Change Analysis in an Architectural Model: A Design Rationale Based Approach

Prasanta Bose
Information and Software Engineering Dept.
George Mason University
Fairfax, VA
bose@isse.gmu.edu

ABSTRACT
The architectural model of a system provides a high level description that enables compositional design and analysis of component-based systems. This paper focuses on the problem of evaluation and management of change in an architectural model of the system and presents a design rationale (DR) based approach to such a problem. We consider the types of design rationale knowledge relevant to architectural decisions and describe an initial prototype environment for capture and usage of such knowledge in analysis of change.

Keywords
Architectural changes, design rationale, scope analysis

1. INTRODUCTION
Recent developments in component-based standards and technologies (COM, DCOM, CORBA) have led to the increase in the development of component-based software systems [10]. A major problem in component-based system is handling evolution – how to ensure the integrity of the system as new components and connections are added to or deleted from the system (statically or dynamically) or existing components and interactions are evolved. The architectural model of the system provides a high level of abstraction to the different stakeholders of the system for making extension, customization, and evolution decisions. Recent research has focused on extensions to the ADLs to capture dynamic changes in the architecture [2, 7]. The focus of this paper is on addressing the problem of evolution of component-based software systems. In particular we focus on design-time evolution of the system specified in terms of changes in the architectural model of the system.

1.1 The Problem
The two major problems that arise in considering change of the architectural model are a) Evaluating the change to determine what properties are affected, what mismatches and inconsistencies result, and b) Management of change. The second problem arises when the architecture model is decomposed into model fragments where the model fragments may also be developed and maintained in a distributed environment. The management problem is how to coordinate and manage the changes in the different model-fragments or packages that ensure protection of global properties. We consider the following types of changes that can occur in an architectural design and analysis environment: a) addition/deletion of new components, b) addition of new components that refine or update existing components, and c) reconfiguration and/or extensions of the application architecture in terms of addition or deletion of connections and component bindings to those connectors.

There are two sub-problems in handling change: i) Scope determination. Determining the context relevant to the change. The context identifies the design elements and properties that are impacted, new issues raised and decisions that must be reconsidered. ii) Change integration. Determining what else must change to accommodate the change. This paper focuses primarily on the scope determination problem.

2. Approach
Our approach to the scope determination sub-problem involves using the design rationale (DR) dependency knowledge to determine scope via reachability analysis. A design rationale may capture different kinds of knowledge such as knowledge that justify or validate choice of specific type of connections between components over other choices, the rationale may expose the tacit assumptions about the environment and provide explanation of how certain properties and behaviors are realized. The key idea underlying our approach to evaluation and management of change builds on the concept of using design rationale knowledge to capture dependencies between architectural decisions. The dependencies between decisions are used to propagate change, visualize the effect of change and
determine the scope of change. The result of the scope analysis is a set of design elements and properties that are reachable from the change. Once the scope is determined, we can then re-evaluate the properties that are possibly affected using tools relevant to the properties.

2.1 Types of Design Rationale

We consider the following forms of design rationale knowledge in an architectural model that is relevant to the problem of change evaluation and management:

**Requirement Dependency (R<sub>DR</sub>).** Captures required behavioral dependency relationships between individual component behaviors and specifications and how such dependency relationships are realized by the elements in the architectural model. The DR acts to validate the model and the design decisions in particular against the requirements. We consider the DR for two types of behavior dependency relationship: a) **Causal dependency** – where behavior in one component or subsystem causes a behavior in another component or subsystem. An example of such a relation in an example of an automated gas pump system composed of customer, pump and cashier components having asynchronous behaviors, is the requirement that a customer payment lead to the customer pumping gas from a pump. The design rationale captures the connection specification specifying the connection between the customer and cashier and the connection between the cashier and the pump that ensures such causal relationship between the behaviors. b) **Ordering dependency** – where a specific ordering relation has to be maintained between two or more component behaviors or subsystem behaviors and the behaviors themselves may be individual requirement specifications. An example of such a relation is the requirement of the customer paying for gas before pumping gas in an automated gas pump system. The DR captures the requirement specification (at the application level) and dependency on the connector coordination (or glue) behavior, connection role behavior and component port behavior.

This type of knowledge provides a causal (mechanistic) explanation of the property in terms of the component behaviors and connections that establish the property. The DR is generative in nature since 1) it can be constructed from an underlying semantic representation of the components and connections (in some ADL) constituting the model using analysis or simulation tools, and 2) it defines a finite model (similar in spirit to that of a finite state machine model) which generates the behavior in question or generates behaviors that satisfy the property in question. The constructed DR also provides validation of the architectural model against specified properties and may be captured in the process of model validation.

**Inconsistency Dependency (I<sub>DR</sub>).** Here the DR captures the inconsistency or conflict arising in the context of one or more dependency relationship (required or designed) between components and the architectural decisions that address the inconsistency or conflict. The architectural decisions are typically refinements or revisions to the component or connection model elements or addition of new components and connection (note, the new components may be wrapper interfaces or proxies customized to handle the interactions). The DR provides justification for such refinements or revisions or new components. The following are some of the major types of architectural inconsistencies that characterize this DR: a) **Interface mismatch.** Here the components participating in a dependency relation have incompatible interfaces. For example, in a net-based ordering and payment system for electronic goods, the security interface of the buyer and seller components may not be compatible in the context of a required access-control relationship between the buyer and the seller components. The DR captures this mismatch context as justification for the introduction of a mediator component that addresses the incompatibility. b) **Incongruent properties.** Here the properties or invariants characterizing the dependency relations may be inconsistent. For example, in a client-server architecture for web-based stock prices monitoring – the solution level invariant of client-pull only is inconsistent in the context of a dependency in the application architecture requiring the server application component (the stocks prices database) to notify the user client of changes in the database. The DR captures this property inconsistency and the introduction of the server proxy component that refines the client component to address the violation.

2.2 Change Evaluation: Scope Determination

Scope determination is the first step and plays an important role in reducing the complexity of the evaluation problem, particularly for large-scale systems. Moreover the scope information provides useful knowledge on the possible types of issues that may arise from a change and hence focus the change integration and negotiation process to handle such issues. Different types of changes require different scope determination methods that exploit the R<sub>DR</sub> or the I<sub>DR</sub> or a combination of both. We consider below the usage of DR in evaluating change involving the addition of a component to the system.
2.2.1 Addition of a component

Addition of a component model ($C_{\text{new}}$) to the architectural model of the system lead to addition of new constraints and association relationships that specify the binding of the component to the existing components and connections. The method for determining the scope of such changes involve the following sub-steps and produces the set of components/connections and relevant properties that have to be re-evaluated: 1) Determine the directly related required properties ($P_i$) that involve the component ($C_{\text{new}}$). This step requires classification reasoning (can be semi-automated). 2) Use $R_{DR}$ to find all properties ($P_d$) that are related to $P_i$ and their designs that get impacted. 3) Use $I_{DR}$ to determine all inconsistencies that need to be revisited in the context of $P_i$ and/or $P_d$ properties, and 4) Determine the inconsistencies that are handled by components of type $C_{\text{new}}$.

Let’s consider an example of component addition and subsequent evaluation in the evolution context of an electronic buy-&-sell system. We limit our discussion to the application level architecture. The initial system consists of a buyer and a seller component and connectors for ordering ($C_{\text{ord}}$) and invoicing ($C_{\text{inv}}$) at the application level. The $R_{DR}$ for the dependency relation ($R_{\text{authdep}}$) that the buyer gets authorization to use product only after he has authorized payment by charge captures the ordering and the invoicing connectors, the dependency between them and the authorization connector ($C_{\text{auth}}$) that bridges the two connector to satisfy the dependency relation. Extending the system with new functionality, where the buyer can use his banking services to pay for his order leads to adding a new component ($C_{\text{new}}$) to the system that performs banking operations. Without the $R_{DR}$, analysing the above change would require re-evaluation of all existing properties. Given the $R_{DR}$, since the new component being added plays a role in the invoicing connector ($C_{\text{inv}}$) and the dependency relations between $C_{\text{inv}}$ and other connectors are specified by the $R_{DR}$, we can obtain the scope of the change by obtaining the elements related by the $R_{DR}$ in the example.

2.3 Change Management

The major problem of change management involves coordination of change across model-fragment packages (that may also be physically distributed when doing distributed software engineering such that the global integrity of the models is ensured as changes take place locally in each model. The models captured in each package may correspond to a stakeholder perspective or may correspond to an aspect/feature of the system [e.g. performance model, security Model] or correspond to the structural units defining a system composition. We consider a peer-to-peer relationship between the model-fragment packages. That is, model elements (say component/connection) defined local to a package ($M_i$) may get used in another package ($M_x$) and simultaneously package-$x$ may be client for model elements defined in package-$y$. In such a peer-to-peer relationship, the client may impose constraints (i.e. introduce changes) on the server elements or the server may introduce changes to the locally defined elements. In either case, the changes must be propagated. The approach to change management in such a context is based on using the DR to capture the dependencies between the packages and using the DR to coordinate change occurring in the packages.

Figure 1: The model fragment packages and the RDR view for the gas-pump example.

Let’s consider the following example in the context of an automated gas-pump example. Consider a model fragment package $M_y$ capturing gas-dispensing and $M_y$ capturing manual gas-payment. Moreover, let’s assume that $M_x$ locally defines the model for pump and customer and uses the component model for cashier that is defined in package $M_x$. The package $M_y$ defines the model for cashier and uses the customer model defined in package $M_x$. The problem is coordinating change across $M_y$ and $M_x$ that ensures satisfaction of the global ordering constraint - pay-before-pump property ($P_1$) or the property that if the customer has paid then he gets gas ($P_2$) [6]. Given evolution in $M_y$ from manual payment to automated payment at the pump, the problem is coordinating the change in the two model fragment packages such that $P_1$ and $P_2$ are preserved. We use the example to explain the approach taken to address the problem.

Consider the use of the DR ($R_{DR}$) knowledge that captures the dependency constraint between taking-gas connection in $M_x$ and paying-for-gas connection in $M_y$ with solution based on cashier component behavior that mediates the two connection and ensures satisfaction of the ordering constraint $P_1$. Figure 1 informally shows the model fragment packages and the $R_{DR}$ knowledge captured (in UML). The $R_{DR}$ in essence defines a view (as shown in Figure 1) which uses elements defined in $M_x$ and $M_y$ and...
captures the dependency constraint (P₁) between Mₓ and Mᵧ. Change in the model-fragment package, Mₓ (addition of the pump role to the connection) triggers change in the DR view. The change in the view is propagated using the dependencies captured by the DR to determine what else must change (the scope) and the required changes to ensure satisfaction of the constraint defined by P₁. The resulting changes that pertain to elements defined in other model-fragment packages are propagated to those packages. Continuing with the gas-pump example, such propagation and change in the view to preserve property P₁ would consist of changes in the pump behavior defined in package Mₓ. It is to be noted that computing the changes that are necessary to integrate the required changes may require use of tools that infer the new set of values to be propagated. In the above approach, the DR acts to negotiate the changes across model fragments, where the negotiation is driven by global properties.

3. Rationale Capture and Usage Environment
We have done an initial prototype of the change evaluation and management concepts in the context of the WinWin process model and support for collaborative software engineering [1]. The WinWin artifact dependency structure provides a means to capture the design rationale that gets interpreted by the stakeholder. A major problem with the WinWin artifact meta-model and the associated support tool is lack of conceptual integration with some ADL (current efforts are being made to integrate with architecture artifacts via the attachment mechanism) and hence corresponding integration with the associated ADL support environment. Such conceptual and practical integration is necessary in order to exploit the design rationale captured by the WinWin artifact dependencies and develop automated tools that support change evaluation and management.

We have been experimenting with an extended WinWin artifact model and support tools that is based on the conceptual integration of WinWin artifact meta-model with an extended ACME [3] like ADL. We use UML [11] (stereotype) constructs to represent the extended model elements (similar in spirit to the work in [8]). The conceptual integration is based on an interpretation mapping of the WinWin artifact meta-model elements to elements defining the architectural model. The prototype is implemented using extensions to the Rose support tool [9]. Current support for change analysis and change management is limited to scope determination via what-if analysis. The tool generates the visibility graph in response to a change in one of the model-elements.

4. Related Work
The work described in this paper is related to the work in the areas of design rationale (DR) and software architectures. Much of the research on design rationale [4] have focused on design rationale capture external to the context of software design where such rationale is generated (as part of the design and/or modeling activities) and gets used. Due to the detachment from the actual modeling and design context - the captured rationale is not useful for change analysis. The software architecture work has been primarily focused on ADLs and tools for analysis and simulation of models captured by the models [5,6]. There has been more recent work on dynamic architectures modeling, run-time support for architectural changes[7]. The work described here is focused on design time changes.

ACKNOWLEDGMENTS
The author thanks USC-CSE for providing support for this work.

5. REFERENCES