Programmer documentation

xxx,xxx

February 12, 2013

Contents

1 ViFrame framework

The ViFrame framework is composed of two main parts — a library called *BaseLib* and a standalone desktop application both developed in C++. The application uses QT library to generate the user interface. Since ViFrame needs to work with modules in the form of dynamically load libraries it is platform-dependent. One of the most emphasized requirements when developing ViFrame was to make the implementation of new modules that participate in the visualisation as easy as possible. Since the modules make sense only within the visualization pipeline, another important aspect was to allow easy integration of the modules into the pipeline. Despite the requirement on simple implementation of new module the framework is complex, but it still can be used in simple way.

1.1 Pipeline

The visualisation pipeline is composed of modules. Modules are the base computational units and they inherit from the BaseLib::iModule base class. Data between modules are exchange as data objects. Data object is a named container, that holds values stored under string ids. Base classBaseLib::iModule does not provide methods for data handling, these methods are presented in separated abstract class BaseLib::Interfaces::SequentialAcces and BaseLib::Interfaces::RandomAcces. To enable module the exchange of data objects, there is a need for passing information what inner data an object holds. This is accomplished by using Sockets. The socket is a class that describes data objects which can then be passed from one module to another to share information about the contained data objects. Each module can be in three different states: closed (default state), opened and prepared. The module's state can be changed by calling fallowing methods *Open*, *Prepare*, *Close*. After calling *Open* method, a module should open all sources. While after call of *Prepare* method, a module can be asked about data objects through implemented interface. Finally, the *Close* method will result into closing module, the state of module atfer calling of *Close* method should be the same like the state after creation or before the *Open* method was called.

Some modules may require setting up some variables that drive the module behavior (e.g., path to source file, number of iterations, ...). A module can declare that given variable is the module property. The module property consist from associated variable, name and description. Module properties are shown and can be set up in Properties dialog in the application. To enable connection checking when assembling the pipeline, a concept of contract and socket is used. Each module is responsible for proper specification of its input and output sockets. The socket holds description of the data object that the module requires on input or offers on output. In case of connection every input socket must be subset of connected output socket. Because it is possible that multiple modules share output socket specification (name-value array of objects passed), a concept of levels is used to solve this situation. Data value is described not only by its name but also by the number of the module that created it. The numbers

are assigned to modules in such a way that they create a topological numbering over the pipeline. This enables existence of multiple data with the same name. Many modules just add new values or modify given values, so their output is determined by their input. To simplify manipulation inside these modules a programmatic support is provided by BaseLib so that passing an object in term of specification can be solved by calling several pre-implemented functions.

2 BaseLib

2.1 Overview

As can be seen from diagram 1 the BaseLib library can be split into several parts. We list all parts and give at least brief comment to each.

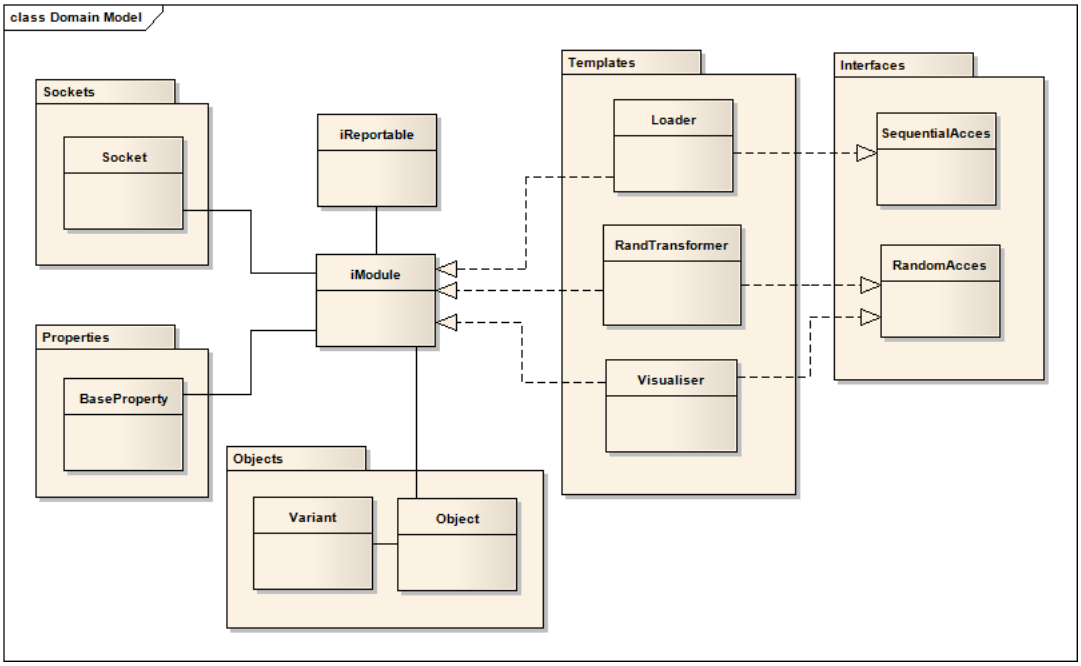

Figure 1: BaseLib diagram

2.1.1 Sockets

This part contains only Socket class. Purpose of Socket is describe input as well as output socket. For both cases the same class is used, difference is only in way in which Socket is used. Socket class describes not only the data but also can describe their meaning. This can by archived by using Name and Description variables. Socket class is in fact the description of the data object. Levelling

(see 2.1.3 and 3.9 for more detail) is also supported with proper methods. The interface that can be used to obtain described data objects is also specified by Socket class.

2.1.2 Properties

Contains classes which are used as the module properties. The diagram 2 shown a dependency in Properties namespace/package. Properties can be set before

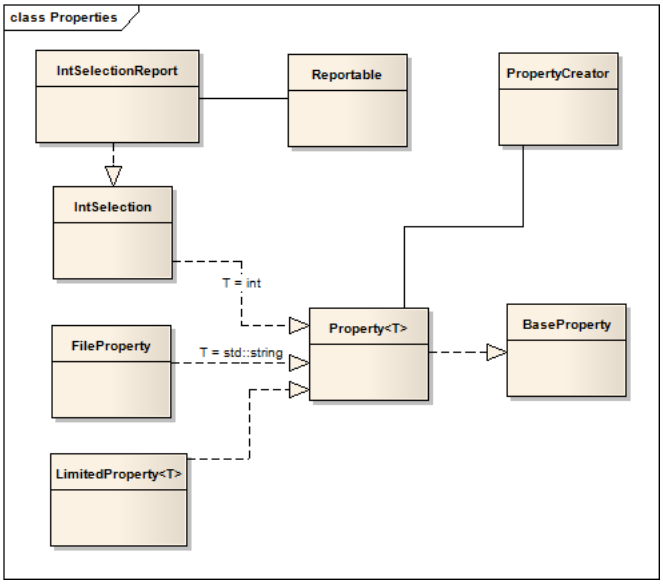

Figure 2: diagram of Properties namespace

execution in Properties dialog, which is generated by application. In fact properties are the only way in which module can obtain information from the user. So the set up of module is secured through module's properties. Application support six types of properties which are the fallowing:

- Bool - CheckBox
- String, Int, Double - TextBox
- IntSelection - ComboBox
- StringFile - TextBox with button for file selection

After dash the graphical element that is used to set given property type is given. The process of properties setup and more details about properties are described in section 2.5.

2.1.3 Objects

Contains classes for data object which is basically data container. The data object is a basic data unit handled between modules. Data object does not provide any functionality that would enable interaction between Socket and data object. So data object must be constructed by module directly. Data object contains objects that inhered from class Variant. Variant is value base class. It doesn't in way restrict value that can be hold in the data object. All that class that want to be value carriers must inherit Variant as a base class and provide implementation of two methods. These methods are Get Type and To String, second method is used to get value into representation in which it can be presented to the user. The purpose of first method is identify data value in framework, so the method should return as unique value as possible.

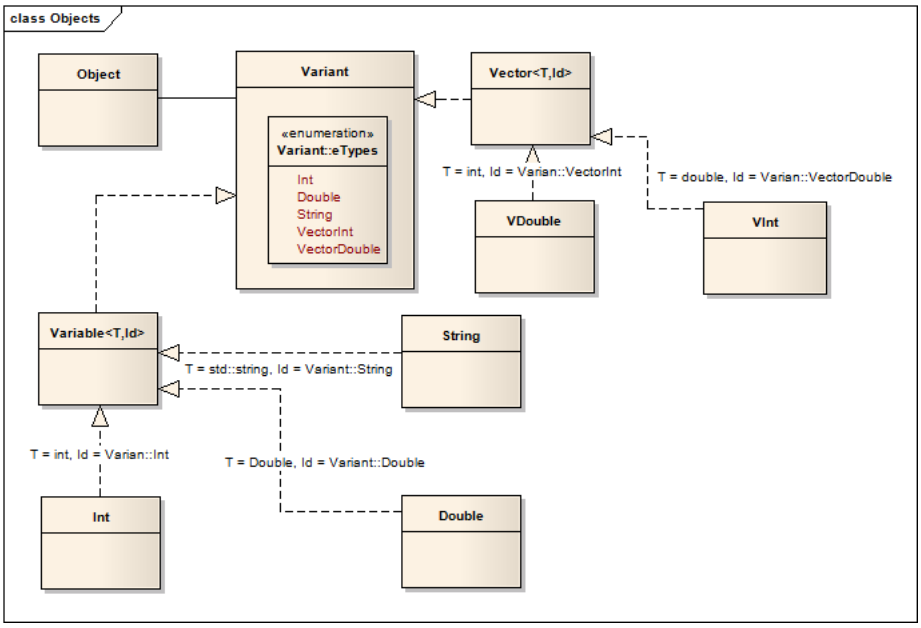

Figure 3: diagram of Object namespace

Diagram 3 show that BaseLib contains several pre-implemented value types. The names of the classes are almost all self explanation, just note that prefix V mean vector.

2.1.4 Modules - templates

The class iModule contains number of methods and to work properly must holds number of conditions. So to work properly the module have to contain some utility code. However when writing new module much of these utility code can be reused. To make implementation of new module easier there are three

classes-templates. They could be use as a base for a new module. Each template corresponds to certain module purpose.

- *Loader* is a base class that contains all necessary code for a module which is meant to be data source.
- *RandTransfomer* is literally a c++ template. It can be used as a base for modules which want to work as data transformers and need access to all data at once.
- *Visualiser* is c++ template as well as RandTransfomer, in fact they are very similar. But visualiser provide less function then RandTransfomer and so is easier to be used.

For more detail or implementation information please see documentation for certain class-module or see 4.

2.1.5 Interfaces

Is some application the access to all the data at once can be required on the other hand in some application the data can be so large, that they cannot be loaded into memory at once. To enable usage of ViFrame in both cases there are two different interfaces that are used to handle data between modules.

- *SequentialAcces* enable module to work always with only one object.
- *RandomAcces* provide access to all data by index. All the data are loaded into memory.

There is also concept of data owning. The concept is used to determine responsibility for data object release. For more information please see 3.8.

2.2 Data object

Data in the application are stored in data objects which are the base data unit transmitted between modules. A data object is a class that contains the object's name and a collection of named values. Each data object can contain at most one value of a given name. The values that are stored in the data object are derived from a base value type called Variant. The Variant is a base class that specifies the interface that is mandatory for every data value. Adding a new data type is accomplished by deriving from the Variant type.

We can easily imagine a situation where two connected modules has the same name for their output socket. Which mean that both modules try to save data under same name, we can call this situation name collision. If there wouldn't be any solution and the issue is silently ignored then the second module would have to deal with this situation on it's own. By ignoring the situation and simply rewriting pointer to the data in data object will the module cause memory leak. To prevent such situation a concept of levelling is used. For more information about this concept please see 3.9.

2.3 Exception handling

Framework has it't own base class for exception BaseLib::Exception which inherit from *std::exception*. Using this base class for exception is preferred. Additional support is provided by class *ExceptionHolder*, which is design to carry the exception between calling of modules. By stage when the exception is thrown exception is caught and handled. If it's thrown in *Open*, *Prepare* or *Close* (or any inner version of those methods) it's caught and transferred as parameter and thrown again. The code which done this can look like this:

```
// exception holder class
ExceptionHolder ex;
// call Open on SourceModule
SourceModule->Open(ex);
// any thrown exception is stored in ex variable
// if any exception was thrown this will re-throw it,
// otherwise nothing happen
ex.Rethrow();
```

Instead of direct call of *Rethrow* the caller can call *ContainsException*, and if the exception was thrown do some work before calling *Rethrow*. *ExceptionHolder* is design to carry not only exception from ViFrame framework but also exceptions that inherit from *std::exception*.

2.4 Communication with the user and the application

To enable communication from module to the user and to the application, application register at the module class that inhered from *iReportable*. Module then can communicate by calling a method on the registered class. Methods correspond with type of communication for they can be used to. There are four type of communication:

- Text message to the user.
- Progress report, used to report progress in computation.
- ConnectionChanged
- InterfaceChanged

The first one can be used in any time. This message has no influence on application execution at all and is meant only to communicate something to the user. The second one used used in computation phase when the module is with data objects. It also doesn't have any real influence on application execution. The last two type on other hand are used for system purpose. They should be used only in under certain conditions. The purpose of them is to report the possibility of change in the module's connection of interface.

2.5 Properties

Module's properties are set through Properties class. Module simply give list of properties and all setting is done with properties classes so there is no call of any Module methods. In this way base module class *iModule* provide all what is needed which is access method to properties and properties storage. Properties store their value into user defined variable. The fallowing example demonstrate usage of property class.

```
// Module class
class MyModule : BaseLib:iModule
{
// variable where property value will be stored
int mTargetDimension;
public:
// ctor
MyModule()
{
  // we use static method CreateLimited
  // this method create property with given name,
  // description and restrict property range
  Properties::BaseProperty* property =
    BaseLib::Properties::CreateLimited(
      "Target dimension: ",
      "Output main data dimension.",
      mTargetDimension,
      1,
      1048576);
  // now we add property into properties container
  mProperties.push_back(property);
  // now user can set property called "Target dimension: "
  // which takes as a valid values integers
  // between 1 and 1048576. When user confirm
  // property change the value will be saved directly
  // into variable mTargetDimension.
}
```

Module does not have any control of property setting. Because of that property validation must be done in property class and if module want to react on property change the callback method or similar mechanism must be used. The second issue is pre-solved by *Reportable* class and existence of *ReportProperty*. We demonstrate this on another example:

```
class MyModule : BaseLib:iModule, BaseLib::Properties::
    Reportable
{
int mTargetDimension;
public:
  virtual void OnChange(BaseProperty* prop)
  {
    // here we can react on property changes
  }
MyModule()
{
  mProperties.push_back(
    BaseLib::Properties::CreateReport(
```

```
        "T.dim:",
        "Description",
        mTargetDimension,
        this);
    );
}
}
```

When property *T.dim* changes the method *OnChange* with pointer to changed property as parameter is called. Now we take closer look on the property validation. It must be done in property setting function

```
bool SetProperty(const TYPE& data, std::string& log)
```

If any error occurs method should return false and store error message in log parameter. There are two more base facts about properties that should be mentioned. The first fact is existence of *SetPropertyLoad* function. This function is used when loading pipeline (see. module load 3.6 for more details about module loading). The second fact is that FiVrame application is able to work only with predefined properties. So it's possible to inherent existing class and change it's behaviour like validation. For this purpose for example the *Property¡Type¿* base class can be used. But it's no possible to create new property base directly on base class *BaseProperty*.

2.6 Module and the application

Application load modules from sub-folders of *Modules* folder which is located in same folder as application executable file. By using sub-folders a tree structure in the list of modules can be created. Modules are stored as dynamically loaded libraries dll-files. So in order to use your module in application it must be compiled into dll file, we also recommend use same compiler as was used to compiled main application. There is one additional condition. It necessary to implement three methods which are used by application when loading module. These methods are:

```
_DYNLINK void* CreateModule()
_DYNLINK void DeleteModule(void* module)
_DYNLINK const BaseLib::Description* GetDescription()
```

Implementation example is given in 5. This functions are declared in BaseLib Module.hpp header file.

3 BaseLib::iModule

Class *iModule* is base class for the framework module. *iModule* class contain few pure virtual function which are

```
virtual void* AccessInterface(const int& interfaceId) ;
virtual bool CheckIntegrity();
virtual void innerOpen();
virtual void innerPrepare();
```

```
virtual void innerClose();
virtual void ValidateOutputSocket();
```

This functions must be implemented by user. While some of this function Other functions are pre-implemented. The suitability of pre-implementation depends on purpose of usage *iModule* base class. There is no reason for list function in module here while this list can be found in doxygen documentation. Instead of that we present few examples of usage to give user a brief example how solve some tasks or how is module used in application.

3.1 Uncompleted modules

The module output interface can strongly depend on connected modules or values of properties. In such case the module can be unable to work as a source until some requirements are satisfied. The function *bool PrepareForConnection()*is used to check is ready to be used as a source.

3.2 Module (dis)connect

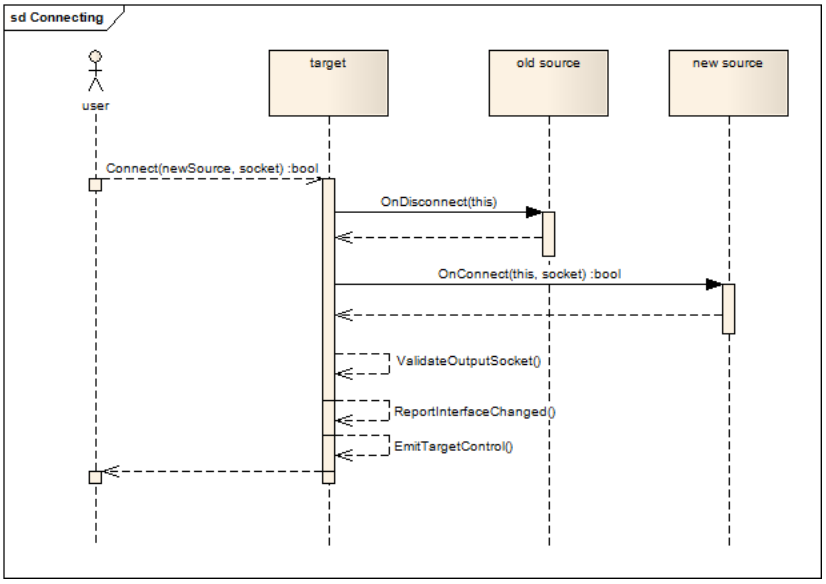

Figure 4: Connecting, call digram

Modules are connected and disconnected from Connection dialog. If the user try to connect two module a function *bool Connect(iModule*, $size_t$)* is called. This function is used for connecting as well as disconnecting. The function is called on module target. The example of calling is given in diagram 4. If old source does not exist calling *OnDisconnect* is skipped. If new source is nullptr calling *OnConnect* is skipped. New source can refuse connection by returning

false from *OnDisconnect* it that case target should not save new source as source module. On the end three functions which ensure propagation of the Socket changed are called.

There is one more function for establishing connection. This function is *bool ConnectPure(iModule∗, size$_t$)*. The function is used for purpose of pipeline loading. The function should still call *OnDisconnect* and *OnConnect* but the target should accept connection without any check. The module should not even update output Socket. Methods for the output Socket update will be called later by the application.

3.3 Changing module properties

Module can change the number and type of properties by simply changing content of *mProperties*. This can be done generally at any time expect when user is working with module properties. The user can work with module properties only through the Properties dialog. There is actually no way how to find out if Properties dialog is open. Still there exist function in which changing properties is relatively save. Such function is for example *bool Connect(iModule∗, size$_t$)*. The reason for safety of this function is that it's called from Connection dialog and so Properties dialog is closed.

3.4 Module output

The module can change it in output socket specification as a reaction to change of the property or as a result of connecting new source. Module output socket can be change at any time except between *Open*, *Close* calls and in *OnConnect* method. When module interface is changed interested modules should be notify. The notification can be send by calling *ReportInterfaceChanged* method. To made working with output socket easier *iModule* contains two protected variables *mOutputSocketAdd*, *mOutputSocketPublic*. The second one is used as a public output socket. The first one can be used in free way. When user deriving directly from *iModule* procedure *ValidateOutputSocket* must be also implemented. This procedure should check that value of *mOutputSocketPublic* correspond to the module properties and inputs. In the default implementation this function is called to refresh output socket specification (value in *mOutputSocketPublic*).

3.5 CutOff

If user decide to delete the module from the pipeline all the connection must be disconnected. Is such case all input modules must be disconnected but also all modules which has the module as input must be disconnected from it. To simplify this the method *CutOff* is used, this method do exactly what is describe in previous sentence. The procedure disconnect all sources and call *OnCutOff* on all targets (modules which have this module as a source). To enable this call, the list of targets must be kept in proper way. If on the module *OnCutOff* is called

the module should disconnect the caller module without calling *OnDisconnect* on the caller.

3.6 Module load

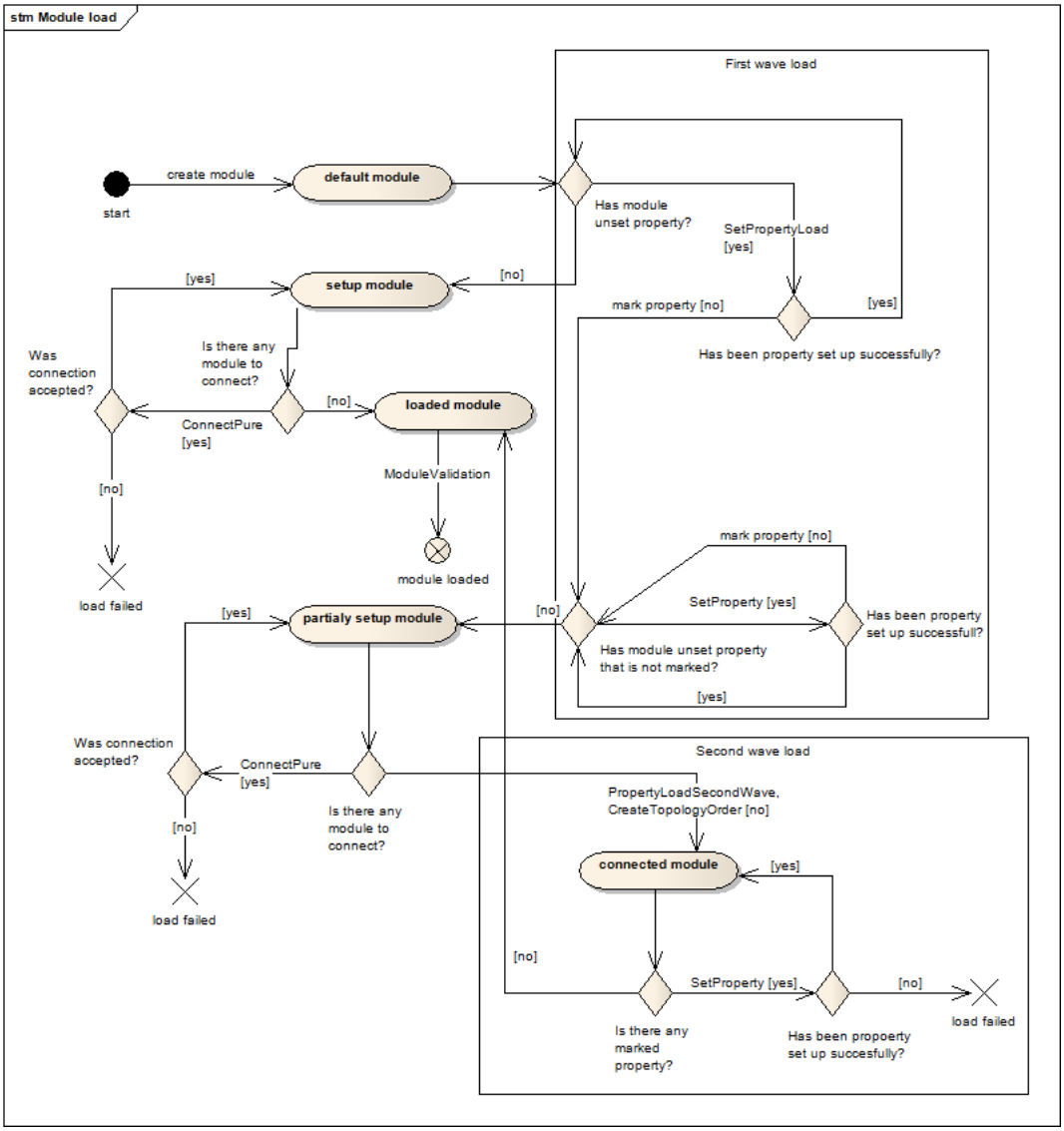

Figure 5: Module load diagram

The module number of input socket, specification of input or output socket and completely whole module interface can be influenced by the module's prop-

erties and modules which are connected to it. When the pipeline is load it's not effective simulate the user when he first assemble it. Instead of this two phase load and unchecked connection establishment are used. In the diagram 5 a process of single module load is presented. The main difference between the First wave load and the Second wave load is that in the second case is guaranteed that all sources for given module are fully loaded and in valid state. This is archived by constructing a topological ordering. The Second wave load is done in topological order.

3.6.1 Module validity

The variable *mValid* is used to indicate if module is in valid state. This variable should be only set to false by *ConnectPure* function. So it's used only in the load phase. If *mValid* is set to false *EmitTargetControl* will do nothing. The function which is responsible for setting *mValid* to true value is *ModuleValidation()*.

3.7 Computation

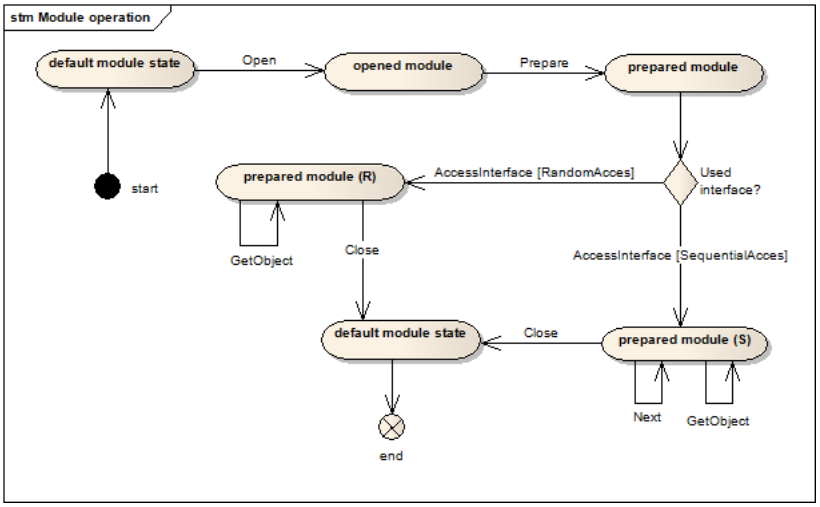

Figure 6: Module computation diagram

The module it self does not contains the method for data handling. To access module data the one of the interfaces should be used. The diagram 6 describe the work/computation phase of single module. The important notice is, that module state before calling *Open* and after calling *Close* should be the same. The annotation of prepared module (R) and prepare module (S) is there only to emphasize the different between usage the Random and the Sequential interface. The used interface determine not only the methods that are used to obtain data but also the way in which data owning is handled (see 3.8 for more detail).

3.7.1 Open, Close, Prepare

When working with source module the methods like *Open*, *Prepare* are called. These function guaranteed that even when they are called multiple times their inner equivalents will be called just once.

3.8 Data owning

Because the data are handled between modules and single data object can be required multiple times there is need to define responsibility for final releasing of the data. This is done by concept of data owning. This differs for each Interface.

- *SequentialAcces* The module give up owning if the object is passed with *GetObject* method. It the *GetObject* method is not call and all the data are just iterate by calling *Next* the source module is responsible for releasing all the objects. But when *GetObject* is called there is no way in which target module can return ownership to the source module.
- *RandomAcces* If the object is passed by *GetObject* the ownership is passed as well. But if *ClearOutState* is called then the source object get back all the ownership rights. So the source module should after the *ClearOutState* method was called behave like no object has ever been passed.

The module can delete all owned data object when is being closed. From this point of view for the target who used the *RandomAcces* access interface on source is important to make sure that he owes all data he got.

3.9 Levelling

There exists no require on uniqueness of module output object names and ofcourse two same modules can be connected in line. There can be situation that two modules wants to use same name to save theire data. To solve this the name is prefixed with module number. This enable using the same name for multiple modules. The *Socket* as well as *Object* class contain support for this functionality. In case of *Object* all what the module must do is to call*Pass* method on *Object* before the object is passed. Similarly the support for *Socket* is solved.

4 Module templates

As can be seen from sample 5 usage of module template is simple. From this reason we only briefly comment functionality of each template and mention what is user suppose to do in order to use it. For more detail please see doxygen documentation.

4.1 Loader

Loader is pre-implemented template, that is supposed to be used as a data source. The template contains one pure virtual method *GenerateObject* meant to be implemented by the user. The method should return a new object on each call.

4.2 RandTransformer

RandTransformer is designed as a pre-implemented template for module that do transformation over data and need access to all the data at once. The template contains a pure virtual method *Transform* where the transformation is done. The risk when using *RandTransformer* as a base class is that source and target data can be the same. So for this reason, first some transformation function should be composed and then apllied on all data at once.

4.3 Visualiser

Visualiser is similar to the *RandTransformer* and function which the user has to implement here is *Visualise*. The greatest difference is that the source and the target data are never the same.

5 Sample implementation

In this section the simple example of pre-implementation of the module is given. To reduce amount of code the data processing is omitted. Module is meant to be something like Visualisation module. As input module require *VDouble* data type. The module give the same type on output. Module has one property of type *int* named 'Output dimension: '.

```
#ifndef sampleModule_h
#define sampleModule_h

// BaseLib includes
///
#include "Vector.hpp"
#include "Visualiser.hpp"

typedef BaseLib::Modules::Visualiser<BaseLib::Objects::VDouble,
    BaseLib::Objects::VDouble> BaseClass;

class SampleModule : public BaseClass {
private:
  /**
   * Output dimension.
   */
  int mOutputDimension;
public:
  SampleModule()
  : BaseClass("OutData"), mOutputDimension(2) {
  // register property
```

```
  mProperties.push_back(BaseLib::Properties::CreateLimited("
      Output dimension: ", "Output dimension.", mOutputDimension
      , 1, 100));
}
public:
  void Visualise() {
  // test input data size
  int dataSize = DataSize();
  if (dataSize == 0)  { // no data -> finish
    return;
  }

  // determine input dimension
  // we assume that all data has the same size
  int inputDimension = AccesInData(0).Data.size();
  if (inputDimension < mOutputDimension) {
    ReportMessage(BaseLib::iReportable::Error, "inputDimension <
        mOutputDimension");
    throw BaseLib::Exception();
  }

  // iterate input data
  for (int i = 0; i < dataSize; ++i) {
  std::vector<double>& inData = AccesInData(i).Data;
  // data processing can be done here ...
  }

  // save output data
  for (int i = 0; i < dataSize; ++i) {
  std::vector<double>& outData = AccesOutData(i).Data;
  // write output data into outData
  }
}
};

_DYNLINK void* CreateModule() {
  return new SampleModule();
}

_DYNLINK void DeleteModule(void* module) {
  SampleModule* convModule = reinterpret_cast<SampleModule*>(
      module);
  delete convModule;
}

// module description
BaseLib::Description locDescription(std::pair<int, int>(1,0),
  "SampleModule",
  "Short description.",
  "Long description."
  );

_DYNLINK const BaseLib::Description* GetDescription() {
  return &locDescription;
}

#endif // sampleModule_h
```