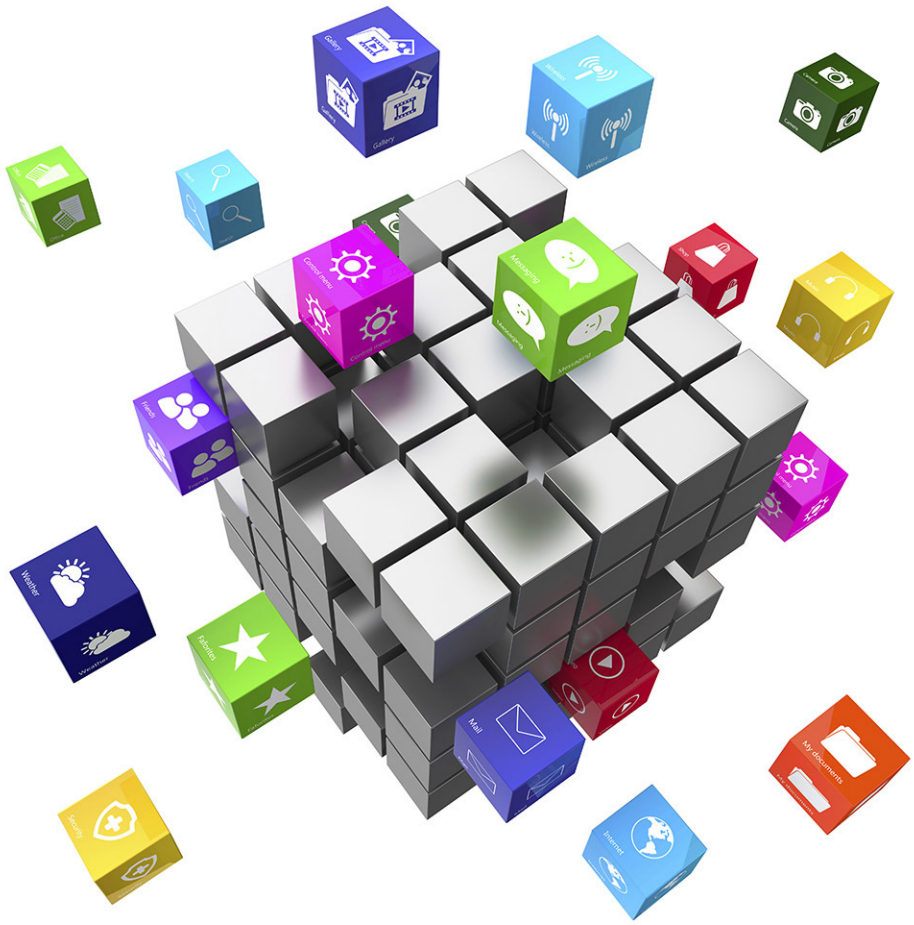
EPRI | ELECTRIC POWER RESEARCH INSTITUTE

Program on Technology Innovation: Web Services Robustness

An Agile Data Integration Ecosystem

3002018641



Program on Technology Innovation: Web Services Robustness

An Agile Data Integration Ecosystem

3002018641

Technical Update, September 2020

EPRI Project Manager

S. Crimmins

ELECTRIC POWER RESEARCH INSTITUTE
3420 Hillview Avenue, Palo Alto, California 94304-1338 ▪ PO Box 10412, Palo Alto, California 94303-0813 ▪ USA
800.313.3774 ▪ 650.855.2121 ▪ askepri@epri.com ▪ www.epri.com

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES

This is an EPRI Technical Update report. A Technical Update report is intended as an informal report of continuing research, a meeting, or a topical study. It is not a final EPRI technical report.

NOTE

For further information about EPRI, call the EPRI Customer Assistance Center at 800.313.3774 or e-mail askepri@epri.com.

ACKNOWLEDGMENTS

The Electric Power Research Institute (EPRI) prepared this report.

Principal Investigators
S. Crimmins
D. Lowe

This report describes research sponsored by EPRI.

EPRI acknowledges the contributions of Emma Batson in the development of this report.

This publication is a corporate document that should be cited in the literature in the following manner:

Program on Technology Innovation: Web Services Robustness, An Agile Data Integration Ecosystem. EPRI, Palo Alto, CA: 2020. 3002018641.

ABSTRACT

Web Services are one of the best ways to implement communication from application to application (A2A) and from business to business (B2B). They are widely deployed and increasingly standardized, however, their success can bring its own problems. As the ecosystem of services evolves, either the producers are saddled with the costs of supporting multiple versions of each service or the consumers are forced to deploy and regression test as each new version is released—even when the new features are not required. To support the rapidly evolving Integrated Energy Network (IEN), this research will identify design patterns and web service technologies that create a robust web service ecosystem of decoupled systems that can evolve independently at least cost.

This research explores how a web service consumer could design the receiving system to continue to process a message even when it includes additional data that are not part of its original schema definition file (XSD). In this way, the producer is free to add additional data to its message without worrying about the impact to existing consumers or bearing the cost of supporting multiple versions. Consumers are freed from moving in lock-step with the publisher and can take advantage of new features on their own schedule.

Keywords
Data integration
Service Oriented Architecture
SOAP
Tolerant Reader
XSD
Web Service

Deliverable Number: 3002018641

Product Type: Technical Update

Product Title: Program on Technology Innovation: Web Services Robustness, An Agile Data Integration Ecosystem

PRIMARY AUDIENCE: Enterprise architects

SECONDARY AUDIENCE: Software architects, system integrators

KEY RESEARCH QUESTION

How can web service producers and consumers be completely decoupled so that producers can add new data elements as quickly as they need to without impacting existing consumers who may not be ready for the new information? Although versioning addresses the issue up to a point, supporting multiple versions has a high cost on the producing system that can be remedied by making the consuming applications tolerant of new data elements. Is it possible to design the receiving system so that it successfully consumes required data while ignoring superfluous elements?

RESEARCH OVERVIEW

This research explores how a web service can be prevented from breaking if the producing system sends data that the web service is not designed to handle. Before implementing a solution, a web service was created using a common web services framework to show that it would break upon receiving a message with extra elements not contained in the original schema definition. After that, the auto-generated code was enhanced to ensure that the web service would successfully process the payload of an incoming message regardless of extraneous data provided.

KEY FINDINGS

- Code auto-generated by a common web services framework will break when data elements are added.
- Regression testing and re-deployment of services add significant overhead to a web services ecosystem that can create inertia in the system that reduces agility.
- The auto-generated code can be enhanced to handle new data elements that allow the producer to evolve independently of the consumer—they become decoupled.
- The nature of enhancing frameworks may differ, but the end result of obtaining a more robust service can still be achieved.

WHY THIS MATTERS

Web Services are one of the best ways to implement communication from application to application (A2A) and from business to business (B2B). They are widely deployed and increasingly standardized; however, their success can bring its own problems. As the ecosystem of services evolves, either the producers are saddled with the costs of supporting multiple versions of each service or the consumers are forced to deploy and regression test as each new version is released—even when the new features are not required. To support the rapidly evolving Integrated Energy Network (IEN), this research will identify design patterns and web service technologies that create a robust web service ecosystem of decoupled systems that can evolve independently at least cost.

HOW TO APPLY RESULTS

Web service developers can use the methods detailed in this report to design their applications in a more robust manner, allowing for additional payload data to be received without breaking the application. Details are provided for both .NET and Java implementations, but the overall concept can be applied to more than just those two programming languages.

LEARNING AND ENGAGEMENT OPPORTUNITIES

The Technology Innovation program may extend research in this area in 2021. Potential areas of research include an implementation of Representation State Transfer (REST) web services or methods for creating a robust web service ecosystem in a multi-company, heterogeneous environment. EPRI seeks guidance on the most valuable next steps.

EPRI CONTACT: Sean Crimmins, Principal Technical Leader, scrimmins@epri.com

PROGRAM: Enterprise Architecture and Integration, P161E

Electric Power Research Institute
3420 Hillview Avenue, Palo Alto, California 94304-1338 • PO Box 10412, Palo Alto, California 94303-0813 USA
800.313.3774 • 650.855.2121 • askepri@epri.com • www.epri.com

CONTENTS

LIST OF FIGURES

1
BACKGROUND

Web Services are one of the best ways to implement communication from application to application (A2A) and from business to business (B2B). They are widely deployed and increasingly standardized. However, as the ecosystem becomes more successful, the number of service consumers increases and its ability to evolve is hindered by the ability of consuming systems to handle additional data elements.

Web service frameworks will auto-generate code to receive data from a web service. While this initially increases the productivity of developers, the code produced is often brittle—it will break if it receives data not in the original specification. This presents a problem to applications that publish data via web services.

As the ecosystem of services evolves, either the producers are saddled with the costs of supporting multiple versions of each service or the consumers are forced to deploy and regression test as each new version is released—even when the new features are not required. These costs create an inertia that resists change or drives complexity through the creation of ever-specialized web services with a small number of consumers.

If the consumers can be coded in a way that is tolerant of new data elements, the costs of versioning and unnecessary code deployments are eliminated. To support the rapidly evolving Integrated Energy Network (IEN), this research will identify design patterns and web service technologies that create a robust service ecosystem of decoupled systems that can evolve independently at least cost.

2
THE TOLERANT READER DESIGN PATTERN

The concept of a robust web service consumer—one that can survive the receipt of unexpected information—was originally discussed by Martin Fowler in his blog post on the Tolerant Reader design pattern:[1]

> My recommendation is to be as tolerant as possible when reading data from a service. If you're consuming an XML file, then only take the elements you need, ignore anything you don't.

It has since been formalized by Robert Daigneau in the book *Service Design Patterns*:[2]

> How can clients or services function properly when some of the content in the messages or media types they receive is unknown or when the data structures vary?

This design pattern describes a specific application of robustness in computer science: "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions."[3]

---

[1] https://martinfowler.com/bliki/TolerantReader.html

[2] http://www.servicedesignpatterns.com/

[3] https://ieeexplore.ieee.org/servlet/opac?punumber=2238

3
AUTO-GENERATED CODE AND WEB SERVICES

Web service frameworks, such as Apache CXF,[4] can auto-generate code based on a web service specification such as a Web Services Definition Language (WSDL) file. While this approach saves upfront development time, it ties the consuming system's code to the service definition provided by the producing system. When—not if—the web service definition changes, the receiving code will break. The two systems have become tightly coupled, defeating one of the reasons for using web services in the first place. To decouple the two systems, the receiver must become tolerant of changes to the service specification. It must become robust enough to handle unexpected input and continue—as long as the required data are available—to function correctly.

Research Methodology

This project set out to confirm that a web service implementation generated using a common web service framework would break if it received data elements not in the original specification. Research then confirmed that it is possible to adjust the auto-generated code so that it does not break when it receives additional data elements. The web service is now tolerant to changes in the data it receives; in other words, it is robust.

Research Steps

The steps described in detail in the next section can be summarized as follows:

1. Auto-generate a consuming web service from a WSDL and confirm that it works.
2. Add a data element to the XSD and confirm that the consuming service fails.
3. Enhance the auto-generated code and confirm that it works for the original data.
4. Reintroduce the additional data element and confirm that the consuming service works.

The remainder of this report provides concrete evidence for the fragility of auto-generated code and demonstrates how to enhance the code to create a tolerant, robust web service consumer.

[4] http://cxf.apache.org/

4
JAVA IMPLEMENTATION

Web Service Background

The SOAP web service implemented for this research is based on the draft standard for IEC 61968-5, Distributed Energy Optimization. Three WSDLs were used for this web service: one for creating distributed energy resources (DER) groups, one for querying DER groups, and one for changing or deleting DER groups. To better illustrate this test case, a scenario is applied where the **DERFunction** element is updated to include **DERNameplate** information on the sending system to accommodate updates to the IEC 61968-8, Application Integration at Electric Utilities – System Interfaces for Distribution Management, schemas.[5] However, the receiving system has not yet made the updates on its side to support this new element in the schema. The result of the sending system providing this extra information for which the receiving system is not prepared is a complete failure on the part of the receiving system to process the message it has received.

The Frameworks and Tools Used

Although SOAP is a language-agnostic standards protocol, Java is one of the more common languages used for implementing a SOAP web service. To replicate the research findings of this report, the following APIs, frameworks, and dependencies were used. The results of this research should be achievable regardless of which programming stack a web service developer uses to construct a web service.

- **JAX-WS:** JAX-WS (Java API for XML Web Services) is a Java programming language API for creating web services, particularly SOAP web services. For RESTful web services, the JAX-RS library would be used.
- **JAXB:** JAX-WS (Java Architecture for XML Binding) is a software framework that maps Java classes to XML representations. This is necessary not only for converting the XML represented within the XSDs and WSDLs to code for creating the service, but also for doing the inverse of changing received Java code into XML. These processes are referred to as *unmarshalling* and *marshalling*, respectively.
- **MySQL:** MySQL is an open source relational database management system (RDMS), used in this research to store EndDeviceGroups and EndDevices as described by the IEC 61968-5 WSDLs and XSDs.
- **Hibernate ORM:** The Hibernate Object-Relational-Mapping framework is an implementation of the Java Persistence API (JPA) specification and is used for mapping Java classes to a relational database such as MySQL.
- **Wsdl2java:** The Apache CXF tool wsdl2java was used to generate fully annotated Java code from a specified WSDL document for web service implementation.

[5] https://www.epri.com/#/pages/product/000000003002016045/?lang=en-US

- **Spring framework:** The spring framework is the backbone of the project, handling everything needed to create a Java enterprise application. Spring itself is multiple projects, each handling its own set of services (for example, spring-web for hosting the web service at a specified endpoint or spring-orm for database persistence).
- **Apache Maven:** Maven is a tool for building and managing Java projects. All dependencies above are listed in a configuration file named *pom.xml*, which Maven uses to download the proper dependencies needed to build the web service. This allows for a project to be built across multiple systems and can be easily imported to other integrated development environments (IDEs) such as Eclipse, NetBeans, and IntelliJ.

These tools are commonly used in the creation of a SOAP web service in Java. Without much customization other than the code to implement the WSDL's defined operations, this web service was used as the basis for Test Case 1: Sending System Requests Creation of DER Group with DERNameplate Information Provided (Request Fails).

Creating and Breaking the Web Service

The first step toward solving any problem is to reproduce the problem. That meant creating a web service that would break upon receiving additional elements it was not expecting. To build this service, Java was chosen as the programming language. The libraries and frameworks described in Section 3 were used as is, with no modifications made other than what was needed to connect to the MySQL database. This process is described at a high level step-by-step next and can be replicated using any other set of WSDLs/XSDs or even a code-first approach.

1. A Maven project was created using the Eclipse IDE. Note that any other IDE such as NetBeans or IntelliJ could be used instead.
2. All dependencies needed for the web service were added to the *pom.xml* file. This includes but is not limited to:
   - Hibernate
   - CXF
   - Spring-core
   - Spring-orm
   - Jax-ws-api
   - Log4j
3. A MySQL database was set up to accommodate data according to the IEC 61968-5 profile's *ExecuteDERGroups.wsdl*.
4. The cxf-codegen-plugin (see https://cxf.apache.org/docs/maven-cxf-codegen-plugin-wsdl-to-java.html) was used to generate Java artifacts from the WSDLs/XSDs.
5. Hibernate's *hbm2java* was used to generate Java artifacts from the database tables.
6. Code was developed to implement the operations defined by *ExecuteDERGroups.wsdl*.

The result was a web service that would create DER groups as described by the WSDL. This web service worked as expected when a sending system submitted a request that contained elements defined by its XSDs. However, upon adding additional elements to represent changes

in the IEEE 1547:2018 standard for DER devices, the receiving system was unprepared for this new data and returned an error to the sending system.

Enhancing the Web Service

The desired outcome of this research was to find and implement a method allowing a web service to continue processing a message's payload when unexpected elements appeared. If the receiving system has no immediate need for new data it is not expecting, this would allow it to continue to process an incoming message—permitting the sending and receiving systems to evolve independently of one another.

Enabling the web service to accept additional elements requires some understanding of how the code behind-the-scenes works. A web service may be implemented in a programming language such as C# or Java, but the message being sent between systems is XML, as required by the SOAP standard. When that message is received, it is then unmarshalled, or converted, from XML to Java or C#. Auto-generated code created to represent each element in the XSDs is used to convert a received message into a Java or C# object. Figure 4-1 illustrates how an XML snippet is converted into Java.

```
/**
 * <p>Java class for DERGroups complex type.
 *
 * <p>The following schema fragment specifies the expected content contained
within this class.
 *
 * <pre>
 * &lt;complexType name="DERGroups">
 *   &lt;complexContent>
 *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
 *       &lt;sequence>
 *         &lt;element name="EndDeviceGroup"
type="{http://iec.ch/TC57/2017/DERGroups#}EndDeviceGroup"
maxOccurs="unbounded"/>
 *       &lt;/sequence>
 *     &lt;/restriction>
 *   &lt;/complexContent>
 * &lt;/complexType>
 * </pre>
 *
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DERGroups", namespace =
"http://iec.ch/TC57/2017/DERGroups#", propOrder = {
    "endDeviceGroup"
})
public class DERGroups {

    @XmlElement(name = "EndDeviceGroup", required = true)
    protected List<EndDeviceGroup> endDeviceGroup;

    public List<EndDeviceGroup> getEndDeviceGroup() {
        if (endDeviceGroup == null) {
            endDeviceGroup = new ArrayList<EndDeviceGroup>();
        }
        return this.endDeviceGroup;
    }

}
```

Figure 4-1
Auto-generated code representing a complex group within an XSD

When the receiving system is sent an element that does not match the auto-generated code, an error is thrown and unmarshalling stops. The implementation code for the WSDL operations is never reached. For a receiving system to avoid this error, there are two options.

The first option is to add support for these unknown elements, either by changing the WSDLs/XSDs and rebuilding or by manually modifying the auto-generated code representing

the new XML element. This option is less desirable because the receiving system may not even care about the data it is being sent and changing auto-generated files will lead to data loss should the web service need to be rebuilt for any reason.

The second option is to modify the class responsible for catching errors in the unmarshalling process and add code that handles errors however the developer needs. This was the chosen option for addressing the problem of receiving additional data elements. By modifying the *DetailValidationEventHandler* class, errors encountered in the unmarshalling process can be logged and then ignored instead of using the default behavior of throwing an exception and stopping the unmarshalling process.

Extra precautions should be taken when modifying this class, however, because this class handles all errors encountered during the unmarshalling process. For example, if a sending system did not include an element that the receiving system requires, this same class would be responsible for catching the error. In cases such as these, standards such as those described in IEC 61968-100:2013, Application integration at electric utilities – System interfaces for distribution management, require that the error be caught and returned with a proper error code, while other systems may simply prefer to return a more generic SOAP error message.

Using the second option, the receiving system was able to successfully process a message from the sending system that contained additional elements, saving that error to a log file while continuing to process the message payload. For more details of example use cases, see Appendix A.

5
.NET IMPLEMENTATION

To demonstrate robustness in the .NET Framework, a web service analogous to the one used to demonstrate robustness in Java was developed. It was generated from the same WSDLs and XSDs from the draft standard IEC 61968-5, Distributed Energy Optimization, and tested against requests that were either compliant or included the additional **DERNameplate** element that the service had not been updated to accept.

When using the default .NET Framework architecture, the web service handled noncompliant messages in unpredictable ways. Messages with additional elements would either be processed while ignoring additional elements completely or break upon additional elements, depending on their location. The requirement to include certain elements in the message was not enforced by the web service. We were able to develop the service so that it would give appropriate errors for missing mandatory elements while logging and otherwise ignoring extra elements in the message payload.

Frameworks and Tools Utilized in .NET

Microsoft's .NET Framework is often used to create web services. Although the .NET tools have similar functionality to the Java tools, there are differences in the way .NET facilitates the addition of custom behaviors. Namely, Microsoft provides extensible "dispatcher" classes that act at different points in the process of retrieving and processing SOAP messages. These .NET classes and their potential use for increasing the robustness of web services will be explored later in this section.

The tools commonly used in .NET are almost directly analogous to those used in the Java research:

- **WCF:** WCF (Windows Communication Foundation) is a framework provided by .NET for creating SOAP web services. WCF automatically handles everything from receiving the SOAP message to changing the XML into C# code. The whole process of retrieving messages from the channel is handled by dispatchers and can be customized through extensions.
- **System.Xml:** In a WCF web service, WCF calls System.Xml.XmlSerializer to convert between XML and the defined C# classes. C# achieves this through serialization, which is effectively synonymous to Java's marshalling. System.Xml also provides different tools for parsing, navigating, and modifying XMLs, such as XmlReader and XmlDocument. These classes are present in the standard .NET Framework.
- **Microsoft SQL Server:** Microsoft SQL Server is Microsoft's relational database management system (RDMS), used in this research to store EndDeviceGroups and EndDevices as described by the IEC 61968-5 WSDLs and XSDs.
- **Entity Framework:** Entity Framework is the standard .NET object-relational mapping tool. It can be used to map C# classes to a relational database such as the Microsoft SQL Server.

- **Svcutil:** svcutil.exe was used to auto-generate a C# interface and associated classes complying with the web service contract.
- **Visual Studio:** Visual Studio and its associated tools are also used to install all the other tools except the SQL Server, manage the project's dependencies, and build the project.

Creating and Breaking the Web Service

Like the Java web service, the first step in .NET was to make a standard web service using the default tools and then test it against noncompliant SOAP messages. The following general process was used to create a web service in .NET from WSDLs and XSDs using the tools described above, without modifications:

1. A new WCF Service Application project was created in Visual Studio. For this type of project, Visual Studio automatically takes care of the dependencies that are needed.
2. Microsoft SQL Server was set up to accommodate data according to the IEC 61968-5 profile's *ExecuteDERGroups.wsdl*.
3. WCF's svcutil.exe was run to generate the C# interface and classes from the WSDLs and XSDs. C# provides two different serialization libraries that svcutil.exe can use. By default, svcutil.exe uses the newer but more limited *DataContractSerializer* class. However, the XSDs used complexType components not supported by *DataContractSerializer*, leaving no other option than to go back to using the older but more complete *XmlSerializer* class. *XmlSerializer* does little validation of its own, which may actually be more convenient for improving robustness because this allows for more customization.
4. Entity Framework's ADO.NET Entity Data Model tool was used to generate C# classes from the database tables.
5. Code was developed to implement the operations defined by *ExecuteDERGroups.wsdl*.

The resulting web service would create DER groups as specified by the WSDL if it received XML messages compliant with the XSDs. The *XmlSerializer* does not validate incoming XML messages against the XSDs, so it will throw an error only if it is unable to cast the XML into the generated C# class's type. In other words, the *XmlSerializer* does not immediately notice if a required field is missing because it can simply fail to populate the class's property. However, *XmlSerializer* may throw a null reference error if the XML has additional fields that do not correspond to a property. This is quite the opposite of the desired behavior.

Enhancing the Web Service

Ideally, the .NET web service should be able to gracefully handle unexpected fields and continue processing the rest of the data, but it should throw errors when required fields are missing. The ideal solution should avoid changing auto-generated code as much as possible. However, WCF automatically performs all the calls to *XmlSerializer*, so it is difficult to intercept the default behavior of the class.

Instead, a third option was chosen: custom parsing and modification of the XML message after the server receives the message but before deserialization. Although this method necessitates a more substantial change to the service than modifying the error handler class would, it appears to be a better fit for the .NET Framework's programming model.

WCF provides extensible classes that act at different points during the process of retrieving request messages. When *IDispatchMessageInspector* is implemented and added to the service, it allows the programmer to read and change incoming messages before they interact with any auto-generated code. Outgoing messages created by the auto-generated code can be similarly intercepted. If the XML is parsed to find and remove unexpected elements at this stage, the *XmlSerializer* class will never encounter elements it cannot place into the auto-generated code. *IDispatchMessageInspector* can then be used to change the outgoing message to warn the client about removal of elements.

With this XML parsing infrastructure in place, the service was able to trim XML messages to just their expected elements so that the *XmlSerializer* would not break, and relevant information was successfully passed to the database in the Microsoft SQL Server even when request messages contained unexpected elements. Furthermore, this method allowed for the return of helpful warnings and errors in the reply message. Implementation details are given in Appendix B.

6
FUTURE RESEARCH OPPORTUNITIES

Future research opportunities for this project include guidance on Representation State Transfer (REST) web services given either an XML or JSON payload. In addition, EPRI is looking at providing the full source code for both the Java and .NET implementations in an open source repository such as GitLab.

In addition to the technical challenges involved at coding time, there is the challenge of disseminating this approach across the ecosystem. If a utility creates all of the data consumers for the applications it owns, the implementation is very straightforward. However, that is hardly ever the case. Usually the web service ecosystem is heterogeneous with different technologies, vendors, and companies consuming data from a single source (among other things). How does a utility provide guidance or otherwise influence the implementation approach of each of its data consumers so that the entire ecosystem of services becomes robust and tolerant to change?

EPRI seeks guidance for the priorities of members in this area and for collaboration opportunities in demonstrating this approach.

A
JAVA TEST CASES

Test Case 1: Sending System Requests Creation of DER Group with DERNameplate Information Provided (Request Fails)

In this test case, all dependencies are used as is. None of the code automatically generated by the previously mentioned web service dependencies is modified past its default functionality. The result is a web service that will validate incoming messages against the schema files upon which Java classes are generated.

Figure A-1 shows a snippet from *DERGroups.xsd* showing expected elements within the element **DERFunction**. If anything not represented in Figure A-1 is listed under **DERFunction**, the receiving system will send an error message.

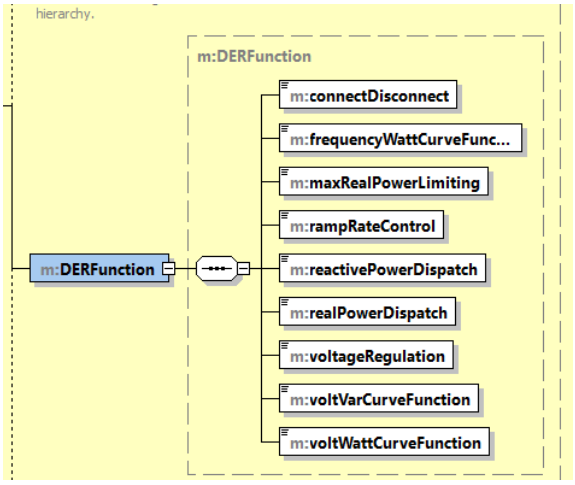


Figure A-1
The DERFunction element of DERGroups.xsd used by the receiving system in this web service

SoapUI is used as a client for this web service to send an example message containing the extra **DERNameplate** element that is not represented in the above schema. During this research, another SOAP client built using Python was used to similar effect. To illustrate the different test cases, however, only the SoapUI client will be shown in this report.

Figure A-2 uses SoapUI to show what happens when a message is sent with the unexpected element.

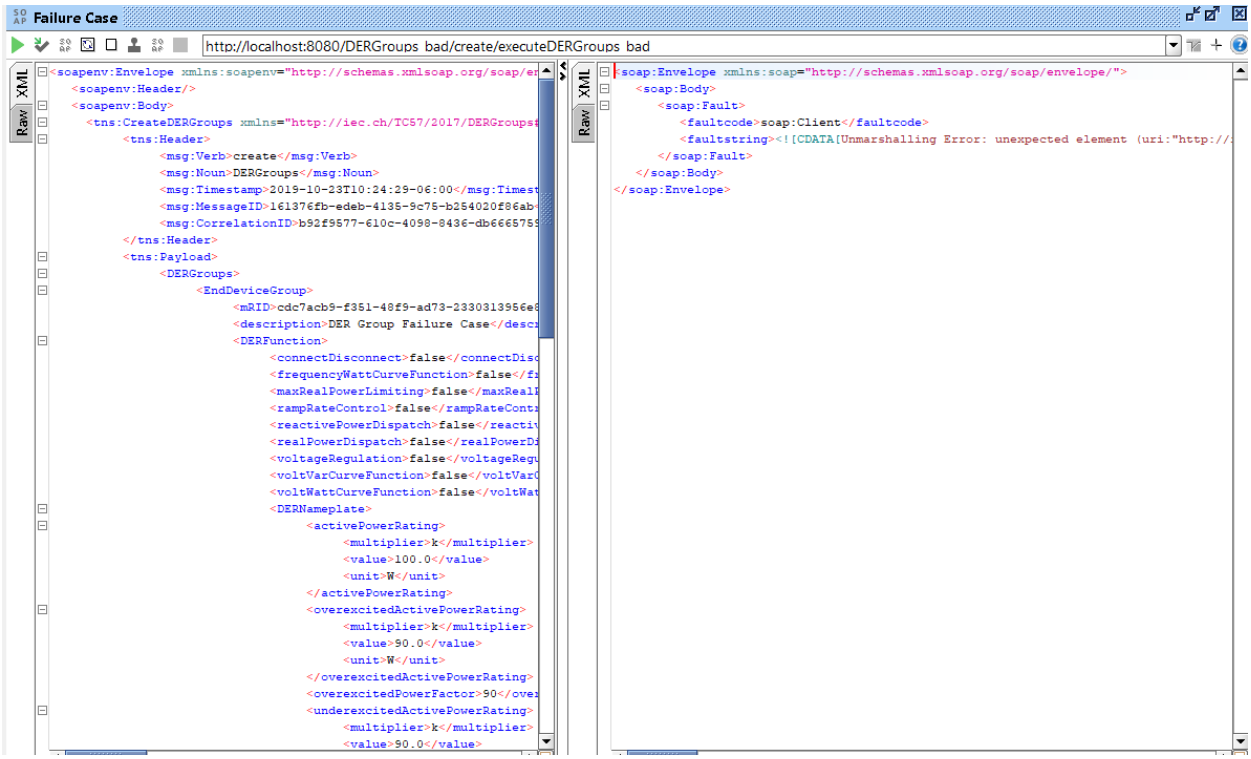


Figure A-2
The extra element causes an error, resulting in no payload processing

Once this error occurs, the web service replies to the request with an error message from the unmarshalling process and stops. The unmarshalling process (mapping the XML to elements represented in Java code) is performed behind-the-scenes before web service method implementation code is reached, resulting in the payload of the message never being processed.

Test Case 2: Sending System Requests Creation of a DER Group with DERNameplate Information Provided (Request Succeeds – Unsupported DERNameplate Element Ignored)

In this test case, a sending system sends a message with an additional element not supported by the receiving system. The system is expected to intercept the error and do something with it (for example, notify logs, move the error message to a specific **Error** element, and so on) without letting that error interfere with processing the rest of the message payload. To accomplish this with the libraries used to create the initial web service, one of the unmarshalling Java classes—*DetailValidationEventHandler*—is modified (see Figure A-3 for example code) so that it logs the error but continues with the unmarshalling process.

```
public class DetailValidationHandler extends DefaultValidationEventHandler {
	private static org.apache.log4j.Logger log = Logger
			.getLogger(DetailValidationHandler.class);
	public static String errorMsg;

	@Override
	public boolean handleEvent(ValidationEvent event) {
		log.debug("handleEvent called");

		if (event.getSeverity() == ValidationEvent.WARNING) {
			return super.handleEvent(event);
		} else {
			log.debug("message => " + event.getMessage());
			errorMsg = event.getMessage();
			log.debug("line number => " +
event.getLocator().getLineNumber());
			return true;
		}
	}
}
```

Figure A-3
Example code for modifying the DetailValidationEventHandler class

Figure A-4 shows an example of how this error could be logged to the console without bringing the rest of the service to a halt.

```
11106036 [http-nio-8080-exec-18] DEBUG com.epri.interceptor.DetailValidationHandler  - handleEvent called
11106041 [http-nio-8080-exec-18] DEBUG com.epri.interceptor.DetailValidationHandler  - message => cvc-complex-type.2.4.d: Invalid content was f
ound starting with element 'DERNameplate'. No child element is expected at this point.
11106042 [http-nio-8080-exec-18] DEBUG com.epri.interceptor.DetailValidationHandler  - line number => 27
11106070 [http-nio-8080-exec-18] DEBUG com.epri.interceptor.DetailValidationHandler  - handleEvent called
11106071 [http-nio-8080-exec-18] DEBUG com.epri.interceptor.DetailValidationHandler  - message => unexpected element (uri:"http://iec.ch/TC57/2
017/DERGroups#", local:"DERNameplate"). Expected elements are <{http://iec.ch/TC57/2017/DERGroups#}reactivePowerDispatch>,<{http://iec.ch/TC57/
2017/DERGroups#}maxRealPowerLimiting>,<{http://iec.ch/TC57/2017/DERGroups#}voltVarCurveFunction>,<{http://iec.ch/TC57/2017/DERGroups#}voltageRe
gulation>,<{http://iec.ch/TC57/2017/DERGroups#}connectDisconnect>,<{http://iec.ch/TC57/2017/DERGroups#}rampRateControl>,<{http://iec.ch/TC57/20
17/DERGroups#}realPowerDispatch>,<{http://iec.ch/TC57/2017/DERGroups#}voltWattCurveFunction>,<{http://iec.ch/TC57/2017/DERGroups#}frequencyWatt
CurveFunction>
```

Figure A-4
Error log when web service encounters unexpected element in a message

The result of this modification is a service that logs the error resulting from elements it does not expect while continuing to move on and process the rest of the well-formed payload (Figure A-5). Checking the database confirms that the requested DER Group was added by the receiving system (Figure A-6 and Figure A-7).

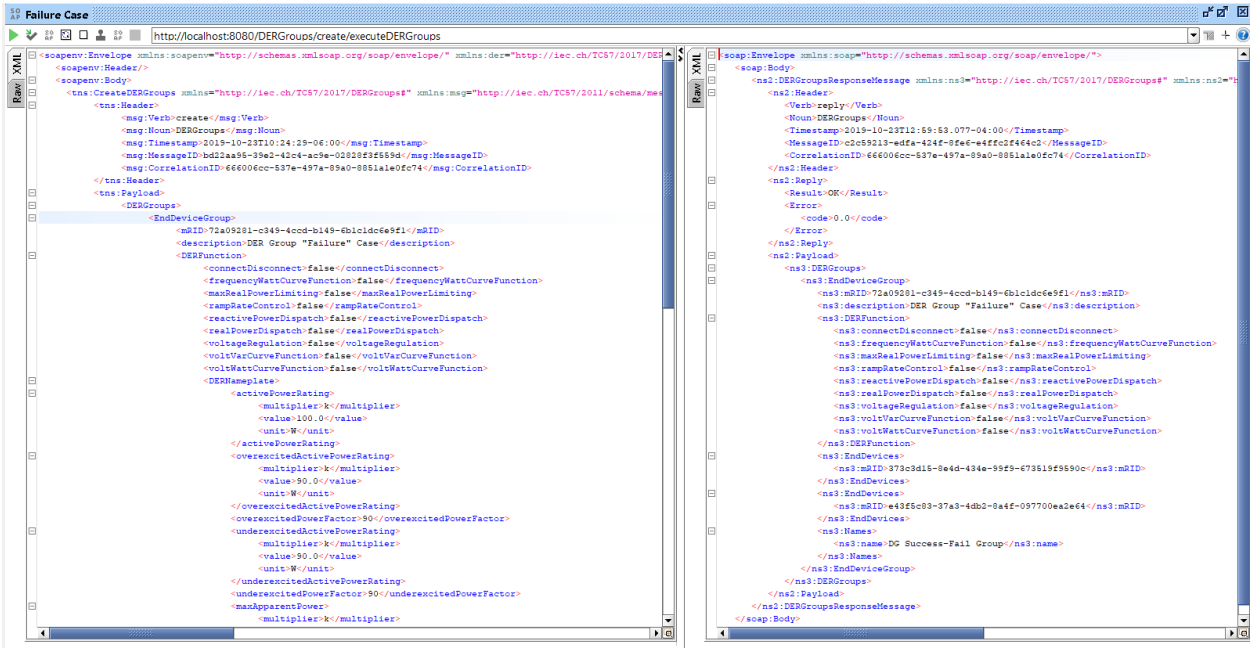
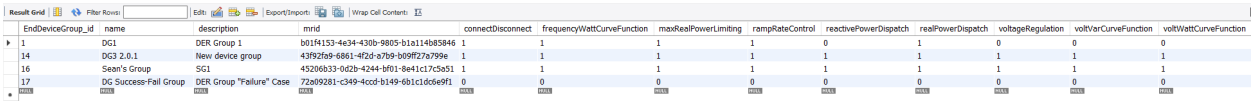
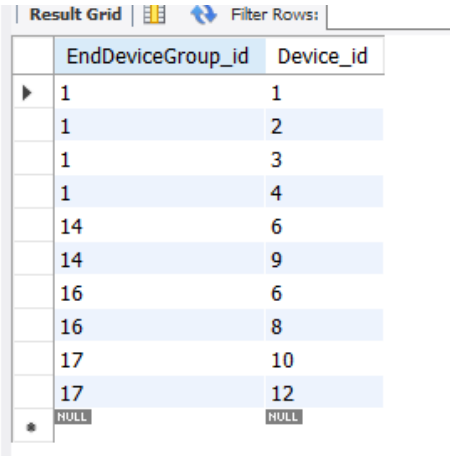


Figure A-5
Receiving system ignores additional elements, processing the rest of the payload



EndDeviceGroup_id	name	description	mrid	connectDisconnect	frequencyWattCurveFunction	maxRealPowerLimiting	rampRateControl	reactivePowerDispatch	realPowerDispatch	voltageRegulation	voltVarCurveFunction	voltWattCurveFunction
1	DG1	DER Group 1	b01f4153-4e34-430b-9805-b1a114b85846	1	1	1	1	0	1	0	0	0
14	DG3 2.0.1	New device group	43f92fa9-6861-4f2d-a7b9-b09ff27a799e	1	1	1	1	1	1	1	1	1
16	Sean's Group	SG1	45206b33-0d2b-4244-bf01-8e41c17c5a51	1	1	1	1	1	1	1	1	1
17	DG Success-Fail Group	DER Group "Failure" Case	72a09281-c349-4ccd-b149-6b1c1dc6e9f1	0	0	0	0	0	0	0	0	0

Figure A-6
EndDeviceGroup table shows the newly created DER Group



Result Grid | Filter Rows:

EndDeviceGroup_id	Device_id
1	1
1	2
1	3
1	4
14	6
14	9
16	6
16	8
17	10
17	12
NULL	NULL

Figure A-7
Join table correctly shows the two devices that were added to the new DER Group

Although this test case was implemented in Java using a specific set of libraries, this research does not need to be restricted to Java only. C#, for example, has a similarly named *ValidationEventHandler* class within its *System.Xml* namespace that could also be overridden if desired. Extra care should be taken when modifying these classes from their original implementation, however, because the modifications could result in unexpected behavior.

Note that for standards such as IEC 61968-100, schema validation errors should be placed into an **Error** element defined by *Message.xsd*, as shown in Figure A-8.

```
<tns:Reply xmlns="http://iec.ch/TC57/2011/schema/message">
    <Result>FAILED</Result>
    <Error>
      <code>2.4</code>
      <level>FATAL</level>
      <details>Invalid Meter(s)</details>
      <ID kind="name" objectType="Meter">X001</ID>
      <relatedID kind="name" objectType="UsagePoint">X001</relatedID>
    </Error>
    <Error>
      <code>2.12</code>
      <level>FATAL</level>
      <details>Invalid UsagePoint(s)</details>
      <ID kind="name" objectType="UsagePoint">XP001</ID>
      <relatedID kind="name" objectType="Meter">X001</relatedID>
    </Error>
 </tns:Reply>
```

Figure A-8
Example error message from Test Case MDS-4b.[6] Note that multiple errors are reported in a single message.

[6] https://www.epri.com/#/pages/product/000000003002014618/?lang=en-US

B
.NET TEST CASES

The WCF framework has different "dispatcher" classes for retrieving messages from the channel. A class that implements the *IDispatchMessageInspector* interface was extended so that messages could be parsed and modified independently of the auto-generated code. Because the *DispatchMessageInspector* class is not present in WCF services by default, there is no danger of accidentally altering other code behaviors when extending this class—though care must be taken not to modify the XML in unintended ways.

The *DispatchMessageInspector* class has two functions: AfterReceiveRequest and BeforeSendReply. AfterReceiveRequest is called once the SOAP request message has been converted to a C# *Message* class object with the XML message as its body. This object can be used to modify the request message and return an object containing any state information that will be needed when constructing a reply, such as errors or warnings. BeforeSendReply is called after the web service has generated a *Message* representing its SOAP reply message. This is used to modify the reply message based on the state information received from AfterReceiveRequest.

Code is then added to AfterReceiveRequest that uses *XmlReader* to create a list of expected elements defined in the XSDs and uses *XmlDocument* to check the XML message's elements against that list. Any elements in the message that do not match an element in the XSD are flagged and removed. Next, *XmlDocument*'s built-in validation function is used to detect any problems in the trimmed XML that would require an error to be thrown and payload processing to come to a halt.

On the BeforeSendReply side, the correlation state object is checked for errors or a nonempty list of removed elements and the reply message is modified to contain the appropriate error or warning.

To instruct WCF to add this extension, a few extra modifications are required—but these are fairly boilerplate. The *DispatchMessageInspector* needs to be added to the *DispatchRuntime* by a class that implements *IEndpointBehavior*, with a code snippet that looks something like the following:

```
public void ApplyDispatchBehavior(ServiceEndpoint endpoint, EndpointDispatcher
endpointDispatcher)
        {
            // add MessageInspector to endpoint
            MessageInspector customInspector = new MessageInspector();
            endpointDispatcher.DispatchRuntime.MessageInspectors.Add(customInspector);
        }
```

In turn, this class needs to be created by a class that inherits from *BehaviorExtensionElement*, like this:

```
public class CustomBehaviorExtensionElement : BehaviorExtensionElement
    {
        protected override object CreateBehavior()
        {
            return new MyEndpointBehavior();
        }

        public override Type BehaviorType
        {
            get
            {
                return typeof(MyEndpointBehavior);
            }
        }

    }
```

Finally, the *BehaviorExtensionElement* class is added to the WCF config file, shown in Figure B-1. WCF will be able to locate and call the message inspector at runtime.

```
<extensions>

   <behaviorExtensions>
     <add name="messageHandler" type="Service.CustomBehaviorExtensionElement, Service" />
   </behaviorExtensions>
</extensions>
<behaviors>
   <endpointBehaviors>
     <behavior name="messageHandler">
        <messageHandler />
     </behavior>
   </endpointBehaviors>
</behaviors>
<services>
   <service name="Service.ExecuteDERGroups">
      <endpoint address="" behaviorConfiguration="messageHandler"
                binding="basicHttpBinding" bindingConfiguration="DERGroups_Binding"
                contract="DERGroups_Port" />
   </service>
</services>
```

Figure B-1
Example snippet of Web.config code for .NET's WCF to insert custom behavior such as a DispatchMessageInspector

Because of the way the auto-generated *EndDeviceGroups* class is structured, *XmlSerializer* can ignore unexpected elements under **DERFunction** such as **DERNameplate**, shown in Figure B-2. However, unexpected elements directly under **EndDeviceGroup** would cause a null reference exception to be thrown. One approach to dealing with this while continuing to process the rest of the otherwise valid message is to collect the improper elements, strip them from the message, and pass through a modified and well-formed message to the web service, shown in Figure B-3.

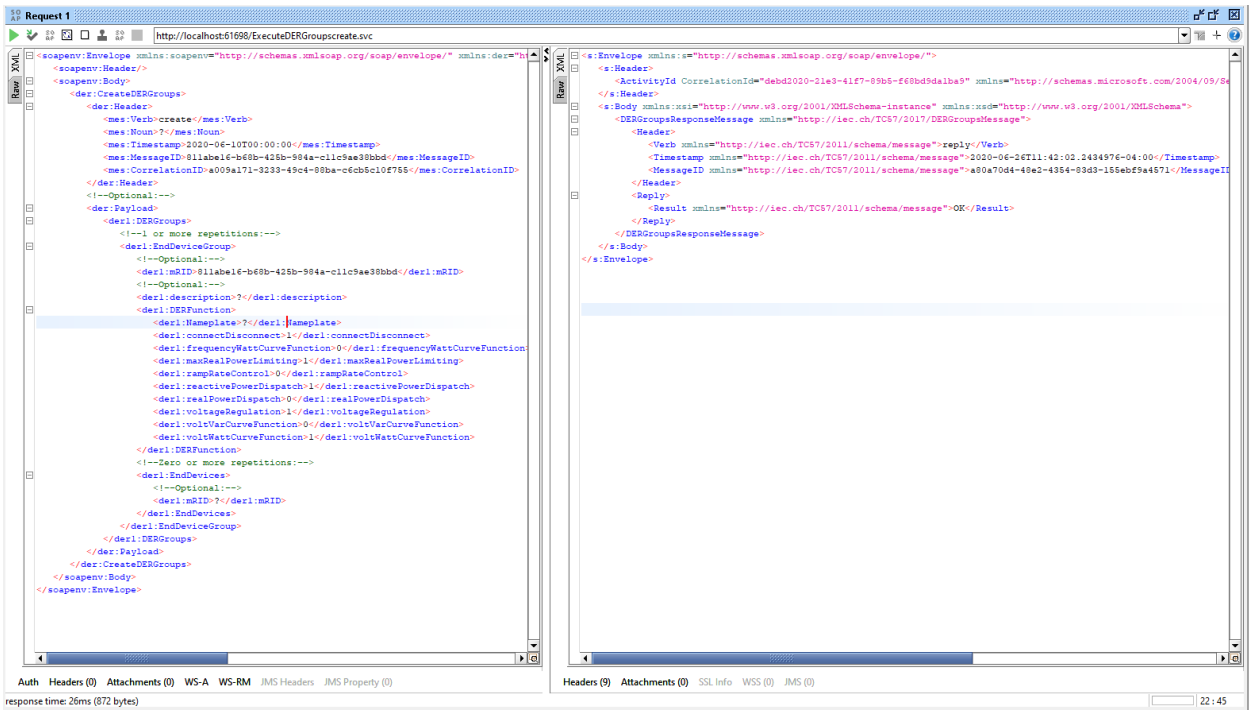


Figure B-2
Default .NET web service with XmlSerializer accepts additional elements under DERFunction but does not give a descriptive warning

```
public object AfterReceiveRequest(ref Message request, IClientChannel channel,
InstanceContext instanceContext)
{
   // Message -> XmlDocument
   (XmlDocument xmlDoc, MemoryStream memoryStream) = GetMessageFromStream(ref request);

   // create XML with invalid elements removed and save to MemoryStream
   (XmlDocument validXml, List<string> ItemsRemoved, string ErrorMessage) =
       XPathValidation.ValidateXml(xmlDoc);
   memoryStream.SetLength(0);
   validXml.Save(memoryStream);

   // XmlDocument in MemoryStream -> Message
   Message modifiedRequest = WriteMessageFromStream(ref request, memoryStream);

   request = modifiedRequest;
   // returns (List<string>, string) containing warning / error data
   return (ItemsRemoved, ErrorMessage);
}
```

Figure B-3
Example code for modifying a SOAP Message object

GetMessageFromStream, WriteMessageFromStream, and ValidateXml are all helper functions whose implementations will be shown in Figure B-4 through Figure B-6. Other objects are from System.Xml or System.IO. The function BeforeSendReply has a similar implementation but should modify the reply message to include warnings and errors instead of validating and removing elements.

```
public (XmlDocument, MemoryStream) GetMessageFromStream(ref Message message)
{
   // write the message to XmlDictionaryWriter using a MemoryStream
   System.IO.MemoryStream memoryStream = new System.IO.MemoryStream();
   XmlDictionaryWriter xmlDictionaryWriter =
       XmlDictionaryWriter.CreateTextWriter(memoryStream);
   message.WriteBodyContents(xmlDictionaryWriter);
   xmlDictionaryWriter.Flush();

   // use XmlDictionaryWriter to write message to XmlDocument for ease of manipulation
   // later
   memoryStream.Position = 0L;
   XmlDocument xmlDoc = new XmlDocument();
   xmlDoc.Load(memoryStream);

   return (xmlDoc, memoryStream);
 }
```

Figure B-4
Example code for writing a Message to an XmlDocument using a MemoryStream and XmlDictionaryWriter

```
public Message WriteMessageFromStream(ref Message message, MemoryStream memoryStream)
{
   // use XmlReader to read XML out of MemoryStream into message
   memoryStream.Position = 0L;
   XmlReader xmlReader = XmlReader.Create(memoryStream);
   Message modifiedMessage = Message.CreateMessage(message.Headers.MessageVersion,
               message.Headers.Action, xmlReader);
   // copy properties and headers of original message; we only want to change the body
   modifiedMessage.Properties.CopyProperties(message.Properties);
   // clear headers before copying them to stay compliant
   modifiedMessage.Headers.Clear();
   modifiedMessage.Headers.CopyHeadersFrom(message.Headers);
   return modifiedMessage;
 }
```

Figure B-5
Example code showing how to read an XmlDocument to a Message from a filled MemoryStream

Note that we reuse the SOAP properties from the original request Message in constructing the new one.

```
public static (XmlDocument, List<string>, string) ValidateXml(XmlDocument xmlDoc)
   {
      ErrorMessage = null;

      // add schemas to validate XML against
      System.IO.MemoryStream writer = new System.IO.MemoryStream();
      xmlDoc.Schemas.Add("http://iec.ch/TC57/2017/DERGroupsMessage",
                         xsdPath+"YourXsd.xsd");
      // add validation behavior
      ValidationEventHandler eventHandler =
                         new ValidationEventHandler(HandleValidationEvent);

      // remove unexpected elements with helper function
      List<string> recordRemoved = RemoveInvalidElements(xmlDoc);

      // validate against schemas
      xmlDoc.Validate(eventHandler);

      return (xmlDoc, recordRemoved, ErrorMessage);
   }
```

Figure B-6
Example code using a helper function to remove invalid elements from an XML and then validating it against the schemas

We will show an example implementation of the helper function RemoveInvalidElements (see Figure B-7 and Figure B-8). You will also want to create a callback function such as HandleValidationEvent to pass to the *ValidationEventHandler*. Ours simply writes the text of any validation error messages to a class string ErrorMessage.

```
static List<string> RemoveInvalidElements(XmlDocument xmlDoc)
   {
      // populate list of all names present in relevant XSDs
      List<string> validNodes = new List<string>();
      Dictionary<string, string> validPairs = new Dictionary<string, string>();
      XmlReader readMessage = XmlReader.Create(xsdPath+"YourXsd.xsd");
      GetNamesFromSchema(readMessage, validNodes, validPairs);

      XmlReader reader = new XmlNodeReader(xmlDoc);
      List<string> InvalidNodes = new List<string>();

      // iterate over XML nodes and check names against list of XSD names
      while (reader.Read())
      {
         if (reader.NodeType == XmlNodeType.Element)
         {
            string name = reader.LocalName;
            string prefix = reader.Prefix;

            // record all names present in XML but not XSDs in list
            // we'll remove these outside the while loop, or else the reader will be
            // interrupted
            if (!validNodes.Contains(reader.LocalName))
            {
               InvalidNodes.Add(reader.LocalName);
            }

          }
      }
      // remove invalid nodes
      foreach (string invalidNode in InvalidNodes) {
         XmlNodeList nodes = xmlDoc.SelectNodes("//*[local-name() = '" + invalidNode +
             "']");
         List<XmlNode> toRemove = new List<XmlNode>();
         // make a reference list so we can safely remove nodes
         foreach (XmlNode node in nodes)
         {
            toRemove.Add(node);
         }
         foreach (XmlNode node in toRemove)
         {
            node.ParentNode.RemoveChild(node);
         }
      }
      return InvalidNodes;
   }
```

Figure B-7
Example implementation of function to parse XSDs and XML and then remove any unexpected elements from the XML

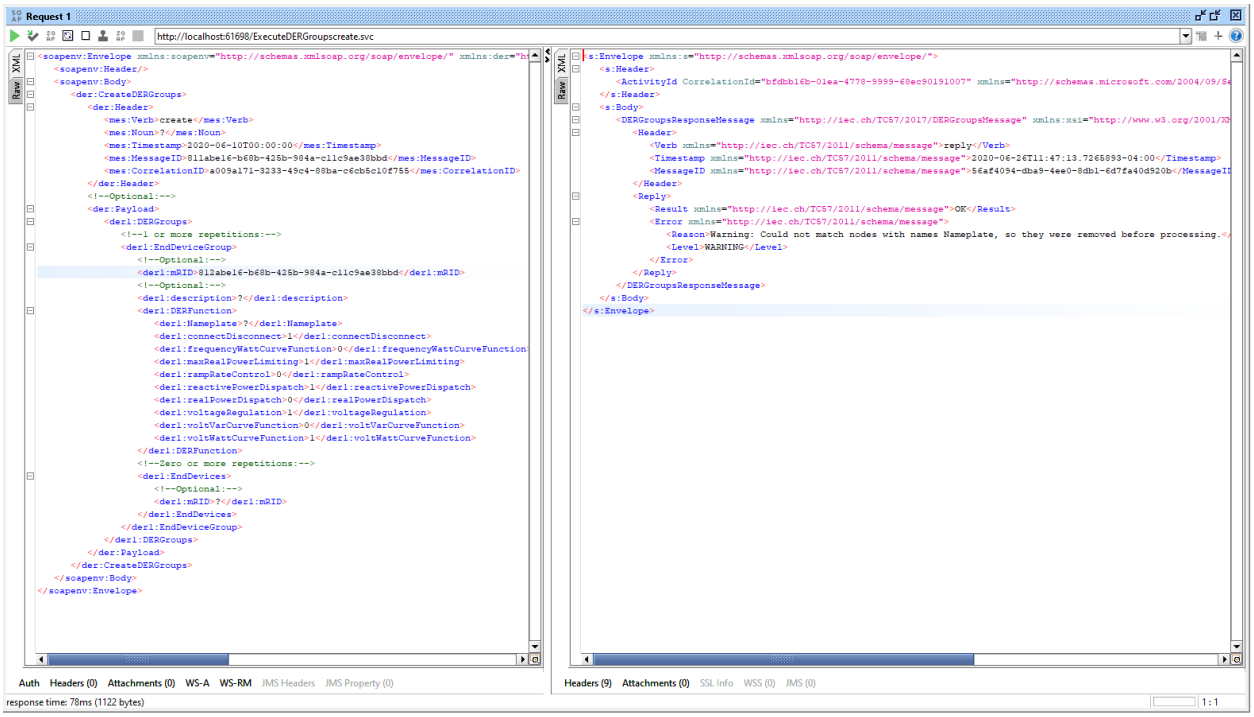


Figure B-8
Example warning message returned after successful handling of unexpected elements in .NET

The Electric Power Research Institute, Inc. (EPRI, www.epri.com) conducts research and development relating to the generation, delivery and use of electricity for the benefit of the public. An independent, nonprofit organization, EPRI brings together its scientists and engineers as well as experts from academia and industry to help address challenges in electricity, including reliability, efficiency, affordability, health, safety and the environment. EPRI also provides technology, policy and economic analyses to drive long-range research and development planning, and supports research in emerging technologies. EPRI members represent 90% of the electricity generated and delivered in the United States with international participation extending to nearly 40 countries. EPRI's principal offices and laboratories are located in Palo Alto, Calif.; Charlotte, N.C.; Knoxville, Tenn.; Dallas, Texas; Lenox, Mass.; and Washington, D.C.

Together…Shaping the Future of Electricity

3002018641

Electric Power Research Institute
3420 Hillview Avenue, Palo Alto, California 94304-1338 • PO Box 10412, Palo Alto, California 94303-0813 • USA
800.313.3774 • 650.855.2121 • askepri@epri.com • www.epri.com