University of Szeged
Faculty of Science and Informatics
Department of Software Engineering

Validating Documents of Web-based Metalanguages Using Semantic Rules

Ph.D. Dissertation

Candidate
Miklós Kálmán

Supervisor
Prof. Tibor Gyimóthy

Submitted to the
Doctoral School of Computer Science

Szeged, 2014

To my family and all who have helped me on my path.

Foreword

It has been a long and challenging journey: I have finally written my PhD dissertation. I started my PhD studies back in 2003. After finishing the three year PhD program my publication activities halted for several years. I started working, which consumed most of my time and didn't have enough energy left after a long day's work.

After several years of absence from the research community I decided to pick up where I left off and pursue my PhD again. I started searching for an updated research area and that led me to validation. I would like to thank Prof. Tibor Gyimóthy and Ferenc Havasi for their collaboration, guidance and help. Furthermore I am extremely grateful for the extended support of my brother and parents along with my good friends Péter Rosztóczy, Ottó Vas and Gergely Molnár for believing in me and persistently guiding me through the tough times and never giving up on me. I would also like to thank László Szabó for his help and coaching as well as Paulina Nagy for her help in proofreading.

I hope my dissertation will inspire others to hang on and finish what they started, since at the end it does provide a great feeling of accomplishment.

Miklós Kálmán, May 2014.

List of Tables

List of Figures

Contents

Validation overview

Validation has become a hotly debated topic over the years. Without validation, large systems can crumble at the hands of malicious data, whether they were intentionally introduced into the system or not. Validation is present in many systems as part of the business logic. When dealing with validation we have to distinguish between structural and content-based approaches. The structural side is responsible for ensuring that the document layout complies with the requirements. For XML documents structural validation entails the process of making them well-formed and ensuring that they only contain elements allowed by the domain schema. Content validation is more challenging since the document's whole context has to be considered and the values themselves validated.

My research started with the compaction of XML documents using semantic rules, which yielded the conception of the SRML language. This language and the concept of semantic rules later led me to the validation space. This dissertation is aimed solely at the validation area with references to the initial SRML 1.0 [36] and the SRML 1.1 [31] (XMI extension) versions of the language.

The dissertation will demonstrate how SRML was extended to the validation space along with validation approaches for XML (SRML 2.0 [35]), Web Forms (jSRML [33]), Google Protocol Buffers (ProtoML [32]) and Web Services (wsSRML [34]). The solutions outlined in the document will also demonstrate ways to correct invalid documents.

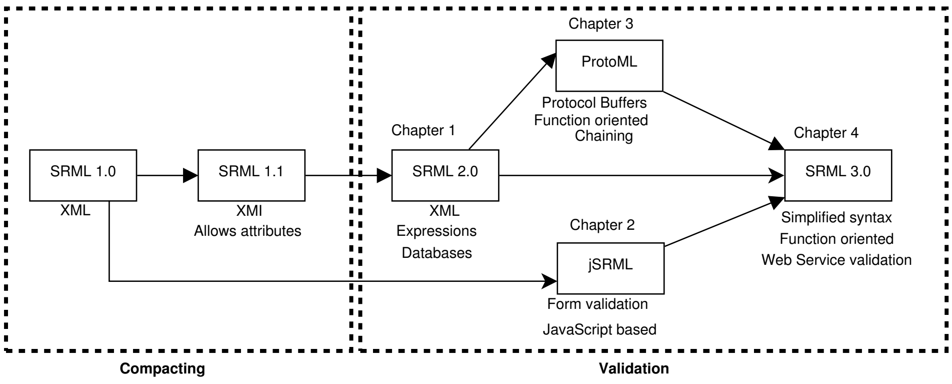


Figure 1: Evolution of SRML versions and their areas

The evolution of SRML can be seen in *Figure 1.* The dissertation is divided into four theses, which are summarized in *Table 1.* The document starts out by demonstrating how the SRML language was extended to the validation space based on [35] by introducing version 2.0 of the language (Chapter 1). The extension provides a way to validate both the structure and content of XML documents. With SRML 2.0 we can define context-sensitive value

Short Thesis Title	**Thesis**	**Publications**
Validating XML documents	Provide a way to validate and correct XML documents using semantic rules through the extension of SRML 1.0.	[35]
Validating Web Forms	Create a new jSRML metalanguage, which is capable of defining semantic rules for the validation and correction of web forms.	[33]
Validating Google Protocol Buffers	Introduce a new metalanguage (ProtoML), which can validate and correct the messages of Google Protocol Buffers.	[32]
Validating Web Services	Combine the previous metalanguages (SRML 2.0, jSRML, ProtoML) into SRML 3.0 and provide a way to validate Web Services.	[34]

Table 1: Theses of the dissertation

requirements for XML documents. This allows more fine-grained rule definitions. The system also provides a way to correct the contents of the document, which is a very useful feature for domains where data integrity is crucial. We also demonstrate how the language can be used in a database context using H2 and triggers.

Chapter 2 details how the jSRML [33] extension of SRML 1.0 can be used to validate web forms. The jSRML language and its engine can validate web forms in a non-obtrusive way. It also details the several modes of validation ranging from *real-time* to *server-side*. The chapter also demonstrates ways to learn jSRML validation rules using decision trees and other machine learning algorithms. This branch of SRML was created parallel to the SRML 2.0 specification and has a different syntax.

Following this the dissertation presents a way to validate Google Protocol Buffers using a new metalanguage ProtoML [32]. This language is also a parallel development to SRML, which paved the way for the final 3.0 version. It introduces concepts like chained functions and simplified rule definitions.

The final chapter of the document focuses on the validation of Web Services using wsSRML [34] and SRML 3.0. The validation engine can run in *native* or *proxy* modes. The latter one allows black-box systems to receive a validation aspect even when the source code is not available. There may be similarities in the language elements of ProtoML, SRML and jSRML since they were all aimed at validation and used SRML 1.0 as their starting points (ProtoML only used the concept of semantic rules). These language branches are combined into SRML 3.0, taking the knowledge and seasoned functionalities of all languages to create a compact yet descriptive validation language.

The views and findings expressed in this dissertation are based on my research and may sometimes feel subjective. However, I believe that the findings are strong enough to stand on their own. Throughout the dissertation I will also be using ”we” as self-reference to credit the co-authors of some of the articles used as the basis of this dissertation.

Chapter 1

Validating XML documents

Thesis: Provide a way to validate and correct XML documents using semantic rules through the extension of SRML 1.0.

Introduction

Data exchange has evolved considerably over the years. Distributed systems share vast amounts of information in a matter of seconds. The most commonly used format for-text based (non-binary) information exchange is XML [15]. There are many advantages to this format. However, it does have its shortcomings. One of these is that since it is text-based there is a possibility that the data it contains is not valid. The structure is completely free and there is no restriction on what elements (text nodes) the user can enter. To provide a structural description the XSD [59] schema was introduced. This schema allows the domain owners of the XML to define the structural requirements. It defines which elements the document can contain, and what the attribute types are.

Examining the exploits against sites and their databases, most of them target the weakest point of these systems: data integrity and validity. Lots of the sites use XML for SOAP [13] operations or data exchange and as such validation is a very important aspect. Most validators can read the XSD file and use it to validate the XML document. This will find most of the structural errors. However, it cannot describe more complex relationships between nodes that may be needed for validation. In an earlier article we introduced the SRML [36] language, which allowed semantic rules to be defined for attribute relationships. The metalanguage was primarily used to compact XML documents based on the rules. Version 1.1 [31] extended the compaction to XMI documents. This opened up a plethora of possibilities in terms of describing relationships between attributes. We decided to extend this language and create an extension to the XSD format that allows these types of rules to be used during XML validation. In the process of this extension support was added for element-based rules, thus simplifying the reference of nodes using the power of XPath [18]. In the earlier definition of the language, referencing nodes within the context yielded unnecessary complications as it was not possible to reference all nodes and attributes. One of the most pressing issues we were faced with was how to store the rules without obstructing the XSD validation itself. The solution used was to bundle the SRML rules into the `appinfo` section of the XSD, which is mostly used by JAXB [54] (Java XML Broker) for marshalling and unmarshalling meta information. We have extended the standard Java XSD validator using a Spring project. The validator first runs the normal XSD validation using its XSD file. This validator ensures that the XML is well-formed [60]. It then reads the `appinfo` and validates the XML using the embedded SRML rules. This way we get the best of both worlds. The normal XML validator will filter out the nodes/attributes which do not conform syntactically, ensure that the XML is well-formed, and perform a type-check on the document domain. After these steps the SRML rules will validate the actual content of the nodes. This way both structure and content validation becomes possible on XML documents.

Schematron [58] uses a similar approach to perform the validation by bundling the rules in the `appinfo` area. One of the biggest advantages our approach has over this leading validation engine is that ours allows for the data to be corrected besides just being validated. This can be very useful in environments where the validation rules can also define how to correct the input and data loss or corruption is not an option. This allows for the input to be validated, and if some items are not valid, however, have corresponding correction rules defined, the data can be sanitized and corrected, thus allowing the data to be transmitted instead of dropping the results due to an invalid input.

We took the idea a step forward by applying the SRML validator to a database context. As most RDBMS tables and records can be represented in XML, it made sense to provide a way for data validation using SRML. This approach enabled us to write the validator in a way that it can be used to validate records on insert/delete/update operations. The solution had its challenges, as we could not just apply the rules to the whole database, as that would warrant a massive memory requirement. The answer to the problem was to load parts of the records into DOM [2] trees depending on what the context of the CRUD operation was working on. This meant only parts of the records were transferred to memory and permitted the construction of a mini XML tree from the records. After the tree was built, the SRML rules could be applied on it just as if it was a standalone XML document.

The following sections will first provide some background information on the technologies as well as a brief introduction to our SRML metalanguage. We will then demonstrate the use of the new validator through an example. This example will be used in the database validation section as well to make it easier to follow.

1.1 Preliminaries

This section is dedicated to providing some color on the technologies and concepts used. We will introduce the XML format, along with the XSD schema definitions and the SRML language. These concepts are essential to understand the later sections of this chapter.

1.1.1 XML

The first concept that must be introduced is the XML format. A more thorough description of the XML documents can be found in [15] and [25]. XML documents are very similar to *HTML* files, as they are both text-based. The components in both are called *elements*, which may contain further *elements* and/or text, or they may be left empty. *Elements* may have attributes like the *a* attribute of *href* elements. *Figure 1.1* demonstrates an example for storing a numeric expression in XML format. This example has an additional attribute called *"type"*, which stores the type of the expression. The values can be *int* or *real*.

```
<expr> <multexpr op="mul" type="real">
 <expr type="int"><num type="int">3</num></expr>
  <expr type="real">
   <addexpr op="add" type="real">
    <expr type="real"><num type="real">2.5</num></expr>
    <expr type="int"><num type="int">4</num></expr>
   </addexpr>
  </expr>
</multexpr> </expr>
```

Figure 1.1: A possible XML form of the expression $3 * (2.5 + 4)$.

DTD and XSD

It is possible to define the syntactic structure of XML documents using DTD [49] (Document Type Definition) files and XSD [59] (XML Schema Definition) schemas. DTD files can only provide the basic structure of XML files (limited to elements and attributes). Taking an XML file containing a numeric expression of *Figure 1.1* we can define the DTD schema in *Figure 1.2*.

XSD is a newer format and can do everything a DTD can, along with additional restriction definitions. The second advantage XSD schemas have over DTDs is that they are also XML based, meaning they are easier to parse and display in a hierarchic manner. XSD documents describe the elements and their attributes just like the DTD. However, they also specify the type

```
<!ELEMENT num (#PCDATA) >
  <!ATTLIST num type ( real | int )#REQUIRED >
<!ELEMENT expr ( num | multexpr | addexpr ) >
  <!ATTLIST expr type( real | int ) #IMPLIED >
<!ELEMENT multexpr ( expr , expr ) >
  <!ATTLIST multexpr op ( mul |div ) #REQUIRED type ( real | int ) #IMPLIED >
<!ELEMENT addexpr ( expr , expr ) >
  <!ATTLIST addexpr op ( add |sub ) #REQUIRED type ( real | int ) #IMPLIED >
```

Figure 1.2: The DTD of the simple expression in *Figure 1.1*

of content that the elements can have, detail the order in which elements can appear or provide a choice of elements for a given context (*Figure 1.3*). The XSD schema can define the format of the nodes or attributes using regular expressions (e.g.: ISBN numbers or an IP address). We will detail the XSD format in more detail when we present how we extend its functionality.

Parsing XML documents

To analyze and validate XML files they must be parsed. There are two ways of parsing XML files: one is based on the DOM [2] (Document Object Model) tree, while the other is a sequential parser called SAX [44].

A DOM tree is a tree containing all the tags and attributes of an XML document as leaves and nodes (*Figure 1.4* is the DOM tree of *Figure 1.1*). This DOM tree is used by the XML processing library for internal data representation. The DOM model is a platform- and language-independent interface that allows the dynamic accessing and updating of the content and structure of XML documents. When DOM tree parsing is used, it makes the XML document handling easier, but it requires more memory to accomplish this, since it creates a tree of the XML in the memory. This method is quite effective on smaller XML documents.

The SAX parser can handle large input XML files, but since it is a file-based parser it can be quite slow, especially when trying to access attributes that are not in the current *element*. The memory requirements of this method are constant and not in direct proportion to the size of the input XML document.

1.1.2 XPath

Before detailing how the validation works, one more technology has to be noted: XPath [18] (XML Path language). The XPath language is based on the DOM (tree) representation of the XML document. It provides an easy way to query for nodes and attributes using expressions. It is widely used in CSS and HTML selectors as well. Our validation engine leverages this language heavily as it allows us to extend SRML to make element and attribute reference much easier.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="expr">
      <xsd:complexType>
         <xsd:choice>
            <xsd:element name="multexpr" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="addexpr" minOccurs="0" maxOccurs="unbounded" />
         </xsd:choice>
           <xsd:attribute name="type" use="optional">
              <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="int" />
                  <xsd:enumeration value="real" />
               </xsd:restriction>
            </xsd:simpleType>
         </xsd:attribute>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="multexpr">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="expr" minOccurs="2" maxOccurs="2"/>
         </xsd:sequence>
         <xsd:attribute name="type" use="optional">
              <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="int" />
                  <xsd:enumeration value="real" />
               </xsd:restriction>
            </xsd:simpleType>
         </xsd:attribute>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="addexpr">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="expr" minOccurs="2" maxOccurs="2"/>
         </xsd:sequence>
         <xsd:attribute name="type" use="optional">
              <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="int" />
                  <xsd:enumeration value="real" />
               </xsd:restriction>
            </xsd:simpleType>
         </xsd:attribute>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 1.3: The XSD of the simple expression in *Figure 1.1*

The most important kind of expression in XPath is the location path. Each path is comprised of a sequence of location steps. A *step* element has three components: an axis, a node test and zero or more predicates. The expression path is evaluated from left to right. The axis specifier describes the context of the navigation element (e.g.: child).

A node test will return all nodes in the document matching the path. Predicates allow further filtering of the results. To better demonstrate how XPath

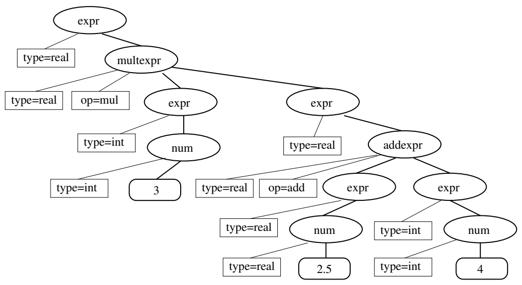


Figure 1.4: XML document DOM tree

can be employed consider the example in *Figure 1.5*. Normally the author attribute would be an element, but we wanted to show attribute references as well to allow better understanding of the XPath topic.

```
<books>
   <book author="J.R.R. Tolkien">
     <title>Lord of the Rings</title>
   </book>
   <book author="J.R.R. Tolkien">
      <title>The Hobbit</title>
   </book>
   <book author="Jules Verne">
     <title>Around the world in 80 days</title>
   </book>
   <book>
     <title>Anonymous</title>
   </book>
</books>
```

Figure 1.5: A simple XML example for books

When using `//book/title` as the XPath query, all `title` nodes will be returned. Adding a `/text()` will only show the text content of those nodes. If we only want to query for example all books by *"Jules Verne"* we would add the *[@author="Jules Verne"]* predicate. If the predicate is supplied with a value then the expression will filter all nodes matching the given attribute. If only the attribute is specified then all nodes matching the expression where that attribute is defined will be returned. For example `//book[@author=‘‘Jules Verne’’]/title/text()` will return *"Around the World in 80 days"*.

It is also possible to query nodes that have the given attribute regardless of their value. For example `//book[@author]/title` will return all titles of books, which have the author attribute defined. XPath can also contain regular expressions and has a lot of in-built functions.

1.1.3 SRML 2.0

The first version of SRML (version 1.0) was described in [36]. We have extended this format with XPath support along with additional features. This chapter describes the new aspects of SRML 2.0 that were introduced to enable the validation of XML documents as well as database reference descriptions. The key differences between SRML 1.0 and 2.0 can be seen in *Table 1.1*. The first definition of SRML focused on compaction and the theoretical description of the rules. However, nowadays this significance was replaced by the importance of data validation and security.

For the validation area we decided to simplify and clean up the language to allow easier rule descriptions without sacrificing flexibility. The new format can be used for data correction as well. The full XSD of the new format can be found in *Appendix A.1*.

Property	**SRML 1.0**	**SRML 2.0**
Main Focus	Compaction/Decompaction	Validation/Correction
Rule reference level	Attributes	Element and Attributes
Potential Application Area	XML Documents	XML and Databases
Rules based on	Attribute Grammars	AG and XPath
Rule Definition	Complex	Simplified with XPath
Numeric Expression Rules	Much overhead	Simplified, inner expression engine
Rule dependencies and storage	DTD and separate SRML file	Encapsulated in the XSD

Table 1.1: Key differences between SRML 1.0 and 2.0

Figure 1.6 shows how the ***addexpr*** section of the XML in *Figure 1.1* can be described in SRML 2.0. The rule definition format covers the type attribute results as well. By default, DTDs and XSDs can not describe how the type attribute changes during a multiplication of an *int* and a *real*. With the help of SRML 2.0 we are able to describe the type change easily. Defining indexed child references is also easier, for example ***../expr[1]/@type*** refers to the first ***expr*** sibling's type attribute. The ../ is an extension to XPath allowing the upward navigation and reference.

The new version of SRML allows and aids the XML validation process, containing several enhancements from which the following should be noted:

XPath support: Using XPath it is now easier to reference attributes and elements in the XML context. Previously it was a tedious job to reference specific attribute instances.

Numeric expressions: The new format also allows numeric expressions to be used during the rule context, making it easier to describe expressions and use them in the rule definitions.

Element and attribute references: It is now possible to reference both attributes and elements. Previously SRML only operated on an attribute level.

```
<rules-for root="addexpr">
<rule-def name="@type">
 <rule-instance>
  <expr>
    <if-expr>
    <expr>
      <binary-op op="or">
        <expr>
        <binary-op op=equal>
          <expr><value-ref path="../expr[1]/@type" /></expr>
          <expr><data>real</data></expr>
        </binary-op>
        </expr>
        <expr>
        <binary-op op="equal">
          <expr>value-ref path="../expr[2]/@type" /></expr>
        <expr><data>real</data></expr>
        </binary-op>
        </expr>
      </binary-op>
    </expr>
    <expr><data>real</data></expr>
    <expr><data>int</data></expr>
    </if-expr>
  </expr>
  </rule-instance>
  </rule-def>
</rules-for>
```

Figure 1.6: An SRML 2.0 example for *type* attribute of the `addexpr` element

Multiple rules for the same context: With this new feature, multiple rules can be defined for the same context. This is important for validation, as it is possible for the document to be considered valid if *any* of the validation rules for that context is fulfilled.

Node relationship for tables: SRML 2.0 introduced the option to describe database tables and thus extend the scope of the rules to the database space as well.

1.2 Validating XML Documents

XML validation plays a very important role in the life of the document. In many cases it is vital to ensure that an XML document is both syntactically and semantically correct. As with many text-based formats, errors can arise from invalid documents. A document has to be both well-formed and valid to pass validation. The term well-formed [60] refers to the fact that all tags/elements have matching pairs, there are no overlapping elements nor do the elements/tags contain invalid characters. Once a document is well-formed, the contents can be validated. XSD allows several ways of defining which parts of the document have restrictions and what the document has to conform to.

We will use an example to demonstrate what an XSD would look like for a bookstore example. This example will then be used throughout the remaining sections to allow a better comparison.

Consider the following use-case: we have a bookstore that sells books using a shopping cart. Each item in the shopping cart is a `book`, which has an *author* attribute, a *title*, an *ISBN number*, a *price* and a *cover* attribute that can be either *digital* or *hardcover*. The item also contains the *quantity* of the books in the cart and the *subtotal* for the given entry as well a *discount*. This is a simplified example as normally one would define an item with a `book` reference and store the quantity on that level, however, to save space and avoid complexity in the example we merged these two elements into a single one. The *ISBN number* has to be a specific format and price can only be a number. The XML of the example can be seen in *Figure 1.7*.

```
<cart xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="cart.xsd" hasDiscount="false">
        <book cover="hardcover">
            <author>J.R.R. Tolkien</author>
            <title>Lord of the Rings</title>
            <isbn>1-12345-123-1</isbn>
            <qty>5</qty>
            <price>100</price>
            <discount>0</discount>
            <tax>25</tax>
            <total>625</total>
            <region>0</region>
        </book>
        <book cover="digital">
            <author>William Shakespeare</author>
            <title>Macbeth</title>
            <isbn>1-12-654321-1</isbn>
            <qty>1</qty>
            <price>100</price>
            <discount>10</discount>
            <tax>35</tax>
            <total>121.5</total>
            <region>1</region>
        </book>
 </cart>
```

Figure 1.7: XML of cart example

In order to ensure that all documents that get entered into our shopping cart system are valid we have to define an XSD schema for this domain. The XSD of the example can be found in *Figure 1.8*. The XSD schema defines what the structure of the `cart` XML files needs to look like. It needs to contain a root (`cart`) element. This element has an attribute called *hasDiscount* and contains `book` child items. The `book` element definition details the elements that a `book` can have, along with their types and requires an attribute called *cover*. This attribute can take on two values: *"digital"* and *"hardcover"*. The `book` has an `ISBN number` whose format is defined as a regular expression. This ensures that all text entered into the ISBN node will need to be in the same format. Once we have the XSD document we can run a document validator on it.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="cart">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="book" minOccurs="0" maxOccurs="unbounded" />
         </xsd:sequence>
           <xsd:attribute name="hasDiscount" type="xsd:boolean" use="optional"/>
      </xsd:complexType>
   </xsd:element>

   <xsd:simpleType name="ISBN-type">
      <xsd:restriction base="xsd:string">
       <xsd:pattern
        value="\d{1}-\d{5}-\d{3}-\d{1}|\d{1}-\d{3}-\d{5}-\d{1}|\d{1}-\d{2}-\d{6}-\d{1}" />
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:element name="book">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="author" type="xsd:string" />
            <xsd:element name="title" type="xsd:string" />
            <xsd:element name="isbn" type="ISBN-type" />
            <xsd:element name="qty" type="xsd:integer" />
            <xsd:element name="price" type="xsd:integer" />
            <xsd:element name="discount" type="xsd:integer" />
            <xsd:element name="tax" type="xsd:integer" />
            <xsd:element name="total" type="xsd:float" />
            <xsd:element name="region" type="xsd:integer" />
         </xsd:sequence>
         <xsd:attribute name="cover">
            <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="paperback" />
                  <xsd:enumeration value="hardcover" />
                  <xsd:enumeration value="digital" />
               </xsd:restriction>
            </xsd:simpleType>
         </xsd:attribute>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 1.8: XSD of cart example

Throughout the dissertation we used Java as the primary language, as it is platform-independent and has a powerful function set. In Java one of the ways of validating against an XSD is achieved by using the Java XML validation API. This validation will filter out invalid results and ensure that all elements are in their proper position and the types of the fields are correct. However, there is no way to describe more complex validation rules. Suppose there are additional rules that need to be satisfied in order for a `cart` to be valid. For example: the `tax` on digital books should always be 0 or if the number of items in the cart is more that two then the *hasDiscount* attribute has to be *true*. The current XSD format does not provide a way to describe or validate against these types of conditions. This is where the power of SRML 2.0 comes in. In the next section we will show how we can extend the XSD format to allow more complex validation rules.

1.2.1 Extending XSD

When trying to extend a format that is widely used, one has to be careful not to break legacy systems that are dependent on it. We had to figure out a way to stay compliant with the original XSD schema, but also allow the description and processing of SRML based validation rules. To overcome this challenge, we opted to use the `appinfo` meta section of the XSD document. This section is usually used for application-specific meta information storage. An example for this would be JAXB (*Figure 1.9*) marshalling meta overrides. JAXB [54] is Java's XML Broker, which is an API used to marshal classes to and from XML. This section seemed like a viable part of the document to insert the SRML rules. We will continue to use the bookstore cart example in the further sections to provide a consistent overview.

```
<xsd:annotation>
       <xsd:appinfo>
           <jaxb:globalBindings collectionType="java.util.Vector"/>
           <jaxb:schemaBindings>
              <jaxb:package name="com.flutebank.custompackage"/>
           </jaxb:schemaBindings>
       </xsd:appinfo>
</xsd:annotation>
```

Figure 1.9: Using appinfo for JAXB binding information.

1.2.2 A validation example using SRML

This section will demonstrate validation scenario of the shopping cart example.

Validation Requirement #1: The cart's *hasDiscount* attribute is true if there are more than 2 books in the cart.

Validation Requirement #2: All books by "J.R.R. Tolkien" should receive a 20% discount.

Validation Requirement #3: All digital books should be tax free.

Validation Requirement #4: The *total* entry of the book is calculated by multiplying the quantity, price and discount values.

The above validation conditions would not be possible with the standard XSD format. We will now demonstrate the rules that allow the description of the validation requirements. To embed rules into the XSD, first we have to define the `appinfo` element in the `annotation` the the following way:

```
<xsd:annotation>
<xsd:appinfo xmlns:srml=http://www.sed.inf.u-szeged.hu/SRMLSchema"
     srml:schemaLocation="srml.xsd">
```

Based on the SRML XSD the top-level definition element is the `srml-def` node. This element contains one or more `rules-for` elements. The `rules-for` elements define the context root of the rule:

```
<srml:srml-def>
<srml:rules-for root="cart" >
```

Each `rules-for` element has one or more `rule-def` entry. This specifies the target attribute or element that will be validated. For example the following will target the cart's *hasDiscount* attribute:

```
<srml:rule-def name="@hasDiscount" mode="correct" match="any" >
```

The mode attribute in the above example tells the validator what to do with the results. The possible values are *"validate"* and *"correct"*. The first mode will perform the validation based on the rule and report any failures it encounters. The second mode (*"correct"*) will perform the validation and if it fails it will then attempt to alter the value based on the expected value defined in the rule. This is an important feature, as it will ensure that the data is correct even when the validation fails. In many cases this mode can recover the XML document and make it valid again. In our example the *hasDiscount* attribute value is automatically corrected if its validation fails.

The *"match"* attribute informs the validation engine what to do with multiple `rule-def` elements. If the match attribute is marked as *"any"* that means that the given validation rule returns true if any of the `srml-instance` rules are matched. Each `rule-def` can contain one or more `rule-instance` elements. These elements can define different rules for the same context. This is useful when a node can be considered valid when any of the listed rules return successfully.

Every `rule-instance` has a `validation-error` and an `expr` element. The `validation-error` element is used to pass in the string that is used when the rules in the instance fail. This string is returned to the user as a validation error, which is more descriptive than throwing a validation exception. The `expr` tag contains the validation rule. *Figure 1.10* shows the validation procedure.

Taking the cart example given earlier, we will present the logic for the rules of each major validation requirement we mentioned along with their corresponding rule snippets.

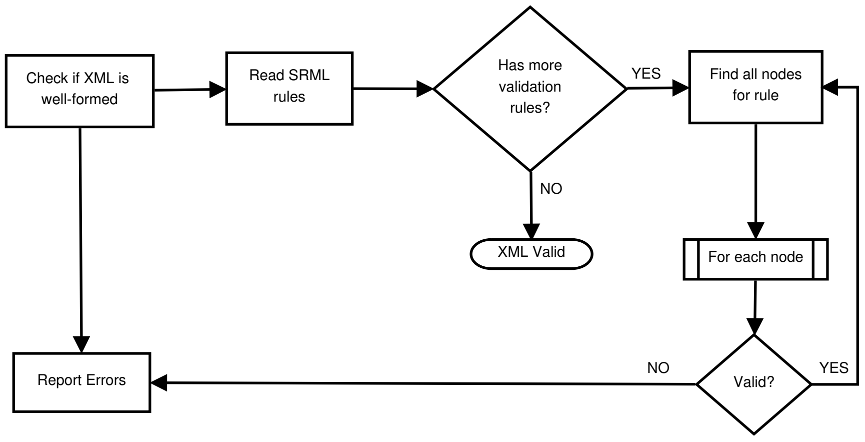


Figure 1.10: XML validation process using SRML

Validation Requirement #1

Requirement: The cart's hasDiscount *attribute is true if there are more than 2 books in the cart.*

The SRML rule for this can be seen in *Figure 1.11.* The rule is rather straightforward; the root is defined as `cart` and contains a `rule-def` name of @*hasDiscount*. The @ sign denotes that it is referencing an attribute. As all paths are converted to XPath, this will reference all *cart.hasDiscount* attributes. The rule uses *"correct"* as the primary validation mode, meaning even if the cart's *hasDiscount* attribute is incorrect the system will attempt to correct it using the expected value based on the rule definition. We can describe the validation rule in the following pseudo-form: Validate the *hasDiscount* attribute of cart: *Count the number of children elements with "book" as their name. If this is greater than two then return true as the validation result, otherwise return false. If the attribute is invalid then correct it based on the rule description.*

Validation Requirement #2

Requirement: All books by "J.R.R. Tolkien" should receive a 20% discount.

This validation requirement needs to reference the `discount` child node of all `book` elements. Since the target of the rule is not an attribute, the @ sign is left out of the reference. The SRML rule for this validation requirement can be seen in *Figure 1.12.*

In this example all `discount` elements are validated that are beneath the `book` elements. The `binary-op` operation is used with an `equal` comparator. The `value-ref` element refers to a value returned by the expression in the path attribute (*`../author`*). This means that it is a sibling (named author) of the current element. The `../` path identifier will go up one level and take the element named in the second part of the path. As our current context is

```
<srml:rules-for root="cart" >
  <srml:rule-def name="@hasDiscount" mode="correct" match="any" >
    <srml:rule-instance>
      <srml:validation-error>Discount value incorrectly set for cart
    </srml:validation-error>
    <srml:expr>
      <srml:if-expr>
        <srml:expr>
          <srml:binary-op op="greater">
            <srml:expr>
              <srml:count-children name="book" />
            </srml:expr>
            <srml:expr>
              <srml:data>2</srml:data>
            </srml:expr>
           </srml:binary-op>
         </srml:expr>
         <srml:expr>
           <srml:data>true</srml:data>
         </srml:expr>
         <srml:expr>
           <srml:data>false</srml:data>
         </srml:expr>
      </srml:if-expr>
    </srml:expr>
  </srml:rule-instance>
 </srml:rule-def>
</srml:rules-for>
```

Figure 1.11: SRML of cart Validation Requirements #1

book/discount the value compare needs to take ***book/author***. This is not a standard XPath identifier, but we decided to implement this to allow easier reference. The pseudo form of the rule is as follows: *Validate the book/discount element content. If the content of the author sibling element is equal to "J.R.R Tolkien" then the discount has to be 20%. If the author is different simply use the discount written in the document.*

In this example the `instance-value` element provides values for the *else* branches of the conditional nodes. This is important since if the validation condition does not match, the actual value should be returned for the attribute/element value in question.

Validation Requirement #3

Requirement: All digital books should be tax-free

This validation rule will reference the `tax` element of the `book`. The condition is that if the *cover* attribute is *digital* then the `tax` value has to be 0, otherwise the actual value will be used. The rule snippet can be seen in *Figure 1.13*. The figure also shows how the `value-ref` element references an attribute value. The example's path of ***../cover*** refers to the parent's cover attribute. Since the mode is set to *"correct"* the validation rule will replace the attribute value of `tax` with *0* if the *cover* attribute of the `book` is *"digital"*.

```
<srml:rules-for root="book">
   <srml:rule-def name="discount" mode="validate" match="any">
      <srml:rule-instance>
         <srml:validation-error>This book is by J.R.R. Tolkien and does not
            have the discount set to 20 percent</srml:validation-error>
         <srml:expr>
            <srml:if-expr>
               <srml:expr>
                  <srml:binary-op op="equal">
                     <srml:expr>
                        <srml:value-ref path="../author" />
                     </srml:expr>
                     <srml:expr>
                        <srml:data>J.R.R. Tolkien</srml:data>
                     </srml:expr>
                  </srml:binary-op>
               </srml:expr>
               <srml:expr>
                  <srml:data>20</srml:data>
               </srml:expr>
               <srml:expr>
                  <srml:instance-value />
               </srml:expr>
            </srml:if-expr>
         </srml:expr>
      </srml:rule-instance>
   </srml:rule-def>
</srml:rules-for>
```

Figure 1.12: SRML of cart Validation Requirements #2

Validation Requirement #4

Requirement: The total entry of the book is calculated by multiplying the quantity, price and discount values

The final validation rule defines the `total` value of the `book`. The new SRML 2.0 library was extended with a regular expression evaluator engine that allows a more precise and less verbose description for this type of rule. In the earlier version of the SRML language a calculation like the above would have taken several lines of `if-expr` and `binary-op` elements and would have seriously degraded the readability. By extending SRML with the `reg-eval` element it is now possible to define mathematical expressions much more easily than before. The snippet for this requirement can be seen in *Figure 1.14*. The validation rule for *total* uses #{..} markers. These inform the expression engine to evaluate the node value with the path expression inside. It is also possible to reference attributes by using the @ marker in the path value. The engine looks up the node/attribute values and replaces them in the expression after which it will evaluate the results.

```
<srml:rules-for root="book">
   <srml:rule-def name="tax" mode="correct" match="any">
      <srml:rule-instance>
         <srml:validation-error>The tax value is not correct as digital books
            are tax free!
         </srml:validation-error>
         <srml:expr>
            <srml:if-expr>
               <srml:expr>
                  <srml:binary-op op="equal">
                     <srml:expr>
                        <srml:value-ref path="../@cover" />
                     </srml:expr>
                     <srml:expr>
                        <srml:data>digital</srml:data>
                     </srml:expr>
                  </srml:binary-op>
               </srml:expr>
               <srml:expr>
                  <srml:data>0</srml:data>
               </srml:expr>
               <srml:expr>
                  <srml:instance-value />
               </srml:expr>
            </srml:if-expr>
         </srml:expr>
      </srml:rule-instance>
   </srml:rule-def>
</srml:rules-for>
```

Figure 1.13: SRML of Validation Requirements #3

```
<srml:rules-for root="book">
 <srml:rule-def name="total" mode="validate" match="all">
  <srml:rule-instance>
   <srml:validation-error>The total value is not correct!</srml:validation-error>
    <srml:expr>
     <srml:reg-eval>
        #{../qty}*#{../price}*(1-#{../discount}/100)*(1+#{../tax}/100)
     </srml:reg-eval>
    </srml:expr>
  </srml:rule-instance>
 </srml:rule-def>
</srml:rules-for>
```

Figure 1.14: SRML of Validation Requirements #4

1.2.3 Using SRML in the field of Databases for Dataset validation

Another large area where the new version of SRML and its validation engine can be leveraged is the field of databases. Nowadays database validation is almost as important as validating the data transmitted from one system to another. Normally semantic validation is done by type-forcing table columns. This means that if one tries to insert a string into an `INT` column then the database engine will report an error. Database tables use pre-defined schemas to ensure that they always contain all fields that are required. One may notice similarities between how RDBMS systems handle and store data to what we outlined with the XSD section of this chapter.

Databases can use triggers to perform input validation. A trigger is a function stored inside the database that takes the input parameters of the actual select/insert/update/delete operation and performs an operation on them. Triggers are usually used in creating audit trails for data modification or used to change the content upon insertion. With triggers, however, it is very complicated to define validation type rules on what the value of the data should be in context to the already existing records.

We decided to dynamically build up a context tree (mini XML) for the given record and allow SRML rules to be executed on it, including XSD type restrictions. This opened up a plethora of possibilities with data modeling and validation. As we have written our validator engine `SRMLXsd` in Java, it made sense to choose a database platform, which allows the utilization of the codebase created for our XML validator. We chose H2 [3] as it is a high performance RDBMS database written purely in Java. It has all the features of major RDBMS systems, but has the benefit of allowing Java classes to be defined as Triggers. *Figure 1.15* shows the validation procedure for databases. The ideas and processes outlined in this section are applicable to all RDBMS systems that allow code to be executed as triggers (e.g.: H2, Oracle, Sybase, Microsoft SQL Server).

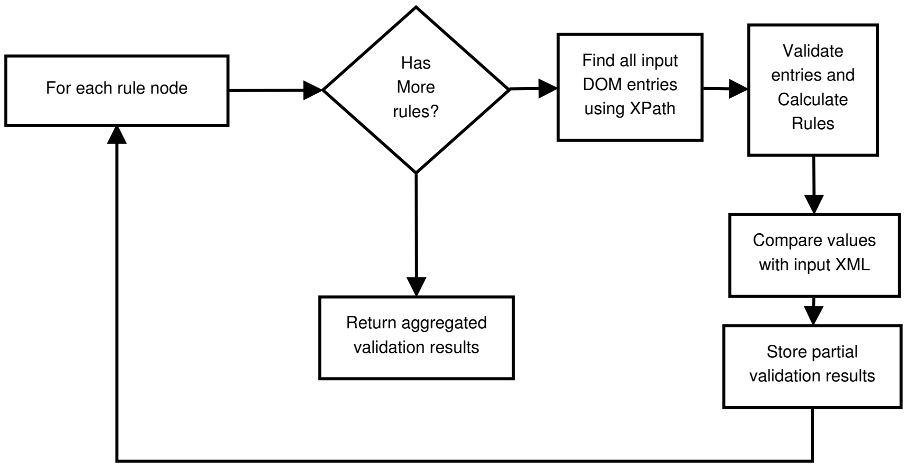

For each rule node
Has More rules?
Find all input DOM entries using XPath
Validate entries and Calculate Rules
Compare values with input XML
Store partial validation results
Return aggregated validation results

Figure 1.15: Validating database records using SRML

There were several challenges during the implementation. One of them was how to extend the rule schema to describe the table relationships and hierarchy. Most database tables have references to other tables and columns (foreign key relationships) that can be modeled using the `database, tables, table, references, reference` tags. With this extension SRML can describe a multi-tier validation scenario (similar to *Figure 1.16*). We will describe the extensions first in short before proceeding further in order to provide a better understanding how they can be used to convert a flat database record into a DOM like tree.

database : The database section stores database-related relationships and definition information. It contains a `tables` and a `references` element.

tables : This element contains `table` elements that can describe the keys of the tables.

table : The table element defines the keys of the tables. They contain a *name* and *key* attribute. These are used to identify the nodes and map them to tables and aid the creation of the DOM tree.

references : This element contains one or more reference element. It is used to store the reference relationships between tables.

reference : The `reference` element has a *root*, *root_key*, *child*, *child_key* attribute used to map out the relationship between records of multiple tables. The keys defined in the table section are used to match the *child_key* entries. This is a form of a foreign key mapping resolution.

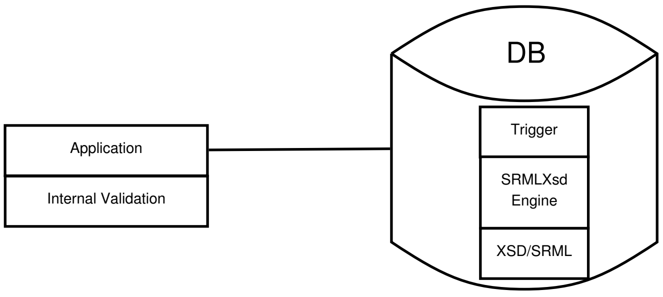


Figure 1.16: Multi-tier validation for applications

Using the above elements it is possible to build up a DOM tree context for a given table row. This is a very powerful addition as it allows references to other tables and their columns using SRML rules. Database related SRML rules have one restriction: as tables have columns and no hierarchic datasets, all `rule-def` references need to use attribute contexts (*attrname*). For example the ***`book/author`*** path maps to a `book` table's `author` column. Using the previous bookstore example, we will demonstrate how SRML rules can be defined as triggers on CRUD operations.

Defining table relationships

In the cart example we had 2 elements: `cart` and `book`. Each element had several children and attributes. For the database example these elements were flattened into two tables, as visible in *Figure 1.17*.

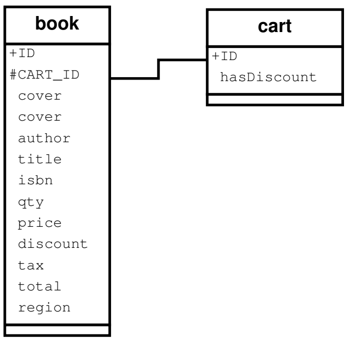


Figure 1.17: Database tables of cart

The SRML definition of the table relationship for the cart example can be seen in *Figure 1.18*. We define the primary key of cart and book and a foreign key relationship between `book.CART_ID` and `cart.ID`. First we have to define the tables that will take part in the context along with their primary keys. This is done with the `srml:table` element. In our example we have a `cart` and `book` table, both with `ID` as their primary keys. The next section of the definition is the `srml:references` element that defines the foreign key relationship between the `cart` and `book` tables. The `root` is the referred table containing the KEY that creates the relation, allowing a DOM tree to be built from the resulting dataset. The generic query that builds this DOM tree would look like: *`SELECT * FROM book WHERE book.CART ID=cart.ID`*. The resulting columns are loaded as attributes along with their values into the DOM tree. This is important to mention, as only the required context will be loaded during the validation. As the CRUD operation is affecting a single row at a time (even if it is part of a transaction), the context will only load the required records into the XML DOM tree.

```
<srml:database>
   <srml:tables>
      <srml:table name="cart" key="ID" />
      <srml:table name="book" key="ID" />
   </srml:tables>
   <srml:references>
      <srml:reference root="cart" root_key="ID" child="book"
         child_key="CART_ID" />
   </srml:references>
</srml:database>
```

Figure 1.18: Table relationship using SRML

Setup Trigger and store in database

We defined a class that can be used as the trigger for all *update, select, insert, delete* operations to leverage the SRML rule engine for data validation. This class implements a `Trigger` interface and has an overridden function that gets called with the old and new rows that the CRUD operation is being performed on. The trigger's classes along with all related classes are packaged into a JAR file and placed on the database engine's classpath so that it is accessible during runtime.

Store SRML XSD inside database

In order for the rules to be accessible by the triggers, they need to be stored in a local table in the database. To achieve this, the XSD file is persisted into a table. This XSD not only contains the SRML rules but also any other XSD restriction we may want to place on the operations. This is useful as we can define what values a given column can take on using standard XSD restrictions and use the engine to validate them during the row validation. This opens up many possibilities, as normal RDBMS systems do not have a way to restrict the actual values a field can have (e.g.: set of values).

Perform row validation using the engine

When all the pre-requisites are in place, the validation is handled automatically with the trigger hook. During CRUD operations (we can define exactly what type of operations the trigger should fire on) the database system will invoke the trigger class and pass in the *previous* row and the new row from the operation. The *previous* row is passed in when an update is being done or when a delete occurs. When the system is performing an insert, then the *previous* row is null. Based on the rows, we look up any affecting validation rules from the SRML set and construct the DOM tree using the reference elements. This DOM tree is assembled using multiple select operations with the reference elements defined (foreign keys). The resultset is then converted into a DOM tree where the attributes of each element are the columns of the table and the nodes themselves are the rows. Taking the rows from the tables in *Figure 1.17*, the system will build a DOM tree described in *Figure 1.19*.

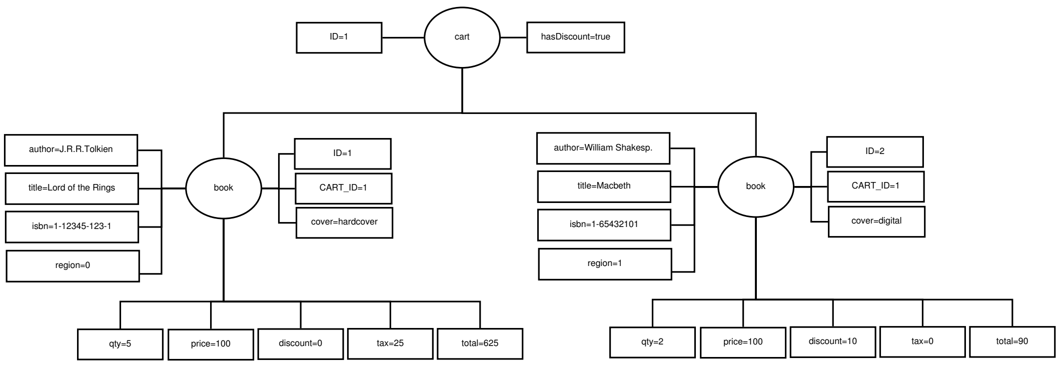


Figure 1.19: DOM tree of the database schemas

After the DOM tree is constructed from the resultset context, the validation proceeds as previously described. If the resultset is not valid an exception is thrown with the text defined in the `validation-error` element. This allows the user of the database system to see what the validation error was, similar to the one in *Figure 1.20*. If the rules had a mode of "correct" then the values are corrected instead of reporting an error. This allows High Availability and Data Oriented systems to retain as much data as possible by correcting input. Data corruption can also happen during network related transmission, making this approach a viable candidate for validation in those fields as well.

```
Error: Validation Error. Message=[The total value is not correct!].
Found=[1625.0]. Expecting=[1125.0]; SQL statement:
insert into BOOK (CART_ID,COVER,AUTHOR,TITLE,ISBN,QTY,PRICE,DISCOUNT,TAX,TOTAL)
VALUES(1,'hardcover','J.R.R. Tolkien','Lord of the Rings',
       '1-12345-123-1',5,100.0,0,125.0,1625.0) [0-169]
SQLState:  null
ErrorCode: 0
```

Figure 1.20: Database validation exception

1.3 Summary

We have shown how the SRML language was extended into the validation space and showed a way to augment the XSD format to allow for both structural and content validation. The aspects of the new SRML format can be summarized the following way:

1. Permits both attribute and element references.
2. Integrates into the `appinfo` section of the XSD, making it easier to deploy.
3. The new format focuses on XML validation in contrast to its predecessor, which focused on making the XML documents smaller.
4. The new validator engine leverages the Java XML Validator, which ensures the well-formedness of the input files aside from the additional validation rules that can be defined on the context of the content itself.
5. Leverages XPath to reference nodes and their values.
6. Makes it possible for the definition of complex validation rules, including regular expressions.
7. Allows the XML document to be corrected using the new SRML rules.

8. Potentially usable in an RDBMS environment. This allows the datasets to be validated using SRML prior to being inserted into the database, using triggers. The datasets and their contexts are built up using a mini-DOM tree permitting the SRML rules to be applied to them. This allows dataset references to existing rows and columns as well.

1.4 Related Work

XML validation has always been a topic of heated discussion amongst the community. There are several advances in the field of XML validation. Most validators, however, only concentrate on semantic validation and do not offer rule based validation scenarios. Currently there are two major pattern/rule based validation projects available that resemble our SRML based approach. Most of the approaches are very well defined and we could have taken one of them as the basis for our extension. The main reason behind going with our SRML 2.0 format was that we have defined the language previously and it has a potential to become a complete solution for both XML validation and correction.

The first project to mention is RelaxNG [39]. It can be considered as the one of the earliest of schema validators. It has a compact syntax and the document is well-defined. It contains non-deterministic content models, however, it does not provide any datatype support and has no support for the XSD numeric occurrence constraints (in XSD it is possible to specify the *minOccurs/maxOccurs* attribute, which will inform the validator of the quantitative property). In RelaxNG the attributes are defined as part of the content model, providing a homogeneous view of the XML tree, similarly to how the DOM tree represents the XML tree. RelaxNG was a merge between Relax and TREX (Tree Regular Expressions for XML). *Figure 1.21* shows a simple rule definition of a simplified book-cart example in RelaxNG format. The definition is similar to how XSD defines the structure. However, does not offer data correction out of the box, making the SRML a better option for this purpose.

One of the best known pattern based validators available is the Schematron [58] project, which was also recorded under ISO/IEC 19757-3:2006. The authors of this project initially started out by extending the Word UML format used by Microsoft products [45] [46] and introduced a language to model the relationships. This approach allows many types of structures to be represented and enables the developer to perform reporting and assertions on them. The project can also use XPath for finding nodes. This approach is focused on validating XML files using rules and assertions. It is very powerful, but lacks the option to correct the document. Our approach not only validates using rules, but also allows the XML document to be corrected if the rule definitions specify it. The subset of the example (total value calculation) in Schematron

```
<start>
    <element name="cart">
      <zeroOrMore>
        <ref name="book_entity"/>
      </zeroOrMore>
    </element>
  </start>

  <define name="book_entity">
    <element name="book">
      <attribute name="cover" > <text/></attribute>
      ...
   </element>
  </define>
```

Figure 1.21: RelaxNG example

can be seen in *Figure 1.22*.

```
 <pattern name="Book Total value check">
    <rule context="book">
      <assert test="total != qty * price * (1- discount /100)*(1+ tax/100)">
         Total Mismatch
      </assert>
    </rule>
  </pattern>
```

Figure 1.22: Schematron Example for value validation

Another project that should be mentioned is CAM [1], which is short for Content Assembly Mechanism. CAM is different from other approaches as it does not define complex grammars, but rather approaches the validation from a structural pattern-matching front. The language allows business rules to be defined using XPath references and corresponding actions (e.g.: $condition : string - length(.) < 11 \; action : setDateMask(YYYY - MM - DD)$). It also allows cross- and current-node conditional validation (e.g.: quantity needs to be between 1 and 100). The rule definition is more compact than SRML, however it still lacks the data correction feature our approach allows.

Chapter 2

Validating Web Forms

Thesis: Create a new jSRML metalanguage, which is capable of defining semantic rules for the validation and correction of web forms.

Introduction

During the initial design of SRML 2.0 we branched another metalanguage from SRML 1.0 called jSRML. This language takes its origins from SRML 1.0 and has some basic similarities to the SRML 2.0 language as well. The jSRML language was re-written from ground up to provide a powerful way to describe validation rules for web forms. In this chapter we will demonstrate how it can be used to create a versatile validation approach, and how it can be leveraged to learn validation rules based on form inputs. We decided to investigate the HTML [48] space as its documents are very similar to that of the XML documents. HTML forms contain fields that are filled out by the users, which are subsequently submitted to the server for processing. The server then processes this information and returns the results or performs an operation with the submitted data. These web forms can range from simple user login forms all the way to online tax returns containing and exchanging sensitive information. Unfortunately this is one of the weakest links in the whole systems, which many hackers try to exploit. The most common form of attacks against web forms is *DoS* [27] (Denial of Service), which basically means that small automated scripts perform constant form posting against sites trying to exploit the data or cause the service to slow down or even crash. This can potentially compromise the site, granting the malicious script access to protected resources. This type of exploit is also used to spam forums and news portals. Even if the data transmission itself is protected using a secure channel (e.g.: SSL) the data entered still needs to be validated prior to performing the processing. Another common exploit method is the notorious *SQL injection attack* [14]. This method is based on the assumption that the fields of the forms are eventually inserted into the database. If the form processor does not filter the input (e.g.: by using prepared statements, or by filtering the fields for SQL commands) then it is very possible to issue SQL commands against the processing database (for example DROP TABLE). Besides a security point, data validity is a crucial aspect as well. Consider a lead generation form where users need to fill in their contact information in order to receive special offers from the provider. If the data entered is incorrect then it can cause a potential lead to be lost causing the owner monetary damage.

One of the most common types of validation scenarios is the user registration form. Here the user fills in his personal information, along with an email and password and submits it for processing. The email address has to be valid, otherwise the provider cannot communicate with the user, the passwords have to conform to some security restrictions...etc. All these requirements can be handled by using some kind of form validation method. The most common is asynchronous validation using *JavaScript* [19]. Using this approach the author of the page writes JavaScript code, which checks the fields of the form, providing visual output to the user (e.g.: if the email has an invalid format then the field may be highlighted). This type of validation can be very powerful and is handled on the client side, which means the user will not experience any lag during the submission. The biggest drawback, however, is that by adding more fields to the form the JavaScript code processing logic becomes more difficult.

The second type of form validation is *Server-side* validation. This basically means that the form data is posted to the server, which then processes the content and returns an error if the form was invalid, or saves the data if it was valid. This is a good approach, however it will cause an overhead when the user has to re-enter the form contents due to a mistype in one of the fields unless the owner explicitly codes the retry logic. The process will not happen asynchronously, meaning the page will be reloaded during the submission (excluding cases when this is handled with an AJAX [23] call).

To provide a solution to these issues, we have created a jQuery [43] based validator called `jSRMLTool`, which leverages the SRML 2.0 language. This was extended to allow form based validation rules. With our new jSRML extension, users will be able to define SRML rules for web forms and their fields, describe relationships and requirements for their content. The engine can be used in any HTML page simply by including the script file in the document and defining the validation rules. This approach ensures that the HTML content is not encumbered with JavaScript code. The jSRML rules need to be placed after each field that is to be validated and the engine will handle the rest. We will detail how this approach works in a later section of this chapter.

An off-site asynchronous implementation of the jSRML engine was also created using Servlets capable of validating forms that employ unique identifiers and jSRML rules. This is a separate service running on a remote machine using stored rules to validate the form and return with any potential validation errors. Our approach also allows another powerful feature: data correction. Thanks to the nature of the jSRML language, it is possible to define self-correcting form validation rules. These rules correct the field values based on the rule definitions, wherever applicable, making the form submission succeed. The Servlet also has provisions to learn potential jSRML rules using the submitted form data and machine learning.

2.1 Preliminaries

Before we introduce our new method, we should cover a few topics in order to make the chapter easier to understand. We will not detail each technology in depth, rather just cover the parts that are relevant to the later sections.

2.1.1 HTML and DOM

Forms are described using the HTML [48] language. These documents have a similar hierarchic structure to XML where each node can contain attributes or additional child nodes. This hierarchic tree-like representation is also known as the *DOM model* (described earlier in Section 1.1.1 of Chapter 1). *Figure 2.1* shows a simple HTML form source with a field. The DOM tree representation of *Figure 2.1* is shown in *Figure 2.2*.

```
<html>
   <head><title>Hello World</title></head>
   <body>
     <h1>Hello World!</h1>
     <form method="post" action="process.php">
       <label for="username">Name:</label><input type="text" name="username" />
       <input type="submit" value="Submit" />
     </form>
   </body>
</html>
```

Figure 2.1: Simple HTML of form

2.1.2 Types of form validation

There are four major types of form validation: *Client-side*, *Server-side*, *Real-time* and *Hybrid*. The difference between them lies where the data is validated and processed.The different types of form validation are summarized in *Table 2.1*.

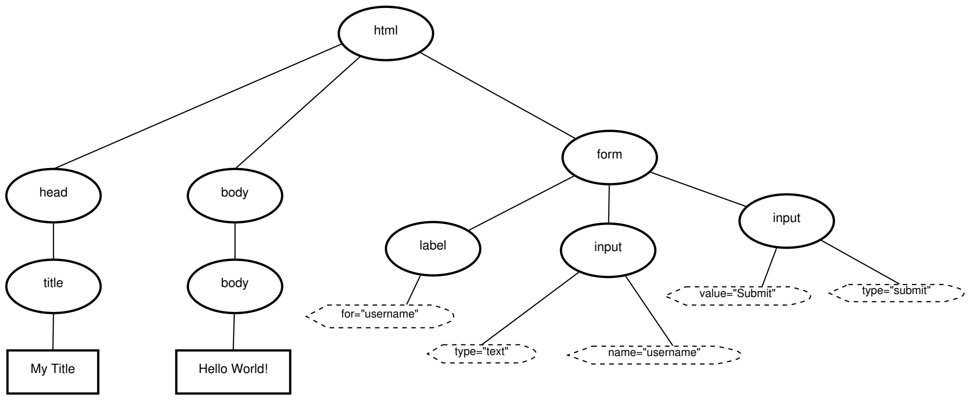


Figure 2.2: DOM tree of the Form Example

Type	Trigger	Processing	Validation logic	Advantage	Disadvantage
Server Side	Form Submit	Sequential	Returned to browser for display of results	Validation logic hidden from user	Validation changes require server updates
Client Side	OnClick intercept	Client side	Shown in browser using JavaScript	Fast, since no data is sent to server	Validation logic visible to users
Real	Field change	Either	Direct call to client and/or Server validation	Field values validated real-time prior to form submission	More traffic required, harder to update
Hybrid	Field change and Submit	Either	Direct calls with round-trip to server	Allows two stage validation, pre-filtering results prior to sending to server	More complex to implement and maintain

Table 2.1: Validation types

2.1.3 The jSRML extension

The previous SRML rule engine implementation used the DOM tree of the XML to perform its operations. Since HTML forms can be considered as DOM [2] trees, it made sense to attempt to apply SRML to this area as well. In this chapter we introduce an extension of SRML (called jSRML) which allows its use in the form validation space. We have created a new rule engine for this purpose using jQuery where the processing is performed in the browser.

The new jSRML language, although being an extension of SRML, is not completely identical to its predecessor as it was rebuilt from ground up taking the positive traits of the previous language version and molding it to become an ideal candidate for describing form validation rules. *Table 2.2* shows the differences between the different versions of SRML.

Property	**SRML 1.0**	**jSRML**
Main Focus	Compaction	Validation/Correction
Reference level	Attributes	Form Field values
Application Area	XML Documents	HTML Forms
Rules based on	Attribute Grammars	XPath and DOM
Rule Definition	Complex	Simplified
Rule Locations	DTD and SRML file	Inline, external, server
Rule Processing	Application side	Client-,Server-side, Mixed

Table 2.2: Key differences between SRML versions

2.2 Extending SRML for form validation

In this section we will present how the SRML language can be extended to aid the validation process. Most *Client-side* validators are simplistic and perform format validation only. If we wanted to create a validation rule that conditionally compared two fields then it would require a larger block of JavaScript. Trying to achieve this on the server would require the validation logic to be implemented there. If for some reason the conditions needed to change then the server code would need to be updated, which can be difficult in production environments.

We took the positive traits of the original SRML 1.0 language and compaction engine (`SRMLTool`) and rebuilt it from the ground up in JavaScript using jQuery to allow exceptional browser performance. We decided to name the extension jSRML and the new rule engine `jSRMLTool` to denote the JavaScript relationship. Previously SRML rules were stored in a separate file, which had its advantages and disadvantages. The advantage was that all the rules were in one location, however, this also meant that it was harder to understand the rules when trying to find a ruleset for a given node context. In the jSRML approach we allow the rules to be defined in-line after each field as well as externally, making it easier to define validation rules.

The second advantage of jSRML is that it is non-obtrusive. In order to use it, only a simple script include is required. When the validation rules need to be updated the rule engine itself will not change, only the rules, reducing the possibility of error. This is a very large benefit compared to the pure JavaScript approaches. If the validation rules need to change then only the affected field rules need to change, no coding experience is needed to perform the update. In case of in-line jSRML, the rules are defined as jSRML snippets. The full XSD of the new jSRML language can be found in *Appendix B.1.*

The jSRML engine can also correct the field values if the rule definition specifies it. This is a huge advantage over other rule- or JavaScript-based validators as it allows the form to correct the errors and still allows the form submission to succeed. A good example would be spell checking in a form prior to submission, which can be accomplished by the using functions in the rule definition. This makes jSRML more versatile as more seasoned developers can extend the engine with additional methods besides the standard operation set that the engine provides.

We have also created a *Server-side* implementation of the jSRML engine using Java Servlets [30], allowing the form to be validated asynchronously against a service. The service code does not change no matter what the rule definitions are. This is accomplished by storing the ruleset on the server-side and performing the validation based on a lookup using a unique form identifier. This Servlet can be used to validate thousands of different forms spanning multiple domains as long as the rules were uploaded beforehand. This allows the engine to be leveraged in an on-demand validation service scenario. The `jSRMLTool` servlet also has an option to learn the validation rules based on the form inputs using extensible machine learning methods. This provides a powerful tool for the owner as it can also "mine" the input and gradually adjust the rules based on what users entered.

2.3 Validation using jSRML

We will show how to define jSRML rules using simple snippets. The current language format allows two ways of defining rules : *in-line* and *external*. The *in-line* mode allows the user to insert the validation rules right below the affected field. This makes the code more readable as the validation rule follows the field itself. *Figure 2.3* shows a simple example of providing an email validation rule using *in-line* jSRML.

```
...
 <input type="text" id="email"  class="row-item" />
  <!--[SRML]
    <validate-input id="email" form="myform" mode="validate">
       <error-text>Invalid email format!</error-text>
       <css invalid="inp-form-error" error-class="form_error_message error" />
       <action valid="" invalid="error" />
       <conditions>
           <expr>
               <text-format value="email" />
           </expr>
   </conditions>
    </validate-input>
-->
...
```

Figure 2.3: jSRML snippet for in-line email validation

To initialize the engine for in-line (default) validation mode, the following steps would be needed:

- Include the *jSRMLTool.js* file at the start of the document.
- Augment the fields with their proper in-line rules.

In-line validation rules are contained in a comment block following the field. The comment starts with the [SRML] tag. The advantage of using comments for the rule storage is that they are non-obtrusive and can be accessed within the DOM model using XPath [18] expressions.

For external includes we use jQuery to load an XML document containing the rules into a DOM object and use that as the source for the engine. As this is not the default mode that the engine uses, there is some extra setup required for this mode to be used. To use external rules the following steps need to be taken:

- Create a script segment with the following contents :

  ```
  var external_rule = http://location-of-srml-rules;
  ```

- Include the *jSRMLTool.js* file.

The major difference between *external* and *in-line* is that there is an extra step required. The presence of an *external_rule* variable informs the `jSRMLTool` engine to load the rules from that location using AJAX during the page load. The rules are then pushed into a rule DOM object for easier access. From this point on the validation process is identical to the *in-line* approach.

2.3.1 Defining validation rules

After demonstrating the two ways to define rules, we will now describe how a rule is built up and how to define more complex ones.

Every jSRML rule definition starts with the `validate-input` tag. This element specifies what the scope of the given rule is using the *id* attribute. The *form* attribute defines which form the rules belong to. This way the *external* and *in-line* rules can both use the same format, making it easy to switch between them. The third parameter is the *mode*, which can have a value of *"validate"* or *"correct"*. The first mode will validate the rule and return accordingly. The *"correct"* mode allows the form input field to be corrected by the actual rule calculation result. This means that if the validation fails, then the field value will be replaced by a pre-defined or calculated value (Expected value) allowing the validation to potentially finish successfully.

The `validate-input` element has 4 child nodes. These can be in any order, but they must exist for the validation to yield proper results. These elements are as follows:

- **error-text:** This element contains the validation message that will be displayed to the user. This message is put in a dynamic *div* element that is created after the field that is being validated. A *div* is an HTML element which can have an *id*, *name* and *class* attribute. Divs are used in modern web pages to provide table-less layouts and define specific regions of the page. For the scope of this chapter it is enough to consider them as containers that can be manipulated similarly to other DOM elements.

- **css:** The css element allows the author to define what CSS classes should be amended to the input field in case of an error and what class the newly created error div should be. CSS [42] stands for Cascading Style Sheets and is widely used in styling web pages. It defines a set of styles and classes, which can be applied to elements in the document.
- **action:** This element allows the definition of additional functions that will be invoked in case of a validation error or success. This allows more extensive callbacks to experienced users who wish to perform custom operations, depending on the output of the form validation results.
- **conditions:** This element stores all of the validation rules.

The `condition` tag contains one or more `expr` tags. The validation succeeds or fails based on the result of these expressions. It is possible to define more conditions for the same field using multiple `expr` nodes. There are several expression types defined in jSRML. We will detail the most important ones along with a brief description.

- **binary-op:** This defines a binary operation. In jSRML we only allow a subset of `binary-op` types on the top level expression, more specifically ones that return a true/false value. Currently these are limited to: *gte*, *gt*, *lte*, *lt*, *date-lte*, *date-lt*, *date-equals*, *date-gt*, *date-gte*, *equals*, *not-equals*, *contains*, *not-contains*,*begins-with* and *ends-with*. The specification also allows the keywords *and* and *or* to enable proper logical operations. We have introduced the *reg-eval* element which, allows references to nodes and most binary operations (+, -, /, *). A `binary-op` contains two `expr` expressions. The operation is performed between the two expressions. The expressions within can also be other binary-ops or one of the expression types described in this chapter.
- **text-length:** The `text-length` element returns the length of the actual field that the rule is defined for.
- **field-length:** This element is similar to `text-length`, however, it also has an attribute called *id* that identifies the specified field whose length needs to be returned.
- **text-value:** This expression will return the value of the actual field that the rule's definition was for.
- **field-value:** Similar to `text-value` but allows the reference of another field's value by *id*.

- **data:** The `data` element allows literals or constants to take part in an expression. An example for this would be when the length of a field has to be larger than 100. In this case the 100 would be added as a `data` tag.
- **text-format:** The `text-format` expression returns true or false based on the type of field value it is matched against. The *value* attribute can be *date*, *numeric*, *email* or *regexp*. This allows easier validation against standard field types used in forms, such as emails, dates or numbers. The *regexp* type allows the definition of a regular expression defined in the *expression* attribute. This allows powerful pattern matching for fields (e.g ISBN number validation).
- **reg-eval:** This expression type allows operations to be defined on more fields at the same time. For example if the field value is only valid if it is the sum of other two fields then a `reg-eval` expression can be used. To reference the value of fields in the expression, one simply needs to enclose the *id* of the fields in brackets (e.g.: $[\{fieldName\}]$).
- **if-expr:** The `if-expr` element allows conditional results to be returned. It takes 3 `expr` expressions. If the result value of the first expression is true then the result of the `if-expr` will be that of the second `expr`, otherwise it will be the third `expr`.
- **has-value:** This element allows a simple check of the field contents. If the field referenced by *id* is empty this element will return false, otherwise it will return true.

The jSRML language allows the form values to be corrected based on the rules. The engine will find the rules for the actual field and if the value of the field is different than the expected value defined then it will use the result of the rule as the actual value. This allows forms to be corrected based on the rule values, making it a very powerful tool in the form validation space.

2.3.2 A form validation example

After introducing the jSRML language and how powerful it can be for form validation, we will provide a summary example to demonstrate how it can be used for form validation.

Consider the form in *Figure 2.4*. This form has multiple fields to better demonstrate how `jSRMLTool` works. The full source of the HTML page can be found in *Appendix B.2*. The following shows some summarized validation rules for the form:

- `Field01 has a minimum length of 5 characters`: the `text-length` element is used, which returns the length of the actual field (in this case the length of *field01*). We then compare this to a constant value of *5* defined in a `data` element. To perform the comparison logical operator, we use a *gte* binary op. This will return true if the first expression's value is larger than the second.

- `Field04 has to be an ISBN number:` This is a special `text-format` case as it is using the `reg-exp` type to define a requirement of an ISBN number. The *expression* attribute defines the actual regular expression that the field's value will be validated against.

- `Field06 has to be the sum of Field02 and Field05:` For this rule we use `reg-eval`, which is coupled with an *"equals"* `binary-op` against the actual text value.

- `Field11 is ‘‘legs’’ if field10 is ‘‘cat’’, ‘‘wings’’ if field10 has a value of to ‘‘bird’’ and can be anything otherwise :` The validation rule contains an `if-expr` to match the value of the other field value against *"cat"*. If the value was *"cat"* then the validation result will return the value *"legs"* as the required field value. Otherwise the results will be the `text-value` of the node and will perform an *"equals"* `binary-op` on it. This is a simple trick to convert the machining of fields to booleans, since if the value matched then we return the current field value and compare that against itself (which will always be true), otherwise we would return *"legs"*.

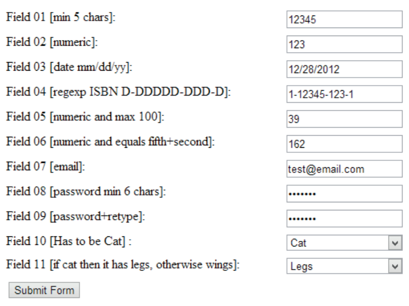
The `jSRMLTool` engine supports all three types of validation described earlier (*Client*, *Server*, *Real-time*). This provides the most versatile and powerful approach since the user is not bound to a single solution.

The following summarizes how the different modes operated in `jSRMLTool`:

- `Client-side:` In this mode the validation is completed using the included jSRMLTool.js file. The rules are extracted using XPath conditions. All *in-line* rules are contained in comments which start with [SRML]. A hook is installed on the *onClick* action of the submit button. When the button is pressed the engine will validate the fields. If the validation is successful (or corrected based on the expected values) then the form is submitted to its original location defined by the "action" attribute of the form. *Figure 2.5* shows the flow of the *Client-side* validation.

- `Server-side:` The engine handles the *Server-side* mode using a separate servlet (called `jSRMLToolServlet`). This servlet uses a unique identifier to associate the rules to each form. This allows multiple forms from different domains to be submitted/validated against the same servlet. To put the validation engine into server mode a variable called *server_validator* needs to be defined with the URL of the servlet. The flow in this case is similar to the *Client-side*. However, all fields are pushed over to the servlet along with the unique identifier. The servlet then performs the validation/correction and returns the data back to the client. The *Server-side* validation flow is shown in *Figure 2.6*.

- `Real-time and Hybrid:` Every rule has a *"method"* attribute. This is not a mandatory attribute and has a default value of *"standard"*. When this attribute is set to *"focus"* then a hook is automatically installed on the *onBlur* event of every field where this attribute is set. This results in a focus change validation trigger. The third allowed value for the *method* attribute is *"real-time"*. This installs a *keydown* listener and performs the validation on every character input. This mode is useful for example in case of password length checks.



Field 01 [min 5 chars]:	12345
Field 02 [numeric]:	123
Field 03 [date mm/dd/yy]:	12/28/2012
Field 04 [regexp ISBN D-DDDDD-DDD-D]:	1-12345-123-1
Field 05 [numeric and max 100]:	39
Field 06 [numeric and equals fifth+second]:	162
Field 07 [email]:	test@email.com
Field 08 [password min 6 chars]:	•••••••
Field 09 [password+retype]:	•••••••
Field 10 [Has to be Cat] :	Cat
Field 11 [if cat then it has legs, otherwise wings]:	Legs

Figure 2.4: Input form

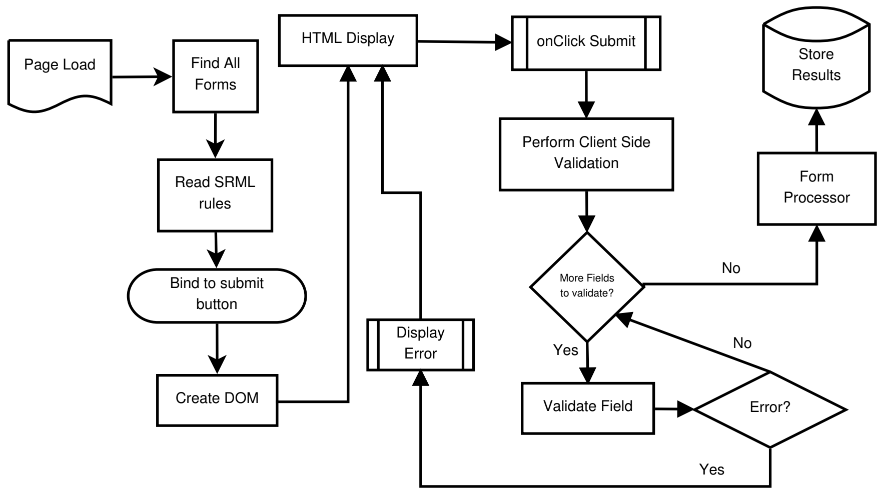


Figure 2.5: Client-Side jSRML

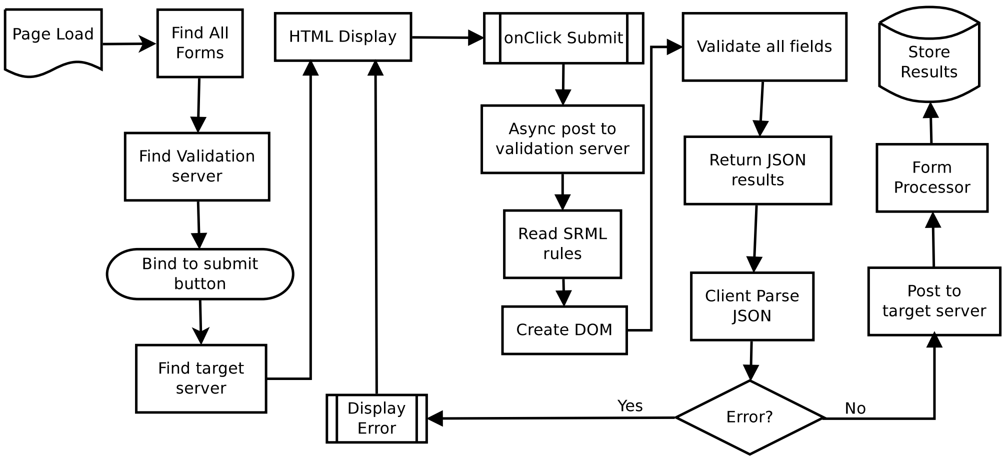


Figure 2.6: Server Side jSRML

2.4 The jSRMLTool Servlet

After introducing the jSRML language and the `jSRMLTool` engine we will now discuss the *Server-side* validation mode in more detail. The `jSRMLTool` servlet has two major roles: *Server-side* form validation and learning jSRML rules. The first role allows a powerful way to provide a service for validating forms across multiple domains. The jSRML rules are stored in the database and are retrieved using unique identifiers. The form is passed in to the Servlet, which performs the validation internally and returns the results to the calling client. This approach hides the rules from the client side, yet still allows powerful validation using jSRML.

2.4.1 Learning jSRML rules

The second role of the `jSRMLTool` engine is learning jSRML rules. This is a powerful addition since it attempts to learn from the form submissions and can propose jSRML rules based on machine learning techniques. In order to learn jSRML rules, the engine has to be put into learning mode using the following steps:

1. Create a JavaScript variable called *server_mode* with a value of *"learn"*. This will put the engine into learning mode. The default value of this variable is *"normal"* .

2. Create a variable called *server_validator* with the location of the validation servlet.

3. Include the jSRMLTool.js file into the header of the form's file similarly to the *client* or *server-side* modes.

4. Augment the form with a hidden variable called *srml_unique*. The value of the variable should be the identifier that will be used to group the form submissions together.

Figure 2.7 demonstrates how the form is intercepted and analyzed. The initial steps are similar to how the *Server-side* validation is handled. A hook will be installed on the form's submit event and will re-route the call to the jSRML Servlet location. The major difference here is that there is no actual jSRML ruleset on the Server-side. It is merely used to intercept any submissions and store the form-value pairs. These values are then analyzed by the learning module and possible jSRML rules are generated. The flow is returned to the client and the form data is pushed to the original target for the form submission. This means that the form operation is not hindered but the traffic is intercepted, saved and submission relayed to its original target.

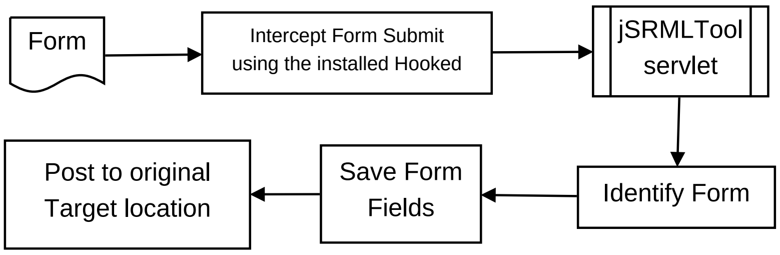

Form
Intercept Form Submit using the installed Hooked
jSRMLTool servlet
Identify Form
Save Form Fields
Post to original Target location

Figure 2.7: Intercepting form data and learning jSRML rules

The learning module has several plugins that process form submissions and adjust the proposed rules accordingly, making the learning a gradual process. Currently the engine has the following learning plugins: *jpFormat*, *jpLength*, *jpCopyContent*, *jpRelationship*, *jpRange*, *jpPredefinedName*, *jpRegExp*.

Each plugin has a *confidence factor* and a *target ratio* that is set by the administrator of the system. If a plugin has a high *confidence value* it means that almost every time the plugin breaches the *target ratio* threshold a rule will be generated. Sometimes it is possible that multiple plugins provide rules for the same field. In cases like this the system chooses the solution with the highest *confidence factor* which surpassed the *target ratio*. The *target ratio* denotes what the minimum expected matching ratio is, which means that if the actual match is lower than this ratio the rule will not be considered as a match. In practice this means the ratio of inputs that match the given rule conditions.

The plugins keep track of their historical form submissions along with their field values. The learning module goes through all the plugins and collects the partial jSRML rule proposals. Once all the plugins are executed the weighed results are analyzed and stored. *Figure 2.8* demonstrates how the learning module works. To increase the efficiency of the learning process it is usually helpful to start a new ruleset with a supervised learning scenario. During this the owner of the form ”teaches” the engine by providing valid sample inputs. The tool also has an import feature which is able to import a CSV file of valid sample data to prime the initial rules. Since the learning module is very extensible, new plugins can be added easily, increasing the learning efficiency of the system.

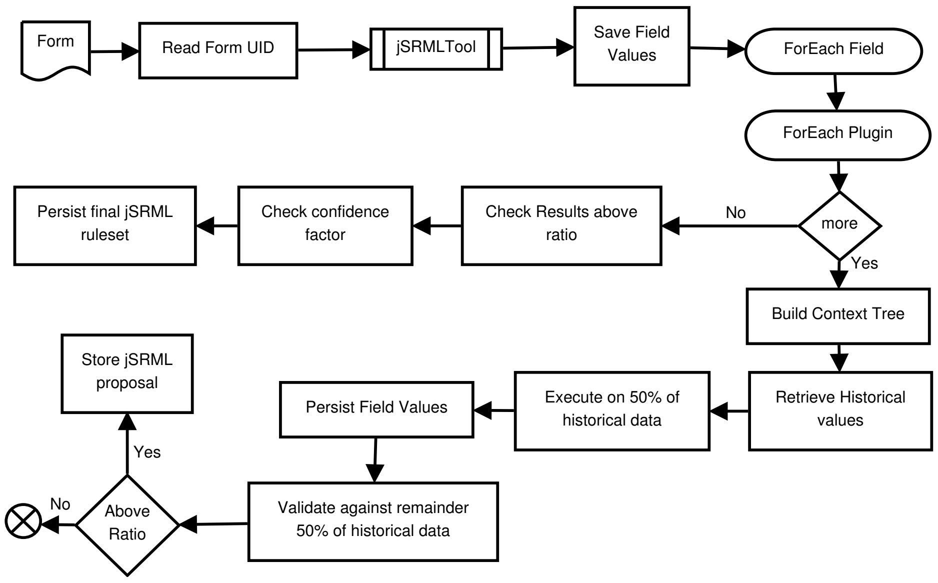


Figure 2.8: jSRMLTool learning process

jpFormat Plugin

This plugin tries to match the type of a given field. It works on a simple approach that every field is a *string* as the weakest type match. It then tries to cast to *date*, *email* and *numeric*. The matching is done by casting and regular expression pattern matching. The results are stored on a fieldname level along with the statistics of the match. The decision adopts over time since it is possible that not all submissions are valid. The plugin has a high success rate at identifying the formats, since the more positive/negative examples it receives the higher probability the match will be.

jpLength and jpRange Plugins

The *jpLength* plugin matches on the length of the fields. Both minimum and maximum lengths are collected and analyzed. The operation is pretty straightforward thanks to the historical data collected. The *jpRange* plugin works similarly, however, with the actual numerical value of the fields. The range, min and max values are adjusted after each positive result. These plugins are dynamic in nature and adjust their values based on the submissions.

jpCopyContent

This plugin is a simple comparator between two fields. It is mostly used in the password, email fields when there is a second field which requires the user to re-type the value to ensure he didn't make a mistake. The operation of this plugin goes through all (F_j, F_k) field pairs and checks what the matching ratio is between them.

jpRelationship

The relationship plugin is aimed at finding relationships between fields and their values. The steps of the plugin are demonstrated in *Figure 2.9*. The learning starts out by extracting the context of the form submissions. Since the context tree has only two levels (including the root) every field is a sibling. This plugin has two sub-modes: *compositional* and *conditional*.

The *compositional* mode finds potential compositions between the other sibling elements. The current version works off sets of two concurrent fields at a time (using more fields would increase the complexity), each field with a minimum length of 3. Based on the possible combinations we build a statistical table to show each field in relation to two other siblings. For composition we check against: `begins-with`, `ends-with`, `contains`. If *field01* is the field the plugin is targeting and *field02* and *field03* are in the current context set then the value is compared against: *[field02][field03]*, *[field03][field02]*, **[field02]*, **[field03]*, *[field02]*[field03]*, *[field03]*[field02]*. The plugin will go through every field as the target field. It then takes the remainder *(n-1)* siblings and splits them into groups of two based on those fields whose lengths are above 3 characters. These combinations are then compared to the historical values of the plugin. Based on the *confidence factor* and ratio provided a jSRML rule is created. *Figure 2.10* shows the compositional method of the plugin.

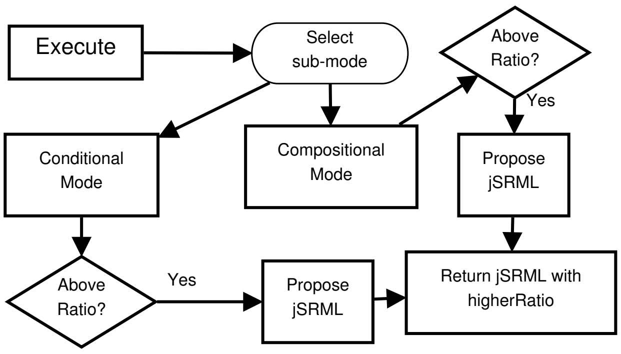


Figure 2.9: jpRelationship Plugin

The second mode of the *jpRelationship* plugin is the *conditional* mode (*Figure 2.12*). This method finds relationships between field values using conditional logic and applying statistical machine learning [28]. The plugin uses

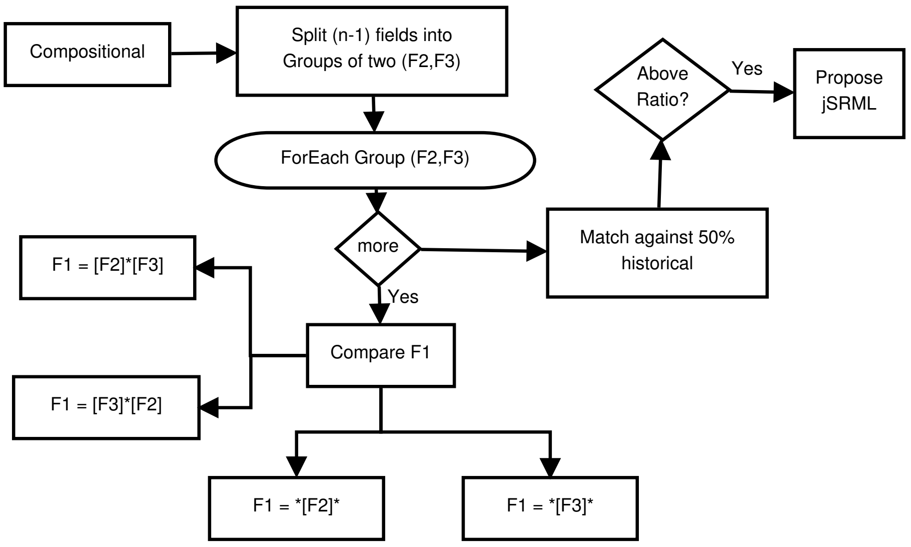


Figure 2.10: jpRelationship Compositional Method

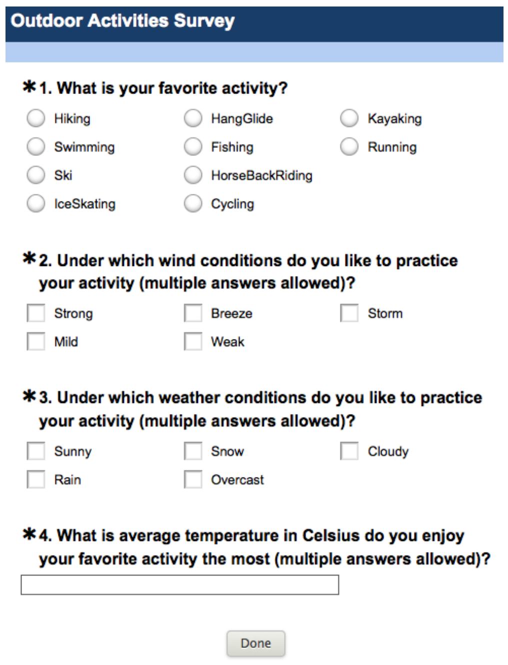
50 percent of all historical data as the learning set. The plugin initially selects the most descriptive field F_k where *k=1,...,n* and bags its context (the remainder *n-1* fields) clustering them into groups of three randomly. These clusters will form a set of decision trees that are focused on learning F_k using a simplified Random Forest [16] approach. It should be noted that the size of the clusters is an experimental value based on the average number of form fields per submission. The term "most descriptive field" refers to the field with the lowest entropy in the results (the field whose values are least random across submissions). This is used to better split the values of the results into smaller chunks, which are then used in the later nodes of the tree. Every tree will have a maximum depth of 3 (as the selected field's bag has 3 other fields that have to be analyzed). Each node's content contains the actual values of targeted field F_k and its top three values (F_k was selected at the start of the algorithm). Every node will select the most descriptive field and its value in the current context. The context is unique to each node and the path that it was created by. This means that every field's possible values in the current node are influenced by the previously selected classifiers leading to the node. We will be using X_i to denote the filter context of a node in each iteration step whose value is unique to the node's path in the tree. Let $X_i := F_k[F_r = V_s(F_m[X_{i-1}]$ where $V_r(F_s[X_i])$ denotes the rth most descriptive value of field F_s filtered by the context defined in X_i. Let $C(F_r[X_i])$ mark the classifier that is selected for field F_r whose values are filtered by the context defined in X_i. During each node the field (F_r) with the most descriptive trait is selected as the classifier (every level of the tree reduces the number of fields to chose from by one). This field's values are then used to create the nodes children ordered by their descriptiveness. Each child node will fix the value of F_r based on the branch

they are in $V_1(F_r[X_i]), ..., V_n(F_r[X_i])$. The main F_k field values and their occurrences are recalculated based on the context in each node. Every node will reduce the possible values of the fields as the context is generalized more going downward in the tree. It is possible that some field values are not discrete, but rather continuous numerical occurrences. To solve this scenario, $W_m(F_s[X_i])$ marks the weighed values of F_s filtered by X_i with a relation of m (possible values $\leq$, $>$). The algorithm chooses a weighed average of numeric values (to ensure that they are not offset too much). For these classifiers the values will partition the results into two sets. The first branch will contain values less than or equal to the classifier value, the second branch will contain values larger than the value. This function is analogous to the $V_m(F_n[X_i])$ value and can be used in the classifier filtering accordingly. However, here the value is not based on the level of descriptiveness but rather the weighed average of the field and its filter chain.

As mentioned earlier each node contains the top three values of the analyzed field (F_k) with their occurrence ratio. The possible values of the fields are influenced by the previously selected classifier values. Before selecting a new classifier the algorithm checks the values of F_k in the nodes. Any node which does not have at least one F_k value above the ratio (currently set to 50%) is ignored from then on and will no longer be processed. The iterations continue until the context bag is not empty or all nodes have terminated without a possible selection. The algorithm only works off the top three values of each field classifier, which may cause an efficiency decrease overall. However, based on the introduced ratio values the margin for extra error can be safely ignored.

To demonstrate the algorithm consider the following example: users answer a set of questions regarding their activities and weather conditions (*activity*[F_1], *wind*[F_2], *weather*[F_3], *temperature*[F_4] where the brackets contain the Field index). The form data was acquired using an online survey using the help of *SurveyMonkey* [55]. The fields *wind* and *weather* allow multiple values to be selected (the form can be seen in *Figure 2.11*). When the user selects multiple values for these fields the form post is handled as multiple submissions to fit the model correctly. The plugin uses 50 percent of the historical data (in our case 2000 submissions) and analyses each field one-by-one. We will demonstrate the *activity* field relationship learning briefly. *Figure 2.13* shows the resulting tree for *activity* (note we only have 4 fields in this form, so it will only need one tree per field. However, the algorithm works on multiple trees as described earlier). The plugin collects the distinct historical values and their counts selecting the top 3 values. In case of *activity* these top 3 distinct values are *"Swimming"* with 610 hits, *"Fishing"* with 239 hits and *"IceSkating"* with 215 hits. The learning set in our example is made up of 2000 form submissions.

The plugin creates a statistical analysis of the other $(C(F_2), C(F_3), C(F_4))$ classifier values. In our example *wind*[F_2] is chosen as it had the most de-



Outdoor Activities Survey

***1. What is your favorite activity?**

○ Hiking	○ HangGlide	○ Kayaking
○ Swimming	○ Fishing	○ Running
○ Ski	○ HorseBackRiding	
○ IceSkating	○ Cycling	

***2. Under which wind conditions do you like to practice your activity (multiple answers allowed)?**

☐ Strong	☐ Breeze	☐ Storm
☐ Mild	☐ Weak	

***3. Under which weather conditions do you like to practice your activity (multiple answers allowed)?**

☐ Sunny	☐ Snow	☐ Cloudy
☐ Rain	☐ Overcast	

***4. What is average temperature in Celsius do you enjoy your favorite activity the most (multiple answers allowed)?**

Done

Figure 2.11: Outdoor Activities Form

scriptive classification (provides the largest separation of results). The top 3 $wind[F_2]$ values are selected and the resultset is filtered on that ($V_1(F_2)$, $V_2(F_2)$, $V_3(F_2)$). If there are numeric values (e.g.: temperature) then the weighed average value is taken as the classifier. This, however, will only classify into two sets so they are only used in later levels of the tree.

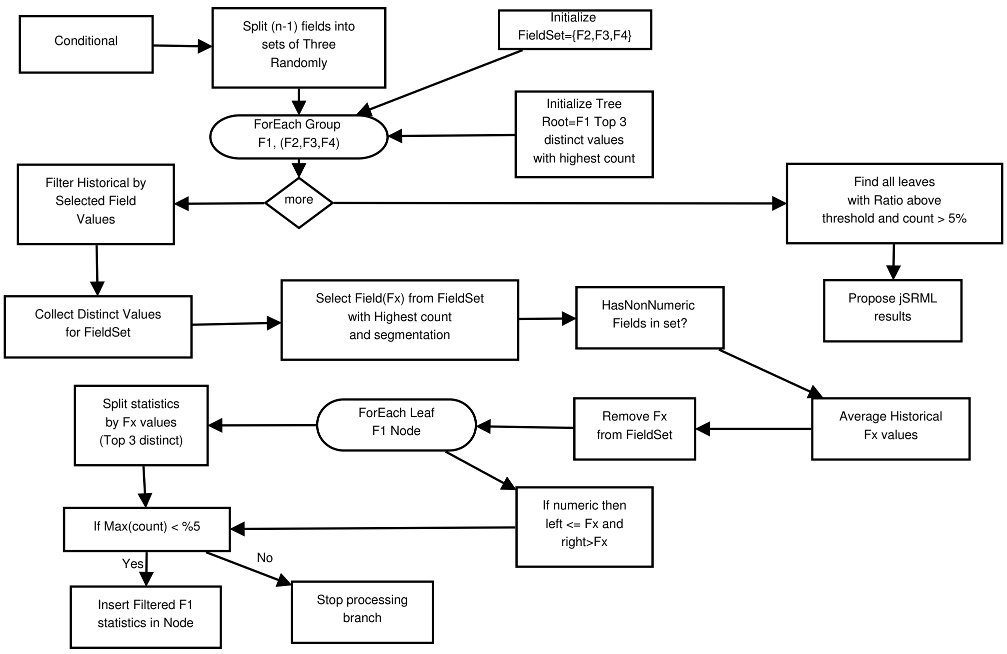

Conditional
Split (n-1) fields into sets of Three Randomly
Initialize FieldSet={F2,F3,F4}
ForEach Group F1, (F2,F3,F4)
Initialize Tree Root=F1 Top 3 distinct values with highest count
more
Filter Historical by Selected Field Values
Find all leaves with Ratio above threshold and count > 5%
Collect Distinct Values for FieldSet
Select Field(Fx) from FieldSet with Highest count and segmentation
HasNonNumeric Fields in set?
Propose jSRML results
Split statistics by Fx values (Top 3 distinct)
ForEach Leaf F1 Node
Remove Fx from FieldSet
Average Historical Fx values
If Max(count) < %5
If numeric then left <= Fx and right>Fx
Yes
No
Insert Filtered F1 statistics in Node
Stop processing branch

Figure 2.12: jpRelationship Conditional Method

The next tree level is created by applying a filter on the classifier results. In the example this means three nodes. The first node will list all entries where the *wind* (F_2) is *"Weak"*, the second sibling will list all entries where the *wind* is *"Strong"* and the third node on this level will list all items whose *wind* attribute is *"Breeze"*.

Based on the new level we recalculate the top three distinct values of the target (F_1) field for each selected value of $V_i(F_2)$. On a database level this basically means that we `select the top 3 distinct values for` F_1 `where value of` F_2 `IN (`$V_1(F_2), V_2(F_2), V_3(F_2)$`)`. The statistics are stored on the node level and are based on the filtered F_2 values.

The next step is to examine the remaining fields and create possible classifiers. The possible values of the fields are reduced by fixing field F_2 to the top three values. Based on the filtering weather (F_3) is chosen and the classifiers become: $C(F_3[F_2 = V_1(F_2)]), C(F_3[F_2 = V_2(F_2)])$ and $C(F_3[F_2 = V_3(F_2)])$ respectively. Taking the first classifier from the left the top three values it generates are *"Sunny"*, *"Rain"* and *"Snow"*. These values are used to filter all nodes on the level. On each level the distinct values of the F_1 are reduced based on the previous classifiers (e.g.: on this level only submission items that have the *weather* and *wind* values specified earlier are used to get the distinct values of the target F_1 field). The top three distinct values of the remaining two classifiers are also generated and added to the tree.

The last level has only one field left to use: *temperature*$[F_4]$. Since this is a numeric value, we take the weighed average of historical values (taking into consideration the field values chosen for F_2 and F_3). Taking the left node as an example (the remaining nodes operate similarly) this classifier becomes $C(F_4[F_3 = V_1(F_3[F_2 = V_1(F_2)])])$. The left branch will be where the value of F_4 is less then or equal to the classifier's single value of *10* (weighed average of submissions for this field after applying the previous classifiers) and the right branch contains statistics on field values larger than this value. Once the tree is built we look at the leaf values. We select whichever ones breach the ratio provided (in our example we set this to be 50 percent). If more than one leaf on the same node breaches this threshold we select the largest one. If they are identical then we select the first one from the left. To avoid too many false positives we also have a concept of coverage ratio. This is set by default to 5 percent. What this entails is that all result counts below 5 percent of the learning dataset will be ignored. In the example this comes to 100 elements, which means that any leaf result below 100 submit matches are ignored. Based on our example the following jSRML rules are proposed:

1. *"Activity"* is "Swimming" (64 percent of the cases) when the *"wind"* is "Weak" and the *"weather"* is "Sunny" with a "temperature above 10 degrees".

2. *"Activity"* is "Swimming" (59 percent of the cases) when the *"wind"* is "Weak" and the *"weather"* is "Rainy" with a "temperature above 16 degrees".

Once a proposed prediction is made it is then checked against the remaining 50 percent of historical data to confirm that the matching ratio is kept. If the ratio is above the target ratio a rule is created. It is important to note that the validation ratio of this learning algorithm is not 100%. This requires the owner of the domain or form to set the thresholds accordingly. It may misclassify valid inputs as false negatives if the threshold is not set correctly. The purpose of the learning here is to provide a direction of validation rules that can then be refined by the domain owner in contrast to the other learning plugins which can classify the inputs with higher confidence. With more plugins and stronger learning algorithms (e.g.: neural networks) the system can evolve to better classify harder relationships as well.

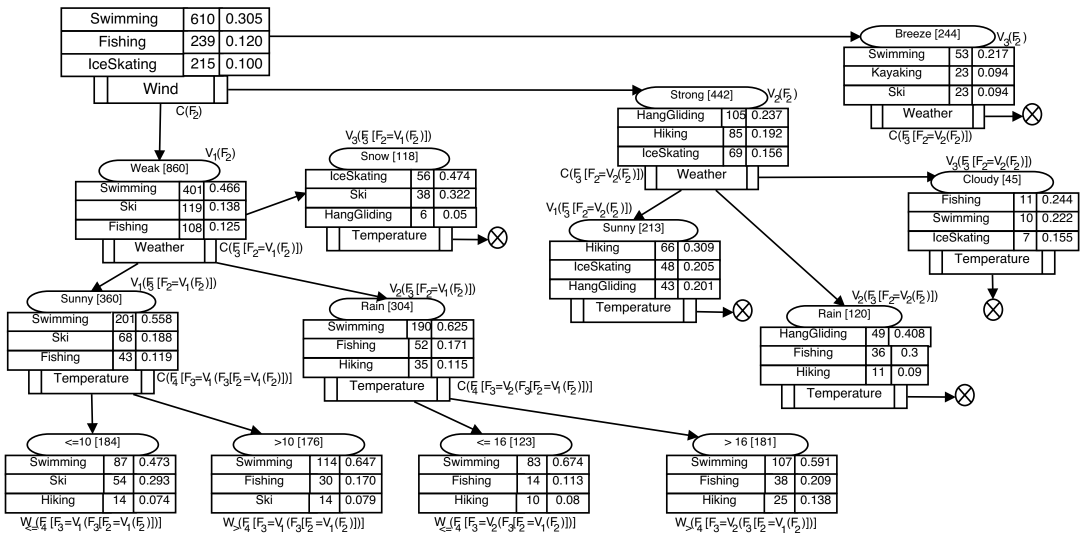


Figure 2.13: Sample tree in the Random Forest

jpPredefinedName

The *jpPredefinedName* plugin works on the assumption that many forms share field names and types. For example a field named *email* usually contains an email address, which has to be in a valid email format. The plugin contains a list of constant names and their corresponding formats. This list is maintained and extended by the administrator of the Servlet.

jpRegExp

The *jpRegExp* plugin is geared towards learning regular expression values for fields. The plugin starts out by analyzing the historical values for the (F_1) field in particular its separator sign occurrence (e.g.: $-, +, @, (,), [,]$). This is built up from the assumption that form fields using regular expressions are usually finite and pre-defined in format. This means that a field will usually follow the same pattern historically if it belongs to the same form domain (e.g.: ISBN number, phone number, Social Security Number...etc). A statistical table is built up of these to determine any potential separator position recurrence. This helps identify possible separators for the field value's regular expression. It also lowers the processing time of the algorithm as now only sets of fixed character lengths need to be checked. The plugin tries to match a separate regular expression for each section. We create a statistical tree, which analyzes each section one character at a time. If there are no separators the algorithm will treat the complete field values as a single section. This will, however, cause uneven length inputs to offset the regular expression result (e.g.: if most inputs were 5 characters long and some were longer then the output can be something like $[A-Za-z]\{5\}[1-9ace]*$). If the range could not be merged into an optimal one then it will contain the subranges per character location (e.g.: $[a-c][f-k][A-Z]\{3\}$). In both section separated and single-section modes each step will try to optimize the ranges into smaller expressions to conserve space. The statistical table contains ratios and statistics on all positions and it will split only when the ratio for the separator is 100%. The separator identification has two modes: *fixed position* and *floating*. In case of the *fixed position* mode the segments are fixed in length as well as the position of the separators. The *floating position* mode has a dynamic position nature (e.g.: the @ sign in emails) in which case the only certain information the plugin has is the number of sections in all inputs.

If the separators and sections are identified correctly then each section is analyzed one position at a time using the similar approach to the above. Depending on the mode (fixed vs floating) the sections lengths are either constant length or dynamic. This, however, will only affect the expression normalization. For each position the possible values are collected and converted into regular expression ranges. After the end of each section the ranges in the actual section are compacted into a potentially shorter representation. This compaction includes replacing a range of $[0-9]$ to $[\backslash d]$ and ranges like $[abcghi]$ to a range of $[a-cg-i]$. Multiple occurrence of similar ranges or types are also checked and introduced (e.g.: $[abc][abc][abc]$ is converted to $[a-c]\{3\}$). Using a sample input of (*ab0-8cz,bc1-akm,dtt-d5e,cog-102*) will generate an output of $[a-d][bcopt][01gt][-][18ad][05ck][2emz]$. In case of the *floating position* mode of the plugin we also utilize the $+$ and $*$ occurrence characters.

Once all segments have been "learned" the results are merged into one complete regular expression and matched against the remaining 50 percent of training data and if the ratio of the match is higher than the provided threshold then a rule is proposed. We have also experimented with reversing the logic of regular expression creation by starting out from the broadest ranges and tightening based on the results. This was also a good approach. However, it provided more false positives due to the generic nature. The system also has an experimental regular expression plugin based on block-wise grouping and alignment algorithm coupled with a simple looping automata based on the concepts outlined in [22]. This algorithm is simplified by the additional information acquired from the potential separators acquired in the first pre-check step. We thought it was worth mentioning it in this section as it can provide a more optimal solution than the statistical approach.

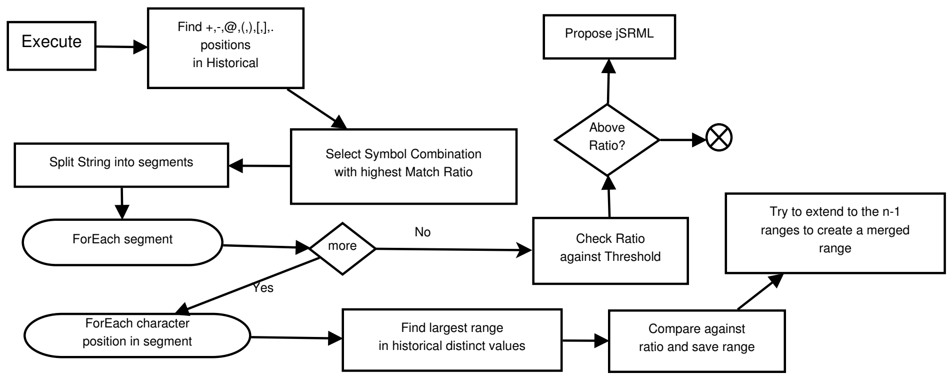

Execute
Find +,-,@,(,),[,],. positions in Historical
Propose jSRML
Above Ratio?
Split String into segments
Select Symbol Combination with highest Match Ratio
ForEach segment
more
No
Yes
Check Ratio against Threshold
Try to extend to the n-1 ranges to create a merged range
ForEach character position in segment
Find largest range in historical distinct values
Compare against ratio and save range

Figure 2.14: jpRegExp Plugin

2.4.2 Programatically evaluating the jSRML learning plugins

The `jSRMLTool` learning process uses a gradual approach to create the rules. The more positive inputs it receives the more effective the rules become. In order to provide a proper baseline it is advisable to feed in some positive form results. The results are summarized in *Table 2.3* where *T* denotes True classification (including positive and negative), *F+* means False positive and *F-* marks False negative with ES and PS marking Empty and Primed initial learning sets. The table includes the percentage results of the input classification (valid/invalid) for a specified plugin type. The learning is far from perfect, but with proper training it can aid the creation of validation rules. The simpler plugins like *jpFormat*, *jpLength*, *jpRange* are rather effective since they dynamically adjust their limits according to the inputs. The more complex plugins like the *jpRegExp* provided solid results. However, it is more resource intensive and take longer to provide the same success ratio. The *jpRelationship* plugin was excluded from the testing scenario as the random nature of the tests would not provide conclusive results on the efficiency of this plugin. We will demonstrate the real-life use of this plugin in a later section of this chapter.

During our tests we experimented with both empty and primed initial learning sets. In case of the empty learning set the number of false positives were considerably higher for the more complex plugins since they leveraged the distinct values and the learning set extensively. We did not run an evaluation on the *jpPredefinedName* plugin since that operates on a set of constant field names (e.g.: email, ip_address, isbn). The *jpCopyContent* plugin was also ignored for this evaluation since the results are based on equality between two fields and the random nature of the experiment offsets the actual findings of the plugin.

To test our plugins we used the following input sources:

- An English dictionary file containing *170,000* words. This is the source of all word subsets.
- A random list of *100,000* words from the dictionary to be used by the *jpLength* plugin.
- An email address list of *130,000* items built up from the dictionary with an added logic to generate valid/invalid emails. The ratio of valid/invalid emails was set randomly. The invalid emails were generated by adding known mistakes to words and symbols. The list also marks which are valid/invalid so that this information can be used in the validation evaluation. This is one of the sources of *jpRegExp*.
- A list of *50,000* phone numbers (matching US phone numbers: (CCC) NNN-MMMM) as the secondary input of *jpRegExp*.
- A list of *50,000* ISBN10 and ISBN13 random items as the tertiary input source for *jpRegExp*.
- A list of *50,000* IPV4 and IPV6 random items as an additional input for *jpRegExp*.
- A list of *250,000* regular expressions based on random expressions (variable in both format and length using $+, -, @, (,), [,]$. This will provide the additional learning set for *jpRegExp*.
- A list of *100,000* items randomly alternating between, *string*, *integer*, *double* and *date* for use with the *jpFormat* plugin.
- A list of *100,000* numbers between 1 and 1 billion. This list is used by the *jpRange* plugin.

Using the above sources we created *1,000* separate forms with random fields. Every form contained multiple fields (one to test each plugin). The *jpPredefinedName* and *jpCopyContent* plugins were ignored for the experiment.

The reason why we chose to run the results on multiple forms was to ensure that the form fields and their contents were more random. For every field of the forms the test randomly selected the "expected" results of the validation. This was used to identify how successful the learning was. Each form was processed with *30,000* inputs with both *Empty* and *Primed* Set approaches to allow a better picture of the plugin efficiencies. The main operation flow of each set is as follows:

- `Empty Learning Set` : For each form randomly select *15,000* values from the corresponding lists for each field and run the engine on them. It must be noted that for this mode the engine cannot determine what the "expected" values are since the inputs are not classified. The engine will try to generate rules for what the "expected" values are by choosing an initial *15,000* inputs. These inputs are analyzed and a set of proposed validation rules are created based on the best fit using the ratios. Following this another *15,000* values are selected from the learning set and are used to observe the validation results. This is not an ideal approach since we cannot ensure that the first batch of inputs were completely valid therefore it will yield more false positives.

  In case of the *jpRegExp* plugin the learning is not perfect due to the randomness of the selection. The remaining *15,000* values are run with each plugin and their classification is verified based on the expected versus the learned rules.

- `Primed Learning Set` : Using this approach the engine randomly selects *15,000* valid inputs for each field of each form based on the expected validation rules. As mentioned earlier every field has an "expected" validation requirement that is created during the form setup. The inputs might not fully overlap the expected target. However, will be considered valid based on its definition. An example for *jpRange* would be an expected range of *[100,000-200,000]*. The random values that fit into the range will be considered valid and will allow the plugin to create its own jSRML rule suggestion. Due to the random selection of valid elements a learned range for the previous criteria might be *[125,000-170,000]* (which is a subset of the original "expected" range). In case of the *jpFormat* plugin items with the expected format (string, integer, date, double) are selected from the list as the initial set. This will be the "valid" set of inputs. In case of *jpRegExp* one of eight predefined expression formats are selected as the "expected" validation rule and values that match this format (these formats are: email, ipv4, ipv6, phone, isbn10, isbn13, webaddress, phone). Afterwards a remaining *15,000* inputs are selected and executed using the rules. During the processing of the remaining inputs the engine

Plugin	T ES	F+ ES	F- ES	T PS	F+ PS	F- PS
jpFormat	64.36 %	25.11 %	10.53 %	94.58 %	3.23 %	2.19 %
jpLength	59.65 %	22.18 %	18.17 %	88.09 %	7.17 %	4.74 %
jpRange	26.78 %	44.06 %	29.16 %	66.31 %	25.41 %	8.28 %
jpRegExp	29.59 %	36.17 %	34.24 %	51.57 %	21.12 %	27.31 %

Table 2.3: Plugin comparison (ES=Empty Set, PS=Primed Set)

checks the learned rule results with the expected classification. Using these we are able to measure the efficiency of the learning.

The results of the forms are averaged and evaluated in *Table 2.3*. Based on the results it is visible that using *Primed* Sets yields the most effective results. From the plugins *jpFormat*, *jpLength* and *jpRange* are the most effective. The regular expression matching *jpRegExp* plugin does provide good results, however, the evolution of the format recognition should be tuned in the future. It should be noted that the current efficiency of the implemented plugins is not at 100%. This can lead to a valid question: how do we validate a form that is only *n%* effective? The short answer is that the acceptance threshold should be set so that the domain owner can accept the efficiency of the results. Even if the results are not 100% it still provides a direction to better tune the validation requirements. The more examples the engine can derive decisions and learn from the higher the efficiency becomes. In a human oriented approach the fields have more relationship and are chosen based on some expected behavior. One might argue if the whole learning validation rules has any relevance in the forms nowadays. We believe that the jSRML language provides a cleaner and more powerful way to define form validation rules. Allowing the option to intercept and potentially learn validation rules in a non-obtrusive way not only allows administrators with a powerful tool to create rule but can also be used to mine the inputs based on the submissions and potentially discover relationships and visitor decision patterns in the submitted form.

Due to the random nature of the previous experiment we felt it would be worthwhile to demonstrate an incremental approach as well for some of the plugins to better observe how the ratios change by gradually introducing more and more positive examples to the experiment. We chose a significantly smaller, more targeted learning set to better demonstrate how the plugins learn the results. This more constrained testbed yielded considerably better results.

For the *jpRegExp* we used a regular expression of $[1-4A-Za-z]\{5\}[-][1-6]\{5\}[-][a-k][A-P]\{8\}[-][1-9A-Za-z]\{8\}$ as the valid format (example valid inputs are: *1QrHk-56566-bPFI1ENNL-TLKir5Qk* and *h2bwM-61632-fELCGFJEM-631237Va*). This is a simpler regular expression then an *email* or *conditional isbn number* expression (matching both *isbn10* and *isbn13* formats), but still provides adequate ground to demonstrate how the plugin's efficiency evolves in proportion to the number of positive examples. The input examples for *jpRegExp* are 30 characters long and randomized on each character so we don't really need a set of tens of thousands of positive examples to learn them.

During the experiment the *jpRange* target range was also reduced to a smaller magnitude. The experiment sets a random range between *100* and *5,000*. The *jpLength* target was randomly selected with an upper limit of *400*, causing the experiment to terminate around *400-500* positive examples with a 100% ratio.

We provide *100* valid inputs for each plugin at the start of the test. We then take *50* positive and *50* negative for each plugin and observe how the rules classify the results and record the incorrect/missed classification counts. We are able to ensure that the the training examples are positive and negative since we select them according to our predefined criteria. We perform this over ten iterations. In each iteration we increase the positive examples by *100* and regenerate the validation rules. These rules are then run against *50* more positive and *50* more negative examples. After the tenth iteration we are priming the experiment with *1,000* positive examples and testing against *500* positive and *500* negative examples. This is a very controlled experiment but it is useful to demonstrate how the ratios converge in proportion to the number of training examples. The results of the experiment can be seen in *Table 2.4*. It can be seen in the figure that with proper and controlled positive inputs the plugins can provide near 100% ratios as well. In the next section we will demonstrate a real-life example where these results can be put into practice.

2.4.3 A Real-world example: Dentistry Treatment Inquiry Form

To evaluate the engine further we have hooked up the `jSRMLTool` servlet to an already functioning form to verify what the engine suggested for the validation rules. This was a more exhaustive test than the previous outdoor activity survey. During this test more plugins of the engine were verified as a whole. We chose [9], which is a site targeted at capturing leads for international clients who are inquiring about dental treatment in Hungary. The booking form contained several fields providing an ideal fit to test some of the plugins. Using the site's form we were tested: *jpFormat* (*Age*, *Country* field), *jpRange* (*Age* field), *jpRegExp* (*Phone* field), *jpPredefinedName* (*Email* field), *jpLength* (*First name*, *Last name*, *Phone*, *Treatment*, *How may we help* fields), *jpCopyContent*(*Confirm* email). The *Treatment* field was used in conjunction with the *Age*, *Gender* and *Country* fields to perform a *jpRelationship* conditional learning. The *Treatment* field had multiple non-conflicting rules generated using different plugins. The system found the range of the length used for the

Analyzed Plugin	Total Examples	Miss Count	Success Ratio
jpLength	100	17	83.00 %
jpLength	200	7	96.50 %
jpLength	300	2	99.33 %
jpLength	400	0	100.00 %
jpRange	100	72	28.00 %
jpRange	200	85	57.50 %
jpRange	300	97	67.67 %
jpRange	400	81	79.75 %
jpRange	500	63	87.40 %
jpRange	600	59	90.17 %
jpRange	700	45	93.57 %
jpRange	800	34	95.75 %
jpRange	900	28	96.88 %
jpRange	1,000	11	98.90 %
jpRegExp	100	98	2.00 %
jpRegExp	200	186	7.00 %
jpRegExp	300	198	34.00 %
jpRegExp	400	146	63.50 %
jpRegExp	500	90	82.00 %
jpRegExp	600	48	92.00 %
jpRegExp	700	22	96.85 %
jpRegExp	800	12	98.50 %
jpRegExp	900	6	99.33 %
jpRegExp	1,000	2	99.80 %

Table 2.4: Plugin Efficiency with gradual positive training examples

input and also used it for the conditional learning.

Our experiment used the site's historical data for lead submissions and ran *537* leads acquired form the site using *Selenium* [51] (scriptable automated tester framework) to emulate the form posting. The results were impressive, since it was able to provide effective validation rules for most fields. The *Phone* field had some weak rule recommendations (e.g.: $[0-4][1-5][0-9]+$). However, the ratios were not high enough due to entries with hyphens and extension numbers along with entries starting with + for international exit codes. Since the target ratio was not breached the plugin's rule recommendation was ignored. The output of the validation rules for each plugin can be seen in *Table 2.5*.

The experiment yielded in providing validation rules based on the results visible in *Table 2.5*. Most of the plugins yielded considerable adaptive results. If we would run the forms with more training examples then the ranges and results would improve as well. The experiment also showed that *"55% of clients requesting All-on-four dental as their treatment are Male, over the age of 50 and live in the UK."* (which is identical to the statement: In 55% of the cases "Treatment" is "All-on-four-dental" when the client is "Male" is from the "UK" has an age above "50"). The learning results also showed that *"61% of Abutment related requests come from Female clients from Ireland who are under 60"*. This provided a good demographic analysis of the visitors and

Field	**Validation Results**	**Plugin**
Age	[35, 70]	`jpRange`
Age	integer	`jpFormat`
Country	$5 < length < 12$	`jpLength`
First Name	$4 < length < 7$	`jpLength`
Last Name	$4 < length < 10$	`jpLength`
Email	email	`jpPredefinedName`
Confirm Email	Email match	`jpCopyContent`
Gender	$4 < length < 6$	`jpLength`
Phone	string	`jpFormat`
Phone	$7 < length < 14$	`jpLength`
Treatment	$4 < length < 37$	`jpLength`
Treatment	conditional	`jpRelationship`
How may we help?	$5 < length < 184$	`jpLength`

Table 2.5: Plugin results for Dentistry Contact form

helped the site adjust their marketing strategies accordingly. Even though data mining was not the focus of the experiment it did provide a direction for future study for the jSRML engine. The experiment proved the viability of such a solution in a real world scenario. The learning is not yet perfect, the rule engine and concept of allowing easier rule definitions substantially outweigh the performance and efficiency shortcomings (which can be tuned by introducing better learning plugins into the system).

2.5 Summary

In this chapter we introduced the jSRML metalanguage and engine. This is a major extension to the SRML language specification to enable it to be used in the form validation space. After showing the background technologies and demonstrating how form validation works, we provided the `jSRMLTool` engine. Our engine allows both *Client-side* and *Server-side* validation modes using the jSRML language. The extension permits non-obtrusive definition of form validation rules. The `jSRMLTool` engine can also correct the form values, making it extremely useful in situations when the submission can contain errors that can be corrected based on rules. We also showed ways to provide real-time validation. Our tool helps in the generation of jSRML rules using machine learning as well. The rules can change over time based on the form inputs. We believe jSRML is a valuable asset in the ever-growing pursuit for providing pristine and valid data acquired from web forms.

2.6 Related Work

In this section we will mention a few approaches to form validation. The first paper we would like to mention is [41]. This article proposes the use of an

XML based rule definition to show field validation. They create an XML file based on the database model itself on both the *Client-* and *Server-side* level. While it is a sound approach it still lacks the flexibility of the user overriding and defining custom conditions. In many cases, structural and type validity is not enough, context validity should also be considered. This means that even though a field's value is correct, it might have dependencies on other fields which are not visible on a database schema level. The approach lacks the option to provide custom hooks and does not provide provisions for data correction.

Another paper that we would like to mention is [21], which proposes that the validation of forms should be part of the model design and handled on the server-side. They leverage Spring MVC as part of their AC-MDSD (Architecture Centric Model Driven Software Development). Although a good approach, it requires the form validation to be coded as part if the datamodel on the server that will process the data. Our `jSRMLTool`'s server mode provides a more comprehensive set of features and does not force the developer to predefine their dataset prior to deploying the processing application.

The next approach we would like to mention is [5]. The author proposes a rule based field validation using JavaScript. The rules themselves are basic but support the comparison and aggregation of multiple field values. The validator engine itself does not have any hooks and does not allow the user to control what should happen if the validation fails. Our approach offers a solution to both and provides a way to dynamically correct the field data, making it a very powerful tool.

The authors of [4] propose an automatic Server-side validation approach for HTML forms. It collects the form elements and stores the validation elements inside a database and provides an interface for the administrators to go in and specify how to validate the given fields. Currently they do not offer too complex validation methods (since the approach is mainly focused on type and format oriented validation). It does not offer dependency or regular expression definitions for the field values. It resembles what we wish to achieve with the Servlet mode of our engine. Our library not only offers the forms to be validated using a centralized server, but also provides the definition of more complex validation rules.

The points discussed in [4] are aimed at server-side validation and are valid for most web forms. The article suggests that people will disable JavaScript, which would render client-side validation useless. This is a long and heated debate in the web community as most modern web pages utilize JavaScript and flash excessively. Disabling JavaScript support will not only render the validation useless but also hinder the usability of the page itself.

We should also mention the approach presented in [11]. This paper introduces a language called EEL (edit engine language) to provide a common way of describing field validation rules. This language was applied in the telecommunications area where several forms were being submitted. Although their approach was aimed at non-HTML forms, and was written purely in C++. It does have a solid syntax and could potentially be extended to be used in a modern web solution (after porting it to JavaScript or a server-side language).

The ideas raised in [24] demonstrate a .NET approach to rule based form validation. It also uses an in-line approach similar to jSRML. The rules can have conditions and it supports regular expressions as well. The rules are not as readable as jSRML and do not provide support for context related rules. For the rule definition it allows the reference of only one other field rather than providing a complete context-based approach. It does provide a solid solution for .NET based forms, which we believe is worth investigating in the future. Our metalanguage, is not limited to one technology stack or implementing language so creating a .NET library isn't hard to envision and implement.

Chapter 3

Validating Google Protocol Buffers

Thesis: Introduce a new metalanguage (ProtoML), which can validate and correct the messages of Google Protocol Buffers.

Introduction

After showing a way to validate XML documents using SRML 2.0 rules, we decided to experiment creating a language capable of validating Google Protocol Buffers (PB). This was a different direction compared to the text-based XML format since PB is binary-based, making its validation a challenge. Binary-based formats have considerably smaller payloads compared to text-based formats, thus more data can be transmitted in the same amount of packets. This advantage comes at a price of the format being boxed in and hard to extend. Most binary formats use a predefined set of fields (similarly to C structs). They often lack standardized validation schemas and usually have no way to describe the relationship between fields or their formats. The only real restriction they offer is specifying the type and name of the field and possibly a set of values they can have (e.g.:ENUMs). The validation task is usually up to the developer (it is rarely encapsulated within the language).

The reason why we chose Google Protocol Buffers (we will be using PB as the abbreviation from now on) was that it is very versatile and has support for various programming languages. Unfortunately the validation side of the messages in PB was not part of the language specification so it also suffers from the same drawback as most binary-based formats.

In this chapter we will introduce a new metalanguage called ProtoML, which provides a standardized way to describe constraints and validation rules for PB messages. The language took the advantages of SRML 1.0 [36] and SRML 2.0 (see Section 1.1.3 in Chapter 1) and revised the language syntax to make the rule definitions easier to define. We also introduce a tool

(`ProtoMLTool`), which generates Java wrapper classes that can validate the PB messages using the ProtoML rules defined. We will start by introducing the basic technologies used throughout to provide easier understanding of topics mentioned later. We will then demonstrate the ProtoML language through a simple example.

3.1 Preliminaries

There are a few terms and technologies that we should introduce first before proceeding to the general discussion topic of the chapter. These help ensure that the reader understands the fundamentals our language is based on.

3.1.1 Google Protocol Buffers

The most important topic to introduce is Google Protocol Buffers [26] (PB). This format provides a binary and lightweight way of serializing structured data. It allows developers to build up a `proto` file that describes the frame/-structure of the messages that will be sent over the wire. These `proto` files are then compiled into native language code (C++, Java, Python, C#...etc.) providing wrapper classes to read and write the message content. One of the advantages is that the language can handle multiple versions of the message frame (`proto` file) by treating new fields as optional. PB is very popular since it just works out of the box without any real in-depth knowledge requirement. *Figure 3.1* shows a message definition for a simple Person message. Every `Person` message has a *Name*, *Sex* (which can be “Female” or “Male”), *Title* (which can be “Mr”, “Mrs”, “Ms”, “Miss”, “Dr”), *Age*, *Income* and *Employed*. It is possible to define the field order and types using C-like definitions and also assign default values for each field. For *Title*, *Sex* and *Employed* ENUMs are used in the message definition allowing a preliminary restriction on the values of the field. Running the `protoc` compiler on the `proto` file will produce wrapper classes to serialize this type of message. We will be using Java in this chapter as the language of choice similar to the rest of the dissertation.

PB messages can be nested within one another allowing more complex message types. We can extend the previous example to serialize `HouseHold` messages that contain a *MemberCount*, *TotalIncome* and one or more nested Person message using the Member field (with the repeated keyword). The PB `proto` message format can be seen in *Figure 3.2*.

3.1.2 DOM model

We have introduced DOM [2] before in Section 1.1.1 of Chapter 1. The reason why we mention DOM in this chapter is that we will be using the DOM model

```
message Person {
  required string Name = 1;

  enum SexType {
    FEMALE = 0;
    MALE = 1;
  }

  enum TitleType {
    MS = 0;
    MRS = 1;
    MISS = 2;
    DR = 3;
    MR = 4;
  }

  enum EmployType {
   N = 0;
   Y = 1;
  }

  required SexType Sex = 2 [default = FEMALE];
  required TitleType Title = 3 [default = MS];
  required int32 Age = 4;
  required int32 Income = 5;
  required EmployType Employed = 6 [default = N];
}
```

Figure 3.1: Simple PB message to serialize Person messages

```
message HouseHold {
  required int32 MemberCount = 1 [default = 0];
  required int32 TotalIncome = 2 [default = 0];
  repeated Person Member = 3;
}
```

Figure 3.2: Nested PB message to serialize HouseHold messages

to represent the rules for ProtoML as well as converting the PB messages into DOM trees. Using a tree representation makes the rule processing and traversing easier. The DOM tree representation of *Figure 3.2* can be seen in *Figure 3.3*.

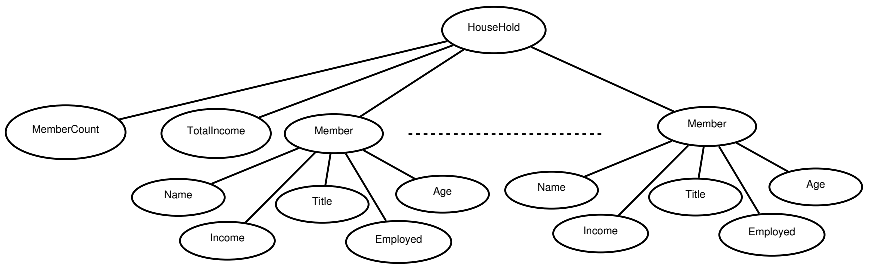


Figure 3.3: DOM representation of Figure 3.2

3.1.3 XPath

Another technology leveraged by ProtoML is XPath [18]. This concept has also been introduced in Section 1.1.2 of Chapter 1 so we will not be discussing it in detail. We use XPath to allow easier description of rules in the ProtoML validation space. Using the power of this query language it is possible to reference fields and siblings within the DOM tree. Since PB messages can be transformed into DOM trees it makes XPath a useful tool for field and value reference as well. XPath can be used to return lists of values, nodes and concrete values. If we take the DOM example of *Figure 3.3* we can query the *Name* of each *Member* in the `HouseHold` message using the XPath defined in *Figure 3.4*.

```
//HouseHold/Member/Name
```

Figure 3.4: XPath query for Names of Figure 3.3

3.2 Discussion

After covering all required background topics we will now introduce the ProtoML metalanguage. The language is XML based and uses functions to extend its descriptive capabilities. It allows the definition of validation and constraint rules for PB messages (represented by the `.proto` file). ProtoML can define multiple constraints on fields depending on their context and values. Using XPath it is able to reference other field values within the message. The language can also work with broader contexts by implementing message buffering on the library side (multiple messages can be placed into one context building up a larger DOM tree for the rules to operate on). The ProtoML rules can specify what action the implementing engine should take upon validation errors ("warn", "fail", "ignore"). There is also a validation mode flag that can notify the engine to potentially correct the value based on the expected rule value if it does not match. In this section we will cover the language basics using the `HouseHold` example described in the earlier sections. The ProtoML rule definitions are stored in `.pml` files that are XML documents with predefined schemas. The generic format of a ProtoML rule file can be seen in *Figure 3.5*.

The ProtoML document has a root `proto-rules` node that can have any number of rule child nodes. Each rule node has five parts: `path`, `mode`, `action`, `constraints` and `value`.

- `path`: This is the XPath of the field the rule is defining (the root here is the message root).

```
<proto-rules>
 <rule>
  <path>root XPATH</path>
  <mode>validate/correct</mode>
  <action>warn/fail/ignore</action>
  <constraints>
   <match>any/all</match>
    <constraint>constraint def</constraint>
...
  </constraints>
  <value>
<match>any/all</match>
<expr>value expression</expr>
  </value>
 </rule>
</proto-rules>
```

Figure 3.5: ProtoML rule format

- `mode`: This tells the implementing library what to do with validation failures. The possible values are *"validate"* and *"correct"*. If *"correct"* is used then the implementing library can attempt to correct the field value using the value definitions.

- `action`: This is a flag for the implementing library. It specifies what action to take if the validation fails on the given node. The values can be *"warn"*, *"fail"*, *"ignore"*.

- `constraints`: This is a block of constraints for the format of the field. This node has one `match` child and can have several `constraint` children. Each `constraint` is a boolean expression that is matched against the field value. If `match` is set to *"all"* then every constraint expression has to evaluate to true in order for the constraint to be fulfilled. If the `match` value is set to *"any"* then the constraint restriction will be satisfied if at least one `constraint` returns true.

- `value`: This block also has a `match` node similar to the `constraints`. In this case, however, the return values are not booleans but actual values that are matched against the field values. The reason why it is possible to have *"any"* as a `match` mode for expected values is that we can define context conditional expected values, which are not always matched.

In the constraints and value nodes it is possible to use multiple internal functions (evaluated from the inside out). The language also supports expression-based evaluation using the `eval()` function allowing the definition of complex arithmetical formulas without the need to daisy-chain the functions together. Referencing the current field value can be accomplished by using the `:self` constant. For the current field's XPath path reference the `:path` constant can be used. The implemented functions of the ProtoML language can be found in *Appendix C.1*.

To demonstrate the ProtoML language in action we will define a few validation rules for the example shown in *Figure 3.2*.

3.2.1 Validation rules for the HouseHold message

Validating the MemberCount field

The condition for this rule is: *"The MemberCount field should equal the number of Members (Person messages) in the HouseHold."* To describe this rule we will use the `count-children()` function in the values definition part of the ProtoML definition. *Figure 3.6* shows the rule snippet. The path of the rule is *`//HouseHold/MemberCount`*.

```
..
<value>
 <match>all</match>
 <expr>
  count-children("//HouseHold",
                 "Member")
  </expr>
</value>
..
```

Figure 3.6: ProtoML definition for MemberCount

Validating the TotalIncome field

The condition for this field is: *"The TotalIncome field is the sum of all Member Income fields"*. This validation rule uses the `for-all()` function to match all XPath elements with Income and use the add method to aggregate the values. *Figure 3.7* shows the rule snippet. The path of the rule is *`//HouseHold/TotalIncome`*.

```
..
<value>
 <match>all</match>
 <expr>
 for-all("//HouseHold/Member/Income",
         "add")
 </expr>
</value>
..
```

Figure 3.7: ProtoML definition for TotalIncome

3.2.2 Validation rules for Member embedded message

This Member field has a type of `Person`, but from a rule point of view the tree will use `Member` as the node name since that is the actual name of the field.

Validation rule for Title

This validation rule is as follows: *"Title can be 'Mrs', 'Ms', 'Miss' or 'Dr' if `Sex` is Female, otherwise it has to be 'Mr' or 'Dr"'.* In the `proto` message definition we defined *Title* as an ENUM of "Mrs","Ms","Miss","Dr","Mr". This forces a structural requirement on the field; however, it cannot determine conditional values. Using ProtoML we can define context sensitive possible values of a field. The rule snippet can be found in *Figure 3.8.* The validation rule first checks the value of the *Sex* field, which is on the same level as the Title field so we can use the `sibling()` method with the `:path` constant to get its XPath. We then compare the value against the word Female. If this evaluates to true then the second parameter of `if()` is evaluated and returned. This second parameter goes on to check if the value of the current field (which is acquired using the `:self` constant) is in the set of values defined. The path of the rule is *`//HouseHold/Member/Title`*. It is important to mention that constraints will return true or false depending on the structural requirements in contrast to the value nodes, which will return the expected value of the field.

```
..
<constraints>
 <match>all</match>
 <constraint>
if(eq(val(sibling(:path,"Sex")),"Female"),
 contains(:self,"Mrs","Ms","Miss","Dr"),
 contains(:self,"Mr","Dr"))
 </constraint>
</constraints>
..
```

Figure 3.8: ProtoML definition for Title

Validation rule for Employed

Validation rule: *"Employed has to be 'N' if Age is below 18".* This validation rule can be specified as either a constraint or a value match. Since the possible value of *Employed* is restricted to *"N"*. For this validation rule we acquire the value of the Age sibling and check if it is less than 18. If it is then the `if()` function will return *"N"* as the required field value. If it is not less than 18 then it will return with the actual field value (using the `:self` constant), which will always provide a successful field validation if this branch is hit. The path of the rule is *`//HouseHold/Member/Employed`*. The validation snippet can be seen in *Figure 3.9.*

```
..
<value>
 <match>all</match>
 <expr>
if(lt(val(sibling(:path,"Age")),18),"N",:self)
 </expr>
..
```

Figure 3.9: ProtoML definition for Employed based on Age

3.2.3 ProtoMLTool library

We have demonstrated the potential of the ProtoML language through a verbose example. In order to perform any actual validation an implementation of the language is required. We have created a draft implementation of the ProtoML language in Java called `ProtoMLTool`, which serves both as a library to execute ProtoML rules and create wrapper code based on the input `.proto` file and `.pml` language ruleset. The generated wrapper code no longer uses the `.pml` file and can be compiled along with the generated PB Java code. This is achieved by converting ProtoML rules into chained function calls and inserted this into a static class wrapper code to gain performance. The library also has a detached execution mode that can execute ProtoML rules on the messages (this mode, however, will require the `.pml` file during runtime as well). *Figure 3.10* shows the code wrapper generation flow of ProtoML.

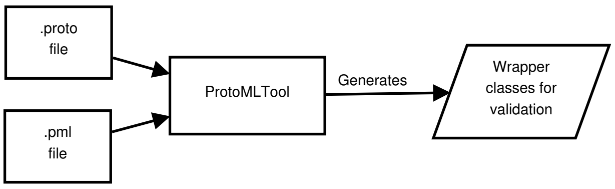


Figure 3.10: ProtoMLTool workflow

The validation flow can be seen in *Figure 3.12*. The class that receives the message needs to include a reference to the generated `PMLValidator` class besides the library dependency (in a form of an Ivy dependency). When the message is received a call to `PMLValidator.Validate` should be made with the current Message as the input parameter to perform the validation. This static method is generated from the `.proto` and `.pml` files so it will always use the proper generated Message classes. It will use the `protoc` generated wrapper classes internally to provide seamless integration with the target codebase. Since PB generates custom types and Enums these generated types are used by the library (for faster processing expressions and functions will invoke the `toString()` methods to match values of Enums). In order to ensure that the package names and types are correct it is important to specify the Java options in the `.proto` file as demonstrated in the snippet in *Figure 3.11*.

```
package examples;

option java_package = "com.protoml.examples";
option java_outer_classname = "HouseHoldProtos";
..
```

Figure 3.11: Proto Message Java Options

The generated `PMLValidator` class contains a *HashMap* of all fields that will need to be validated along with a *validator descriptor*. This *descriptor* contains a method name (which is executed using reflection) along with the flags for the given rule (validate/correct and the action to take upon validation failure). During the processing the incoming Message is converted into a DOM tree and the appropriate reflected method is called on it. This is done by looking up the field XPath in the map and checking if it has any descriptors assigned. Since all functions contained in the rules have their corresponding methods in the library the resulting code is similar to the rule definition.

If the mode was set to "correct" then the engine checks if the field is valid. If it is then the validation terminates successfully. If the result was not valid it will attempt to correct the value using the expected value. The library will retry three times to validate (if it fails again) after replacing the value until it finally terminates with a validation error. In case of the "validate" mode the process is straightforward. The `ProtoMLTool` was written in Java using the Spring Framework and uses *Exp4j* [8] to evaluate the expressions. The DOM manipulation and access is handled by the *JDOM* [29] library.

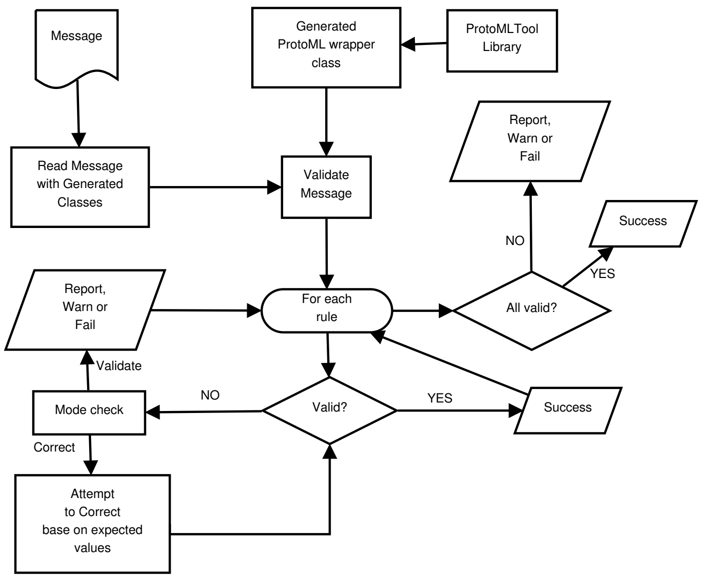


Figure 3.12: ProtoMLTool Validation workflow

3.3 Summary

We have introduced a powerful new metalanguage called ProtoML, which allows the definition of validation rules for Google Protocol Buffer Messages. This language fits nicely into the validation space and contributes to the evolution of SRML. The language provides a clear and concise way of specifying the required value and format of the message fields with a potential to correct invalid field values. We have shown how a draft implementation of a tool that uses ProtoML operates and how it can be used to validate the messages.

3.4 Related Work

Some of the projects mentioned in this section are not fully related to Google Protocol Buffers, but are aimed at providing a descriptive structural validation for binary formats. Since PB is a binary format, most can be applied to it with some modification.

The first one to note is the Piqi [38] project. It was created to provide a framework for cross-platform and language serialization. It has a human readable definition language that can describe the structure of documents. Its tool-set enables the conversion between XML, JSON, Binary, and Protocol Buffers. Unfortunately it does not have real value validation capabilities, only structural verification.

A very common format is Apache Thrift [6]. It allows the authors to define their data types and services as part of a `.thrift` file. This is very similar to what PB does. It also provides a validation function to provide structural validation.

Another project we would like to mention is the ASN.1 [37] ISO standard. It allows Type References, Identifiers and Values. Its language specification also allows complex data types to be defined; however, custom value validation would require a separate implementation.

The author in [57] introduces a way to use Attribute Grammars as semantics in binary formats. A part of this approach is similar to ProtoML since the XML language itself can be extended with semantics similar to how SRML 1.0 leveraged Attribute Grammars. We use a DOM tree (built from the XML and proto message) to describe the binary format. It does not go on to the actual rule context and value definition side, but starts out on a similar path as ProtoML, therefore we thought it was worth mentioning here.

Chapter 4

Validating Web Services

Thesis: Combine the previous metalanguages (SRML 2.0, jSRML, ProtoML) into SRML 3.0 and provide a way to validate Web Services.

Introduction

The final the dissertation covers is the space of web services. Based on the experience and knowledge gained during the development of jSRML and ProtoML we decided to unify their positive traits into the SRML language. This led to the new 3.0 extension of the SRML language. In this chapter we will show how the two metalanguages helped in making the 3.0 version of the SRML language easier to use, more descriptive and contain less overhead than its predecessors. We will demonstrate how the new version along with a new implementation of the rule engine can be leveraged to validate web services. The reason why web services are important is that there has been a paradigm shift in software architectures in a sense that instead of re-writing services over and over the trend now is to re-use and share functionality to reduce cost. Web services bridge the gap between systems distributed over multiple geographic regions, providing an easy way to communicate in a platform independent manner. The advantage of using web services is that the client does not need to know how the data is created or where it comes from. The client's system can implement its own business logic with the consumed data or can connect to other web services as well. Web services are not limited to one programming language, making them ideal for cross-platform communication and service sharing.

The original groundwork was laid by RPC [10], which allowed systems to connect to another machine, invoke remote procedures and return data. Nowadays we live in an age of Service Oriented Architectures where solutions are built by consuming and accessing external services, as part of the business logic. With the emergence of web services considerable changes occurred in IT architectures. Instead of single-use applications the trend has become to write reusable services, that may be consumed by third party systems.

The evolution of the Internet also brought in several commercial and free web services. An example is the National Digital Forecast Database service [47], which allows the retrieval of weather information based on the supplied zip or city. Another example of web services is that of on-line banking clients that provide mobile and web access. With such a high demand on services, validity is an important aspect. Publicly exposed services are under constant attacks [14] [27] (e.g.: injection attacks, invalid data submission, Denial of Service). This is one of the main reasons why validation and data sanitization plays a very important role for both the client and provider side. The service provider needs to ensure that the requested data is in a valid format and will not compromise their system, whereas the priorities of the client involves validating the format and content of the resulting data along with integrating it into their existing infrastructure. Currently the only really viable way to validate either side involves changes to the systems. While this is a great solution, it requires extensive resources to introduce the validation logic into an existing system. If the requirements or the format of the data change over time (e.g.: a new bank account format is introduced) then the backing system needs to be updated and possibly recompiled. The same situation exists for the client side since the consumption data might need to be filtered for a subset.

We will be using WSDL [12] as the interface language for web service description since it is an XML-based format, making SRML a good choice for its validation. We have enhanced the SRML language to version 3.0, which is also being presented as part of the chapter. We have created a rule-based web service validator tool called wsSRML that leverages our SRML language. Using SRML rules we can define the expected format (structure and value) of the services (input and output). The tool is able to run in multiple modes: *native* and *proxy*.

The *native* mode uses the set of SRML rules and augments web service calls with a wrapper class, allowing the validation to occur natively within the code itself. The second mode of operation acts as a proxy system that validates the incoming service requests and relays the potentially corrected version to the provider and vice versa. This permits a transparent validation flow without the need to change the client or provider side. There are several web service validation approaches available. However, most of them are not flexible and are harder to use. We do not aim to provide a replacement for these but rather demonstrate an easy-to-use, highly extensible rule-based approach, which also allows the provision to correct validation errors. This creates an all-in-one solution and enables the users to concentrate on the logic itself rather than how to describe their business rules.

4.1 Background

In this section we will cover two main concepts that will be used throughout the chapter: Web Services and the new SRML 3.0 format. First we will discuss XML briefly since both the rules and the WSDL definitions are based on this metalanguage. The second section will summarize the main aspects of web services in order to provide a generic groundwork. The third topic will cover the extension to the SRML language that will enable the description of the validation rules.

4.1.1 XML

Since we have already introduced XML in Section 1.1.1 we will not be detailing it again. We will, however, be using a Foreign Exchange Trade example in the later sections, which uses XML documents similar to the one shown in *Figure 4.1* as the input and performs operations based on them. It makes sense to mention the example in this section for clarity.

```
<TradeRequest>
   <client_id>AF0103991485</client_id>
   <value_date>2013-11-28</value_date>
   <timestamp>1385648969</timestamp>
   <pair>EUR/USD<pair>
   <bid>1.35895</bid>
   <ask>1.35928</bid>
   <qty>100</qty>
   <action>BUY</action>
   <ip_address>192.168.39.102</ip_address>
</TradeRequest>
```

Figure 4.1: Foreign Exchange Trade transaction in XML

4.1.2 Web Services

Web services provide a standardized way for two machines to communicate over the World Wide Web. There are multiple formats that can be used with web services. Most of the formats are REST-compliant [50] meaning they perform a set of stateless operations that can be repeated numerous times. We will be using the XML-based web service format since it uses XML messages for communication that conform to the SOAP [13] standard. In many cases there is also a machine-readable description of services which is defined using WSDL [12]. This is not a requirement for the endpoints; however, they are needed if automatic code generation is to be used. We will describe this in more detail later since our `wsSRML` is based on the presence of a WSDL description. The high level overview of the web services architecture can be seen in *Figure 4.2*.

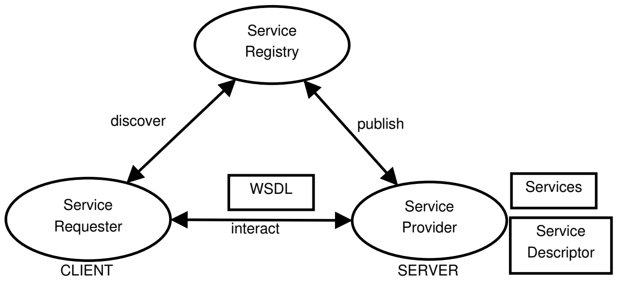


Figure 4.2: Web Service Architecture

The SOAP protocol

The SOAP [13] protocol provides an extensible framework for wrapping XML messages into envelopes. An envelope has a header and a body. The SOAP header is an optional element in the message that can be used to pass in any application-specific data along with the message. The SOAP body is required since it is the payload of the message itself. SOAP messages are relayed using standard network protocols like HTTP, FTP...etc. The data sent over the wire represents the information needed to invoke a service or to marshal the results of the output. A SOAP message defines both the target method's name and set of parameters along with the namespace definitions. *Figure 4.3* shows a SOAP message delivering the XML payload of *Figure 4.1*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <soapenv:Body>
      <performTrade xmlns="http://service.trades.example.com">
      <inputParam>
         <client_id>AF0103991485</client_id>
         <value_date>2013-11-28</value_date>
         <timestamp>1385648969</timestamp>
         <pair>EUR/USD</pair>
         <bid>1.35895</bid>
         <ask>1.35928</bid>
         <qty>100</qty>
         <action>BUY</action>
         <ip_address>192.168.39.102</ip_address>
      </inputParam>
    </performTrade>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 4.3: Example SOAP message of Figure 4.1

WSDL

The Web Service Definition Language [12] is used to describe the messages that are exchanged between the client and provider sides. This definition is protocol agnostic, meaning that it does not care about how the message is relayed. This language describes the available services, their input parameters with types and the results allowing systems to generate code from the WSDL description, making it easier to communicate between the systems. These definitions can be mapped by any platform, language or messaging system. *Figure 4.4* shows a simple WSDL definition of a Foreign Exchange trading service. The service has a `performTrade` method, which takes a single parameter called *inputParam* of type `TradeRequest` (*Figure 4.1* shows an example input) and returns a `TradeResponse`.

In Java the `wsdl2java` tool takes a WSDL and can create classes and an interface that provide the stubs to communicate with the web service. The interface is used as a contract that wraps the message into the required format. The results from the service call are also marshalled into the generated JAXB classes. *Figure 4.5* shows the service call flow using the generated classes. We will be using this feature as part of the *"native compiled"* validation mode of our validator engine.

4.1.3 SRML 3.0

In the previous chapters we have introduced SRML 2.0 (see Section 1.1.3), ProtoML (Chapter 3) and jSRML (Chapter 2). We have decided to create a new version of SRML using the knowledge gained from the additional metalanguages and provide an easier syntax for SRML. In this section we will demonstrate the difference between SRML 2.0 and the new 3.0 version and provide some validation example to better demonstrate the new syntax. This is important to better understand how we use the new version of the language to validate the web services. *Figure 4.7* shows a simple SRML 2.0 validation rule that defines a simple rule for *Figure 4.6*. The rule has a restriction that only considers the document valid if the name of the person is longer than 5 characters. The SRML 2.0 format was aimed at returning the expected value of the actual node rather than separating the expected value from its constraints. This is the reason why the example returns *"SRML_INVALID_ENTRY"* in the else branch of the `if-expr` node. If the length is above 5 characters then the actual node value is returned, making the validation succeed. If the length is less than 5 then it will return *"SRML_INVALID_ENTRY"* as the result, which will not match the actual value causing the validation to fail. This was efficient for smaller documents, which fit into memory but made the definition and processing of more complex rules difficult. In a larger document context the processing of these rules took more time so we had to come up with a new format for performance reasons.

To provide more powerful (faster to process) and easier definition of the rules for larger documents we have extended the language further to SRML 3.0

```
<wsdl:types>
  <schema elementFormDefault="qualified"
   targetNamespace="http://service.trades.example.com"
    xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="performTrade">
    <complexType>
     <sequence>
      <element name="inputParam" type="impl:TradeRequest"/>
     </sequence>
    </complexType>
   </element>
   <complexType name="TradeRequest">
    <sequence>
     <element name="client_id" nillable="true" type="xsd:string"/>
     <element name="value_date" nillable="true" type="xsd:string"/>
     <element name="timestamp" nillable="true" type="xsd:long"/>
     <element name="pair" nillable="true" type="xsd:string"/>
     <element name="bid" type="xsd:double"/>
     <element name="ask" type="xsd:double"/>
     <element name="qty" type="xsd:int"/>
     <element name="action" nillable="true" type="xsd:string"/>
     <element name="ip_address" nillable="true" type="xsd:string"/>
    </sequence>
   </complexType>
   <element name="performTradeResponse">
    <complexType>
     <sequence>
      <element name="performTradeReturn" type="impl:TradeResponse"/>
     </sequence>
    </complexType>
   </element>
   <complexType name="TradeResponse">
    <sequence>
     <element name="trade_id" nillable="true" type="xsd:string"/>
     <element name="client_id" nillable="true" type="xsd:string"/>
     <element name="value_date" nillable="true" type="xsd:string"/>
     <element name="timestamp" nillable="true" type="xsd:long"/>
     <element name="pair" nillable="true" type="xsd:string"/>
     <element name="actual_bid" type="xsd:double"/>
     <element name="actual_ask" type="xsd:double"/>
     <element name="qty" type="xsd:int"/>
     <element name="total" type="xsd:double"/>
     <element name="action" nillable="true" type="xsd:string"/>
     <element name="ip_address" nillable="true" type="xsd:string"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>
```

Figure 4.4: WSDL for the Foreign Exchange Trade service

along with a new validation engine implementation (the full XSD can be found in Appendix D.1). The rule definition was considerably simplified using the experience gained in the ProtoML and jSRML languages. *Figure 4.8* shows the new 3.0 rule definition format of the same name length restriction outlined in *Figure 4.7*.

The new format separates the expected values (under the values node) from the value/format constraints (conditions element). The other considerable

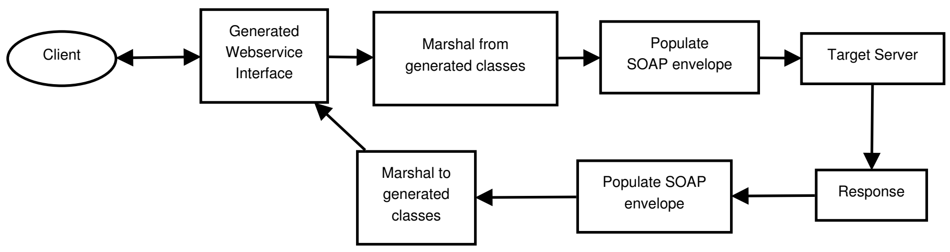

Client
Generated Webservice Interface
Marshal from generated classes
Populate SOAP envelope
Target Server
Response
Populate SOAP envelope
Marshal to generated classes

Figure 4.5: Service Invocation using marshalled classes

```
<person>
  <name>Fred Flintstone</name>
  <phone>1-800-667-1234</phone>
  <email>fred@bedrock.com</email>
</person>
```

Figure 4.6: Person representation in XML.

```
<rules-for root="person">
 <rule-def name="name" mode="validate" match="all">
  <rule-instance>
   <validation-error>Invalid name, minimum length is 5!</validation-error>
    <if-expr>
     <expr>
       <binary-op op="gt">
     <expr>
           <string-length> <instance-value /></string-length>
     </expr>
     <expr><data>5</data></expr>
       </binary-op>
     </expr>
     <expr><data><instance-value /></data></expr>
     <expr><data>INVALID_ENTRY</data></expr>
    </if-expr>
  </rule-instance>
 </rule-def>
</rules-for>
```

Figure 4.7: SRML 2.0 ruleset for Figure 4.6

difference is that the engine received a new validation core, which now works on a function chaining approach (similar to ProtoML). This makes the rule definition easier since we can wrap multiple functions into one large condition (e.g.: *gt()* and *length()* in the example). The full list of functions supported can be found in Appendix D.2.

The extension also introduces the `validation_root` and `validation_record` elements, allowing the ruleset to denote what the document's root is and the records are, similar to the how the root element is defined in XSD. This can be leveraged by documents containing multiple records of the same structure. In case of the `validation_record` we can also define the ID of the record that operations can use to refer to in a simpler approach. The record's ID is described using its XPath to allow more flexibility. This means that it can be

```
<rules-for root="person" >
   <rule-def name="name" mode="validate" match="any">
     <rule-instance>
       <validation-error>Invalid name, minimum length is 5!
       </validation-error>
       <conditions >
         <condition>
           <entry>gt(length(:instance-value),5)</entry>
         </condition>
       </conditions>
       <values />
     </rule-instance>
   </rule-def>
</rules-for>
```

Figure 4.8: SRML 3.0 ruleset for Figure 4.6

on the top level of the record or located somewhere in the record's DOM tree. The example below shows the definition of the new record elements:

```
<validation-doc-root name="rootname" />
<validation-record name="recordelementname" id="/xpath/to/ID" />
```

The key syntactic and usability improvements that SRML 3.0 has over the previous version can be summarized the following way:

- Uses a new function approach syntax introduced to allow functions to be daisy chained together and evaluated easier.
- The `conditions` tag allows listing the conditions that the inspected element has to conform to. The match parameter can be all or any depending on whether or not the requirement is to have all condition expressions met or at least one.
- The `values` tag enables context-specific value definitions to be described. These expressions are evaluated top-down. The first one to match the context child expression conditions will be used as the expected value. This is used to correct the value easier.
- Using the `validation-record` allows the definition of the XML record elements along with their primary IDs.
- With the help of `validation-doc-root` it is possible to define of the XML document root element.

Continuing on with the *TradeRequest* example we can define the SRML definition for the *pair* field of *Figure 4.1* as shown in *Figure 4.9*. The example rule definition uses the `in()` function, which returns *true* if the first parameter is in the set defined in the second parameter. The `set()` method creates a set from a list of values. For simplicity we limit the value-pairs to four currency pairs. The `:instance-value` constant refers to the currently validated value (in the example's case the *pair* field value).

```
<rules-for root="TradeRequest">
 <rule-def name="pair">
    <rule-instance>
       <validation-error>Invalid Pair specified</validation-error>
       <conditions>
          <condition>
             <entry>in(:instance-value,
                    set("USD/CHF",
                        "EUR/USD",
                        "USD/JPY",
                        "GBP/USD"))</entry>
             <params />
          </condition>
       </conditions>
       <values/>
    </rule-instance>
 </rule-def>
</rules-for>
```

Figure 4.9: SRML rules for a Foreign Exchange Trade transaction

4.2 Validating services

We have created a tool called `wsSRML`, which builds on top of the SRML 3.0 language and allows the validation of web services. The new tool provides two ways to validate the request and response of web services. We will be using Java as the main language in this chapter. However, the concepts can easily be applied to other languages as well (e.g.: C#). Our tool provides two ways to validate the request and response of web services. The first way is to perform the validation on the client side by placing the validation process into the generated code. The second is to intercept incoming and outgoing communication to and from the target web service and apply the validation logic inside a proxy service. We will demonstrate how each method works along with their advantages and disadvantages.

We will be using the Foreign Exchange trade example described earlier. The service has a single method called `performTrade`. The method has a single input parameter with a type of `TradeRequest` as described in *Table 4.1.* The result type of the method is `TradeResponse` which is detailed in *Table 4.2.* In the `TradeReponse`'s total field we demonstrate the ways SRML allows for arithmetic expressions to be evaluated: the `eval()` function uses the expression engine for evaluation and the `mul(val(sibling()))` uses an XPath value extraction approach. Both the input and output parameters have restriction requirements mentioned in their corresponding figures. The `TradeResponse`.total field defines an expected value restriction. The method signature in Java can be written as follows: `TradeResponse performTrade(TradeRequest inputParam);`

During the validation phase both *Request* and *Response* parameters are accessible since the function of the wrapper class is making the actual service call. When using the *native* mode the rules are converted to Java code and

Field	Restriction	SRML snippet
client_id	Starts with "AF" and is 12 chars long	and(starts-with(:instance-value,"AF"), eq(length(:instance-value),12))
value_date	Has to be a valid day and at least today	and(has_format(:instance-value,"shortdate"), gte(timestamp(:instance-value,"Ymd"), timestamp(current_date("Ymd"),"Ymd")))
timestamp	Its an integer that is above 1356998400 (2013-01-01)	and(type-check(:instance-value,"int"), gte(:instance-value, 1356998400))
pair	Currency pair. Has to be the following: (USD/CHF, EUR/USD, USD/JPY, GBP/USD)	in(:instance-value,set("USD/CHF", "EUR/USD", "USD/JPY", "GBP/USD"))
bid	Is a float value	type-check(:instance-value,"float")
ask	Is a float value	type-check(:instance-value,"float")
qty	Is an integer and greater than 0	and(type-check(:instance-value,"int"), gte(:instance-value,0))
action	Can be "BUY" or "SELL"	in(:instance-value,set("BUY","SELL"))
ip_address	Valid IP address format	has_format(:instance-value,"ipv4")

Table 4.1: Forex TradeRequest type definition and restrictions

the input/output variables are still in the method's scope. In case of the *proxy* mode the parameters are pushed to a single DOM tree allowing XPath operations on them. This means that the *Response* rules can refer to the *Request* parameter values since they are still available when the server returns the response value. This provides even more powerful validation rules since there are cases when conditions can be defined on the response based on what the request was.

Field	Restriction	SRML snippet
trade_id	Starts with "TRD"	starts-with(:instance-value,"TRD")
client_id	Equals inputParam.client_id	eq(:instance-value, val("inputParam/client_id"))
date	Equals inputParam.value_date	eq(:instance-value, val("inputParam/value_date"))
timestamp	Larger than inputParam.timestamp	gte(:instance-value, val("inputParam/timestamp))
pair	Equals to inputParam.pair	eq(:instance-value, val("inputParam/pair"))
qty	Equals inputParam.qty	eq(:instance-value, val("inputParam/qty"))
action	Equals inputParam.action	eq(:instance-value, val("inputParam/qty"))
actual_bid	Is a float value	type-check(:instance-value,"float")
actual_ask	Is a float value	type-check(:instance-value,"float")
ip_address	Equals inputParam.ip_address	q(:instance-value, val("inputParam/ip_address"))
trade_total	If the action was SELL then value = qty * actual_bid otherwise value = qty * actual_ask	if-else(eq(val(sibling(:instance-path,'action')),"SELL"), mul(val(sibling(:instance-path,'actual_bid')), val(sibling(:instance-path,'qty'))), eval("../*actual_ask*∗../qty"))

Table 4.2: Forex TradeResponse type definition and restrictions

Most validators only concentrate on the request side. One might argue why the server side should handle the response of the service. From a security point there are cases when man-in-the-middle attacks can intercept and shape/change the traffic to exploit the system for their own advantage. The other situation when response validation is needed is when the service itself is an aggregation of multiple services, which may not all, be valid. In these situations providing response-based rules ensure a higher level of validity. The response may be structurally valid, but content validation can only be done with more advanced techniques. SRML 3.0 allows an easy way to define the expected values as well. In the Foreign Exchange example the response needs to be validated to ensure that the purchase of currency pairs was performed according to the request. If the response was not validated then any downstream system utilizing the output of the response and using it further for in their own business logic would need further validation. By allowing a rule-based approach both request and response can be validated in the same ruleset. Our system also allows the errors to be corrected using the validation rules, making it more than a simple validation engine.

In order to define the SRML rules for the Forex trade service we will need to use some of the internal functions of SRML 3.0 (Appendix D.2 contains the full list of available functions). The `rules-for` element defines the method name the rules are describing to and the *name* parameter in the `rule-def` specifies which parameter name that the rule is pertaining to. If the name is @*result* then the rule is referring to the result value of the method. Normally in SRML the @ sign refers to an attribute reference. In the `wsSRML` space we use it to denote the result value. The rule's `condition` will provide the constraints on the format of the parameters or result. The `value` section will define the expected values of the given node. Using these expected values the values of the input/output parameters can be corrected. Since the previous figures only showed the actual `entry` contents we will show a full SRML snippet to demonstrate what a full field rule definition looks like. The SRML definition snippet that defines the rules for the result's `trade_id` can be seen in *Figure 4.10*.

```
<rules-for root="performTrade">
 <rule-def name="@result/trade_id">
    <rule-instance>
      <validation-error>TradeID in response has an invalid format!</validation-error>
      <conditions>
         <condition>
            <entry>starts-with(:instance-value,"TRD")</entry>
            <params />
         </condition>
      </conditions>
      <values/>
   </rule-instance>
 </rule-def>
</rules-for>
```

Figure 4.10: SRML rules for validating the `trade_id` field in the response.

In the SRML ruleset for the Foreign Exchange Trade example we used `complexType` definitions, meaning that both the input and output parameters were not primitive types. To specify which field the rule is referring to, we specify its full path starting with the parameter name as the root. This allows the rules to refer to more complex structures and provides a granular validation schema for services. In case of primitive types the rule definitions are straightforward as the `rule-def` name parameter will be a single string specifying the name of the parameter being targeted.

4.2.1 Native validation mode

The first mode of the `wsSRML` engine we will demonstrate is the native validation mode. Once the stub generation is completed using `wsdl2java` our tool will parse the SRML ruleset and augment the generated code. The SRML rules are analyzed and the wrapper class is generated on top of the interface. *Figure 4.11* demonstrates how the native mode generates the wrapper class.

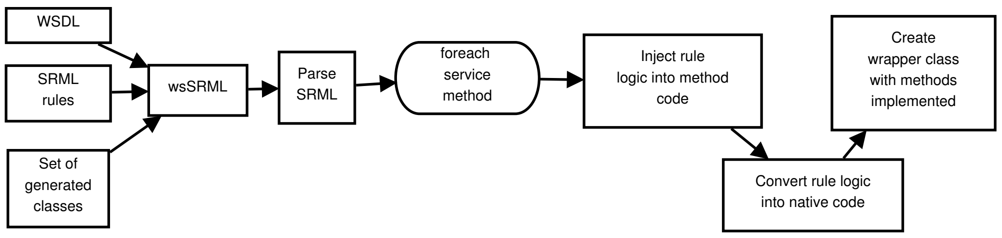

WSDL
SRML rules
Set of generated classes
wsSRML
Parse SRML
foreach service method
Inject rule logic into method code
Convert rule logic into native code
Create wrapper class with methods implemented

Figure 4.11: Native validation class generation

The validation wrapper class contains the business logic that is translated from the SRML file. The logic is implemented using reflection. Since the rules may contain functions that can be chained together the validator library needs to be included into the project that wishes to leverage the validation. *Figure 4.12* shows what the validation flow looks like in case of this mode.

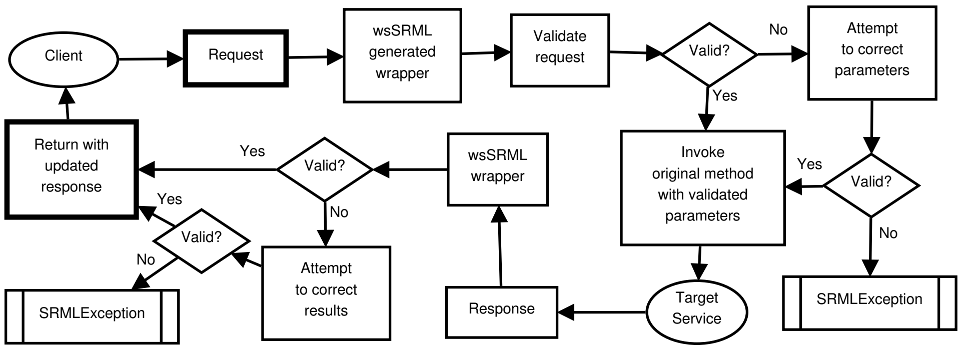


Figure 4.12: Native validation flow using wsSRML

Taking the previous Foreign Exchange service example we use `wsdl2java` to generate the stubs and run `wsSRML` on the ruleset and the resulting classes. The engine then uses the rules and creates a new class (e.g.: `TradeServiceSRML`) and injects a reference to the previous interface generated from the `wsdl2java` output. This new augmented wrapper class will have all the methods of the original interface; however, the method content will be populated. The contents contain the Java equivalent of the rules ending with the actual call to the original interface's corresponding method. This will ensure that the flow remains the same as the original approach, but adds the validation aspect to the methods. Method reflection was a better candidate here since each rule function can easily be converted into Java code and can be chained together and no external layers are needed to process the rules. The wrapper class provides a way to correct the input and output parameters. If the constraint fails then an `SRMLException` is thrown. When the business logic needs to be changes then the tool can regenerate the wrapper class using the new SRML rule file. This is similar to how we solved the ProtoML rule native code generation in Section 3.2.3 in Chapter 3.

- **Advantages**: Fast, native compiled validation that can be used in any project that need web service validation.
- **Disadvantages**: Since the rules are compiled into the code the business logic cannot be changed on the fly; it has to be recompiled, which may be hard in a production setting.

4.2.2 Proxy-based validation mode

The second mode `wsSRML` supports is the *proxy-based* mode. This mode is useful in situations when the client and server cannot be updated with the validation code. Normally it is very difficult to alter legacy systems with new business logic or validation rules. Using the *proxy* approach we introduce a proxy servlet between the client and server. The clients request the services from the servlet, which then passes the requests on to the target server. During the process the proxy will use the provided SRML rules to validate and potentially correct the incoming and outgoing requests. *Figure 4.13* shows the validation flow in case of the proxy based validation.

There are three operation modes the proxy servlet can run in: *real-time rule loading* mode, *compiled rule plugin* mode and *SOAP intercept* mode. The proxy will perform the validation in the request phase. If the validation fails then an exception is thrown and the error is returned in the response. In case the server returns data that is invalid the engine will try to correct the results using the rules. This is not as fast as a native version. However, does provide more flexibility in replacing the validation rules without any considerable downtime.

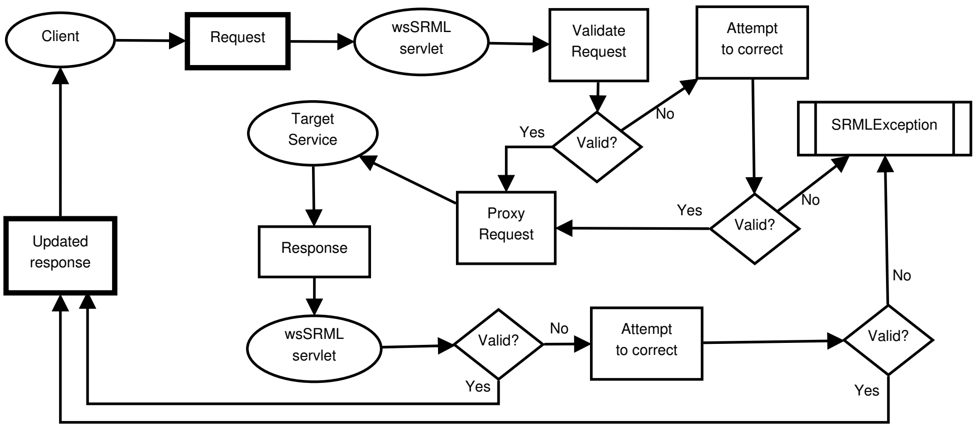


Figure 4.13: Proxy based validation flow

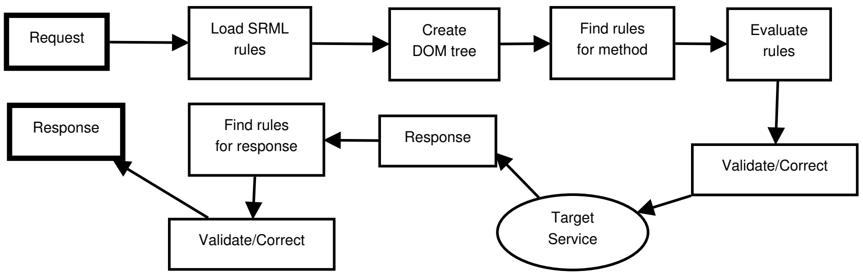


Figure 4.14: Real-time proxy flow

In case of the *real-time proxy* the initial setup is the same since the augmented wrapper class is generated, but the rules are not compiled to native code, instead every request starts out by converting its input parameters into a DOM tree using the `wsSRML.convertToDOM()` method. *Figure 4.14* shows the real-time proxy validation flow. It is not as optimal as a *native compiled* version, but allows the switching of the ruleset during runtime. *Figure 4.15* shows the augmented `performTrade` method in Java when running in *real-time* proxy mode.

During the *compiled rule plugin* mode the rules are compiled into classes and bundled into a JAR file similar to how the native compiled mode operates. The advantage here is that the rules will not need to be processed on each request but rather passed in to the proxy service to handle the request. This is considerably faster than the *real-time* rule processing since the rules are not processed over and over and the parameters are not converted to DOM trees upon every request. The drawback is that it is more difficult to change the business rules in production since they require downtime and a recompile of the rule JAR file. *Figure 4.16* shows the compiled rule JAR mode. Every service running in this mode is deployed in its own context and has a custom class loader associated with it. There is a challenge here since Java cannot use multiple versions of the dependency classes. To resolve this issue, we

```
public com.example.trades.service.TradeResponse
        performTrade(com.example.trades.service.TradeRequest inputParam)
             throws SRMLException{
    if (tradeService == null)
      _initTradeServiceProxy();

    Element inputDOM = wsSRML.convertToDOM(inputParam);

    List<Element> inputRules = wsSRML.findInputRules(inputDOM);
    for (Element rule : inputRules){
       wsSRML.applyRule(rule, inputDOM);
    }

    com.example.trades.service.TradeRequest validatedInput =
           wsSRML.reflectDOMtoObject(inputDOM,
           com.example.trades.service.TradeRequest.class);

    TradeResponse response =  tradeService.performTrade(validatedInput);

    Element resultDOM = wsSRML.convertToDOM(response);
    Element mergedDOM = wsSRML.mergeDOM(inputDOM,resultDOM);

    List<Element> outputRules = wsSRML.findResultRules(resultDOM);
    for (Element rule : outputRules){
       wsSRML.applyResultRule(rule, resultDOM, mergedDOM);
    }

    com.example.trades.service.TradeResponse validatedOutput =
          wsSRML.reflectDOMtoObject(resultDOM,
          com.example.trades.service.TradeResponse.class);

    return validatedOutput;
  }
```

Figure 4.15: Java source of the performTrade method in the real-time validation mode

use an approach similar to how OSGi works. The plugins are sand-boxed to their own environments and versions of the classes. We use IVY as the dependency management framework. The *wsSRML* proxy servlet will load all the JAR files upon startup and expose each into its own endpoint. This allows a single *wsSRML* servlet to expose and validate multiple web services on different endpoints providing a service store approach. This concept can be extended even further to potentially provide validation as a service for clients of different domains.

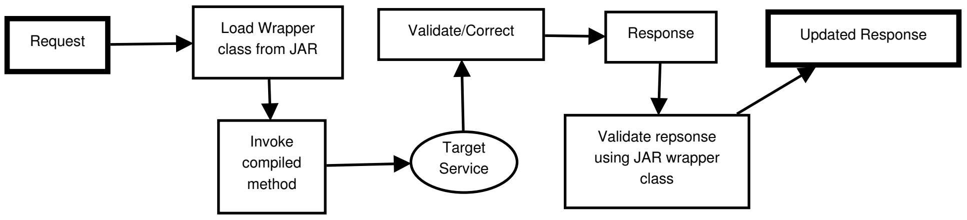


Figure 4.16: Compiled Plugin proxy flow

The third mode of *wsSRML* is based on intercepting the raw SOAP messages. This mode is pure proxy since no stubs or wrapper classes are generated. It takes the SOAP message from the request and applies the rules on the raw XML document and updates it wherever necessary. This approach is similar to the real-time mode in the sense that the rules are looked up and applied on every SOAP message. It is more transparent as no generated stubs are needed for the validation to work. It operates purely on the SOAP message that is processed into a DOM document (as it is also an XML document). The speed is not the most optimal since the ruleset is parsed upon each request and response. *Figure 4.17* shows the SOAP interception mode of the tool.

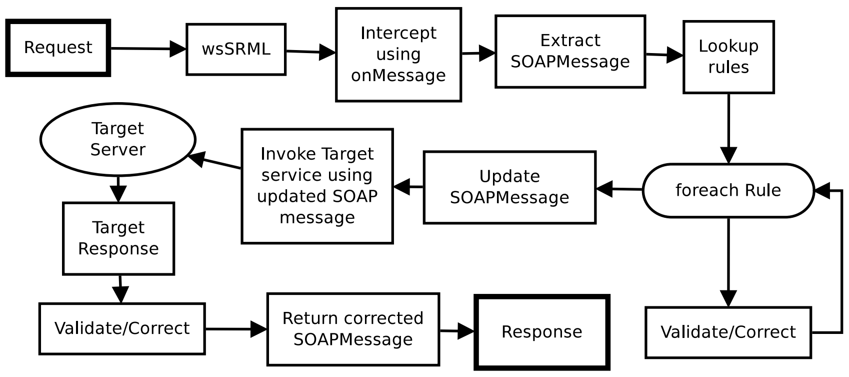


Figure 4.17: SOAP Intercept flow

- **Advantages**: Usable in situations when the client and/or server code is unavailable or cannot be modified. Allows real-time swapping and extension of validation rules without any potential downtime.
- **Disadvantages**: Slower than the native compiled version. Requires a proxy servlet to be deployed to perform the interception adding an extra level of complexity.

4.3 Summary

In this chapter we presented a way to validate web services using SRML 3.0 with the help of the `wsSRML` engine. The engine allows both *native* and *proxy* modes enabling the validation of legacy black box systems, providing a way to add validation logic to systems where the code cannot be modified but validation needs to be added. In case of native validation, our tool can wrap the generated interface stubs and provide native validation logic, offering better performance. The rules ensure a more comprehensive validation experience than formal approaches. Our validation engine also provides a way to correct the values of web service calls (both request and response values).

4.4 Related Work

All of the works outlined in this section are very notable. However, they do not provide a single solution for all the features our approach does. Our solution is not aimed at replacing them, but rather providing an all-in-one solution for validation of both the request and response of web services, providing easy-to-read rules that are extensible with function hooks. Our `wsSRML` also has provision to potentially correct the request and response of web services, making it even more attractive for applications where a higher validation level is required. Most approaches are based on the assumption that services should be validated only before deploying them into a production environment. We do not make such an assumption and feel that service validation is important not only during design and implementation, but also during actual usage to avoid potential errors. Our approach provides an in-built solution to validate legacy systems when the source code is not accessible or updating it is not an option.

In [17] they provide a model to validate SOAP message bodies. The message is fed into the model and returns an error to the user if it detects a malicious request. The engine is tuned to be usable for legacy systems. This approach is similar to ours; however, their WSIVMXML input specification contains simplistic restriction type rules only. Our approach fully leverages SRML, which allows conditional and complex validation rules to the defined not only for input parameters but also for output results.

In [40] a framework is introduced that is able to monitor and validate web service interactions. It uses OWL-S to define the semantics of Web service and employs a procedural programming approach. The monitoring framework intercepts and analyzes the traffic between a web service and the connected clients. It is fully automated and occurs during runtime. It has a CSM (constraint specification management) system that can create a pattern type approach for constraints against the named parameters. For the validation side it uses CLVMs (Content Level Validation Managers) and queues to process the incoming request. This is similar to how our chained validation rules work since we can define multiple rules and dependencies for the parameters. It does not have the option to define conditional rules compared to our solution.

The main idea behind [7] is to provide formal validation for Web Services composition that is extracted from BPEL specifications. It uses a model called *Event_B*. This model is a set of variables that evolve through events by encoding state transition systems. The rules are more formal and complex than the SRML rule definition. It does have a solid base to provide powerful constraints using THEOREMS. The description of *Event_B* allows a conditional nature. However, due to its complexity it has a very steep learning curve. The model does not cover the response of the services.

In [56] the authors provide a model-based validation strategy that is able to differentiate between correct and incorrect configurations and behaviors. The model requires the domain owners to choose abstract models that describe the affected systems and specify the incorrect configurations and behavior. They run the validation on a controlled set of realistic data by splitting the on-line system into a two slices: an on-line slice and a validation slice. Their rules were based on the `A` assertion language. The language describes a set of assertions against the elements that are typed objects themselves. The model described is able to detect connectivity, capacity and security issues. Our rule-based validation provides a more understandable and easier-to-describe ruleset. We also provide a way to validate legacy black box systems where a validation slice would not be possible.

We feel it is also important to mention some of the larger frameworks available to the community that solve service validation as well. One of these is the Spring Framework for Java, which allows a comprehensive web service validator engine using the Spring Web Services [53] project. It can also handle interception and manipulation of SOAP payloads with the help of e.g. the `PayloadValidatingInterceptor`. Our solution is also based on Java and uses Spring as the configuration framework. We use Apache CXF for the `PhaseInterceptors`, that is similar to *Spring-WS*. We must emphasize that our approach is not only a simple validator, but it can be considered as a full solution for validation and data correction. While all of the functionality can be coded in other languages and frameworks, it would still require the developers considerable effort to define rules and be able to provide additional features besides the validation.

Since validation is not language dependent, we should also mention a non-Java approach: ASP.NET web API 2 [20]. This framework allows the fields and models used in the web services to be validated using the *IValidatableObject* interface. This is very similar to how a rule-oriented approach would work. However, here the validation logic is coded into the application itself and it is more complicated to define complex relationship logic as part of the member validation. In many cases it may be enough to just validate the model and its contents. However, there are situations when the content can be corrected by applying logic that specifies what we were actually expecting in the context. This is where SRML 3.0 rules shine. They allow the user to not only specify what the data validation logic is but also define what the expected value for the given member is.

Another great .NET validator that should be mentioned is the FluentValidation [52] API. This framework makes use of lambda expressions to define rules. It allows the definition of Validators that specify rules in the style of `RuleFor(expr).function()`. An example validation rule for validating a field with a length range and providing an error message would look something like the following with this API:

```
RuleFor(customer => customer.Address).Length(20, 250).WithMessage("Invalid Length")
```

This requires the code itself to contain the validation logic, making it similar to our Native validation mode. The rule definition in SRML 3.0 for the same example would look like:

```
<rules-for root="retrieveCustomer">
 <rule-def name="@result/Address">
    <rule-instance>
      <validation-error>Invalid Length!</validation-error>
      <conditions>
         <condition>
            <entry>and(gt(length(:instance-value),20),lt(length(:instance-value),250)
                   )</entry>
            <params />
         </condition>
      </conditions>
      <values/>
   </rule-instance>
 </rule-def>
</rules-for>
```

In the above example we could have used the *between()* function to reduce the length, but we thought it might be worthwhile to show another way of chaining conditions in the new language. This SRML rule can be placed in a separate file or can be bundled in the code depending on what use-case better fits the scenario, making it more versatile.

Summary in English

The importance of data validation has been gaining more and more ground. It is essential that the data transmitted between systems communicating between each other is valid. One of the most common formats for information exchange is XML (eXtensible Markup Language). In this dissertation, four validation spaces are covered: XML, Google Protocol Buffers, Web Forms and Web Services. The chapters demonstrate the evolution of the SRML language, which was originally created to provide a way to compact XML documents. With the extension it is now possible to provide a concise way to define semantic rules for validation tasks. The extensions of the language not only allow validation but also introduce the option to correct and shape the data. This trait provides a solid framework for systems where data might be corrupted, but can be corrected with a set of semantic rules.

1. Validating XML documents

Thesis: Provide a way to validate and correct XML documents using semantic rules through the extension of SRML 1.0.
The first area of the dissertation focuses on the extension of the SRML language to permit its use in the validation space. The new version (SRML 2.0) of the language has several novel improvements compared to its predecessor that can be summarized as follows:

XPath support: Using XPath, it is now easier to reference attributes and elements in the XML context. Previously it was a tedious job to reference specific attribute instances. Earlier the reference was based on Attribute Grammars, which made descriptions more difficult.

Numeric expressions: The new format also allows numeric expressions to be used during the rule context, making it easier to describe expressions and use them in the rule definitions.

Element and attribute references: The rules can now reference both attribute and elements. Previously SRML only operated on an attribute level.

Multiple rules for the same context: With this new feature, multiple rules can be defined for the same context. This is important for validation, as it is possible that the document may be considered valid if *any* of the validation rules for that context is fulfilled.

Rule encapsulation in XSD: The rules themselves can be encapsulated in the `appinfo` section of the XSD, making the validation and structural description available in the same context.

Node relationship for tables: SRML 2.0 introduced the option to describe database tables thus extending the scope of the rules to the area of databases.

The chapter also demonstrates a potential way to validate database records using the table relationship feature of the SRML 2.0 language. This is achieved by using database triggers that fire upon specific database operations. The biggest challenge during this endeavor was how to represent database records as DOM trees. The records were flattened out and the rule structure was updated to allow the definition of the relationship between tables, using an approach similar to how foreign keys work. Using these keys we can join the records together and use the columns as attributes. This is an exciting and important area since it offers a fresh approach to validating and potentially correcting records using rules.

Summary of the thesis and own contributions

- The SRML 1.0 language was extended into the validation space. The original language was aimed at providing a way to make the XML documents smaller, more compact using semantic rules.
- The new format integrates closely with the XSD validation schema, making it portable and allowing both structural and content validation logic to be deployed in a single document.
- The new language provides XPath support and allows numerical expressions, simplifying the rule definitions.
- Another novel result for the extension is that it also provided a way to validate database records with semantic rules.
- The extension allowed the contents of the XML documents to be corrected using the rule definitions.

The majority of the topics and approaches outlined in the thesis are my contributions as the result of my research. The ideas demonstrated in the thesis were published in [35].

2. Validating Web Forms

Thesis: Create a new jSRML metalanguage, which is capable of defining semantic rules for the validation and correction of web forms.

Web forms are used to capture information in many areas of the Internet. It is vital that the data entered is valid not only from the user's point of view (confidential information, credit cards...etc), but also from a domain owner (lead capture, user details...etc). There are many form validation engines and approaches available. However, they are either too complex to use and maintain or require significant effort to update once the form fields change or the logic needs to be updated. This chapter introduces a new metalanguage called jSRML which is a semantic rule based validation language for web forms. Since web forms are HTML-based, which is similar to XML, it made sense to investigate this field as well. The `jSRMLTool` engine was built using jQuery provides a non-obtrusive way to define validation logic for forms of any domain.

The rules are highly extensible and allow for external functions to be used. The language has provision to correct the form contents in case of an invalid form submission. The engine has multiple operating modes ranging from real-time all the way to servlet-based service oriented validation schemas providing versatile application.

We also demonstrate a way to learn jSRML validation rules using machine learning techniques. Our learning engine is plugin-based which makes it highly extensible. This is an exciting area and potentially aids the domain owner to setup their validation rules, also providing an initial form of data mining on the fields by discovering relationships between them.

Summary of the thesis and own contributions

- The jSRML metalanguage was created, which is able to describe semantic validation rules for web forms. The new language is extensible and allows the use of external functions.
- The approach is non-obtrusive and is able to insert and define semantic rules in-line with the code of the form fields.
- The language allows context-oriented rule-definitions, making it a powerful tool for conditional value validation.
- The jSRML rules are able to correct the invalid field values using the rule definitions allowing the form submissions to succeed.
- The `jSRMLTool` validation tool can be executed in all four validation modes (Server-side, Client-side, Real-time, Hybrid).

- A servlet implementation of the validation engine was also implemented, which is able to provide *Validation as a Service* (VaaS) approach for forms of multiple domains.
- The validation engine's servlet can also be hooked up to intercept form values and store the results. The results are then fed into a set of machine learning plugins, which are able to suggest validation rules for the forms. This learning module also provides a way to discover relationships between field values making it a minimalistic data-mining approach.

The results of this thesis are entirely based on my contributions and are outlined in [33].

3. Validating Google Protocol Buffers

Thesis: Introduce a new metalanguage (ProtoML), which can validate and correct the messages of Google Protocol Buffers.
One of the most widespread binary-based formats for information exchange is Google Protocol Buffers. It allows a structured representation of messages and has support for various programming languages. Performing validation on these messages is not a straightforward task and it is up to the developer to implement the logic itself. To solve this problem, the thesis provides the definition of the ProtoML language. It derives its roots from SRML in a sense that it also provides semantic rules to describe the content of PB messages.

The ProtoML language brings new functionalities to the table. One of the most prominent traits is the support for function chaining and the use of external functions. Using the function oriented approach, ProtoML rules have a considerably smaller footprint compared to SRML 2.0 rules. The chapter also demonstrates the `ProtoMLTool` engine, which leverages the language for validation. The engine can be used as a library to execute validation on protocol buffer messages dynamically or natively. The native mode analyzes the `.pml` rule file and is able to create native Java code to perform the actual validation. The language also allows provision to correct messages, making it also very powerful for situations where data sanitization is essential.

Summary of the thesis and own contributions

- A new metalanguage called ProtoML was created, which is capable of validating and correcting the messages of Google Protocol Buffers using semantic rules.
- The metalanguage provided a function-oriented approach, allowing the functions to be chained together, giving ProtoML rules a considerably lighter footprint compared to SRML rules.

- The `ProtoMLTool` validation engine is able to generate Java code from the `.proto` file and the ProtoML ruleset. This allows native validation performance for Google Protocol Buffer messages.
- The validation engine can also be run in detached mode, which allows the validation rules to be fed into it during runtime.

The development and implementation of ProtoML is completely the result of my research which, were published in [32].

4. Validating Web Services

Thesis: Combine the previous metalanguages (SRML 2.0, jSRML, ProtoML) into SRML 3.0 and provide a way to validate Web Services.

The final area of the dissertation is the web service validation space. Using the positive traits and functionalities of ProtoML and jSRML, we have merged the functionality with that of SRML 2.0, creating yet another extension in the form of SRML 3.0. The key syntactic and usability improvements that SRML 3.0 has over the previous version can be summarized the following way:

- Uses a new function-oriented approach, which allows functions to be daisy chained together and evaluated easier.
- The `conditions` tag allows listing the conditions that the inspected element has to conform to. The *match* parameter can be all or any depending on whether or not the requirement is to have all condition expressions met or at least one.
- With the help of the `values` tag context-specific value definitions can be described. These expressions are evaluated top-down. The first one to match the context child expression conditions will be taken as the expected value. This is used to correct the value easier.
- Using the `validation-record` element, the definition of the XML record elements can be defined along with their primary ID attributes.
- With the help of `validation-doc-root` it is possible to define of the XML document root element.

Using this new version of SRML, we applied it to the field of web services. Web services can communicate in an XML based format as well (SOAP messages), making them ideal candidates for validation. We have built a new engine called `wsSRML`, which is able to validate and potentially correct web service requests and responses using SRML rules.

The engine has two operational modes: *native* and *proxy*. The native mode uses the WSDL file of the service and the SRML rule file and augments the `wsdl2java` output classes with the validation logic. This provides near-native validation performance since the validation rules are converted into a sequence of operations and are invoked whenever the wrapper class's methods are called.

The second operational mode is the *proxy* mode. This allows black-box and legacy systems to be enhanced with validation without the need to update their codebase. It uses a servlet to route the service call through which can validate real-time, intercept-based or using compiled rule plugins. This addition to the SRML language provides another powerful tool in the validation arsenal.

Summary of the thesis and own contributions

- The SRML 2.0, jSRML, ProtoML languages were combined into a new version of SRML. This latest extension took all the advantages of the other metalanguages and integrated it into SRML.
- The new SRML 3.0 extension provides function-oriented rule definitions, which can be daisy-chained together providing an easier description.
- The extension also separates the conditions from the expected values, making the definitions easier to read and process.
- The `wsSRML` validation engine is able to validate and correct the Request and Response of web services. The tool can operate in two modes: native and proxy.
- The engine is able to generate Java code from the SRML 3.0 rules and inject the validation logic into the wrapper classes generated by `wsdl2java`. This allows the validation logic to be executed in-line with the actual service calls.
- It is possible to run the engine in proxy mode, which will intercept the traffic using a servlet and apply the validation logic on the service packets. This mode offers a plugin submode as well, making a single servlet capable of validating multiple web service endpoints (similar to how the servlet validation mode of `jSRMLTool` worked). This can be useful in situations when the system cannot be updated, however, validation logic needs to be introduced.

The latest 3.0 extension of SRML along with the wsSRML validation engine are purely based on my results. The content of the thesis are based on [34].

Conclusion

The dissertation demonstrated how the author extended the SRML language into the field of validation. During the evolution of the language several metalanguages were created, which helped the creation of the final 3.0 version. The dissertation demonstrated a way to validate XML documents, web forms, Google Protocol Buffers and Web Services. These cover the most common formats used for information exchange, making the results of the dissertation relevant and viable solutions for every-day use. In the future we plan to extend the language even further into the binary-format validation space, providing approaches for validating distributed documents spread out over a cluster (e.g.:Hadoop).

Magyar nyelvű összefoglaló

Napjainkban egyre nagyobb szerepet kap az adatok validációja. Kulcsfontosságú, hogy a rendszerek között átvitt adat helyes legyen. Az információ cseréhez leggyakrabban az XML nyelvet használják. A disszertáció keretein belül négy területre tértünk ki: XML, Google Protocol Buffers, Webes űrlapok, Webszolgáltatások. A fejezetek végigvezetik az SRML nyelv evolúcióját, amelyek lehetővé teszik a szemantikus szabályok pontos definícióját a validációs feladatok ellátására. A nyelv kiterjesztései nem csak a validációt teszik lehetővé, de potenciálisan képesek az adatokat kijavítani. Ez a tulajdonság egy stabil keretet biztosít azon rendszereknek, ahol kulcsfontosságú az adatok minősége, viszont gyakori lehet azok sérülése. Ezen esetekben az SRML kiterjesztései szemantikus szabályokkal képesek az adat hibákat helyrehozni.

1. XML dokumentumok validációja

Tézis: Az SRML 1.0 nyelv kiterjesztése, melynek segítségével az XML dokumentumok validációja és javítása lehetségessé válik.

A disszertáció első fejezete az SRML nyelv kiterjesztését tárgyalja, hogy alkalmas legyen a validációra. Az új verzió (SRML 2.0) számos újítást tartalmaz az elődjéhez képest.

Az új SRML verzió számos újítást tartalmaz, melyek közül az alábbiakat célszerű kiemelni:

XPath támogatás: Az XPath segítségével könnyebbé válik az XML attribútumokra és elemekre való hivatkozás. Korábban a hivatkozásokat az Attribútum Nyelvtanoknál ismert módon kezelték, amely bár sokoldalú leírást tett lehetővé, bonyolult definíciókat eredményezett.

Numerikus kifejezések: Az új formátum lehetőséget biztosít arra, hogy numerikus kifejezéseket használjunk a szabály definíció során. Ez jelentősen leegyszerűsíti a szabályok leírását és olvashatóbb formát biztosít.

Elem és attribútum hivatkozások: Korábban csak attribútum hívatkozást engedélyezett a nyelv. A kiterjesztésnek köszönhetően most már mindkét entitás típusra hivatkozhatunk.

Kontextuson belül több szabály definíció engedélyezése: Ennek az új jellemzőnek köszönhetően több szabályt lehet definiálni ugyanazon kontextuson belül. Ez a validációhoz rendkívül fontos funkcionalitás, mivel lehetőség nyílik több, akár a környezettől függő szabály definiálásra. Ilyen esetben a validáció akkor lesz sikeres, ha legalább egy szabály teljesül a vizsgált elemre, vagy attribútumra.

Szabályok beágyazása XSD file-ba: Lehetőség van a szabályokat a validációs XSD dokumentum `appinfo` részében definiálni. Ennek köszönhetően a tartalmi és struktúrális validációt egy helyen lehet leírni.

A fejezet bemutat egy módszert az adatbázis rekordok validációjára az SRML 2.0 táblahivatkozásainak segítségével. Ezt a funkcionalitást adatbázis triggerek használatával képes a validációs motor elérni. A triggerek végrehajtása az adatbázis műveletek során történik. Ez egy érdekes és fontos terület, mivel egy újabb megközelítést biztosít a rekordok validációjára és potenciális javítására.

Tézis összefoglalása és saját eredmények

- Bemutattuk, miként lehet kiterjeszteni az SRML 1.0 nyelvet a validáció terére. Az eredeti nyelv specifikációja az XML dokumentumok kompaktálását célozta szemantikus szabályokkal.
- Az új formátum integrálódik az XSD validációs sémába, amely egy hordozható megoldást hoz létre. Ezen megoldásban mind a struktúrális, mind a tartalmi validációs logika egy közös dokumentumban jelenik meg.
- Az új nyelv támogatja az XPath hivatkozásokat és a numerikus kifejezéseket, melyek segítségével jelentősen leegyszerűsödnek a szabálydefiníciók.
- A kiterjesztés egy további jelentős újítást is bemutat: az adatbázisok rekordjainak validációját szemantikus szabályokkal. Ez a kiterjesztés, az adatbázisok triggerei segítségével azok műveletei során képes a rekordok tartalmát manipulálni.
- Bemutattuk, hogy miként lehet a kiterjesztés segítségével az XML dokumentum hibás értékeit kijavítani.

Az SRML validációra való kiterjesztése elsősorban az én kutatásom eredménye, melyet a [35] publikáció részletez.

2. Webes űrlapok validációja

Tézis: A jSRML metanyelv létrehozása, amely képes szemantikus szabályok alkalmazásával validálni és javítani a webes űrlapokat.
Az Internet számos területén Webes űrlapokat használnak adatbevitelre. Mind a felhasználó (személyes adatok, hitelkártya információk… stb.), mind a weboldal tulajdonos (hirdetésre jelentkező adatok, regisztrációs adatok…stb.) számára nagyon fontos, hogy a bevitt adat helyes legyen. Számos űrlap validációs motor és algoritmus létezik, viszont gyakran túl bonyolult a használatuk vagy nehezen lehet őket módosítani, ha a validációs logikát változtatni kell. Ez a fejezet bemutat egy új metanyelvet a jSRML-t, amely egy szemantikus szabály alapú validációs nyelv a webes űrlapok validálására. A `jSRMLTool` motor jQuery segítségével került kifejlesztésre, amely képes nem tolakodó módon validálni a különböző felhasználási területek űrlapjait.

A szabályok széles körben kiterjeszthetők és lehetővé teszik a külső függvények felhasználását a validációhoz. A nyelv lehetőséget ad az űrlapok tartalmának javítására hibás beküldések esetén. A `jSRMLTool` motor több üzemmódban képes működni, amely a valós idejű validációtól egészen a servlet alapú szolgáltatás modellig terjed.

A fejezet keretein belül megmutatunk egy lehetséges módot a jSRML szabályok mesterséges intelligencával történő tanulására. Ennek az alkalmazásával nem csak a validációs szabályok előállítását tudjuk megkönnyíteni, de képes egy kezdetleges adatbányászatot végezni a beküldött mezők értékei alapján.

Tézis összefoglalása és saját eredmények

- Létrejött a jSRML metanyelv, amely képes szemantikus validációs szabályokat definiálni a webes űrlapok számára. Az új nyelv jól bővíthető és lehetőség van külső függvények használatára is.

- A megoldás nem tolakodó és képes az űrlapok kódjába beszúrni a validációs szabályokat.

- A nyelv segítségével lehetőség nyílik kontextus-függő szabályokat definiálni, amelyek hasznos eszköz lehet a feltételes érték validáció terén.

- A jSRML szabályok képesek a hibás mezők értékeit kijavítani a szabályaik segítségével. Ez potenciálisan sikeressé teheti az űrlap beküldési folyamatát.

- A `jSRMLTool` validációs motort mind a négy validációs üzemmódban lehet használni (Szerver-oldali, Kliens-oldali, Valós-idejű, Hibrid)

- A validációs motor mellé egy servlet alapú implementáció is kifejlesztésre került, amely képes számos domain különböző űrlapjainak VaaS-aként (Validáció-mint-szolgáltatás) működni.
- A validációs servlet képes az űrlapok értékeit elfogni és azokat tárolni. Ezt eredményeket ezt követően a tanuló modul számos gépi tanulású algoritmussal ellátott plugin (bővítmény) segítségével kiértékeli, majd validációs szabályokat javasol. A tanuló modul képes a mezők és értékei közti összefüggéseket feltárni és ezzel egyfajta adatbányászati eszközként is alkalmazható.

A jSRML kifejlesztése és alkalmazása a webes validációra teljes mértékben az én tudományos munkám eredménye, amelyet a [33] folyóiratban publikáltunk.

3. Google Protocol Buffer üzenetek validációja

Tézis: Létrehozni egy metanyelvet (ProtoML), amely a Google Protocol Buffers üzeneteit képes validálni és kijavítani.
Az egyik legelterjetteb bináris formátum, amelyet információ cserére használnak a Google Protocol Buffers (PB). Népszerűségét annak köszönheti, hogy strukturált reprezentációt ad az üzeneteknek és számos programozási nyelvet támogat. Ezen üzenetek validációja nem egyértelmű feladat, amely eddig a programozók által való implementálásra várt. A fenti problémára hivatott megoldást biztosítani a ProtoML nyelv. Ezen metanyelv alapjait az SRML adja abból a szempontból, hogy ez a nyelv szintén szemantikus szabályokat használ a PB üzenetek mezői közötti összefüggések leírásához.

A ProtoML nyelv számos újítást is hoz magával. Az egyik legjelentősebb tulajdonság a függvények összefűzének támogatása. A függvény orientált megközelítésnek köszönhetően a ProtoML szabályok kisebb mérettel bírnak az SRML 2.0 szabályokhoz képest. A fejezetben bemutatkozik a ProtoMLTool motor is, amely a nyelvet felhasználva végzi a PB üzenetek validációját. A `ProtoMLTool` libraryként való használata dinamikus validációt tesz lehetővé. A rendszert lehet natív módban is használni, amely során a .pml szabály file vizsgálata után a motor képes Java kódot generálni, amelyben a validációt utasításokra fordítja. Ez nagy teljesítményű és gyors validációt tesz lehetővé. A ProtoML nyelv lehetőséget biztosít a PB üzenetek javítására is, amely számos esetben fontos lehet.

Tézis összefoglalása és saját eredmények

- A kifejlesztett ProtoML metanyelv képes a Google Protocol Buffers üzeneteit szemantikus szabályokkal validálni és kijavítani.

- A nyelv függvény-orientált megközelítést használ, mely során egymásba ágyazhatók a függvények. Ez jelentősen kisebb szabály méretet eredményez a korábbi SRML szabályokhoz képest.

- A ProtoMLTool validációs motor a `.proto` leíró állomány és ProtoML szabályokat felhasználva képes Java validációs kódot generálni, amelyet beleilleszthetünk a meglévő kódba. Ez közel natív teljesítményt nyújthat a Google Protocol Buffers validációjára.

- A validációs motor képes külön is futni, és a szabályokat futási időben alkalmazni.

A ProtoML nyelv és funkcionalitása teljes egészében kutatásom eredményeit képezik, amelyeket a [32] publikáció részletez.

4. Webszolgáltatások validációja

Tézis: A korábbi metanyelvek (SRML 2.0, jSRML, ProtoML) egyesítésével létrehozni egy új SRML 3.0 nyelvet, valamint megoldást találni a webszolgáltatások szemantikus szabályokkal történő validációjára.

A disszertáció utolsó validációs területe a webszolgáltatásokat tárgyalja. A ProtoML és a jSRML pozitív tulajdonságaival kiterjesztettük az SRML 2.0 nyelvet, amely segítségével előállt az SRML 3.0. Ennek az új SRML nyelvnek a segítségével a webszolgáltatások validációját céloztuk meg. Kiindulópontunk az volt, hogy a webszolgáltatások is képesek XML alapú nyelven kommunikálni (SOAP üzenetek), amely miatt ideális jelölt lett a validáció alkalmazására. Az újabb kiterjesztés alkalmazásához elkészítettük a `wsSRML` validációs motort is, amely képes SRML szabályokkal validálni és potenciálisan kijavítani a webszolgáltatás kéréseket és válaszokat. A motornak két működési üzemmódja van: natív és proxyzott. A natív mód esetén a szolgáltatás WSDL állománya és az SRML szabály file segítéségével képes a `wsdl2java` kimenetének osztályait validációs funkcionalitással kiegészíteni. A rendszer ilyenkor a validációs szabályokat műveletek sorozatára alakítja, amelyet a rendszer többi részével lefordíthatunk. Ilyenkor a wrapper osztályok metódusainak hívásakor automatikusan validálva lesz az adatforgalom.

A második üzemmód a proxyzott mód. Ennek során lehetséges zárt, illetve hagyományos rendszereket a forráskódjuk módosítása nélkül validációval bővíteni. Ehhez egy Servlet-et használunk, amely a hívó és a tényleges szolgáltatás közé integrálódik és elkapja a forgalmat. A validációt képes valós időben, interceptor alapú módban, vagy lefordított bővítmény formájában végezni.

Az új SRML 3.0 jelentősen bővíti a validációs területeit a korábbi verziókhoz képest.

Tézis összefoglalása és saját eredmények

- Az SRML 2.0, jSRML, ProtoML nyelvek integrációjával létrejött az SRML 3.0 metanyelv. Ez a nyelv a többi nyelv összes előnyét átvette, amely eredményeként egy új, hatékonyabb validációs nyelv jött létre.
- Az új 3.0 kiterjesztés függvény-orientált szabály definicókon alapul, ahol a függvényeket egymásba ágyazhatjuk (hasonlóan a ProtoML-hez). Ennek köszönhetően sokkal átláthatóbbak a szabályok, jelentősen rövidebbek és gyorsabban feldolgozhatók.
- A kiterjesztés szétválasztja a feltételekre vonatkozó szabályokat az elvárt értékekre irányuló szabályoktól, mely eredményeképp jobban áttekinthetőbbek lesznek a definíciók.
- A `wsSRML` validációs motor képes validálni és kijavítani a webszolgáltatások *Kérését* (Request) és *Válaszát* (Response). A motor két üzemmódban alkalmazható: *natív* és *proxyzott*.
- A motor, felhasználva az SRML 3.0 szabályokat, képes Java kódot generálni a szabályokból és ezeket beleinjektálni a `wsdl2java` által generált wrapper osztályokba. Ennek segítségével a validációs logikát a tényleges szolgáltatás hívásával együtt végezhetjük. A motor, proxyzott üzemmód esetén elfogja a szolgáltatás forgalmát egy köztes servlet segítségével. A szolgáltatás ezt követően validálja illetve kijavítja az üzeneteket mielőtt továbbítaná az eredeti webszolgáltatásnak. Ennek az üzemmódnak van egy *plugin* almódja is, amelynek köszönhetően a servlet képes számos webszolgáltatás validációját ellátni (hasonlóan, ahogy a jSRMLTool VaaS megközelítése működött).

Az SRML kiterjesztése webszolgáltatások validálására teljes mértékben az én tudományos munkám eredménye, melyet a [34] publikáció tartalmaz.

Összefoglaló

A disszertáció szemléltette, hogy miként lett kibővítve az SRML nyelv, hogy alkalmazható legyen a validáció terén is. A nyelv fejlődése során számos metanyelv jött létre, amely jelentősen hozzájárult a végső SRML 3.0 kialakításában. A dolgozat bemutatta, milyen módszerrel lehet XML dokumentumokat, webes űrlapokat, Google Protocol Buffers üzeneteket és webszolgáltatásokat validálni. Ezek a területek lefedik a rendszerek közti információcsere leggyakoribb módjait, mely alapján releváns témát jelentenek. A jövőben tovább akarjuk terjeszteni a nyelv korlátait a bináris állományok validációjára, illetve validációs eljárást nyújtani az klasztereken osztott dokumentumok validációjára is (pl.: Hadoop).

Appendix A

Validating XML documents

A.1 XSD of SRML 2.0

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:element name="srml-def">
      <xs:complexType>
         <xs:sequence>
                <xs:element ref="database" minOccurs="0" maxOccurs="1" />
            <xs:element ref="rules-for" minOccurs="1" maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="database">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element ref="tables" minOccurs="1" maxOccurs="unbounded" />
               <xs:element ref="references" minOccurs="1" maxOccurs="unbounded" />
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="tables">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element name="table" minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:attribute name="name" type="xs:string" />
                     <xs:attribute name="key" type="xs:string" />
                  </xs:complexType>
               </xs:element>
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="references">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element name="reference" minOccurs="1" maxOccurs="unbounded">
```

```
              <xs:complexType>
                 <xs:attribute name="root" type="xs:string" />
                 <xs:attribute name="root_key" type="xs:string" />
                 <xs:attribute name="child" type="xs:string" />
                 <xs:attribute name="child_key" type="xs:string" />
              </xs:complexType>
           </xs:element>
        </xs:choice>
     </xs:sequence>
  </xs:complexType>
</xs:element>



<xs:element name="rules-for">
  <xs:complexType>
     <xs:sequence>
        <xs:choice>
           <xs:element ref="rule-def" minOccurs="1" maxOccurs="unbounded" />
        </xs:choice>
     </xs:sequence>
     <xs:attribute name="root" type="xs:string" />
     <xs:attribute name="key" type="xs:string" use="optional" />
  </xs:complexType>
</xs:element>


<xs:element name="rule-def">
  <xs:complexType>
     <xs:sequence>
        <xs:element ref="rule-instance" minOccurs="1" maxOccurs="unbounded" />
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="required" />
     <xs:attribute name="mode" default="validate" use="optional">
        <xs:simpleType>
           <xs:restriction base="xs:string">
              <xs:enumeration value="validate" />
              <xs:enumeration value="correct" />
           </xs:restriction>
        </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="match" default="any" use="optional">
        <xs:simpleType>
           <xs:restriction base="xs:string">
              <xs:enumeration value="any" />
              <xs:enumeration value="all" />
           </xs:restriction>
        </xs:simpleType>
     </xs:attribute>
     <xs:attribute name="key" type="xs:string" use="optional" />
  </xs:complexType>
</xs:element>

<xs:element name="rule-instance">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="validation-error" type="xs:string" />
        <xs:element name="expr" type="ExprType" minOccurs="1"
           maxOccurs="unbounded" />
     </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="ExprType">
```

```
      <xs:choice>
         <xs:element ref="binary-op" minOccurs="1" maxOccurs="1" />
         <xs:element ref="attribute" minOccurs="1" maxOccurs="1" />
         <xs:element name="data" type="xs:string" minOccurs="1"
            maxOccurs="1" />
         <xs:element name="no-data" minOccurs="1" maxOccurs="1"
            type="xs:string" />
         <xs:element ref="if-element" minOccurs="1" maxOccurs="1" />
         <xs:element ref="if-all" minOccurs="1" maxOccurs="1" />
         <xs:element ref="if-any" minOccurs="1" maxOccurs="1" />
         <xs:element ref="if-expr" minOccurs="1" maxOccurs="1" />
         <xs:element name="current-attribute" minOccurs="1"
            maxOccurs="1" type="xs:string" />
         <xs:element name="position" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:attribute name="element" type="BinaryOpTypes" />
               <xs:attribute name="from" default="begin">
                  <xs:simpleType>
                     <xs:restriction base="xs:string">
                        <xs:enumeration value="begin" />
                        <xs:enumeration value="current" />
                        <xs:enumeration value="end" />
                     </xs:restriction>
                  </xs:simpleType>
               </xs:attribute>
            </xs:complexType>
         </xs:element>
         <xs:element name="instance-value" minOccurs="1" maxOccurs="1" />
         <xs:element name="count-children" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:attribute name="name" type="xs:string" />
               <xs:attribute name="key" type="xs:string" />
            </xs:complexType>
         </xs:element>
         <xs:element name="count-siblings" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:attribute name="name" type="xs:string" />
               <xs:attribute name="key" type="xs:string" />
            </xs:complexType>
         </xs:element>
         <xs:element name="reg-eval" minOccurs="1" maxOccurs="1"
            type="xs:string" />
         <xs:element name="value-ref" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:attribute name="path" type="xs:string" />
            </xs:complexType>
         </xs:element>
      </xs:choice>
   </xs:complexType>

   <xs:element name="binary-op">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="expr" minOccurs="2" maxOccurs="2"
               type="ExprType" />
         </xs:sequence>
         <xs:attribute name="op" type="BinaryOpTypes" use="required" />
      </xs:complexType>
   </xs:element>

   <xs:element name="attribute">
      <xs:complexType>
         <xs:attribute name="element" type="BinaryOpTypes" use="required" />
         <xs:attribute name="num" type="xs:integer" default="0" />
         <xs:attribute name="from" default="begin">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="begin" />
                  <xs:enumeration value="current" />
                  <xs:enumeration value="end" />
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
         <xs:attribute name="attrib" type="xs:string" use="required" />
      </xs:complexType>
   </xs:element>

   <xs:element name="if-element">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="expr" minOccurs="2" maxOccurs="2"
               type="ExprType" />
         </xs:sequence>
         <xs:attribute name="from" default="begin">
            <xs:simpleType>
               <xs:restriction base="xs:token">
                  <xs:enumeration value="begin" />
                  <xs:enumeration value="end" />
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>


   </xs:element>

   <xs:element name="if-all">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="expr" minOccurs="3" maxOccurs="3"
               type="ExprType" />
         </xs:sequence>
         <xs:attribute name="element" type="xs:string" default="srml:all" />
         <xs:attribute name="attrib" type="xs:string" default="srml:all" />
      </xs:complexType>
   </xs:element>

   <xs:element name="if-any">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="expr" minOccurs="3" maxOccurs="3"
               type="ExprType" />
         </xs:sequence>
         <xs:attribute name="element" type="xs:string" default="srml:all" />
         <xs:attribute name="attrib" type="xs:string" default="srml:all" />
      </xs:complexType>
   </xs:element>


   <xs:element name="if-expr">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="expr" minOccurs="3" maxOccurs="3"
               type="ExprType" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

```
   <xs:simpleType name="BinaryOpTypes">
      <xs:restriction base="xs:string">
         <xs:enumeration value="add" />
         <xs:enumeration value="sub" />
         <xs:enumeration value="mul" />
         <xs:enumeration value="div" />
         <xs:enumeration value="exp" />
         <xs:enumeration value="equal" />
         <xs:enumeration value="not-equal" />
         <xs:enumeration value="less" />
         <xs:enumeration value="greater" />
         <xs:enumeration value="or" />
         <xs:enumeration value="xor" />
         <xs:enumeration value="and" />
         <xs:enumeration value="nor" />
         <xs:enumeration value="contains" />
         <xs:enumeration value="concat" />
         <xs:enumeration value="begins-with" />
         <xs:enumeration value="ends-with" />
         <xs:enumeration value="equal-rounded" />
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Appendix B

Validating Web Forms

B.1 Full XSD of jSRML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="validate-input">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="error-text" minOccurs="1" maxOccurs="1" type="xs:string" />
                <xs:element name="css" minOccurs="1" maxOccurs="1">
                    <xs:complexType>
                        <xs:attribute name="invalid" type="xs:string" />
                        <xs:attribute name="error-class" type="xs:string" />
                    </xs:complexType>
                </xs:element>
                <xs:element name="action" minOccurs="1" maxOccurs="1">
                    <xs:complexType>
                        <xs:attribute name="valid" type="xs:string" />
                        <xs:attribute name="invalid" type="xs:string" />
                    </xs:complexType>
                </xs:element>
                <xs:element ref="conditions" minOccurs="1" maxOccurs="1" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" />
                <xs:attribute name="form" type="xs:string" />

                <xs:attribute name="required-field" default="false" use="optional">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="true" />
                            <xs:enumeration value="false" />
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="mode" default="validate" use="optional">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="validate" />
                            <xs:enumeration value="replace" />
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="method" default="standard" use="optional">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="standard" />
                            <xs:enumeration value="real-time" />
                            <xs:enumeration value="focus" />
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
    </xs:element>


    <xs:element name="conditions">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="expr" type="ExprType" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>


    <xs:complexType name="ExprType">
        <xs:choice>
            <xs:element ref="binary-op" minOccurs="1" maxOccurs="1" />
            <xs:element name="data" type="xs:string" minOccurs="1"
                maxOccurs="1" />
            <xs:element ref="if-expr" minOccurs="1" maxOccurs="1" />
            <xs:element name="text-length" minOccurs="1" maxOccurs="1" />
            <xs:element name="text-value" minOccurs="1" maxOccurs="1" />
            <xs:element name="field-length" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:attribute name="id" type="xs:string" />
                </xs:complexType>
            </xs:element>
            <xs:element name="field-value" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:attribute name="id" type="xs:string" />
                </xs:complexType>
            </xs:element>
            <xs:element name="reg-eval" minOccurs="1" maxOccurs="1"
                type="xs:string" />
            <xs:element name="text-format" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:attribute name="value">
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="numeric" />
                                <xs:enumeration value="date" />
                                <xs:enumeration value="email" />
                                <xs:enumeration value="regexp" />
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="expression" use="optional"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="has-value" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:attribute name="id" type="xs:string" />
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>

    <xs:element name="binary-op">
        <xs:complexType>
```

```
            <xs:sequence>
                <xs:element name="expr" minOccurs="2" maxOccurs="2"
                    type="ExprType" />
            </xs:sequence>
            <xs:attribute name="op" type="BinaryOpTypes" use="required" />
        </xs:complexType>
    </xs:element>

    <xs:element name="if-expr">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="expr" minOccurs="3" maxOccurs="3"
                    type="ExprType" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name="BinaryOpTypes">
        <xs:restriction base="xs:string">
            <xs:enumeration value="equals" />
            <xs:enumeration value="not-equal" />
            <xs:enumeration value="lt" />
            <xs:enumeration value="gt" />
            <xs:enumeration value="gte" />
            <xs:enumeration value="lte" />
            <xs:enumeration value="date-lt" />
            <xs:enumeration value="date-lte" />
            <xs:enumeration value="date-eq" />
            <xs:enumeration value="date-gte" />
            <xs:enumeration value="date-gt" />
            <xs:enumeration value="contains" />
<xs:enumeration value="not-contains" />
            <xs:enumeration value="begins-with" />
            <xs:enumeration value="ends-with" />
            <xs:enumeration value="and" />
            <xs:enumeration value="or" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

B.2 Full source of example

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
   <title>jSRML demo</title>
   <link rel="stylesheet" type="text/css" href="resources/css/simple.css">
  <script src="resources/js/jquery-1.8.3.js"></script>
  <script src="resources/js/jSRMLTool.js"></script>
 </head>
 <body>

 <script type="text/javascript">

    function error(id){
        alert("Callback function for error on validation "+id);
    }


     $(document).ready(function(){
    //  alert("starting");
        initializeSRML();
    });
   </script>

  <div class="form-cnt">
  <h2>Validation Example using jSRML</h2>
   <form method="post" action="post-result" id="myform" name="myform">
        <div class="row-cnt">
            <div class="row-label">
                Field 01 [min 5 chars]:
            </div>
            <div class="row-field">
              <input type="text" name="field_01" id="field_01" value="12345" class="row-item" />
                <!--[SRML]
                    <validate-input id="field_01" form="myform" mode="validate">
                        <error-text>The size needs to be larger than 5 characters</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <binary-op op="gte">
                                    <expr><text-length/></expr>
                                    <expr><data>5</data></expr>
                                </binary-op>
                            </expr>
                        </conditions>
                     </validate-input>

                -->
            </div>
        </div>
        <br style="clear:both" />
        <div class="row-cnt">
            <div class="row-label">
                Field 02 [numeric]:
            </div>
            <div class="row-field">
              <input type="text" name="field_02" id="field_02"  value="123" class="row-item" />
                <!--[SRML]
                     <validate-input id="field_02" form="myform" mode="validate">
                        <error-text>Invalid number format!</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <text-format value="numeric" />
                            </expr>
                        </conditions>
                     </validate-input>
                -->
              </div>
         </div>
                 <br style="clear:both" />
         <div class="row-cnt">
            <div class="row-label">
                Field 03 [date MM/dd/yyyy]:
             </div>
            <div class="row-field">
              <input type="text" name="field_03" id="field_03"
                value="12/28/2012" class="row-item" />
                <!--[SRML]
                     <validate-input id="field_03" form="myform" mode="validate">
                        <error-text>Invalid number format!</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
```

```
                <conditions>
                    <expr>
                        <text-format value="date" />
                    </expr>
                </conditions>
             </validate-input>
        -->
      </div>
 </div>
        <br style="clear:both" />
<div class="row-cnt">
    <div class="row-label">
        Field 04 [regexp ISBN D-DDDDD-DDD-D]:
    </div>
    <div class="row-field">
      <input type="text" name="field_04" id="field_04"
        value="1-12345-123-1" class="row-item" />
        <!--[SRML]
             <validate-input id="field_04" form="myform" mode="validate">
                <error-text>Invalid ISBN format!</error-text>
                <css invalid="inp-form-error" error-class="form_error_message error" />
                <action valid="" invalid="error" />
                <conditions>
                    <expr>
                        <text-format value="regexp"
                          expression="^\d{1}-\d{5}-\d{3}-\d{1}$" />
                    </expr>
                </conditions>
             </validate-input>
        -->
      </div>
 </div>
        <br style="clear:both" />
<div class="row-cnt">
    <div class="row-label">
        Field 05 [numeric and max 100]:
    </div>
    <div class="row-field">
       <input type="text" name="field_05" id="field_05" value="39" class="row-item" />
        <!--[SRML]
             <validate-input id="field_05" form="myform" mode="validate">
                <error-text>Invalid number format! Maximum value 100!</error-text>
                <css invalid="inp-form-error" error-class="form_error_message error" />
                <action valid="" invalid="error" />
                <conditions>
                    <expr>
                        <text-format value="numeric" />
                    </expr>
                    <expr>
                        <binary-op op="lt">
                            <expr><text-value/></expr>
                            <expr><data>100</data></expr>
                        </binary-op>
                    </expr>
                </conditions>
             </validate-input>
        -->
      </div>
 </div>
        <br style="clear:both" />
<div class="row-cnt">
    <div class="row-label">
        Field 06 [numeric and equals fifth+second]:
    </div>
```

```
            <div class="row-field">
                <input type="text" name="field_06" id="field_06" value="162" class="row-item" />
                <!--[SRML]
                     <validate-input id="field_06" form="myform" mode="validate">
                        <error-text>Invalid number format! Should be numeric and the sum of
                          fifth+second!</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <text-format value="numeric" />
                            </expr>
                            <expr>
                                <binary-op op="equals">
                                    <expr><text-value/></expr>
                                    <expr>
                                        <reg-eval>[{field_02}]+[{field_05}]</reg-eval>
                                    </expr>
                                </binary-op>
                            </expr>
                        </conditions>
                     </validate-input>
                -->
              </div>
         </div>
                <br style="clear:both" />
        <div class="row-cnt">
            <div class="row-label">
                Field 07 [email]:
            </div>
            <div class="row-field">
              <input type="text" name="field_07" id="field_07"
               value="test@email.com" class="row-item" />
                <!--[SRML]
                     <validate-input id="field_07" form="myform" mode="validate">
                        <error-text>Invalid email format!</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <text-format value="email" />
                            </expr>
                    </conditions>
                     </validate-input>
                -->
              </div>
         </div>
                <br style="clear:both" />
        <div class="row-cnt">
            <div class="row-label">
                Field 08 [password min 6 chars]:
            </div>
            <div class="row-field">
                <input type="password" name="field_08" id="field_08"
                 value="1234567" class="row-item" />
                <!--[SRML]
                     <validate-input id="field_08" form="myform" mode="validate">
                        <error-text>Minimum 6 characters</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <binary-op op="gte">
                                    <expr><text-length /></expr>
                                    <expr><data>6</data></expr>
                                </binary-op>
                            </expr>
                        </conditions>
                     </validate-input>
                -->
              </div>
         </div>
                <br style="clear:both" />
        <div class="row-cnt">
            <div class="row-label">
                Field 09 [password+retype]:
            </div>
            <div class="row-field">
                <input type="password" name="field_09" id="field_09"
                value="1234567" class="row-item" />
                <!--[SRML]
                     <validate-input id="field_09" form="myform" mode="validate">
                        <error-text>Value does not match Field 08</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <binary-op op="equals">
                                    <expr><field-value id="field_08" /></expr>
                                    <expr><text-value /></expr>
                                </binary-op>
                            </expr>
                        </conditions>
                     </validate-input>
                -->
              </div>
         </div>
                <br style="clear:both" />
        <div class="row-cnt">
            <div class="row-label">
                Field 10 [Has to be Cat] :
            </div>
            <div class="row-field">
                <select name="field_10" id="field_10" class="row-item">
                    <option value="cat">Cat</option>
                    <option value="bird">Bird</option>
                </select>
                 <!--[SRML]
                     <validate-input id="field_10" form="myform" mode="validate">
                        <error-text>Please select cats!</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <binary-op op="equals">
                                    <expr><text-value /></expr>
                                    <expr><data>cat</data></expr>
                                </binary-op>
                            </expr>
                         </conditions>
                      </validate-input>
                -->
              </div>
         </div>
               <br style="clear:both" />
        <div class="row-cnt">
            <div class="row-label">
                Field 11 [if cat then it has legs, otherwise wings]:
            </div>
            <div class="row-field">
                <select name="field_11" id="field_11" class="row-item">
                    <option value="legs">Legs</option>
                    <option value="wings">Wings</option>
                </select><br style="clear:both" />
                 <!--[SRML]
                     <validate-input id="field_11" form="myform" mode="validate">
                        <error-text>Cats have legs and Birds have wings!</error-text>
                        <css invalid="inp-form-error" error-class="form_error_message error" />
                        <action valid="" invalid="error" />
                        <conditions>
                            <expr>
                                <binary-op op="equals">
                                  <expr>
                                    <if-expr>
                                        <expr>
                                            <binary-op op="equals">
                                                <expr><field-value id="field_10" /></expr>
                                                <expr><data>cat</data></expr>
                                            </binary-op>
                                        </expr>
                                        <expr><data>legs</data></expr>
                                        <expr><text-value /></expr>
                                    </if-expr>
                                  </expr>
                                  <expr><text-value /></expr>
                                </binary-op>
                            </expr>
                         </conditions>
                        </validate-input>
                -->
              </div>
         </div>
                <br style="clear:both" />
          <div class="row-cnt">
            <div class="row-label">
        <input type="submit" value="Submit Form"  /><br/>
        </div>
        </div>
        <div class="row-field">&nbsp;</div>
                <br style="clear:both" />
   </form>
  </div>

 </body>
 </html>
```

Appendix C

Validating Google Protocol Buffers

C.1 Function List of ProtoML

- `add(`v_1`,`v_2`)` : Returns the sum of v_1 and v_2.
- `and(`c_1`,`c_2`)` : Returns true if both conditions are true.
- `begins-with(`s_1`,`s_2`)`: Returns true if s_1 begins with s_2.
- `camelcase(`s_1`)` : Converts s_1 to camelcase.
- `child(xp,name)` : Returns the list of child nodes of the XPath expression stored in *xp* which have a name of *name*.
- `contains(`v_1`,`$m_1, ..., , m_n$`)` : Returns true if v_1 is present in the set of $(m_1, ..., , m_n)$.
- `count-children(xp,name)` : Returns the number of children under the XPath xp with the name of `name`.
- `div(`v_1`,`v_2`)` : Divides v_1 with v_2 .
- `ends-with(`s_1`,`s_2`)` : Returns true if s_1 ends with the string in s_2.
- `eq(`v_1`,`v_2`)` : If v_1 is equal to v_2 the value is true.
- `eval(expr)` : Evaluates the expression in *expr*. It is possible to refer to XPath elements using their XPath in $\${path}$ placeholders (e.g.: *eval(*$\${//HouseHold/TotalIncome} + 1$*)*. The expression can also contain functions.
- `for-all(xp,op)` : This is an iterative function that will query all fields that match the XPath *xp* and execute the named function of *op* on it. It is usable on all two-parameter functions. It takes the current aggregate value $(op(op(op(v_1, v_2), v_3), ..., v_{n-1}))$ as the first parameter and the actual value (v_n) as the second.
- `ge(`v_1`,`v_2`)` : If v_1 is greater or equal than v_2 then this function returns true.
- `gt(`v_1`,`v_2`)` : If v_1 is greater than v_2 then this function returns true.
- `if(`c_1`,`v_1`,`v_2`)` : If c_1 is true then the result is v_1, otherwise v_2.
- `if-all(xp,`c_1`,`v_1`,`v_2`)` : Matches the condition c_1 on all fields returned by XPath *xp*. If the result was true for all the end result if v_1, otherwise v_2.

- `if-any(xp,`c_1`,`v_1`,`v_2`)` : Works similarly to how if-all works with the difference that here the value if v_1 if at least one field evaluated true on the c_1 condition.
- `index-of(`s_1`,`s_2`)` : Returns the index of s_2 if it is a substring of s_1, otherwise returns -1.
- `le(`v_1`,`v_2`)` : If v_1 is less or equal than v_2 then this function returns true.
- `length(`s_1`)` : Returns the length of the s_1 string.
- `lowercase(`s_1`)` : Converts s_1 to lowercase.
- `lt(`v_1`,`v_2`)` : If v_1 is less than v_2 then this function returns true.
- `mul(`v_1`,`v_2`)` : Returns the multiplied value of v_1 and v_2.
- `or(`c_1`,`c_2`)` : Returns true if either c_1 or c_2 is true.
- `path(node)` : Returns the XPath of a given field defined in *node*.
- `regex(`v_1`,expr)` : Evaluates v_1 against the regular expression described in *expr*.
- `round(`v_1`,dec)` : rounds the value in v_1 to a decimal places of *dec*. If *dec* is set to 0 then the value is rounded to an integer.
- `sibling(xp,name)` : Returns the field node with the name of *name* resulting from the query of XPath *xp*. This can be used for example when accessing a field on the same level of the message (sibling(:path,"fieldname")). This will only return the actual field, not its value. If the value needs to be retrieved then it has to be surrounded by a `val()` function call.
- `sub(`v_1`,`v_2`)` : Subtract v_2 from v_1.
- `substring(`s_1`,`s_2`)` : Returns true if s_2 is a substring of s_1.
- `typeof(`v_1`,type)` : returns true if the type of v_1 matches *type* . Currently the following *types* are supported: *integer*, *float*, *string*. Types are usually inferred by the `proto` message definition. However, using this allows type forcing on string definitions.
- `uppercase(`s_1`)` : Converts s_1 to uppercase.
- `val(xp)` : Returns the value defined by the XPath *xp*. If the XPath is a list then the first element value is returned.

Appendix D

Validating Web Services

D.1 XSD of SRML 3.0

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:element name="srml-def">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="database" minOccurs="0" maxOccurs="1" />
            <xs:element ref="validation-doc-root" minOccurs="0" maxOccurs="1" />
            <xs:element ref="validation-record" minOccurs="0" maxOccurs="1" />
            <xs:element ref="rules-for" minOccurs="1" maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="database">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element ref="tables" minOccurs="1" maxOccurs="unbounded" />
               <xs:element ref="references" minOccurs="1" maxOccurs="unbounded" />
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="tables">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element name="table" minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:attribute name="name" type="xs:string" />
                     <xs:attribute name="key" type="xs:string" />
                  </xs:complexType>
               </xs:element>
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="references">
      <xs:complexType>
         <xs:sequence>
```

```
            <xs:choice>
               <xs:element name="reference" minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:attribute name="root" type="xs:string" />
                     <xs:attribute name="root_key" type="xs:string" />
                     <xs:attribute name="child" type="xs:string" />
                     <xs:attribute name="child_key" type="xs:string" />
                  </xs:complexType>
               </xs:element>
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

 <xs:element name="validation-doc-root">
      <xs:complexType>
 <xs:attribute name="name" type="xs:string"  use="required"/>
      </xs:complexType>
   </xs:element>

    <xs:element name="validation-record">
      <xs:complexType>
 <xs:attribute name="name" type="xs:string" use="required"/>
 <xs:attribute name="id" type="xs:string" use="optional"/>
      </xs:complexType>
   </xs:element>



   <xs:element name="rules-for">
      <xs:complexType>
         <xs:sequence>
            <xs:choice>
               <xs:element ref="rule-def" minOccurs="1" maxOccurs="unbounded" />
            </xs:choice>
         </xs:sequence>
         <xs:attribute name="root" type="xs:string" />
         <xs:attribute name="key" type="xs:string" use="optional" />
      </xs:complexType>
   </xs:element>


   <xs:element name="rule-def">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="rule-instance" minOccurs="1" maxOccurs="unbounded" />
         </xs:sequence>
         <xs:attribute name="name" type="xs:string" use="required" />
         <xs:attribute name="mode" default="validate" use="optional">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="validate" />
                  <xs:enumeration value="correct" />
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
         <xs:attribute name="match" default="any" use="optional">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="any" />
                  <xs:enumeration value="all" />
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
         <xs:attribute name="key" type="xs:string" use="optional" />
      </xs:complexType>
   </xs:element>

   <xs:element name="rule-instance">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="validation-error" type="xs:string" />
            <xs:element name="conditions" type="ConditionType" minOccurs="1"
               maxOccurs="1" />
   <xs:element name="values" type="ValuesType" minOccurs="1"
               maxOccurs="1" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:complexType name="ConditionType">
    <xs:element name="condition" minOccurs="0" maxOccurs="onbounded">
    <xs:complexType>
      <xs:attribute name="match" default="any" use="optional">
             <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="any" />
                  <xs:enumeration value="all" />
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
    <xs:sequence>
    <xs:element name="entry" minOccurs="1" maxOccurs="1" />
<xs:element name="params" minOccurs="0" maxOccurs="1" />
</xs:sequence>
    </xs:complexType>
    </xs:element>
   </xs:complexType>

   <xs:complexType name="ValuesType">
   <xs:complexType>
    <xs:attribute name="match" default="any" use="optional">
    <xs:simpleType>
 <xs:restriction base="xs:string">
                  <xs:enumeration value="any" />
                  <xs:enumeration value="all" />
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
    <xs:element name="value-entry" minOccurs="0" maxOccurs="onbounded">
    <xs:sequence>
    <xs:element name="eval" minOccurs="0" maxOccurs="1" />
    <xs:element name="entry" minOccurs="1" maxOccurs="1" />
<xs:element name="params" minOccurs="0" maxOccurs="1" />
</xs:sequence>

    </xs:element>
    </xs:complexType>
   </xs:complexType>

</xs:schema>
```

D.2 Functions of SRML 3.0

- `add(`v_1`,`v_2`)`: Returns the sum of v_1 and v_2

- `and(`c_1`,`c_2`)`: Returns true if both conditions are true
- `begins-with(`s_1`,`s_2`)`: Returns true if s_1 begins with s_2
- `camelcase(`s_1`)`: Converts s_1 to camelcase
- `child(xp,name)`: Returns the list of child nodes of the XPath expression stored in ”xp” which have a name of ”name”
- `contains(`v_1`,`m_1`,...,`m_n`)`: Returns true if v1 is present in the set of (m1,...,mn)
- `count-children(xp,name)`: Returns the number of children under the XPath xp with the name of name
- `current_timestamp()`: Returns the current timestamp
- `current_date(f)`: Returns the current date using the format specified in ”f”. E.g.: ”YYYY-mm-dd”
- `div(`v_1`,`v_2`)`: Divides v_1 with v_2
- `ends-with(`s_1`,`s_2`)`: Returns true if s_1 ends with the string in s_2
- `eq(`v_1`,`v_2`)`: If v_1 is equal to v_2 the value is true
- `eval(expr)`: Evaluates the expression in ”expr”. It is possible to refer to XPath elements using their XPath. The expression can also contain functions
- `for-all(xp,op)`: This is an iterative function that will query all fields that match the XPath ”xp” and execute the named function of ”op” on it. It is usable on all two-parameter functions. It takes the current aggregate value (op(op(op(v_1,v_2),v_3) , ... , v_{n-1})) as the first parameter and the actual value (v_n) as the second
- `ge(`v_1`,`v_2`)`: If v_1 is greater or equal than v_2 then this function returns true
- `gt(`v_1`,`v_2`)`: If v_1 is greater than v_2 then this function returns true
- `has_format(`v_1`,f)`: Returns true if v_1 has a format of ”f”. Currently the following formats are supported: *email, ipv4, ipv6,url, date, shortdate*
- `if(`c_1`,`v_1`,`v_2`)`: If c_1 is true then the result is v_1, otherwise v_2
- `if-all(xp,`c_1`,`v_1`,`v_2`)`: Matches the condition c_1 on all fields returned by XPath ”xp”. If the result was true for all the end result if v_1, otherwise v_2
- `if-any(xp,`c_1`,`v_1`,`v_2`)`: Works similarly to how if-all works with the difference that here the value if v_1 if at least one field evaluated true on the c_1 condition
- `index-of(`s_1`,`s_2`)`: Returns the index of s_2 if it is a substring of s_1, otherwise returns -1
- `le(`v_1`,`v_2`)`: If v_1 is less or equal than v_2 then this function returns true
- `length(`s_1`)`: Returns the length of the s_1 string
- `lowercase(`s_1`)`: Converts s_1 to lowercase
- `lt(`v_1`,`v_2`)`: If v_1 is less than v_2 then this function returns true
- `mul(`v_1`,`v_2`)`: Returns the multiplied value of v_1 and v_2
- `or(`c_1`,`c_2`)`: Returns “true” if either c_1 or c_2 is “true”
- `path(node)`: Returns the XPath of a given field defined in “node”
- `regex(`v_1`,expr)`: Evaluates v_1 against the regular expression described in “expr”

- `round(`v_1`,dec)`: Rounds the value in v_1 to a decimal places of “dec”. If “dec” is set to 0 then the value is rounded to an integer
- `sibling(xp,name)`: Returns the field node with the name of “name” resulting from the query of XPath “xp”. This can be used for example when accessing a field on the same level of the message (sibling(`:path`,“fieldname”)). This will only return the actual field, not its value. If the value needs to be retrieved then it has to be surrounded by a `val()` function call.
- `sub(`v_1`,`v_2`)`: Subtract v_2 from v_1
- `substring(`s_1`,`s_2`)`: Returns “true” if s_2 is a substring of s_1
- `timstamp(s,f)`: Returns the timestamp based on the date specified in “s” using the format of ”f”
- `type-check(`v_1`,type)`: Returns “true” if the type of v_1 matches “type”. Currently the following types are supported: integer, float, string
- `uppercase(`s_1`)`: Converts s_1 to uppercase
- `val(xp)`: Returns the value defined by the XPath ”xp”. If the XPath is a list then the first element value is returned

Bibliography

[1] Cam. [Online]. Available: https://wiki.oasis-open.org/cam/

[2] Document object model (DOM), http://www.w3.org/dom/. [Online]. Available: http://www.w3.org/DOM/

[3] "H2 database engine, http://www.h2database.com/html/main.html." [Online]. Available: http://www.h2database.com/html/main.html

[4] "Automated server-side form validation," in *International Conference on Informatics.* Electronics & Vision (ICIEV), May 18-19 2012, pp. 61–64.

[5] H. Adorf, "Form validation with Rule Bases," 2010. [Online]. Available: http://blog.mgm-tp.com/2010/10/test-data-generation-part1

[6] S. M. K. M. Agarwal, A., "Thrift: Scalable Cross-Language Services Implementation," 2007. [Online]. Available: http://thrift.apache.org/static/files/thrift-20070401.pdf

[7] I. Ait-Sadoune and Y. Ait-Ameur, "A Proof Based Approach for Modelling and Verifying Web Services Compositions," in *14th IEEE International Conference on Engineering of Complex*, 2009, pp. 1–10.

[8] F. Asseg, "Exp4j, http://www.objecthunter.net/exp4j/," 10 2011. [Online]. Available: http://www.objecthunter.net/exp4j/

[9] BCI, "Dental implant abroad," 2014. [Online]. Available: http://www.dental-implantabroad.co.uk/ental-implant-overseas

[10] A. Birrell and B. Nelson, "Implementing remote procedure calls," Xerox Corporation, Tech. Rep. Technical Report CSL-83-7, 1983.

[11] L. Blando, "A Framework for a Rule-Based Form Validation Engine," *http://wiki.lassy.uni.lu/Special:LassyBibDownload?id=324*, 1999.

[12] D. Booth and C. Liu, "Web Services Description Language Version 2.0," W3C Recommendation, http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/, Tech. Rep., June 26 2007.

[13] D. Box and D. Ehnebuske, "Simple Object Access Protocol (SOAP) 1.1," World Wide Web Consortium, http://www.w3.org/TR/SOAP/, Tech. Rep., 2000.

[14] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," in *International Conference on Applied Cryptography and Network Security (ACNS)*, LNCS, Ed., vol. 2, 2004.

[15] T. Bray, J. Paoli, and C. Sperberg-McQueen. (1998) Extensible markup language. [Online]. Available: http://www.w3.org/TR/REC-xml

[16] L. Breiman, "Random Forests," in *Proceedings of Machine Learning*, 2001, pp. 5–32.

[17] R. Brinhosa and C. Westphall, "Proposal and Development of the Web Services Input Validation Model," in *IEEE Network Operations and Management Symposium (NOMS)*, 2012, pp. 643–646.

[18] J. Clark and S. DeRose. (1999) XML Path Language (XPath) Version 1.0. [Online]. Available: http://www.w3.org/TR/xpath

[19] D. Cockford, *Javascript: The Good Parts.* O'Reilly, 2008.

[20] M. Corporation, "Asp.net web api 2," http://www.asp.net/web-api, Tech. Rep., 2014.

[21] E. Escott and P. Strooper, "Model-Driven Web Form Validation with UML and OCL," in *Lecture Notes in Computer Science*, 2012, vol. 7059, pp. 223–225.

[22] H. Fernau, "Algorithms for learning regular expressions from positive data," *Information and Computation*, vol. 207, no. 4, pp. 521–541, 2009.

[23] J. Garret. Ajax: A New Approach to Web Applications. [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[24] J. Giannoudis, "Rule based validation for asp.net." [Online]. Available: http://www.codeproject.com/Articles/367214/Rule-Based-Validation-for-ASP-NET

[25] C. Goldfarb and P. Prescod, *The XML Handbook.* Prentice-Hall, 2001.

[26] Google. (2008) Protocol Buffer, http://code.google.com/apis/protocolbuffers/docs/overview.html.

[27] M. Handley, "Internet Denial-of-Service Considerations," IAB, RFC4732, Tech. Rep., 2006.

[28] T. R. Hastie, T. and J. Friedman, *The Elements of Statistical Learning.* Springer, 2001, no. ISBN 0–387–95284–5.

[29] J. Hunter. (2000) JDOM, http://jdom.org/docs/apidocs/. [Online]. Available: http://jdom.org/docs/apidocs/

[30] J. Hunter and W. Crawford, *Java Servlet Programming*, 2nd ed. O'Reilly, 2001.

[31] M. Kálmán, “An approach for compacting XMI documents,” *Acta Cybernetica*, vol. 17, no. 2, pp. 289–310, 2005.

[32] M. Kálmán, “ProtoML: A rule-based validation language for Google Protocol Buffers,” in *Proceedings of the 8th International Conference for Internet Technology and Secured Transactions (ICITST)*, London, UK, December 9-12 2013, IEEE Computer Society, pp. 193–198.

[33] M. Kálmán, “Versatile form validation using jSRML,” *Acta Cybernetica*, 2014 (Accepted for publication).

[34] M. Kalman, “Rule-based web service validation,” in *Proceedings of the 21st International Conference on Web Services (ICWS)*, Alaska, USA, June 27 - July 2 2014 (Accepted for publication), IEEE Computer Society.

[35] M. Kálmán and F. Havasi, “Enhanced XML validation using SRML,” *International Journal of Web & Semantic Technology (IJWeST)*, vol. Volume 4, no. October, pp. 1–18, 2013.

[36] M. Kálmán, F. Havasi, and T. Gyimóthy, “Compacting XML documents,” in *Journal of Information and Software Technology*, vol. 48, no. 2. Elsevier, February 2006, pp. 90–106.

[37] J. Larmouth, *ASN.1 Complete.* ISBN: 978-0-122-33435-1: Academic Press, November 1999.

[38] A. Lavrik, “PIQI.” [Online]. Available: http://piqi.org/doc/piqi

[39] C. League and K. Eng, “Schema-Based Compression of XML Data with Relax NG,” *JCP*, vol. 2, no. 10, pp. 9–17, 2007.

[40] Z. Li, Y. Jin, and J. Han, “A Runtime Monitoring and Validation Framework for Web Service Interactions,” in *Proceedings of the 2006 Australian Software Engineering Conference*, (ASWEC’06), Ed., 2006.

[41] Z. Liang, “A field-oriented approach to web form validation for Database-Isolated Rule,” in *Proceedings of IEEE International Conference on Systems Man and Cybernetics (SMC).* IEEE, 2009.

[42] H. Lie and B. Bos, *Cascading Style Sheets, designing for the Web.* Addison Wesley, 1999.

[43] C. Lindley, *jQuery Cookbook.* O’Reilly Media, 2009.

[44] W. Means and M. Bodie, *The Book of SAX.* ISBN: 1-886411-77-8: No Starch Press, 2002.

[45] P. Montero, M. Hedler, and N. Kutscherauer, *Effiziente Business Rules fuür XML-Dokumente*, Heidelberg, 2011.

[46] P. Montero and J. Sieben, *Professionelle XML-Verarbeitung mit Word. WordML und SmartDocuments*, Heidelberg, 2006.

[47] NDFD. (2013) National digital forecast database. [Online]. Available: http://graphical.weather.gov/xml/

[48] D. Raggett and A. Hors, "HTML 4.0 specification," W3C, Tech. Rep., April 1998.

[49] J. Refsnes. Introduction to DTD, http://www.w3schools.com/dtd/dtd_intro.asp.

[50] L. Richardson and S. Ruby, *RESTful Web Services.* ISBN: 978-0-596-52926-0: O'Reilly Media, 2007.

[51] SeleniumHQ, "Selenium, http://docs.seleniumhq.org," 2014.

[52] J. e. a. Skinner, "Fluent validation api," http://fluentvalidation.codeplex.com, Tech. Rep., 2013.

[53] Springsource, "Spring web services (spring-ws)," http://projects.spring.io/spring-ws/, Tech. Rep., 2014.

[54] Sun Microsystems, "Java Architecture for XML Binding (JAXB)," 2009. [Online]. Available: http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[55] SurveyMonkey, "Surveymonkey online surveys." [Online]. Available: http://www.surveymonkey.com

[56] A. Tjang and F. Oliveira, "Model-Based Validation for Internet Services," in *28th IEEE International Symposium on Reliable Distributed Systems*, 2009, pp. 61–70.

[57] W. Underwood, "Grammar-Based Specification and Parsing of Binary File Formats," *The International Journal of Digital Curation*, vol. 7, no. 1, 2012.

[58] E. van der Vilst, *Schematron.* ISBN: 9780596527716: O'Reilly, 2007.

[59] E. van der Vlist, *XML Schema.* O'Reilly, 2001.

[60] XMLBlueprint. (2002) Well-formed and valid XML Documents. [Online]. Available: http://www.xmlblueprint.com/help/html/topic_118.htm