1, all architecture rationale and architecture elements which 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 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 (Sparx Systems, 2005) 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 3.3), 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 only. 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. Further classifications to support analysis and traceability are beyond the scope of this article.

5. A case study

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 payments and settlements of the financial centre Guangzhou and its neighbouring cities. The Electronic Fund Transfer System (EFT) was built by the first 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 of the High Value Payment System (HVPS) and the Settlement Account Management System (SAM).

The EFT System took about two years to design, develop and test, employing thirty designers and developers. High-value payments are instructions exchanged between the central bank and participating banks to transfer and settle payments. It is the backbone of the financial system in the region. As such, the EFT system has to be reliable, secure and efficient. At the time of the system 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 foundation of the system is the architecture which provides the processing services infrastructure for the online fund transfers between the local banks and the central bank.

The system design was extensively specified and documented. However, it is still difficult for anyone outside the original development team to understand its intricate design. Using the AREL representation, we retrospectively capture the design rationale of the EFT system. We demonstrate the advantages of being able to trace and explain the architecture from the design reasoning perspective. It provides important information that was missing from most of the design specifications in which the focus is on design organisation, interface and behaviour. We illustrate the payment message processing layer and its design rationale traceability below using Enterprise Architect and the AREL tool.

5.1. Design rationale representation

The design specifications of the EFT system 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.

Fig. 6 is an example of a 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 of the decision are R4_1_3, R2_3_1, R2_3_4 and R2_3_5. They 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 Fig. 7. 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 needs a pair of processes by each party to listen and to send messages. Whilst the sender process is waiting for an acknowledgement, it cannot process other messages and hence the concurrency 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. The constraint or the implication of such a 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. This intricate design relationship 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.

5.2. 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 impact analysis and backward tracing for analysing root causes.

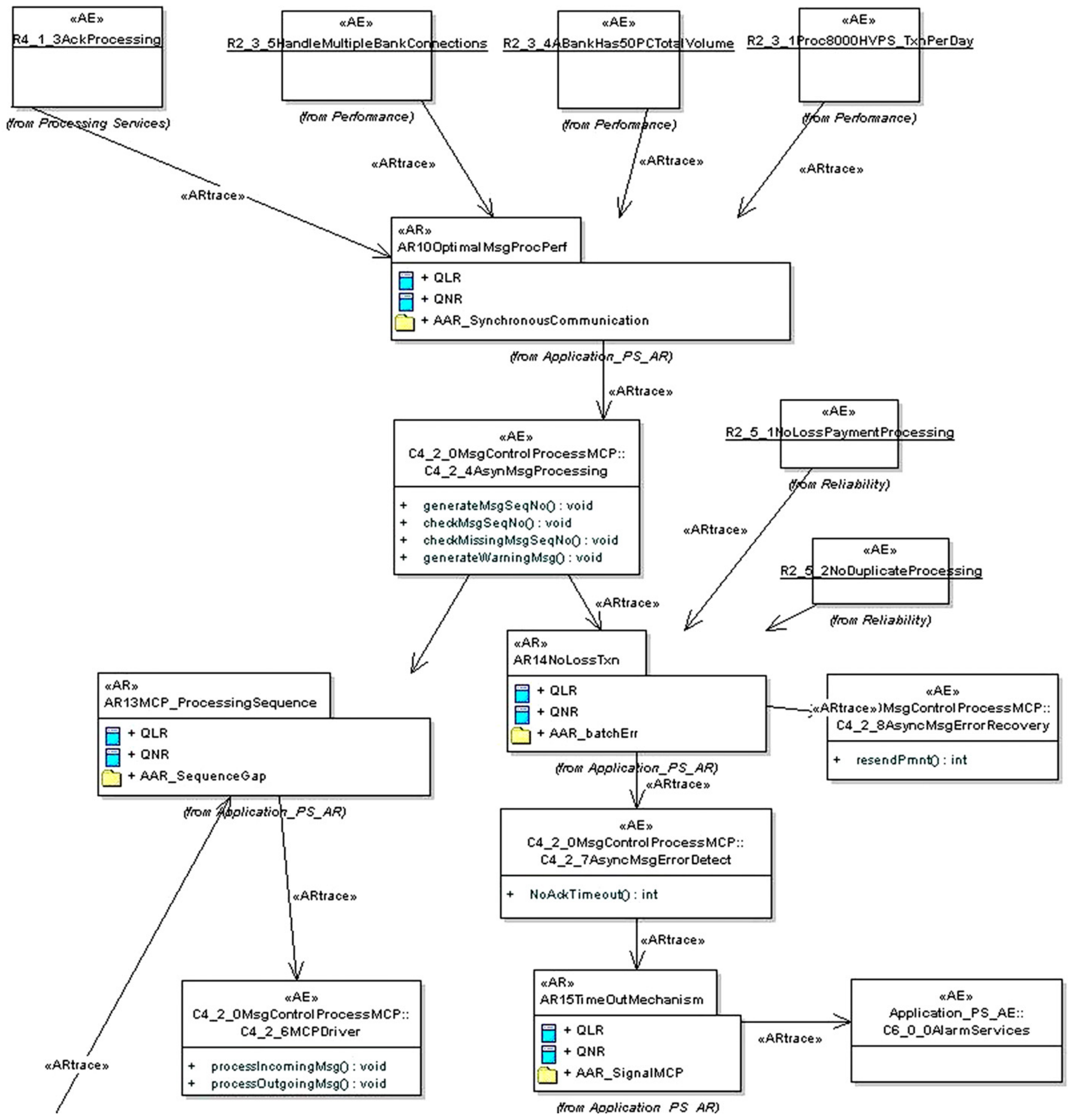
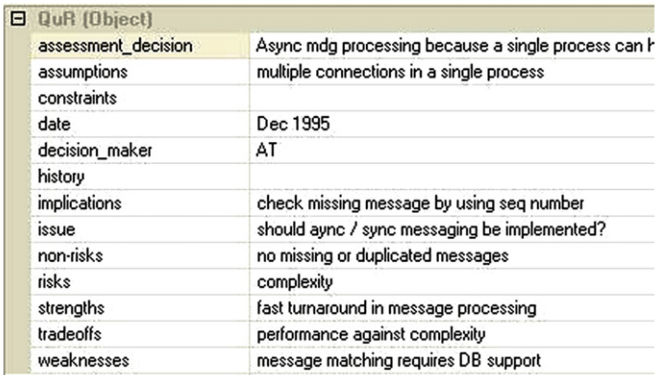

«AE» R4_1_3AckProcessing (from Processing Services)
«AE» R2_3_5HandleMultipleBankConnections (from Performance)
«AE» R2_3_4ABankHas50PCTotalVolume (from Performance)
«AE» R2_3_1Proc8000HVPS_TxnPerDay (from Performance)
«ARtrace»
«AR» AR10OptimalMsgProcPerf
+ QLR
+ QNR
+ AAR_SynchronousCommunication
(from Application_PS_AR)
«AE» C4_2_0MsgControlProcessMCP:: C4_2_4AsynMsgProcessing
+ generateMsgSeqNo() : void
+ checkMsgSeqNo() : void
+ checkMissingMsgSeqNo() : void
+ generateWarningMsg() : void
«AE» R2_5_1NoLossPaymentProcessing (from Reliability)
«AE» R2_5_2NoDuplicateProcessing (from Reliability)
«AR» AR14NoLossTxn
+ QLR
+ QNR
+ AAR_batchErr
«AE» C4_2_0MsgControlProcessMCP:: C4_2_8AsyncMsgErrorRecovery
+ resendPmnt() : int
«AR» AR13MCP_ProcessingSequence
+ QLR
+ QNR
+ AAR_SequenceGap
«AE» C4_2_0MsgControlProcessMCP:: C4_2_7AsyncMsgErrorDetect
+ NoAckTimeout() : int
«AE» C4_2_0MsgControlProcessMCP:: C4_2_6MCPDriver
+ processIncomingMsg() : void
+ processOutgoingMsg() : void
«AR» AR15TimeOutMechanism
+ QLR
+ QNR
+ AAR_SignalMCP
«AE» Application_PS_AE:: C6_0_0AlarmServices

Fig. 6. Asynchronous message processing decision and its design impact.

5.2.1. Forward tracing

Using requirement R4_1_3 as an example, we use forward tracing to identify any impact the payment acknowledgement requirement has on the architecture design. Fig. 8 is the resulting UML diagram of the forward trace. 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.

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.



⊟ QuR (Object)

assessment_decision	Async mdg processing because a single process can h
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 aync / 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

Fig. 7. Details of the design rationale AR10.

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.

Architects could use the AREL tool to specify the required viewpoints, the tool would traverse the AREL

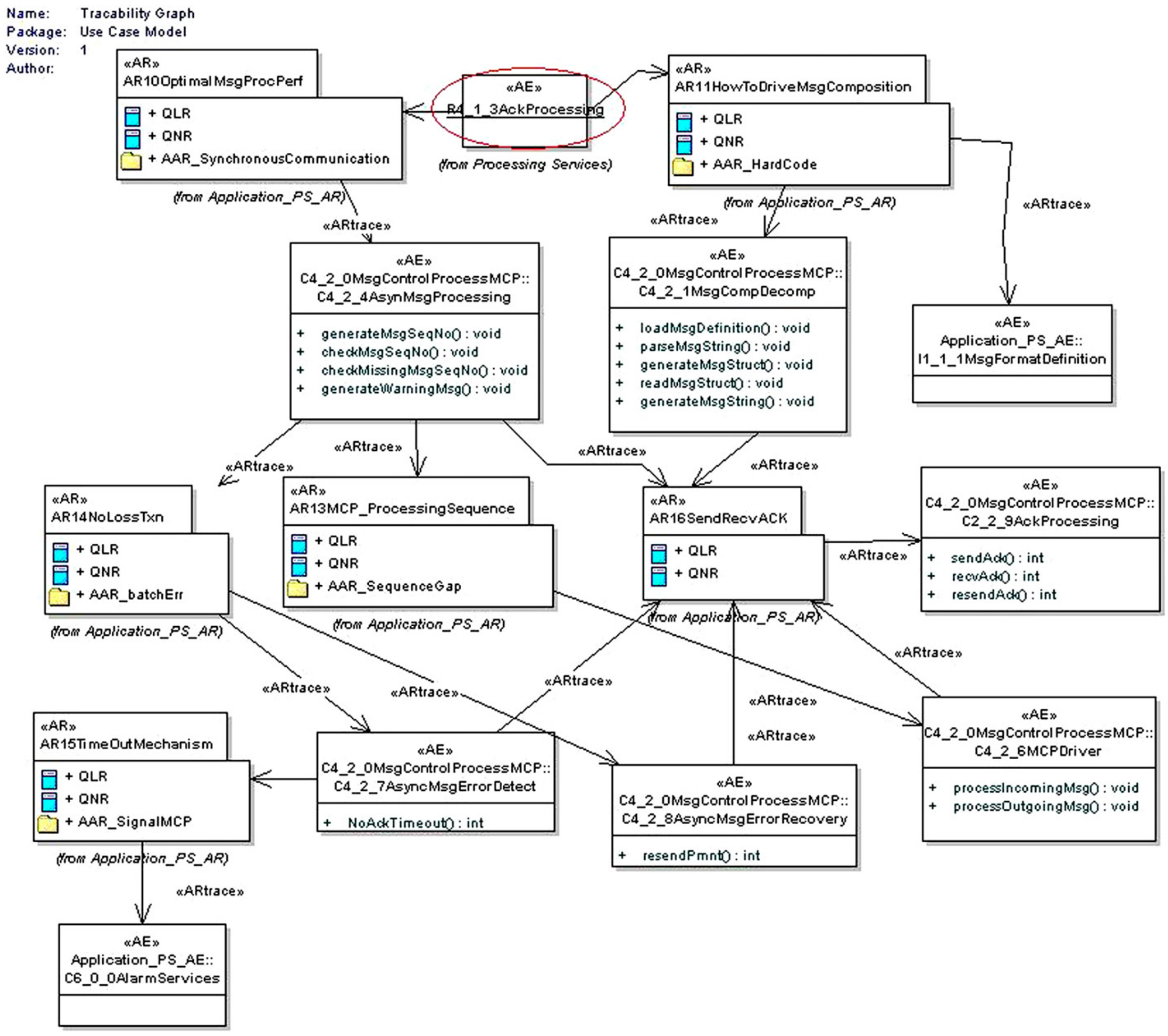

Name: Tracability Graph
Package: Use Case Model
Version: 1
Author:
«AR» AR10OptimalMsgProcPerf
+ QLR
+ QNR
+ AAR_SynchronousCommunication
(from Application_PS_AR)
«AE» R4_1_3AckProcessing
(from Processing Services)
«AR» AR11HowToDriveMsgComposition
+ QLR
+ QNR
+ AAR_HardCode
(from Application_PS_AR)
«ARtrace»
«AE» C4_2_0MsgControlProcessMCP:: C4_2_4AsynMsgProcessing
+ generateMsgSeqNo() : void
+ checkMsgSeqNo() : void
+ checkMissingMsgSeqNo() : void
+ generateWarningMsg() : void
«AE» C4_2_0MsgControlProcessMCP:: C4_2_1MsgCompDecomp
+ loadMsgDefinition() : void
+ parseMsgString() : void
+ generateMsgStruct() : void
+ readMsgStruct() : void
+ generateMsgString() : void
«AE» Application_PS_AE:: I1_1_1MsgFormatDefinition
«AR» AR14NoLossTxn
+ QLR
+ QNR
+ AAR_batchErr
«AR» AR13MCP_ProcessingSequence
+ QLR
+ QNR
+ AAR_SequenceGap
«AR» AR16SendRecvACK
+ QLR
+ QNR
«AE» C4_2_0MsgControlProcessMCP:: C2_2_9AckProcessing
+ sendAck() : int
+ recvAck() : int
+ resendAck() : int
«AR» AR15TimeOutMechanism
+ QLR
+ QNR
+ AAR_SignalMCP
«AE» C4_2_0MsgControlProcessMCP:: C4_2_7AsyncMsgErrorDetect
+ NoAckTimeout() : int
«AE» C4_2_0MsgControlProcessMCP:: C4_2_8AsyncMsgErrorRecovery
+ resendPmnt() : int
«AE» C4_2_0MsgControlProcessMCP:: C4_2_6MCPDriver
+ processIncomingMsg() : void
+ processOutgoingMsg() : void
«AE» Application_PS_AE:: C6_0_0AlarmServices

Fig. 8. Forward tracing for impact analysis.

model to create trace results (see Section 6). Forward traceability 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 allow 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.

5.2.2. Backward tracing

Using the same design, we analyse the root-causes of having the alarm server by backward tracing. We use the AREL tool to trace backwards from C6_0_0, the resulting AREL graph is shown in Fig. 9. 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

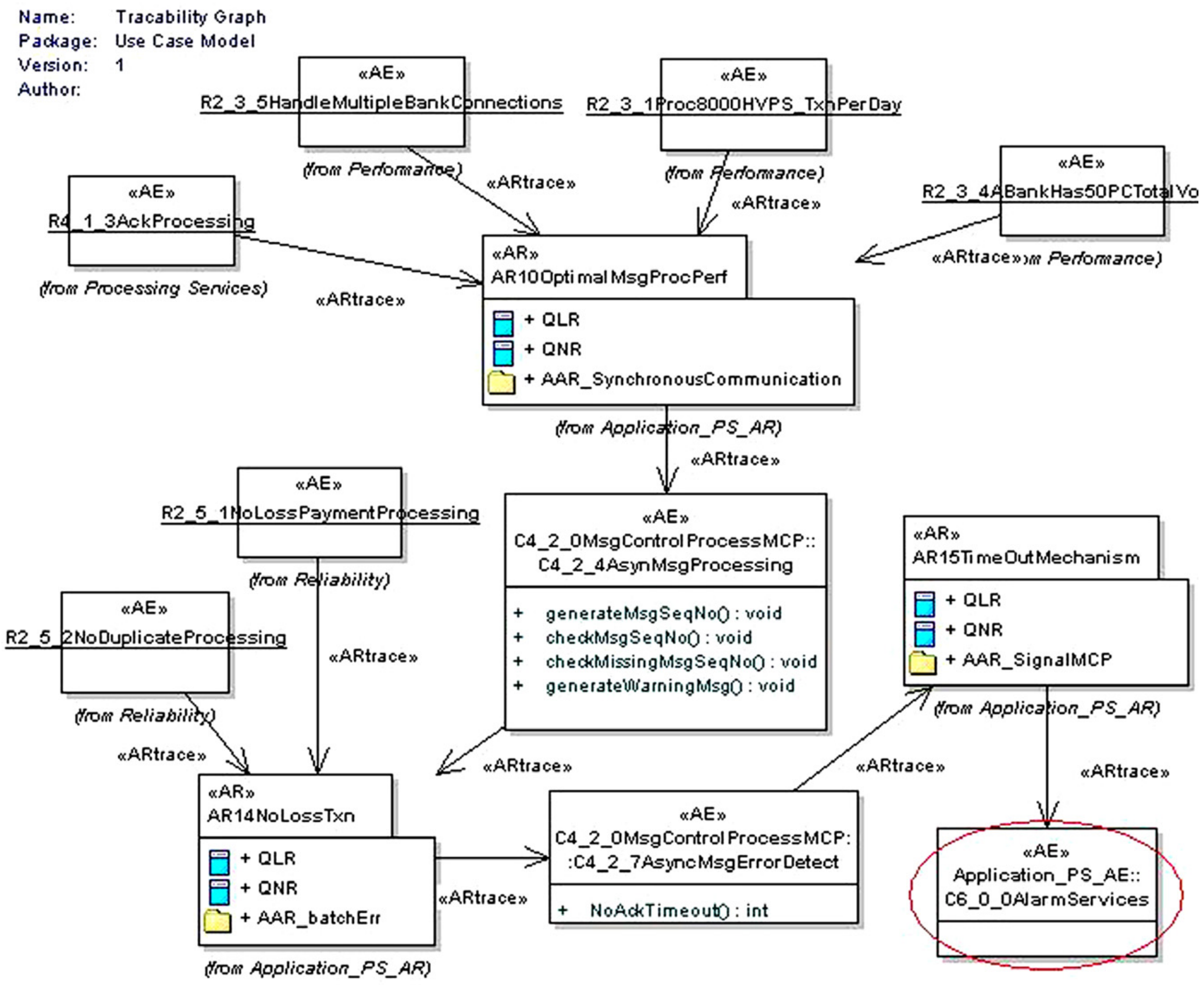

Name: Tracability Graph
Package: Use Case Model
Version: 1
Author:
«AE» R2_3_5HandleMultipleBankConnections (from Performance)
«AE» R2_3_1Proc8000HVPS_TxnPerDay (from Performance)
«AE» R2_3_4ABankHas50PCTotalVo (from Performance)
«AE» R4_1_3AckProcessing (from Processing Services)
«ARtrace»
«AR» AR10OptimalMsgProcPerf
+ QLR
+ QNR
+ AAR_SynchronousCommunication
(from Application_PS_AR)
«AE» R2_5_1NoLossPaymentProcessing (from Reliability)
«AE» R2_5_2NoDuplicateProcessing (from Reliability)
«AE» C4_2_0MsgControlProcessMCP:: C4_2_4AsynMsgProcessing
+ generateMsgSeqNo() : void
+ checkMsgSeqNo() : void
+ checkMissingMsgSeqNo() : void
+ generateWarningMsg() : void
«AR» AR15TimeOutMechanism
+ QLR
+ QNR
+ AAR_SignalMCP
(from Application_PS_AR)
«AR» AR14NoLossTxn
+ QLR
+ QNR
+ AAR_batchErr
(from Application_PS_AR)
«AE» C4_2_0MsgControlProcessMCP: :C4_2_7AsyncMsgErrorDetect
+ NoAckTimeout() : int
«AE» Application_PS_AE:: C6_0_0AlarmServices

Fig. 9. Backward tracing for root-cause analysis.

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 in a UML diagram. 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.

5.3. Tracing architecture design evolution

Since architectures can have a long live-span and are subject to enhancements and adaptations over time, the

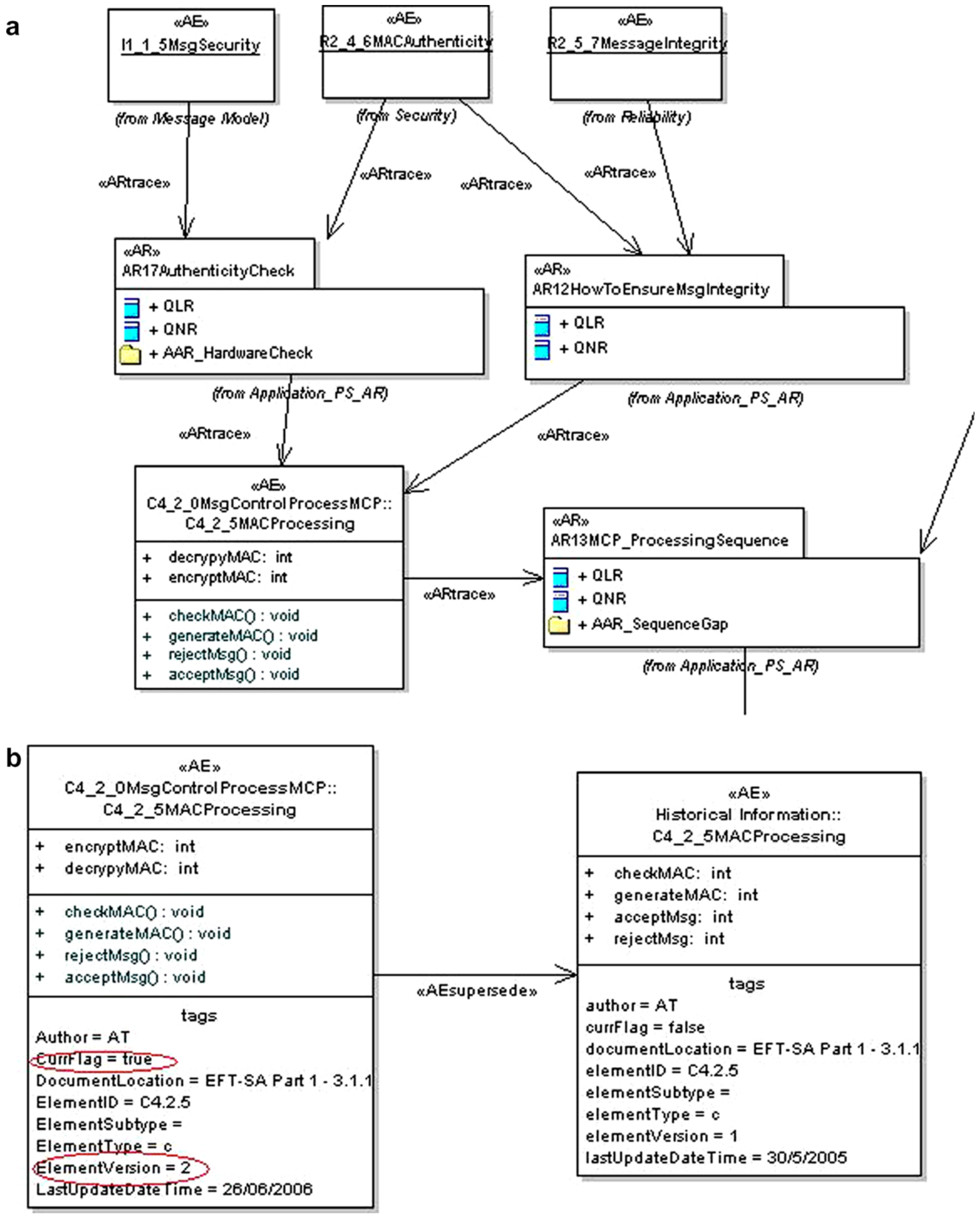


Fig. 10. (a) MAC processing a and (b) superseded architecture element.

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 Fig. 10a, 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 ⟨⟨AEsupersede⟩⟩ links it with version 2 of the C4_2_5, which is in the current AREL model. This is shown in Fig. 10b. Since version 1 of C4_2_5 MAC has now been replaced, its ⟨⟨ARtrace⟩⟩ links to AR12 and AR13 are now obsolete, therefore these links are made non-current. New ⟨⟨ARtrace⟩⟩ links are created to be related to AR12 and AR13 for version 2 of the C4_2_5.

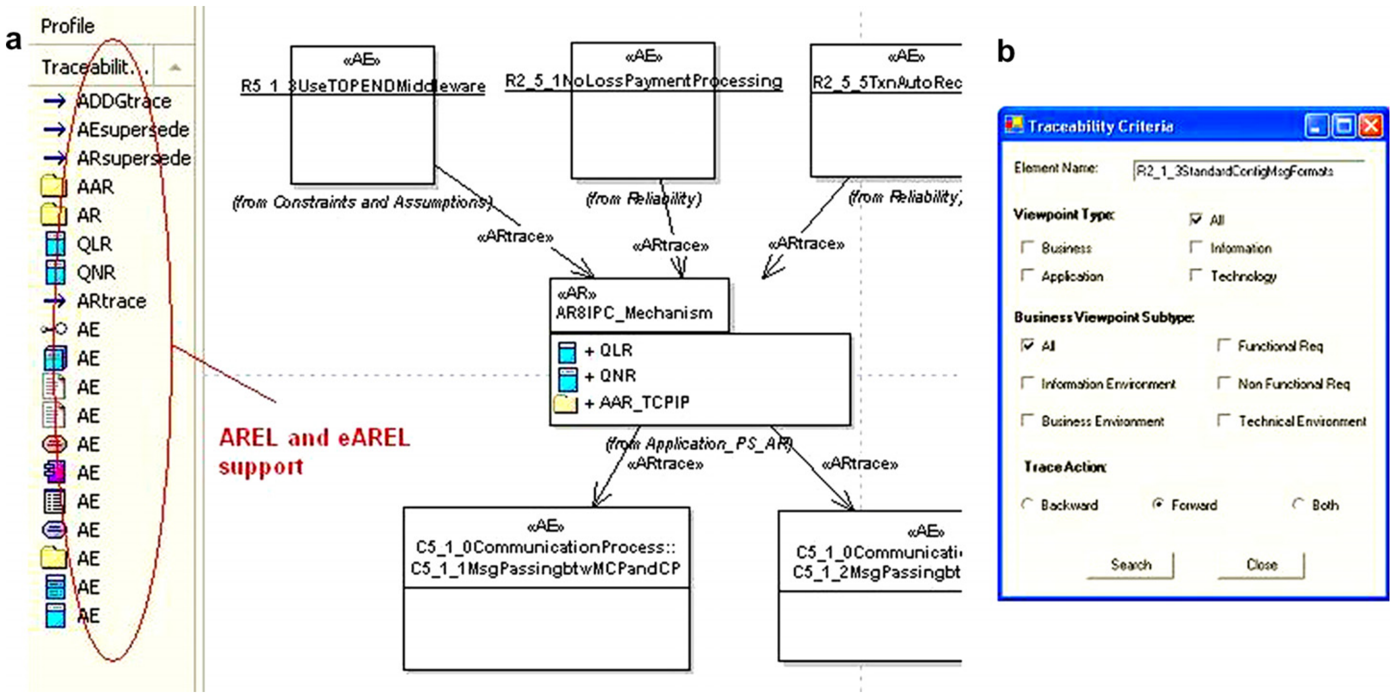
In the above example, we have demonstrated how a design can be superseded by its replacement. The supersession of AR works in the same way. eAREL can be useful in many cases. For instances, an architect may wish to investigate the extent of design changes in a particular release, or investigate the impact 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. eAREL has the following characteristics:

- Capture of decision and element changes – evolutionary changes to AE or AR can be captured through the ⟨⟨AEsupersede⟩⟩ and ⟨⟨ARsupersede⟩⟩ relationships respectively.
- Manual trace – the tracing of eAREL is a manual process of traversing the UML diagrams using the ⟨⟨AEsupersede⟩⟩ and ⟨⟨ ARsupersede⟩⟩ links.

6. Tool support

A UML tool, Enterprise Architect version 5.00.767 (Sparx Systems, 2005), is used to capture and construct AREL and eAREL models. Existing and new design elements created in UML can be modelled as AEs to serve the dual-purpose of being a design specification as well as being part of a design rationale model. It helps to reduce data entry.

Enterprise Architect provides a convenient way to allow templates to be added to extend the standard UML. Our implementation of the design rationale capture template is by way of creating Enterprise Architect profiles for ⟨⟨AR⟩⟩ and ⟨⟨AE⟩⟩ stereotypes (Tang, 2005). Once installed, architects can create AREL model elements by dragging and dropping from a tool-box during design modelling and filling in the necessary information (see



Traceability Criteria

Element Name: R2_1_3StandardConfigMsgFormats

Viewpoint Type: [✓ All] Business Information Application Technology

Business Viewpoint Subtype: [✓ All] Functional Req Information Environment Non Functional Req Business Environment Technical Environment

Trace Action: Backward [✓ Forward] Both

Search Close

Fig. 11. (a) Extended UML profile to support AREL rationale capture and (b) AREL traceability tool.

Fig. 11a). This process is non-intrusive and can be easily incorporated into existing design processes. We built the AREL Tool in .Net to perform the traceability functions (Tang, 2006).

The tool support for design traceability and reasoning is a combination of using Enterprise Architect and the AREL tool. The following are the use cases for retrieving design rationale from the AREL models:

(1) Understand design rationale – since design rationale AR is captured within Enterprise Architect in a graphical form, architects can retrieve the information by inspecting the encapsulated qualitative design rationale QLR, quantitative design rationale QNR and alternative design options AAR. This is a matter of delving into the details of the AR node.
(2) Forward and backward tracing – a reasonably large system would contain many architecture elements and decisions which can be difficult to trace, therefore an automated tool would facilitate the tracing process by showing only results which are of interest. The following are the steps in using the tools:
    (i) Specify the AREL model.
    (ii) Specify the trace criteria by (a) identifying the architecture element to be the starting point of the trace; (b) specifying the scope of the trace using viewpoints and sub-viewpoints; (c) specifying whether tracing backward or forward or both. See example in Fig. 11b.
    (iii) The results of the trace is generated as a UML diagram in Enterprise Architect. Figs. 8 and 9 are examples of trace results.

Currently the AREL tool is a separate program and not integrated as one of the menu options within Enterprise Architect. This is due to the compiler version not compatible with Enterprise Architect because of the Standard Operating Environment (SOE) mandated by the university. As a result, evolution trace is currently a manual procedure and the automated functions to create and traverse eAREL cannot be implemented. This is another example of how environmental factors have a chain effect on the architecture design of a system.

7. Conclusion

The lack of design rationale and its impact on architecture design are well documented (Bosch, 2004; Perry and Wolf, 1992). The argumentation-based design rationale methods have not resolved this issue entirely because they do not relate design elements to design justifications (Herbsleb and Kuwana, 1993). Additionally, these methods are ineffective in capturing and communicating design reasoning (Shipman and McCall, 1997).

In this article, we have introduced a rationale-based architecture model (AREL) to capture architecture design rationale. The AREL model uses two types of reasoning support: motivational reasons and design rationale. Motivational reasons induce architecture design issues in that design decisions are required to resolve them. Architecture design decisions are justified by architecture design rationale which is comprised of qualitative rationale, quantitative rationale and alternative design options. Design solutions and design objects are created as a result of design decisions.

AREL provides a new perspective to explain and help architects understand architecture design. The intricately inter-dependent design objects sharing common requirements, assumptions, constraints and decisions can be related and reasoned. This reasoning support complements the structural and interface documentation commonly used in design specifications.

In order to facilitate the understanding of the architecture design to support architecture verification and maintenance, 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 dependent 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 design evolution of an architecture element or an architecture rationale. The implementation of AREL and its traceability are supported by the AREL tool as well as the UML tool Enterprise Architect.

We have used a partial architecture design of a payment system to demonstrate the application of AREL and its traceability. This has been done retrospectively for an existing system. During this exercise, we have recovered over fifty key architecture decisions we consider to be worthwhile documenting to recapture the architecture rationale. Although we can show AREL's explanatory and reasoning power to help understand an existing system, we are unable to demonstrate AREL's reasoning capabilities to support new system development. It is our intention to test this aspect of AREL in the future.

Acknowledgements

We would like to thank Mr. A. Lau of NCR and Mr. Z.T. Duan of the Guangzhou Electronic Banking Settlement Center to give us their permissions to reference the EFT system in our research. We also thank Sparx Systems for providing Enterprise Architect to support our work.

References

Ali-Babar, M., Gorton, I., Kitchenham, B., 2006. A framework for supporting architecture knowledge and rationale management. In: Rationale Management in Software Engineering. Springer, pp. 237–254.

Bosch, J., 2004. Software architecture: the next step. In: Proceedings 1st European Workshop on Software Architecture (EWSA). St Andrews, UK. pp. 194–199.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures: Views and Beyond, first ed. Addison Wesley.

Conklin, J., Begeman, M., 1988. gIBIS: a hypertext tool for exploratory policy discussion. In: Proceedings ACM Conference on Computer-Supported Cooperative Work. pp. 140–152.

Egyed, A., 2001. A scenario-driven approach to traceability. In: Proceedings 23rd International Conference on Software Engineering (ICSE). pp. 123–132.

Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural mismatch or why it's hard to build systems out of existing parts. In: Proceedings 17th International Conference on Software Engineering (ICSE). pp. 179–185.

Gotel, O.C.Z., Finkelstein, A.C.W., 1994. An analysis of the requirements traceability problem. In: Proceedings International Conference on Requirements Engineering (RE). pp. 94–101.

Gruber, T., Russell, D., 1996. Generative design rationale: beyond the record and replay paradigm. In: Moran, T., Carroll, J. (Eds.), Design Rationale: Concepts, Techniques and Use. Lawrence Erlbaum Associates, pp. 323–350 (Chapter 11).

Han, J., 1997. Designing for increased software maintainability. In: Proceedings International Conference on Software Maintenance. IEEE Computer Society Press, pp. 278–286.

Han, J., 2001. TRAM: a tool for requirements and architecture management. In: Proceedings 24th Australasian Computer Science Conference. IEEE Computer Society Press, Gold Coast, Australia, pp. 60–68.

Haumer, P., Pohl, K., Weidenhaupt, K., Jarke, M., 1999. Improving reviews by extended traceability. In: Proceedings 32nd Hawaii International Conference on System Sciences.

Herbsleb, J.D., Kuwana, E., 1993. Preserving knowledge in design projects: what designers need to know? In: Proceedings of the Conference on Human Factors in Computing Systems, April 24–29. ACM Press, New York, pp. 7–14.

Hilliard, R., 2001. Viewpoint modeling. In: Proceedings of 1st ICSE Workshop on Describing Software Architecture with UML.

Hughes, T., Martin, C., 1998. Design traceability of complex systems. In: Proceedings 4th Annual Symposium on Human Interaction with Complex Systems. pp. 37–41.

IEEE, 1996. IEEE/EIA standard-industry implementation of ISO/IEC 12207:1995. Information Technology – Software Life Cycle Processes (IEEE/EIA Std 12207.0-1996).

IEEE, 1997. 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).

IEEE, 2000. IEEE Recommended Practice for Architecture Description of Software-Intensive System (IEEE Std 1471-2000).

Koning, H., van Vliet, H., 2005. A method for defining IEEE Std 1471 viewpoints. The Journal of Systems and Software 79, 120–131.

Kruchten, P., Lago, P., Vliet, H. v., Wolf, T., 2005. Building up and exploiting architectural knowledge. In: Proceedings 5th IEEE/IFIP Working Conference on Software Architecture.

Kunz, W., Rittel, H., 1970. Issues as elements of information systemsCenter for Planning and Development Research. University of California, Berkeley.

Lassing, N., Rijsenbrij, D., Vliet, H.V., 2001. Viewpoints on modifiability. International Journal of Software Engineering and Knowledge Engineering 11 (4), 453–478.

Lee, J., 1991. Extending the Potts and Bruns model for recording design rationale. In: 13th International Conference on Software Engineering. pp. 114–125.

Lee, J., 1997. Design rationale systems: understanding the issues. IEEE Expert 12 (3), 78–85.

Lee, J., Lai, K., 1996. What's in design rationale? In: Moran, T., Carroll, J. (Eds.), Design Rationale: Concepts Techniques and Use. Lawrence Erlbaum Associates, pp. 21–52 (Chapter 2).

Maclean, A., Young, R., Bellotti, V., Moran, T., 1996. Questions, options and criteria: elements of design space analysis. In: Moran, T., Carroll, J. (Eds.), Design Rationale: Concepts Techniques and Use. Lawrence Erlbaum Associates, pp. 53–106 (Chapter 3).

McCall, R., 1991. PHI: a conceptual foundation for design hypermedia. Design Studies 12 (1), 30–41.

Nuseibeh, B., 2004. Crosscutting requirements. In: Proceedings 3rd International Conference on aspect-oriented software development. pp. 3–4.

Parnas, D., Clements, P., 1985. A rational design process: how and why to fake it? IEEE Transactions on Software Engineering 12, 251–257.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software. ACM SIGSOFT 17 (4), 40–52.

Pinheiro, F.A.C., 2000. Formal and informal aspects of requirements tracing. In: Workshop em Engenharia de Requisitos. Brazil. pp. 1–21.

Pinheiro, F.A.C., Goguen, J.A., 1996. An object-oriented tool for tracing requirements. IEEE Software 13 (2), 52–64.

Potts, C., 1996. Supporting software design: integrating design methods and design rationale. In: Moran, T., Carroll, J. (Eds.), Design Rationale: Concepts, Techniques and Use. Lawrence Erlbaum Associates, pp. 295–322 (Chapter 10).

Ramesh, B., Jarke, M., 2001. Towards reference models for requirements traceability. IEEE Transactions on Software Engineering 27 (1), 58–93.

Roeller, R., Lago, P., van Vliet, H., 2005. Recovering architectural assumptions. The Journal of Systems and Software 79, 552–573.

Savolainen, J., Kuusela, J., 2002. Framework for goal driven system design. In: Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02).

Shipman III, F., McCall, R., 1997. Integrating different perspectives on design rationale: supporting the emergence of design rationale from design communication. Artificial Intelligence in Engineering Design, Analysis, and Manufacturing 11 (2).

Simon, H., 1981. The Sciences of the Artificial. MIT Press.

Smith, W., 1998. Best Practices: Application of DOORS to System Integration. QSS Quality Systems and Software, 1999 So. Bascom Av., Suite 700, Cambell, CA 950008, USA.

Spanoudakis, G., Zisman, A., Pérez-Miñana, E., Krause, P., 2004. Rule-based generation of requirements traceability relations. The Journal of Systems and Software 72 (2), 105–127.

Sparx Systems, 2005. Enterprise Architect V5.00.767. <http://www.sparxsystems.com/>.

Tang, A., 2005. An UML Profile Extension to Support Architecture Decision Traceability using Enterprise Architect. <http://www.ict.swin.edu.au/personal/atang/ArelStereotypePackage.zip>.

Tang, A., 2006. An AREL Tool for Traceability and Validation. <http://www.ict.swin.edu.au/personal/atang/AREL-Tool.zip>.

Tang, A., Han, J., 2005. 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). IEEE, USA, pp. 135–144.

Tang, A., Babar, M., Gorton, I., Han, J., 2006a. A survey of architecture design rationale. The Journal of Systems and Software. doi:10.1016/j.jss.2006.04.029.

Tang, A., Nicholson, A., Jin, Y., Han, J., 2006b. Using Bayesian belief networks for change impact analysis in architecture design. The Journal of Systems and Software. doi:10.1016/j.jss.2006.04.004.

The Open Group, 2003. The Open Group Architecture Framework (v8.1 enterprise edition). <http://www.opengroup.org/architecture/togaf/#download>.

Toulmin, S., 1958. The Uses of Argument. Cambridge University Press.

Tyree, J., Akerman, A., 2005. Architecture decisions: demystifying architecture. IEEE Software 22 (2), 19–27.

Watkins, R., Neal, M., 1994. Why and how of requirements tracing. IEEE Software 11 (4), 104–106.