# Accessing Data from Sensor Observation Services: the **sos4R** Package

Daniel Nüst*
daniel.nuest@uni-muenster.de
http://www.nordholmen.net/sos4r

January 5, 2011

**Abstract**

The sos4R package provides easy and simple, yet powerful access to OGC Sensor Observation Service instances. The package supports both encapsulation and abstraction from the service interface for novice users as well as powerful request building for specialists.

sos4R is motivated by the idea to add a missing link between the Sensor Web and tools (geo-)statistical analyses. It implements the core profile of the SOS specification and supports temporal, spatial, and thematical filtering of observations. This document briefly introduces the SOS specification. The package's features are explained extensively: exploration of service metadata, request building with filters, function exchangeability, result data transformation.

The package is published under GPL 2 license within the geostatistics community of 52°North Initiative for Geospatial Open Source Software.

## Contents



*Institute for Geoinformatics, University of Muenster, Germany.

# 1 Introduction

The **sos4R** package provides classes and methods for retrieving data from an OGC Sensor Observation Service (Na, 2007). The goal of this package is to provide easy access with a low entry threshold for everyone to information available via SOSs. The complexity of the service interface shall be shielded from the user as much as possible, while still leaving enough possiblities for advanced users. At the current state, the output is limited to a standard data.frame with attributed columns for metadata. In future releases a tighter integration is planned with upcoming space-time packages regarding data structures and classes. This package uses S4 classes and methods style (Chambers, 1998).

The motivation to write this package was born out of perceiving a missing link between the Sensor Web community (known as Sensor Web Enablement (SWE) Initiative[1] in the OGC realm) and the community of (geo-)statisticians. While the relatively young SWE standards get adopted more by data owners (like governmental organizations), we see a high but unused potential for more open data and spatio-temporal analyses based on it. **sos4R** can help enabling this.

[1] http://www.opengeospatial.org/projects/groups/sensorweb

The project is part of the geostatistics community[2] of the 52 °North Initiative for Geospatial Open Source Software[3]. **sos4R** is available, or will be available soon, on CRAN.

On the package home page, http://www.nordholmen.net/sos4r/, you can stay updated with the development blog, find example code and services, and download source packages.

This software is released under a GPL 2 license[4] and contributions are very welcome. Please consult section 10 for details.

The package **sos4R** is loaded by

```
> library("sos4R")
```

This document was build for **package version 0.1-08**.

## 1.1 Related Specifications

The Open Geospatial Consortium[5] (OGC) is an organisation which provides standards for handling geospatial data on the internet, thereby ensuring interoperability.

The Sensor Observation Service (SOS) is such a standard and provides a well-defined interface for data warehousing of measurements and observations made by all kinds of sensors. This vignette describes the classes, methods and functions provided by **sos4R** to query these observations.

Storing and providing data in web services is more powerful than local file copies (with issues like being outdated, redundancy, ...). Flexible filtering of data on the service side reduces download size. That is why SOS operations can comprise flexible subsetting in temporal, spatial and thematical domain. For example "Provide only measurements from sensor MySensor-001 for the time period from 01/12/2010 to 31/12/2010 where the air temperature below zero degrees".

In general, the SOS supports two methods of requesting data: (i) HTTP GET as defined in the OOSTethys best practice document[6] with key-value-pair (KVP) encoding of request, and (ii) POST as defined in the standard document with requests encoded in eXtensible Markup Language (XML). Both request types always returns XML documents as response.

Standards that are referenced, respectively used, by SOS are as follows.

**Observations and Measurements (O&M)** O&M (Cox, 2007) defines the markup of sensor measurements results. An observation consists of information about the observerd geographic feature, the time of observation, the sensor, the observed phenomenon, and the observation's actual result.

**Sensor Model Language (SensorML)** SensorML (Botts, 2007) is used for sensor metadata descriptions (calibration information, inputs and outputs, maintainer).

[2]http://52north.org/communities/geostatistics/
[3]http://52north.org/
[4]http://www.gnu.org/licenses/gpl-2.0.html
[5]http://www.opengeospatial.org/
[6]http://www.oostethys.org/best-practices/best-practices-get

**Geography Markup Language (GML)** (Portele, 2003) defines markup for geographical features (points, lines, polygons, ...).

**SweCommon** SWE Common describes data markup and is contained in the SensorML specification.

**Filter Encoding** Filter Encoding (Vretanos, 2005) defines operators and operands for filtering values.

**OWS Common** OGC Web Services Common (Whiteside, 2007) models service related elements that are reusable across several service specifications, like exception handling.

## 1.2 Terms and Definitions

The OGC has a particular set of well-defined terms that might differ from usage of words in specific domains. The most important are as follows[7].

**Feature of Interest (FOI)** The FOI represents the geo-object, for which measurements are made by sensors. It is ordinarily used for the spatial referencing of measuring points, i.e. the geoobject has coordinates like latitude, longitude and height. The feature is project specific and can be anything from a point (e.g. the position of a measuring station) or a real-world object (e.g. the region that is observed).

**Observation** The observation delivers a measurement (result) for a property (phenomenon) of an observed object (FOI). The actual value is created by a sensor or procedure. The phenomenon was measured at a specific time (sampling time) and the value was generated at a specific point in time (result time). These often coincide so in practice the sampling time is often used as the point in time of an obsrevation.

**Offering** The offering is a logical collection of related observations (similar to a layer in mapping applications) which a service offers together.

**Phenomenon** A phenomenon is a property (physical value) of a geographical object, e.g. air temperature, wind speed, concentration of a pollutant in the atmosphere, reflected radiation in a specific frequency band (colours).

**Procedure** A procedure creates the measurement value of an observation. The source can be a reading from a sensor, simulation or a numerical process.

A more extensive discussion is available in the the O&M specification (Cox, 2007). The Annex B of that document contains the examples of applicating some terms to specific domains, aerosol analysis and earth observations, which are repeated here for elaboration in table 1.

[7] Based on `http://de.wikipedia.org/wiki/Sensor_Observation_Service`

| O&M | Particulate Matter 2.5 Concentrations | Earth Observations |
|---|---|---|
| Observation::result | 35 ug/m3 | observation value, measurement value |
| Observation::procedure | U.S. EPA Federal Reference Method for PM 2.5 | method, sensor |
| Observation::observedProperty | Particulate Matter 2.5 | parameter, variable |
| Observation::featureOfInterest | troposphere | media (air, water, ...), Global Change Master Directory "Topic" |

Table 1: Domain specific variants of O&M terms.

# 2 Supported Features

The package provides accessor functions for the supported parameters. It is recommended to access options from the lists returned by these functions instead of hard-coding them into scripts.

```
> SosSupportedOperations()

[1] "GetCapabilities"    "DescribeSensor"     "GetObservation"
[4] "GetObservationById"

> SosSupportedServiceVersions()

[1] "1.0.0"

> SosSupportedConnectionMethods()

   GET   POST
 "GET" "POST"

> SosSupportedResponseFormats()

[1] "text/xml;subtype=&quot;om/1.0.0&quot;"
[2] "text/xml;subtype=&quot;sensorML/1.0.1&quot;"
[3] "text/csv"

> SosSupportedResponseModes()

[1] "inline"

> SosSupportedResultModels()

[1] "om:Measurement" "om:Observation"
```

The output of the following calls are named lists which are simplified here for brevity using `toString()`.

```
> SosSupportedSpatialOperators()

[1] "BBOX, Contains, Intersects, Overlaps"

> SosSupportedTemporalOperators()

[1] "TM_After, TM_Before, TM_During, TM_Equals"
```

### 2.1 Supported Services and Implementations

**sos4R** supports the core profile of the SOS specification. But the possible markups for observations is extremely manifold due to the flexibility of the O&M specification. Sadly, there is no common application profile for certain types of observations, like simple measurements.

Therefore, the undocumented profile of the **52 °North SOS implementation**[8] was used as a guideline. It is not documented outside of the source code. Observations returned by instances of this implementation are most likely to be processed out of the box.

In the author's experience, **OOSThetys SOS implementations**[9] utilise the same or at least very similar profile, so responses of these service instances are probably parsed without further work as well.

Please share your experiences with other SOS implementations with the developers and users of **sos4R** (see section 9).

## 3 Default Options

Two kinds of default values can be found in (function calls in) **sos4R**: (i) default depending on other function parameters, and (ii) global defaults. Global defaults can be inspected (not set!) using the following functions. If you want to use a different value please adapt the respective argument in function calls.

```
> SosDefaultConnectionMethod()

[1] "POST"

> SosDefaults()

$sosDefaultCharacterEncoding
[1] "UTF-8"

$sosDefaultDescribeSensorOutputFormat
[1] "text/xml;subtype=&quot;sensorML/1.0.1&quot;"

$sosDefaultGetCapSections
[1] "All"

$sosDefaultGetCapAcceptFormats
[1] "text/xml"

$sosDefaultGetCapOwsVersion
[1] "1.1.0"

$sosDefaultGetObsResponseFormat
[1] "text/xml;subtype=&quot;om/1.0.0&quot;"

$sosDefaultTimeFormat
```

[8] http://52north.org/communities/sensorweb/sos/
[9] http://www.oostethys.org/

```
[1] "%Y-%m-%dT%H:%M:%OS"

$sosDefaultFilenameTimeFormat
[1] "%Y-%m-%d_%H:%M:%OS"

$sosDefaultTempOpPropertyName
[1] "om:samplingTime"

$sosDefaultTemporalOperator
[1] "TM_During"

$sosDefaultSpatialOpPropertyName
[1] "urn:ogc:data:location"

$sosDefaultColumnNameFeatureIdentifier
[1] "feature"

$sosDefaultColumnNameLat
[1] "lat"

$sosDefaultColumnNameLon
[1] "lon"

$sosDefaultColumnNameSRS
[1] "SRS"
```

The process of data download also comprises (i) building requests, (ii) decoding responses, and (iii) applying the correct R data type to the respective data values. This mechanism is explained in detail in see section 6. The package comes with a set of predefined encoders, decoders and converters.

```
> SosEncodingFunctions()
> SosParsingFunctions()
> SosDataFieldConvertingFunctions()
```

# 4 Creating a SOS connection

The operation `SOS(...)` is a construction method for classes encapsulating a connection to a SOS. It prints out a short statement when the connection was successfully established (i.e. the capabilities document was received) and returns an object of class `SOS`.

```
> mySOS = SOS(url = "http://v-swe.uni-muenster.de:8080/WeatherSOS/sos")

Created SOS for URL http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
```

To create a SOS connection you only need the URL of the service (i.e. the URL which can be used for HTTP GET or POST requests).

The optional parameters use default settings (see section 3):

- `method`: The transport protocol. Currently available are GET.

- `version`: The service version. Currently allowed are Currenlty available are 1.0.0.
- `parsers`: The list of parsing functions. See section 6.3.
- `encoders`: The list of encoding functions. See section 6.2.
- `dataFieldConverters`: The list of conversion functions. See section 6.4.
- `curlHandle`, `curlOptions`: Settings of the package **RCurl**, which is used for HTTP connections. Please consult the packags specification before using this.
- `timeFormat`: The time format to be used or decoding and encoding time character strings to and from `POSIXt` classes.
- `verboseOutput`: Trigger parameter for extensive debugging information on the console, see section 7.2.

There are accessor methods for the slots of the class.

```
> sosUrl(mySOS)

[1] "http://v-swe.uni-muenster.de:8080/WeatherSOS/sos"

> sosVersion(mySOS)

[1] "1.0.0"

> sosTimeFormat(mySOS)

[1] "%Y-%m-%dT%H:%M:%OS"

> sosMethod(mySOS)

[1] "POST"
```

The default connection method is HTTP POST, but since not all SOS support this a GET connection is possible was well. The latter is less powerful, especially regarding filtering operations. Section 6.4 contains an example of such a connection.

The following slots are best described in section 6.

```
> sosParsers(mySOS)

> sosDataFieldConverters(mySOS)
```

# 5 SOS Operations

**sos4R** implements the SOS core profile of version 1.0.0 comprising the operations GetCapabilities, DescribeSensor and GetObservation. This document focusses on the practical usage of the operations, so the reader is refered to the specification document for details.

The methods mirroring the SOS operations all contain debugging parameters `inspect` and `verbose` as described in section 7.2.

## 5.1 GetCapabilities

The GetCapabilities operations is automatically conducted during the connecting to a SOS instance. The response is the **capabilities document**, which contains a detailed description of the services capabilities. It's sections describe: service identification, service provider, operations metadata (parameter names, ...), filter capabilities, contents (lists offering descriptions). Please see section 8.2.3 of the SOS specification for details. If you want to inspect the original capabilities document it can be re-requested using

```
> sosCapabilitiesDocumentOriginal(sos = mySOS)
```

The actual operation can be started with the following function. It returns an object of class `SosCapabilities` which can be accessed later on by the function `sosCaps()` from an object of class `SOS`.

```
> getCapabilities(sos = mySOS)
```

The parameters of the operation are:

- `sos`: The SOS connection to request the capabilities document from.
- `inspect` and `verbose`: See section 7.2.

The respective **parts of the capabilities document** are modelled as R classes and can be accessed with these functions:

```
> sosServiceIdentification(mySOS)
> sosServiceProvider(mySOS)
> sosFilter_Capabilities(mySOS)
> sosContents(mySOS)
```

The contents part is described in detail in section 5.3.1.

The function `sosTime(...)` returns the time period for which observations are available within the service. To be precise, it accesses the ows:Range element of the parameter eventTime in the description of the GetObservation operation.

```
> sosTime(mySOS)
```

```
Object of class OwsRange; spacing:  NA , rangeClosure:  NA
FROM  2008-02-14T11:03:02.000+01:00  TO  2011-01-05T18:00:00.000+01:00
```

The operations supported by the SOS are listed in the ows:OperationsMetadata element, which is modelled as an R class, `OwsOperationsMetadata`, which contains a list of objects of class `OwsOperation` which in turn describe the allowed parameter values for calls to the operation. The metadata element and single observations can be inspected with the following functions. This is rarely needed for users of the package so we abstrain from further details at this point.

```
> sosOperationsMetadata(mySOS)
> sosOperation(mySOS, "GetCapabilities")
> sosOperation(mySOS, sosGetCapabilitiesName)
> sosOperation(mySOS, sosDescribeSensorName)
> sosOperation(mySOS, "GetResult")
```

The allowed response formats (the file format/encoding of the response), the response modes (for example inline or as attachment) and the result models (a qualified XML name of the root element of the response) differ for every operation of the service. The following accessor methods return either (i) a list (named by the operation names) of vectors (with the actual allowed parameter values), or (ii) with the `unique` parameter set to `TRUE`, a unique list of all allowed values. Please be aware that these are not allowed for all operations, not are all options supported by **sos4R**.

```
> sosResponseFormats(mySOS)
> sosResponseMode(mySOS)
> sosResultModels(mySOS)
```

Some exemplary outputs of the operations are:

```
> sosResponseMode(mySOS, unique = TRUE)

[[1]]
[1] "inline"

[[2]]
[1] "resultTemplate"

> sosResultModels(mySOS, unique = TRUE)[[1]]

[1] "om:Observation"

> sosResponseMode(mySOS)[[sosGetObservationByIdName]]

[[1]]
[1] "inline"

[[2]]
[1] "resultTemplate"

> sosResultModels(mySOS)[[sosGetObservationName]][3:4]

[[1]]
[1] "om:CategoryObservation"

[[2]]
[1] "om:SpatialObservation"

> sosResponseFormats(mySOS)[[1]]

NULL
```

## 5.2 DescribeSensor

The DescribeSensor operation is specified in clause 8.3 of the SOS specification and its response is modeled in Sensor Model Language[10] (SensorML) and Transducer Markup Language[11] (TML) specifications.

[10] http://www.opengeospatial.org/standards/sensorml
[11] http://www.opengeospatial.org/standards/tml

> The DescribeSensor operation is useful for obtaining detailed information of sensor characteristics encoded in either SensorML or TML. The sensor characteristics can include lists and definitions of observables supported by the sensor. [...]

The parameters of the operation are as follows. Please see section 2 of this document for supported values of request parameters.

- `sos`: The SOS connection to request a sensor description from.
- `procedure`: The identifier of the sensor, so one of the character strings returned by `sosProcedures(...)`.
- `outputFormat`: The format in which the sensor description is to be returned. The default is `text/xml;subtype='sensorML/1.0.1'`.
- `inspect` and `verbose`: See section 7.2.

```
> sensor.1.1 <- describeSensor(sos = mySOS,
+                  procedure = sosProcedures(obj = mySOS)[[1]][[1]])
```

```
Object of class SensorML (wraps unparsed XML, see @xml for details).
```

## 5.3 GetObservation

The GetObservation operation is specified in clause 8.4 of the SOS specification. In this section, all matters around requesting data are explained — from extracting query parameters from metadata, and sending the request, till finally extracting data values and coordinates from the response.

A few utility functions exist to minize a user's amount of work to create usual requests. They accept normal R types as input and return the respective class from **sos4R** with useful default settings. These function's names follow the pattern with `sosCreate [name of object] ()` and exist for spatial and temporal filters.

### 5.3.1 Metadata Extraction for Request Building

It is recommended to extract the identifiers of procedures et cetera that are to be used for queries from the metadata description provided by the service, the capabilities document (see section 5.1. This often ensures forward compatiblity and minimizes typing errors. The offerings are the "index" of the service and therefore we concentrate on the contents section of the capabilities here.

The class `SosContents` simply contains a list of objects of the class `SosObservationOffering` which one can get directly from the connection object:

```
> sosOfferings(mySOS)
> sosOfferings(mySOS, name = "Rain")
```

The output when printing this list is quite extensive, so we concentrate on just on element of it in the following.

The offerings list is named with the offering identifier, so the following statements return the same list.

```
> sosOfferingIds(mySOS)
> names(sosOfferings(mySOS))
> sosName(sosOfferings(mySOS))
```

The offering identifier is is used in the example below to extract the offering description of temperature measurements. The offerings list is a standard R list, so all subsetting operations are possible.

**Note:** The order of the offering list (as all other lists, e.g. procedures or observed properties) is not guaranteed to be the same upon every connection to a service. So indexing by name (though counteracting the mentioned forward compatibility, as names might change) is recommended at at least one point in the analysis so that changes in the contents of a service result in an error.

```
> off.temp <- sosOfferings(mySOS)[["ATMOSPHERIC_TEMPERATURE"]]
```

```
Object of class SosObservationOffering;  id:  ATMOSPHERIC_TEMPERATURE , name:  Temperature
        time:  GmlTimePeriod: [ GmlTimePosition [ time: 2008-11-20 15:20:22 ]  -->  GmlTim
        procedure(s):  urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2
        observedProperty(s):  urn:ogc:def:property:OGC::Temperature
        feature(s)OfInterest:  urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-1
        responseFormat(s):  text/xml;subtype="om/1.0.0", application/zip , responseMode(s)
        intendedApplication:  NA
        resultModel(s):  ns:Measurement, ns:Observation
        boundedBy:  urn:ogc:def:crs:EPSG:4326, 46.611644 7.6103, 51.9412 13.883498
```

Metadata about the whole **offering** are identifier, name, and spatial and temporal extends.

```
> off.temp.id <- sosId(off.temp)

[1] "ATMOSPHERIC_TEMPERATURE"

> off.temp.name <- sosName(off.temp)

[1] "Temperature of the atmosphere"

> off.temp.boundedBy <- sosBoundedBy(off.temp)

$srsName
[1] "urn:ogc:def:crs:EPSG:4326"

$lowerCorner
[1] "46.611644 7.6103"

$upperCorner
[1] "51.9412 13.883498"
```

The offerings also contains metadata about the format and model that are supported.

```
> sosResultModels(off.temp)
```

```
     resultModel      resultModel
"ns:Measurement" "ns:Observation"
```

```
> sosResponseMode(off.temp)
```

```
    responseMode     responseMode
        "inline" "resultTemplate"
```

```
> sosResponseFormats(off.temp)
```

```
                 responseFormat                   responseFormat
"text/xml;subtype=\"om/1.0.0\""              "application/zip"
```

The structure of bounding box is not re-usable out of the box, as it simple returns a named list of lower and upper corner.

```
> str(sosBoundedBy(off.temp))
```

```
List of 3
 $ srsName    : chr "urn:ogc:def:crs:EPSG:4326"
 $ lowerCorner: chr "46.611644 7.6103"
 $ upperCorner: chr "51.9412 13.883498"
NULL
```

The optional attribute `bbox` can be used to obtain a bounding box matrix as used by package **sp**.

```
> off.temp.boundedBy.bbox <- sosBoundedBy(off.temp, bbox = TRUE)
```

```
                min      max
coords.lat 46.61164 51.94120
coords.lon  7.61030 13.88350
```

The **temporal** extend is modeled as an object of the respective class of the element in the offering description, which normally is a gml:TimePeriod, but does not have to be. The last two statements in the following snipped show how one can access the actual data and what their class is.

```
> off.temp.time <- sosTime(off.temp)
```

```
GmlTimePeriod: [ GmlTimePosition [ time: 2008-11-20 15:20:22 ]  -->  GmlTimePosition [ tim
```

```
> str(off.temp.time)
```

```
Formal class 'GmlTimePeriod' [package "sos4R"] with 9 slots
  ..@ begin        : NULL
  ..@ beginPosition:Formal class 'GmlTimePosition' [package "sos4R"] with 4 slots
  .. .. ..@ time                 : POSIXlt[1:1], format: "2008-11-20 15:20:22"
  .. .. ..@ frame                : chr NA
  .. .. ..@ calendarEraName      : chr NA
  .. .. ..@ indeterminatePosition: chr NA
  ..@ end          : NULL
  ..@ endPosition  :Formal class 'GmlTimePosition' [package "sos4R"] with 4 slots
```

```
  .. .. ..@ time                 : POSIXlt[1:1], format: "2011-01-05 18:00:00"
  .. .. ..@ frame                : chr NA
  .. .. ..@ calendarEraName      : chr NA
  .. .. ..@ indeterminatePosition: chr NA
  ..@ duration     : chr NA
  ..@ timeInterval : NULL
  ..@ frame        : chr NA
  ..@ relatedTimes : list()
  ..@ id           : chr NA
NULL
```

```
> off.temp.time@beginPosition@time
```

```
[1] "2008-11-20 15:20:22"
```

```
> off.temp.time@endPosition@time
```

```
[1] "2011-01-05 18:00:00"
```

```
> class(off.temp.time@endPosition@time)
```

```
[1] "POSIXt"  "POSIXlt"
```

The structure of these elements is very flexible and also not self-explanatory. Therefore the parameter `convert` can be used to try to create R objects and return these instead. Please be aware that this might not work for all services.

```
> off.temp.time.converted <- sosTime(off.temp, convert = TRUE)
```

```
$begin
[1] "2008-11-20 15:20:22"

$end
[1] "2011-01-05 18:00:00"
```

```
> str(off.temp.time.converted)
```

```
List of 2
 $ begin: POSIXlt[1:1], format: "2008-11-20 15:20:22"
 $ end  : POSIXlt[1:1], format: "2011-01-05 18:00:00"
NULL
```

Furthermore the offering comprises **lists of procedures, observed properties, and features of interest**. In our example the feature and procedure identifiers are the same — this does not have to be the case.

**Note:** The order of these lists is not guaranteed to be the same upon every connection to a service.

```
> sosProcedures(off.temp)
```

```
[1] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
[2] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
```

```
> sosObservedProperties(off.temp)

$observedProperty
[1] "urn:ogc:def:property:OGC::Temperature"

> sosFeaturesOfInterest(off.temp)

$featureOfInterest
[1] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"

$featureOfInterest
[1] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
```

All of the above can not only be requested for single offerings but also for complete SOS connections or for lists of offerings. The following examples only print out a part of the returned lists.

```
> sosProcedures(mySOS)[1:2]

$RAIN_GAUGE
[1] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
[2] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"

$LUMINANCE
[1] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
[2] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"

> sosObservedProperties(mySOS)[1:2]

$RAIN_GAUGE
$RAIN_GAUGE$observedProperty
[1] "urn:ogc:def:property:OGC::Precipitation1Hour"


$LUMINANCE
$LUMINANCE$observedProperty
[1] "urn:ogc:def:property:OGC::Luminance"

> sosFeaturesOfInterest(mySOS)[1:2]

$RAIN_GAUGE
$RAIN_GAUGE$featureOfInterest
[1] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"

$RAIN_GAUGE$featureOfInterest
[1] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"


$LUMINANCE
$LUMINANCE$featureOfInterest
[1] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"

$LUMINANCE$featureOfInterest
[1] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
```

For (parts of a) list of offerings:

```
> sosProcedures(sosOfferings(mySOS)[4:5])

     ATMOSPHERIC_PRESSURE
[1,] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
[2,] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
     ATMOSPHERIC_TEMPERATURE
[1,] "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
[2,] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
```

```
> sosObservedProperties(sosOfferings(mySOS)[4:5])

$ATMOSPHERIC_PRESSURE
$ATMOSPHERIC_PRESSURE$observedProperty
[1] "urn:ogc:def:property:OGC::BarometricPressure"


$ATMOSPHERIC_TEMPERATURE
$ATMOSPHERIC_TEMPERATURE$observedProperty
[1] "urn:ogc:def:property:OGC::Temperature"
```

```
> sosFeaturesOfInterest(sosOfferings(mySOS)[4:5])

                  ATMOSPHERIC_PRESSURE
featureOfInterest "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
featureOfInterest "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
                  ATMOSPHERIC_TEMPERATURE
featureOfInterest "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93"
featureOfInterest "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"
```

The results of these calls are lengthy and not included here. Please carefully inspect the structure in each case, as these functions will return named lists of lists and not combine procedures from different offerings. Consequently, some procedures could appear several times, but the association to the offering is still intact which is preferred at this stage.

### 5.3.2 Basic Request

```
> getObservation(sos = mySOS, ...)
```

The mandatory attributes are `sos`, `offering`, `observedProperty` and `responseFormat`. The other parameters are set to `NA` and not used when building the request.

Please see section 8.4.2 of the SOS specification for details and section 2 of this document for supported values of request parameters.

- `sos`: The service connection to be used, an object of class `SOS`.
- `offering`: The offering to be used, either the identifier as a character string or an object of class `SosObservationOffering`.

- `observedProperty`: The observed property of the desired observations. The default is all observed property of the offering, `sosObservedProperties(obj = offering)`.
- `responseFormat`: The format of the response document. The default is text/xml;subtype='om/1.0.0'.
- `srsName`: The name of the spatial reference system that should be used for the geometries in the response.
- `eventTime`: A list of objects of class `SosEventTime` which specify the time period(s) for which observations are requested. See section 5.3.5 for more information.
- `procedure`: A list of procedure identifiers for which observations are requested. See section 5.3.6 for more information.
- `featureOfInterest`: An object of class `SosFeatureOfInterest` which specifies the feature for which observations are requested. See sections 5.3.7 and 5.3.6 for more information.
- `result`: An object of class `OgcComparisonOps` for result filtering with filter expressions from Filter Encoding. See section 5.3.8 for more information.
- `resultModel`: The qualified XML name of the root element of the response, e.g. `om:Measurement`. The available models of a service can be found in the service metadata using `sosResultModel(...)`.
- `responseMode`: The response mode defines the form of the response, e.g. inline, out-of-band, or attached. The available models of a service can be found in the service metadata using `sosResponseMode(...)`.
- `BBOX`: A bounding box to be used only in HTTP GET connections (parameter is discarded for POST connections). The format must one character string with `minlon,minlat,maxlon,maxlat,srsURI?`, the spatial reference system is optional.
- `latest`: A boolean parameter to request the latest observation only (see example below) — this is not standard conform but only supported by **52°North** SOS.
- `saveOriginal`: Saves a copy of the response document in the current working directory. See section 5.4 for an example.

A request to retrieve the latest measured value is also possible, although not (!) standard conform. **52°North** SOS realizes this specific request by requesting a sampling time with the fixed value "latest".

```
> obs.temp.latest <- getObservation(sos = mySOS, offering = off.temp,
+                   latest = TRUE)
```

```
Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 2 observation(s) having 2 result values [ 1, 1 ].
```

The returned data is an XML document of type om:Observation, om:Measurement, or om:ObservationCollection which holds a list of the former two. All three of these have corresponding S4 classes, namely `OmObservation`, `OmMeasurement`, or `OmObservationCollection`.

### 5.3.3 Response Subsetting

Subsetting of elements in an `OmObservationCollection` can be done just like in a normal list (in fact, it just wraps at list of observations at this point), i.e. with the operators `[` and `[[`.

```
> length(obs.temp.latest)

[1] 2

> obs.temp.latest[[1]]

Object of class OmObservation;
        procedure: urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
        observedProperty: NA
        foi: urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
        samplingTime: GmlTimePeriod: [ GmlTimePosition [ time: 2009-09-28 13:45:00 ]  -->
        result dimensions: 1, 3

> obs.temp.latest[2:3]

$OmObservation
Object of class OmObservation;
        procedure: urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
        observedProperty: NA
        foi: urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
        samplingTime: GmlTimePeriod: [ GmlTimePosition [ time: 2011-01-05 18:00:00 ]  -->
        result dimensions: 1, 3

$<NA>
NULL
```

Addionally, indexing is also possible via identifiers of procedure, observed property, and feature of interest.

```
> index.foiId <- sosFeatureIds(obs.temp.latest)[[1]]
> index.foiId

[1] "urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111"

> obs.temp.latest[index.foiId]

$OmObservation
Object of class OmObservation;
        procedure: urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
        observedProperty: NA
        foi: urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
        samplingTime: GmlTimePeriod: [ GmlTimePosition [ time: 2009-09-28 13:45:00 ]  -->
        result dimensions: 1, 3
```

```
> index.obsProp <- sosObservedProperties(off.temp)
> obs.temp.latest[index.obsProp]

list()

> index.proc <- sosProcedures(obs.temp.latest)[1:4]
> index.proc.alternative1 <- sosProcedures(off.temp)[1:4]
> index.proc.alternative2 <- sosProcedures(mySOS)
> obs.temp.latest[index.proc]

$OmObservation
Object of class OmObservation;
        procedure: urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
        observedProperty: NA
        foi: urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
        samplingTime: GmlTimePeriod: [ GmlTimePosition [ time: 2009-09-28 13:45:00 ]  -->
        result dimensions: 1, 3

$OmObservation
Object of class OmObservation;
        procedure: urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
        observedProperty: NA
        foi: urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
        samplingTime: GmlTimePeriod: [ GmlTimePosition [ time: 2011-01-05 18:00:00 ]  -->
        result dimensions: 1, 3
```

### 5.3.4 Result Extraction

**Data Values** can be extracted from observations, measurements and observation collections with the function `sosResult(...)`. The function returns an object of class `data.frame`. In the case of collections, it automatically binds the data frames (you can turn this off by adding `bind = FALSE` as a parameter). Additional metadata, like units or definitions, is accessible via `attributes(...)` for every column of the data frame.

```
> obs.temp.latest.result.2 <- sosResult(obs.temp.latest[[2]])

                 Time
1 2011-01-05 18:00:00
                                                                     feature
1 urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
  urn:ogc:def:property:OGC::Temperature
1                                  -2.9

> obs.temp.latest.result.2

                 Time
1 2011-01-05 18:00:00
                                                                     feature
1 urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
  urn:ogc:def:property:OGC::Temperature
1                                  -2.9
```

```
> obs.temp.latest.result <- sosResult(obs.temp.latest[1:2])

                                Time
OmObservation  2009-09-28 13:45:00
OmObservation1 2011-01-05 18:00:00
                                                                              feature
OmObservation  urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
OmObservation1 urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
               urn:ogc:def:property:OGC::Temperature
OmObservation                                   20.3
OmObservation1                                  -2.9

> obs.temp.latest.result

                                Time
OmObservation  2009-09-28 13:45:00
OmObservation1 2011-01-05 18:00:00
                                                                              feature
OmObservation  urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
OmObservation1 urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
               urn:ogc:def:property:OGC::Temperature
OmObservation                                   20.3
OmObservation1                                  -2.9

> temperature.attrs <- attributes(
+          obs.temp.latest.result[["urn:ogc:def:property:OGC::Temperature"]])

$name
[1] "urn:ogc:def:property:OGC::Temperature"

$definition
[1] "urn:ogc:def:property:OGC::Temperature"

$`unit of measurement`
[1] "Cel"
```

**Spatial Information** can be stored in an observation in several ways: (i) as a usual data attribute which is directly contained in the result `data.frame`, (ii) within a feature collection in the observation. In the latter case the utility functions `sosCoordinates(...)` and `sosFeatureIds(...)` can be used to extract the coordinates respectively the identifiers from `OmObservationCollection` or `OmObservation` classes. A variety of feature types gml:Point or sa:SamplingPoint are supported by `sosCoordinates(...)`.

```
> obs.temp.latest.foiIDs <- sosFeatureIds(obs.temp.latest)
> obs.temp.latest.coordinates.all <- sosCoordinates(obs.temp.latest)
> obs.temp.latest.coordinates.1 <- sosCoordinates(obs.temp.latest[[1]])
```

An observation collection also contains a bounding box of the contained observations, which can be extracted with the function `sosBoundedBy(...)`. The optional attribute `bbox` can be used to obtain a bounding box matrix as used by package **sp**.

```
> sosBoundedBy(obs.temp.latest)

$srsName
[1] "urn:ogc:def:crs:EPSG:4326"

$lowerCorner
[1] "46.611644 7.6103"

$upperCorner
[1] "51.9412 13.883498"

> sosBoundedBy(obs.temp.latest, bbox = TRUE)

                min      max
coords.lat 46.61164 51.94120
coords.lon  7.61030 13.88350
```

The combination of data values and coordinates strongly depends on the use case and existing spatial information. In the case of coordinates encoded in the features, a matching of the two data frames can easily be accomplished with the function `merge()`.

```
> obs.temp.latest.coords <- sosCoordinates(obs.temp.latest)
> result.names <- names(obs.temp.latest.result)
> coords.names <- names(obs.temp.latest.coords)
> print(toString(result.names))

[1] "Time, feature, urn:ogc:def:property:OGC::Temperature"

> print(toString(coords.names))

[1] "lat, lon, SRS, feature"

> # Manually merging though not neccessary:
> obs.temp.latest.coords <- sosCoordinates(obs.temp.latest)
> obs.temp.latest.data <- merge(
+         x = obs.temp.latest.result,
+         y = obs.temp.latest.coords,
+         by.x = result.names[[2]],
+         by.y = coords.names[[4]])
```

The default column name for the feature identifiers is `feature`. If the name of the feature identifier attribute in the data table matches (which is the case for **52°North** SOS), `merge` does not need additional information. In that case, the merging reduces to the following code:

```
> obs.temp.latest.data <- merge(x = obs.temp.latest.result,
+         y = obs.temp.latest.coords)
> print(obs.temp.latest.data)

                                                             feature
1 urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
```

```
2 urn:ogc:object:feature:OSIRIS-HWS:efeb807b-bd24-4128-a920-f6729bcdd111
                 Time urn:ogc:def:property:OGC::Temperature      lat      lon
1 2011-01-05 18:00:00                                  -2.9 51.94120  7.61030
2 2009-09-28 13:45:00                                  20.3 46.61164 13.88350
                       SRS
1 urn:ogc:def:crs:EPSG:4326
2 urn:ogc:def:crs:EPSG:4326
```

And in that case, you can even save that step by specifying the attribute `coordinates` of the function `sosResult` which includes the merge of data values and coordinates.

```
> sosResult(obs.temp.latest, coordinates = TRUE)
```

### 5.3.5 Temporal Filtering

The possibly most typical temporal filter is a period of time for which measurements are of interest.

```
> # temporal interval creation based on POSIXt classes
> lastWeek.period <- sosCreateTimePeriod(sos = mySOS,
+         begin = (Sys.time() - 3600 * 24 * 7), end = Sys.time())
> lastWeek.eventTime <- sosCreateEventTimeList(lastWeek.period)
```

Please note that the create function `sosCreateEventTimeList()` also wraps the created objects in a list as expected by the method `getObservation(...)`.

What was the average temperature during the last week?

```
> obs.lastWeek <- getObservation(sos = mySOS,
+         offering = off.temp,
+         procedure = sosProcedures(off.temp),
+         eventTime = lastWeek.eventTime)
```

```
Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 1 observation(s) having 671 result values [ 671 ].
```

```
> obs.temp.lastWeek.result <- sosResult(obs.lastWeek)
> summary(obs.temp.lastWeek.result)[
+         4,"urn:ogc:def:property:OGC::Temperature"]
```

```
[1] "Mean   :-1.188  "
```

The default temporal operator is "during", but others are supported as well (see section 2). The next example shows how to create a temporal filter for all observations taken **after** a certain point in time. Here the creation function creates just one object of class `SosEventTime` which must be added to a list manually before passing it to `getObservation(...)`.

```
> lastDay.instant <- sosCreateTimeInstant(
+         time = as.POSIXct(Sys.time() - 3600 * 24), sos = mySOS)
> lastDay.eventTime <- sosCreateEventTime(time = lastDay.instant,
+         operator = SosSupportedTemporalOperators()[["TM_After"]])
> print(lastDay.eventTime)
```

```
Object of class SosEventTime:
        TM_After: GmlTimePosition [ time: 2011-01-04 18:17:00 ]
```

### 5.3.6 Spatial Filtering

The possibly most typical spatial filter is a bounding box[12] within which measurements of interest must have been made. Here the creation function returns an object of class `OgcBBOX`, which can be wrapped in an object of class `SosFeatureOfInterest`, which is passed into the get-observation call.

```
> sept09.period <- sosCreateTimePeriod(sos = mySOS,
+         begin = as.POSIXct("2009-09-01 00:00"),
+         end = as.POSIXct("2009-09-30 00:00"))
> sept09.eventTimeList <- sosCreateEventTimeList(sept09.period)
> obs.sept09 <- getObservation(sos = mySOS,
+                 offering = off.temp,
+                 eventTime = sept09.eventTimeList)

Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 2 observation(s) having 5291 result values [ 2647, 2644 ].

> request.bbox <- sosCreateBBOX(lowLat = 50.0, lowLon = 5.0,
+                 uppLat = 55.0, uppLon = 10.0,
+                 srsName = "urn:ogc:def:crs:EPSG:4326")
> request.bbox.foi <- sosCreateFeatureOfInterest(spatialOps = request.bbox)
> obs.sept09.bbox <- getObservation(sos = mySOS,
+         offering = off.temp,
+         featureOfInterest = request.bbox.foi,
+         eventTime = sept09.eventTimeList)

Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 1 observation(s) having 2644 result values [ 2644 ].
```

Unfiltered versus spatially filtered coordinates of the responses:

```
> print(sosCoordinates(obs.sept09)[,1:2])

                   lat      lon
OmObservation  46.61164 13.88350
OmObservation1 51.94120  7.61030

> print(sosCoordinates(obs.sept09.bbox)[,1:2])

                  lat    lon
OmObservation 51.9412 7.6103
```

More advanced spatial filtering, for example based on arbitrary shapes et cetera, is currently not implemented. This could be implemented by implementing subclasses for `GmlGeometry` (including encoders) which must be wrapped in `OgcBinarySpatialOp` which extends `OgcSpatialOps` and can therefore be added to an object of class `SosFeatureOfInterest` as the spatial parameter.

[12] http://en.wikipedia.org/wiki/Bounding_box

### 5.3.7 Feature Filtering

The feature can not only be used for spatial filtering, but also to query specific FOIs. The following example extracts the identifiers from an offering and then creates an object of class `SosFeatureOfInterest`, which is passed into the get-observation call.

```
> off.temp.fois <- sosFeaturesOfInterest(off.temp)
> request.fois <- sosCreateFeatureOfInterest(
+         objectIDs = list(off.temp.fois[[1]]))
> encodeXML(obj = request.fois, sos = mySOS)
```

```
<sos:featureOfInterest>
 <sos:ObjectID>urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93</sos
</sos:featureOfInterest>
```

```
> obs.lastWeek.fois <- getObservation(sos = mySOS, offering = off.temp,
+         featureOfInterest = request.fois, eventTime = lastWeek.eventTime)
```

```
Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 1 observation(s) having 671 result values [ 671 ].
```

```
> print(sosFeaturesOfInterest(obs.lastWeek.fois))
```

```
$OmObservation
Object of class GmlFeatureCollection; id: NA; 1 featureMembers: <S4 object of class "GmlFe
```

### 5.3.8 Value Filtering

Value Filtering is realized via the slot `result` in a GetObservation request. The filtering in the request is based on comparison operators and operands specified by OGC Filter Encoding (Vretanos, 2005).

The classes and methods of this specification are not yet implemented, but manual definition of the XML elements is possible with the methods of the package **XML**.

The following code example uses a literal comparison of a property:

```
> # result filtering
> filter.value <- -2.3
> filter.propertyname <- xmlNode(name = ogcPropertyNameName,
+                 namespace = ogcNamespacePrefix)
> xmlValue(filter.propertyname) <- "urn:ogc:def:property:OGC::Temperature"
> filter.literal <- xmlNode(name = "Literal",
+                 namespace = ogcNamespacePrefix)
> xmlValue(filter.literal) <- as.character(filter.value)
> filter.comparisonop <- xmlNode(
+         name = ogcComparisonOpGreaterThanName,
+         namespace = ogcNamespacePrefix,
+         .children = list(filter.propertyname,
+         filter.literal))
> filter.result <- xmlNode(name = sosResultName,
+         namespace = sosNamespacePrefix,
+         .children = list(filter.comparisonop))
```

Please consult to the extensive documentation of the **XML** package for details. The commands above result in the following output which is inserted into the request without further processing.

```
> print(filter.result)
```

```
<sos:result>
 <ogc:PropertyIsGreaterThan>
  <ogc:PropertyName>urn:ogc:def:property:OGC::Temperature</ogc:PropertyName>
  <ogc:Literal>-2.3</ogc:Literal>
 </ogc:PropertyIsGreaterThan>
</sos:result>
NULL
```

Any object of class `OgcComparisonOpsOrXMLOrNULL`, which includes the class of the object returned by `xmlNode(...)`, i.e. `XMLNode`. These object can be used in the GetObservation request as the `result` parameter.

First, we request the unfiltered values for comparison, then again with the filter applied. The length of the returned results is compared in the end.

```
> # request  values for the last week.
> obs.lastWeek <- getObservation(sos = mySOS,
+         eventTime = lastWeek.eventTime,
+         offering = sosOfferings(mySOS)[["ATMOSPHERIC_TEMPERATURE"]])
```

```
Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 1 observation(s) having 671 result values [ 671 ].
```

```
> # request  values for the week with a value higher than 0 degrees.
> obs.lastWeek.filter <- getObservation(sos = mySOS,
+         eventTime = lastWeek.eventTime,
+         offering = sosOfferings(mySOS)[["ATMOSPHERIC_TEMPERATURE"]],
+         result = filter.result)
```

```
Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 1 observation(s) having 482 result values [ 482 ].
```

```
> print(paste("Filtered:", dim(sosResult(obs.lastWeek.filter))[[1]],
+         "-vs.- Unfiltered:", dim(sosResult(obs.lastWeek))[[1]]))
```

```
[1] "Filtered: 482 -vs.- Unfiltered: 671"
```

#### 5.3.9 Result Exporting

A tighter integration with data structures of packages **sp** or **spacetime** (both available on CRAN) is planned for the future. Please consult the developers for the current status.

As an example the following code creates a `SpatialPointsDataFrame` (can only contain one data value per position!) based on the features of a result.

```
> library("sp")
> obs.lastWeek <- getObservation(sos = mySOS,
+         offering = off.temp,
+         procedure = sosProcedures(off.temp),
+         eventTime = lastWeek.eventTime)
```

```
Finished getObservation to http://v-swe.uni-muenster.de:8080/WeatherSOS/sos
        --> received 1 observation(s) having 671 result values [ 671 ].
```

```
> # Create SpatialPointsDataFrame from result features
> coords <- sosCoordinates(obs.lastWeek[[1]])
> crs <- sosGetCRS(obs.lastWeek[[1]])
> spdf <- SpatialPointsDataFrame(coords = coords[,1:2],
+         data = data.frame(coords[,4]), proj4string = crs)
> str(spdf)
```

```
Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame':        1 obs. of  1 variable:
  .. ..$ coords...4.: Factor w/ 1 level "urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4
  ..@ coords.nrs : num(0)
  ..@ coords     : num [1, 1:2] 51.94 7.61
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "lat" "lon"
  ..@ bbox       : num [1:2, 1:2] 51.94 7.61 51.94 7.61
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2] "lat" "lon"
  .. .. ..$ : chr [1:2] "min" "max"
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. ..@ projargs: chr "+init=epsg:4326"
```

## 5.4 GetObservationById

The operation GetObservationById is defined in clause 10.1 of the SOS specification and not part of the core profile. But it is implemented as it is quite simple. The response is the same as described in the previous section. Optional parameters, and their defaults and supported values, are the same as in GetObservation requests.

In this case the returned observation collection contains an om:Measurement element, which contains just one measured value and is parsed to an object of class `OmMeasurement`. The result extraction works the same as with om:Observation objects.

```
> obsId <- getObservationById(sos = mySOS, observationId = "o_3508493")
```

```
Object of class OmMeasurement, procedure urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4
        featureOfInterest: Object of class GmlFeatureProperty, href: NA, feature: Object o
srsName: NA, srsDimension: NA, srsDimension: NA, uomLabels: NA, relatedObservation: NA, re
        sampledFeatures: list(sampledFeature = <pointer: 0x6738210>);
        samplingTime: GmlTimePosition [ time: 2010-12-26 12:45:00 ];
        result: Object of class GmlMeasure; value: 1014; uom: hPa
```

```
> # str(obsId)
> # obs[[1]]
> sosResult(obsId, coordinates = TRUE)
```

```
              urn:ogc:def:property:OGC::BarometricPressure     lat    lon
OmMeasurement                                         1014 51.9412 7.6103
```

```
                                            SRS
OmMeasurement urn:ogc:def:crs:EPSG:4326
                                                                                       feature
OmMeasurement urn:ogc:object:feature:OSIRIS-HWS:3d3b239f-7696-4864-9d07-15447eae2b93
```

Just as for `getObservation()` you can save the orginal response document. It is saved into the current working directory and the name starts with the observation identifier. You can also read it back using the function `sosParse()`.

```
> obsId <- getObservationById(sos = mySOS, observationId = "o_3508493",
+                  saveOriginal = TRUE)
> .files <- list.files(getwd())
> .observationFiles <- c()
> for(.f in .files) { # %in% not working with Sweave
+         if(length(grep("^o_", .f, value=TRUE)) > 0)
+                 .observationFiles <- c(.observationFiles, .f)
+ }
> obsId <- parseFile(sos = mySOS, file = .observationFiles[[1]])
```

# 6 Changing Handling Functions

The flexibility of the specifications that model the markup requests and responses, especially the observation encoding, is too high to handle all possible cases within **sos4R**. Thus an equally flexible mechanism for users to adopt the steps of encoding and decoding documents to their needs is needed.

The process of data download comprises (i) building request, (ii) encoding requests, (iii) sending and receiving data, (iv) decoding responses, and (v) applying the correct R data type to the respective data values. This can be seen as a fixed, ordered workflow a user has to follow where each step build upon the input of the previous. To ensure flexibility within these steps of the workflow but also to maximize reusability of existing functionality, a mechanism to exchange the functions that are used in these steps is provided.

Step (i), the building of requests, i.e. the assembly of the request parameters into an R object, is documented in section 5.3. Step (iii), the sending of the sending and receiving of documents to respectively from a service, does not need to be influenced directly but the user (apart from the connection method).

In the remainder of this section it is explained how this applies to the steps (ii), (iv) and (v) of the fixed workflow.

## 6.1 Include and Exclude Functions

The functions used in the exchangeable steps are organized in lists. To base your own list of functions on the existing ones, thereby not having to start from scratch, you can combine the default list of functions with your own. Use the following functions:

To add your own function, simply add it as a named argument. You can add as many as you like in the `...` parameter. If a function with that identifier already exists in the default list it will be replaced by your function. For further adjustments you can explicitly include and exclude functions by identifier. Please be aware that inclusion is applied first, then exclusion. It is also important that you also have to include that functions you just added manually!

Examples of function list generation with parsing functions:

```
> parsers <- SosParsingFunctions(
+         "ExceptionReport" = function() {return("Got Exception!")},
+         include = c("GetObservation", "ExceptionReport"))
> print(names(parsers))
```

```
[1] "GetObservation"  "ExceptionReport"
```

```
> parsers <- SosParsingFunctions(
+                 "ExceptionReport" = function() {return("Got Exception!")},
+                 include = c("GetCapabilities"))
> print(names(parsers))
```

```
[1] "GetCapabilities"
```

```
> parsers <- SosParsingFunctions(
+                 exclude = names(SosParsingFunctions())[1:15])
> print(names(parsers))
```

```
[1] "CategoryObservation"                "CountObservation"
[3] "TruthObservation"                   "TemporalObservation"
[5] "ComplexObservation"                 "text/csv"
[7] "text/xml;subtype=\"om/1.0.0&quot\""
```

## 6.2 Encoders

The current list of a connection's encoders can be accessed with

```
> sosEncoders(mySOS)
```

A complete list of the existing encoders names:

```
> names(sosEncoders(mySOS))
```

```
[1] "GET"  "POST" "SOAP"
```

Here the idea of organizing the encoding functions becomes clear: One base encoding function is given, which is a generic method that must exist for alle elements that need to be encoded.

```
> myPostEncoding <- function(object, sos, verbose) {
+         return(str(object))
+ }
> # Will fail:
> mySOS2 = SOS(sosUrl(mySOS),
+         encoders = SosEncodingFunctions("POST" = myPostEncoding))
```

## 6.3 Parsers/Decoders

The terms parsing and decoding are used as synonyms for the process of processing an XML document to create an R object. XML documents are made out of hierarchical elements. That is why the parsing functions are organized in a listed, whose names are the elements' names that can be parsed.

The current list of a connection's parsers can be accessed with the following function.

```
> sosParsers(mySOS)
```

A complete list of the elements with existing encoders is shown below.

```
> names(sosParsers(mySOS))

 [1] "GetCapabilities"                     "DescribeSensor"
 [3] "GetObservation"                      "GetObservationById"
 [5] "ExceptionReport"                     "Measurement"
 [7] "member"                              "Observation"
 [9] "ObservationCollection"               "result"
[11] "DataArray"                           "elementType"
[13] "encoding"                            "values"
[15] "GeometryObservation"                 "CategoryObservation"
[17] "CountObservation"                    "TruthObservation"
[19] "TemporalObservation"                 "ComplexObservation"
[21] "text/csv"                            "text/xml;subtype=\"om/1.0.0&quot\""
```

Here the idea of organizing the encoding functions becomes clear: For every XML element that must be parsed there is a function given in the list.

If you want to replace only selected parsers use the include parameter as described above. You can also base your own parsing functions on a variety of existing parsing functions. For example you can replace the base function for om:ObservationCollectionm, named `ObservationCollection`, but still use the parsing function for om:Observation within your own function if you include it in the parser list. The existing parsing functions are all named in the pattern `parse[ElementName](...)`. Please be aware that some parsers contain require a parameter of class SOS upon which they rely for encoding information.

```
> # Create own parsing function:
> myER <- function(xml) {
+         return("EXCEPTION!!!11")
+ }
> myParsers <- SosParsingFunctions("ExceptionReport" = myER)
> mySOS2 <- SOS(sosUrl(mySOS), parsers = myParsers)

Created SOS for URL http://v-swe.uni-muenster.de:8080/WeatherSOS/sos

> # Triggers exception:
> response <- getObservation(mySOS2,
+         offering = sosOfferings(mySOS2)[[1]],
+         observedProperty = list("Bazinga"))
> print(response)
```

```
[1] "EXCEPTION!!!11"
```

To disable all parsing, you can use the following function. This just “passes through” all received data.

```
> SosDisabledParsers()
```

This list only contains the top-most parsing functions for the operation responses and exception reports. This is also the recommended way to start if you want to set-up your own parsers. This is also a way of directly inspecting the responses.

The next example shows how the response (in this case the request is intentionally incorrent and triggers an exception) is passed through as an object of class `XMLInternalDocument`:

```
> names(SosDisabledParsers())

[1] "GetCapabilities"    "DescribeSensor"     "GetObservation"
[4] "GetObservationById" "ExceptionReport"

> mySOS2 <- SOS(sosUrl(mySOS), parsers = SosDisabledParsers())

Created SOS for URL http://v-swe.uni-muenster.de:8080/WeatherSOS/sos

> response <- getObservation(mySOS2,
+         offering = sosOfferings(mySOS2)[[1]],
+         observedProperty = list("Bazinga"))
> class(response)

[1] "XMLInternalDocument"

> print(xmlName(xmlRoot(response))) # XML functions

[1] "ExceptionReport"
```

### 6.4 Data Converters

A list of named functions to be used by the parsing methods to convert data values to the correct R type, which are mostly based on the unit of measurement[13] code.

The conversion functions always take two parameters: `x` is the object to be converted, `sos` is the service where the request was received from.

The available functions are basically wrappers for coercion functions, for example `as.double()`. The only method exploiting the second argument is the one for conversion of time stamps which uses the time format saved with the object of class `SOS` in a call to `strptime`.

```
> value <- 2.0
> value.string <- sosConvertString(x = value, sos = mySOS)
> print(class(value.string))
```

[13] http://en.wikipedia.org/wiki/Units_of_measurement

```
[1] "character"
```

```
> value <- "2.0"
> value.double <- sosConvertDouble(x = value, sos = mySOS)
> print(class(value.double))
```

```
[1] "numeric"
```

```
> value <- "1"
> value.logical <- sosConvertLogical(x = value, sos = mySOS)
> print(class(value.logical))
```

```
[1] "logical"
```

```
> value <- "2010-01-01T12:00:00.000"
> value.time <- sosConvertTime(x = value, sos = mySOS)
> print(class(value.time))
```

```
[1] "POSIXt"  "POSIXct"
```

The full list of currently supported units can be seen below. It mostly contains common numerical units which are converted to type `double`.

```
> names(SosDataFieldConvertingFunctions())
```

```
 [1] "urn:ogc:data:time:iso8601"       "urn:ogc:property:time:iso8601"
 [3] "urn:ogc:phenomenon:time:iso8601" "time"
 [5] "m"                               "s"
 [7] "g"                               "rad"
 [9] "K"                               "C"
[11] "cd"                              "%"
[13] "ppth"                            "ppm"
[15] "ppb"                             "pptr"
[17] "mol"                             "sr"
[19] "Hz"                              "N"
[21] "Pa"                              "J"
[23] "W"                               "A"
[25] "V"                               "F"
[27] "Ohm"                             "S"
[29] "Wb"                              "Cel"
[31] "T"                               "H"
[33] "lm"                              "lx"
[35] "Bq"                              "Gy"
[37] "Sv"                              "gon"
[39] "deg"                             "'"
[41] "''"                              "l"
[43] "L"                               "ar"
[45] "t"                               "bar"
[47] "u"                               "eV"
[49] "AU"                              "pc"
[51] "degF"                            "hPa"
[53] "mm"                              "nm"
```

```
[55] "cm"                               "km"
[57] "m/s"                              "kg"
[59] "mg"                               "uom"
[61] "urn:ogc:data:feature"
```

The current list of a connection's converters can be accessed with

```
> sosDataFieldConverters(mySOS)
```

The following connection shows a typical workflow of connecting to a new SOS for the first time, what the errors for missing converters look like, and how to add them to the SOS connection.

```
> # GET connection
> MBARI <- SOS("http://mmisw.org/oostethys/sos",
+         method = SosSupportedConnectionMethods()[["GET"]])

Created SOS for URL http://mmisw.org/oostethys/sos

> myOff <- sosOfferings(MBARI)[[1]]
> myProc <- sosProcedures(MBARI)[[1]]
> mbariObs1 <- try(
+         getObservation(sos = MBARI, offering = myOff,
+                 procedure = myProc)
+ )
> warnings()

NULL
```

There are warnings about unknown units of measurement and a swe:Quantity element (which describes a numeric field) without a given unit of measurement (which it should have as a numeric field). The next example creates conversion functions for these and subsequently results in more fields in the final result.

```
> myConverters <- SosDataFieldConvertingFunctions(
+         "S/m" = sosConvertDouble,
+         "http://mmisw.org/ont/cf/parameter/sea_water_salinity" = sosConvertDouble)
> MBARI <- SOS("http://mmisw.org/oostethys/sos",
+         method = SosSupportedConnectionMethods()[["GET"]],
+         dataFieldConverters = myConverters)

Created SOS for URL http://mmisw.org/oostethys/sos

> mbariObs2 <- getObservation(sos = MBARI, offering = myOff,
+         procedure = myProc)

> toString(names(sosResult(mbariObs1)))

[1] ""

> toString(names(sosResult(mbariObs2)))

[1] ""
```

# 7 Exception Handling

When working with sos4R, two kinds of errors must be handled: service exceptions and package errors. The former can occur when a request is invalid or a service encounters internal exceptions. The latter can mean a bug or illegal settings within the package. To understand both types of errorenous states, this sections explains the contents of the exception reports returned by the service and the functionalities to investigate the inner workings of the package.

## 7.1 OWS Service Exceptions

The service exceptions returned by a SOS are described in OGC Web Services Common (Whiteside, 2007) clause 8. The classes to handle the returned exceptions in sos4R are `OwsExceptionReport`, which contains a list of exception reports, and `OwsException`, which contains slots for the parameters exception text(s), exception code, and locator. These are defined as follows and can be implementation specific.

**ExceptionText** Text describing specific exception represented by the exceptionCode.

**exceptionCode** Code representing type of this exception.

**locator** Indicator of location in the client's operation request where this exception was encountered.

The standard exception codes and meanings are accessible by calling

```
> OwsExceptionsData()
```

directly in **sos4R** and are shown in table **??**. The original table also contains the respective HTTP error codes and messages.

```
> response <- try(getObservationById(sos = mySOS,
+                            observationId = "obs1000", inspect = TRUE))
```

```
*** POST! REQUEST:
<sos:GetObservationById xsi:schemaLocation="http://www.opengis.net/sos/1.0 http://schemas.
 <sos:ObservationId>obs1000</sos:ObservationId>
 <sos:responseFormat>text/xml;subtype=&quot;om/1.0.0&quot;</sos:responseFormat>
</sos:GetObservationById>
*** RESPONSE:
 <?xml version="1.0" encoding="UTF-8"?>
<ows:ExceptionReport version="1.0.0" xsi:schemaLocation="http://schemas.opengis.net/ows/1.
  <ows:Exception exceptionCode="InvalidParameterValue" locator="ObservationId">
    <ows:ExceptionText>The GetObservationById operation is only supported for single obser
  </ows:Exception>
</ows:ExceptionReport>
** RESPONSE DOC:
<?xml version="1.0" encoding="UTF-8"?>
<ows:ExceptionReport xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xsi="http://www.w3.o
  <ows:Exception exceptionCode="InvalidParameterValue" locator="ObservationId">
```

| exceptionCode | meaningOfCode | locator |
|---|---|---|
| OperationNotSupported | Request is for an operation that is not supported by this server | Name of operation not supported |
| MissingParameterValue | Operation request does not include a parameter value, and this server did not declare a default parameter value for that parameter | Name of missing parameter |
| InvalidParameterValue | Operation request contains an invalid parameter value | Name of parameter with invalid value |
| VersionNegotiationFailed | List of versions in 'AcceptVersions' parameter value in GetCapabilities operation request did not include any version supported by this server | None, omit 'locator' parameter |
| InvalidUpdateSequence | Value of (optional) updateSequence parameter in GetCapabilities operation request is greater than current value of service metadata updateSequence number | None, omit 'locator' parameter |
| OptionNotSupported | Request is for an option that is not supported by this server | Identifier of option not supported |
| NoApplicableCode | No other exceptionCode specified by this service and server applies to this exception | None, omit 'locator' parameter |

Table 2: Exception Data Table (without HTTP columns).

```
    <ows:ExceptionText>The GetObservationById operation is only supported for single obser
  </ows:Exception>
</ows:ExceptionReport>

Object of class OwsExceptionReport; version: 1.0.0; lang: NA;
 1 exception(s) (code @ locator : text):
  InvalidParameterValue @ ObservationId : The GetObservationById operation is only support
```

If an exception is received then it is also saved as a warning message.

## 7.2 Inspect Requests and Verbose Printing

The package offers two levels of inspection of the ongoing operations indicated by two boolean parameters, `inspect` and `verbose`. These are available in all service operation calls.

**inspect** prints the raw requests and responses to the console.

**verbose** prints not only the requests, but also debugging statements.

The option `verboseOutput` when using the method `SOS(...)` turns on the verbose setting for all subsequent requests made to the created connection unless deactivated in an operation call. By using `verboseOutput` you can also debug the automatic GetCapabilities operations when creating a new SOS connections.

The output with these parameters enabled is too extensive to show within this document.

```
> off.4 <- sosOfferings(mySOS)[[4]]
> getObservation(sos = mySOS, offering = off.4,
+         procedure = sosProcedures(off.4)[[1]], latest = TRUE,
+         inspect = TRUE)
> getObservation(sos = mySOS,        offering = off.4,
+         procedure = sosProcedures(off.4)[[1]], latest = TRUE,
+         vebose = TRUE)
> verboseSOS <- SOS(sosUrl(mySOS), verboseOutput = TRUE)
```

## 8 Getting Started

The **demos** are a good way to get started with the package. Please be aware that the used SOSs might be temporarily unavailable.

```
> demo(package = "sos4R")
```

Additionally, there is a list of services on the project homepage (http://www.nordholmen.net/sos4r/data/) and a few SOS URLs are available via the function `SosExampleServices()`.

```
> SosExampleServices()

$`52 North SOS: Weather Data, station at IFGI, Muenster, Germany`
[1] "http://v-swe.uni-muenster.de:8080/WeatherSOS/sos"

$`52 North SOS: Water gauge data for Germany`
[1] "http://v-sos.uni-muenster.de:8080/PegelOnlineSOSv2/sos"

$`52 North SOS: Air Quality Data for Europe`
[1] "http://v-sos.uni-muenster.de:8080/AirQualityEurope/sos"

$`OOTethys SOS: Marine Metadata Interoperability Initiative (MMI)`
[1] "http://mmisw.org/oostethys/sos"

$`OOTethys SOS: Gulf of Maine Ocean Observing System SOS`
[1] "http://www.gomoos.org/cgi-bin/sos/oostethys_sos.cgi"
```

## 9 Getting Support

If you want to ask questions about using the software, please go first to the **52°North forum** for the geostatistics community at http://geostatistics.forum.52north.org/ and check if a solution is described there. If you are a frequent user please consider subscribing to the geostatistics **mailing list** (http:

//list.52north.org/mailman/listinfo/geostatistics) which is linked to the forum.

# 10 Developing sos4R

### Code Repository

You can download (and also browse) the source code of **sos4R** directly from the **52 °North** repository:

- **SVN resource URL**: https://svn.52north.org/svn/geostatistics/main/sos4R. Please read the documentation (especially the posting guide) of the **52 °North** repositories[14]. Anonymous access for download is possible.
- **Web access**: https://svn.52north.org/cgi-bin/viewvc.cgi/main/sos4R/?root=geostatistics

### Developer Documentation

See the developer documentation at the **52 °North** Wiki for detailed information on how to use the checked out source project: https://wiki.52north.org/bin/view/Geostatistics/Sos4R. You will find a detailed description of the folder and class structure, the file naming scheme, and an extensive list of tasks for future development.

Please get in touch with the community lead[15] of the geostatistics community if you want to **become a contributor**.

# 11 Acknowledgements

The project was generously supported by the **52 °North** Student Innovation Prize for Geoinformatics 2010.

# 12 References

Botts, M., 2007, OGC Implementation Specification 07-000: OpenGIS Sensor Model Language (SensorML)- Open Geospatial Consortium, Tech. Rep.

Chambers, J.M., 2008, Software for Data Analysis, Programming with R. Springer, New York.

Cox, S., 2007, OGC Implementation Specification 07-022r1: Observations and Measurements - Part 1 - Observation schema. Open Geospatial Consortium. Tech. Rep.

Cox, S., 2007, OGC Implementation Specification 07-022r3: Observations and Measurements - Part 2 - Sampling Features. Open Geospatial Consortium. Tech. Rep.

[14] http://52north.org/resources/source-repositories/

[15] http://52north.org/communities/geostatistics/community-contact

Na, A., Priest, M., Niedzwiadek, H. and Davidson, J., 2007, OGC Implementation Specification 06-009r6: Sensor Observation Service, http://portal.opengeospatial.org/files/?artifact_id=26667, Open Geospatial Consortium, Tech. Rep.

Portele, C., 2003, OGC Implementation Specification 07-036: OpenGIS Geography Markup Language (GML) Encoding Standard, version: 3.00. Open Geospatial Consortium, Tech. Rep.

Vretanos, P.A., 2005, OGC Implementation Specification 04-095: OpenGIS Filter Encoding Implementation Specification. Open Geospatial Consortium, Tech. Rep.

Whiteside, A., Greenwood, J., 2008, OGC Implementation Specification 06-121r9: OGC Web Services Common Specification. Open Geospatial Consortium, Tech. Rep.