VT plugin design proposal

ABI team 33
(Stefan Versluys, Guus Bonnema, Jeroen Kleijn)

21/02/2015

Modification History		
Guus Bonnema	21-02-2015	Initial proposal document

Contents

1 Problem background and description

1.1 Why do we need a plugin?

The XMD subsystem was designed to be independent from the VT subsystem. Still these subsystems do have to cooperate in order to show VT output during graphical design of a network.

In order to have a loose coupling between the two subsystems the idea is to have a plugin structure where the VT tool designer is able to integrate a verification tool in a dynamic manner.

1.2 What should the plugin do?

The plugin should run one or more of the verification tools and redirect standard output to the graphical designer, that will display the information in a console like fashion.

The plugin should also be able to return specific information of any level (info, warning or error) to the graphical tool. This could be information about one specific component, a channel, or it could be information about a group of components and channels.

Implementation of the text warning and error interface is the minimum requirement.

Standard output and Standard error In order to connect the standard output and standard error to a widget, one can use a subclass of `QProcess` and connect the output and error channels to the right widgets using signals and slots as follows:

```
QProcess process = new MyProcess();
process.setProcessChannelMode(QProcess::MergedChannels);
process.start("<processname>");
```

The subclass of `QProcess` could contain something like:

```
class myProcess() : public QProcess
{
  Q_OBJECT
  public:
    myProcess() {
      edit = new QTextEdit();
      edit->resize(500,500);
      edit->setWindowTitle("Standard output");
      edit->show();
      connect(this, SIGNAL(ReadyReadSignalOutput),
              this, SLOT(readStdOutput));
      connect(this, SIGNAL(Error(QProcess::ProcessError)),
              this, SLOT(readError()));
    };

      ~myProcess() {};
  private slots:
    void readStdOutput() {
      edit->append(readAllStandardOutput());
    }
    void readError() {
      edit->append("An Error Occured! Error Code is: "
                   +QString::number(error()));
    };
}
```

with the method

2 VT plugin function interface

2.1 What interfaces do we need?

text information Lines of information in a default color. This is the minimum interface requirement for tracking the running verification tool.

Info, warning, error Lines of information with a specified label. This is an extra and optional requirement for tracking the running verification tool.

Component or Channel Lines of information concerning a component. The effect could translate into a color of the object on screen plus a line a text. This is a nice to have requirement for tracking the running verification tool.

2.2 How to define the interfaces?

Each interface is a class containing pure virtual methods. Each method is connected with the user interface through signals and slots. The initiative for issuing a message originates from the verification tool. The user interface should show these messages in text or in graphics.

The proposal for an interface will be methods with the following signature:

```
virtual bool run_json_string(QString jsonString) = 0;
virtual bool run_json_file(QString filename) = 0;
virtual bool run_json_xmas(std::map<std::string, <XMASComponent *>>);
```

A plugin should implement all three methods and return `true` if successful, `false` if the run was not successful. The executable should be run sequentially, not concurrently, as concurrency is catered for in the verification tools.

The plugin should also connect the standard output and standard error to the graphical text window.

Optional additions If possible errors and warning should be printed in a different color. If possible, errors and warnings should be incorporated into the graphical view of the network.

2.3 Plugin definition

The plugin is a Qt program that implements one or more interfaces. The verification tool connects to the interface methods by calling them from the code.

2.4 Changes in the verification tool

The verification tools are classes that we can either extend through subclasses or use as-is. In both cases, the verification tools do not need any change to redirect standard output or standard error to the plugin. They do need change for component directed messages.

2.5 How to implement parameters?

The VT must relay the parameter.

3 Qt Plugin architecture

3.1 General description

The QT system supports two forms of plugin, one that supports a QT extension through plugin, and the other that supports general extension of an application through a plugin. It is the second form that we are discussing in this document.

3.2 Plugin structure

An application plugin using QT is based on defining an interface and a registration with the QT system using a `QObject` derived class. After registration the QT system provides a class that will load a program and determine whether it is a plugin, and if so, determine whether it is the plugin we are looking for.

The class implementing the plugin has the following structure:

```
class <type>Plugin : public QObject, <type>Interface
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "unique.prefix.<type>Interface" FILE "<type>plugin.json")
    Q_INTERFACES(<type>Interface)

public:
    QString <method>(const QString &message) Q_DECL_OVERRIDE;
};
```

In this code one can replace `<type>` by the type of interface we are creating. The example this code is started from is an echo server where the interface was called `EchoInterface`. The `<method>` should be defined as pure virtual in the interface. The interface should contain a virtual destructor and only pure virtual methods or the moc-compiler will complain.

Following this structure one can implement a plugin loader. For example the following code was copied from the example of an echo plugin in the `QtCreator`:

```
bool EchoWindow::loadPlugin()
{
    QDir pluginsDir(qApp->applicationDirPath());
#if defined(Q_OS_WIN)
    if (pluginsDir.dirName().toLower() == "debug" || pluginsDir.dirName().toLower() == "r
        pluginsDir.cdUp();
#elif defined(Q_OS_MAC)
    if (pluginsDir.dirName() == "MacOS") {
        pluginsDir.cdUp();
        pluginsDir.cdUp();
        pluginsDir.cdUp();
    }
#endif
    pluginsDir.cd("plugins");
    foreach (QString fileName, pluginsDir.entryList(QDir::Files)) {
        QPluginLoader pluginLoader(pluginsDir.absoluteFilePath(fileName));
        QObject *plugin = pluginLoader.instance();
        if (plugin) {
            echoInterface = qobject_cast<EchoInterface *>(plugin);
            if (echoInterface)
                return true;
        }
    }

    return false;
}
```

It first determines for a file whether it is a plugin and subsequently which interface it implements.