Acta Cybernetica **21** (2014) 553–583.

Versatile Form Validation using jSRML

Miklós Kálmán*

Abstract

Over the years the Internet has spread to most areas of our lives ranging from reading news, ordering food, streaming music, playing games all the way to handling our finances online. With this rapid expansion came an increased need to ensure that the data being transmitted is valid. Validity is important not just to avoid data corruption but also to prevent possible security breaches. Whenever a user wants to interact with a website where information needs to be shared they usually fill out forms and submit them for server-side processing. Web forms are very prone to input errors, external exploits like SQL injection attacks, automated bot submissions and several other security circumvention attempts. We will demonstrate our jSRML metalanguage which provides a way to define more comprehensive and non-obtrusive validation rules for forms. We used jQuery to allow asynchronous AJAX validation without posting the page to provide a seamless experience for the user. Our approach also allows rules to be defined to correct mistakes in user input aside from performing validation making it a valuable asset in the space of form validation. We have created a system called jSRMLTool which can perform hybrid validation methods as well as propose jSRML validation rules using machine learning.

Introduction

Information exchange has become a vital part of our lives. The Internet is the key channel to provide the means to digitally exchange data between its users. The number of users hooked up to the Internet is increasing day by day. Social networking sites engulf the ether and integrate with our lives. With this growth comes an ever-increasing amount of data being transmitted. Users perform their daily tasks online, giving out information, submitting data on sites. Data integrity and security is a vital concept in this eco-system. The most common form of user initiated information exchange are web pages. These pages are written in HTML[1] and may contain web forms that consist of fields. These fields are filled out by the user, which are then submitted to the server for processing. The server then processes this information and returns the results or performs an operation with the

*University of Szeged, Department of Software Engineering, Dugonics tér 13., H-6720 Szeged, Hungary, +36 70 3684910, email: mkalman@inf.u-szeged.hu

DOI: 10.14232/actacyb.21.4.2014.3

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 as many hackers try to exploit sites through their forms. The most common form of attacks against web forms is *DoS*[2] (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*[3]. 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). Aside from 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*[4]. 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[5] call).

To provide a solution to these issues we have created a jQuery[6] based validator called jSRMLTool which leverages the SRML[7] language we introduced in one of our earlier articles. This language was extended to allow form based validation rules. The original SRML specification targeted XML document compaction and

decompaction. 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 chapter of this article.

An off-site asynchronous implementation of the jSRML engine was also created using Servlets capable of validating forms using 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 provision to learn potential jSRML rules using the submitted form data and machine learning.

We will start out by providing some basic background on the technologies used throughout the article. We then continue on to show the extension made to the SRML language that allow for the definition of form validation rules. Afterwards we will demonstrate the potential of learning jSRML rules using the jSRMLTool servlet and evaluate the results. We end the article by an analysis of related work in this field finishing off with a summary and our plans for future expansion.

1 Preliminaries

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

1.1 HTML and DOM

Forms are described using the HTML[1] 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*[8] (Document Object Model). *Figure 1* shows a simple HTML form source with a field. The DOM tree representation of *Figure 1* is shown in *Figure 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 *Figure 3*.

```
<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 1: Simple HTML of form

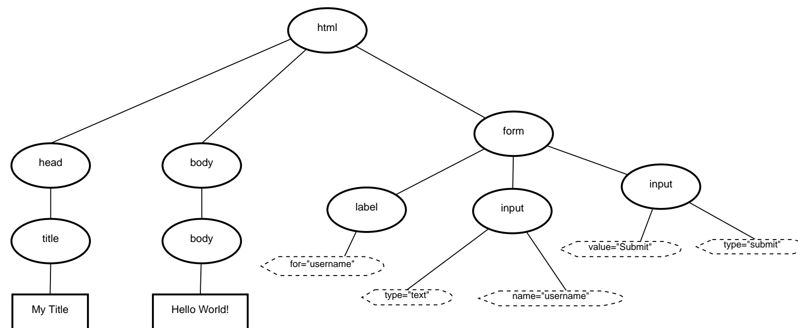

Figure 2: DOM tree of the Form Example

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 realtime prior to form submission	More traffic required, harder to update
Hybrid	Field change and Submit	Either	Direct calls with roundtrip to server	Allows two stage validation, pre-filtering results prior to sending to server	More complex to implement and maintain

Figure 3: Validation types

1.3 SRML

The SRML[9] metalanguage was introduced to allow the description of semantic rules that can be used to compact and decompact XML[10] documents. The term

compaction comes from the fact that it is able to remove specific attributes based on rules and can recreate the same value (therefore restoring) at any later time. The original SRML rule engine implementation used the DOM tree of the XML to perform its operations. Since HTML forms can be considered as DOM[8] trees it made sense to attempt to apply SRML to this area as well. In this article 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 similar 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. *Figure 4* 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

Figure 4: Key differences between SRML versions

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 engine 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 [11].

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 aside from the standard operation set that the engine provides.

We have also created a *Server-side* implementation of the jSRML engine using Java Servlets[12] 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 extendable 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.

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 5* shows a simple example of providing an email validation rule using *in-line* jSRML.

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 expressions. XPath[13] is a query language allowing the easy access and manipulation of nodes and their content within a DOM tree.

```
...
 <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 5: jSRML snippet for in-line email validation

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.

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 predefined 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 article 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[14] 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, like 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.

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 6*. This form has multiple fields to better demonstrate how jSRMLTool works. The full source of the page can be found in [15]. 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 7* 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 8*.

- `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
Submit Form	

Figure 6: Input form

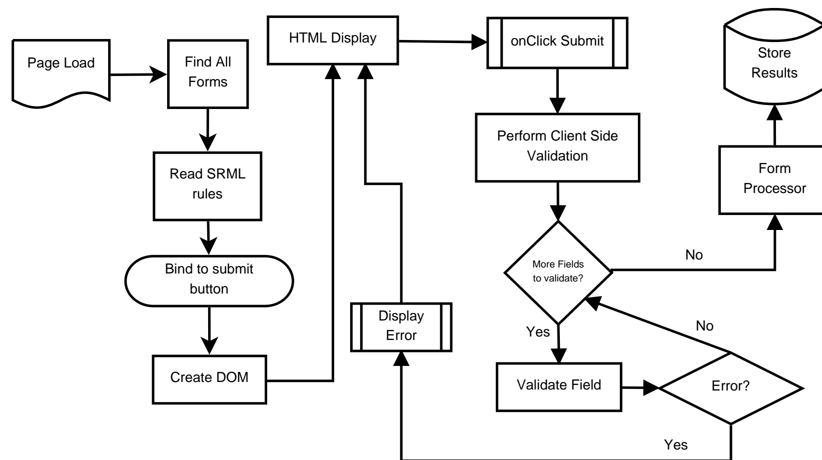

Figure 7: Client-Side jSRML

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

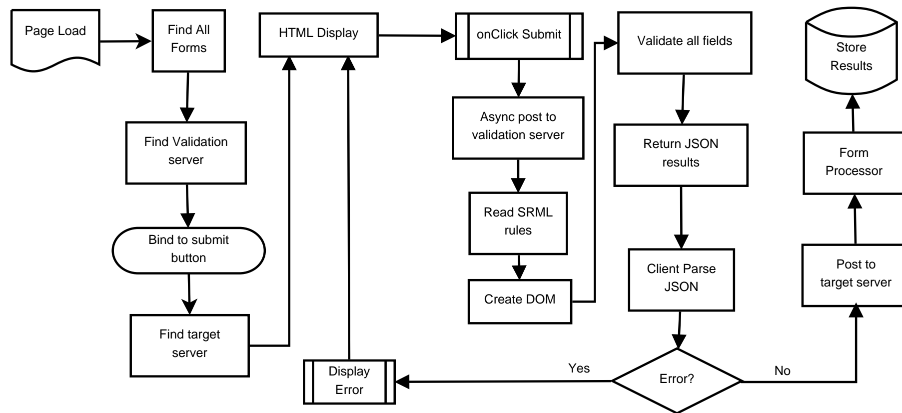

Figure 8: Server Side jSRML

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 servers. 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.

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 9 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.

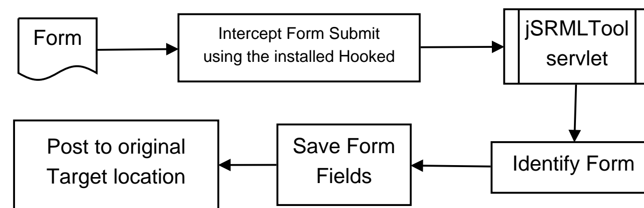

Figure 9: Intercepting form data and learning jSRML rules

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.* We will detail each learning plugin in this section.

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 10* 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. Sometimes previous valid form submissions are also available in bulk. 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. This can increase the learning efficiency of the system.

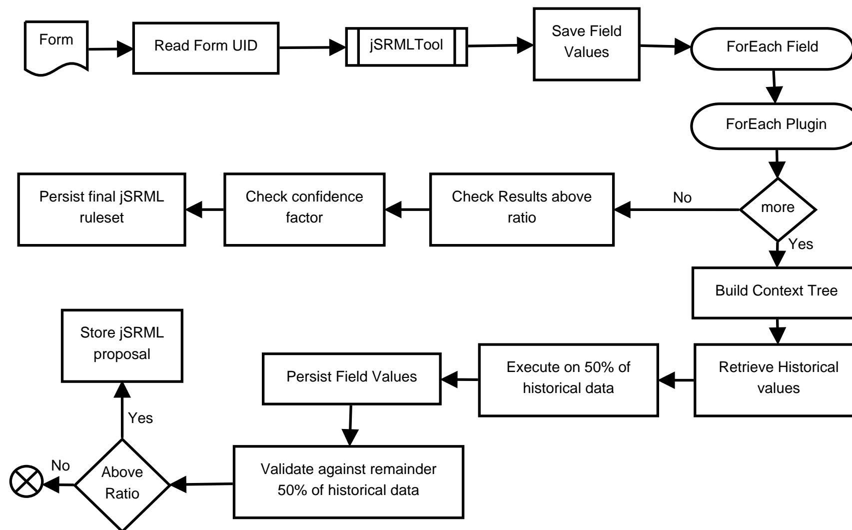

Figure 10: jSRMLTool learning process

4.1.1 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.

4.1.2 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.

4.1.3 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 retype 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.

4.1.4 jpRelationship

The relationship plugin is aimed at finding relationships between fields and their values. The steps of the plugin are demonstrated in *Figure 11.* 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 12* shows the compositional method of the plugin.

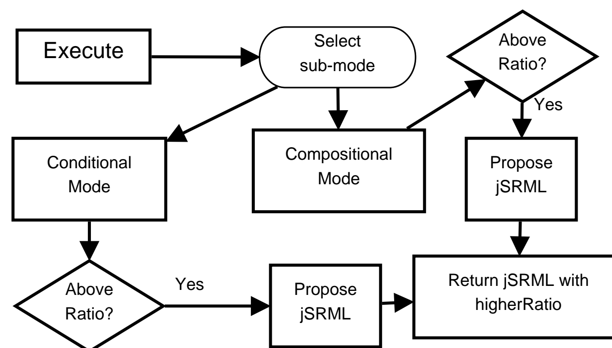

Figure 11: jpRelationship Plugin

The second mode of the *jpRelationship* plugin is the *conditional* mode (*Figure 14*). This method finds relationships between field values using conditional logic and applying statistical machine learning[16]. The plugin uses 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

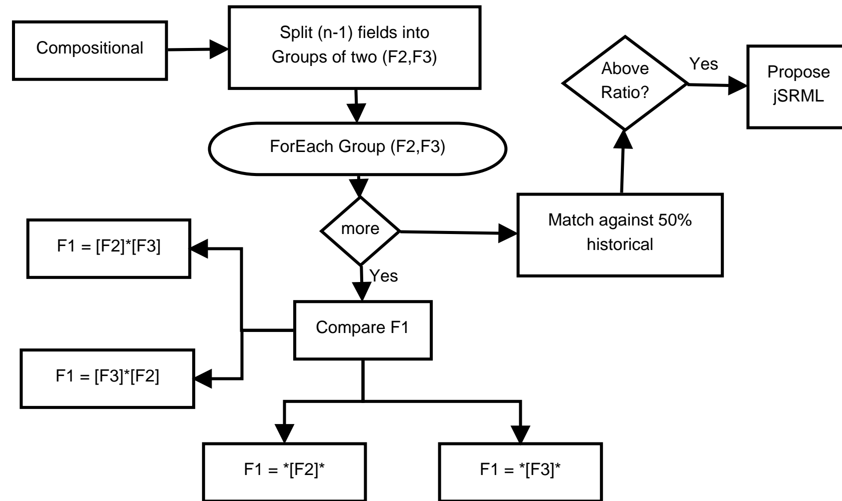

Figure 12: jpRelationship Compositional Method

that are focused on learning F_k using a simplified Random Forest[17] 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

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 13: Outdoor Activities Form

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*[18]. The fields *wind* and *weather* allow multiple values to be selected (the form can be seen in *Figure 13*). 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 15* shows the resulting tree for *activitity* (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 descriptive 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.

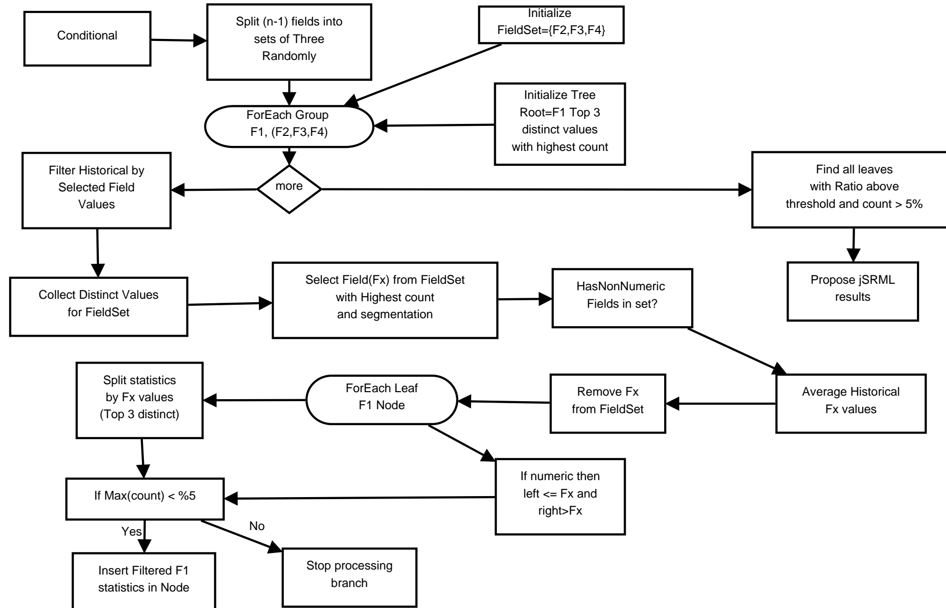

Figure 14: jpRelationship Conditional Method

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 remainder 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 remainder two classifier 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 remainder 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 remainder 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 mis-classify 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.

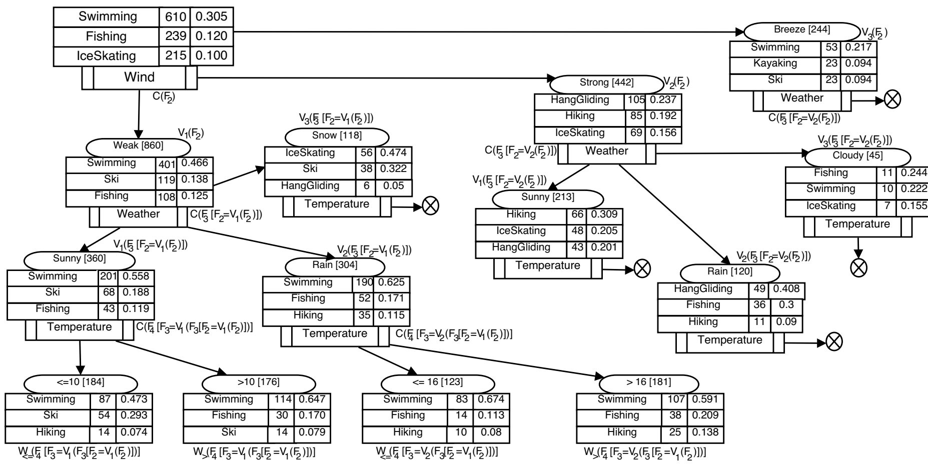

Figure 15: Sample tree in the Random Forest

4.1.5 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.

4.1.6 jpRegExp

The regular expression 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 remainder 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 [19]. 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.

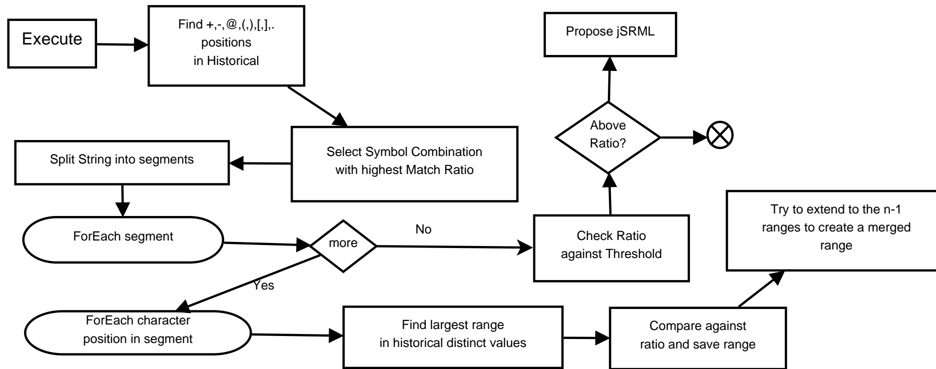

Figure 16: jpRegExp Plugin

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 *Figure 17* 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 would 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 article.

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.

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 %

Figure 17: Plugin comparison (ES=Empty Set, PS=Primed Set)

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 remainder *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 remainder *15,000* inputs are selected and executed using the rules. During the processing of the remainder inputs the engine 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 *Figure 17*. Based on the results it is visible that using *Primed* Sets yields the most effective results. From the plugins *jpFormat*, *jpLength* and *jpRange* yield the best results. 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 are 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 *Figure 18*. 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.

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 %

Figure 18: Plugin Efficiency with gradual positive training examples

4.3 A Real-world example: Dentistry Treatment Enquiry 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. In the earlier example we only demonstrated the engine use for the *jpRelationship* plugin. During this test more plugins of the engine were verified as a whole. We chose [20] which is a site targeted at capturing leads for international clients who are enquiring 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 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*[21] (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

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`

Figure 19: Plugin results for Dentistry Contact form

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 *Figure 19*.

The experiment yielded in providing validation rules based on the results visible in *Figure 19*. 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 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).

5 Related Work

There have been several advances and research done in the field of form validation. In this section we will mention a few along with how the approaches handle validation. The first paper we would like to mention is [22]. 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 good 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 [23]. This article 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 [24]. This 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 [25] 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 does bare some similarities to 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 provide the definition of more complex validation rules.

The points discussed in [25] are aimed at server-side validation and are valid for most web forms. The article does however suggest 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 [26]. 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 [27] 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.

6 Summary and future work

In this article we introduced the jSRML metalanguage and engine. This is a major extension to the SRML language specification to allow 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 allows 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 also helps in the generation of jSRML rules using machine learning. 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.

In the future we plan to simplify the rule definitions simplifying the syntax and by providing more out of the box types (to avoid longer binary-op conditions). We also plan to extend the servlet service to allow easier updating and creation of rules for server-side validation alongside enhancing the learning module to provide more powerful and efficient rules. We plan to investigate the option to generate test data based on the rules defined in the form file to help test driven development as well as exploring other languages aside from Java for the library implementation.

References

[1] Raggett, D., Hors, A. L. and Jacobs, I., 1998, *"HTML 4.0 specification," W3C*, http://www.w3.org/TR/REC-html40/

[2] Handley, M., 2006, *Internet Denial-of-Service Considerations*, IAB, RFC4732

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

[4] Crockford, D., 2008, *Javascript: The Good Parts.* O'Reilly, 2008.

[5] Garret, J.J, 2005, *Ajax: A New Approach to Web Applications*, http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[6] Lindley, C., 2009, *jQuery Cookbook.*, O'Reilly Media

[7] Havasi, F., 2002, *XML Semantics Extension*, Acta Cybernetica Vol 15 No. 2, pages 509-528

[8] Hégaret, P., 2005, Document Object Model (DOM), W3C, http://www.w3.org/DOM/

[9] Kálmán, M., and Havasi, F. et al, 2006, *Compacting XML documents.*, Journal of Information and Software Technology, Volume 48 Issue 2, February 2006, pages 90-106

[10] Bray, T., Paoli, J. and Sperberg-McQueen, C., 1998, *Extensible markup language*, XML 1.0 W3C recommendation, http://www.w3.org/TR/REC-xml

[11] Kálmán, M., 2013, *The complete XSD of jSRML* http://www.srml-language.com/jSRML/jSRML.xsd

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

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

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

[15] Kálmán, M., 2013., *The complete HTML source of the example* http://www.srml-language.com/jSRML/jSRML-example.txt

[16] Hastie, T., Tibshirani, R. and Friedman, J., 2001, *The Elements of Statistical Learning*, Springer. ISBN 0–387–95284–5.

[17] Breiman, L., 2001., *Random forests.*, *Machine Learning*, 45:5–32, 2001.

[18] *SurveyMonkey Online Surveys*, 2013, http://www.surveymonkey.com

[19] Fernau, H., 2009., *Algorithms for learning regular expressions from positive data*,Inf. Comput., Volume 207, Number 4, pages 521–541, Academic Press, Inc.,Duluth, MN, USA

[20] Beauty and Confidence, 2013, *Dental Implant Abroad Booking page* http://www.dental-implantabroad.co.uk/ental-implant-overseas/

[21] SeleniumHQ, 2013, http://docs.seleniumhq.org/

[22] Liang, Z., 2009, *A field-oriented approach to web form validation for Database-Isolated Rule*, Man and Cybernetics, SMC 2009. IEEE International Conference on Systems, 11-14 Oct. 2009, pages 4607–4612

[23] Escott, E., Strooper, P., et al, *Model-Driven Web Form Validation with UML and OCL*, 2012, Lecture Notes in Computer Science Volume 7059, 2012, pages 223-235

[24] HansMartin, A., 2010, *Form validation with Rule Bases* http://blog.mgm-tp.com/2010/10/test-data-generation-part1

[25] Saha, T.K. and Ambia, A., 2013, *Code Generation Tools for Automated Server-side HTML form Validation*, International Journal of Computer Science and Management Research, Volume 2, Issue 1, 2013, pages 1265–1271

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

[27] Giannoudis, J., 2012., *Rule Based Validation for ASP.NET* http://www.codeproject.com/Articles/367214/Rule-Based-Validation-for-ASP-NET

Received 23rd May 2013