Supporting change propagation in the maintenance and evolution of service-oriented architectures

Hoa Khanh Dam and Aditya Ghose
School of Computer Science and Software Engineering
University of Wollongong
New South Wales 2522, Australia
{hoa, aditya}@uow.edu.au

***Abstract*—As Service-Oriented Architecture (SOA) continues to be broadly adopted, the maintenance and evolution of service-oriented systems become a growing issue. Maintenance and evolution are inevitable activities since almost all systems that are useful and successful stimulate user-generated requests for change and improvement. A critical issue in the evolution of SOA is change propagation: given a set of primary changes that have been made to the SOA model, what additional secondary changes are needed to maintain consistency across multiple levels of the SOA models. This paper presents how an existing framework can be applied to effectively support change propagation within a SOA model. We also propose to extend this framework with a minimal modification strategy that helps select change options in a manner that accommodates the structural and semantic dimensions of SOA models.**

***Keywords*-Service oriented architecture; maintenance and evolution; change propagation**

I. Introduction

Service Oriented Architecture (SOA), with its potential to significantly improve the development of high quality and complex systems, has attracted an increasing amount of interest from the research and business communities. The emergence of cloud computing together with a growing services-based economy are motivating enterprise transformation towards SOA to achieve agility, collaboration and efficiency. In fact, according to a survey conducted by Forrester Research[1] in May 2009, 75% of IT executives at Global 2000 organizations planned to adopt SOA by the end of 2009. On the one hand, SOA can be seen as an agile business architecture in which customers and suppliers are treated as business entities collaborating in a system of services. On the other hand, SOA is a technology architecture which helps realize the business services with an agile, interoperable and loosely-coupled suite of services that can be used within multiple business domains [1].

The combination of service oriented business and technical architecture gives SOA the ability to significantly reduce the gap between business and technology concerns. However, in order to leverage this advantage, it is important to provide modeling support in which business analysts and application architects can exchange views and share understanding. Among recent efforts in providing modeling support for SOA (e.g. SOMF [2], SOMA [3]), service oriented architecture modeling language (SoaML[2]) has emerged as a promising standard which offers the capability to model a SOA at the enterprise, system and systems of systems level. SoaML models show how business entities (e.g. people, organizations, and systems) collaborate via services within a SOA and how such services link to other parts (e.g. business processes, data, and business rules) of the SOA.

In recent years, the ever-changing business environment demands constant and rapid evolution of an organisation. As a result, changes to the SOA models of such an organisation is inevitable if the architecture are to remain useful and to reflect the current state of service provision and consumption. For example, initial changes in a SOA model which are made to include a new service provider may lead to secondary changes made to relevant service contracts and service composition. Such changes may lead to further changes in the implementation components and so on. The ripple effect that an initial change may cause in a SOA is termed *change propagation*. In a large modern organization which may have hundred of thousands of services connected to each other and to its structure and business processes, it becomes costly and labour intensive to correctly propagate changes. However, there has been very little work on dealing with changes in SOA [4]. Since SOA is a relatively new technology, maintenance of service-oriented systems has not been so far a critical issue. However, if we are to be successful in the long-term adoption of service-oriented development of software systems that remain useful after delivery, it is now crucial for the research community to provide solutions and insights that will improve the practice of maintaining and evolving service-oriented systems. More specifically, there is a need for techniques and tools that provide more effective automated support for change propagation within an SOA model. We do not believe that change propagation can be fully automated, since there are decisions that involve tradeoffs where human expertise is

[1] http://blogs.zdnet.com/service-oriented/?p=2053

[2] http://www.omg.org/spec/SoaML

required. However, it is possible to provide tool support in tracking dependencies, determining what parts of the SOA are affected by a given change, and, as in this paper, determining and making secondary changes.

The mainstream software maintenance and evolution face the same challenge in terms of change propagation, i.e. how to propagate changes so that consistency is preserved between different software artefacts. Although much of the work in this area addresses the issue at code level (e.g. [5], [6]), an increasing number of tools and techniques have been proposed to deal with changes at the model level (e.g. [7]), in line with better recognition of the importance of models in the software development process [8]. In this context, an agent-oriented change propagation framework [9] has been proposed by the first author to deal with propagating changes through design models. The key ideas of this approach are that (1) a particular style of representation for repair plans — inspired by Belief-Desire-Intention (BDI) style agents [10] — allows a large number of possible inconsistency resolutions to be represented compactly; furthermore, (2) these repair plans can be automatically generated from the Object Constraint Language (OCL) constraints; and finally, (3) using a cost calculation can reduce the number of options to be presented to the user. Previous work has shown the effectiveness of this framework in supporting change propagation within agent-oriented design models [11], [12], [13], [14]. Our recent work [15] has also indicated that the framework is applicable to deal with changes in enterprise architectures.

In this paper, we present how this change propagation framework can be applied to deal with changes within SOA models. In particular, we will show how changes are propagated across a number of models developed using the Service oriented modelling language (SoaML). We however argue that a major limitation of this change propagation framework resides in the use of a cost-based approach to select repair options. In fact, assigning costs to change actions is, to some extend, arbitrary, and the cheapest cost heuristic, as used in the existing framework, may not always lead to the best way to resolve inconsistencies. Therefore, we will propose to substitute the cost calculation with a minimal modification strategy that helps select change options in such a way that it accommodates both the structural and semantic dimensions of SOA models. This approach, in our view, better reflects the conceptual distance between the modified model and the original one.

The organization of this paper is as follows. In the next section, we briefly describe SoaML and use a running example of a dealers network to illustrate how SoaML can be used to model a SOA. Section III serves to discuss the change propagation framework. Related work is briefly presented in section IV. We conclude by discussing future directions in section V.

II. SoaML

Service oriented architecture modeling language (SoaML) is an emerging standard adopted by the Object Management Group[3] (OMG), which aims to support the activities of modeling and design of services within a service oriented architecture. SoaML has been developed as a Unified Modelling Language (UML) profile, which is the core modeling standard of OMG. In this section, we briefly describe how SoaML can be used as an architectural language to provide technology independent and a standard way to create, communicate and leverage a SOA. For the illustration purpose, we use a running example motivated by a business scenario in which a community of independent dealers, manufacturers, and shippers want to able to work together by providing and using each other's services, without the need to redesign their business processes or systems [16]. We will now show how to use SoaML to define a SOA for the community to enable such an open and agile business environment.

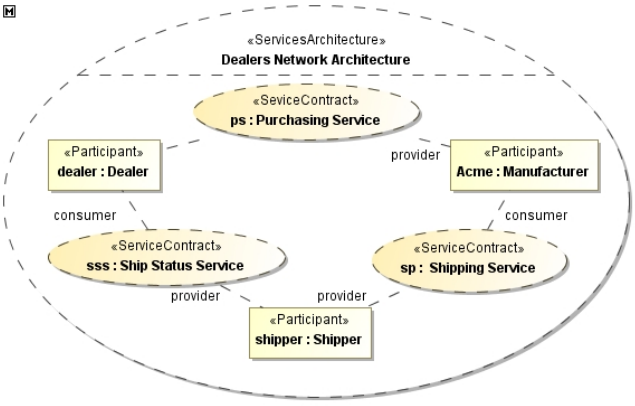

Figure 1. Dealer network architecture

In this example, the community initially has a number of participants: a manufacturer named Acme, its dealers, and a shipping company which Acme uses to ship products to its dealers. This business collaboration can be described using SoaML "ServiceArchitecture" as in figure 1. It shows a high-level and contextual view of a network of participant roles providing and consuming services. For instance, Acme plays the role of a "consumer" with respect to the "Shipping Service" and the shipper plays the role of a "provider" which respect to the same service. It is noted that a participant can play multiple roles in different services within architecture. For instance, Acme also plays the role of a "seller" in the "Purchasing Service".

Each service is represented in SoaML as a ServiceContract. For instance, details of the "Shipping Service" ServiceContract are defined in figure 2, which shows the terms

[3]http://www.omg.org

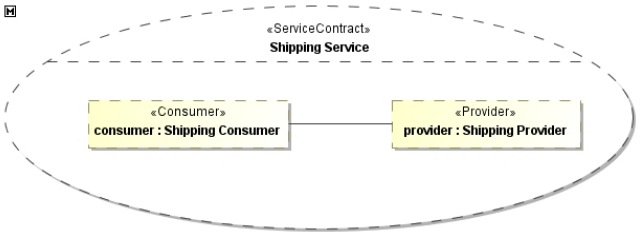

Figure 2. Shipping service

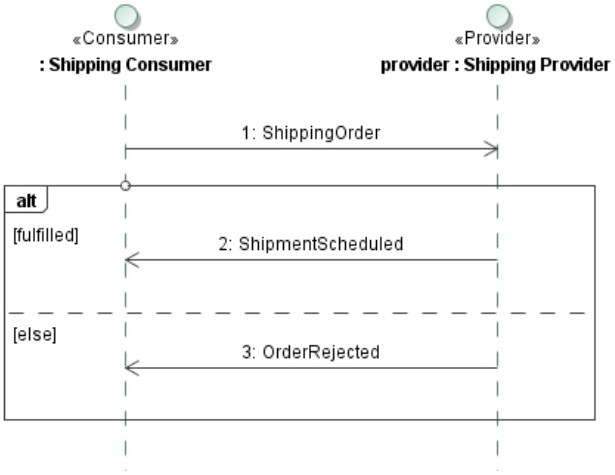

Figure 3. Shipping service

and condition of "shipping" as well as define two roles: "Shipping Consumer" and "Shipping Provider". More details describing the flow of information (as well as products, services and obligations) between the participants can be specified using a UML behavior, which can be either UML activity, interaction, or state diagrams. Figure 3 shows an example of a simple choreography of the service contract: the shipping consumer sends a "ShippingOrder" to the shipping provider, and the shipping provider sends back either a "ShipmentScheduled" or a "OrderRejected". Such a behaviour depicts how the message are choreographed in the service contract, i.e. what flows between the participants, when and under what conditions.

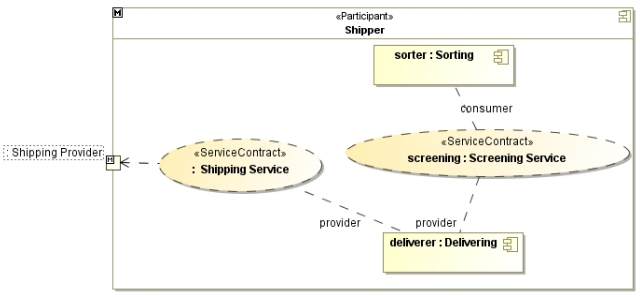

Figure 4. Shipper's components

Figure 4 shows shows an example of the service architecture of a participant, i.e. the Shipper, that complies with the community architecture. The shipper has a "Sorting" component which is responsible for sorting and screening packages and a "Delivering" component which is responsible for delivering packages. The shipper also delegates the shipping service to the delivering shipping component. The Shipper has a "Shipping Provider" ServicePort which is compatible with the "Shipping Provider" role that it plays in the Shipping Service Contract. Figure 5 shows an activity diagram which defines the business process of the Shipper. It it is noted that for the brevity of the example, this is just a portion of the business process - but it shows the correspondence between the information flows of the SOA and activities within a participant.

Figure 5. Shipping business process

A. SoaML metamodel

It is noted that since SoaML is a UML profile, it extensively uses and extend UML elements and structures such as collaborations, parts, ports, and composite structures. Figure 6 is an excerpt of a SoaML metamodel which shows the relationships between major elements in a SOA defined using SoaML. A ServiceArchitecture is a UML collaboration which consists of a number of ParticipantRoles participating in a ServiceContractUse, each of which corresponds to a Participant type. A Participant has a set of Ports which represent features of the Participant where the service is offered or consumed . There are two types of Ports: ServicePort – a port where a service is offered, and Request Port – a port where a service is consumed. Those ports have a ServiceInterface type. Each ServiceContractUse has a corresponding ServiceContract as a service specification. There are different roles involve in a ServiceContract, each of which has a type, which must be a ServiceInterface. Furthermore, an important part of the ServiceContract is the choreography, which is a UML Behaviour such as may be shown on an interaction diagram, or activity diagram that is owned by the ServiceContract. The choreography specifies exchanges between the ServiceRoles in a ServiceContract.

Consistency requirements upon a model are often expressed using its metamodel and a set of constraints that specify conditions that a well-formed and consistent model should satisfy. Constraints may describe syntactic and semantic relationships between model elements. They may also be used to prescribe coherence relationships between different views of a model, i.e. intra-model or horizontal consistency as defined in [17]. In addition, constraints can

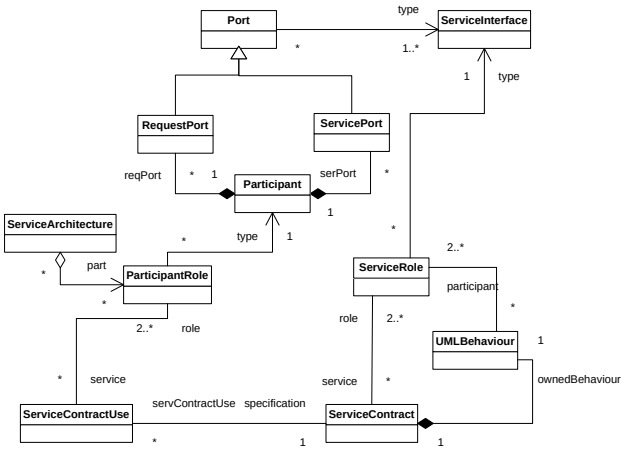

Figure 6. An excerpt of a SoaML metamodel

be used to impose best practices, industry standards, compliance requirements and any other specific requirements related to a particular domain. We use the Object Constraint Language (OCL) [18]. to specify constraints. Below are examples of two consistency constraints for SoaML models that are expressed using OCL.

C1: Participants in a service contract should have services ports defined on the participant type which have a ServiceInterface type defined in the service contract.
Context ParticipantRole inv C1:
self.service→forAll(scu : ServiceContractUse | scu.specification.role→exists(r : ServiceRole | r.type = self.type.serPort.type))

C2: The owned behaviour which describes the choreography of a service contract should have participants which play a role in the service contract.
Context ServiceContract inv C2:
self.ownedBehaviour.participant→forAll(sr : ServiceRole | sr.service→includes(self))

Such consistency constraints and the metamodel form a important foundation for the process of propagating changes in a SoaML model which we describe in the next section.

III. Change propagation

In the previous section, we have briefly shown how business analysts and SOA developers can use SoaML as a modeling language to build a SOA enterprise architecture. In this context, such a business and technology architecture is represented as a SoaML model. When a SoaML model is modified (due to changes in requirements or environments), typically some primary changes are made and then additional, secondary, changes are made to this model as a result. Change propagation is the process of determining and making these secondary changes. Recently, a number of existing modelling tools (e.g. IBM Rational Software Architect[4], NoMagic's MagicDraw[5], Modelio[6]) have provided support for SoaML modelling. These tools significantly help business analysts and SOA developer quickly develop a new SOA for their organizations. However, they provide very limited support for propagating changes during the maintenance and evolution of the existing SOA models.

In this section, we will briefly explain how we support change propagation in SoaML models by adopting a generic change propagation framework which has been previously developed by the first author [9]. This change propagation framework has been developed based on the conjecture that, given a suitable set of consistency constraints, change propagation can be done by fixing inconsistencies caused by primary changes. In other words, we propagate changes by finding places in a model where the desired consistency constraints are violated, and fixing them until no inconsistency is left in the model. This framework provides a "change propagation assistant" that helps business analysts or SOA developers by suggesting additional (secondary) changes once primary changes have been made. Figure 7 shows an overview of the change propagation framework.

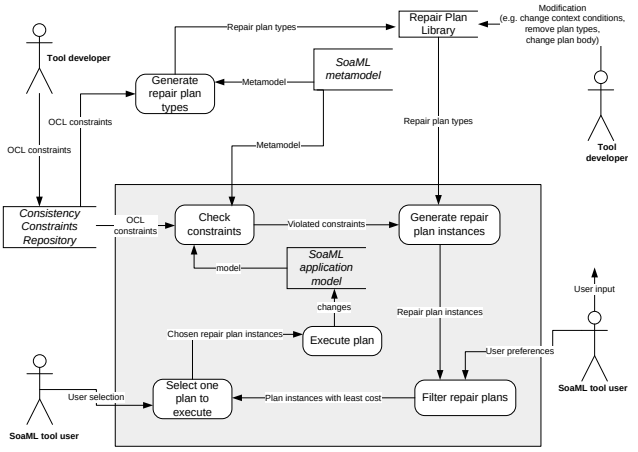

Figure 7. Change propagation framework for SoaML

In this change propagation framework, there are three important data items: a collection of consistency *constraints*, a SoaML *model*, and a collection of *repair plans*. The change propagation framework consists of the following key activities:

1) After consistency constraints are written (either by SoaML tool developers or tool administrators), the repair plan types are automatically generated from the constraints and form a library of repair plans.
2) The user (i.e. either a business analyst or a SOA developer) uses the SoaML tool to make some primary

[4]http://www.ibm.com/software/awdtools/architect/swarchitect
[5]http://www.magicdraw.com
[6]http://modeliosoft.com

changes to the SoaML model and then the user invokes the tool to starts propagating changes.

a) Consistency constraints previously defined are checked if they still hold in the modified SoaML model. Violated constraints are identified in this step.
b) Repair plan instances (i.e. repair options) for the violated constraints are automatically generated based on the library of repair plan types.
c) Repair options are then filtered to eliminate infeasible repair options (e.g. cyclic) or to reflect the user preferences. Previous work [11] uses a cost-based approach to select repair plans based on their costs. In this paper, we propose an alternative minimal-change approach to repair plan selection.
d) The repair options that successuflly pass the previous filtering round are presented to the user and ask for their selection.
e) The selected repair plan option is executed, and it updates the SoaML model.

As a step towards automated change propagation, the repair plans are generated automatically (at design time) from the OCL consistency constraints and form a *repair plan library* which is used at run time. A key consequence of generating plans from constraints, rather than writing them manually, is that, by careful definition of the plan generation scheme, it is possible to guarantee that the plans generated are correct, complete, and minimal, i.e. there are no repair plans to fix a violation of a constraint other than those produced by the generator; and any of the repair plans produced by the generator can fix a violation. However, we also allow the users to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed.

We have developed a translation that takes a OCL constraint as input (for example constraint C1 and C2) and generates repair plans that can be used to fix violations [9, Chapter 6]. Such a translation can be developed by considering all the possible ways in which a constraint can be false, and hence all the possible ways in which it can be made true. For instance, repair plans to make constraint C1 true are defined by choosing all the elements *scu* in *self.service* that makes C1' false[7], and for each such element either delete it from *self.service* or make C1' true for it. The repair plan generator also has rules for making the sub-constraint C1' true, which match three ways of making this constraint true: picking one element *r* in *scu.specification.role* and making constraint C1" true[8] for this element; adding an existing element to *scu.specification.role* and making the constraint true for it; and creating a new element, adding it to *scu.specification.role*, and making the constraint true for it. Plan generation rules are then applied to generate repair plans for sub-subconstraint C1" and so on. Such repair actions are eventually translated to various change actions (e.g. addition, deletion, modification, creation) made to a SoaML model. Those repair plans form a library of plans for fixing violations of constraints C1 and C2, which are instantiated at run time. The plan generation rules cover most of the OCL expressions including attributes (e.g. $<$, $>$, $<>$, and $=$), navigations (e.g. *self.service*), boolean connectives (e.g. *and*, *or*, etc.), set expressions (e.g. *forAll*, *exists*), and addition or deletion involving derived sets (e.g. *select*, *union*, *reject*) [9, Chapter 6].

Let us use the dealers network example in section II to illustrate how this change propagation process works for SoaML models in practice. We assume that there is a new requirement that the shipping company needs to comply: shipping packages needs to be scanned upon arrival to the shipping company to establish their status, and ensure that they are screened by a regulatory authority to determine if they should be held. The following scenario may take place:

- The SOA developer create two roles (i.e. ServicePart in the SoaML metamodel) "ShippingOrganization" and "RegulatoryAgent".
- The SOA developer then creates a choreography between these roles in the new service in a form of a UML activity diagram as shown in figure 8.

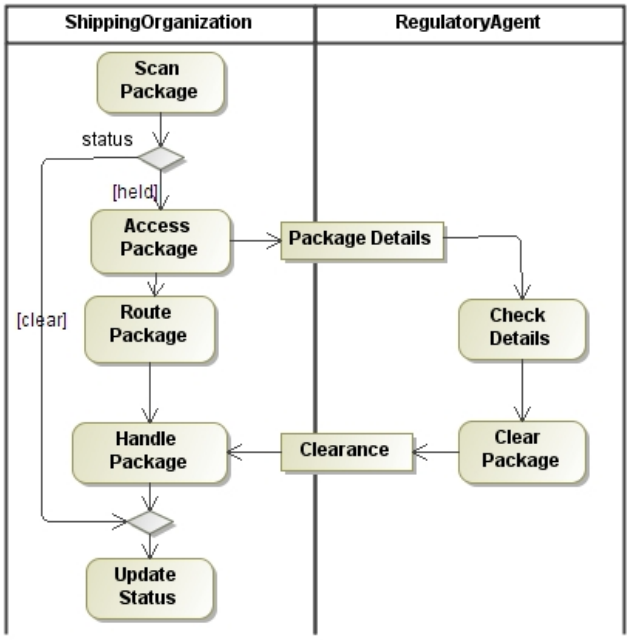

Figure 8.

At this point, the SOA developer may wish to ask

[7] C1' $\stackrel{\text{def}}{=}$ scu.specification.role→exists(r : ServiceRole | r.type = self.type.serPort.type)

[8] C1" $\stackrel{\text{def}}{=}$ r.type = self.type.serPort.type

our change propagation system what other artefacts he/she should alter to correctly propagate the new change. In this example, consistency constraints C1 and C2 (refer to section II for details of these constraints) are checked and found to be violated. Therefore, to fix those violations, repair plan instances are generated from the library of plan types which discussed earlier. These would result in the following change actions made to the SoaML model of the dealers network[9]:

1) Create a ServiceInterface "RegulatingInterface".
2) Create a new ServiceContract "Regulating Service".
3) Connect "ShippingOrganization" with "Regulating Service".
4) Connect "RegulatoryAgent" with "Regulating Service".
5) Connect "RegulatoryAgent" with "RegulatingInterface", i.e. "RegulatoryAgent" is a type of 'RegulatingInterface".
6) Create a new Participant "RegulatoryAuthority".
7) Create a new ParticipantPart "regAuth" which has a type of "RegulatoryAuthority".
8) Create a new ServiceContractUse "regService" which has a type of "Regulating Service".
9) Connect "regAuth" with "regServie" such that "regAuth" plays the "RegulartoryAgent" role.
10) Connect "shipper" with "regServie" such that "shipper" plays the "ShippingOrganization" role.
11) Create a ServicePort "regPort" which has a type of "RegulatingInterface".
12) Connect "regPort" to "regAuth".

Figure 9 shows a portion of those secondary changes. It is however noted that the above repair options are however only one of the multiple ways of making changes to the SoaML model to remove inconsistencies. In the next section, we discuss an approach to select between multiple change options.

A. Minimal change strategy to select repair options

In practice, there can be multiple applicable repair options for resolving a given inconsistency. Choosing between those different possible repair options can depend on various factors such as the cause of inconsistencies, or even factors other than consistency that contribute to a good design (e.g. experience, knowledge on the future evolution of the architecture, architecture styles). In general, many of these dependencies may not even be capable of being formulated formally and being captured without extra knowledge provided by the user. As a result, it is expected that the execution of repair actions requires user interaction.

In some cases, the number of different ways of fixing a inconsistency can however be very large. Therefore, it is also important not to overwhelm the user with a large

[9]Actions 1-4 make constraint C2 hold for the new SoaML model, and remaining actions make constraint C1 hold.

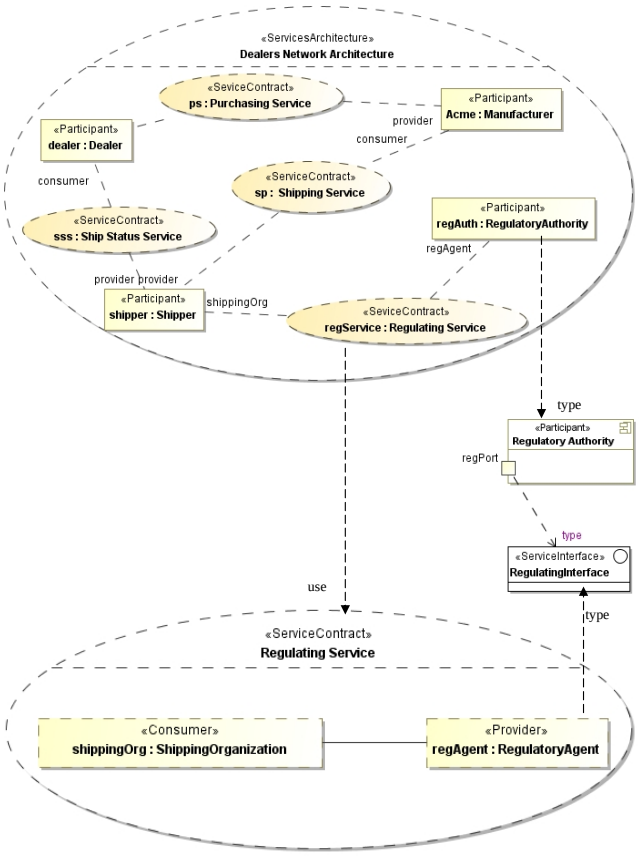

Figure 9. "Regulating Service" service choreography

number of choices. For example, it is necessary to prevent infeasible repair options (e.g. repair actions that result in infinite cycles) from being presented to the user. The issue of repair plan selection has been addressed in [11] by defining a suitable notion of *repair plan cost* that takes into account the important cascading nature of change propagation and fixing inconsistencies. More specifically, this approach provides a cost calculation component that is responsible for calculating the cost of each repair plan instance. This cost calculation takes into account that fixing one violated rule may also repair or violate others as a side effect, and thus the cost calculation algorithm computes the cost of a given repair plan instance as including the cost of its actions (using basic costs defined by the user), the cost of any other plans that it invokes directly, and also the cost of fixing any rules that are made false by executing the repair plan.

The cost-based approach, to some extends, reflects the user preferences in terms of biasing repair plans that have cheaper costs. In addition, the user may use this mechanism to adjust the change propagation process. For example, if he/she wishes to bias the change propagation process towards adding more information then he/she may assign lower costs to actions that create new entities or add entities,

and higher costs to actions that delete entities. However, this cost-based approach also has some major limitations. More specifically, the cheapest cost heuristic may not always lead to the best way to resolve inconsistencies. Choices amongst alternative repair plans are necessarily driven by domain-specific consideration, and cannot be adequately captured in a cost-based approach.

We argue that the best inconsistency resolution is the one for which the resulting model, after having fixed all violations, is "conceptually closest" to the original model. Therefore, we adopt a minimal-change approach to filter repair options in our change propagation framework, which have been proposed by the second author in the context of auditing compliance for business processes described using the Business Process Modelling Notation[10] (BPMN) [19]. In this context, an important step is defining what it means for a SoaML model to minimally deviate from another. This task is however complicated since there is no consensus on the semantics of SoaML given the fact that it is still an emerging standard. In this paper, we focus on describing minimal changes for a service choreography and leave other parts of the SOA for future work.

In SoaML, a service choreography can be specified using a UML activity diagram in which each swimlane represents a service participant. We encode this representation of a service choreography into semantically-annotated diagrams called Semantic Process Networks (or SPNets) [19]. A SPNet is a digraph (V, E) in which each node is of the form $\langle ID, nodetype, owner\rangle$[11] and each edge is of the form $\langle\langle u, v\rangle, edgetype, condition\rangle$. Each event, activity, decision, or fork/join in an activity diagram maps to a node, with the *nodetype* indicating whether the node was obtained from an *event*, *activity*, *decision* or *fork/join* respectively in the activity diagram. The *ID* of nodes of type event, decision or activity refers to the *ID* of the corresponding event, decision or activity in the activity diagram. The *owner* attribute of a node refers to the service role associated with the swimlane from which the node was obtained. The *edgetype* of an edge can be either *control* or *object* depending on whether the edge represents a control flow or object flow in the activity diagram. The *condition* associated to an edge describes the guard condition, set to true by default, controlling the flow of the process. It is noted that a unique SPNet exists for each UML activity diagram.

Based on the SPNets, we then define a class of proximity relations that allow us to compare alternative modifications of a service choreography in terms of how much they deviate from the original model. Due to space limitation, in this paper we present one typical class of proximity relations: *structural proximity*. Another type of proximity relation that we have explored but is not presented in this paper is *semantic proximity* which involves the use of effect annotations for process models [20].

Each SPNet *spn* is associated with a proximity relation $\leq_{spn}$ such that $spn_i \leq_{spn} spn_j$ denotes that spn_i is closer to *spn* than spn_j. We define $\leq_{spn}$ as a tuple $\langle \leq^V_{spn}, \leq^E_{spn}\rangle$ where $\leq^V_{spn}$ is a proximity relation associated with the set of nodes V of *spn*, and $\leq^E_{spn}$ is a proximity relation associated with the set of edges E of *spn*.

The proximity relations $\leq^V_{spn}$ and $\leq^E_{spn}$ can be defined in different ways to reflect various intuitions. For instance, the following set inclusion-oriented definition might be useful in some situations: $spn_i \leq^E_{spn} spn_j$ iff $V_{spn} \triangle V_{spn_i} \subseteq V_{spn} \triangle V_{spnj}$, where A $\subseteq$ B denotes the symmetric difference of sets A and B[12]. Alternatively, set cardinality-oriented proximity measurement can be useful in other situations, which is defined as: $spn_i \leq^E_{spn} spn_j$ iff $|V_{spn} \triangle V_{spni}| \leq |V_{spn} \triangle V_{spnj}|$[13]. Such alternative definitions can be applied for the $\leq^V_{spn}$ proximity relation. Both $\leq^V_{spn}$ and $\leq^E_{spn}$ define the *structural proximity* of one SPNet to another.

Let now use the dealers network example to illustrate how our structural proximity relations can be used to select repair options in the change propagation framework. Assume that in addition to the two consistency constraints C1 and C2, there is another constraint that should be taken into account: *packages known to be held by a regulatory authority must not be routed by a shipping company until the package is known to be cleared by the regulatory authority*. This constraint is an example of a domain-specific constraint which reflects a compliance requirement. Due to the introduction of this constraint, changes previously made to the SoaML model of the dealers network architecture, particularly the service choreography, cause a violation of this constraint. In fact, as can be seen in figure 8, a package is routed before clearance notification is received from the regulatory agent.

Although there is a number of alternative resolutions for this inconsistency, we present here two simple resolutions for the purpose of illustrating our minimal change approach. However, the same techniques can be applied for other alternatives. Figure 10 and 11 show how the existing regulating service choreography can be modified into two different ways to conform with the new constraint. Let SC_1 be the service choreography represented in figure 10 , SC_2 be the service choreography depicted in figure 11, and SC_0 be the existing one in figure 8. We now use the proximity relations defined earlier to identify which one is more preferable in terms of minimal deviation from the original service choreography.

With regard to the proximity relation $\leq^V_{spn}$, it can be easily seen that SC_1 and SC_2 share all their nodes with SC_0, and

[10] http://www.bpmn.org

[11] Originally, a SPNet also includes an immediate effect and cumulative effect. These two concepts are used for semantic proximity which due to space limitation we do not present in this paper.

[12] The symmetric difference of sets A and B is the set of all elements of A or B which are not in both A and B.

[13] $|A|$ denotes the cardinality of set A

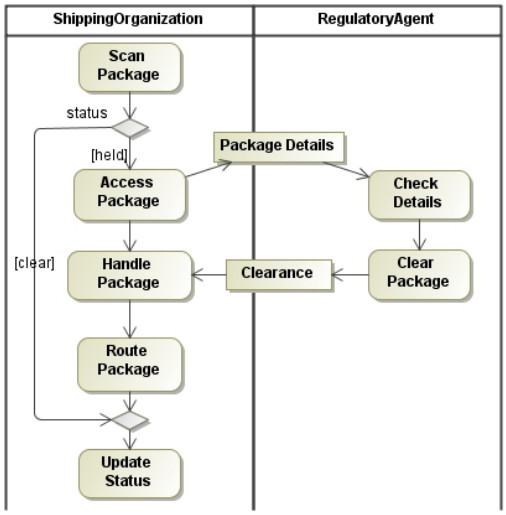

Figure 10. Resolved "Regulating Service" service choreography (SC_1)

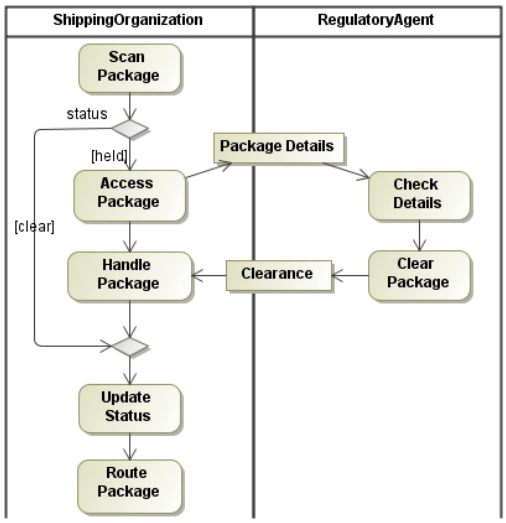

Figure 11. Resolved "Regulating Service" service choreography (SC_2)

therefore, no comparison can be made across this structural dimension. Figure 12 shows the differences between SC_1 and SC_2 and SC_0 in terms of the proximity relation $\leq^E_{spn}$. The significant edge difference between SC_1 and SC_0 includes the "Route Package"→ "Handle Package"edge. SC_2 also differs with SC_0 across some edges including "Update Status"→ "Route Package". If an inclusion-oriented definition for proximity is applied, we would not be able to differentiate SC_1 and SC_2 with regard to structural proximity to SC_0. On the other hand, if we choose to apply the cardinality-oriented definition, wewould determine $SC_2 \leq^E_{spn} SC_1$ as $|SC_1 \triangle SC_0|$ = 6 and $|\ SC_2 \triangle SC_0\ |$ = 4. It means that SC_2 is closer to SC_0 than SC_1 and consequently is the preferable repair option.

$SC_1 \triangle SC_0$	AssessPackage→ HandlePackage (SC_1), DecisionNode → RoutePackage (SC_1), RoutePackage → UpdateStatus (SC_1), DecisionNode → UpdateStatus (SC_0), RoutePackage → HandlePackage (SC_0), AssessPackage→ RoutePackage (SC_0)
$SC_2 \triangle SC_0$	AssessPackage→ HandlePackage (SC_2), UpdateStatus → RoutePackage (SC_2), AssessPackage → RoutePackage (SC_0), RoutePackage → HandlePackage (SC_0)

Figure 12. Edge difference of SC_1 and SC_2 with regard to $SC0$

IV. Related work

As mentioned earlier, there is limited work that specifically addresses maintenance and evolution in a service-oriented environment although this issue is important and growing more difficult [4]. The work in [21] proposes an top-down approach to analyze the impact of changes in business processes upon the source code, and uses this analysis to identify affected system components. On the other hand, the work in [22] proposes a bottom-up approach in which they provide a set of generic guidelines for assessing changes made to a service or its implementation and their impact to the business processes and other consumers of the service. Change impact analysis is however only loosely related to our work. Change impact analysis techniques aim to assess the extent of the change, i.e. the artefacts, components, or modules that will be impacted by the change, and consequently how costly the change will be. Our work is more focused on implementing changes by propagating changes between SOA artefacts in order to maintain consistency as the SOA evolves.

In the area of change propagation for SOA-based environment, Ravichandar et. al. [23] has recently proposed a set of inference rules between use cases, sequence diagrams and service specifications. Such rules are used to determine elements in one artefact that are directly or indirectly impacted by the changes in the other artefact. These rules are also defined to propagate the changes across those specific types of models. Recent work in [24] also aims to automate change propagation by identifying specific change propagation rules for all types of changes in SOA solution design. Similarly to our work, their change propagation framework is flexible and extensible in which it can accept any model types as long as they are compliant to the Meta-Object Facility[14] (MOF) standard. However, those approaches suffer from the correctness and completeness issue since the rules are developed manually by the user. As a result, there is no guarantee that these rules are complete (i.e. that there are no inconsistency resolutions other than those defined by the rules) and correct (i.e. any of the resolutions can actually fix

[14]http://www.omg.org/mof

a corresponding inconsistency).

There has been a wide range of work in the area of consistency management of models. Some of them (e.g. [25]) addresses the issue of consistency checking instead of resolving inconsistencies. Other work such as the Viewpoints approaches (e.g. [26]), inconsistencies arising between individual viewpoints are detected by translating into a uniform logical language. Such inconsistencies are resolved by having meta-level inconsistency handling rules, which have to be defined by user. However, these approaches suffer from the correctness and completeness issue as discussed earlier. In addition, a significant effort is required to manually hardcode such rules when the number of consistency constraints increases or changes. In order to deal with this issue, [27] has proposed an approach for automatically generating repair options by analyzing consistency rules expressed in first order logic and models expressed in xLinkIt. They did not take into account dependencies among inconsistencies and potential interactions between repair actions for fixing them. In other words, their work considers repair actions as independent events, and thus does not explicitly deal with the cascading nature of change propagation. The work proposed in [28] aims to fix inconsistencies in UML models at the level of detailed design (instead of at the higher level of SOA). In addition, their work does not provide options for how to repair inconsistencies, but only suggests starting locations (entities in the model) for fixing the inconsistency. Their recent work [29] has addressed this issue by only considers a single change at a time, is potentially incomplete and is not suitable for change propagation, and does not consider the creation of model elements.

V. Conclusions and Future Work

In this paper, we have presented an approach to support change propagation in the maintenance and evolution of service oriented architecture. Although we have applied our approach to SoaML, an emerging standard for modelling SOAs which has been adopted by the OMG, our ideas and results can be applied to other SOA modelling languages. The key idea of our approach is a change propagation framework which takes a SoaML metamodel and a set of consistency constraints as inputs and proposes additional (secondary) changes once primary changes have been made to a SoaML model. Change propagation is driven by fixing consistency constraint violations caused by either primary or secondary changes made to the SoaML model.

Using this change propagation framework, change options are represented in terms of repair plan types which allows us to abstractly represent certain classes of concrete ways of fixing a consistency constraint. The representation in terms of repair plan types can represent compactly a large number of repair options, and captures nicely the cascading nature of repairing constraint violations, and the way that a given constraint violation may be repaired in a number of ways. In addition, using this approach, SoaML tool developers do not need to write resolution or change propagation rules and consequently save substantial time. More importantly, they avoid the issues of soundness and completeness, since repair plan types are automatically generated from the OCL consistency constraint, and are guaranteed to be sound and complete [9, Chapter 6].

The current cheapest cost heuristic used in this change propagation framework is, to some extend, arbitrary, and may not always lead to the best way to resolve inconsistencies. We have argued that the best inconsistency resolution is the one for which the resulting SOA model, after having fixed all violations, is conceptually closest to the original model. This conceptual distance is represented as a class of proximity relations which can capture both structural and semantic proximity. In this paper, we have defined the structural proximity of one service choreography to another and illustrated how it can be used to select among alternative inconsistency resolutions.

Our future work involves exploring proximity relations on other parts of a SOA such as the quality of service (QoS) constraints specified in service contracts. Our long-term goal is to develop a class of proximity relations for the whole SoaML model which captures both the structure and semantic of a SOA, and use this to repair plan selection in the context of change propagation. Another important part of our future work involves integrating the change propagation framework into an existing SoaML tool. This would allow use to perform further case studies to evaluate the effectiveness and efficiency of our approach.

References

[1] C. Casanave, "Enterprise service oriented architecture using the OMG SoaML standard," Model Driven Solutions, Inc, White Paper, December 2009.

[2] M. Bell, *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*. Wiley & Sons, 2008.

[3] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley, "SOMA: a method for developing service-oriented solutions," *IBM Syst. J.*, vol. 47, no. 3, pp. 377–396, 2008.

[4] K. K. Grace A. Lewis, Dennis B. Smith, "A research agenda for service-oriented architecture (SOA): Maintenance and evolution of service-oriented systems," Carnegie Mellon Software Engineering Institute (SEI), Technical Note CMU/SEI-2010-TN-003, March 2010.

[5] V. Rajlich, "A methodology for incremental changes," in *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Process in Software Engineering*, Cagliary, Italy, May 2001, pp. 10–13.

[6] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 284–293.

[7] A. van Deursen, E. Visser, and J. Warmer, "Model-driven software evolution: A research agenda," in *Proceedings 1st International Workshop on Model-Driven Software Evolution (MoDSE)*, D. Tamzalit, Ed. University of Nantes, 2007, pp. 41–49.

[8] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: Model-driven development." *IEEE Software*, vol. 20, no. 5, pp. 14–18, 2003.

[9] K. H. Dam, "Supporting software evolution in agent systems," Ph.D. dissertation, RMIT University, School of Computer Science and IT, 2009.

[10] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*. Springer-Verlag, 1996, pp. 42–55.

[11] K. H. Dam and M. Winikoff, "Cost-based BDI plan selection for change propagation," in *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller, and Parsons, Eds., Estoril, Portugal, May 2008, pp. 217–224.

[12] K. H. Dam, M. Winikoff, and L. Padgham, "An agent-oriented approach to change propagation in software evolution," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2006, pp. 309–318.

[13] K. H. Dam and M. Winikoff, "Generation of repair plans for change propagation," in *Agent-Oriented Software Engineering VIII*, ser. Lecture Notes in Computer Science, M. Luck and L. Padgham, Eds., vol. LNCS 4951. Springer Berlin / Heidelberg, April 2008, pp. 132–146.

[14] ——, "Evaluating an agent-oriented approach for change propagation," in *Proceedings of the Ninth International Workshop on Agent Oriented Software Engineering*, M. Luck and J. J. Gomez-Sanz, Eds., Estoril, Portugal, May 2008, pp. 61–72.

[15] H. K. Dam, L.-S. Le, and A. Ghose, "Supporting change propagation in the evolution of enterprise architectures," in *The 14th IEEE International Enterprise Distributed Object Computing Conference (EDOC) (to appear)*, October 2010.

[16] Object Management Group, "Service oriented architecture Modeling Language (SoaML) Version 1.0 - Beta 2," http://www.omg.org/spec/SoaML/1.0/Beta2/PDF, 2009.

[17] G. Spanoudakis and A. Zisman, "Inconsistency management in software engineering: Survey and open research issues," in *Handbook of Software Engineering and Knowledge Engineering*, K. S. Chang, Ed. World Scientific, 2001, pp. 24–29.

[18] Object Management Group, "Object Constraint Language (OCL) 2.0 Specification," http://www.omg.org/docs/ptc/03-10-14.pdf, 2006.

[19] A. Ghose and G. Koliadis, "Auditing business process compliance," in *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 169–180.

[20] K. Hinge, A. Ghose, and G. Koliadis, "Process seer: a tool for semantic effect annotation of business process models," in *EDOC'09: Proceedings of the 13th IEEE international conference on Enterprise Distributed Object Computing*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 49–58.

[21] H. Xiao, J. Guo, and Y. Zou, "Supporting change impact analysis for service oriented business applications," in *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments at International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, p. 6.

[22] L.-J. Zhang, A. Arsanjani, A. Allam, D. Lu, and Y.-M. Chee, "Variation-oriented analysis for SOA solution design," *Services Computing, IEEE International Conference on*, vol. 0, pp. 560–568, 2007.

[23] R. Ravichandar, N. C. Narendra, K. Ponnalagu, and D. Gangopadhyay, "Morpheus: Semantics-based incremental change propagation in SOA-based solutions," in *SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 193–201.

[24] R. Sindhgatta and B. Sengupta, "An extensible framework for tracing model evolution in SOA solution design," in *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2009, pp. 647–658.

[25] A. Egyed, "Instant consistency checking for the uml," in *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006, pp. 381–390.

[26] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," *IEEE Trans. Softw. Eng.*, vol. 20, no. 8, pp. 569–578, 1994.

[27] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 455–464.

[28] A. Egyed, "Fixing inconsistencies in UML models," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 292–301.

[29] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in uml design models," in *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 99–108.