IMOROD Developper Guide

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

BIOLOGICALLY INSPIRED
ROBOTICS GROUP (BIRG)

Sébastien GAY

Department of Computer Science

Institute of applied sciences (INSA) Lyon

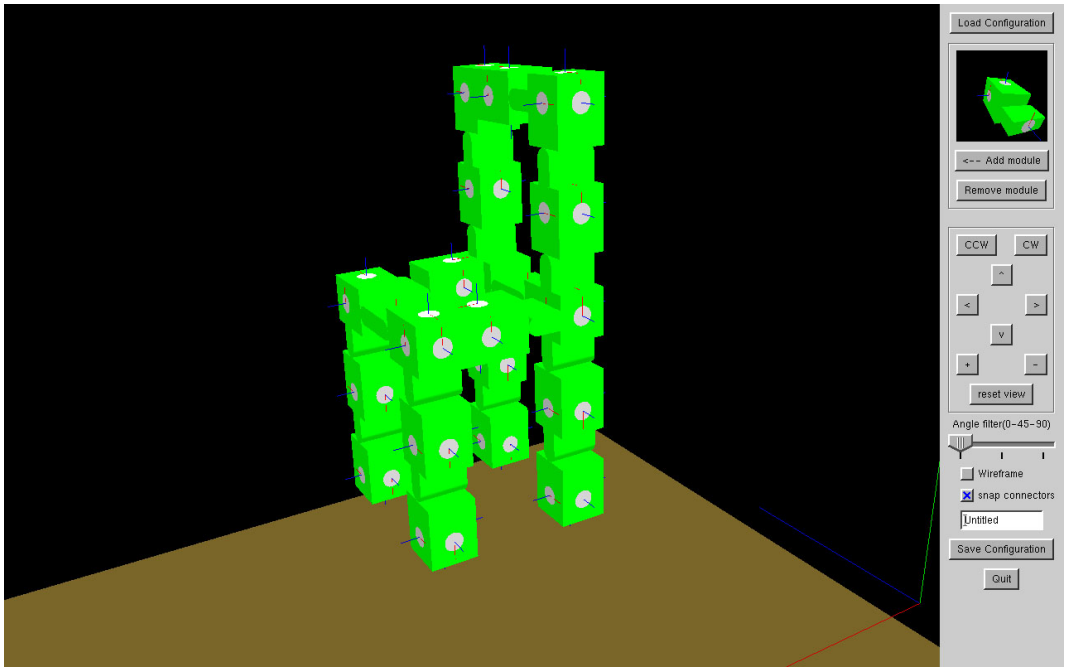
2007

Contents

Chapter 1

Introduction

IMOROD stands for ***I****nterface for* ***MO****dular* ***RO****bots* ***D****esign*. IMOROD is a 3D interface meant to very easily assemble robotic modules to build complex modular structures. It has been designed originally in the framework of the project Roombots to intuitively create furniture made with Roombots robot modules, but can be extended to the design of any modular robotic structure. As IMOROD may be further developed, this guide is meant to help the developer to understand the structure and functioning of the program.



This program is based on the GLOW library, which is a very simple widgets and user

interface design library, implemented in C++ and thus fully object oriented. GLOW acts also as a C++ wrapper for OpenGL and GLUT. These two libraries have been implemented in C and are thus not well adapted to object oriented languages. GLOW implements a set of objects, mapping their member functions to OpenGL and GLUT procedures. This enables the developer to use OpenGL and GLUT functions in C++ objects.

The full reference of the GLOW library as well as some tutorials can be found at http://glow.sourceforge.net/ as well as in the GLOW package.

Chapter 2

Structure of the Program

THIS chapter details the structure of IMOROD.

2.1 General structure

IMROD is split into three modules, or packs of functionalities.
The three modules are called :

- Furniture Design
- Room Arrangement Design
- Reconfiguration Visualization

This implies three different windows. The *Furniture Design* window contains every tools necessary for the user to manipulate and connect modules in a scene, in order to compose a more complex structure. The *Room Arrangement* window is dedicated to the positioning of previously created furniture in a virtual room. The *Reconfiguration Visualization* is useful for visualizing the robot reconfiguring, transforming from one shape to another according to a previously computed sequence.

The relevant window can be chosen by the user at application startup on the *Module Selection* window. The next figure (Figure 2.1) shows the communication between the application modules and with extern applications.

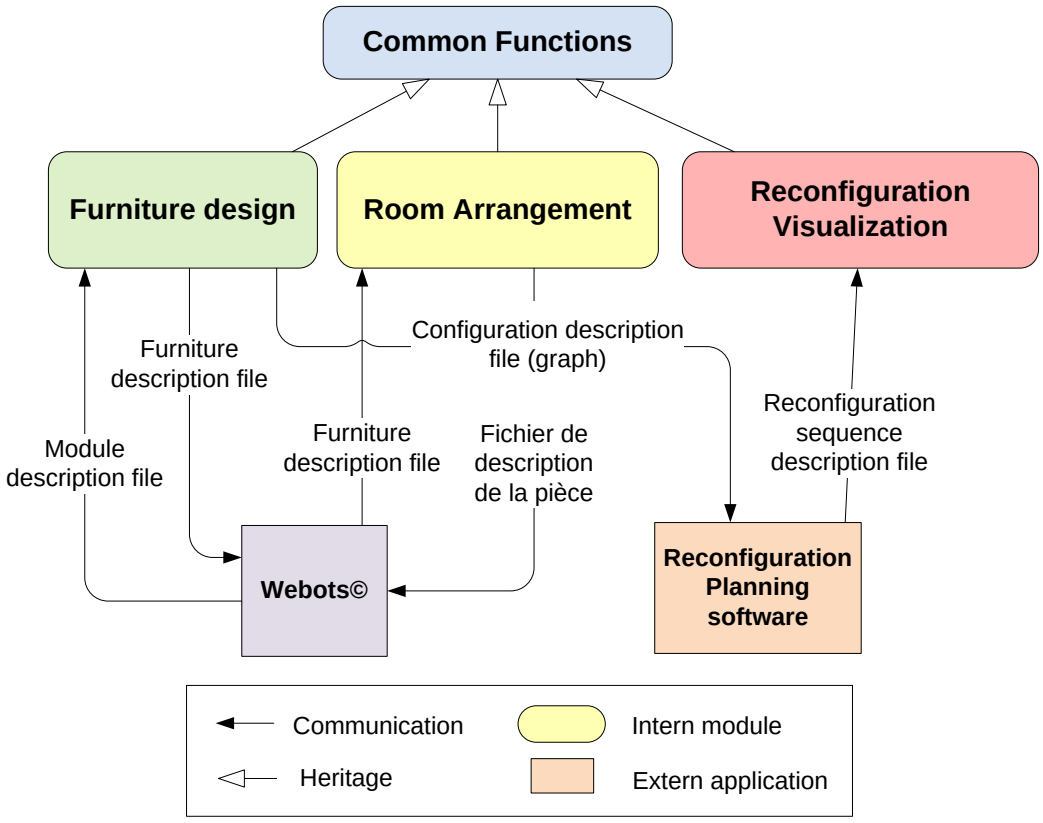


Figure 2.1: IMOROD communication model.

2.2 Detailed structure

This section details the structure of the solution, starting with the windowing system, and continuing with the scene objects to finish with the graph structure. Since only the *Furniture Design Module* has been implemented, nothing related to the two other application modules is described.

2.2.1 Windowing system

As explained before, IMOROD is composed of three main windows, one for each application module, which inherit from a common class : *DesignWindow*. The *DesignWindow* class inherits from the *GlowWindow* class, from the GLOW library. *DesignWindow* class contains two interesting objects : *DesignWidgetsPanel* and *DesignDrawingComponent*. *DesignWidgetsPanel* inherits from *GlowWidgetSubwindow* and contains all the widgets of the window (buttons, checkbox etc...). *DesignDrawingComponent* inherits from *GlowComponent* and contains the 3D scene (OpenGL scene).

Each of the three windows *FurnitureDesignWindow*, *RoomArrangementWindow* and *ReconfigurationWindow* inherits from the *DesignWindow* class and contain their own DranwingComponent and WidgetsPanel.

Figure 2.2 presents IMOROD's windowing system structure (UML 2.0 standard).

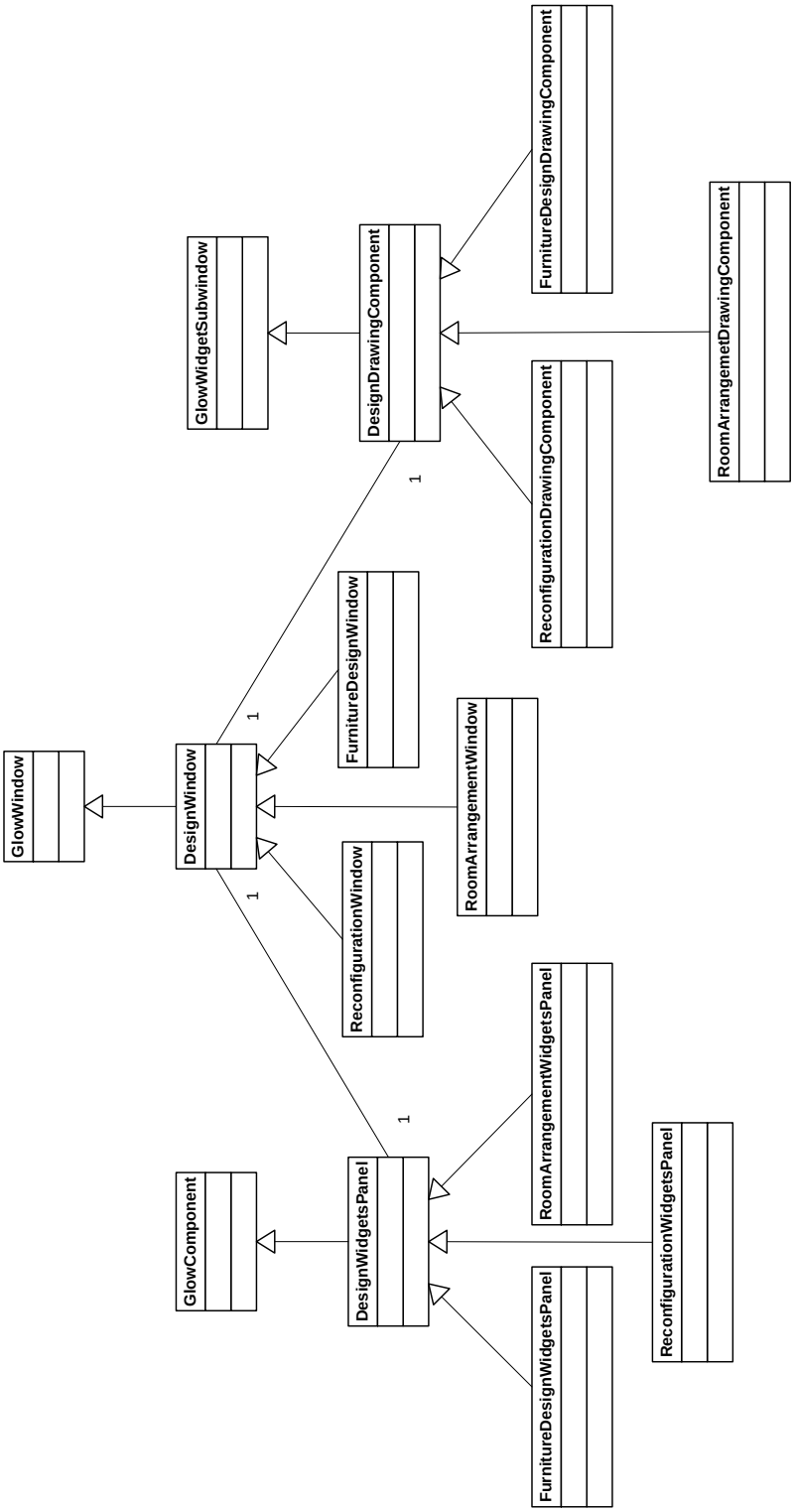


Figure 2.2: IMOROD windowing system structure

The *DesignDrawingComponent* class oontains the scene to be drawn. This scene contains elements which is a generic term for each object of the scene. A module, for instance, is an element. A module is composed of subparts, connector and of a servo. A connector is also a subpart and a servo is also composed of subparts. Examples of subparts are cubes, cylinders...

Figure 2.3 is the UML model of the classes composing the scene.

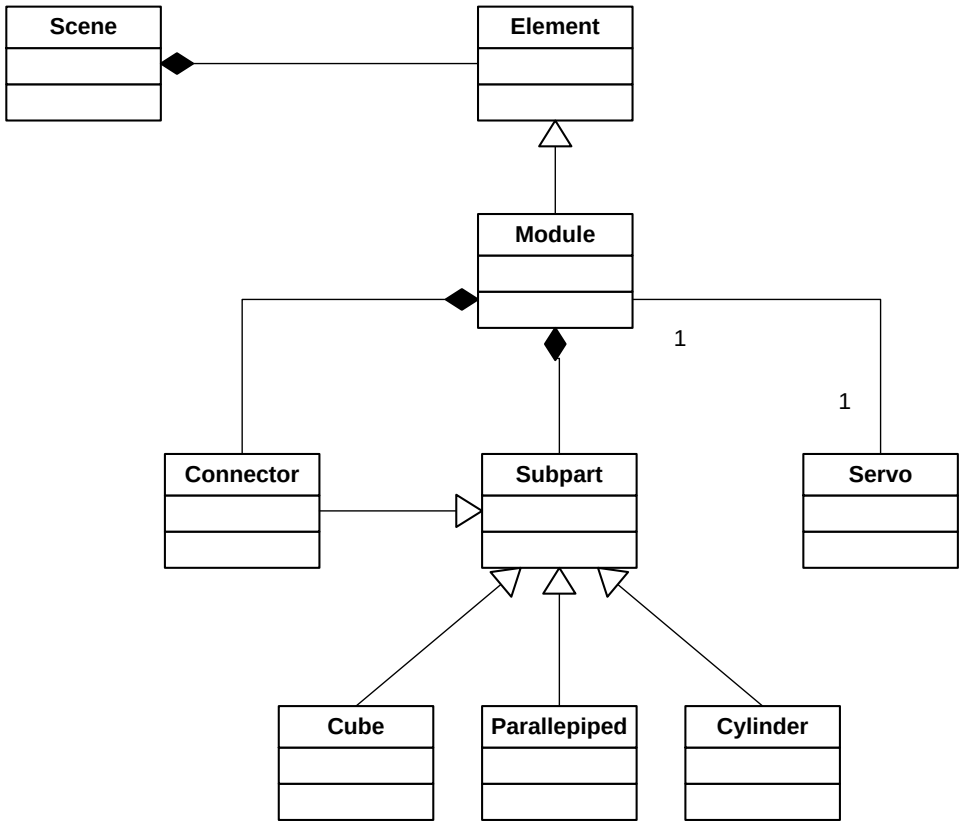


Figure 2.3: UML diagram of the classes composing the scene

The goal of the *Furniture Design Module* is to construct a configuration using virtual robotic modules and obtain a configuration description understandable by the reconfiguration planning software. The format of the file as input of this software is XML with a graph structure. Thus we maintain a graph description of the configuration where each node of the graph is a particular connector of a particular module and each edge is a set of exactly two nodes (ie. a connection).

Figure 2.4 is the UML class diagram of the graph structure we imagined.

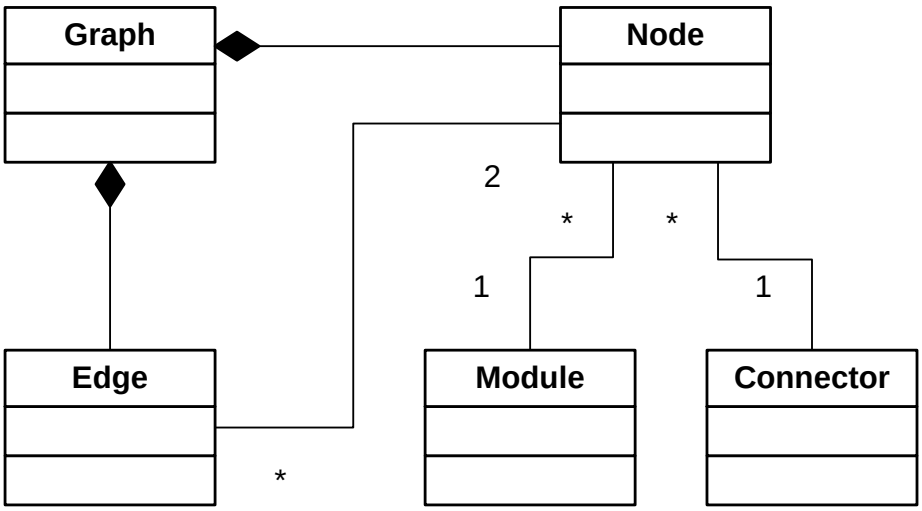


Figure 2.4: UML diagram of the graph structure

Chapter 3

Class Reference

THIS chapter is a simplified class reference and describes the classes and main functions of the solution at the time this document has been written. Only the *Furniture Design Module* has been implemented yet so the classes concerning the two other modules are not present here.

The sections have the same structure than the folders (filters) in the Visual Studio solution.

3.1 Windowing system

3.1.1 DesignWindow

Although not all the classes of this section are abstract classes, they are supposed to be inherited by other classes that would specify them : for instance add new control behaviors, new widgets ...

3.1.1.1 DesignWindow

Inherits from *GlowWindow* from the GLOW library.
This class constitutes the main window. It handles everything concerning mouse and keyboards events.
Important functions :

- **int Pick**(double x, double y, double delX, double delY) : handles mouse selection events when the user clicks on an element. It gets the name of the clicked element and selects it (with a red box).

x and **y** : mouse position when click event occurred.
delX and **delY** : maximum error tolerance for picking. For instance if delX = delY = 3, then the user can click 3 pixels around the object and still select it.

- **bool OnBeginPaint**(void) : Occurs before the scene is (re)drawn
- **void OnEndPaint**(void) : Occurs after the scene is (re)drawn
- **void OnReshape**(int myWidth, int myHeight) : Occurs when the user resizes the window
- **virtual void OnMouseDrag**(int x, int y) : occurs when the users drags the mouse (click + move)
- **virtual void OnMouseUp**(Glow::MouseButton button, int x, int y, Glow::Modifiers modifiers) : occurs when the users pushes one of the button of the mouse down
- **virtual void OnMouseDown**(Glow::MouseButton button, int x, int y, Glow::Modifiers modifiers) : occurs when the users releases one of the button of the mouse
- **virtual void OnKeyboard**(Glow::KeyCode key, int x, int y, Glow::Modifiers modifiers) : occurs when the user presses on a key

3.1.1.2 DesignWidgetsPanel

Inherits from *GlowWidgetSubwindow* from the GLOW library.
This class handles everything linked to widgets (creation, events...).
Important functions :

- **virtual void OnMessage**(const GlowPushButtonMessage &message) : The function to tackle buttons click events
- **virtual void OnMessage**(const GlowCheckBoxMessage &message) : The function to tackle checkbox events
- **virtual void OpenFileWindowCallBack**(const char *fileSelected, bool OK = true) = 0 : callback function for the open file dialog (called after the dialog is closed).
- **void RepositionWidgets**(int startPosition, int yOffset) : When inserting a new widget between two already placed widgets, this function repositions vertically the controls considering the new inserted control.

startPosition : the starting position, in the list of widgets, of the controls to reposition.
yOffset : the offset to add to each y coordinate of the controls to reposition.

3.1.1.3 DesignDrawingComponent

Inherits from *GlowComponent* from the GLOW library. This class handles everything linked to the scene manipulation : movements of modules, connections and disconnections. Its main role is to make the link between the *DesignWindow* and *Scene* classes.

3.1.1.4 MovementPanel

This class inherits from the *GlowPanelWidget* class of the GLOW library and is a new widget containing all the widgets to manipulate the scene (translation, rotation, zoom).

3.1.2 FurnitureDesignWindow

Since we did not need to implement new functionalities to the *DesignDrawingComponent* class, we did not implement any *FurnitureDesignDrawingComponent* class to specify it.

3.1.2.1 FurnitureDesignWindow

Inherits from *DesignWindow*.
Important functions :

- **fileErrors Load**(const char *filename) : Loads a configuration from a file, using the parameter filename as input. if there is a file error, the function returns an enumeration value for the corresponding type of error.

3.1.2.2 FurnitureDesignWidgetsPanel

Inherits from *DesignWidgetsPanel*.
This class adds some widgets to the *DesignWidgetsPanel* and handles their events.

3.1.3 ModuleWidget

These classes defines a new widget : the widget representing the current module in the top left hand corner of the window.

3.1.3.1 ModuleWidget

Inherits from GlowWidget from the GLOW library.
This class defines and draws the widget.
GlowViewManipulator* widgetManipulator : the object enabling the user to rotate the drawing component inside the widget.

3.1.3.2 ModuleWidgetDrawingComponent

Inherits from GlowComponent from the GLOW library.
This class defines the content of the drawing region of the widget (the module).

3.1.4 ModuleSelectionWindow

This class defines the window for selecting one of the three application module when the application starts.

3.1.5 OpenFileWindow

This class defines the dialog for selecting a configuration in the configuration list.

3.1.6 ReconfigurationSequencesWindow

Under developpement. 

3.1.7 RoomArrangementWindow

Under developpement. 

3.2 OpenGL scene

3.2.1 Scene

This class defines the scene and the actions that can be done on the scene.
Important functions :

- **void Connect**(int moduleID1, int connectorID1, int moduleID2, int connectorID2, float angle) : Connects the two specified modules by their specified connectors and with the specified angle.

- **void Disconnect**(int moduleID1, int connectorID1, int moduleID2, int connectorID2) : Disconnects two connectors.

- **void Snap**(int moduleID) : snaps the element of ID elementID to the an eventual near module : when dragging a module around, if the snapping mode is enabled and the moving modules gets near from another module, this function automatically positions and connects the two modules.

3.2.2 Element

A scene contains objects defined as elements. This class defines the large concept of element and the generic actions that can be performed on it and is meant to be inherited by more specific classes.
Important functions :

- **virtual void Translate**(const Vec3f &translation) : translate an element by the specified translation vector.

- **virtual void Rotate**(const Vec3f &pointOffset, const Vec3f &axis, float angle, bool absoluteFrame = false) : Rotates the element around the axis of direction vector axis and passing through the the point of relative coordinates pointOffset and of the specified angle. The last parameter specifies if the basis to consider is absolute or relative.

- **const Mat4f &GetTransformationMatrix**(void) const : returns the transformation matrix of the element. This transformation matrix is a 4x4 matrix in standard OpenGL format.

3.2.3 Module

Inherits from *Element*.
This class describes the module object and the specific actions that can be applied like the rotation of its servo.
Important functions :

- **void RefreshConnectorPositions**(void) : computes the absolute position of each connector of the module and stores it into a global list. This is useful to quickly find out which connector is the nearest from a specific connector when snapping two modules.

- **void RotateServo**(float angle) : rotates the servo on its degree of freedom.

3.2.4 SubPart

A module contains objects defined as subparts. This class defines the large concept of subpart and the generic actions that can be performed on it and is meant to be inherited by more specific classes. Examples of subparts include a cube, a cylinder etc...

3.2.5 Connector

Inherits from *Subpart*.
Defines a connector which is a particular type of subpart.

3.2.6 Servo

Inherits from *Subpart*.
This class defines a servo which can be composed of other subparts, here, a cylinder and a parallelepiped.

3.2.7 Parallelepiped, Cylinder, Cube

These classes are particular types of subparts and are sufficiently explicit in themselves.

3.3 Graph structure

3.3.1 Graph

This class represents the graph structure of the robotic configuration. It's composed of a set of nodes (robot modules and connectors) and edges (connections between modules).

- **bool EdgeExists**(int moduleID1, int connectorID1, int moduleID2, int connectorID2) const : returns true if the particular connection between the specified connectors of the specified modules.
- **void GetSubGraph**(int moduleID, int connectorID, vector<int> &subGraph, int originModuleID) const : returns the subgraph connected to the specified module and connector. This is useful when some modules are connected to a servo and one try to rotate this servo. The whole connected subgraph should be rotated as well.
- **fileErrors Save**(char *inputFilename, char *outputFilename) const : saves the current configuration graph in a XML file and adds it to the list of configurations.

3.3.2 Node

Defines a node of the graph as a specific connector of a specific module.

3.3.3 Edge

Defines an edge as a set of two nodes. Important functions :

- **bool IsEqual**(int moduleID1, int connectorID1, int moduleID2, int connectorID2) const : returns true if the edge represents the connection between the specified modules and connectors.
- **bool Contains**(int moduleID, int connectorID) const : returns true if the specified node (module and connector) appears in this edge, ie. if it's connected.
- **bool Contains**(int moduleID) const : the same way as the precedent, returns true if the specified module is connected (by at least one connector).

Chapter 4

Visualize reconfiguration

UNDER development.