A Model for Specifying Rationale Using an Architecture Description Language

Lakshitha de Silva[1] and Dharini Balasubramaniam[2]

School of Computer Science, University of St Andrews, St Andrews, KY16 9SX, UK.
[1]`lakshitha.desilva@acm.org`
[2]`dharini@cs.st-andrews.ac.uk`

Abstract. Besides structural and behavioural properties, rationale plays a crucial role in defining the architecture of a software system. However, unlike other architectural features, rationale often remains unspecified and inaccessible to tools. Existing approaches for recording rationale are not widely adopted. This paper proposes a simple model for capturing rationale as part of an architecture specification and attaching it to elements in the architecture. The bi-directional links between rationales and elements enable forward and backward traceability. We describe a textual architecture description language named Grasp that implements this model, and illustrate its capabilities using an example.

1 Introduction

Software architecture [9,11] establishes a crucial foundation for the systematic development and evolution of software. It provides a high level abstraction of the structure and behaviour of a system in terms of its constituent elements and their interactions. An architecture also reflects *rationale*, the reasoning behind design decisions that guided its creation. Rationale is an intrinsic component of software architecture [9], representing alternatives, trade-offs, assumptions, constraints and others factors considered during its design.

A number of architecture description languages (ADLs) have been developed over the years to formally specify architectures. Most ADLs are conceptually based on the primitives of components, connectors, interfaces and configurations, though wide variations exist in the treatment of these primitives [7]. However, most current ADLs are unable to effectively describe rationale. While a number of techniques (e.g. [13,15,10,6]) have been proposed to capture and represent rationale, these have not been extended to ADLs.

The importance of explicitly recording architecture rationale has been widely discussed [9,2,12]. The general consensus is that the tendency to modify software without due consideration to rationale often causes *architecture erosion*. Furthermore, the complex nature of rationale, which is usually a blend of design trade-offs, technical limitations and other constraints, is not completely reflected in the implementation. Therefore, the availability of an effective mechanism to capture rationale from the outset of the design process is imperative to retaining the engineering quality, performance and maintainability of a system.

This paper proposes an approach for specifying rationale as part of an architecture model using *Grasp* ADL. Grasp allows the association of rationale descriptors with elements in the architecture. Both formal expressions and natural language can be used for specifying rationale. An expression in a rationale description may refer to external requirements, quality attributes or elements in the architecture. We describe these concepts with the aid of an example. The paper concludes by outlining the current status and planned future work.

Our work on rationale is part of a larger research agenda for controlling architecture erosion in software systems. We hypothesise that, by maintaining consistency between an architectural specification and its implementation, it is possible to minimise erosion. In order to verify this hypothesis, a simple but expressive model of rationale and an ADL that can support the specification and evaluation of this model are required.

2 Related Work

Tyree and Akerman [15] use document templates to formally record design decisions. However such techniques face the difficulty of linking documents to other forms of architectural models, which in turn hampers the ability to trace rationale to elements in the architecture using tools. Practitioners may also find it hard to keep documentation up to date with large evolving architectures.

The Archium model [6] describes software architectures as compositions of design decisions, giving due prominence to rationale. However, the effort required to model a conceptual design-decision-oriented architecture and then transform this into a basis for implementation may inhibit industrial adoption.

The Architecture Rationale Element Linkage (AREL) model attempts to capture architecture rationale with traceability [13]. AREL promotes architecture rationale to a first-class entity and establishes relationships between rationale and elements in the architecture. These relationships form a causality chain, providing the basis for backward and forward traceability. The rationale model presented in this paper was largely influenced by AREL. Our work both simplifies and extends the AREL model. It is simpler because alternative rationales are excluded and no distinction is made between qualitative and quantitative rationale. We extend AREL by treating rationale as a statement of "reasons", where a reason can optionally be bound to a system requirement, a quality attribute or an element within the architecture itself.

Zhu and Gorton [16] use UML profiles to model design decisions in an architecture specification. This approach also models non-functional requirements and associates them with architectural elements. Capturing rationale as well as design decisions could extend its usefulness. The relationships between quality attributes and rationales in our model were influenced by this work.

Another UML-based approach proposed by Carignano et al. [4] focuses on the architectural design process and its environment rather than the design outcome. While process and environmental factors may significantly influence architectural design, we believe that rationale should be an inherent part of the design outcome in order to be useful for architecture analysis.

Rationale management systems (e.g. SAURAT [3]) provide tool support for recording, managing and associating rationale with various software artefacts. Although these tools are useful for externally retaining rationale of evolving systems, our work makes rationale an intrinsic part of an architecture specification.

3 Conceptual Model

The proposed rationale model consists of three primary entities. They are *architecture element* (AE), *rationale* and *reason*. An AE has zero more associated rationales justifying its purpose. A rationale is a conjunction of one or more reasons. A reason is a logical expression that evaluates to a Boolean outcome. If a reason in a rationale evaluates to false, then every dependent AE is considered to have failed its rationale. Any reason expressed in natural language alone is treated as a logically true statement. Figure 1 formalises this rationale model.

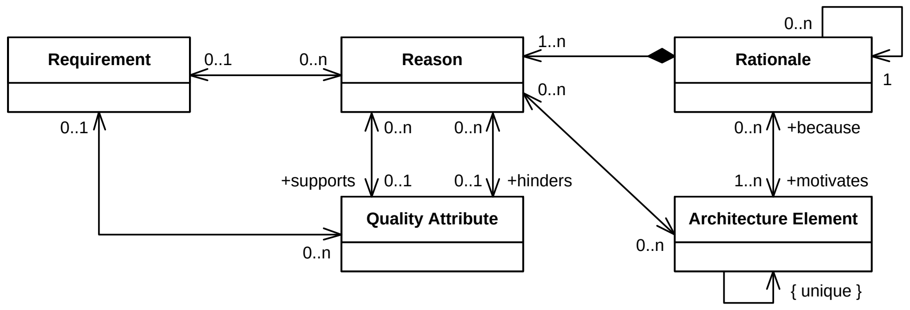

Fig. 1. Model for associating rationale with architecture elements.

A rationale and an AE have a bi-directional association with each other. A rationale *motivates* the existence or behaviour of one or more AEs. Conversely, an AE can be attached to zero or more rationale entities. Since an AE is justified by a rationale associated with it, the term *because* is used to brand this association. It is noted that a rationale can cross-cut many AEs. Our model supports this possibility as a single rationale can be associated with many AEs.

An AE X can also act as a reason for the rationale of another AE Y since X can cause or require the existence of Y in the same architecture. However, direct or indirect cyclic references are not permitted. Thus, relationships between rationales and AEs form an acyclic graph similar to that of AREL. The $\{$*unique*$\}$ constraint applied to the AE entity ensures that each AE instance is unique and therefore does not become a reason for its own rationale.

A rationale may extend another, effectively inheriting its reasons. However, multiple-inheritance is not permitted in the interest of simplicity and clarity.

This model also enables associating *quality attributes* (QAs) as motivating reasons for rationales. A QA is a constraint that should hold when a system implements and delivers its services [14]. Typical drivers for QAs are non-functional requirements. QAs that can be described quantitatively are specified as a collection of properties in this model. Since design choices that support certain QAs may negatively impact others, the association between reasons and QAs are twofold. A reason may *support* one or more QAs. A reason may also *hinder* one or more QAs while in the process of justifying some design decision.

Lastly, reasons and QAs may be attributed to *requirements*, which in this context are references to an external requirement specification. We consider detailed specification of requirements to be outwith the scope of our rationale model.

We note that, in contrast to the rationale model in ISO 402010 [5], this approach does not model architecture decisions. It instead treats an element as a realised design decision (i.e. a design outcome) justified by some rationale(s).

4 Modelling Rationale with Grasp

Grasp is a textual ADL that can define rationale and associate it with elements of a software architecture. It implements the conceptual model presented in Sect. 3. We reuse the case study of an electronic fund transfer system (EFT) used to illustrate the AREL approach [13] in order to demonstrate architecture specification in Grasp. The outcome of applying Grasp to a portion of this model, namely the asynchronous messaging subsystem, is shown in Listing 1.

The EFT system is designed to execute high-value online fund transfers between local banks and the central bank in China. A key requirement for the messaging subsystem is performance and hence, the designers chose to use an asynchronous messaging strategy to achieve this critical quality attribute. Subsequent design decisions and outcomes were directly or indirectly influenced by the decision to build an architecture that supports asynchronous messaging.

The AREL approach uses UML profiles to model rationale, AEs and their associations. These UML elements in the AREL example were manually translated into their equivalent Grasp constructs. The next few subsections discuss different aspects of a Grasp architectural model with the help of this example.

4.1 Modelling Rationale

The Grasp architecture description of the messaging subsystem begins with explicit declarations of QAs and references to external requirements. In the EFT example, Rq_AckProcessing points to an existing requirement that states all messages should be acknowledged while the two QAs, CommPerformance and CommReliability, define performance and reliability qualities that should be exhibited by the messaging subsystem in terms of properties. Once declared, the two QAs and the requirement become motivating reasons for describing rationale.

The Grasp example declares four rationales named AR10, AR13, AR14 and AR15, each with its own reasons. Each rationale is also tagged with a descriptive name using the annotation feature in Grasp. Annotations are name-value pairs useful for providing additional information without altering the semantics of a Grasp construct. Rationale AR10 has two reasons behind it: to achieve CommPerformance and to satisfy Rq_AckProcessing. Grasp uses the keyword *supports* to relate reasons to QAs and requirement references. The reason for rationale AR13, on the other hand, is an expression that relates to an AE referenced by parameter M. This expression checks whether the set of properties of M includes property AsyncComm using the *subsetof* operator. Note that the AE passed as parameter M should exist in the namespace of the context in which AR13 is evaluated. The remaining two rationales are declared in a similar manner.

4.2 Modelling System Structure

For specifying the static runtime structure of an architecture, Grasp follows the popular components and connectors paradigm [11,14]. In addition to these primitives, Grasp also supports layer, interface, link and check elements as its other building blocks. An abstract reusable construct called *template* helps to define composite structures from which components and connectors are instantiated.

The chosen example consists of only components. A component roughly maps to the ≪AE≫ stereotyped class in AREL's UML-based model. A base template AsyncComponent is extended by every other template except AlarmServices. The AsyncComm property, defined in AsyncComponent and inherited by all extending templates, identifies those components that implement the asynchronous design.

The *system* block in the Grasp specification contains a number of *components* that are instantiated from templates. However, the interconnections (i.e. wirings) between these components are not included as this information is missing in the

Listing 1. Grasp specification of a partial EFT system architecture [13].

```
architecture Example {
    requirement Rq_AckProcessing;

    quality_attribute CommPerformance {
        property HandleMultipleBankConnections = true;
        property MaxVolumePercentageFromOneBank = 50;
        property MinTransactionsPerDay = 8000;
    }
    quality_attribute CommReliability {
        property NoLossPaymentProcessing = true;
        property NoDuplicateProcessing = true;
    }

    @(Desc="OptimalMsgProcPerformance")
    rationale AR10() {
        reason supports CommPerformance;
        reason supports Rq_AckProcessing;
    }

    @(Desc="ProcessingSequence")
    rationale AR13(M) {
        reason [AsyncComm] subsetof M.properties();
    }

    @(Desc="NoLossTransaction")
    rationale AR14(M) {
        reason supports CommReliability;
        reason [AsyncComm] subsetof M.properties();
    }

    @(Desc="TimeOutMechanism")
    rationale AR15(M) {
        reason [AsyncComm] subsetof M.properties();
    }

    template AsyncComponent() {
        property AsyncComm;
    }
    template AsyncMsgProc() extends AsyncComponent {}
    template AsyncMCPDrv() extends AsyncComponent {}
    template AsyncErrDet() extends AsyncComponent {}
    template AsyncErrRec() extends AsyncComponent {}
    template AlarmSvcs() {}

    system PaymentGateway {
        component MsgProc = AsyncMsgProc() because AR10;
        component MCPDrv = AsyncMCPDrv() because AR13(MsgProc);
        component ErrDet = AsyncErrDet() because AR14(MsgProc);
        component ErrRec = AsyncErrRec() because AR14(MsgProc);
        component Alarm = AlarmSvcs() because AR15(ErrDet);
    }
}
```

original case study. Grasp provides a *link* construct to specify wiring among components.

4.3 Binding Rationale to Architecture Elements

The next step in building a Grasp specification is to associate rationale to various elements in the architecture. As shown in Listing. 1, each *component* instance has a *because* clause that attaches a previously declared rationale to that component. Some rationales in this example accept arguments that refer to other components within the same namespace. A rationale can be attached in a similar manner to any type of AE including templates. A rationale associated with a template is inherited by a component or connector instantiating that template.

This example also illustrates the dependency graph that forms with AEs and rationales in a Grasp model. For instance, the Alarm component is bound to rationale AR15, which in turn is tied to ErrDetect through its reason. ErrDetect is associated with AR14, which depends on MsgPro, which depends on AR10.

4.4 Evaluating Rationale

A rationale is evaluated in the context of the AE to which it is attached. The application context of a rationale is an important aspect in this model. A rationale cannot be evaluated as a free-standing entity and therefore must be associated with at least one AE for this purpose. At the same time, not all AEs associated with a given rationale may satisfy that rationale. Hence, a rationale is evaluated within the context of each AE it motivates, independent of other associated AEs.

In our example, rationale AR10 is tested within the context of MsgProc, AR13 within the context of MCPDriver, AR14 within both ErrDetect and ErrRecovery, and AR15 within Alarm. In the case of rationale AR14, which attaches to two components, it may possibly pass with one component and fail with the other.

4.5 Traceability

The usefulness of a rationale model largely depends on its ability to trace dependencies between rationale and AEs. Such a model should be able to provide *forward tracing* and *backward tracing* [13]. Forward tracing allows a change to a given rationale or an AE to be traced downwards through the graph to every other rationale and AE affected by that change. Thus it facilitates impact analysis during design modification, offering the means to understand the effects of a change prior to its implementation. Backward tracing assists the discovery of factors that affect a given AE by tracking upwards through the graph. It helps developers understand the justifications for design outcomes along with sensitive points in the architecture. An AE that traces back to a large number of rationales can be treated as highly sensitive to changes in the architecture.

Figure 2(a) illustrates a forward trace that starts from CommPerformance. This trace will essentially expose the impact on the architecture if any of the performance criteria given in CommPerformance were to change. As shown, rationale AR10 is directly affected, which in turn affects MsgProc. The impact

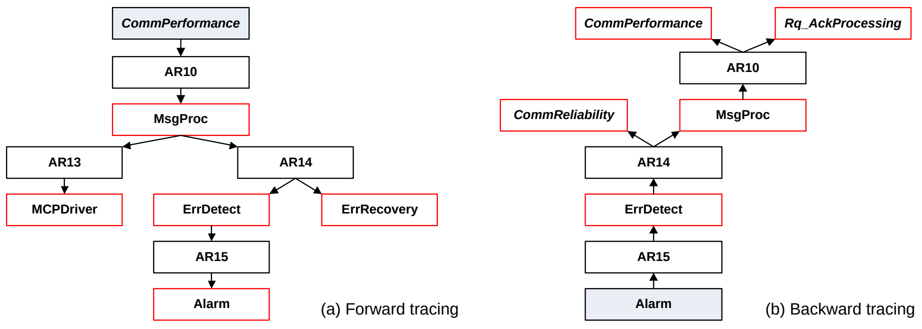

Fig. 2. Traceability in the Grasp model.

continues to propagate down the dependency graph affecting AR13, AR14 and beyond. As it stands, all five components in the example architecture are affected by a change to the performance quality attribute.

In the complementary backward tracing example in Fig. 2(b), the trace starts from Alarm and moves up towards the root of the graph looking for nodes that have the potential to affect Alarm. This diagram shows that component Alarm is affected by a change to components ErrDetect or MsgProc, QAs CommReliability or CommPerformance, or requirement Rq_AckProcessing. Backward tracing also provides a clear view of the justification behind a design outcome.

5 Implementation Status

The rationale model described in this paper has been defined and incorporated into the Grasp language. A Grasp compiler, developed using the ANTLR parser generator framework [8], is currently available. Work on tools to evaluate rationale, check consistency properties and visualise the architecture is in progress.

6 Conclusions and Future Work

The importance of capturing rationale as part of an architecture has been widely discussed. However, the adoption of rationale techniques has been weak. The Grasp ADL attempts to address this problem with a simple rationale specification mechanism coupled with strong tool support for architectural design.

An extension to the Grasp model to capture alternative architectural decisions and their rationales is also being investigated. An important consideration here is the tradeoff between the usefulness of recording design alternatives and the simplicity of the rationale model essential for encouraging adoption. Furthermore, we are currently extending Grasp to support dynamic architectural properties, enabling the evaluation of architecture rationale during system execution. This work is the first step of a larger agenda to control architecture erosion by maintaining the correspondence between architecture and implementation in all relevant aspects including structure, behaviour and rationale.

Another vital concern is the evaluation of the effectiveness of the rationale model in addressing the above-mentioned issues. Building architecture models of existing software and comparing system evolution with and without the Grasp framework will enable us to carry out this evaluation. Grasp is currently being used to capture the software architectures of constraint solvers as part of an EPSRC-funded project [1]. The architecture specifications from this project will provide some of the required case studies.

It is possible to consider alternative mechanisms, such as separate views of structure, behaviour and rationale of an architecture that are somehow linked to enable traceability among them. Given the need to link architecture and implementation in order to minimise erosion, our approach aims to simplify the process and combine these aspects in a single representation with visualisation tools providing suitable abstractions for users.

Acknowledgment

This work is supported through a PhD studentship awarded by Scottish Informatics and Computer Science Alliance (SICSA) and University of St Andrews.

References

1. Balasubramaniam, D., de Silva, L., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Dominion: An architecture-driven approach to generating efficient constraint solvers. In: Proc. of the 9th Working IEEE/IFIP Conference on Software Architecture. p. 4 (2011)
2. Bosch, J.: Software architecture: The next step. In: Proc. of the 1st European Workshop on Software Architecture. pp. 194–199 (2003)
3. Burge, J.E., Brown, D.C.: SEURAT: integrated rationale management. In: Proc. of the 30th International Conference on Software Engineering. pp. 835–838 (2008)
4. Carignano, M.C., Gonnet, S., Leone, H.P.: A model to represent architectural design rationale. In: Proc. of WICSA/ECSA 2009. pp. 301–304 (2009)
5. ISO/IEC/IEEE: ISO/IEC 42010: Systems and software engineering – Architecture description. ISO/IEEE (2009), (Draft: ISO/IEC WD4 42010)
6. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Proc. of the 5th Working IEEE/IFIP Conference on Software Architecture. pp. 109–120 (2005)
7. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
8. Parr, T.: ANTLR Parser Generator. `http://www.antlr.org/` (2011)
9. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
10. Savolainen, J., Kuusela, J.: Framework for goal driven system design. In: Proc. of the 26th International Computer Software and Applications Conference. pp. 749–756 (2002)
11. Shaw, M., Garlan, D.: Software Architecture: Perspective of an Emerging Discipline. Prentice Hall (1996)
12. Tang, A., Babar, M.A., Gorton, I., Han, J.: A survey of architecture design rationale. Journal of Systems and Software 79(12), 1792–1804 (2006)
13. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80(6), 918–934 (2007)
14. Taylor, R., Medvidovic, N., Dashofy, E.: Software Architecture: Foundations, Theory, and Practice. Wiley (2009)
15. Tyree, J., Akerman, A.: Architecture decisions: Demystifying architecture. IEEE Software 22(2), 19–27 (2005)
16. Zhu, L., Gorton, I.: UML profiles for design decisions and non-functional requirements. In: Proc. of the 2nd Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent. p. 8 (2007)