Pattern-based Customisable Transformations for Style-based Service Architecture Evolution

Aakash Ahmad
Lero - The Irish Software Engineering Research Centre
School of Computing, Dublin City University
Dublin, Ireland
ahmad.aakash@computing.dcu.ie

Claus Pahl
Lero - The Irish Software Engineering Research Centre
School of Computing, Dublin City University
Dublin, Ireland
cpahl@computing.dcu.ie

***Abstract*—Service-based architecture have now become commonplace, creating the need to address their systematic maintenance and evolution. We propose a layered pattern-based transformation framework to support a stepwise and incremental Service-Oriented Architecture (SOA) evolution. The framework enables higher-level abstract and system-level operational transformation of SOA elements to facilitate architectural evolution. Higher-level transformations are defined by combining the basic transformation operators and transformation patterns. An abstraction layer encapsulates these primitive transformations into declarative user-defined transformation rules. SOA-specific architectural styles are applied to refine the transformed design to complete a style-based SOA evolution. An electronic payment system case study is used to demonstrate the architectural evolution at different abstraction levels.**

***Keywords*-Web Services Architecture; Modeling and Transformation; Service Architecture Evolution;**

I. Introduction

Service-Oriented Architecture (SOA) is considered as a business centric, architectural approach to support the design and development for distributed, enterprise systems [8]. The existing theory and practices on such service-based software like [8], [2], [3] primarily focus on its initial design or development efforts. However, as service-based systems are developed and deployed the major concern shifts toward their maintenance and evolution issues [15], [14] to accommodate the changing requirements; thus prolonging the productive life for existing software.

A taxonomy of service-oriented software research have been detailed in [15] that prioritises the current and future research agendas, explicitly highlighting the needs for SOA evolution. In contrast, the current academic and leading industrial efforts like [12], [7], [18] concentrate on legacy modernisation towards service softwares thus lacking-on an explicit evolution within SOA. Alternatively, the work proposed in [17], [23] enables a dynamic design and evolution for service orchestrations to facilitate a runtime adaptation for (web services in) service-based architectures.

With a review into the state of the art (discussed in Section VIII), we believe that rigorous processes, frameworks and patterns etc; are lacking in the afore-mentioned (and other relevant) efforts to support a coherent, stepwise evolution for SOAs. This leads toward the contribution of the proposed research, i.e. *to develop a coherent framework supporting customisable transformation driven evolution for service architectures at different abstraction levels*. In contrast to the relevant existing initiatives like [23], [12] the proposal is technically innovative in that it supports an incremental transformation of SOA elements at different abstraction (structure, design, architecture) levels through:

- An operational layering with a focus on operation and execution (what and how to change) that consists of basic transformation operators and patterns. A pattern notion allows to categorise the composed transformations in terms of their impact on source and target architecture elements.
- A user-defined customisable layer focusing on design aspects (why to evolve) allows a rule-based declarative specification of the transformation goals to generate service design. SOA-specific architectural styles are applied to refine the transformed design in order to complete a style-based service architecture evolution.

In addition, the structural and semantic properties of SOA elements are preserved at all abstraction levels to maintain an overall target architecture integrity. Once a transformation framework is developed, we can systematically address the (structural and behavioral) evolution issues for SOAs.

This paper outlines the proposed research and is structured as follows. Section II details an electronic payment system case study to illustrate the architecture transformation framework presented in Section III. Sections IV and V outline the operational layer (transformation operators and transformation patterns); while the user-defined layer (transformation rules and refinements) are presented in Section VI and VII, respectively. Finally, related work is presented in Section VIII; following the conclusions and outlook in Section IX.

II. Electronic Payment Application Case Study

The case study involves an electronic bill presentment and payment (EBPP) reference model published by NACHA [1].

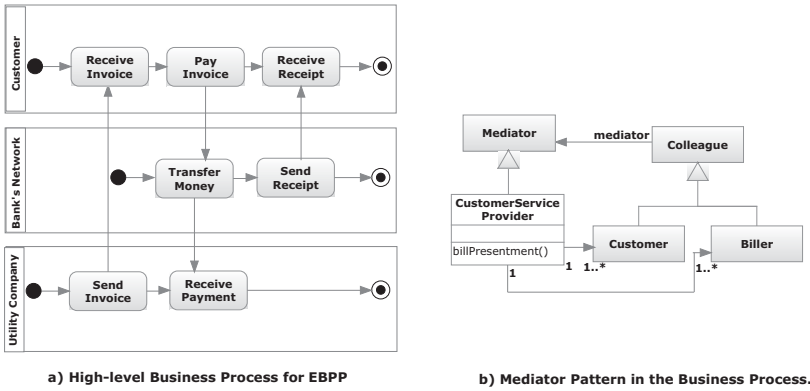

Figure 1. The High-level Business Process and Mediator Pattern for EBPP Case Study.

It represents a typical electronic transaction scenario where the customers and businesses interact illustrated in Figure 1a, as a high level business process for EBPP. Three participants (roles) are exhibited at this level, customer, banks network and utility company. Periodically, a utility company bills their customers with an amount of money corresponding to the consumption of the delivered services. Customers receive their bills and decide the payment. A payment on the date due will eliminate the debt of the customer, otherwise the debt is accumulated. After the payment transaction is completed, the bank's network sends the remittance information to the customer and the biller.

Based on EBPP, it has been realised that a process participant is playing the role of a mediator. Specifically, we identify that a customer service provider (Bank Network) is mediating between customers and the utility company for electronic payment. This relation is analogous to the mediator pattern, illustrated in Figure 1b.

We strive to enable a development based evolution for SOA elements at different abstraction levels. The evolution activities mainly consist of i) transforming the architectural elements to generate a service-oriented design and ii) refining the design by applying the SOA specific architectural styles to evolve the target service architecture. It is vital to preserve the structural and semantic properties of the architectural elements during evolution to maintain a consistent architectural representation at all abstraction levels, detailed in the next section.

III. Service Architecture Transformation Framework

The proposed service architecture transformation framework is illustrated in Figure 2. We use two organisational perspectives to clarify the transformation layers and architectural views.

- The architectural view (in Figure 2b) explains the aspects of an architecture (SOA elements) that can be the concerns of transformation at different abstraction levels. These views are developed using the EBPP application transformation at different abstraction levels.
- Different transactional abstractions (in Figure 2a) allow operational, management and design aspects to be separated during architecture evolution.

A. Structural Level

It describes the operational aspects of the SOA elements (atomic, composite services) in terms of their fundamental structure and relationships in a given service-based architecture. Due to space reasons a minimal structure is presented in Figure 2b based on an extension of the UML 2.0 Profile for Software Services [13], complying to the UML 2.0 meta model.

B. Design Level

It utilises the structural descriptions to instantiate a service-oriented design in terms of a comprehensive architecture of individual services, service functionality and service-level communications. For example in Figure 2b, the *BillCreation service* is communicating with the *Tariff* service through (a mediator) *CustomerServiceProvider* to get the customer billing information, guided by the EBPP case study.

C. Architectural Style Level

It abstracts the design-level details to present a high-level system architecture that comprises of design elements, externally visible properties and high-level relationships in the form of architectural styles and patterns. In Figure 2b, the architecture-style view shows that *CustomerServiceProvider* is refined by applying the Enterprise Service Bus (ESB) style architecture [9] that mediates among the services in the evolved EBPP systems. The motivation behind the style application is to minimise the counter productive transformations to enable a robust style-based target architecture. Note that we use the term 'architectural style' here, but the aim is to encompass what is also commonly referred to as 'architecture/design pattern' detailed in [9].

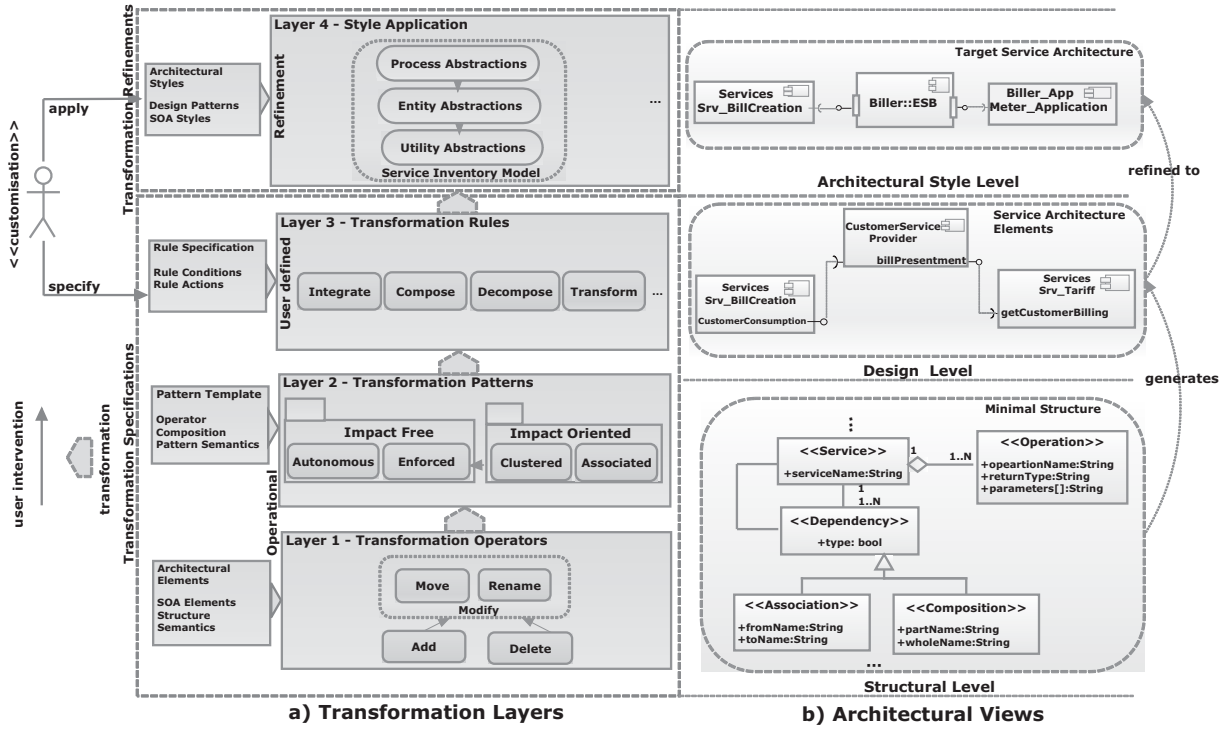

Figure 2. The Layered Transformation Framework for SOA Evolution.

We briefly discussed the architectural perspective above, while the transformation layers are detailed in the following sections.

IV. Transformation Operators

The transformation operators perform the structural changes to the architectural elements. We believe that the role of transformation/evolution operator(s) is of central importance to the successful design or architecture level evolution as demonstrated in [6], [20]. We go beyond these existing solutions to develop an operator calculus containing the fundamental change operators like: Add, Remove, Rename and Move with defined structural constraints. By composing the atomic change operators into our identified service architecture transformation patterns (defining the operational layer) we propose to preserve the structural properties of architectural elements during transformation at different levels. Three fundamental transformation operators are defined as:

- *Add:* The format of adding an architectural element is Δ*Add(ae(pl))* for an architectural element *ae* and a parameter list *pl*. For example, adding a service Srv-BillCreation (as in Figure 2b) can be specified as: Δ*Add(Srv(Srv-BillCreation))*.
- *Remove:* Removing an architectural element is specified as Δ*Rem(ae(pl))*.
- *Modify:* Two structural modification operators are Move (ΔMov) and Rename (ΔRen). The modification of an architectural element with ΔMod first removes the architectural element and then adds the new architectural elements: $\Delta Mod = \Delta Rem(ae1(pl1)); \Delta Add(ae2(pl2))$. The modification transformations in this case are operationally commutative.

Note, that ';' is the sequence operator – the only composition operator we introduce here. The aim of this operational layer is to support change representation (through change logs) and analysis (change impact determination). The transformation operators along with the architectural elements are composed into the transformation patterns.

Structural Constraints: During transformations these operators preserve the structural properties of the elements. We have semantically defined the structural constraints that ensure the transformational integrity for SOA elements, summarised as e.g. a service must contain one or more operations to provide or request the required functionality:

$$\forall srv_i \in SRV \ . \ \exists opr_n \in \text{OPR with } |n| \geq 1. \qquad (1)$$

for services SRV and operations OPR. Provider or requester services can communicate through association or composition type dependencies DEP, expressed as follows where the implication indicates a direction of dependency:

$$\forall dep(srv_i, srv_j) \in DEP \ . \ srv_j \rightarrow srv_i, \wedge srv_i, srv_j \in SRV \qquad (2)$$

The semantic properties for service operation(s) state that the addition or removal (T) of operations is semantically dependent on the service in which the operation is contained (expressed through implication).

$$\Delta T srv_i \rightarrow \Delta T opr_n \text{ with } |n| \geq 1. \quad (3)$$

When in a service dependency either of the services involved are removed, the corresponding dependency must be explicitly removed:

$$\Delta T(srv_i, srv_j) \rightarrow \Delta T(dep(srv_i, srv_j)) \quad (4)$$

A service violating these properties is an orphaned service and denoted as $srv_\emptyset$ (service with no operations and no dependencies), representing inconsistent transformations.

V. Transformation Patterns

At the architectural level of abstraction many system evolutions follow certain common patterns detailed for an ad-hoc peer-to-peer architecture in [10] or component based architectures as [22], [21]. Due to an inherent distributed nature of SOAs it is still an open challenge to consider the change impact on source and target architecture. By taking advantage of regularity in the space of common service architecture transformation we have identified four architectural transformation patterns in terms of how a transformation impacts its context (source and target architecture elements).

This layer formulates the primitive transformations by applying some recurring architectural transformation scenarios by a sequential composition of the layer 1 operators in a transformation pattern template. The transformation operations are the operational units of change management; the pattern categorisation here aims at change impact determination. Together with the structural constraints defined earlier (using transformation operators) the transformation patterns ensure the individual pattern's integrity during operator composition. A high-level structural view for the transformation patterns is presented in Fig. 3. In Fig. 3, affected (added or removed) services and operations are highlighted in grey, while affected dependencies are presented as dotted directed lines.

- Autonomous Transformations: applies if the addition or removal of SOA elements has no transformational impact on the existing system, see Fig. 3a.
- Clustered Transformations: the addition of new architectural elements (services) triggers the addition or removal of the corresponding operations or dependencies in a cluster of other services, see Fig. 3b.
- Associated Transformations: applies for addition or removal of a group of functionally associated architectural elements, see Fig. 3c.
- Enforced Transformations: applies if addition or removal (refactoring) of SOA elements is required to maintain transformational consistency, Fig. 3d.

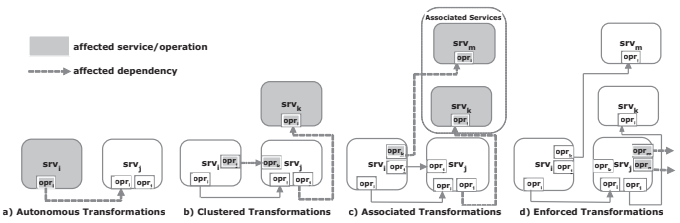

Figure 3. Architectural transformation patterns and their impact on the source and target architecture elements.

Example: We only illustrate the autonomous transformation based on a composition of layer 1 operators, see Fig. 3a. This pattern-based operator composes primitive transformations to define a pattern-associated higher-level transformations of SOA elements. The primitive transformations develop the foundation for user-defined design transformation rules at layer 3. The sequence is an autonomous transformation that adds a service, an operation and a corresponding dependency by preserving the structural constraints presented in Fig. 2b, can be specified as;

$\Delta Add(SRV(srv_i))$; $\Delta Add(OPR(opr_i, srv_i))$; by (1)
$\Delta Add(DEP(srv_i, srv_j))$ by (2), (4)

VI. Transformation Rules

Based on the core operational transformation calculus and the categorisation through the transformation patterns, the next layer addresses customisable, i.e. user-definable transformations. A declarative, rule-based transformation specification – consisting of goals, constraints and a definition can encapsulate the complex primitive transformations and enable a high-level design transformations addressing the purpose of evolution in terms of architectural concerns. The rule is specified in terms of the SOA elements to be transformed as:

- Goal: we propose four core transformation goals: *Integrate* (connects a provider and requester service), *Compose* (creates an abstraction for composing a number of SOA elements), *Decompose* (separates one SOA element into individual, connected elements), and *Transform* (replaces a SOA element without structural changes) on which users can build up. In Fig. 2b at the design level, the Srv-BillCreation is integrated with Srv-Tariff to getCustomerBilling information, through a mediator (customerServiceProvider). The designer can specify this as *Integrate(Srv-BillCreation, Srv-Tariff)*.
- Constraint: application-specific integrity constraints can be specified to ensure that the structure and semantic properties of SOA elements are preserved. The properties enhance operator composition for selected SOA elements. For *Srv-BillCreation, Srv-Tariff* $\in SRV$ we require $\exists$ *ASC(Srv-BillCreation, Srv-Tariff)* $\in DEP$.
- Definition: can be done by formulating primitive transformations (operator composition guided by transfor-

mation patterns) to execute (design-level) transformation rules on SOA elements, illustrated in Fig. 2b at design level

Δ *Add(SRV(Srv-BillCreation))* ;
Δ *Add(OPR(customerConsumption,nil,nil))* ;
Δ *Add(ASC(Srv-BillCreation,Srv-Tariff))*

The list of four core rule goals presented are examples of commonly used transformation rules in the context of SOA. The user can combine these rules to specify new rules for SOA element transformation.

VII. Architectural Styles and Refinement

As a last step, we apply SOA-specific architectural styles to refine the transformed design in order to complete the style-based service architecture evolution. Currently, these architectural patterns are confined to the common service inventory models detailed in [9]. These models are generic, which not only makes them common, but also customisable.

The aim of the architectural refinement using architectural styles is to enable a common organisation of service logic and establish a separation of business-centric and non business-centric service logic in the target architecture. Applying the architectural styles to transformed designs minimises the counter-productive transformational effects possibly caused by the pattern based transformations.

Example: Referring back to the Fig. 2b (viewed at the design level) a drawback of the transformed design is that it focuses on service-to-service level communication and directly connected services also known as point-to-point service communication in [5]. This violates the principle of a loosely coupled service design and therefore, should be refined (refactored) to obtain a robust and extensible target architecture. In order to improve the transformed design and moving towards a pattern-based evolved architecture for EBPP application, the technical services on the biller side have been implemented as an Enterprise Service Bus (ESB) pattern [9], as in Fig. 2b (viewed at the architectural level). In the evolved architecture, ESB facilitates the exposition of services using a centralized bus and handles an efficient messaging mechanism among services.

VIII. Related Work

With an increased adaptation of service-oriented software, the ultimate challenge lies in developing the processes, patterns, tools and environments to support a systematic evolution to prolong its productive life span. These claims are explicitly highlighted as the future research agendas in SOA research taxonomy [15] that is followed by a series of workshops focusing on the Maintenance and Evolution of Service-Oriented Systems (MESOA) [14]. Our work is an attempt in realising these potentials; by enabling an incremental transformation of SOA elements (atomic and composite services) at different abstraction level to facilitate a style-based evolution for service architecture.

Our work is partially guided by the Software Architecture EVolution Model [20] that enables a structural evolution of software architecture at different abstraction (meta, architecture, application) levels. However, the scope and applicability for SAEV is generally limited to traditional (component and connector kind of) architectures that operate in a controlled environment compared to distributed, loosely coupled (web services) architecture. We propose to extend the basic idea behind SAEV and tailor it to develop a coherent service architecture transformation framework. The proposed framework supports an incremental transformation of SOA elements (at, structure, design, architecture-style levels) to enable a refined and systematic service architecture evolution. In addition, we enforce structural constraints on SOA elements (service, operation, service dependency, etc.) to ensure transformation integrity for provider and requester services in the target SOA.

Based on a meta-model for both source and target architecture, the work proposed in [4] supports the automation of the architectural migration towards SOA using graph transformation rules over a model of the annotated source code. It mainly assumes that service-oriented systems are developed and deployed for internal integration and usage where there is some control over the deployed services. However, the recent emergence of a market for third-party services and cross organisational SOAs formulate a different set of challenges from an engineering perspective as detailed in [11]. In contrast, we plan to provide the customisation of transformation rules that is lacking in the discussed approaches for legacy migration towards SOA. In doing so, we aim at enabling an incremental transformation of service-oriented systems at different abstraction while taking into consideration the change impact analysis (for provider and requester) of SOA element (distributed services).

In the context of style-based SOA, [16] propose to obtain a service based architecture model that satisfies the requirements of the concrete architecture and complies with the constraints and vocabulary defined for a specific architectural style. In order to achieve this, the authors encode style templates within model transformations and then utilise model weaving that merges the architectural model with a model of the architectural style of choice. We specifically concentrate on developing a layered framework that is guided by operational and user-defined architectural transformations to facilitate SOA evolution, rather than a model transformation solution. Finally, we reviewed the UML4SOA [17] that defines a high-level domain specific language for modelling and transforming web service orchestrations to support the dynamic service composition. Our approach however is limited to a declarative composition of atomic services (at layer 3) to include composed services in service architecture model as detailed in [19].

IX. Conclusions and Outlook

With an increasing maturity of service-orientation, service-based architectures will evolve. This creates problems specific to the services platform where cross-organisational architectures are common. In this context, architectural transformation techniques are required that; i) take the impact of change into account as we do with our transformation patterns and ii) allow recurring change needs to be addressed through a customised transformation framework as we do with our user-defined declarative rules. We enable a transformation driven service architecture evolution at different abstraction levels in a unified, high-level framework independent of specific implementation details.

Preliminary Validations for the framework layers 1 and 2 (operational transformations) have been performed using XSLT based transformations that require further refinements. In the future, we plan to enable a declarative rule-based composition of atomic web services guided by [19] to included composite services in service design. An interesting task is the identification of some transformation anti-patterns that emerge as a result of some counter-productive transformations. The identification and resolution (refactoring) of the transformation anti-patterns can significantly contribute towards achieving dependable and cost-effective design and architectural transformations. In an overall context, we plan to proceed towards fully automating and validating these transformations in a formal way. Graph-based formalisms will provide the underlying rigour of the framework.

Acknowledgments

This work is supported, in part by Science Foundation Ireland through grant 03/CE2/I303_1 to Lero - The Irish Software Engineering Research Centre.

References

[1] NACHA - The Electronic Payments Association. Electronic Bill Presentment and Payment. http://www.nacha.org/.

[2] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A Method for Developing Service-Oriented Solutions. IBM, 2008.

[3] M. Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. John Wiley & Sons, 2008.

[4] R. Correia, C. Matos, R. Heckel, and M. El-Ramly. Architecture Migration Driven by Code Categorization. In *European Conference on Software Architecture*, 2007.

[5] C. de la Torre Llorente. Model-Driven SOA with Oslo. *The Architecture Journal*, 21:10–15, 2009.

[6] J. Dong, Y. Zhao, and Y. Sun. XSLT-based Evolutions and Analyses of Design Patterns. *Software: Practice & Experience*, 39:773–805, 2009.

[7] The Eclipse Foundation. Web Tools Platform (WTP) Project. http://www.eclipse.org/webtools/.

[8] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2009.

[9] T. Erl. *SOA Design Patterns*. Prentice Hall, 2009.

[10] D. Garlan, J. Barnes, B. Schmerl, and O. Celiku. Evolution Styles: Foundations and Tool Support for Software Architecture Evolution. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture*, 2009.

[11] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[12] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G.Koutsoukos, and L.F. Andrade. Architectural Transformations: From Legacy to Three-Tier and Services. In *Software Evolution*, pages 139–170. 2008.

[13] S. Johnston. UML 2.0 Profile for Software Services, IBM developerWorks, http://www.ibm.com/developerworks/rational/, 2005.

[14] G.A. Lewis, D.B. Smith, N. Chapin, and K. Kontogiannis. MESOA 2009: 3rd International Workshop on Maintenance and Evolution of Service-Oriented Systems. *IEEE International Conference on Software Maintenance*, 2009.

[15] G.A. Lewis, D.B. Smith, and K. Kontogiannis. A Research Agenda for Service-Oriented Architecture (SOA): Maintenance and Evolution of Service-Oriented Systems. 2010.

[16] M. López-Sanz, J.M. Vara, E. Marcos, and C.E. Cuesta. A Model-Driven Approach to Weave Architectural Styles into Service-Oriented Architectures. In *1st International Workshop on Model Driven Service Engineering and Data Quality and Security*, 2009.

[17] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *12th IEEE International Conference on Enterprise Distributed Object Computing* , 2008.

[18] Microsoft Corporation. Microsoft Visual Studio 2010. http://www.microsoft.com/visualstudio/.

[19] S.R. Ponnekanti and A. Fox . Sword: A Developer Toolkit for Web Service Composition. In *11th International World Wide Web Conference*, 2002.

[20] N. Sadou, D. Tamzalit, and M. Oussalah. How to Manage Uniformly Software Architecture at Different Abstraction Levels. In *24th International Conference on Conceptual Modeling*, 2005.

[21] D. Tamzalit, M. Oussalah, O. Le Goaer, and A.D. Seriai. Updating Software Architectures: A Style-based Approach. In *International Conference on Software Engineering Research and Practice*, 2006.

[22] D. Tamzalit, N. Sadou, and M. Oussalah. Evolution Problem Within Component-Based Software Architecture. In *Eighteenth International Conference on Software Engineering & Knowledge Engineering*, pages 296–301, 2006.

[23] H. Verjus and F. Pourraz. A Formal Framework For Building, Checking And Evolving Service Oriented Architectures. In *Fifth European Conference on Web Services*, 2007.