Grant Agreement No.: 604656

Project acronym: NanoSim

Project title: A Multiscale Simulation-Based Design Platform for Cost-Effective CO_2 Capture Processes using Nano-Structured Materials (NanoSim)

Funding scheme: Collaborative Project

Thematic Priority: NMP

THEME: [NMP.2013.1.4-1] Development of an integrated multi-scale modelling environment for nanomaterials and systems by design

Starting date of project: 1st of January , 2014

Duration: 48 months

WP N°	Del. N°	Title	Version	Contributors	Lead beneficiary	Nature	Dissemin. level	Delivery date from Annex I	Actual delivery date dd/mm/yyyy
1	D1.5	Porto API documentation	1	Thomas F. Hagelien	SINTEF	Other	PU	31/01/2014	31/09/2014

1 Description

The current implementation of Porto supports different ways of extending the framework. The PortoShell is a key feature that is built on a mature and feature rich scripting language suitable for writing general purpose scripts, and serves as the main interface for Porto. Many of the features of Porto are therefore implemented as extensions for the scripting language. When the framework is extended with support for new file formats, model implementations etc., it is imperative that this functionality is made available to the scripting platform.

The indented readers of this document are software developers that want to extend the framework, or simply wants use the existing functionality.

Porto is a data-centric framework with great focus on entities and their relationship. Entities are either stored in the *internal storage* (database), represented in source code as structs, classes or modules, or weakly represented through references to external data sources (*external storage*). The physical representation of an external storage is dictated by the different simulators. Adaptors for these formats are necessary for being able to retrieve (and write) data for use in a multiscale workflow in a generic way. Ideally it should be possible to define a workflow between scales without making any presumptions about the involved simulation tools and file formats. The processes involved should simply share a single reference (ID) that refers to a collection of data that the individual adaptors and driver will be able to work with.

One possible way of working with Porto is to map out the entire state of the software application to be developed and represent this as a collection of entities. Based on the meta-data schemas that describes the format of the entities, it is possible to generate source code that represents the entities in a native way of the target programming language. For instance, in C, an entity would perhaps best be represented as struct, in Java, C# and C++ as a class, in Fortran as a Type within a module, etc. The generated code could include all communication with the framework and memory management such as caching, allocation and deallocation.

This document describes the current state of the different application programming interfaces (API) in Porto - for making use of the existing functionalities, as well as creating custom extensions.

2 Porto Plugin API

The Porto Plugin API defines the interface for creating custom extensions in the language of C++. This API will rely heavily on the Qt Plugins API. Recommended reading is "How to Create Qt Plugins".

2.1 ISoftPlugin

Brief Description

The ISoftPlugin class is the abstract interface that is needed for creating plugins for extending SoftShell.

Public Functions

```
virtual void registerPlugin(QScriptEngine *engine) = 0
```

Detailed Description

Extending SoftShell with plugins requires the use of the QtPlugin framework.

The ISoftPlugin is the abstract interface class that all plugins needs to derive. The interfaces is built on the QtPlugin framework and is identified by the string literal `"org.sintef.no/ISoftPlugin/0.1"` as it is compatible with the SOFT5-framwork interface.

Since the only purpose of the plugin is to extend the scripting engine (QScriptEngine) the only callback function is *`registerPlugin`* which lets the plugin modify properties of the current *engine* object.

To include the interface class in your code, include the following statement in the .pro –file:

```
include($$(PORTOBASE)/../portotools/src/plugins/common/common.pri)
```

API

`registerPlugin` `(QScriptEngine *engine) -> void`
A callback function called by framework that passes a pointer to a QScriptEngine object.

Creating a new plugin-class

This example shows how a class derived from QObject can also be declared as a plugin:

```
#include <QObject>
#include <QScriptEngine>
#include "isoftplugin.h"

class MongoPlugin : public QObject, public ISoftPlugin
{
  Q_OBJECT
  Q_PLUGIN_METADATA(IID "org.sintef.soft/ISoftPlugin/0.1")
  Q_INTERFACES(ISoftPlugin)
public:
  virtual ~MongoPlugin();
  virtual void registerPlugin(QScriptEngine *engine) override;
};
```

3 External storage API

The External Storage API defines the generic interface to all *external* file formats that will be used in the Porto framework. By "*external*" we mean the 3rd-party and in-house formats used by different simulator software. The only requirement of the external file format is that data is transformable into a keywork-value type of data model. The keyword should be specified by a (unique?) text string representing property or group, and value being a set of data (primitives, lists or arrays).

Note: The current interface is very immature and will need to be developed further to be more suitable for real world implementations.

The next version of this API should include classes that supports the data model by defining at leas the following classes:

- IExternalStorageArray
- IExternalStorageObject
- IExternalStorageValue
- IExternalStorageError

3.1 IExternalStorageDriver

The IExternalStorageDriver class is an interface (true virtual abstract class) that defines the API for all derivative drivers to be implemented.

Functions

	`IExternalStorageDriver(IExternalStorageDriver *driver)`
`bool`	`open (char const *url) [virtual]`
`string`	`name() const [virtual]`
`string`	`version() const [virtual]`
`string`	`readString (char const *propertyName) [virtual]`
`bool`	`readDouble (char const *propertyName, double *target) [virtual]`
`bool`	`readInt (char const* propertyName, int *target) [virtual]`

`bool`	`readDoubleArray (char const * propertyName, int rank,`
	`int const *dimensions, double *target) [virtual]`

`bool`	`readIntArray (char const * propertyName, int rank,`
	`int const *dimensions, int *target) [virtual]`

`int`	`getRank (char const *propertyName) [virtual]`
`bool`	`getDimensions (const char *propertyName, int *target) [virtual]`
`int`	`getNumProperties() const [virtual]`
`bool`	`getPropertyName (int index, char *target) const [virtual]`

API

```
This is an interface class and must be derived/specialized for each
external driver to be embedded into the Porto framework. All virtual
methods must be overridden and implemented in the derivative classes.
```

`IExternalStorageDriver` `()`
`Constructor.`

bool open (char const *url) [virtual]
Connect to data source indicated by *url*.

---

string name () [virtual]
Return the current driver name.

---

string version () [virtual]
Return the current driver version.

---

string readString(char const *propertyName) [virtual]
Get string value for item with the key *propertyName*.

---

bool readDouble (char const *propertyName, double *target) [virtual]
Get double value for item with the key *propertyName*. The value is stored in *target*. Returns *true* if successful, otherwise returns false.

---

bool readDoubleArray (char const * propertyName, int rank, int const *dimensions, double *target) [virtual]
Get array of doubles values for the item with the key *propertyName*. The *rank* defines number of dimensions the array or matrix has. Dimensions receives the actual size of each rank. The values are stored in *target*. Returns *true* if successful, otherwise returns false.

---

bool readIntArray (char const * propertyName, int rank, int const *dimensions, int *target) [virtual]
Get array of integer values for the item with the key *propertyName*. The *rank* defines number of dimensions the array or matrix has. Dimensions contains the actual size of each vector. The values are stored in *target*. Returns *true* if successful, otherwise returns *false*.

---

int getRank(char const *propertyName) [virtual]
Get the rank of the array or matrix value for the item with the key *propertyName*.

---

bool getDimensions(char const *propertyName, int *target) [virtual]
Get the dimension sizes for each vector of the array or matrix value for the item with the key *propertyName*.

---

int getNumProperties() [virtual]
Get the number of properties defined in the current dataset.

---

bool getPropertyName(int index, char *target) [virtual]
Get the name of the property with index *index*. The resulting string is copied to the buffer *target*. Returns *true* if successful, otherwise returns *false*.

3.2 ExternalStorage

Functions

void registerExternalStorageDriver (char const *name, IExternalStorageDriver *driver) [static]
ExternalStorage addExternalStorageDriver (const char *name) [static]
IExternalStorageDriver *driver()

API

This is the content class of the external storage driver. This class follows the software design pattern known as *strategy pattern.*

```
void registerExternalStorageDriver(char const *name,
     IExternalStorageDriver *driver) [static]
```
Register an external storage driver in the framework.

ExternalStorage addExternalStorageDriver(char const *name) [static]
Create a new ExternalStorage embedded with the driver registered as *name*.

IExternalStorageDriver *driver()
Returns the pointer to the currently embedded driver.

---

4 PortoShell API

The following Porto standard build-in modules are extensions to the standard ECMAScript Language and provide generic basic functionality.

Module	Description
Process Module	Execute and communicate with external programs
EventLoop Module	Running and leaving even loops
UDP Socket Module	Provides UDP socket communication
Console Module	Console output utility
File System Module	File I/O module
File System Watcher Module	Monitoring of file and directories
Httpd module	Web server functionality

4.1 Process Module

The Process Module allows for the execution of external applications from scripts.

Functions

Process()
execute (program, [arguments])
arguments()
setArguments([arguments]**)**
program()
setProgram(program**)**
start()
readAllStandardError()
readAllStandardOutput()
waitForFinished(msecs = 30000**)**
write (buffer)

Signals

finished()
readyReadStandardError()
readyReadStandardOutput()
started()

API

Process ()
Constructor

---

execute (program, [arguments]**)**
Starts the program *program* with the arguments *arguments*. This function will wait until the process has finished.

---

arguments() -> [arguments]
Returns the arguments used to start the process with.

---

setArguments ([arguments]**)**
Set the *arguments* to pass to the program to be executed. This function must be called before **start().**

---

program () -> program
Returns the program the process was last started with.

---

setProgram (program**)**
Set the *program* the be started. This function must be called before **start().**

---

start ()
Asynchronously starts the given *program* in a new process. The Process will emit the **started()** if the program successfully starts, otherwise it will emit **error()**.

---

readAllStandardOutput() -> buffer
Returns all available data written from the running program into the standard output channel.

---

readAllStandardError() -> buffer
Returns all available data written from the running program into the standard error channel.

---

waitForFinsihed(msec=30000) -> true | false
Blocks until the process has finished, or until *msec* milliseconds have passed. Returns *true* if the process finished, otherwise returns *false*.

---

write(buffer) -> numBytes
Writes the contents of *buffer* to the process' standard input channel. Returns the number of bytes written, or -1 if an error occured.

Detailed Description

Process can be used to start and stop programs and has the ability to read from the out channels *stdout* and *stderr*. In addition the script can write to the process's standard input.

The porto Process module is based on the QProcess class defined in QtCore/QProcess.

Example

In the following example the Process module, together with the EventLoop module, is used to run shell process (*ls*) and display its outputs.

```
/* File: process-test.js */

/* Create an event loop */
var eventLoop = new EventLoop();

/* Create and set up the external process */
var dir = new Process();
dir.setProgram("ls");
dir.setArguments(["-al"]);
```

```
/* Set up the signal/slots mechanism with callbacks */
dir["readyReadStandardOutput()"].connect(function(){
  print (dir.readAllStandardOutput());
});

dir["readyReadStandardError()"].connect(function(){
  print (dir.readAllStandardError());
});
/* Start the execution */
dir.start();

/* Run the eventloop (forever) */
eventLoop.exec();
```

Results

```
PortoShell 0.1.33
Source license: LGPLv3

For help, type :help

> :ld process-test.js
total 8
drwxrwxr-x 2 thomas thomas  96 Sep 30 23:15 .
drwxr-xr-x 7 thomas thomas 240 Sep 30 23:13 ..
-rw-rw-r-- 1 thomas thomas 333 Sep 30 23:15 test.js
```

4.2 EventLoop Module

Functions

EventLoop ()
exec ()
isRunning ()
wakeUp ()
exit (returnCode = 0)
quit ()

Detailed Description

The EventLoop provides a means of entering and leaving an event loop. The module is based on the QtCore/QEventLoop

API

EventLoop()
Constructor.

exec() -> retVal
Enters the event loop and waits until **exit()** is called. Returns the value passed to **exit()**.

---

isRunning() -> true | false
Returns *true* if the event loop is running, otherwise returns *false*.

---

wakeUp()
Wakes up the event loop.

---

exit(exitCode=0)
Tells the event loop to exit with return code *exitCode*.

---

quit()
Same as **exit**(0).

4.3 UDP Socket Module

Functions

UdpSocket ()
bind (hostAddress, port)
hasPendingDatagrams ()
readDatagram ()
writeDatagram (datagram, hostAddress, port)

Signals

readyRead ()

Detailed Description

The UdpSocket module provides a User Datagram Protocol (UDP) socket. UDP is a lightweight message-oriented network protocol for connectionless transfer of information (datagrams).

API

UdpSocket()
Constructor.

---

bind(hostAddress, port)
Enables the UdpSocket to receive datagram. The signal **readyRead()** is emitted whenever a datagram arrives to the specified *hostAddress* on the given *port*.

---

hasPendingDatagrams() -> true | false
Returns *true* if the udp socket has at least one datagram waiting to be read.

---

readDatagram() -> buffer
Reads a datagram from the UdpSocket and returns the text string.

---

writeDatagram(datagram, hostAddress, port**)**
Send a *datagram* to the given host *hostAddress* at port *port*.

4.4 Console Module

Functions

console.log (buffer)
console.info (buffer)
console.error (buffer)
console.warn (buffer)
console.trace (buffer)

Detailed Description

This module defines node.js compatible console object functions for sending output to stdout and stdin.

4.5 File System Module

In order to work with files in JavaScript we have to extend the language. The File System Module defines an object '*fs*' which defines functions to support this.

Functions

fs.exists (fileName)
fs.copy (source, destination)
fs.remove (fileName)
fs.rename (oldPath, newPath)
fs.currentPath ()
fs.setCurrentPath (pathName)
fs.readFile (fileName, callback)
fs.writeFile (fileName, buffer, callback)

API

fs.exists (fileName) -> true | false
Checks whether *fileName* exists or not.

```
if (fs.exists ('/tmp/myfile')) {
  print ('the file exists');
}
```

`fs.copy` `(fileName, newName) -> true | false`

Copies the *fileName* to *newName*. Returns *true* if successful, otherwise returns *false.*

`fs.remove``(fileName) -> true | false`

Removes the file specified by *fileName.* Returns *true* if successful, otherwise returns *false.*

`fs.rename``(oldPath, newPath) -> true | false`
Renames the file or path specified by *oldPath* into *newPath.* Returns *true* if successful, otherwise returns *false*.

`fs.currentPath``() -> pathName`
Returns the path *of the* current directoy.

`fs.setCurrentPath``(pathName) -> true | false`
Sets the current working directory to *pathName.* Returns *true* if successful, otherwise returns *false.*

`fs.readFile` `(fileName, callback)`
Reads a file. A *callback* function needs to be provided to capture the file contents and possible errors

```
fs.readFile(filename, function (err,data){
  if (err)
    throw ('Couldn't read file due to' + err);
  console.log(data); // Print file contents to standard output
});
```

`fs.writeFile` `(fileName, buffer, callback)`
`Stores` *`buffer`* `into a file named` *`fileName`*`.`

```
fs.writeFile(filename, data, function (err){
  if (err) {
    throw ('Couldn't write file due to' + err);
  } else {
    console.log('The file was successfully saved');
  }
});
```

4.6 File System Watcher Module

The File System Watcher module provides functions for monitoring files and directories for modifications. It uses the signal/slot mechanism to notify about changes. This module should therefore be used in conjunction with the EventLoop module.

Functions

FileSystemWatcher ()
addPath (path)
addPaths ([paths])
directories ()
files ()
removePath (path)
removePaths ([paths])

Signals

directoryChanged (directoryName)
fileChanged (fileName)

API

FileSystemWatcher()
Constructor.

---

addPath(path) -> true | false
Adds *path* to the watcher. If *path* specifies a directory, the signal **directoryChanged(**dirName) will be emitted when *path* is modified, renamed or removed. If the *path* specifies a file, the signal **fileChanged (**fileName) will be emitted when *path* is modified, renamed or removed. Returns *true* is successful, otherwise returns *false.*

---

addPaths([paths]) -> true | false
Adds a list of *paths* to the watcher. See **addPath()**

---

directories() -> [dirs]
Return a list of paths to directories that are being watched.

`files``() -> [files]`
Returns a list of paths to files that are being watched.

`removePath``(path) -> true | false`
Removes the specified *path* from the watcher. Returns *true* if successful, otherwise returns *false*.

`removePaths``([paths]) -> [paths]`
Removes the specified list of *paths* to the watcher. Returns a list of paths which unsuccessfully registered.

Example

```
var event = new EventLoop();
var fsw = new FileSystemWatcher();
fsw.addPath("/tmp/test.txt");
fsw["fileChanged(QString)"].connect(function(file){
    console.log("file",file,"did change");
});

event.exec();
```

4.7 Httpd module

Functions

HttpServer ()
start ()
stop ()
setRootDir ()

Detailed Description

The HttpServer is not a major feature of Porto and is currently not mature enough to be very useful. The intention of the module is, in the future, to be able to provide a GUI to the framework through a web browser.

5 PortoShell Modules API

The PortoShell modules are JavaScript programs that resides in `$(PORTOBASE)/bin/modules`. These are different utilities and methods for performing a variety of operations. The directory/folder structure of everything under the `modules` directory serves as the *namespace* for the module, and the different modules can be imported through the function *`require()`*.

5.1 porto

5.1.1 porto.collection

Functions

setName (name)
name ()
setVersion (version)
version ()
count ()
instances ()
findInstance (label)
registerRelation (fromLabel, toLabel, relation)
registerEntity (entity, label)
accept (visitor)

Detailed Description

The collection module provides an interface for working with the contents of the *entity* called *collection.* The collection is a container of other entities and collection and the relationship beween these.

5.1.2 porto.macro

Functions

expandFile (fileName, {ext})
expand (buffer, {ext})
setVersion (version)

Detailed Description

The macro module provides the functionality to expand a javascript expression within a text string. The expression needs to be defined with the following markup:

Regular Expression	**Type**	**Example**
@{*expr*}	Block	@{obj.someValue}, @{1+2+3+4+5+6}
@value	Value	@foobar

The curly brackets are useful when the template requires contents to continue directly after the evaluated expression, or when we need to evaluate a more complex expression.

Example

```
var $m = require ('porto.macro');
$m.expand('This is a test');
$m.expand ('Hello, @ext.value!', {value: 'World'});
```

```
This is a test
```

```
Hello, World!
```

5.1.3 porto.mvc

Functions

create ({obj})

Detailed Description

The Model-View-Controller (MVC) module is a utility built on the `porto.macro` module to provide implementers of code generators with a high-level design that supports the separation of information.

Example

```
/* A generic code generator framework */

__main__ = function (args){
    var modelFile = args[1];
    var viewFile = args[2];
    fs.readFile(modelFile, function(err, data){
      if (err) throw (err);
      try {
          generate = require('soft.mvc').create({
            model: JSON.parse(data),
            view: viewFile
          });
      } catch (e) {
          console.error(e);
          return;
      }
      try {
          console.log(generate({filename: modelFile}));
      } catch (e) {
          console.error(e);
      }
    });
};
```

5.1.4 porto.autorun

Functions

run ({obj})

Detailed Description

The `porto.autorun` is a workflow runner under development. The documentation will be updated.

5.1.5 porto.entity

Functions

```
db ({driverInfo})
```

Detailed Description

The entity module allows for creating entity constructors based on storage information and entity name/version.

Example

```
/* Create factory */
var entityDb = require (‘soft.entity’).db({
  driver: “mongodb”,
  database: “MyEntityDb”,
  collection: “MyCollection”
});

/* Create constuctor (MyEntity class) */
MyEntity = entityDb.using(‘demoentity’, ‘1.0-SNAPSHOT-1’);

/* Create a new instance of the entity */
var myEntity = new MyEntity();
```

5.1.6 porto.storage

Functions

```
connect ({driverInfo})
```

Detailed Description

The storage module provides a generic function for connecting to a storage device.

Example

```
/* Define a data storage connection */
var storage = require(‘soft.storage’).connect({
    driver:      “mongodb”,
    uri:         “mongodb://localhost”,
    database”:   “MyStorage”,
    collection”: “MyCollection”
});


/* Use the entity factory to create a new entity type */
MyEntity = require(‘soft.factory.entity’).createEntity(
  ‘demoentity’,
  ‘1.0-SNAPSHOT-2’,
  function (err){
      if (err) throw (err);
  });

/* Define how the entity should store its contents */
MyEntity.prototype.store = function() {
    storage.write(this);
}

MyEntity.prototype.read = function() {
    var self = this;
    storage.read(this.id, function (bson) {
      self.set(JSON.parse(bson.asString()));
    }
}
```

5.1.7 porto.storage.mongo

Functions

`find (query)`
`read (uuid, callback)`
`write (entity, callback)`
`driver ()`

Detailed Description

This is a specialization of *porto.storage* where driver for the MongoDB is defined. This class should not be used directly. Use the *porto.storage* module instead.

5.1.8 porto.factory.entity

Functions

`createEntity (entityName, entityVersion, callback)`

Detailed Description

A utility module for creating a constructor for an Entity.

5.1.9 porto.utils.metastore

Functions

`connect ({connectInfo})`

Detailed Description

The *porto.utils.metastore* is a utility that hardcodes the use of MongoDB for storing meta-data. The utility program found in $PORTOBASE/bin/register-entity.sh makes use of this module to create a console application for registering meta-entities.

Example

```
/*
 * register-entity.sh
 * A utility to commit meta-data into the metadata-database
 */

__main__ = function (args)
{
    if (args.length <= 1) {
         console.error("fatal error: no input files");
         return undefined;
       }

    /* Connect to the meta-data database */
    var metaStorage = require ('soft.utils.metastore').connect(
      {
          uri: 'mongodb://localhost',
          database: 'meta',
          collection: 'entities'
      });
    args.shift();
    args.forEach(function(file){
      fs.readFile(file, function(err, data) {
          if (err) {
            print ("error:", err);
            return;
          }
          if (!metaStorage.store (data)) {
            print ("Failed to write data");
          }
      });
    });
}
```

5.2 utils

The utils modules are a selction of 3rd party utility modules that are included in the framework for enriching the scripting experience. They are included in the source code and adopted to Porto in order for them to be available for all in cases where they are being used as part of production scripts.

5.2.1 utils.base

A base class for JavaScript inheritance.

Online documentation: http://dean.edwards.name/weblog/2006/03/base/

5.2.2 utils.jjv

A JavaScript JSON Validator

Online documentation: https://github.com/acornejo/jjv

5.2.3 utils.jlinq

jLinq allows for complex queries on arrays of JSON data.

Online documentation: http://hugoware.net/projects/jlinq

5.2.4 utils.quantities

quantities is JavaScript library for handling scientific calculations involving quantities (units).

Online documentation: https://github.com/gentooboontoo/js-quantities

5.2.5 utils.time

A tiny module used for timing function-calls and JavaScript operations

5.2.6 utils.underscore

Underscore is a large library of functional programming helpers in JavaScript.

Online documentation: http://underscorejs.org/