Proceedings Paper:
Webster, D, Townend, P and Xu, J (2012) Interface refactoring in performance-constrained web services. In: Proceedings - 2012 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2012. 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 11-13 Apr 2012, Kingkey Palace Hotel, Shenzhen, China. Institute of Electrical & Electronic Engineers, 111 - 118 . ISBN: 978-0-7695-4643-8. ISSN: 1555-0885.

https://doi.org/10.1109/ISORC.2012.23

Interface Refactoring in performance-constrained Web Services

David Webster, Paul Townend, Jie Xu
School of Computing,
University of Leeds
Leeds, UK
{D.E.Webster, P.M.Townend, J.Xu}@leeds.ac.uk

Abstract— **This paper presents the development of REF-WS an approach to enable a Web Service provider to reliably evolve their service through the application of refactoring transformations. REF-WS is intended to aid service providers, particularly in a reliability and performance constrained domain as it permits upgraded *'non-backwards compatible'* services to be deployed into a performance constrained network where existing consumers depend on an older version of the service interface. In order for this to be successful, the refactoring and message mediation needs to occur without affecting functional compatibility with the services' consumers, and must operate within the performance overhead expected of the original service, introducing as little latency as possible. Furthermore, compared to a manually programmed solution, the presented approach enables the service developer to apply and parameterize refactorings with a level of confidence that they will not produce an invalid or 'corrupt' transformation of messages. This is achieved through the use of preconditions for the defined refactorings.**

Keywords- Web Services, Refactoring, Reliable Evolution

I. INTRODUCTION

Software evolution is a fundamental part of software engineering and its need to be addressed has been well understood for many years [1]. Activities involved in software evolution range from the repair of defects to the addition of new functionality to a system.

Modern computing systems are moving towards a Software as a Service (SaaS) paradigm, which can be characterized as a separation of the possession and ownership of software from its use [2]. One of the motivations for using Software as a System (SaaS) is for software developers to be able to respond to the industrial needs of evolving software in contrast to traditional software maintenance processes. Such processes are viewed as too slow to meet these rapidly evolving needs [3]. Whilst this separation has its benefits, the downside of this arrangement is a loss of control over the service software from the consumer's perspective compared to locally running software.

In the environment of an open marketplace for services it is common for a service provider to offer the provision of service to consumers without knowing specific details about them, for example a contact address. This is in contrast to a consumer contracting a service provider to provide a bespoke service specifically for that consumer. A fundamental challenge to evolving a service (and its interface) in this setting, therefore, is that the service provider may not know the contact details or otherwise be practically able to contact each consumer of their service to notify them of changes to their interface or specification [4].

Depending on the agility of the consumer software's maintainer (in its response to update the software in order to comply with the new service interface) and the speed of the change to the service's interface there is a risk of a delay between this change and the adaptation and redeployment of the consumer software. This is problematic when the consumer assumes that the service interface will remain functionally static leading to what Jones et al [5] describes as an *'abnormal event'* that leads to a *'fault'*. In addition to just purely functional changes to the service, consumers may be dependent on QoS characteristics of the service, for instance the real-time performance and latencies.

A research challenge that this paper aims to address, therefore, is in resolving the conflict between a service provider that needs to evolve its interface to accommodate new requirements and the assumption from a set of service consumers who either assume that the service interface will remain unchanged [5-7] or is unable to adapt due to constraints such as development agility.

The scope of our work is demonstrated by the upgrade to an exemplar workflow that dynamically integrates sensors from an ubiquitous sensor network for the purpose of region surveillance, constrained by run-time QoS (latency) requirements. This is presented through a graphical mapping interface for the human decision maker that defines the desired response time. Within the scale of the exemplar, the performance overhead of executing the refactoring transformations has been demonstrated to be significant.

A. Refactoring for Evolution

'Refactoring' [8, 9] is a commonly used engineering practice in software evolution and is viewed as a prerequisite to adding new functionality or features to software systems. Often this is performed to ease the effort required to extend the software by improving its design first before adding new features. Survey work conducted by Dig [10] concluded that of API changes that break compatibility with existing applications, over 80% of these changes are refactorings.

B. The Challenge of Applying Refactoring to Web Services

As described earlier, the motivation of the work within this paper is to use the technique of refactoring to help address the challenge of enabling the service provider to evolve their service over time and maintain backwards-compatibility. Despite being commonly used in the Object-

Oriented Programming (OOP) domain and widely supported by OOP software development tools, the technique of refactoring has yet to be explored in the Web Services domain. A resolution to this problem, however, is not just a trivial case of recoding existing OOP refactoring tools and warrants deeper analysis of the models that need to be refactored in Web Services compared to OOP. This is in addition to evaluating the performance overheads of any proposed solution. In this paper we introduce our approach to solve this problem, titled REF-WS.

The outline of the remainder of this paper is as follows. Section II outlines the motivating scenario based on an extension to a sensor integration scenario based on our previous work. Section III details the modelling challenges inherent with refactoring Web Services and utilizes the OMG's MOF as a point of reference. Section IV details our approach for refactoring the service interface's structural model in addition to our assumptions and scope. Section V details the implementation of this approach. Finally, Section VI concludes the work and outlines scope for future work.

II. MOTIVATING SCENARIO

In this section an exemplar is presented that demonstrates the evolution of the interface to a Web Service based on a sensor integration scenario. This exemplar is informed by and extends the scope of our previous published work based on a software demonstrator conducted for the completed NECTISE programme [11].

In the demonstrator system a workflow is run to dynamically integrate simulated sensors from an ubiquitous network for the purpose of region surveillance for a rapidly moving Point of Interest (POI). This is presented through a time-sensitive graphical mapping interface for the human decision maker. The desired response time for the user defined by the mapping interface governs the overall completion time of the workflow. The impact of this is that in order to satisfy the desired response time for the user the workflow needs to be able to contact the available sensors, retrieve their responses in the presence of latencies and then aggregate these responses within the pre-defined time constraint. If a sensor fails to respond during the requested time then that particular request thread from the workflow is terminated and the collection of returned responses are then aggregated and returned to the map client. **Figure 1** provides an overview of the sensor integration workflow.

In this paper we extend that work by performing an upgrade on the sensors which requires a non-backwards compatible change to the service interface to be made. Version 1 of the sensors provide the coordinates of a single detected POI. In the upgraded sensors (Version 2), this POI information is augmented with the geo-location of the sensor itself in addition to the time that it detected the POI and requires a structural change to the service interface. The difference between the interfaces can be shown illustrated in **Figure 2**.

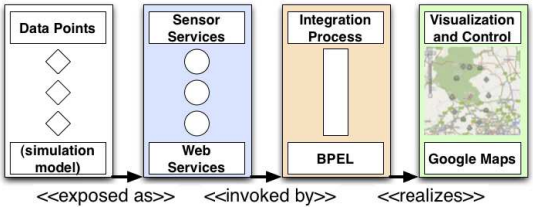

Figure 1: Sensor Integration Workflow

The approach presented in this paper, permits the service developer to refactor Version 1 of the interface in multiple stages until a point where it is declared as Version 2, based on a collection of primitive refactorings. Due to the need to constrain this work's scope, not all possible primitive refactorings are detailed in this paper, however, the exemplar does provide a realistic use-case to drive the scope of the refactorings that are discussed and analyzed in the remainder of this paper.

The QoS requirement of the exemplar furthermore informs the performance evaluation for the refactoring solution. Whilst from the perspective of the service developer the overall refactoring will be at design-time rather than at run-time. From the service consumer's perspective, other than tolerating a downtime due to the redeployment of the upgraded service, the refactored service needs to operate within the QoS constraints defined by the original consumers of the sensor service. As a result of this the evaluation of our approach requires that latency overheads be considered.

Figure 3 illustrates the overhead times that are introduced to enable the transformation between the Version 1 request and response messages to those that the new version of the service consumes and produces. We define these as follows:

$$T_{QoS} < T_{request} + T_{logic} + T_{response}$$

Where T_{QoS} is the response time defined by the service

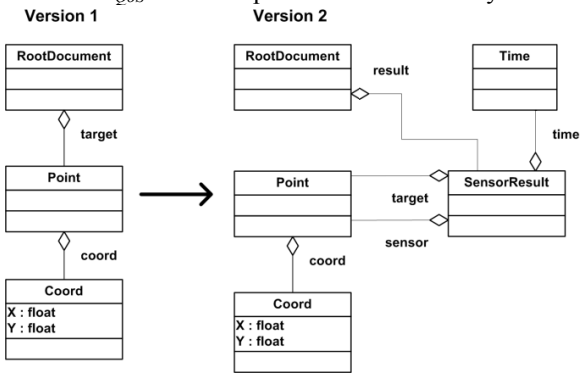

Figure 2: Version 1 and Version 2 of the Sensor Interface Schema

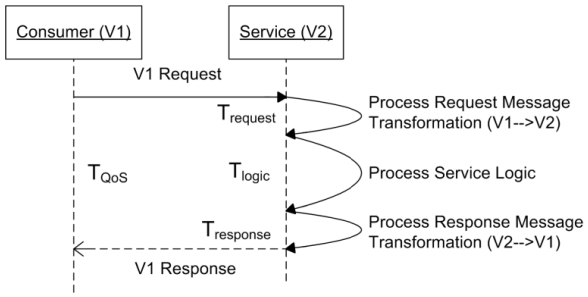

Figure 3: Timing constraints for Web Service

consumer; $T_{request}$ is the time taken for the transformation of the Version 1 request message into a Version 2 request message; T_{logic} is the time taken for the service logic to run; and $T_{response}$ is the time for the time taken for the transformation of the Version 2 response message into a Version 1 response message.

III. MODELLING

In this section the uniqueness of the Web Services models is investigated compared to the more common OOP structural models that are used in refactoring. There are fundamental differences between the two which prevent the simple use of OOP refactoring tools for Web Service refactoring. These differences are summarized as in terms of:

- System boundaries
- Meta-modeling layers

A. System Boundaries

A fundamental difference between the refactoring of Web Services and the refactoring that is typically applied to OOP languages is in the system boundary of the model/code being refactored. Refactoring approaches typically assume that the refactoring system has access to all the method interface code that is being refactored and the all the code that calls this interface. If this is not the case then this interface becomes what Fowler [8] names a *'published interface'*. Despite refactoring often taking place in the internals of software components, many refactorings do change the software interface and break the encapsulation of the change. This change to a published interface is problematic if the software developer does not have access to all client code that consumes the interface as is the case in the Web Services domain. In the scenario of a programming language such as Java this would be comparable to the API provided by a software library.

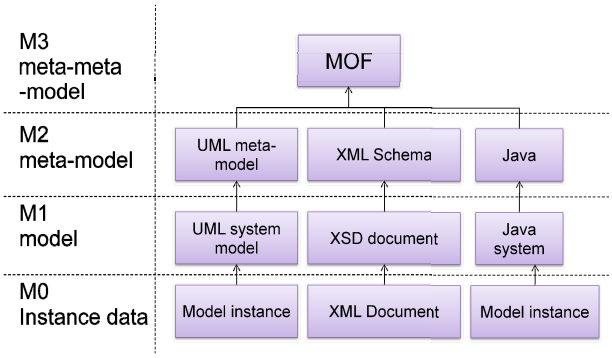

Figure 4: Overview of MOF layers

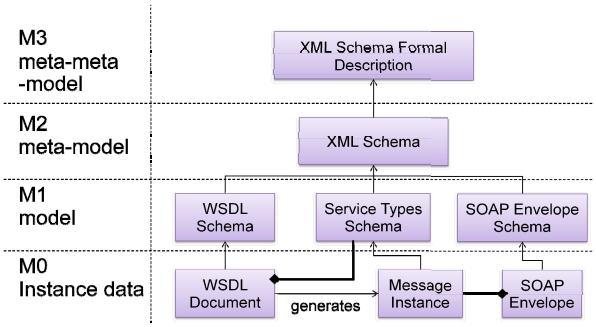

Figure 5: Web Services applied to MOF layers

B. Web Services and Meta-Modelling

At a technical level Web Services are composed of the following group of specifications: WSDL, XML Schema and SOAP. Web Services Definition Language (WSDL) is an XML-based machine-processable interface definition language for describing the operations and messages that are exchanged between a client and service provider [12].

XML Schema is a formal language used to describe and validate the structure of XML documents, and is applied to the WSDL document's message definitions. Finally, the SOAP (initially standing for Simple Object Access Protocol) protocol specification defines an XML-based stateless protocol for the exchange of structured and typed information across web peers, typically in a Web Services environment [13].

To illustrate the comparison between the OOP model (as typically the target of refactoring) and the combination of models that are used to develop and deploy a Web Service, the OMG's MetaObject Facility (MOF) [14] is used as a framework for reference here. The MOF forms a basis for Model-Driven Engineering by defining a layered meta-data framework for defining system models and associated meta-models. These four layers are as follows:

- **Meta-meta-model (M3) :** Defines an abstract language for defining meta-data.
- **Meta-model (M2) :** Defines the languages used for describing the structure and semantics of models. UML is an example of a meta-model.
- **Model (M1) :** Defines the model of the system that we want to describe.
- **Instance data (M0) :** Describes the real-world system information applied to the M1 model.

Figure 4 provides an overview of these layers and the application of different language meta-models.

To use the WSDL document in a Web Service, it needs to be interpreted by a middleware system. This middleware then exposes a Web Service endpoint and generates and receives SOAP messages based on the operations, messages and XML types defined in the WSDL and associated XML Schema document.

In reference to **Section III.A**, unlike refactoring applied purely to OOP source code; applying refactoring to Web Services will require the management of how refactoring changes made to the WSDL document and its SOAP binding propagate through the middleware and into the HTTP/SOAP message sent and received from the service consumer. **Figure 5** illustrates the linking between the various schemas and documents associated with the deployment of a Web Service against the MOF layers.

In summary, applying the technique of refactoring to Web Services is not just a 1:1 mapping to an OOP meta-model, such as Java. Instead we have the XML Schema

meta-model for defining structural elements of messages, WSDL for the operations in addition to the unique parts of WSDL and the SOAP binding, for example the PortType, Binding and Service definition. Of these standards, the XML Schema is the closest to the OOP model in terms of feature overlap, therefore, in this paper we can initially take advantage of this similarity and exploit existing refactoring theory.

IV. Approach and Analysis

In this section our approach to refactoring Web Services (*titled 'REF-WS'*) is introduced and detailed. Based on the refactorings defined by Opdyke [9] in **Figure 6**, we have plotted the applicability the primitive refactorings against the WSDL and XML Schema models. In order to constrain this initial work's scope we focus on addressing refactorings applied to the XML Schema (message structure) part of the Web Service first. As a consequence of this decision, the changes to the operation definitions in the WSDL document and their effects on the middleware are not addressed in this initial work at the current time.

Within the scope of this work, the following assumptions are made:

- Services evolve their functionality whilst building on existing functionality. In this assumption, as services extend their functionality the logic of the previous functionality is not changed substantially. Therefore, functionality of *Vn+1* is a superset of *Vn* whereby services are refactored then evolved.
- Mismatch between interfaces only occurs at the structural level. Protocol mismatches are outside the scope of this work.
- Services are invoked in request/response mode.
- Messages do not contain binary sections, for example raw image data.

Category	Refactoring	XML Schema	WSDL
Creating a Program Entity:	creating an empty class	X	
	creating a member variable	X	
	creating a member function		X
Deleting a Program Entity:	deleting an unreferenced class	X	
	deleting an unreferenced variable	X	
	deleting a set of member funcitons		X
Changing a Program entity:	changing a class name	X	
	changing a variable name	X	
	changing a member function name		X
	changing the type of a set of variables and functions	?	?
	changing access control mode	N/A	N/A
	adding a function argument		X
	deleting a function argument		X
	reordering function arguments		X
	adding a function body	N/A	N/A
	deleting a function body	N/A	N/A
	convert an instance variable to a variable that points to an instance	N/A	N/A
	convert variable references to function calls	N/A	N/A
	replacing statement with function call	N/A	N/A
	inlining (ie inline expanding) a function call	N/A	N/A
	changing the superclass of a class	N/A	N/A
Moving a Member Variable:	moving a member variable to a superclass	X	
	moving a member variable to a subclass	X	

Figure 6: Applicability of Opdyke's primitive refactorings to Web Services

A. Refactoring Transformation Definition

The REF-WS approach proposed in this paper defines refactoring transformations based on two layers relative to the MOF framework. The primitive refactoring transformations themselves apply to the M1 *'Model'* level which affect messages created by the Web Services middleware at the M0 *'Message'* level. As applied to the motivating scenario exemplar in **Section 2**, the three primitive refactoring transformations applied to perform the desired refactoring are as follows:

- **Create new ComplexType.**
- **Create new Element.**
- **Move Element.**

These primitive refactoring transformations when applied to the service interface schema can be applied individually to iteratively transform the Version 1 of the service interface schema illustrated in **Figure 8**. to what is eventually declared to be Version 2. As the transformations are applied in sequence their application can each be checked against their defined preconditions, as is in keeping with the theory of Opdyke's original refactorings. We, therefore, define the preconditions for the refactoring transformations required to implement the exemplar transformation of **Figure 8** in **Figure 9**.

B. Layered Transformations

Due to the Web Service provider's lack of control beyond the service interface boundary; after refactoring there is still a mismatch present between messages sent to and from the service implementing the old interface schema and the service that implements the new refactored schema. This can be contrasted with OOP source code refactoring, where access to code on both sides of the class interface boundary is assumed.

In order to resolve this message mismatch these messages must be transformed. In our approach the specification of this M0 message transformation is derived from the M1 level refactoring that has been applied to the service interface schema. This linkage can be shown diagrammatically in **Figure 7** which illustrates a high-level model for describing the linkage between the messages, schemas and refactoring transformations.

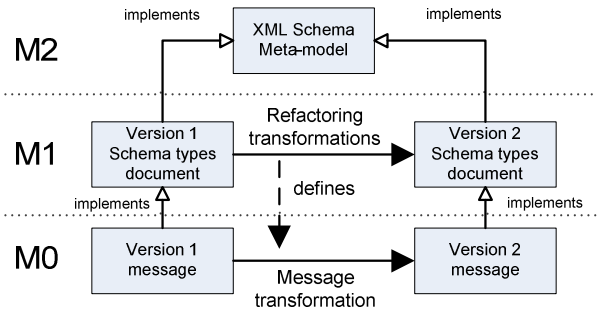

Figure 7: Transformation layers

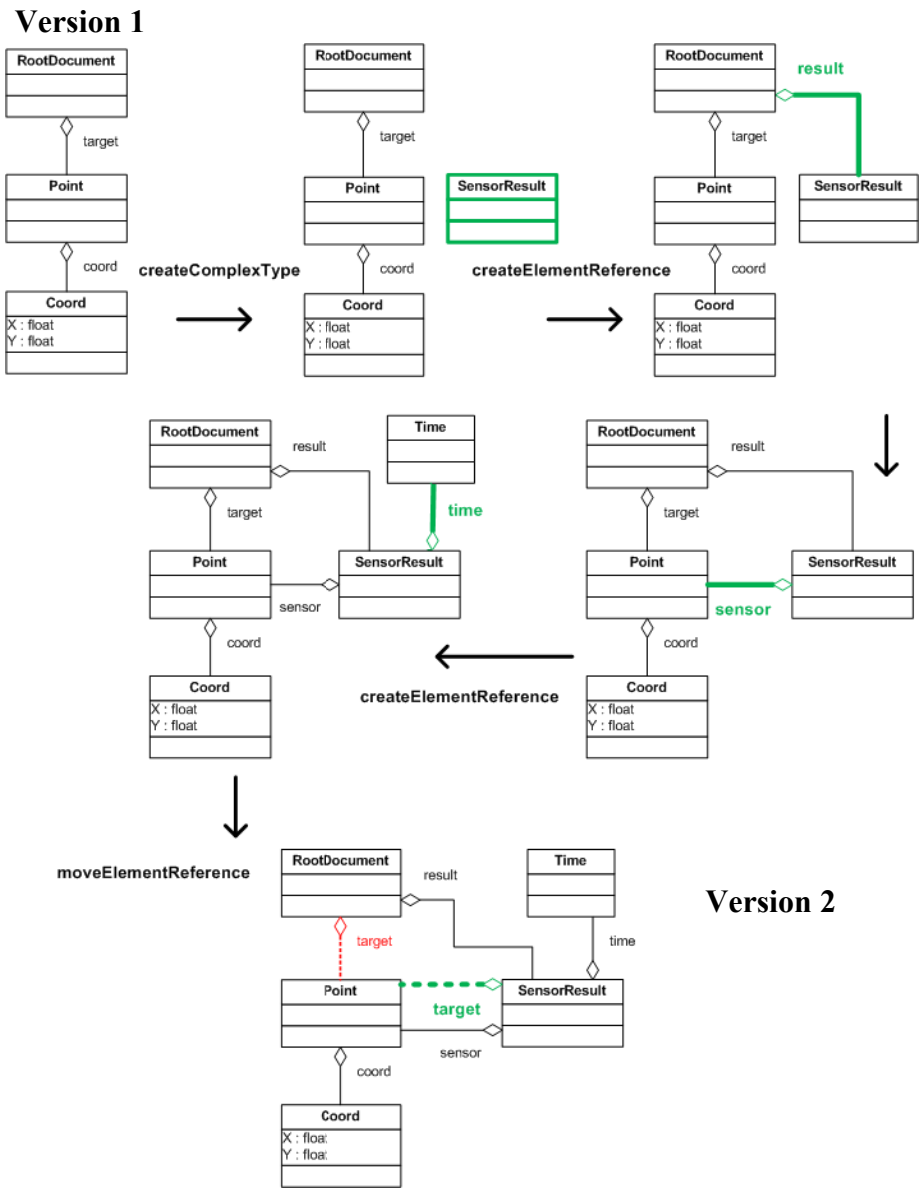

Figure 8: Overview of refactoring the sensor interface structural schema from Version 1 to Version 2

Refactoring	Parameters	Pre-conditions
createNewComplexType	newComplexTypeName	∀ complexType ∈ complexTypes ~hasName(newComplexTypeName, complexType.name)
createNewElement	sourceComplexTypeName targetComplexTypeName newElementName	∃! isComplexType(s) ∧ hasName(s, sourceComplexTypeName) ∧ ∃! isComplexType(t) ∧ hasName(t, targetComplexTypeName ∧ ∃! ~hasElement(s, newElementName)
moveElement	sourceComplexTypeName targetComplexTypeName sourceElementName	∃! isComplexType(s) ∧ hasName(s, sourceComplexTypeName) ∧ ∃! isComplexType(t) ∧ hasName(t, targetComplexTypeName ∧ ∃! hasElement(s, sourceElementName) ∧ ∃! ~hasElement(t, sourceElementName)

Figure 9: Primitive refactorings and pre-conditions

V. IMPLEMENTATION

For the implementation of our approach we have leveraged the Atlas Transformation Language (ATL) associated with the Eclipse project. ATL permits elements in a meta-model to be matched using 'match rules' defined against the source model. A 'target pattern' is optionally defined for the rule and is used to generate elements of the target model. In order to refine the 'match rules' part of the transformation, the OMG's Object Constraint Language (OCL) can be applied; this restricts the elements of the source model that the transformation is applied to.

A. Model-level Transformations

Each of our low level refactorings is implemented in a separate ATL rule respectively. The refactorings that apply to the M1 level are defined against the M2 meta-model. It is crucial to stress at this point that this means that the refactoring transformations are applicable to any XML types schema defined by the XML Schema meta-model, rather than on a specific service's interface. To achieve this generality the names of specific ComplexTypes and Elements are defined by parameters passed to the transformation rules. The scalability of the approach is driven by the definition of primitive refactoring transformations that can be reused and recombined into large refactoring transformations to suit the service developer.

After defining the refactoring rules in ATL the models and meta-models need to be defined in ECore. ECore is the Eclipse equivalent of the OMG's MOF and implements comparable concepts in the M2 meta-model. We take advantage of the Eclipse Modelling suite's included ability to import models defined in XML Schema and convert them to an ECore representation bi-directionally using an ECore 'generator model'. The principal mappings between XML Schema and ECore are as follows: **ComplexType → EClass** and **Element → EReference**.

In addition to these two mappings an additional EClass must be defined. The WS-I Basic Profile standard rule R2204 [15] defines, *"A document-literal binding in a description must refer, in each of its soapbind:body element(s), only to wsdl:part element(s) that have been defined using the element attribute"*. As a consequence of this requirement, a 'DocumentRoot' EClass needs to be created in order to contain this orphan element/EReference.

The refactoring rules that we define in ATL are then repeatedly applied in the desired sequence to the ECore model of the service interface. The rules are defined in ATL's *'refining mode'* to the service interface model representation of ATL. This permits both the source and target models to be instantiations of the same meta-model.

In our approach we map our preconditions for the refactoring to the OCL part of the matched rules. **Figure 10** shows the implementation for the MoveElementReference refactoring in ATL. The 'from' part of the matched rule implements the validity checks for the transformation.

B. Instance-level Transformations

In order to define the M0 instance (message) level transformation for the specific service interface, the transformation needs to be derived from the M1 model-level refactoring transformations automatically and without user intervention. To enable this linkage between the two transformation levels we define a meta-model named SUDO. This is a simple model-based representation of an M0 instance level transformation that can easily allow the generation of a textual representation of the ATL transformation. As per the tail part of **Figure 10**, the M0 level message transformation is derived from the rule's input parameter information and defined against our SUDO meta-model. A simple Java-based script has been developed that will read in the SUDO model and output a usable ATL file.

C. Performance Overview

In order to assess the execution performance of the M0 message transformation, a series of 200 serial executions of the the refactoring transformations required to perform the exemplar's composite refactoring were performed. The aim of this test was to determine the overhead of executing the transformation in addition to the time that the Web Service middleware would take handle the service request (T_{logic}). The experimental setup for each test run was kept identical, hence, we did not expect to see a large degree of variation in

```
-- @nsURI MM=http://www.eclipse.org/emf/2002/Ecore
-- @path MMSUDO=/ISORC2012/metamodels/SUDO.ecore

module MoveElementReference;
create OUT : MM, SUDO : MMSUDO refining IN : MM,
parameters : PARAMS;

rule MoveElementReference  {

  from
    targetClass : MM!EClass,
    sourceClass : MM!EClass,
    sourceReference : MM!EReference
    (
      sourceClass.name =
      thisModule.getParameter('sourceClassName')
                                --'PointType'
      and targetClass.name =
      thisModule.getParameter('targetClassName')
                                --'DocumentRoot'
      and sourceReference.name =
       thisModule.getParameter('sourceReference')
                                --'coord'
      and not targetClass.eStructuralFeatures
              ->exists(ref|ref.name =
      thisModule.getParameter('sourceReference'
    )
  to
    package : MM!EClass
    (
      eStructuralFeatures <- sourceReference
    ),
    --Populate the instance level transform
    scriptRoot : MMSUDO!Script
    (
       rule <- matchRule
    ),
    matchRule : MMSUDO!Rule
    (
      sourceModelMatch <-
      thisModule.getParameter('sourceClassName'),
      moveElement <- moveElementRule
    ),
    moveElementRule : MMSUDO!MoveElement
    (
      element <-
      thisModule.getParameter('sourceReference'),
      target <-
      thisModule.getParameter('targetClassName')
    )
}
```

Figure 10: MoveElementReference implementation in ATL

the performance results. The test system was a dual-core 3GHz Intel Core 2 Duo CPU with 6GB RAM running Windows 7. Following our test of the $T_{response}$ overheads, we have observed that the majority of executions of the message transformation introduced less than a millisecond of overhead and fall well within our T_{QoS} threshold of 2 seconds for the user interface response time threshold. The results, however, indicate three peaks in the overhead message transformation time; however, given the identical experimental setup, we put this down to competing processes on the operating system. The results demonstrate that for the low-level refactoring tasks, such as those presented against the exemplar, this performance overhead can be managed and falls well within the QoS threshold of the map client. **Figure 11** shows the resultant graph of the tests. Based on this, however, further work needs to be conducted to assess the performance overhead when larger chains of refactorings are used in addition to larger message payloads.

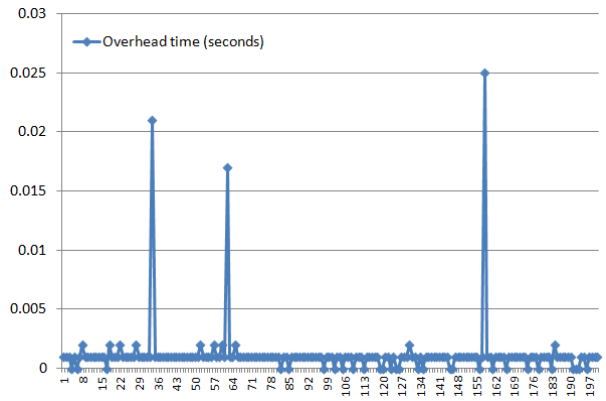

Figure 11: Time overhead of transformation execution

VI. Existing Work

Existing work that can inform a solution to the Web Service evolution problem can be divided into two areas. These can be described as:

1. Mapping between schemas of functionally similar equivalent services [16-19]
2. Refactoring OOP code [10, 20-23].

Work conducted under the (1) category, is motivated by the scenario of multiple heterogeneous service interfaces being used in services that provide similar functionality from a consumer's perspective. The approaches, therefore, aim to revolve the interface mismatch associated with this heterogeneity through mappings.

A characteristic for approaches in (1) is that the mapping is performed 'after-the-fact' with respect to the creation of the new/alternate service interface. A benefit of this is that the mappings are able to take advantage of *'a posteriori'* knowledge for example instance data for each schema. The downside of this approach is the lack of provenance engineering knowledge of how the service interface evolved when considering evolved services rather than heterogeneous services from differing providers. As a result, the mappings are non-deterministic given differing interfaces and need to be evaluated on their accuracy.

The mechanism behind the majority of these approaches involves referencing the service interface elements to a common ontology through a combination of 'lifting' and 'lowering' mappings. Whilst this work aims to semi-automate mappings between schemas, the set of mappings supported by these approaches are limited and do not define a clear methodology for defining new mappings. Approaches to mapping are generally limited to moving elements between schemas with the more advanced work being able to semi-automate string splitting and concatenation in addition to other more advanced operations, for example, key-value mapping and default value insertion. The main limitations of existing work are primarily that they do not provide a clear methodology on how to deal with more complex mappings, for instance mappings involving numerical transformation. Secondly is the major assumption in relying that a high-level domain model (often expressed as an ontology) has been created for the document types that the schemas substantiate. In the case of [18] this is captured through electronic questionnaires.

The approaches associated with category (2) aim to reduce the disruption to software library consumers associated with the upgrading of OOP software libraries. These are motivated as a response to the cost and error-prone nature of manually developing mediation components between OOP software interfaces [23] and aim to provide a repeatable and deterministic engineering approach. These approaches involve the recording of human directed and piecemeal code refactoring in order to generate a mediator to translate calls for the old code into to calls to the new code. A limitation with this work is that the techniques documented are presented against a specific programming language, for instance, Java.

VII. Conclusions and Future Work

In this paper we have presented an approach to enable Web Service developers for the first time to evolve their service interfaces in a reproducible and safe manner through refactoring. We have also conducted an initial evaluation of the execution time overhead of the message transformation produced from this refactoring when applied to an exemplar.

A. Conclusions

The approach permits upgraded 'non-backwards compatible' services to be deployed into a network where existing consumers depend on an older version of the service interface. The solution that we have provided has been motivated by the difference in the models inherent to Web Services compared to the more commonly used OOP model used in traditional refactoring. A major contribution of our work, therefore, is the original application of refactoring in the previously unexplored Web Services domain.

The scope of this work has been defined through the use of an exemplar and has supported the definition of a small reusable set of low-level refactorings that can be parameterized for application to particular service interface instances. The definition of a set of pre-conditions, in

keeping with Opdyke's original work (that defined the research field of refactoring) enables the service developer to apply and parameterize Web Service refactorings with a level of confidence that these will not produce an invalid or 'corrupt' transformation on the message and can be performed in a reproducible manner.

Performance evaluation for the approach is based on the user's expected QoS of response time with latency constraints driven from the user interface in the use-case. A challenge addressed here is that significant time overheads for the message transformation would limit the approach's use in a production environment; however, within the scale of the exemplar the performance overhead of executing the transformation has been demonstrated not to be significant. This does leave scope, however, to conduct further testing with the transformation of larger SOAP messages.

B. Future Work

As this paper represents our initial work in addressing the service refactoring problem, there are a number of limitations to its scope that have been documented along the way. Primarily is the decision to limit the applied refactoring in the exemplar to just the XML Schema defined part of the service interface and omitting the ability to perform changes to the operation names in the WSDL document. The motivation behind this decision is so that we could address the part of the interface which most closely resembles the Class→Association relationship that exists in OOP systems, for instance Java, before broadening the scope of our work and building on established refactoring theory.

Other than the basic pre-conditions described here, we intend to expand on these and apply tool-supported model checking to assure the correctness of more complex refactorings. This will be more useful when one aims to determine whether the correctness of low-level refactorings still holds true when composed into higher level refactorings.

In this paper we have developed refactorings that apply at the M1 model layer; however, there is scope for developing refactorings that apply at the M0 (message) layer. Examples of these include unit conversion and geospatial coordinate transformations. To be able to evaluate these would require the development of an evaluation framework to guarantee that information capacity has been preserved over the transformations and that fidelity has not been lost in the richness of the information.

ACKNOWLEDGMENT

The work reported in this paper has been supported in part by the NECTISE programme jointly funded by BAE Systems and the U.K. EPSRC Grant EP/D505461/1, the UK EPSRC WRG platform project (No. EP/F057644/1), the National Basic Research Program of China (973) (No. 2011CB302602), the UK TSB STRAPP project (No. 1926-19253), and the Major Program of the National Natural Science Foundation of China (No. 90818028).

REFERENCES

[1] M. M. Lehman, "Laws of Software Evolution Revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, 1996.

[2] M. Turner, Budgen, D., Brereton, P., "Turning Software into a Service," *IEEE Computer Society,* vol. 36, 2003.

[3] K. Bennett and N. Gold, "Achieving ultra rapid evolution using service-based software," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, 2001,.

[4] G. Canfora and M. D. Penta. (2006) Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*. 10-17.

[5] C. Jones*, et al.*, "A Structured Approach to Handling On-Line Interface Upgrades," in *26th Annual International Computer Software and Applications Conference*, Oxford, England, 2002.

[6] K. Sycara*, et al.*, "Dynamic discovery and coordination of agent-based semantic Web services," *Internet Computing, IEEE,* vol. 8, pp. 66-73, 2004.

[7] L. Baresi*, et al.*, "Toward Open-World Software: Issue and Challenges," *IEEE Computer,* vol. 39, IEEE Computer Society, 2006.

[8] M. Fowler*, et al.*, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley Professional, 1999.

[9] W. Opdyke, "Refactoring Object-Oriented Frameworks," 1992.

[10] D. Dig and R. Johnson, "The Role of Refactorings in API Evolution," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, 2005.

[11] D. Webster*, et al.*, "Migrating Legacy Assets through SOA to Realize Network Enabled Capability," in *New Directions in Web Data Management 1*, A. Vakali and L. C. Jain, Eds., First ed: Springer, 2011.

[12] D. Booth*, et al.*, "Web Services Architecture," W3C, 2004.

[13] W3C, "SOAP Version 1.2 Part 0: Primer (Second Edition)," http://www.w3.org/TR/soap12-part0/, 2007.

[14] Object Management Group (OMG) "Meta Object Facility (MOF) Core Specification. Version 2.4.1," 2011.

[15] WSI Web Services-Interoperability Organization, "Basic Profile Version 1.1," 2006.

[16] T. De Giorgio*, et al.*, "SAWSDL for Self-adaptive Service Composition," in *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*. vol. 5872: Springer Berlin / Heidelberg, 2009, pp. 907-916.

[17] L. Cavallaro and E. Di Nitto, "An approach to adapt service requests to actual service interfaces," presented at the Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, New York, NY, USA, 2008.

[18] C. Drumm*, et al.*, "Quickmig: automatic schema matching for data migration projects," presented at the Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, Lisbon, Portugal, 2007.

[19] F. Scharffe*, et al.*, "Ontology mediation patterns library v2. Deliverable D4.3.2," SEKT project (IST-2003-506826), 2005.

[20] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," presented at the Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005.

[21] S. Roock and A. Havenstein, "Refactoring Tags for automatic refactoring of framework dependent applications," presented at the Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP), 2002.

[22] M. Blaha and W. Premerlani, "A catalog of object model transformations," in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, 1996, pp. 87-96.

[23] J. H. Perkins, "Automatically generating refactorings to support API evolution," *SIGSOFT Softw. Eng. Notes,* vol. 31, pp. 111-114, September 2005.