Formalizing 'Traceability' for Architectural Evolutions

Liang-Jie Zhang, Vishal Dwivedi*, and Nianjun Zhou
*Institute for Software Research, Carnegie Mellon University, USA
Email: vdwivedi@cs.cmu.edu
IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA
E-mail: {zhanglj, jzhou}@us.ibm.com

Abstract

Software architectures evolve over time, and so do the models that represent them. For a domain like Service Oriented Architecture (SOA) this is particularly true because most SOA solution designs are based on modification of existing assets that change over time. However, today there exists only limited work that reasons about this evolution. In this work we present our framework for traceability of evolving architectures that we apply for SOA solution design. Our design approach is based on an iterative process that utilizes a set of solution patterns to guide architects in the SOA solution design. Our approach utilizes historical data about pattern enablement and uses that to guide architects in selecting the right patterns. To ensure that the right patterns are used, we use a template matching approach that enforces conformance by allowing only the right set of artifacts to be composed together. We demonstrate how our framework can be applied to compose and trace evolving SOA solutions based on three views – the artifact view, profile view and compliance view.

1. Introduction

Software Architectures have evolved over time as a meaningful way of abstracting system properties, and thus reasoning about prospective quality attributes, design alternatives and their documentation. However, today's architectures must be planned with their evolution in consideration – a process that requires objective analysis and skilled design by architects, who have to ensure business and quality goals over the lifetime [1] of the systems they design. Planning and engineering such evolutions is not only necessary, but almost critical for domains like Service Oriented Architectures (SOAs) where the architectural elements have high dependency on external elements. In this work, we outline our approach towards a planned and traceable evolution in the services domain.

Service Oriented Architectures have been widely adopted as an architectural framework for creating reusable components. However, the design of SOAs differs from other traditional architectures in many ways. These systems tend to be large, and hence the design activity usually tends to be collaborative, based on multi-phase process-based approaches such as SOMA [2] or INSOAP [3]. Apart from composing services, architects need to take care of various design aspects of service oriented architectures. For instance, most SOA design approaches in practice need to consider domain specific design features like granularity of services [4], goal-satisfaction [2], quality attributes and other design elements. All this ensures that the SOA design is never a one shot process. The architecture evolves over time, and the evolution is not only in terms of completeness, but also maturity and design quality. Architects need an approach to trace such evolutions, but current engineering methods and tools provide limited support for such planning and reasoning.

In this paper we propose a framework for modeling and planning traceability of evolving SOA based architectures. We provide a set of SOA solution patterns to guide architects through the process of consuming and configuring SOA elements for the design of SOA solutions. We leveraged three views – the artifact view, profile view and compliance view [8], to illustrate artifact's design history, its composition, and the adherence to guidelines respectively. Not only these views provide support for design, but architects can also trace the evolution of design through these views.

Kindly note, that unlike traditional component-oriented software architecture descriptions that use components and connectors as key design elements, we primarily use artifacts and relationships to design SOAs. We do so, because the granularity of our designs are somewhere between business-architectures and IT architectures – a relation described by Martin Assmann et al [5] in his work. In our SOMA-ME [6] based SOA modeling we deal with a wide variety of design elements ranging from business functions to web-services that are critical to SOA design.

2. Problem Statement

As software's evolve, the best way to plan their evolution is through the evolution of their architecture. However, planning for such architectural evolutions is done in an ad hoc manner today, with almost no tooling support and very limited theoretical foundation.

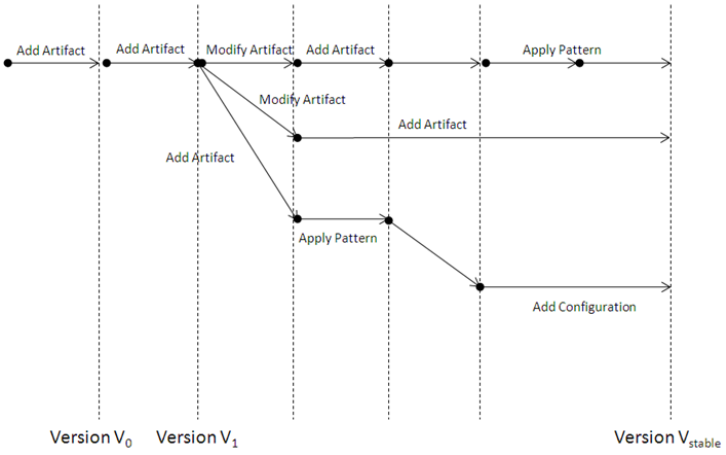


Figure 1: Tracing the evolution of Solution Architecture

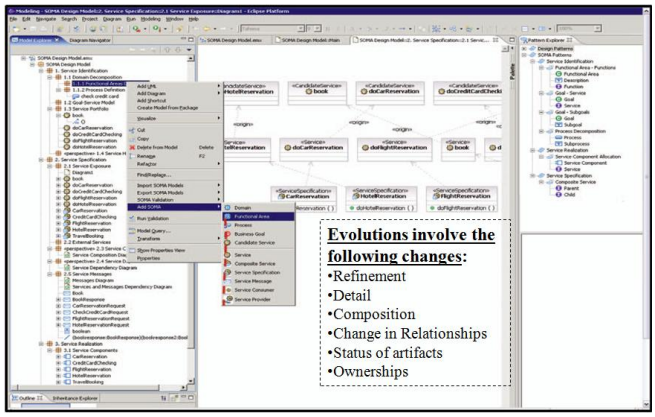


Figure 2: Evolutions in SOMA-ME based models

We have been exploring ways to streamline such adhoc evolutions through guided processes, and pre-existing solution patterns that provide design decisions made earlier in the same scenario. A more principled approach to design such evolutions can not only ensure traceability, but also allow better reasoning about design quality.

In particular, this paper addresses the following questions:

- How do we enable such an architectural evolution through well defined repeatable patterns?
- How do we trace an architectural evolution to provide guarantees about design quality?
- Can we provide a theoretical framework that can be used for tool support for designing such evolving architectures?

The goal of this paper is to provide a theoretical framework for modeling and planning traceability of evolving SOA based architectures. We primarily rely on SOMA based solution design, but we adopt an abstraction based approach to simplify our modeling and formalization. Towards the end, we describe the tool-support for validating our approach.

3. Preliminaries

The domain-specific meta-model and the model elements of our SOA model are detailed in the SOMA-ME article [6]. However, in this section we would formalize the descriptions by abstract representations. A formal definition of the component system provides a foundation for further extension and analysis. We would extend this model by refinement to demonstrate our modeling approach.

Our SOA solution model consists of various A*rtifacts*, each of which has unique properties. These artifacts are connected together via *Relationships* that describe the relationship between the artifacts. Designers combine these artifacts to form *Solution-models*. In this section we formalize these elements.

3.1 Artifacts

Artifacts are defined as architectural building blocks for service oriented architecture. They may be computational or functional entities, or they may describe certain architecturally relevant metrics. Each of these artifacts has a collection of attributes, whose value is instantiated by designers while composing a solution.

[*ArtifactName, AttributeName*]

Artifact

name: *ArtifactName*

attributes: $\mathbb{F}$ *AttributeName*

At any point the Solution model has a set of uniquely named Artifacts defined above, and at any instance an AbstractSyntax defines an assembly of some unique artifacts, where uniqueness is interpreted as artifacts having unique names and values for their attributes.

Abstract Syntax

artifacts: $\mathbb{F}$ *Artifact*

$\forall\, c_1, c_2 : artifacts \mid c_1 \neq c_2 \bullet c_1.\, name \neq c_2.name$

$\wedge\, c_1.\, attributeValues \neq c_2.\, attributeValues$

3.2 Relationships

Relationships define the interaction between artifacts. The types of relationships vary with respect to the kinds of interactions between the artifacts. The relationship types

include basic types such as *Association, Generalization, Implementation, Composition, Support* and temporal orderings such as *pre* and *post* processing. These relationships define the semantic ties that exist between the artifacts.

3.3 The SOA Solution model

The SOA Solution model is a collection of architectural elements. Each such element is an instance of the type *"Artifact"* that can be used by a designer to model SOA solutions.

[AttributeValue, SOA-Element]

SOA−Element

type: *ArtifactName*

name: *ElementName*

attributes: *AttributeName* $\twoheadrightarrow$ *AttributeValue*

The designers combine a finite set of SOA-Elements to form a SOA solution model. Each of these SOA-Elements is a unique instance of an Artifact type.

SOA−Solution−Model

elements: $\mathbb{F}$ *SOA−Element*

$\forall e_1, e_2 : elements \mid e_1 \neq e_2 \bullet e_1.name \neq e_2.name$

$\wedge e_1.attributeValues \neq e_2.attributeValues$

The above description of artifacts and their composition is purely syntactic in nature. One of the reasons for this is that UML is mostly free form and any semantic restriction has to be enforced via constraints. For the sake of simplicity, we model an "Oracle" that maps the syntactic descriptions of the solution model to their semantic meaning. We assume that this Oracle provides a relation that maps the possible values of attributes to their values. We assume that there is a relation *valueOf ()* that provides this value by mapping the types to the possible allowed values:

valueof(): *AttributeType* $\rightarrow$ *AttributeValue*

The Oracle provides a function that can check if the attributes of the SOA-Element have values that are permitted by the ArtifactType.

$\mathcal{M}_{artifact}$: *Artifact* $\rightarrow \mathbb{P}$ *SOA−Element*

$\forall c$: *Artifact* $\bullet \mathcal{M}_{artifact}(c) \subseteq \{e$: *SOA−Element*

$\mid e.type = c.name \wedge$

$e.attributes = \{a$: *c.attributes*; *v*: *AttributeValues*

$\mid valueOfRelationship\ (v) = a\}\ \}$

The Oracle provides another function that describes how an *AbstractSyntax can be defined as a* set of possible system models derived from the meaning of the abstract model.

$\mathcal{M}_{syntax}$: *AbstractSyntax* $\rightarrow \mathbb{P}$ *SOA−Solution−Model*

$\forall a$: *AbstractSyntax* $\bullet \mathcal{M}_{syntax}(a) =$

{soaModel : SOA−Solution−Model

$\mid soaModel.elements = \bigcup \{c: a.artifacts \bullet \mathcal{M}_{artifact}(c)\}\ \}$

This Oracle is implemented in SOMA-ME through various configuration files that automatically define the values for artifact compositions and its properties that are type-checked for correctness. Since the SOMA-ME uses a UML meta-model with a weekly typed artifact relationships based composition, such a type-checking enforces the correctness of the models.

4. Traceable Architecture Evolution

Designing SOA solutions involves multiple phases such as shown in Figure 3. Our SOA modeling approach is based on a generic SOA design approach [18], which is a multi-step design process. Throughout the design phase, the architects are guided by multiple domain-specific patterns, transformations and other previously used design decisions. The architects can trace these decisions and the evolution of the architecture through multiple views that describe this historical information.

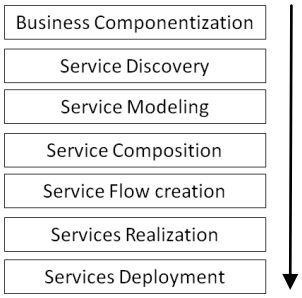


Figure 3: Phases of SOA solution design

This section provides a formal description of our approach that we utilize for tool support.

4.1. Formalizing traceability

We model the evolution of the solution architecture as a *forest of version trees*. Here each node is a graph representing the solution architecture at a particular time point. We formally define it as:

VForest [*Type*]

nodes: $\mathbb{P}$ *VLabel*

dref: *VLabel* $\rightarrow\!\!\!\!\!+$ *Type*

parent: *VLabel* $\rightarrow\!\!\!\!\!+$ *VLabel*

roots: $\mathbb{P}$ *VLabel*

In the above, *VLabel* represents version labels, the mapping *dref* dereferences version labels, and the mapping parent returns the parent of non-root nodes in the forest. We formalize the SOA solution as a collection of architectural artifacts. Each artifact is identified by a label of type *AId*. A specific solution artifact can be identified by its version label,

AVersion AId × VLabel

A collection of specific artifact versions is thus modeled by the type Aid $\rightarrow\!\!\!\!\!+$ VLabel

The collection of changes to an artifact is modeled as follows:

AChange ::=
add⟪AVersion⟫
| *delete*⟪AVersion⟫
| *modify* ⟪ AId × VLabel × VLabel ⟫
| *combine*⟪Aversion × AVersion$^+$⟫
| *replace*⟪ Aversion$^+$ × AVersion⟫

The following defines a predicate for checking whether a change affects a given artifact version:

isChangedBy : *Aversion* $\leftrightarrow$ *AChange*

$\sim$*isChangedBy*(*av′*, *add*(*av*))

isChangedBy(*av′*, *delete* (*av*)) $\Leftrightarrow$ *av′* = *av*

isChangedBy(*av′*, *modify* (*a*, *u*, *v*)) $\Leftrightarrow$ *av′* = (*a*,*u*)

isChangedBy(*av′*, *combine* (*avs*, *av*)) $\Leftrightarrow$ *av′* $\in$ *ran avs*

isChangedBy(*av′*, *replace* (*avs* , *av*)) $\Leftrightarrow$ *av′* $\in$ *ran avs*

The Abstract SOA Solution
A SOA solution is modeled as a collection of architectural artifacts. In the temporal space, the version of the Solution is modeled as a collection of artifact versions along with the delta set of changes since the last version.

Abstract Solution

artifacts: *Aid* $\rightarrow\!\!\!\!\!+$ *VLabel*

delta: $\mathbb{P}$ *AChange*

We model the history of the solution composition as a pair consisting of the collection of artifacts and a collection of versions of the solution at different time points. This historical information is stored in the database for future reference.

ModelHistory

artifacts: *Aid* $\rightarrow\!\!\!\!\!\!+$ *VForest* [*Artifacts*]

solutionModels : *VForest* [*SOA solution models*]

4.2 Tracing the evolution of the SOA Solution Architecture

In this section we formally traceability in terms of the evolution of artifacts for the composition of the SOA solution. The following function defines the set of artifacts that result from a change in a particular artifact:

directChange: *Aversion* ×*AChange* $\rightarrow\!\!\!\!\!+$ $\mathbb{P}$ *AVersion*

---

dom directChange = *isChangedBy*

directChange(*av*, *delete*(*av*)) = Ø

i$\in$ 1..*n* $\Rightarrow$ *directChange*(*avs*, *combine* (*avs*, *av*)) = {*av*}

i$\in$ 1..*n* $\Rightarrow$ *directChange*(*avs*, *replace* (*avs*, *av*)) = {*av*}

directChange((*a*,*u*), *modify*(*a*,*u*,*v*))= {(*a*,*v*)}

The function *forwardTrace* defines the indirect change, i.e., that change downstream in the solution model.

forwardTrace: *Aversion* × *ModelHistory* $\rightarrow\!\!\!\!\!+$ $\mathbb{P}$ *AChange*

---

forwardTrace (*av*, *solutionModels*) =

if $\exists$ *V* $\in$ *solutionModels.nodes*;

c $\in$ *solutionModels.dref*(*V*).*delta* • *isChangedBy* (*av*,*c*)

then {*c*} $\cup$ $\cup$

{*av′* : *directChange*(*av*,*c*) • *forwardTrace*(*av′*, *solutionModels*)}

else Ø

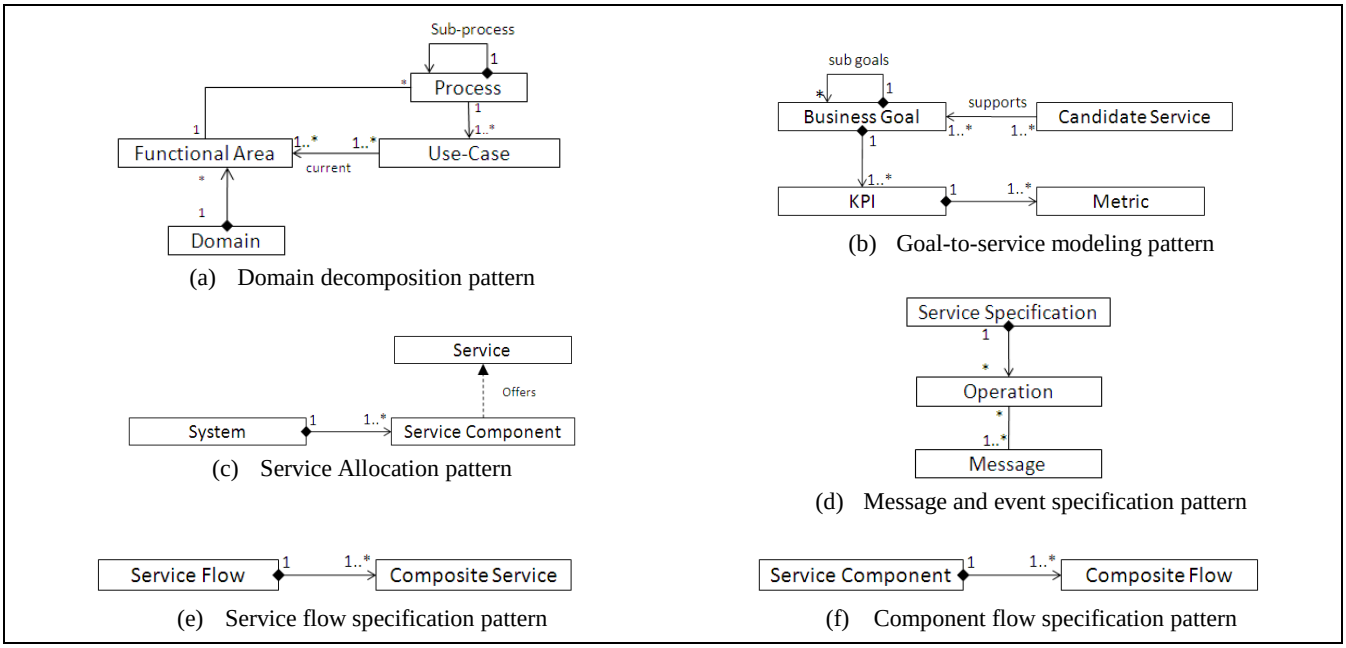


Figure 4: Design Patterns used in our SOA solution model

5. SOA Solution Composition using Patterns

Our solution design utilizes many domain specific SOA patterns that are applied by architects to refine an existing model. Table 1 lists the solution patterns that we use in SOMA-ME. These patterns act as reusable templates that implement the relationships between SOA artifacts and they enforce pre-defined specifications and rules.

Pattern Type	***Pattern Name***
Identification Patterns	• Goal-to-service modeling pattern • Domain decomposition pattern • Existing asset analysis pattern
Specifications patterns	• Component flow specification pattern • Service flow specification pattern • Message and event specification pattern
Realization patterns	• Service allocation pattern • Service layers pattern

Table1: Solution Patterns

An example of a pattern is the "*message specification pattern*", that provides a template for defining the message types, sources, and message formats. The message types include input and output message types and sources are files (e.g. XML schema files) or URLs. The message formats could be string, integer, or complex type. The message specification in this case is done by applying a solution pattern that provides a predefined template for a service that architects use to automatically create placeholders in the solution model with the required attributes provided by the pattern.

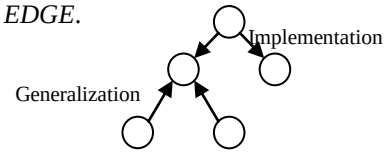
In the next section we would formalize the representation of the SOA patterns for a clear representation.

5.1 Formalizing patterns

We define pattern as a graph that is:

- *Directed*, i.e. for each edge one has a start vertex and an end vertex,
- *Typed*, i.e. vertices and edges are grouped into several distinct classes,
- *Attributed*, i.e. vertices and edges have associated attribute-value pairs to describe additional information about the pattern, and
- *Ordered*, i.e. the edges incident with a particular vertex have a persistent ordering.

The pattern has nodes of types *Artifact* and edges of type *EDGE*.



ELEMENT::= vertex <<ℕ>> | edge <<ℕ>> [ID, LABEL]
VERTEX == ran *vertex* ARTIFACT−ID == ID
EDGE == ran *edge*
EDGE-ID == ID
EDGE: ℙ RELATIONSHIP

RELATIONSHIP = {Association, Generalization, Implementation, Composition, Support}

AttributeInstanceSet == ARTIFACT−ID ⇸ ATTR-VALUE

EdgeInstanceSet == EDGE-ID⇸EDGE-LABLE-VALUE

The *vertexLabel* represents labels assigned to the artifacts through the *AttributeInstanceSet*. The *edgeLabel* represents labels assigned to the edges through the *EdgeInstanceSet*. The relation Λ maps the directionality of the pattern edges.

Pattern

V: vertex ⇸ Artifact

E: $\mathbb{F}$ EDGE

Λ: VERTEX ⇸ seq(EDGE × DIR)

vertexType: vertex⇸ ARTIFACT−ID

vertexLabel: vertex ⇸ AttributeInstanceSet

edgeType: EDGE⇸ EDGE-ID

edgeLabel: vertex ⇸ EdgeInstanceSet

description: DESCRIPTION

$\Lambda \in V \rightarrow iseq(E \times DIR)$

$\forall e \in E \bullet \exists_1 v, w : V \bullet (e,in) \in ran(\Lambda(v))$

$\wedge (e, out) \in ran(\Lambda(w))$

5.1 Creating solution models using patterns

The architects use the solution patterns to model the SOA solution. They begin with an empty solution model, and the application of these patterns refines the model to add details, eventually leading to a complete SOA solution.

The process of creating a SOA solution model is summarized in the following steps:

(a) The first step is to capture business requirements, where they define the environment within which the solution patterns are created.

(b) The second step is to identify SOA artifacts. The SOA artifacts are computational or functional entities that are combined together to design SOA solutions.

(c) The third step is to identify attributes and constraints for SOA artifacts. The attributes and constraints are associated with specific SOA artifacts to describe the unique features of that artifact. Additional artifacts might be derived and vice versa.

(d) The next step is to derive solution pattern rules. The solution pattern rules are identified to realize the relationships between artifacts and capture the attributes/constraints of specific artifacts.

(e) The architects are guided by the pattern application history to choose appropriate patterns. They can then apply a pattern to the solution model, which refines the solution with additional details. Every legitimate pattern application is also recorded in the database.

Not all patterns can be applied to every architectural artifact, and therefore, we need to select the patterns that are applicable for a given set of artifacts. We check this pattern applicability by a template matching approach, as provided by J. Dong et al [7]. Our template match logic is based on the fact that a template graph *f* can be matched with another graph *g* by computing the cross-correlation between them and computing the degree of match. The formula, *Cross-Correlation (u) = Σ f(x-u) •g(x)* (where f(x) and g(x) are two vectors, x = 1…n, and u is an offset), shows how to calculate such a cross correlation. The larger is this value, the higher is the potential they match. This is further normalized to accommodate large values.

We model the pattern graphs and solution models as attribute vectors and compute the normalized cross correlation values.

$$Cross\text{-}Correlation_{\text{normalized}} = \frac{\Sigma f(x).g(x)}{|f(x)|.|g(x)|}$$

Our earlier work [8] describes our approach for creating such attribute vectors and using similarity scores for the pattern match. We store these normalized *Cross-Correlation* values as historical data-points for individual artifact nodes of the evolution forest. We model such intermediate configurations as a relation that records which versions make legitimate configurations.

CMatrix Aid ↔Aid

Valid ArtifactConfigurations

SolutionHistoryA, *SolutionHistoryB*

conformanceMatrix : *VForest* [*CMatrix*]

correlation: ℙ (*VLabel* × *Vlabel* × *Vlabel*)

Thus Patterns in our approach not only help to compose SOA solutions but they also ensure repeatability of the designs by allowing architects to check the historical decisions, based on which they can change their designs. Although, unlike most common SOA patterns, such as the ones defined by Erl et al [9], our patterns are domain-specific and tied to our SOA modeling approach.

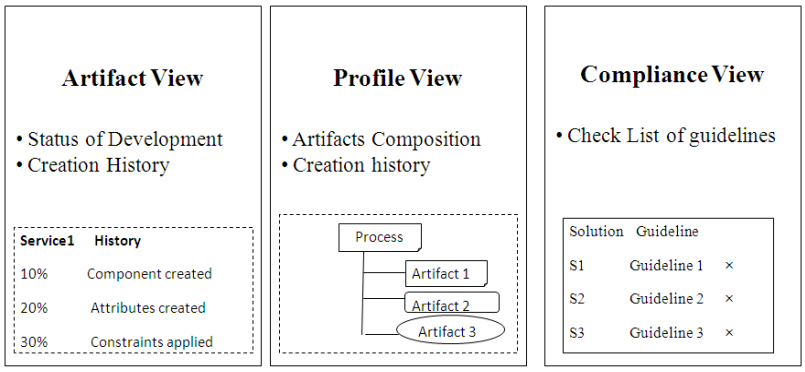


Figure 5: Views for traceability

6. Views to support traceability and tooling

We leveraged three views [8] to allow the architects to trace the evolution of SOA solution design. These views allow the architects to trace the individual artifacts, their composition, their development and if they are used as per the guidelines.

6.1. Views to support traceability

This section describes the three views.

Artifact view

The artifact view provides details of the individual artifacts to the designer. The key details include the status of their development, their creation history that describes the trace of their development. For every architectural artifact, its creation status is recorded as percentage and displayed when a specific artifact is selected. The combination of individual artifact status and creation history view provides a reference for SOA designer to estimate the workload of the remaining design process. Also, an analysis or change propagation path can be traced to reveal to what extent a change affects an artifact under investigation.

Profile view

The profile view provides the details of the artifact's composition and its creation history. This view is mainly utilized to view the composition of processes that displays the corresponding architectural artifacts used to compose it. Along with the information about composition, this view also provides the types of composition, such as whether it is created by (i) applying patterns ii) import other business processes, or (iii) from an existing asset. The cascading profile view provides a traceable path of artifact creation from source to end-product, thus allowing for validation.

Compliance view

The compliance view describes the compliance of SOA solution to a set of guidelines. The compliance of the solution is traced as a checklist based table listing whether an artifact meets the guideline. The guidelines vary from constraints related to usage of artifacts, their dependencies and other project management related constraints. A listing of these constraints enables the designers to avoid mismatches and built correct assemblies.

In summary, these multi-dimensional views provide capability of visualizing detailed information regarding historical usage and constraints for individual artifacts, and thus facilitating the traceability of the evolution path.

6.2. Tooling support

A part of our approach is implemented by the SOMA-ME tool. This tool provides the architects to create SOA solution-models using domain specific patterns. The SOA solution design activity itself comprises of three main steps of *Service Identification*, *Service Specification*, and *Service Realization*, each of which is aided by the patterns described in Table 1. The tooling provides context-aware menus to SOMA patterns, transformations and other analysis with traceability as one of them. Details about the tool are available in [6].

7. Comparison with related work

Our approach can be characterized as model-driven design of Service Oriented Architectures (SOAs) using domain-specific patterns, where the designs evolve over time. This paper formalizes this notion of evolution that we later implement using tool support. The ideas presented in this paper are closely related to three other areas of research: *pattern-based SOA* design, *formal representations of architecture* and *architecture evolutions*.

Pattern based SOA design: Our approach for model-driven SOA design is based on using domain specific patterns for designing SOAs. The modeling elements for our design utilize a meta-model that is instantiated to define concrete solution models. The patterns in this

domain encode reusable design solutions that architects can use to model SOAs. A similar approach was earlier demonstrated by Zdun and Dustdar [10], who used pattern primitives as an intermediate abstraction to formally model the solutions. Likewise, other researchers have used patterns for composing enterprise architectures [11], and pattern detection using similarity scoring [12] to model architectures.

*Formalizing Architecture representations: W*e used Z-notation to formalize the SOA-solution design process. Our formal modeling utilizes the graph based formulation of Zhou et al [15] that they used for modeling SOA solutions. In many ways, this step of the formal design is similar to the formal software engineering technique developed by Garlan et al [13], Booch and Rumbaugh [14] and others. This level of formalization helps to simplify the complex domain-specific meta-models such as the one used in SOMA-ME, and thus allowing for a clear definition of the problems, goals and solutions. The representation in Z-notation also enables us to check for the syntactic correctness and can be used for further verification.

Architecture Evolution: Our model for SOA design is formalized as a continuous evolutionary process that can be traced and utilized for making future design decisions. There has been similar work in the academia towards formalization and tooling of planned architecture evolution. Garlan and colleagues at Carnegie Mellon have created *Æ*vol [16] that helps architects to represent, plan, and analyze software evolutions from an architectural perspective. Medvidovic et al in their work [17] provided an approach for runtime modification of software architectures. In this paper, we do not model runtime evolutions. Instead, our model is based on creation of a repository of historical design decisions and architectural changes that can be traced in future and applied in similar scenarios. Patterns in our world capture the reusable design decisions that can be applied based matching criteria.

8. Conclusions

In this work we presented our pattern based SOA design approach that supports traceability of evolving architectures. The SOA solution design activity itself comprises of three main phases of Service Identification, Service Specification, and Service Realization, each of which is aided by the patterns. Our approach utilizes historical data about pattern enablement and uses that to guide architects in selecting the right patterns. We believe that planning and tracing evolution of architectures can not only help in creating better designs but can also ensure repeatability via tracing historical decisions.

9. References

[1] M. Jazayeri. On Architectural Stability and Evolution. In Proc. of Ada-Europe'02, 2002

[2] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A method for developing service-oriented solutions. IBM Systems Journal, 47(3):377-396, 2008

[3] Abdelkarim Erradi, Sriram Anand, and Naveen N.Kulkarni: SOAF: An Architectural Framework for Service Definition and Realization. IEEE SCC 2006: 151-158

[4] Naveen N. Kulkarni, Vishal Dwivedi: The Role of Service Granularity in a Successful SOA Realization - A Case Study. SERVICES I 2008: 423-430

[5] Martin Assmann, Gregor Engels: Transition to Service-Oriented Enterprise Architecture. ECSA 2008: 346-349

[6] L.-J. Zhang, N. Zhou, Y.-M. Chee, A. Jalaldeen, K. Ponnalagu, R. R. Sindhgatta, A. Arsanjani, and F. Bernardini, "SOMA-ME: A Platform for the Model-Driven Design of SOA Solutions," IBM Systems Journal, vol. 47, no. 3, pp. 397-413, 2008.

[7] Jing Dong, Yongtao Sun, Yajing Zhao: Design pattern detection by template matching. SAC 2008: 765-769

[8] Liang-Jie Zhang, Zhi-Hong Mao, Nianjun Zhou: Design Quality Analytics of Traceability Enablement in Service-Oriented Solution Design Environment. ICWS 2009: 944-951

[9] Thomas Erl, "SOA Patterns", Prentice Hall Publications, 2009.

[10] Uwe Zdun and Schahram Dustdar, "Model-driven and pattern-based integration of process-driven SOA models", Int. J. Business Process Integration and Management, Vol. 2, No. 2, 2007

[11] Sabine Buckl, Alexander M. Ernst, Josef Lankes, Kathrin Schneider, Christian M. Schweda: A Pattern based Approach for constructing Enterprise Architecture Management Information Models. Wirtschaftsinformatik (2) 2007: 145-162

[12] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, Spyros T. Halkidis: Design Pattern Detection Using Similarity Scoring. IEEE Trans. Software Eng. 32(11): 896-909 (2006)

[13] Robert Allen, David Garlan: A Formal Basis for Architectural Connection. ACM Trans. Softw. Eng. Methodol. 6(3): 213-249 (1997)

[14] Grady Booch, Ivar Jacobson and James Rumbaugh, "The Unified Software Development Process". Prentice Hall. ISBN 978-0-201-57169-1.

[15] Nianjun Zhou, Liang-Jie Zhang: A Graph Theory Based Impact and Completion Analysis Framework and Applications for Modeling SOA Solution Components. IEEE SCC (1) 2008: 145-154

[16] David Garlan, Bradley R. Schmerl: *Æ*vol: A tool for defining and planning architecture evolution. ICSE 2009: 591-594

[17] Roshanak Roshandel, André van der Hoek, Marija Mikic-Rakic, Nenad Medvidovic: Mae - a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. 13(2): 240-276 (2004)

[18] Liang-Jie Zhang, Ali Arsanjani, Abdul Allam, Dingding Lu, Yi-Min Chee, Variation-Oriented Analysis for SOA Solution Design, IEEE SCC 2007: 560-568