Validating Documents of Web-based Metalanguages Using Semantic Rules

Summary of the Ph.D. Dissertation

Miklós Kálmán

Supervisor:

Prof. Tibor Gyimóthy
Head of the Software Engineering Department

Doctoral School of Computer Science
Department of Software Engineering
University of Szeged

Szeged
2014

Introduction

Validation has earned a solid place in the ever-growing world of information exchange. Systems are sharing roughly 1,000 petabytes every day. It is vital that data integrity is achieved and maintained throughout the life of the data. The need for data validation ranges from web forms through document exchange, engulfing the area of database records all the way to web service transmissions. The most common text-based format for information exchange is carried out with the help of XML[6] Documents. This format is text-based and is capable of describing hierarchic relationships. Many languages and formats are based on XML as it is easy to extend. In an earlier article [15] we presented a way to define semantic rules to compact XML files with the help of our SRML metalanguage. The initial versions (1.0, 1.1) of this metalanguage were aimed at making XML documents more compact by removing attributes that could be re-calculated using semantic rules.

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.	[14]
Validating Web Forms	Create a new jSRML metalanguage, which is capable of defining semantic rules for the validation and correction of web forms.	[12]
Validating Google Protocol Buffers	Introduce a new metalanguage (ProtoML), which can validate and correct the messages of Google Protocol Buffers.	[11]
Validating Web Services	Combine the previous metalanguages (SRML 2.0, jSRML, ProtoML) into SRML 3.0 and provide a way to validate Web Services.	[13]

Table 1: Theses of the dissertation

The dissertation describes the evolution of the SRML language along with two additional metalanguages, jSRML[12] and ProtoML[11], which were branched from the initial SRML specification to provide validation. The jSRML language is aimed at providing a way to describe validation rules for web forms, while ProtoML provides a way to define validation logic for Google Protocol Buffers[7]. The dissertation also covers a fourth area of validation: web services. We demonstrate how combining the languages and experience gained throughout the research yielded the latest SRML 3.0[13] version. Using the new extension, it is possible to validate both requests and responses of web services. We will detail each extension in this document. The high level overview of the SRML versions can be seen in *Figure 1.* The four theses of the dissertation can be seen in *Table 1.*

1 Validating XML Documents

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

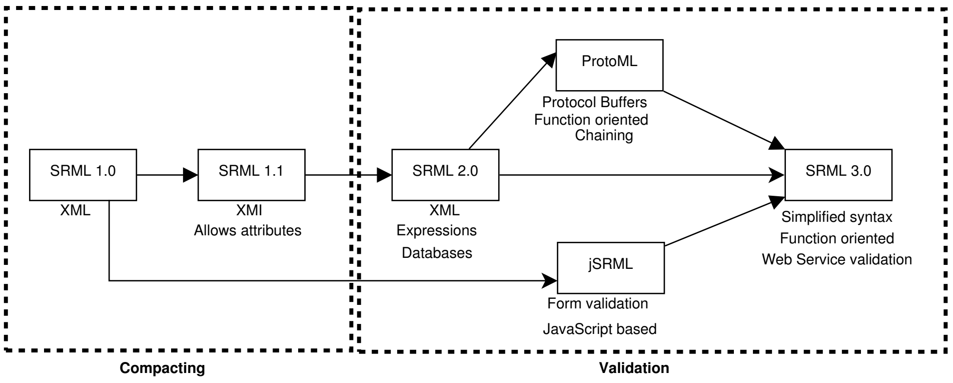

Figure 1: Evolution of SRML versions and their areas

The first topic of the dissertation is XML validation. The most common ways to describe the structure of an XML is using a DTD[17] file or an XSD schema[18]. The latter one is more advanced as it permits the description of structural elements, their types and allowed element definitions. XSD schemas are powerful yet lack an important aspect: content validation. To describe the content restrictions of an XML file, one must use third party solutions. To provide a rule-oriented approach, we decided to extend our original SRML 1.0 metalanguage to the validation space. *Table 2* outlines the key differences between the two versions of SRML.

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 2: Key differences between SRML 1.0 and 2.0

The new version of SRML provides several novel features from which the following are the most prominent:

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: The rules can now reference both attributes 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.

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.

When extending the SRML language, we wanted to ensure that we can incorporate it with the XML structural validation process. Since the XSD document is used for the structural validation it was a prime candidate. By using the `appinfo` section of the XSD schema, we can provide a non-obtrusive way to store and deliver the semantic validation rules. This allows the XSD document to define both structural and content validation rules. *Figure 2* shows the validation process using XSD and SRML. To leverage the capabilities of the SRML 2.0 extension, a new validation engine was built: `SRMLXsdTool`.

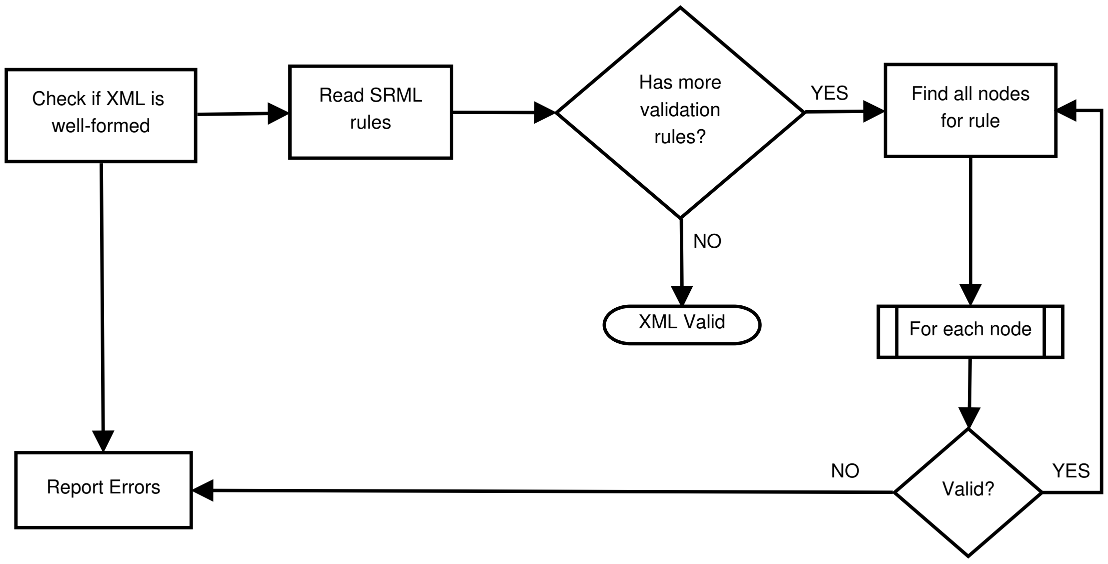

Figure 2: XML validation process using SRML

We also experimented with applying the power of SRML to the database record validation space. We provided a way to inject SRML rules into the databases and perform record validation using triggers. To accomplish this, we used an H2[2] database, which allowed the loading of Java classes as database triggers. Our validation engine was written in Java so we were able to implement trigger classes which can fire during specific database operations. The biggest challenge was how to represent database records as DOM[1] trees. DOM trees are hierarchic representations of XML documents. 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. The database record validation flow can be seen in *Figure 3*.

Another key improvement that SRML 2.0 introduced was data correction. With the new SRML extension, it is now possible to correct the content of XML documents using the semantic rules. If the mode of the rule is set to *"correct"* then the validation engine will calculate the expected value of each node that it is validating and use it as the value in case the validation fails. This provides a powerful way to correct documents, and this trait is reflected in the jSRML and ProtoML branches as well.

1.1 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

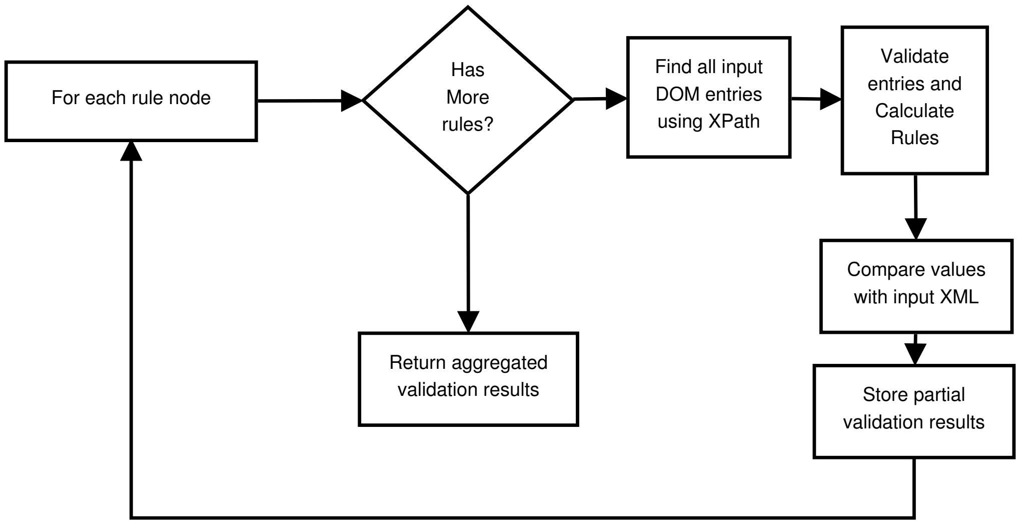
For each rule node
Has More rules?
Find all input DOM entries using XPath
Validate entries and Calculate Rules
Compare values with input XML
Store partial validation results
Return aggregated validation results

Figure 3: Validating database records using SRML

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 this thesis are my contributions as the result of my research. The ideas demonstrated in the thesis were published in [14].

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.

The second area that the dissertation covers are web forms. The Internet has engulfed most of our lives. More and more people are coming online to spend their time and do their work (e.g.: shopping, tax returns...etc). With this growth, the importance of data validation plays a vital role. Users exchange information with each other and it is very important that the resulting data is valid and is not corrupted. The most common way users communicate and enter data on the Internet is using web forms. Web forms contain fields that are filled out by the users, which are then submitted to a server for processing. The server processes this information and returns the results or performs operations on 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 system, which many hackers try to exploit. The forms are contained within HTML[16] pages, which have a similar format to XML documents so the DOM model is also applicable to them. This makes web forms an ideal candidate for the use of a semantic rule validation approach.

We have extended SRML 1.0 to create a new metalanguage called jSRML that allows the validation of web forms. It should be noted that the language was created parallel to SRML 2.0, thus the reason why that was not used as the starting point instead of version 1.0. The language constructs and syntax is similar, however, the feature set is considerably different. With our new jSRML extension, users are 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 perform all additional validation tasks automatically.

The engine supports multiple types of validation, which are summarized in *Table 3.* We took the positive traits of the original SRML 1.0 language and its 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 rule-set 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 read the validation rules.

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-time	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 3: Validation types

The second advantage of jSRML is that it is non-obtrusive. In order to use it, only a simple script include is required. If the validation rules need to change then only the affected field rules need to be updated. No coding experience is needed to perform the update, reducing the possibility of error, making it a very large benefit compared to the pure JavaScript approaches.

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 field values to be corrected and allows the form submission to succeed. A good example would be spell checking in a form prior to submission, which can be accomplished by using the 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.

The `jSRMLTool` engine supports all four types of validation described earlier (*Client*, *Server*, *Real-time*, *Hybrid*). 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 4* shows the flow of the *Client-side* validation.

- `Server-side`: The engine handles the *Server-side* mode using a separate servlet (`jSRMLTool-Servlet`). This servlet uses a unique identifier to associate the rules to each form, allowing multiple forms from different domains to be submitted/validated against the same servlet. The flow 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 5*.

- `Real-time and Hybrid`: Every rule has a *"method"* attribute, which is not mandatory 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.

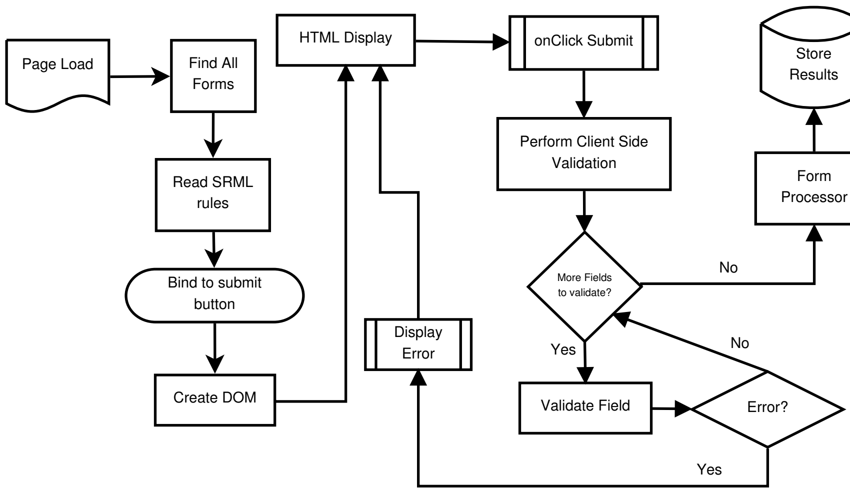

Figure 4: Client-Side jSRML

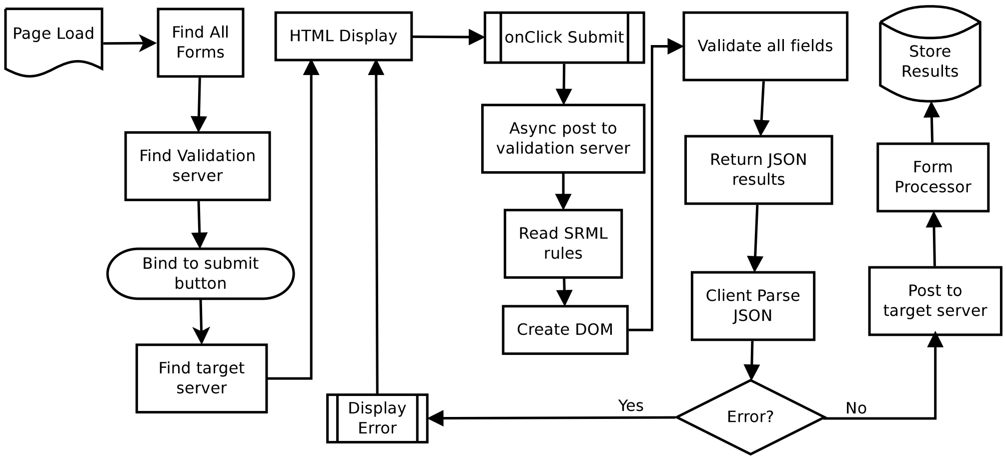

Figure 5: Server-Side jSRML

As mentioned previously, we have created a *Server-side* implementation of the jSRML engine using Java Servlets[10], 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 rule-set 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 `jSRMLToolServlet` 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. A hook will be installed on the form's submit event and will re-route the call to the `jSRMLToolServlet` location. The major difference here is that there is no actual jSRML rule-set on the server. 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.

The learning module has several plugins that process form submissions and adjust the proposed rules accordingly, making the learning a gradual process. The module will perform the learning on 50% of the available form field examples. Once the rules are proposed it will then test their efficiency on the remaining 50%. This is important to avoid over generalizing the proposed learning rules. 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 these 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, i.e. 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, weighed, then the results are analyzed and stored. *Figure 6* demonstrates how the learning module works. To increase the efficiency of the learning process, it is usually helpful to start a new rule-set with a supervised learning scenario. During this the owner of the form "teaches" the engine by providing valid sample inputs. Sometimes previous valid form submissions are also available in bulk. The tool also has an import feature which can 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. This can increase the learning efficiency of the system.

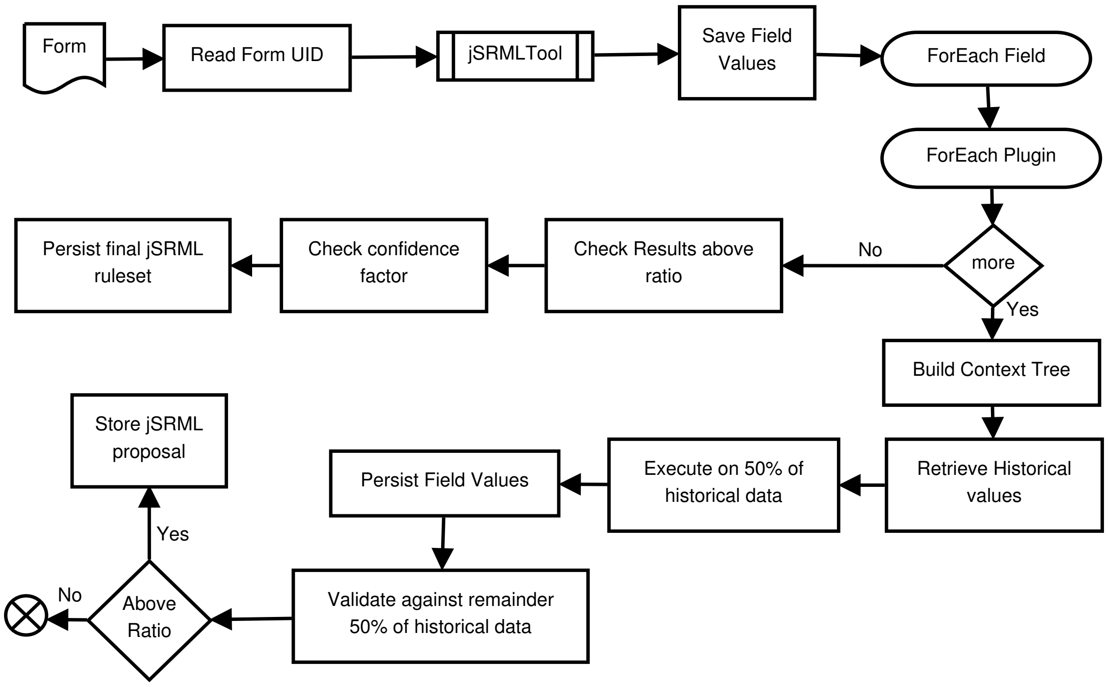

Figure 6: jSRMLTool learning process

We have evaluated the plugins programatically and also applied it to a real-world scenario. If the training examples are positive and do not contain too much noise then the learning engine is able to provide near perfect results. *Table 4* shows the experimental results of learning a form with multiple fields. It shows how increasing the positive inputs improves the learning ratio. Our *jpRelationship* plugin is also able to discover relationships between fields so it can be considered as a form of minimalistic data mining engine as well.

2.1 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.

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 4: Plugin Efficiency with gradual positive training examples

- 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 [12].

3 Validating Google Protocol Buffers

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

The thesis demonstrates ProtoML[11] as a solution to validate Google Protocol Buffers[7] (PB). This is 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 harder 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 the Google Protocol Buffers format was chosen was that it is highly versatile and has support for various programming languages. Unfortunately the validation side of the messages in PB was not part of its language specification so it also suffers from the same drawbacks as most binary-based formats. The ProtoML language is XML-based and uses functions to extend its descriptive capabilities. It allows the definition of validation and constraint rules for PB messages (using their `.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 field value based on the expected rule value if it does not match.

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 rule-set. The tool was written using the Spring Framework and uses *Exp4j*[3] to evaluate the expressions. The DOM manipulation and access is handled by the *JDOM*[9] library. The generated wrapper code no longer uses the `.pml` file and can be compiled along with the `protoc` generated Java code. This is achieved by converting ProtoML rules into chained function calls and inserting them 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 7* shows the code wrapper generation flow of ProtoML.

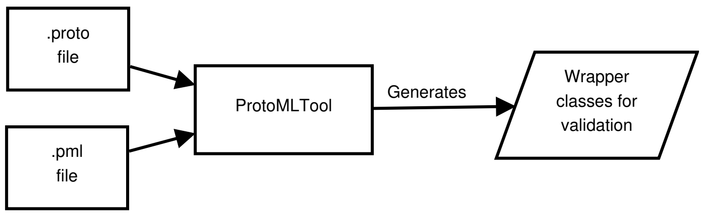

Figure 7: ProtoMLTool work-flow

The validation flow can be seen in *Figure 8*. The class that processes the PB message needs to include a reference to the generated `PMLValidator` class aside from 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* format. Since PB generates custom types and Enums, these generated types are used by the library.

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 invoked. 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.

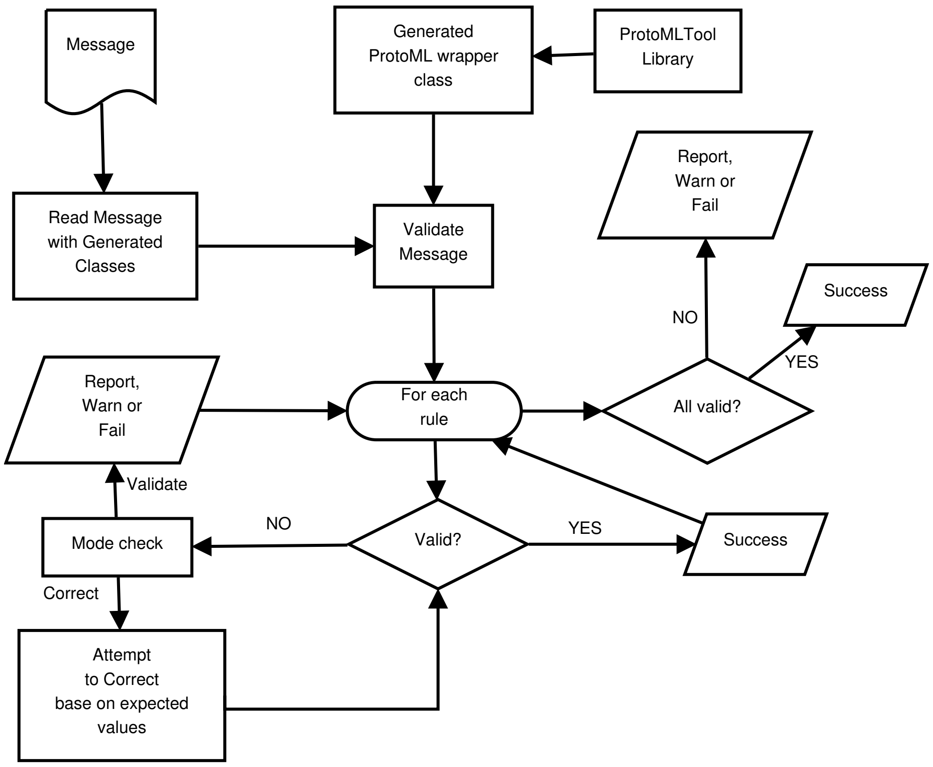

Figure 8: ProtoMLTool Validation work-flow

3.1 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 rule-set. 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 executed during runtime.

The development and implementation of ProtoML is completely the result of my research which, were published in [11].

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 space of 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. Publicly exposed services are under constant attacks [5][8] (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 the system, whereas the priorities of the client are validating the format and content of the resulting data along with integrating it into their existing infrastructure. Currently the only real 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 consumed data may need to be filtered for a subset.

Since web services can use XML to communicate with their consumers (using SOAP[4]) it was also a good choice as a target for SRML-based validation. Using the experience and knowledge gained from 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 as described in [13]. The two metalanguages helped make the new version of the language easier to use, more descriptive and contain less overhead than its predecessors. The new version along with a new implementation of the rule engine (`wsSRML`) can be leveraged to validate web services. The new SRML 3.0 format separates the expected values (under the values node) from the value/format constraints (`conditions` element). The other considerable 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.

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.

The new `wsSRML` tool allows 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.

During the validation phase both *Request* and *Response* parameters are accessible, since the function of the wrapper class is making the actual service call. This provides even more powerful validation rules since there are cases when conditions can be defined for the response-based on what the request was. The reason why both requests and responses may need to be validated is for better security and validity. Most validators only concentrate on the request side. 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. Our system also allows the errors to be corrected using the validation rules themselves, making it more than a simple validation engine.

The `wsSRML` engine has two operation modes: *native* and *proxy*. Each have their advantages and disadvantages. The flow of the native mode starts with the stub generation. After `wsdl2java` has generated the stub classes, our tool will parse the SRML rule-set and augment the code. The SRML rules are analyzed and the wrapper class is generated on top of the interface. *Figure 9* demonstrates how the native mode generates the wrapper class.

The validation wrapper class is augmented with 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 10* shows what the validation flow looks like in case of this mode.

The advantage of the *native* mode is that it is fast, provides native compiled validation that can be used in any project that needs web service validation. The disadvantage is that the business logic cannot be changed on the fly; it has to be recompiled, which may be hard in a production setting.

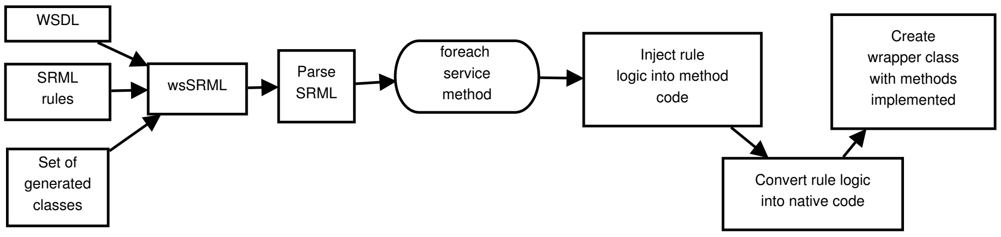
WSDL
SRML rules
Set of generated classes
wsSRML
Parse SRML
foreach service method
Inject rule logic into method code
Convert rule logic into native code
Create wrapper class with methods implemented

Figure 9: Native validation class generation

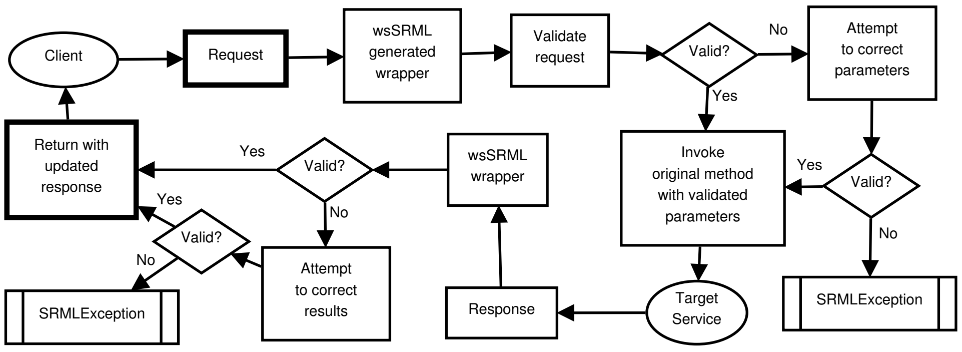

Figure 10: Native validation flow using wsSRML

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. 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. This is similar to how jSRML solved the server-side validation. *Figure 11* shows the validation flow in case of the proxy-based validation.

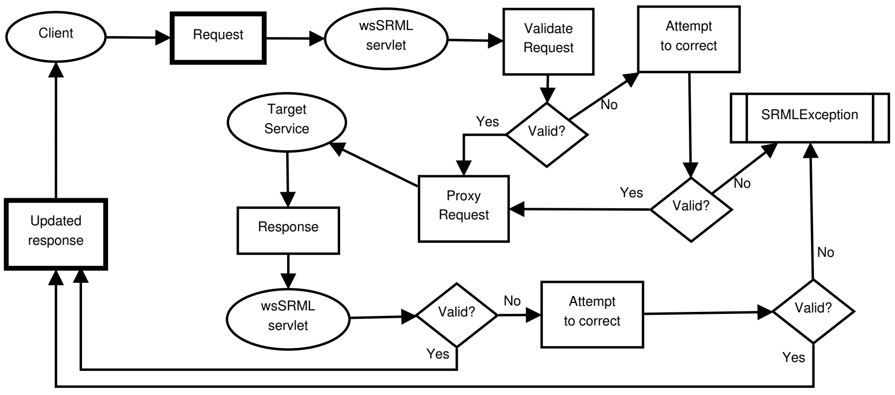

Figure 11: Proxy based validation flow

There are three operation submodes 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.

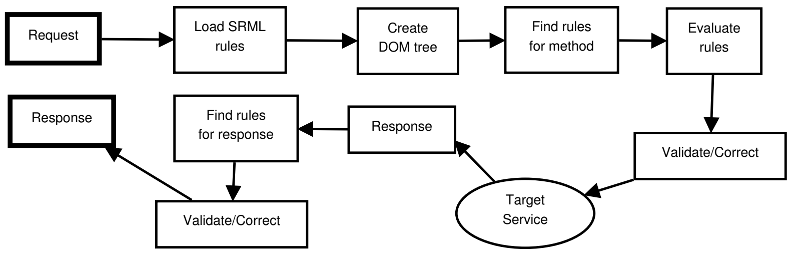

Figure 12: Real-time proxy flow

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 and applying the validation rules on them. *Figure 12* shows the real-time proxy validation flow.

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 13* shows the compiled rule plugin 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 use an approach similar to how OSGi works. The plugins are sand-boxed to their own environments and versions of the classes. The *wsSRML* proxy servlet will load all the plugin 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.

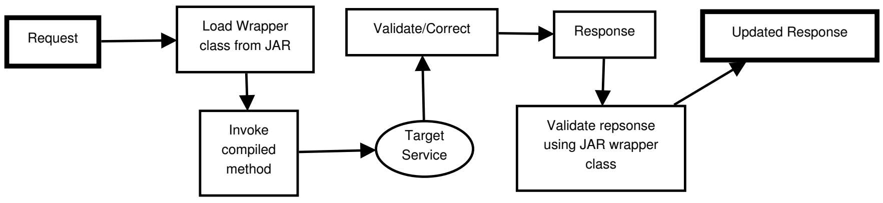

Figure 13: Compiled Plugin proxy flow

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) and the appropriate rules are executed. The speed is not the most optimal since the rule-set is parsed upon each request and response. *Figure 14* shows the SOAP interception mode of the tool.

The advantage of the proxy mode is that it is usable in situations when the client and/or server code

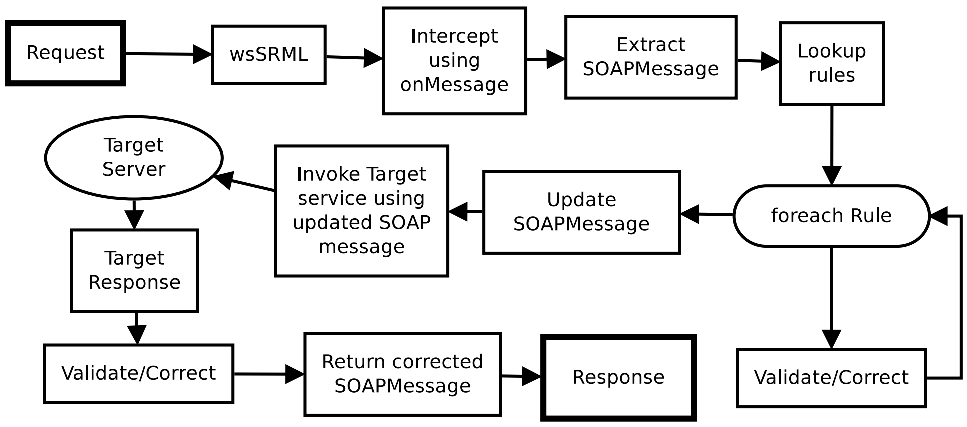

Figure 14: SOAP Intercept flow

is unavailable or cannot be modified. It also allows real-time swapping and extension of validation rules without any potential downtime. The disadvantage of this mode is that it is considerably slower than the native compiled version and it requires a proxy servlet to be deployed to perform the interception, adding an extra level of complexity.

4.1 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 [13].

5 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).

References

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

[2] H2 database engine, http://www.h2database.com/html/main.html.

[3] F. Asseg. Exp4j, http://www.objecthunter.net/exp4j/, 10 2011.

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

[5] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In LNCS, editor, *International Conference on Applied Cryptography and Network Security (ACNS)*, volume 2, 2004.

[6] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language, 1998.

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

[8] M. Handley. Internet Denial-of-Service Considerations. Technical report, IAB, RFC4732, 2006.

[9] J. Hunter. JDOM, http://jdom.org/docs/apidocs/, 2000.

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

[11] 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)*, pages 193–198, London, UK, December 9-12 2013, IEEE Computer Society.

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

[13] 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.

[14] M. Kálmán and F. Havasi. Enhanced XML validation using SRML. *International Journal of Web & Semantic Technology (IJWeST)*, Volume 4(October):1–18, 2013.

[15] M. Kálmán, F. Havasi, and T. Gyimóthy. Compacting XML documents. In *Journal of Information and Software Technology*, volume 48, pages 90–106. Elsevier, February 2006.

[16] D. Raggett and A.L Hors. HTML 4.0 specification. Technical report, W3C, April 1998.

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

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