University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering
Centre of Computer Graphics and Data Visualization

MVE - 2

Handbook

Draft version

Milan Frank and coll.
29/06/05 (Version: beta-2)

Table of Content

1.Foreword....2
2.Introduction....2
2.1.MVE-2 structure....3
3.Module map execution engine....3
4.Module Creation....5
4.1.Quick start....5
4.2.Module API reference....6
4.3.Module setup creation....9
5.Data type creation....10

1. Foreword

During my Ing, (Msc.) studies at the Computer Graphics and Data Visualization prof. Skala introduced to me a principle of modular pipeline-based data processing. I was impressed by the simple, genial idea of interconnection of several well defined modules together to solve a problem. Those times an object-oriented programming was my hobby. Thus, an idea of object-oriented modular system using modern programming technology infested my mind. Several years later this idea is realized in the form of MVE-2.

Milan Frank

2. Introduction

MVE-2 is a modular environment based on data-flow principle. It offers general and easy-to use interface for creation of modules (active code) and data objects that can be shared among modules. Core of the system is a runtime with interesting features that allows good module interconnection capabilities. Two most prominent features are cyclic interconnection and module driven sub-map execution.

There are two ways how to define a module interconnection. The first one is via GUI (*MapEditor*). It is intuitive graphical interface that allows full control on the module map. The GUI generates a XML representation of a module map that can be edited manually by arbitrary text editor. The XML representation can be executed directly by *RunMap* utility from the command line.

The MVE-2 system is rewritten from scratch and does not have a single piece of code common with MVE developed in 1996. This is mostly due to platform change, from Win32 to .NET. Backward compatibility is planned by a set proxy modules. But, there will be a problem of data conversion, which means a huge memory and time efficiency overhead.

Huge difference between the recent version and the new version lies in the generality of the core. Previous MVE (1996) system had a set of build-in data types in the core. It was very difficult to add a new one. MVE-2 has no core data types. It offers only a abstract base class (and an interface) for all data types. Thus any module library can define its own data types.

The generality of the core evoke a thought that the MVE-2 core is not only a "visualization"

environment. It can be applied to a variety of problem solution that have a data-flow nature. It depends only on a particular module library, its data types and modules. For example, a set of modules for text processing can be easily implemented.

However, there is a need to define a set of common data types to allow module developers create modules compatible with each other. That is why we offer a Numerics and Visualization library that contains a set of basic data set for Mathematics, Visualization and Computer Graphics.

Design of Visualization library is similar to VTK (Visualization ToolKit developed by Kitware Inc.). The MVE-2 structures are more general and more “object-oriented”. Many differences arise from *C#*/.NET and *C++* environments.

2.1. MVE-2 structure

Following picture (Figure - 1) demonstrates the general structure of MVE-2 system. The *Core* provides base classes for modules and data structures and module management system (runtime). Particular modules and data structures are implemented in modules libraries. They depend on *Core* and may cooperate with each other.

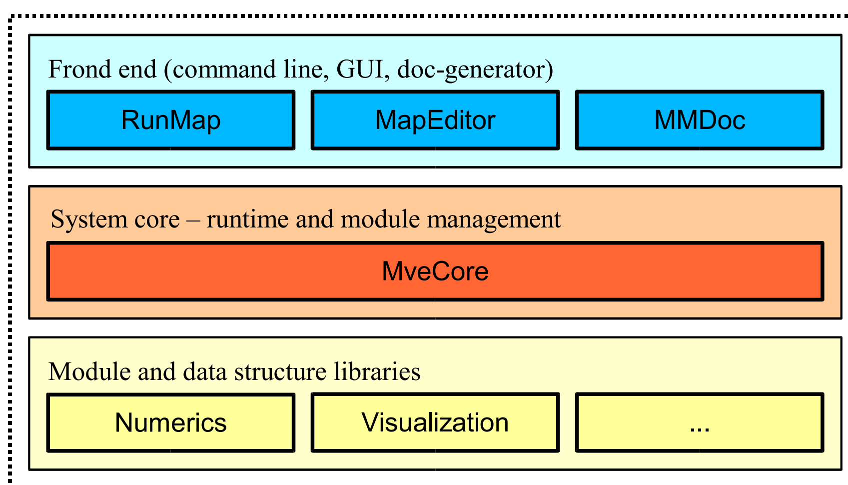


Figure 1: This figure illustrates a general structure of MVE-project. Each sub-block represent one .NET assembly.

The frond-end parts allows to use MVE by non-programmer users. The *RunMap* provides command line interface to execute a module map defined in XML file. The *MapEdior* is user friendly GUI that allows intuitive editing and executing of module maps. The *MMDoc* is a tool for automatic generation of modules and data structures documentation.

3. Module map execution engine

By word of graph theory a **module map** is a directed multigraph consisting of modules (nodes) and interconnections (directed edges). If there are no edges leading to a module, the module is called *source module*. If there are no edges leading from a module, the module is called *terminal module*. If a module have edges leading from the module and each one starts in port marked as *non-evoke-update*, the module is also *terminal module*. If there is a module without any connection, the module is also considerer as a terminal module. All other modules are called *filters*.

One **module map execution** is defined as correct execution of each terminal module. Correct execution of a module means that a module is executed when all input data are ready on its

auto-update ports. The only exception are *DelayModules* (will be explained later). This simple rule gives us a recursive formula to execute the whole map. The recursive descent is called an *update*. The ascending process is called *execution*. Obviously, the update process ends with *source modules* also *terminal modules* are executed latest.

The place of connection is called ***port***. Each connection starts at one output port (the place the data came from) and ends at one input port (the place the data got to). Number of connections can start at output port. Only one connection can be connected to an input port. Each port has a type that accept or is its source. Thus there is a type checking mechanism that allows compatible connections only.

The ***DelayModule*** is a special module that acts as a memory with initialization. It returns value from N-1 step (of the map execution) connected to its *input* port. In the first step it returns the value connected to the *initial* port. This module is handled differently by the runtime than ordinary modules. It allows a cyclic connection.

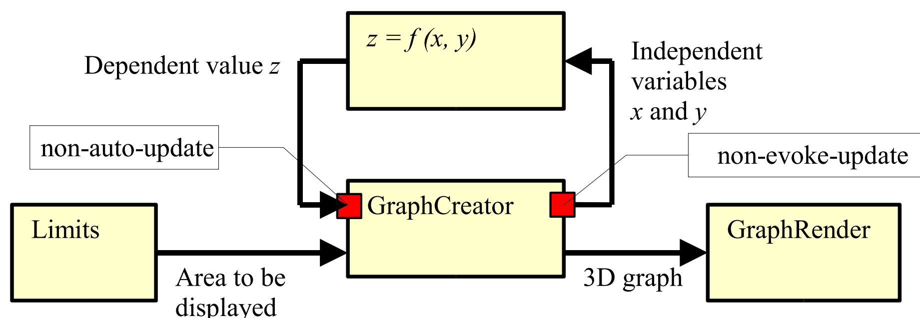
Another way to create a correct cycle is using a special output port, marked as *non-evoke-update*. In short, it means the port has its initial value and reading from this port (data request) never causes execution (validation) of the module.

If we remove all connections that end in input ports of delay modules or end in *non-evoke-update* input ports, the module map graph must become a tree.

Input ports can be marked as ***non-auto-update*** (*auto-update* is implicit). The implicit *auto-update* behaviour means that all input data are ready before execution of the module. The *non-auto-update* behaviour means that port gives-up this implicit data preparation. On the other hand, the port can request data at any time during its execution (by *UpdateInput* call). Thus, a module can control execution of the part of the module map that is connected to its *non-auto-update* port. We call this part a *sub-map*.

The opposite for *non-auto-update* input port is the previously mentioned ***non-evoke-update*** output port. Its feature is that it always has a valid value and reading of this value never causes execution of the module.

The primary reason for introducing *non-evoke-update* and *non-auto-update* port is shown on the following example. The *non-evoke-update* port can be a source of data for a module driven *sub-map*. The *sub-map* starts at the *non-auto-update* port and is driven by it. See the following picture.



The module map on the picture draws a 3D graph of a $z=f(x,y)$ function in the area limited by the *Limits* module. The whole graph is drawn during one map execution. It means, the terminal module runs only once as well as its source. On the other hand, the $z=f(x,y)$ module runs *n*-times. The *n* depends on the *GraphCreator*. The *GraphCreator* exposes the *x* and *y* variables on

non-evoke-update output port. Then the *GraphCreator* calls *UpdateInput* on the *non-auto-update* input port. It causes execution of the $z = f(x, y)$ module. Then the *GraphCreator* reads the input port and saves the value to its internal structure and repeats with different *x*, *y* values. After creation of enough samples it returns some representation of the generated 3D graph. Finally the *GraphRenderer* is executed.

It is obvious that a sub-map can contain a module with non-auto-update port. Thus, we obtain a sub-map of a sub-map. This is the reason for introducing the *map-level* term. All terminal modules are of level zero. The rest of module-map is marked to satisfy following rules.

- Source module have to be of lower level than target module if and only if the input port is *non-auto-update*.
- Source module can be of higher level than target module only if the target module is a *DelayModule* or source port is *non-evoke-update*.
- Other interconnections have to be between modules of the same level.

The marking of module levels is fully automated and the consistency is checked before execution. If there is some inconsistency user is notified. The map level can be displayed in the *MapEditor*.

The execution mechanism is rather complicated due to allowed cycles, *DelayModules*, *non-auto/evoke-update* ports. The interconnection possibilities are huge. Current version of the runtime should handle all combinations that “make sense”. But, nothing is perfect. If our user find some interconnection that should work but does not, please do not hesitate and inform someone from MVE-2 developer team.

4. Module Creation

Modules in the MVE-2 are organized in assembly/assemblies (this is a .NET term). Each assembly can contain arbitrary number of modules, data objects and others .NET entities organized in arbitrary number of namespaces. The *MapEditor* shows all public modules in all assemblies loaded in library directories (see the MveCore.config file). The *MapEditor* shows the modules in a tree hierarchy organized by namespaces.

4.1. Quick start

In this chapter we describe how to create a simple module. It is something like classic “hello world” application. This module prints input data as a string using the standard *ToString()* method.

Each module has to be derived from *Zcu.Mve.Core.Module* abstract class and has to override constructor and *Execute* method. The constructor typically creates ports and of course, may initialize user data. The *Execute* method implements the “activity” of the module.

In this particular example, the constructor creates one input port named *input* that accepts any instance of *IDataObject*. The first line in *Execute* method reads data from *input* port and the second one prints them on the standard output (console).

```
public class ConsolePrinter : Zcu.Mve.Core.Module
{
  public ConsolePrinter()
  {
    this.AddInPort("input", typeof(Zcu.Mve.Core.IDataObject));
  }

  public override void Execute()
  {
    IDataObject data = (IDataObject) GetInput("input");
    Console.WriteLine(data.ToString());
  }
}
```

Let's have a more detailed look on this example. The *ConsolePrinter* class is derived from an abstract class named *Module* defined in the system core. This way it became understandable for the runtime and visible via *ModuleView* in *MapEditor*. It is clear that a module class has to be public, otherwise is inaccessible from the outside of the assembly.

The standard place where to create ports is the constructor. Creation of port itself is realized by *AddInPort* and *AddOutPort*. The first parameter of these methods is a port name. It is a module-unique identifier of particular port. The second parameter defines data type that is accepted by the port. The runtime checks whether interconnection makes a type conflict. In this particular example we accept instances of all classes or structures that implement *IDataObject*. Thus we accept everything because all data objects have to implement this interface (or to be derived from *DataObject*, which implement *IDataObject* interface itself).

The *Execute* method is an implementation of the module activity. In the first line *GetInput* method reads data from port the named “input”. This method returns the *IDataObject*. Thus, usually we have to typecast it. Now we have a reference to the incoming data in *Execute* method. These data are processed in the second line where we print them by standard .NET method call.

It is very important to note that we must not modify the incoming data because they can be shared among number of modules. The only module permitted to modify them is the source module of the data. This is a weak point of all typical implementation of the pipeline systems. Due to performance and complexity reasons we did not introduce some permissions rules system that can assure data object read only access. Therefore it is only a matter of decency of the the developer not to modify the incoming data. Following example shows the typical use of *GetInput* method.

```
ScalarNumber num = (ScalarNumber)GetInput("input");
```

The way how to expose data to an output port is via *SetOutput* call. This method lets MVE runtime know that new data are ready. If the *SetOutput* is not called the runtime interprets the previously exposed data as “still valid and not modified”. If such thing happened in the first execution the null reference exception will be probably thrown by a module while reading content of such port.

4.2. Module API reference

Each module is derived from *Zcu.Mve.Core.Module*. This base class is quite complex and contains number of methods to be called, number of methods to be overridden and several events that can be used.

A module exist as long as whole module map. It means that its constructor is called when the module is added to a module map. The module is disposed when it is removed from a map or a module map is disposed at all. The instance of a module exists during unspecified number of module map runs. Thus, it has to be written with respect to this fact.

4.2.1. Mandatory Overriding

Execute

Main method that implements activity of an object. Typically reads an input and generates an output.

Constructor

An implicit constructor is a typical place to create ports and to do any other initialization. See also *SimulationStart* and *ModuleCreate* events to do some special initialization.

4.2.2. Possible Overriding

InvokeSetup

This method returns an instance of *ModuleSetup*. This user control (*ModuleSetup*) is displayed in the module setup dialog in *MapEditor*.

Implicit behaviour (without overriding) automatically creates a property grid that allows simple editing of elemental properties. By overriding programmer may return its own

WriteConfig(XmlElment config)

This method can fill the xml element with arbitrary data. These data are stored by MVE runtime in an xml representation of module map in *config* node .

Typical use of it is for a module specific configuration data.

It is obvious that these data should remain “small”. For example: it is recommended to save file name of some data file but it is not recommended to save these data itself.

Implicit behaviour (without overriding) creates a subelement Properties, where all public properties of the module are saved. For each property a subnode is created named after the property, with attribute named "type", which contains a "ToString" result of the property type. Text of each property elemnt is created by calling ToString upon the property value, or ToString (null,Globals.nfi) if the Iformattable interface is implemented by property type.

ReadConfig(XmlElment config)

This method is the opposite to *WriteConfig* method. The config xml node contains data read from xml representation of module map.

Default implementation of this method tries to find subelement "Properties" within which it searches for property values. See Writeconfig for details about formatting. Tries to find corresponding property (correct name and type) and sets its value. Only works for predefined value types (Double, Single, IntXX, UIntXX, Boolean, Decimal) and String!

DeepCopy()

This method is used by MapEditor whenever user copy a module. (CTRL+C, CTRL+V operation). Instance creation, name, position and orientation of a module is implemented by

DeepCopy of Module (base.DeepCopy()). All other data have to be handled by user.

Author of module do not have to implement this method at all but user can encounter unfriendly behaviour of such module. The copy of such module have implicit setting.

4.2.3. Methods to be called

AddInputPort(string name, Type type, [bool required, bool autoUpdate])

This method add an input port to a module. The first parameter is locally unique identifier of the port. The second one defines type of the port. The *Type* can be easily obtained by `typoeof` operator. Two other parameters are optional. The *required* parameter define whether the absence of connection to the port cause error during module map execution. Implicit value is *true*. The *autoUpdate* parameter allows programmer to create input port that have control on dependent sub-map.

AddOutputPort(string name, Type type, [bool evokeUpdate])

This method add an output port to a module. The first and the second parameters have the same meaning as in the *AddInputPort* case. The last parameter is optional and allows to create non-evoke-update port. Implicit value is *true*. In case of *non-evoke-update* port it is necessary to give some implicit value before a module map is executed.

Constructor is not the only place were to call AddInput/OutputPort method. Ports can be added any time except module-map execution.

GetInput(string name)

Read value of input port. Caution! Content of such data must not be modified!

SetOutput(string name, DataObject data, [bool dataSame])

Expose data at output port.

dataSame is optional flag passed to modules reading the port. It allows optimization of calculation in case the outgoing data are the same as in previous execution call.

UpdateInput (string portName)

The port named *portName* has to be marked as *non-auto-update*. This method cause execution of sub-map connected at particular port. After the method is finished the data are ready at particular port.

RemoveInPort/RemoveOutPort()

This method remove a port from module. It must not be called during module map execution.

IsDataSame(string portName)

Returns true if data at port are the same as in previous module execution. See the third parameter in *SetOutput* method.

ProgressInfo(float progress)

Inform runtime how the Execute of module works. Expected range is (0.0 to 1.0)

4.2.4. Events

SimulationStart/SimulationEnd

Start/End of module map execution.

Prototype:

```
void SimulationStartCallBack(object source);
void SimulationEndCallBack(object source);
```

PortsChanged

Any change in ports configuration (add, remove, data type change). Important for MapEditor.

Prototype:

```
void PortsChangedCallBack(object source, Port port, PortChangeType
change);
```

StateChanged

Indicate the change in module state (waiting/updating/running). Important for MapEditor.

Prototyp:

```
void StateChangedCallBack(object source, ModuleState newState);
```

Connect(portFrom, portTo, allowed) / Disconnect(port)

Event is set before connection/disconnection to/from input port. It is possible to disable the connection operation via allowed parameter.

Prototype:

```
void ConnectCallBack(object source, OutPort portFrom, InPort portTo, out
bool allowed)
void DisconnectCallBack(object source, InPort portTo, out bool allowed)
```

DataReady

Event is set after data are exposed in output port connected to input port of current module.

Prototype:

```
void DataReadyCallBack(object source, InPort inPort);
```

ModuleCreate

This event is set after the module is added into module map. The difference between constructor and the *ModuleCreate* is that the instance of module map is accessible.

It is possible to cancel the module creation via throw a *MveException*.

Prototype:

```
void ModuleCreateCallBack(object source)
```

4.3. Module setup creation

There are two ways to create a module setup. The first one is to derive a ModuleSetup class, which is derived from the UserControl and may contain standard WinForm content. In ModuleSetup child it is necessary to override OnOK and OnStorno method that are called as a reaction on particular user click.

The second way is to use implicit behaviour of the InvokeSetup method. It returns an automatically generated PropertyGrid, which allows to modify values of module properties of a primitive data type. It is possible to disable a property from browsing via property grid by *[Browsable()]* attribute. It is also possible to define a description and a category of a property via *[Description()]* and *[CategoryAttribute()]*. For more details see *GuifiedModule* in *Examples* library. There is also

possible to combine these two methods and add the *PropertyGrid* as a component of the *SetupDialog*.

5. Data type creation

Extensive recognition should proceed introduction of a new data type. Some useful data structure may already exist together with number of useful modules. Existence of duplicity in data structures is very contra productive and result into incompatibility of modules.

Data type (API) have lesser number of methods than module (API) but each one is declared abstract. Thus, overriding is mandatory. Each data object that is about to be shared among modules have to be derived from *Zcu.Mve.Core.DataObject* or implement *Zcu.Mve.Core.IDataObject.*

Good example is *ScalarNumber*. It can be found in *Examples* source code. It is simple representation of one scalar value. Following text explain and comment all important parts.

Private field *val* represent the scalar data itself. Access is possible by public *Val* property. Implicit value is set in parameterless constructor. Each correct data type have to define an implicit constructor. Otherwise, the *XmlLoader* and *XmlSaver* will not work.

ReadData and ***WriteData*** are important methods called by *XmlLoader* and *XmlSaver*. Implementation of these methods define the process of serialization to and from XML file. The idea is "author of data structure knows best how to efficiently and clearly transform the structure to and from XML representation". The *ReadData* method should be able to handle XML file that is not actually written by the *WriteData* method.

The ***DeepCopy*** method have to allocate new memory space and copy whole content of data structure. This method is used by *DelayModules* to store result of previous step.

The ***CheckConsistence*** method is important for complex data structures where some inconsistencies may appear. (e.g. index of triangle vertex that exceed the number of points) In such cases the method should return *false* and let user know there is something wrong in the data structure. It is also recommended to put on standard output a message that explain what is wrong. It is also possible to return *true* and write a warning (e.g. Collapsed triangle – data consistent but suspicious)

There is also recommended to override ***ToString*** method. This method is used by *ConslePrinter* module.