Nilus
an XML Generator Interface for Tethys

The river god Nilus, son of Oceanus and Tethys. Statue in Vatican City.

Tethys 3.1 Release

Jeff Cavner – San Diego State University
Sean Herbert – Scripps Institution of Oceanography
Marie A. Roch – San Diego State University

Contents

1. Overview

The Nilus package is an application programming interface (API) library that can create documents that can be stored in Tethys. These documents are formatted using the conventions of extensible markup language (XML), a specification language used by Tethys to store data in a consistent manner.

Figure 1 – Nilus on Coptic fabric (Egypt, 4th century, Wikipedia)

Nilus can generate documents for any of the schemata that Tethys uses to store data. This library is designed for developers who wish to add the capability to generate Tethys-ready outputs to their software. Nilus contains functions to create and populate data fields, and then generate an XML document from those data.

The Nilus library is written in Java. If the program that will be using Nilus is written in Java, Nilus's Java archive (jar) files can be used directly. Many other programming languages (e.g. Matlab, R, C++) can work with jar files as well. This makes it relatively straight forward to add Tethys output functionality to detection, classification, and localization packages, or to write software that translates the output of existing packages. (Some users may prefer to use Tethys's translation,

or source, maps which provide a non-programmatic way to import data into Tethys from existing databases, spread sheets, or comma separated value files.)

In this manual, we will concentrate on showing how to generate documents for describing detections of animals, but the methods described here are appropriate for any of the Tethys data types. The use of Nilus requires that the detector source code be modified. Specifically, it is modified to make a series of API function calls describing the effort that the detector is undertaking. Further calls are made to create detection elements and specify the characteristics of each detection. Typical characteristics include start time, species and call type, but can include measurements of calls such as received level, signal to noise ratio, peak frequency. Custom measurements can also be made, permitting Nilus to store researcher specific observations in Tethys. Calls to generate detection elements can be made as each element is detected or in a batch mode once all detections have been processed, although this would require the application to store the detections internally. Regardless of the method used, the concept remains the same: call functions for each detection, call functions related to that detection's parameters, and from them produce XML output.

Throughout this document we will reference classes, objects and methods that are documented in the Nilus API documents found bundled with the Tethys install in Tethys\NilusXMLGenerator\target\apidocs. Open file index.html in the apidocs folder with a web browser (e.g. Google Chrome or Firefox) to see the class documentation.

2. Installation

Nilus is a standalone Java archive library (.jar). The library is called nilus-X.X.jar where X-X is the version number and is located in the NilusXMLGenerator\target folder relative to the root folder of the Tethys program folder. Programs using Java must be configured to include this folder in their class path. Being standalone, it can be utilized on machines where Tethys is not installed as long as it can be accessed.

3. Use with MATLAB

Matlab maintains two sets of path directory lists for loading Java libraries: static and dynamic. The Nilus jar must be on the static path. When you create a Tethys query handler (dbInit), the Nilus jar will be automatically added to the static path. Alternatively, the path to the Nilus jar can be added in the javaclasspath.txt file which must be stored in the folder returned by Matlab command `userpath`.

The tutorial in the next section uses Matlab as an example, and you can verify that the installation is correct.

4. Generating XML

The following example shows how Nilus can be used to generate a Detections document. It provides an overview of the commonly used Nilus functions and what they produce. Producing other types of documents is very similar.

Nilus provides Java objects that correspond to XML elements. There are two special Java classes that Nilus provides to assist you in doing this. The first is the marshaller. It takes the Java objects that represent XML elements and converts them to XML. The second is the Helper class. It provides functionality that can make life simpler. Examples of things that it can do include assisting you in creating subelements to a Java class and providing type conversions. For example, the XML schemata frequently require specific types such as xs:integer or xs:datetime. The helper class provides methods such as toXsInteger or timestamp that let you convert numbers and dates to appropriate types. Examples using these Java classes are detailed in the example sections on pgs. 4, 16, and 17.

While the documentation shows the methods that can be used with object instances of each class, the following example provides a concrete example of how they can be used and introduces the major design patterns that one would use in coding an XML generator. The example is in MATLAB, but with appropriate modifications would be applicable in any language that supports making calls to Java.

Example: Creating a Detections Document in MATLAB

Before starting, we will create a query handler using function dbInit which is part of the Tethys MATLAB client. As the query handler is created, it will also set paths such that that the Nilus libraries are accessible. The query handler object that dbInit returns can be used to communicate with the Tethys server. Note that if the javaclaspath.txt has been modified (section 3), this step is not needed if you do not need to communicate with the Tethys server.

```
% Set up a query handler (as described in the MATLAB cookbook)
% YourTethysServerAddress is the name of the machine you are
% running the Tethys server on.  Some examples:
% localhost - Server on the same machine as Matlab
% physeter.my.org - machine named physeter on network my.org
% 75.2.44.127 - IP address of server
query_h = dbInit('Server', 'YourTethysServerAddress')
```

As with using any new Java package, the first step is to import the Nilus library into the MATLAB environment:

```
% We will use this font to denote portions of a Matlab program
% or things that you might type at the Matlab prompt.

import nilus.*  % Make nilus classes accessible
```

Now we can create a Detections object – the overarching Java object that will eventually contain all of the information about this particular set of detections, corresponding to a Tethys Detection document. Documents in Tethys have a set of schemata that dictate what type of data elements can be present and also provide for extensibility so that users can insert custom data in certain specific places. To populate a document, you need to have some understanding of the data that are to be contained in it. The Tethys schemata can either be looked at directly on your machine or on the Tethys web site. For example, if you created a database instance in directory Documents\metadata, the schemata directory would be in: metadata\lib\schema.

These files are best viewed with an XML viewer such as Oxygen XML Editor. Alternatively, you can look at the online version of the schemata on the Tethys web site.

In this example, we will create a document that describes effort to acoustically find a species and records detections of that species. In the Tethys schemata, this is a detections document and has the top level schema shown in Figure 2.

```
detections = Detections(); % Create a Detections document
```

If the Java class paths have not been set up correctly, you will see an error: Unrecognized function or variable 'Detections'. If this is the case, the most likely cause is that dbInit was not executed previously or the javaclasspath.txt was not modified.

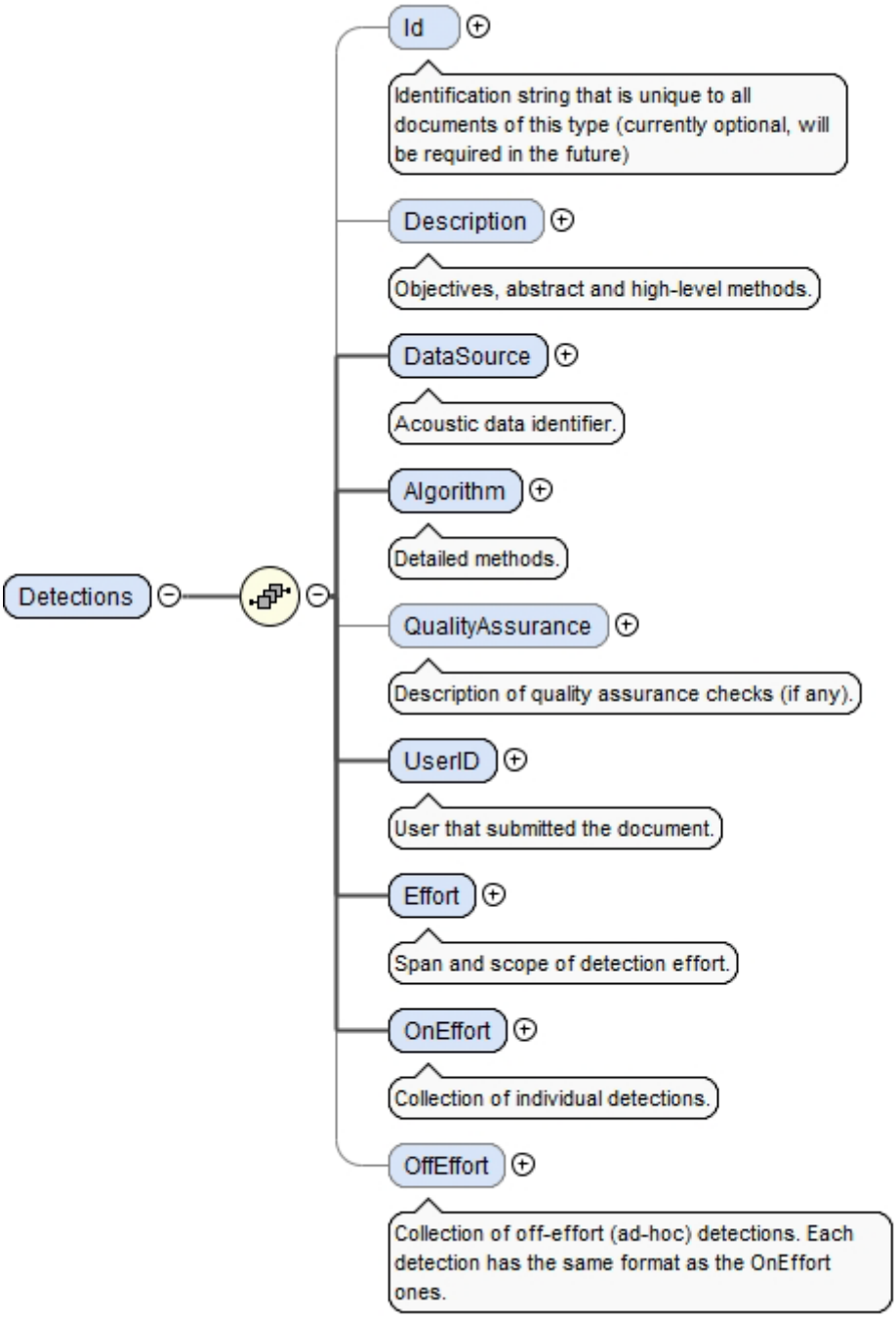


Figure 2 - Top level of the detections schema. Dark lines show mandatory elements that must be present. Elements connected with lighter lines are optional, for example the Description element is optional. Any element followed by a ⊕ represents a structure that contains sub-elements that are not shown here.

Two important classes that we will be using are the MarshalXML and Helper classes. We will create a marshalling object which will let us convert, or marshal, the detections object to XML:

```
marshaller = MarshalXML();
```

We have not populated the detections object yet, but we can still ask the marshaller to show us what has been populated:

```
marshaller.marshal(detections)
```

which produces the following empty XML document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Detections xsi:schemaLocation="http://tethys.sdsu.edu/schema/1.0 tethys.xsd"
xmlns:ns2="http://tethys.sdsu.edu/schema/1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
```

None of the elements within the Detections document have been set or created yet, so it is not too surprising that this XML is empty. In general, elements that are simple types (e.g. character or numeric data) can be set immediately. An example of this can be seen in the UserId element that specifies who conducted the analysis. This is one of the few top-level elements in the Detections schema that are a simple type:

```
detections.setUserId('KPayne')
```

The second important class that helps us with document generation is the Helper class. It provides methods that help us build objects.

```
helper = Helper();
```

Other fields in the detections document that are composed of subelements must be created before they can be populated. While these could be created and added separately, a Helper class has been written to assist with several tasks including initializing complex elements.

Let us initialize the *required* elements using the helper object:

```
helper.createRequiredElements(detections)

marshaller.marshal(detections)
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Detections xsi:schemaLocation="http://tethys.sdsu.edu/schema/1.0 tethys.xsd"
xmlns:ns2="http://tethys.sdsu.edu/schema/1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <DataSource/>
   <Algorithm>
      <Software></Software>
      <Parameters/>
   </Algorithm>
   <UserId>KPayne</UserId>
   <Effort/>
   <OnEffort/>
</ns2:Detections>
```

We see that the required elements have now been added and we also see that the UserId that we set earlier is present. You can see from Figure 2 that the mandatory elements: DataSource, Algorithm, UserId, Effort, and OnEffort were all required and are therefore created by the createRequiredElements method.

Note that createRequiredElements is not mandatory, but is very useful. If we had not invoked this method, we would have had to create elements individually and set them. For example, for the Algorithm element, we could have done the following:

```
algorithmType = AlgorithmType()
detections.setAlgorithm(algorithmType);
```

Note that this would not have set Algorithm's mandatory child elements: Software and Parameters. In general, the helper object's createRequiredElements can save a lot of time.

Metadata Variables for the Major Sections

With the Detections object created, a "preamble" should be set up next. This is the block of information pertinent to the entire detections document, the metadata. This includes entries such as the effort start and end times, an identifier for the person who prepared the document, the name of the software used, the source of the data, and so on.

DataSource

The DataSource element provides information that would be needed to find out details about the deployment where the data were collected. With the data source identified, the user could query the deployments to find out when and where the data were collected, sampling rates, duty cycles, etc. It is required and has several children that are all simple types. Let us set some of them by getting the DataSource and invoking set methods for the various fields:

```
% To get the DataSource from the detections object set a variable by %
% using the following method call against detections:

dataSource = detections.getDataSource();
```

This will give us access to the simple types for Project, Site and Deployments so that we can set them with their values from above.

```
% In this example, we are generating detections for the Deployment with
% deployment id CINMS04-C
dataSource.setDeploymentId('CINMS-04C');
```

Algorithm

The Algorithm element lets us know what methods were used to find detections. It is critical to denote the algorithm and its parameters.

If you have used the helper to createRequiredElements at the detections level then the algorithm block exists and you can get it with the following:

```
algorithm = detections.getAlgorithm();
```

And then set its components:

```
algorithm.setSoftware('my super-duper detector')
algorithm.setVersion('3.14')
```

Algorithm Parameters

Parameters is an extensible user-defined section of the schema. It lets users create arbitrary XML that describes the settings for the algorithm. It could be something as simple as an element <ParameterString> that contains a setting string to a complete XML hierarchy. See Roch et al 2016 (https://doi.org/10.1016/j.ecoinf.2015.12.002) for a discussion of this. Two examples:

```
<Detections>
 …
 <Algorithm>
  <Method>analyst</Method>
  <Software>Raven Pro</Software>
  <Version>1.4 build 48</Version>
  <Parameters>
   <SpectrogramWindowSize>512</Spec…>
   <Brightness>50</Brightness>
   <Contrast>50</Contrast>
   <SpectrogramLength_s>100</Spec…>
   …
  </Parameters>
 </Algorithm>
 …
</Detections>
```

```
<Detections>
 …
 <Algorithm>
  <Method>tonal detector</Method>
  <Software>silbido</Software>
  <Version>1.1</Version>
  <Parameters>
   <Framing>
      <Advance_ms>2</Advance_ms>
      <Length_ms>8</Length_ms>
   </Framing>
   <Threshold_dB>10</Threshold_db>
   …
  </Parameters>
 </Algorithm>
 …
</Detections>
```

Figure 3 – Sample user-defined parameters from Roch et al. 2016.

All algorithm parameters are user defined and are set using the methods described in a later section called Setting User Defined Algorithm Parameters on pg. 19.

UserId

```
%The UserId data is set at the Detections container level and
%was set earlier but we provide it here for completeness
% along with the other required %sections.

detections.setUserId("Kpayne");
```

Lists

Before we continue, it worth mentioning that there are several types of Nilus objects that can contain one or more subelements. These are represented as lists and we discuss the general mechanism for using these in this section.

All lists support the .add() method for adding list items. For example in the API documents in the NilusXMLGenerator/targets folder you will find that the object returned from getKind() is a list of DetectionEffortKind objects: List<DetectionEffortKind>.

Adding to it can be done directly with the add function. Assuming that we had generated newItem to add to the kinds list, we could do the following:

```
% This won't work yet as newItem is not defined, we will see concrete
% examples of using add later.
detections.getKind().add(newItem);
```

Some list objects however are Element Lists and as such they
Are described in the API like the following:

List<Element> any

'Any' here denotes its inheritance and means that they support the getAny() function that returns a list object that then can be added to using the add function.

```
params.getAny().add()
```

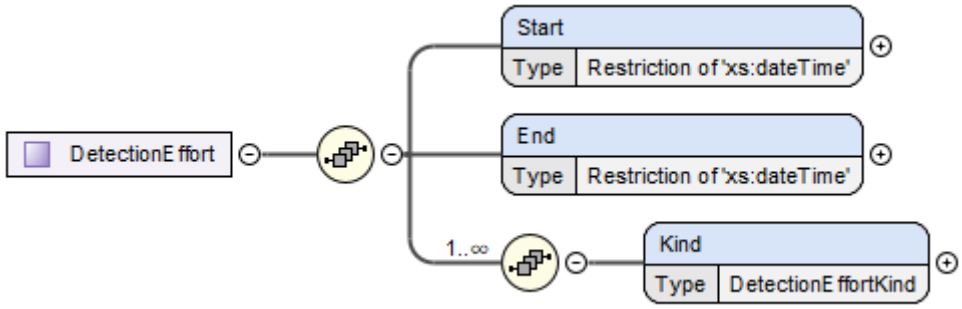
Lists will become important below for kinds, parameters, and detections.

Effort

To start an Effort section use the DetectionEffort function. Using the object created by the following you can easily set the start and end times.

effort = DetectionEffort()

From the schema shown below you can see that the Effort section has 3 required components as denoted by the darker connecting lines. Start and End are dateTime types and Kind(s) are complex types. There can be 1 to infinity of Kinds within an Effort.



Use the helper to create timestamps for the start and end times. The following arguments to timestamp produce a date/time, like the following 2014-01-01T12:00:00.000+00:00

```
% We indicate that we analyzed between 2014-01-01T12:00:00Z
% and 2015-01-01T12:00:00Z.  Arguments
% Y, M, D, H, M, S, ms, UTC-offset (min)
 effort.setStart(helper.timestamp(2014, 1, 1, 12, 0, 0, 0, 0))
 effort.setEnd(helper.timestamp(2015, 1, 1, 12, 0, 0, 0, 0))
```

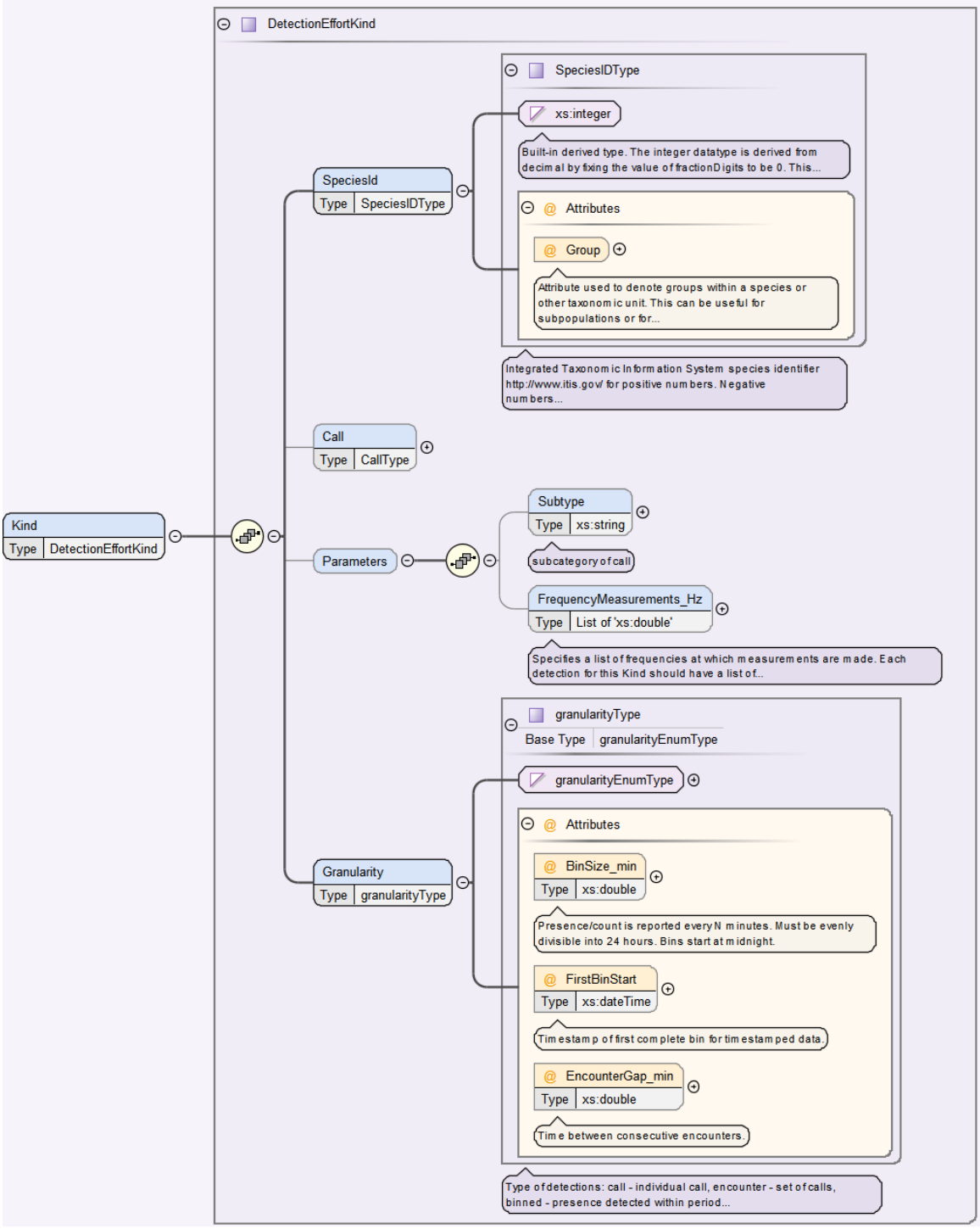
Kinds

The Kind element is an element that can be repeated. This can be seen in the schema by the notation 1..∞ which means that one or more Kind elements may be present. When multiple instances of an element are allowed, they are treated as a list that requires special processing.

In these cases, we use a getKind() method to obtain the list of Kind elements. For other lists, a different get method will be provided. This returns an empty list to which we can add:

```
% this returns a list object that individual kinds will be added to.
kinds = effort.getKind()
```

Below is an expanded version of the elements within the Kind.



To build an individual kind we do the following:

```
kind = DetectionEffortKind();
```

To specify the species, we need a taxonomic serial number from the Integrated Taxonomic Information System:

```
% Specify species for which we are looking.
% Taxonomic serial number(TSN) for sperm whales:  https://itis.gov/
species = 180488;
```

If the TSN is unknown, the ITIS collection in Tethys can be queried for that information. For a sperm whale, we might query using the Latin name *Physeter macrocephalus* or with a user defined abbreviation. NOAA has defined the NOAA.NMFS.v1 species abbreviation map where sperm whales are abbreviated as Pm. In both cases, we will use Tethys library XQuery functions:

```
% Use the handler to query for a Latin name
species = ...
query_h.QueryTethys('lib:completename2tsn("Physeter macrocephalus")')

%Or a species abbreviation map:
species = query_h.QueryTethys('lib:abbrev2tsn("Pm","NOAA.NMFS.v1")')
```

In both cases, species will be set to 180488. Once the TSN is known, we need to set the species to sperm whale. As the speciesId is defined as a type in its own right (SpeciesIDType), we need to create an instance of the SpeciesIDType first and then set its value. We then set the speciesId:

```
speciestype = SpeciesIDType();
speciestype.setValue(helper.toXsInteger(species));
kind.setSpeciesId(speciestype)
```

We next set the type of call that we are detecting.

```
% effort is to detect this type of call
calltype = CallType();
calltype.setValue('Clicks');
kind.setCall(calltype);
```

and the granularity which specifies whether we are reporting individual detections, starts and ends of detections, or detections within a binned time interval.

```
granularitytype = GranularityEnumType.fromValue('encounter');
granularity = GranularityType();
granularity.setValue(granularitytype);
kind.setGranularity(granularity);
```

When we are through building the required sub-elements of the kind we add the kind to the kinds list that we built earlier and link the effort to the detections object:

```
kinds.add(kind);    % additional kinds could be added

% Now associate the effort with the detections object
detections.setEffort(effort)
```

If we were to print what we have so far, we would produce something like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<ty:Detections xmlns:ty="http://tethys.sdsu.edu/schema/1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://tethys.sdsu.edu/schema/1.0 tethys.xsd">
    <DataSource>
        <Project>CINMS</Project>
        <Deployment>4</Deployment>
        <Site>C</Site>
    </DataSource>
    <Algorithm>
        <Software>Triton</Software>
    <UserId>KPayne</UserId>
    <Effort>
        <Start>2014-01-01T12:00:00Z</Start>
        <End>2015-01-01T12:00:00Z</End>
        <Kind>
            <SpeciesId>180488</SpeciesId>
            <Call>clicks</Call>
            <Granularity>encounter</Granularity>
        </Kind>
    </Effort>
    <OnEffort>
    </OnEffort>
</ty:Detections>
```

OnEffort

With the **`detections`** object created and the preamble values set, the stage has been set for individual detections to be added. By doing so the empty `<OnEffort>` section of the XML document that was created above with helper.createRequiredElements(detections) will be appended with each `<Detection>`.

As in the previous steps, there are On Effort fields required for Tethys import (species ID and call), as well as a multitude of optional fields. The full list of parameters is in the bundled javadoc, as well as in Tethys' XML schema itself. There is also the option to create User Defined parameters, as described in the Other Cases section on page 17. Let us begin by creating a single detection.

```
Detection = Detection();
```

Now we append the relevant data to the Detection:

```
% Just like above we use the helper's timestamp function to give the
% effort a start date.  The timestamp function also takes ISO8061
% strings, e.g. "2014-01-01T12:00:00"
Detection.setStart(helper.timestamp(2014,1,1,12,0,0,0,0))

%SpeciesId - Similar to what we did in effort, create an instance
%of the speciesIDType, populate it and set the speciesId
speciestype = SpeciesIDType();
speciestype.setValue(helper.toXsInteger(species));
% Now set the species type to the Detection object
Detection.setSpeciesId(speciestype);
```

We now create a list of calls for this detection. In most cases, there will be only one call, but when reporting over time spans (granularity encounter or binned), there may be more than one call type for a specific species between the start and end time. As a consequence, we need to access a list:

```
% call
helper.createElement(detection, 'Call')
callList = detection.getCall(); % Get the empty list
callList.add('Clicks');  % Create instance of Call
```

The optional Detection element Parameters contains measurements for a call. We show an example of creating the Parameters element and setting the received level of the detected signal.

```
%Here we can set a parameter under the parameter element which
%must be created first.  There are other setter functions for
%other parameters.
helper.createElement(Detection, 'Parameters')
DetectionParameter = Detection.getParameters()
DetectionParameter.setReceivedLevelDB(helper.toXsDouble(4.6))
```

The `Detection` now contains the following data:

```
<Detection>
    <Start>2014-01-01T13:00:00Z</Start>
    <SpeciesId>180488</SpeciesId>
    <Call>Clicks</Call>
    <Parameters>
        <ReceivedLevel_dB>4.6</ReceivedLevel_dB>
    </Parameters>
</Detection>
```

Now that a `Detection` has been created and set, it can be added to the set of detections:

```
detections.addDetection(Detection);
```

This process is repeated for each detection that is found. Once the detection process is complete, we can generate an XML document that describes our detection effort and the detections that were found:

```
%This will print the XML to the standard output

marshaller.marshal(detections)

%This will write the output to a file

xml_out = 'C:\Detector\Output\SOCAL_Detections.xml'
marshaller.marshal(detections, xml_out)
```

Congratulations, you have successfully generated a Tethys XML detections document! It should be formatted the same as the example XML output above, with each `<Detection>` added to the `<OnEffort>` section. This document can now be added to a Tethys server using the data import methods described in the Tethys manual.

Case Study: Integrating data from spreadsheet into Nilus

This section shows an example of data that were provided by a user in the form of three spreadsheets. **These spreadsheets are in the documentation folder under NilusData**. The user had created a spreadsheet with detection data (SOCAL_U_01_automatic_UBW_jst.xls), information on a local set of species abbreviations that were not added to Tethys (Species Effort_Beaked whale detector.xls), and an effort spreadsheet that identified deployments during which the user was looking for beaked whales along with the type of beaked whale for which effort was conducted.

The Kind elements for Effort for this document would be added very similarly to what has been shown in the Effort section on pg.10. In this example, we skip this portion and move on to reporting the occurrences of beaked whales that were identified by the detector.

We assume that query_h has been initialized to point to Tethys server, e.g.

```
query_h = dbInit();

%In a situation presented by the three spreadsheets included with the
%documentation where one sheet is detections and the other effort and
%the last a species code file. We can read from all of them and build a
%detections document.  Filenames are hard coded for this example.

query_h = dbInit();
detectionfile = 'SOCAL_U_01_automatic_UBW_jst.xls';
effortfile = 'BW_Effort.xls';
speciesfile = 'Species Effort_Beaked whale detector.xls';

%Find out detection effort by first finding the deployment id from the
%detections filename

match = regexp(detectionfile, ...
    '(?<depid>.*)_automatic.*', 'names');
depid = match.depid;


% Read in the effort, detection, and species spreadsheets
eff = readtable(effortfile, 'Sheet', 'Effort times');
det = readtable(detectionfile, 'Sheet', 'Sheet1');
speciestable = readtable(speciesfile, 'Sheet' , 'Sheet1') % 'Species
code'

%The readtable function reads in a sheet from a spreadsheet, the
%effortfile, detectionfile, and spiecesfile variables are the full path
%to the spreadsheets on your file system.

%The readtable function builds a MATLAB object with fields with the
%same fieldnames from the table.  For example below is how your species
```

```
%code table should be formatted, and when using the readtable function
%like so: speciestable = readtable(speciesfile, 'Sheet' , 'Sheet1') it
%will build the object speciestable, whose columns can be addressed as
%speciestable.SpeciesCode, speciestable.CommonName and
%speciestable.Scientificname
```

	A	B	C
1	**Species code**	**Common name**	**Scientific name**
2	Ham	Northern Bottlenose Whale	*Hyperoodon ampullatus*
3			
4	Zcv	Cuvier's Beaked Whale	*Ziphius cavirostris*
5			
6	Mde	Blainville's Beaked Whale	*Mesoplodon densirostris*
7			
8	Mbi	Sowerby's Beaked Whale	*Mesoplodon bidens*
9			
10	Meu	Gervais' Beaked Whale	*Mesoplodon europaeus*
11			
12	Mmi	True's Beaked Whale	*Mesoplodon mirus*
13			
14	Mst	Stejneger's Beaked Whale	*Mesoplodon stejnegeri*
15			
16	Mho	Deraniyagala's Beaked Whale	*Mesoplodon hotaula*
17			
18	BWC	Cross Beaked Whale	*Ziphiidae species*
19			
20	BWG	Gulf of Mexico Beaked Whale	*Ziphiidae species*
21			
22	BW29	29 kHz Beaked Whale	*Ziphiidae species*
23			
24	BW31	31 kHz Beaked Whale	*Ziphiidae species*

```
%Next we find the entry in the effort table based on the deployment id
%derived from the detection file name. This object will have the
%information for the effort needed for the detections document

effentry = eff(strcmp(eff.Deployments, depid), :);

sciencename = strip(speciestable.ScientificName);
abbreviations = strip(speciestable.SpeciesCode);

% Detection documents require that your species names be tsn integers
query_h.QueryTethys('lib:completename2tsn("Ziphius cavirostris")')

%For each kind within an effort and detection within OnEffort you will
%need the species tsn.  You may want to build a MATLAB structure using
%the two vectors, sciencename and abbreviations where the abbreviations
%are the key in a lookup for the scientific name and then use the above
%function to covert the scientific name to a tsn.

% In this example Project, Deployment and Site variables can be derived
%from the effentry.Deployments field and the effentry.Sites field and
%used to fill out the DataSource section.
```

```
% The start and end times of the effort can also be determined from the
%StartEffort and EndEffort columns in the effentry object

start = effentry.StartEffort;
stop  = effentry.EndEffort;

% These times will need to be converted into ISO8601 formats like this:
start = dbSerialDateToISO8601(start)

%Making the kinds for the effort section requires looping through all
%the unique species in the species table and making a kind for each
%one.

% The onEffort section requires a loop that goes through the detection
spreadsheet object 'det'.

OnEffort = detections.getOnEffort()
DetectionList = OnEffort.getDetection()
    for d = 1:size(det,1)

      detection = Detection();
      % time stuff, dates converted from Excel format
      start = det.StartTime(d) + offset_epoch('excel');
      stop  = det.EndTime(d) + offset_epoch('excel');
      detection_8601 = dbSerialDateToISO8601([start; stop]);
      detection.setStart(helper.timestamp(detection_8601{1}));
      detection.setEnd(helper.timestamp(detection_8601{2}));

      % Look up the species code (we could do this faster by caching
      % the results)
      detected_species = det.SpeciesCode{d};

      tsn = str2num(server.QueryTethys(sprintf('lib:abbrev2tsn("%s",
      "SIO.SWAL.v1")', detected_species)));

      species_int = helper.toXsInteger(tsn);
      speciestype = nilus.SpeciesIDType();
      speciestype.setValue(species_int);

      detection.setSpeciesId(speciestype)

      % call
      helper.createElement(detection, 'Call')
      callList = detection.getCall();
      acall = javaObject('nilus.Detection$Call')
      acall.setValue(det.Call{d})
      callList.add(acall)

      DetectionList.add(detection);
    end
```

Other Cases – User Defined Parameters

Portions of the Tethys schemata allow users to place any type of information they would like in the document. This allows Tethys to provide extensibility while maintaining standards for things that should be stored in a similar manner. The philosophy used in the Java classes that represent arbitrary XML (e.g. parameters provided to a detection or localization algorithm or a non-standard parameter measurements on a detection) is to generate an empty list at the time the element is created.

Any XML element may be added to this list. In general, we will use the getAny() method on element that contains the arbitrary XML and then user the Helper class's addAnyElement method to add element name and value pairs to the list. Two concrete examples of doing this are provided below.

Setting User Defined Detection Parameters

If a detection parameter is required that is not included with Nilus they can be created with two different helper functions, createElement and AddAnyElement.

```
% Let's say that we have already constructed a detection object
% For the purpose of this example we will build our own detection
% Using the following simple method call
detection = Detection()
```

The first thing we should do is to create the Parameters element for the detection. Since it is not a required element we will make it from scratch using the Helper Class. After we are done constructing it, we will retrieve its object and store it in a variable called params, like so:

```
% this makes the element Parameters against its parent detection object
helper.createElement(detection, 'Parameters')

% this retrieves the Parameters element and stores it in a variable
params = detection.getParameters()
```

Now we need to create the user defined element within the Parameters element using the variable params as the parent.

```
helper.createElement(params, 'UserDefined');
% Grab the user defined element
userdefined = params.getUserDefined();

% User defined elements contain a list, access the list with getAny()
% It will be empty when we first start
userdefined_list = userdefined.getAny();

% Add things to the user defined list
% As there is no type information for user defined values
% everything is stored as a string.
helper.AddAnyElement(userdefined_list, 'Example_Parameter', '56')
```

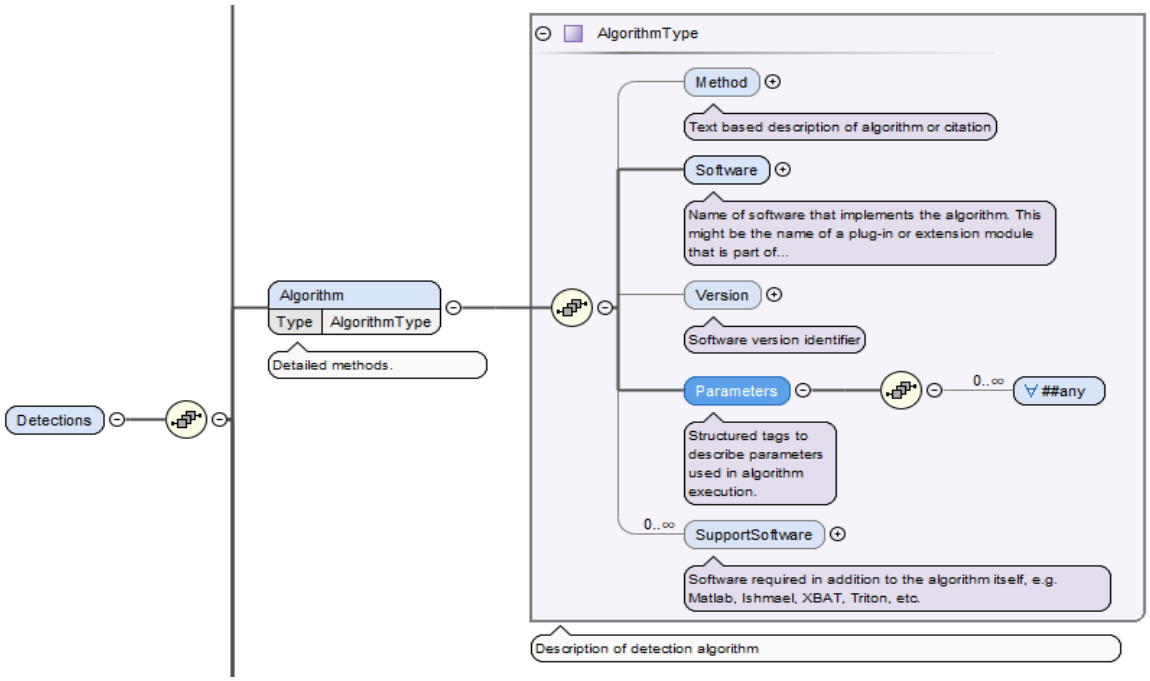
Now if you marshal the detection you should see the following:

```
 marshaller.marshal(detection)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Parameters>
   <UserDefined>
      <Example_Parameter>56</Example_Parameter>
   </UserDefined>
</Parameters>
```

Setting User Defined Algorithm Parameters

As shown below in the schema diagram for detection documents there are Parameters in the Algorithm section at the Detections level



We will start by showing some housekeeping necessary for setting up the parent objects, starting with the Detections container, for adding user defined parameters to an algorithm. Keep in mind that these objects were built above in other example, we repeat them for clarity.

```
% just setting up the parent objects
 Detections = Detections()
 algoType = AlgorithmType()
 Detections.setAlgorithm(algoType)
 Algorithm = Detections.getAlgorithm()
```

```
%At this point we have built all the parent objects necessary to give
%us the Algorithm object which is stored in the variable Algorithm.
%With this object we can create the required fields in Algorithm which
%are 'Parameters' and 'Software' using the following:

 helper.createRequiredElements(Algorithm)
```

Now that the Parameters element exists we can get it and its List with the following two lines.

```
 params = Algorithm.getParameters()
 param_list = params.getAny()
```

Now we simply add a user defined parameter to the param_list similar to the way we used AddAnyElement above.

```
 helper.AddAnyElement(param_list,'Test_Parameter','99');
```

It is possible to build arbitrarily complex XML, but this requires a bit more code. For Matlab users, a function is provided in the Matlab client called dbStruct2DOM that will allow you to convert Matlab structures to XML.

As an example, if we had the structure p with the following fields:

Threshold_dB: 10
Framing.Advance_ms: 2
Framing.Length_ms: 8

We could add these to the parameters as follows:

```
 dbStruct2DOM(p, param_list)
```

When we marshalled the detection document, the Parameters section of the algorithm would look like this:

```
<Parameters>
   <Threshold_dB>10</Threshold_dB>
   <Framing>
      <Advance_ms>2</Advance_ms>
      <Length_ms>8</Length_ms>
   </Framing>
</Parameters>
```

One can look at the source code for dbStruct2DOM to see how this is done if this type of flexibility is needed for other languages.

Nested classes

Some of the public classes generated from the schema are nested. For example, in the deployment schema, the Channel element has a nested class called Sampling that describes the instrument's sampling regimen.

In Java, we would create an instance of the Channel class (channel in this example) and then write something like this:

```
channel.Sampling sampling = channel.new Sampling();
```

This is a bit more difficult to do with Matlab's interface to Java. As a consequence, we wrote a function called createSubclass that takes two arguments, an instance of the parent class, and a string with the subclass name. Assuming that channel held an instance of Channel, we would write:

```
sampling = createSubclass(channel, 'Sampling');
```

The createSubclass can be found in the NilusExamples directory or in this Matlab Answers post. If you are very familiar with Java, you may wish to use Peter Li's suggestion which does not rely on a wrapper function.

5. Caveats / Troubleshooting

There are several places where it is easy to become confused.

Mismatched types

Much of the code is generated automatically from the Tethys schemata, and this can sometimes lead to situations that are unexpected and not very intuitive. For example, in most cases, numbers are simple types of double or int. However, when the schema places a restriction on values, such as a range restriction or making it optional, the Java types Double or Integer are used. These are classes that wrap the underlying type, and the value that you wish to populate must be of the appropriate type. Timestamps must be in the XMLGregorianCalendar type.

If you are using writing in Java, you can create an instance of the appropriate class. The Helper class provides methods that let you create instances of these for other languages:

- timestamp – Creates an XMLGregorianCalendar from a variety of formats including strings compliant with xs:DateTime. See the apidocs for details.
- toXsDecimal – Creates a Java BigDecimal object from a double
- toXsDouble – Creates and xs:double instance from a floating point number
- toXsInteger – Creates an xs:integer type from an integer.

See the API docs for details in folder NilusXMLGenerator/target/apidocs .

Lists

Lists of objects can be confusing for many new users to Nilus as they involve retrieving an empty list and then adding to it. For details, see the discussion on page 10.

6. Conclusion

In this introduction, we have provided a preliminary guide to building Detections XML documents for import into Tethys using the Nilus XML Generation interface. All information on the Nilus API can be found in the API documents as mentioned at the bottom of the Overview section included in the distribution.