PMON: Perl Meta Objects Notation for perfSONAR-PS services

Authors	Maxim Grigoriev
Date	01-01-2008
Current Version	1.0.1

Document Change Log

Document			
Version number	**Date**	**Description of change**	**People**
1.0.1	01-01-08	Added more details	Maxim Grigoriev
1.0	27-12-07	First draft issued	Maxim Grigoriev

Table of Contents

1. Introduction

This document presents and describes API designed for binding perfSONAR Relax NG or XML into the perl class library. The main idea behind such binding is to provide flexible and extendable API which will allow developers of the perfSONAR perl based services easily describe any service interface by writing simple perl data structures. This approach will let you to work with data from perfSONAR XML messages using your own class structures. I call it Perl Meta Objects Notation (or PMON) because its very close to the same idea as JSON[1] (in this case as expressing more complex markup grammar by utilizing some basic language constructs). The PMON framework will handle all the details of converting data to and from XML or DOM objects based on your instructions. PMON was designed to allow you a high degree of control over the translation process with custom callbacks and SQL mapping of the hierarchical XML entities on the flat SQL database schema. Even more importantly to note that it creates heavily documented, easy readable, highly supportable and maintainable perl class hierarchies.

2. PMON framework

2.1. Why another framework

PMON uses the power of perl data structures to define the rules of how perl objects are converted to or from XML (the *binding*). Or more explicitly to and from the DOM tree. Why not to use just DOM instead? DOM is just a representation of the XML data structure. It does not provide any facility to hook up easily some custom callback aligned with some external data schema to the particular element in the tree. The current spectrum of available XML APIs for perl is limited to tree walking. The PMON framework is an attempt to bring event based callbacks into the DOM and add some functional validation inside of such binding as well as provide mechanism to map any arbitrary SQL database schema to the hierarchical structure of the DOM tree. Importance of such mapping technique is exposed by the lack of robust support for native XML data type among the current list of "freeware" storage engines where the SQL RDBM is more or less a standard. Also, PMON uses perl as meta-language and it uses perl to create perl OO API. It eliminates tedious job of describing each XML element by another language structures. Any developer can concentrate on implementing semantic rules, protocols and actual functionality instead of wasting any time on following XML or Relax NG schema syntax. Moreover, there is a auto generated test suite for the each class of the generated API and every method call is documented. The internal structure of every class is clean and it was designed to stay clean, readable and easily understandable by any other perl developer. It utilizes almost every known of the perl's best practices [2]

and uses *Class::Fields* and *fields* packages to provide tighter encapsulation and *Class::Accessor* to provide complete list of accessors and mutators. Behavioral semantic of the each class in the class tree is the same and implements absolutely the same callbacks to allow transparency and propagation of the implemented callbacks across the API.

2.2. Element definition

The basic XML element definition is represented as perl's hash reference:

<element-variable> = {'attrs' => { <attributes-definition>, ' xmlns' ' => ' namespace-id },

'elements' => [<elements-definition>],

'text' => <text-content>,

'sql' => {<sql-mapping-definition>}

}

Where each *<xxxx-definition>* can be expressed in EBNF[3] as

<attributes-definition> = (attribute-name '=>' attribute-value) (',' <attributes-definition>)*

attribute-name = string

namespace-id = string

attribute-value = 'scalar'| ('enum:'(attribute-name (',' attribute-name)*))

<elements-definition> = ('[' element-name '=>' (

<element-variable>|

'['<element-variable>']'|

'['(<element-variable>',')+']'|

'[' ('[' <element-variable> ']' ','?)+ ']'),

, conditional-statement? ']')*

conditional-statement = ('unless'|'if'):'(registered-name (',' registered-name)*)

registered-name = attribute-name|element-name

element-name = string

<text-content> = 'scalar'|conditional-statement

<sql-mapping-definition> = (sql-table-name '=>' '{' sql-table-entry '}') (',' <sql-mapping-definition>)*

sql-table-entry = (sql-entry-name '=>' '{' entry-mapping '}') (',' sql-table-entry)*

sql-table-name = string

sql-entry-name = string

entry-mapping = 'value' '=>'(element-name|('[' element-name (',' element-name)+ ']')) (',' if-condition)?

if-condition = 'if' '=>' attribute-name ':' attribute-value

In the attributes definition the *'xmlns'* attribute is reserved for the namespace id .This id must be registered within *perfSONAR_PS::Datatypes::Namespace* class. This is just an id, if it will be changed in the future then actual namespace URI can be used instead. There is some conditional logic allowed. An attribute value can be *'scalar'* which stays for perl string or enumerated list of allowed names.

The list of sub-elements is represented as array reference. The choice between several possible sub-elements is introduced as array reference to the list and having multiple sub-elements of one kind is represented as reference to array consisted with single element variable of that kind. For example:

- *elements => [parameter => [$parameter]]* - defines list of 'parameter' sub-elements
- *elements => [parameter => $parameter]* - defines single 'parameter' sub-element
- *elements => [parameters => [$parameters,$select_parameters]]* - defines choice between two single 'parameters' sub-elements of different type
- *elements => [datum => [[$pinger_datum],[$result_datum]]]* - defines choice between two lists of 'datum' sub-elements of different type

In elements definition the third member is an optional conditional statement which represents validation rule. For example *'unless:value'* conditional statement will be translated into the perl's conditional statement ***&& !($self->value)*** where *'value'* must be registered attribute or sub-element name and this condition will be placed in every piece of code where perl object is serialized into the XML DOM object or from it.

2.3. perfSONAR data model

The root of the perfSONAR schema and the root of the OO API built by PMON is the Message object. It exists in the perfSONAR base namespace identified by *nmwg* id. The schema is versioned. The most current root package name for Message class is *perfSONAR_PS::Datatypes::v2_0::nmwg::Message*.

The base schema is completely defined by *perfSONAR_PS::DataModels::DataModel* module. This is a simple perl package and not the class, because it has only data definitions. The current DataModel has implemented definitions from *filter.rnc, nmtopo.rnc, nmbase.rnc, nmtm.rnc, nmtl3.rnc, nmtl4.rnc* and brought it together. There is no SQL mapping definitions in the base data model allowed, because SQL mapping is service specific. Any service specific extension of this base schema must be extended in the separate data model package as it was done for PingER service. The PingER data model can be viewed as an example and it is contained in the *perfSONAR_PS::DataModels::PingER_Model*. Any other extension data model package can be created for any other service.

Let's look on example of the ***parameter*** element. It is described in the base model as:

```
$parameter   = {attrs  => {name => 'scalar',  value => 'scalar', xmlns => 'nmwg'},

                    elements => [],

                    text => 'unless:value',

                };
```

That means the parameter element has *name* and *value* attributes, does not have any sub-elements and might have optional (only when value attribute is undefined) text content. For PingER service it was extended as:

```
 $parameter->{'attrs'} = {name =>
'enum:count,packetInterval,packetSize,ttl,valueUnits,startTime,endTime,deadline,t
ransport,setLimit',

                                value => 'scalar', xmlns => 'nmwg'};
```

Where explicit enumeration was added for the list of allowed values of the *name* attribute and SQL mapping was defined as:

```
  $parameter->{sql} = {metaData => {count => {value =>  ['value' , 'text'],

                                                    if => 'name:count'},

                                   packetInterval=> {value =>  ['value' , 'text'],
```

```
                    if => 'name:packetInterval'},
                packetSize=> {value => ['value' , 'text'] ,
                    if => 'name:packetSize'},
                ttl=> {value =>  ['value' , 'text'],
                    if => 'name:ttl'},
                deadline => {value =>  ['value' , 'text'] ,
                    if => 'name:deadline'},
                transport => { value =>   ['value' , 'text'],
                    if => 'name:transport'},
            },
        time    =>    {     start => {value => ['value' , 'text'],
                    if => 'name:startTime'},
                end  =>   { value =>  ['value' , 'text'],
                    if => 'name:endTime'},
            },
        limit   =>  {     setLimit => { value =>  ['value' , 'text'],
                    if => 'name:setLimit'},
            },
    };
```

Please note that **time** table name was made up to allow mapping of the time range related entities**.** Also, **limit** table name was made up to provide non-existed paging mechanism in order to limit the size of the requested dataset in the response message. As it clear from the example above, the schema can be modified in any time without affecting the rest of the service API. For example extra attribute *'type'* can be added and the whole API framework rebuild in the matter of seconds. Then developer can utilize this extra attribute *type* by calling ***->type*** on the parameter object in the message handling class and new functionality will be supported in the matter of minutes.

2.4. API class introspection

In order to provide the same class interface throughout the API, the same universal constructors and same set of method calls were implemented. Currently the constructor body includes initialization part as well and this part is slightly different for different modules, but it will be reduced to generic constructor and some custom initialization part in the future where generic constructor might be inherited from some root Object class. Currently every constructor "knows" how to initialize the whole class from the DOM object, XML string fragment or from the reference to the perl hash structure of the named class fields. Also, it "knows" how to handle arrays of the class fields with support for elements identified by **id** or **metadataIdRef.** It supports special fields, named *idmap* and *refidmap* for that purpose. It "knows" how to serialize object of the class into the DOM or XML text string and how to map contents of the object on some SQL schema. By issuing *registerNamespaces* call to the root object one can get the hash with all namespaces utilized by every sub-element in the object. It utilizes another special filed, called *nsmap* for that. This field is an object of the *perfSONAR_PS::Datatypes::NSMap* class as a container for map between local names and namespace ids. Also, every class supports method named *merge* and its able to merge the object of any class to the another object of the same class recursively.

2.5. SQL mapping

Lets look again on the example of the SQL mapping definitions for PingER service from chapter 2.3. Every object of the API class tree implements querySQL call. It accepts reference to the empty hash and passes this reference through the whole objects tree while filling contents of the hash with initialized contents of the "objects of interest". The "object of interest" defined by that SQL mapping definition. For example, for definition:

{metaData => {count => {value => ['value' , 'text'], if => 'name:count'},

for existed Parameter object with name set to *'count'* and *'value'* or text content set to *'100'* it will add metaData => {*'count' => { 'eq' => '100'}}* into the hash. The resulted hash for *metaData* table will be passed as argument to the SQL query provided by *Rose::DB::Object* API.

2.6. Building API classes

The whole API building process is coded inside of single module called *perfSONAR_PS::DataModels::APIBuilder*. This module exports several configurable parameters and two function calls, namely *buildAPI* and *buildClass*. The buildAPI is the only function needed to build whole API. Also, the list of parameters includes:

- **$DATATYPES_ROOT** - top directory name where to place versioned datatypes API, its Datatypes by default
- **$API_ROOT** - root package name for the API , perfSONAR_PS by default
- **$TOP_DIR** - top directory where to build API (this will be part the very top of the API directory tree, lib by default)

- **$SCHEMA_VERSION** – to distinguish different schema versions, v2_0 by default
- **$TEST_DIR** - top directory where to build tests files

In order to provide these configurable parameters the service specific script was created for PingER service under the name ***util/pinger_schema.pl***. For the perfSONAR PingER service with SQL storage it will build the whole OO API in the directory of choice as well as automated test suit. As an option, it is able to build related database objects if database is created and tables are present. It uses Rose::DB[4] module from CPAN for dynamic loading of the database tables.

2.7. utilizing PMON for perfSONAR-PS

The major utilization scenario of the PMON framework comes when there is a need to build a service with support for SQL database based storage engine. In that case PMON framework will provide the complete solution and the rest of development for Measurement Point will be contained in writing actual MP agent class which will be doing the real time measurements and then utilizing PMON to inject measured data into the published XML message. For Measurement Archive the whole development will be dedicated to writing the actual message handler and implementing actual protocol specifications. For example, for PingER MA it is *perfSONAR_PS::Datatypes::PingER* class, inherited from *perfSONAR_PS::Datatypes::Message* class with actual implementation of SetupData request and MetaDataKey request handlers. The *perfSONAR_PS::Datatypes::Message* is an abstract class, extending *perfSONAR_PS::Datatypes::v2_0::nmwg::Message* class. This extra class exists because the root Message class from the PMON is message type agnostic.

3. References

1. JSON, http://www.json.org
2. Perl Best Practices, by Damian Conway
3. EBNF - http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form
4. Rose::DB, http:://search.cpan.org/dist/Rose-DB/