TITLE

MIGRATING OF SALOME MODULES ON NEW SALOME GUI ARCHITECTURE

Specification

Revision history:

Version	**Reviser**	**Revision date**	**Remarks**
0.1	VSR	27/06/05	

Table of content

1. Introduction 3

2. General information 3

2.1. The structure of the document 3

2.2. Terms and abbreviations 3

3. C++ modules migrating 3

3.1. Configuration and building 3
- 3.1.1. msg2qm 3
- 3.1.2. SALOME GUI 4
- 3.1.3. Additional configuration procedures 4
- 3.1.4. Building of module's GUI library 4

3.2. Sources adopting 5
- 3.2.1. Module 5
- 3.2.2. Data model 6
- 3.2.3. Selection handling 7
- 3.2.4. Menus, toolbars and context menus management 7
- 3.2.5. Resources and preferences handling 8
- 3.2.6. View management 10
- 3.2.7. Object Browser, python console, etc. 10

3.3. Launching 11

4. Python modules migrating 11

4.1. Python API 11

4.2. Configuration and building 11
- 4.2.1. msg2qm 12
- 4.2.2. Menu resources 12

4.3. Launching 12

4.4. Caveats 12
- 4.4.1. Multi-desktop environment 12
- 4.4.2. Workspace 12

APPENDIX 13

Check SALOME GUI procedure 13

1. INTRODUCTION

From version 3.0.0 SALOME platform proposes new SUIT-based GUI architecture. This architecture includes multi-desktop dockable-windowed interface, common tabbed workspace for all integrated viewers, new selection mechanism based on customizable selectors, unified resource manager, unified menu/toolbars/popup menus and viewers handling, common exception/signals handling, embedded CORBA engine simplifying implementation of persistence operations for the no-CORBA-engine modules, etc.
Moreover in contrast to the versions 2.x.x in SALOME 3 GUI is separated from the KERNEL implementation and provided as additional distribution.

These changes in the SALOME architecture assume the migration of all SALOME-based modules in order to be compliant with new GUI. Some parts of code need re-implementing and/or adopting according to the new rules.

2. GENERAL INFORMATION

2.1. The structure of the document

This document describes the actions to be performed for the custom SALOME modules to bring them in SALOME 3 compatible state and refers both to C++ and pure Python modules. This document does not concern changes connected with modifications of KERNEL implementation.

2.2. Terms and abbreviations

- *SUIT* – **S**ALOME **U**ser **I**nterface **T**oolkit – GUI toolkit library.

3. C++ MODULES MIGRATING

The SALOME GUI has significantly changed from version 2.x to 3.0. Almost all basic mechanisms were re-implemented. Since the GUI is now separated from the KERNEL the procedure of configuration/building of SALOME-based modules includes some additional steps and checks.
This section overviews main changes which should be taken into account when porting existing C++ modules to new GUI architecture. The procedure of porting of pure Python modules is described in the next section.
Only GUI changes are described here in this document. The changes of module engine connected with the modifications of KERNEL are out of scope of this document.

3.1. Configuration and building

To allow module sources to be compilable with new GUI some additional checks should be performed by the configure script.

3.1.1. msg2qm

msg2qm is a Qt tool which is used for the converting of *.po text files to the *.qm resources. Unfortunately this tool is not included to the standard Qt distribution. It is built and distributed as separate prerequisite product with the SALOME 3 installation procedure. To compile *.po resource files it is necessary to add msg2qm check step to the configure procedure. The corresponding **check_msg2qm.m4** file is included in the KERNEL sources package (salome_adm/unix/check_files folder). It is just necessary to modify configure.in.base file and make_commence.in file:

configure.in.base :

```
…
echo
echo ---------------------------------------------
echo testing msg2qm
echo ---------------------------------------------
echo

CHECK_MSG2QM
…
echo
echo ---------------------------------------------
echo Summary
echo ---------------------------------------------
echo

variables="cc_ok lex_yacc_ok python_ok … msg2qm_ok … Kernel_ok"
…
```

make_commence.in :

```
…
# msg2qm
MSG2QM = @MSG2QM@
…
```

Note: module is not obliged to provide *.po files if it does not need to use internationalization mechanism. There is also no need now to put empty *.po files to the module's GUI package - these files can be just omitted.

3.1.2. SALOME GUI

To ensure at the configure step that SALOME GUI distribution is set correctly the configuration procedure may need additional check. The check_GUI.m4 file provides this procedure (see APPENDIX, Check SALOME GUI procedure). To include it in the configuration script put this file in your adm_local/unix/check_files folder and modify configure.in.base file:

```
…
echo
echo ---------------------------------------------
echo Testing GUI
echo ---------------------------------------------
echo

CHECK_SALOME_GUI
…

echo
echo ---------------------------------------------
echo Summary
echo ---------------------------------------------
echo

variables="cc_ok lex_yacc_ok python_ok … Kernel_ok SalomeGUI_ok"
…
```

3.1.3. Additional configuration procedures

Depending on what SALOME GUI packages the custom module depends on (through include files or/and linkage) some additional check procedures may be needed (e.g. BOOST flags may be required). In this case these procedures should be added manually by modifying configure.in.base and make_commence.in files.

3.1.4. Building of module's GUI library

Makefile.in of the module's GUI library should be modified to take into account additional compilation/linkage flags (see previous section), e.g.:

```
…
CPPFLAGS += -I${GUI_ROOT_DIR}/include/salome
```

```
CXXFLAGS += -I${GUI_ROOT_DIR}/include/salome
LDFLAGS  += -L${GUI_ROOT_DIR}/lib/salome -lSalomeApp
…
```

In SALOME 3 default GUI library name does not have postfix "GUI". So the library target should be modified in the Makefile.in (for example, for HELLO sample module):

```
…
# Libraries targets
LIB = libHELLO.la
…
```

There is a possibility to define the name of the module's GUI library directly in the user preferences file (see 3.3 and 4.3 below). In this case it is not necessary to rename library target in the Makefile.in.

3.2. Sources adopting

3.2.1. Module

The class `SalomeApp_Module` provides basic functionality for all custom modules: it is the main GUI module class. Each custom module should implement its own module GUI class inherited from `SalomeApp_Module`. The instance of this class should be created and exported each time when application requests it by `createModule()` function, e.g.:

```
…
extern "C" {
  Standard_EXPORT CAM_Module* createModule() {
    return new GeometryGUI();
  }
}
…
```

Some methods of this class should be re-implemented in successors in order to customize the module's behavior:

Method	Description
public:	
void **initialize**(CAM_Application*);	This method is called once when the instance of the module is created by the application (when the module is loaded first time for each study). This method is usually responsible for the creation of main menu, toolbars and context popup menu actions, preparation of the popup menu manager, creation/getting of the engine reference instance, etc.
void **windows**(QMap<int, int>&) const;	This method should fill in the list of dockable GUI elements (ObjectBrowser, LogWindow, PythonConsole) which should be shown by default when the module is activated.
void **viewManagers**(QStringList&) const;	Similar to the **windows**() this method should provide the list of compatible views (OCCViewer, VTKViewer, etc).
QString **engineIOR**() const;	This method should return module engine IOR.
void **contextMenuPopup**(const QString&, QPopupMenu*, QString&);	This **optional** method can be defined in order to override default mechanism of context popup menus handling (through popup menu manager, see 3.2.4 below).
void **createPreferences**();	This **optional** method can be defined to export preferences by using preferences handling mechanism (see 3.2.5 below).

void **studyActivated**();	This **optional** method can be defined, for example, to notify the engine that active study is changed by user.
protected:	
CAM_DataModel* **createDataModel**();	This **optional** method should be re-implemented if the module has own data model which is different from default one (see 3.2.2 below).
SalomeApp_Selection* **createSelection**() const;	This method takes place in the context popup menus definition process (via popup menu manager). The goal of this method is to analyze currently selected objects and create an instance of `QtxPopupMgr::Selection` class (or its successor) to define rules which are taken into account when popup manager collects the menu commands for the popup menu being created. The rules for each command are usually created in the **initialize()** method (refer to 3.2.4 for details).
public slots:	
bool **activateModule**(SUIT_Study*);	This method is called each time when the module is activated by the application. This method can activate menus, toolbars created previously by **initialize**() method. In addition, it is the right place to set custom selectors to the Object browser or viewers, etc.
bool **deactivateModule**(SUIT_Study*);	This method is called each time when the module is deactivated by the application (e.g. when user activates another module). It should deactivate menus and toolbars shown by **activate**() method. It should also disable its own selectors and enable other ones which were active before the module was activated.

Note: when overriding virtual methods do not forget to call base implementation (except the cases when it is really necessary to hide default implemetation).

3.2.2. Data model

Data model classes are responsible for the transient presentation of the module's data. It means data internal organization and the way how the data should be presented in GUI (in the Object browser, viewer windows, etc.). By default SALOME-based modules use the data model from **SalomeApp** package. This data model is based on `SALOMEDS_Study` and represents the structure of the study basing on `SALOMEDS_SObject` objects.
If the module does not use default SALOMEDS-based data model, it is necessary to create it's own data model and export it in the `createDataModel()` method of the successor of the `SalomeApp_Module` class.

The data model classes are defined in **SalomeApp** package. Custom module may inherit these classes and override the default behavior when necessary. In some cases it can be reasonable to use the corresponding classes from CAM package (refer to **LIGHT** sample module for example).

SalomeApp_DataModel	Data model itself, it is responsible for the creating, loading, saving and updating of the internal data structure.
SalomeApp_DataObject	This class presents the elementary unit of the data model. It includes presentation methods like **name**(), **icon**(), **toolTip**(), etc.
SalomeApp_ModuleObject	This is the root object of the data model.

Note that `SalomeApp_ModuleObject` class uses virtual inheritance, because it inherits the `SUIT_DataObject` class through the different branches.

3.2.3. Selection handling

Access to the selection is provided via `SalomeApp_SelectionMgr` class. It allows getting of the list of the currently selected objects including sub-objects for the complex objects (like nodes of mesh, etc), modifying of the selection (by adding/removing objects to the selection), clearing the selection, etc. The class `SalomeApp_DataOwner` class together with `SALOME_InteractiveObject` provides the way of identifying data objects when they are involved in the selection mechanism.
The data objects in SALOME GUI are identified by their entries in the `SALOMEDS_Study` (as it was in previous versions of SALOME).

If the module has own data model it also probably needs own data objects identification mechanism. In this case the data owner class should be implemented (refer to **LIGHT** sample module for the example).
Moreover, in this case module may need to define its own selector class(es) (`SUIT_Selector` class successors). When module is activated it should install own selectors to the object browser, viewer windows and disable other previously installed selectors. These own selectors will then control all the selection operations and notify the selection manager when the selection is changed. And when the module is deactivated it is necessary to deactivate own selectors and activate again previous ones.

3.2.4. Menus, toolbars and context menus management

There is no support for XML-based menu definition files in SALOME 3 GUI as it was before. Instead all the GUI actions should be hard-coded in the module GUI implementation.
The usual place where to do it is the successor of the `SalomeApp_Module` class.
This class defines a family of `createAction()`, `createMenu()` and `createTool()` methods in order to create actions and associate them with the main menu, toolbars and popup menus. This can be done in the `initialize()` method of the module.

The created menus and toolbars are not shown immediately after creation. To do it `activate()` method of the module class should call `setMenuShown( true )` and `setToolShown( true )` methods.
And vice versa, `deactivate()` method should hide the menus and toolbars by calling `setMenuShown( false )` and `setToolShown( false ).`

Note: for the internationalization of the menus, the corresponding *.po resource files should be used.

The processing of the context popup menus is possible by two ways. The simplest way is overriding of the `contextMenuPopup()` method, then filling in the popup menu with commands manually according to the current selection and menu context (object browser, viewer, etc.) and then processing of the chosen action in the corresponding slot.

Another way is the using of the popup menu management mechanism. First it is necessary in the `initialize()` method of the module to create all actions which should be presented in the context popup menu, then push all these actions to the popup manager and define the lexical rule for each action. Each time when the popup menu is requested these rules define, should some action appear in the menu or not and should it be checked on or off (for switching actions).
Then it is necessary to override `createSelection()` method which should create the instance of `SalomeApp_Selection` class (or its successor). This class analyzes the current selection and defines some variables which are engaged in the lexical rules definition.
For example,

```
void GeometryGUI::initialize( CAM_Application* app )
{
  …
  // create “Erase All” action
  createGeomAction( 214, “ERASE_ALL” );
  // get the popup menu manager
  QtxPopupMgr* mgr = popupMgr();
  // push the action to the popup manager
  mgr->insert( action(  214 ), -1, -1 );
  // define the rule for the action
  mgr->setRule( action( 214 ), "isActiveViewer=true \\
                and $client in {'OCCViewer' 'VTKViewer'}", true );
  …
}
```

In the above example the “Erase All” action will appear in the popup menu only if there is active viewer and context popup menu is requested for the OCC or VTK viewer. The <isActiveViewer> and <$client> variables are defined by the `GEOMGUI_Selection` class methods:

```
void GEOMGUI_Selection::init( const QString& client,
                              SalomeApp_SelectionMgr* mgr )
{
  …
  // remember the client
  myPopupClient = client;
  …
}

QtxValue GEOMGUI_Selection::param( const int ind, const QString& p ) const
{
  QtxValue val( SalomeApp_Selection::param( ind, p ) );
  if ( !val.isValid() ) {
    …
    // the “client” parameter is requested
    if ( p=="client" ) return myPopupClient;
    // the “isActiveViewer” parameter is requested
    if ( p == "isActiveViewer" ) return QtxValue( isActiveViewer() );
    …
  }
  …
}
```

It is possible to use `SalomeApp_Selection` class instance – it defines a lot of helpful variables, but if its functionality is not enough for your needs you can implement own class inheriting from `SalomeApp_Selection` or from `QtxPopupMgr::Selection`.

The second way seems some more difficult to implement but in this case you will not need to re-analyze all code responsible for the popup menu creation each time when you add some new action or introduce new selection rules.

3.2.5. Resources and preferences handling

SALOME application provides unified access to resources and preferences via Resource manager.
It is initialized when the application is started and automatically loads resource files for all modules and packagese mentioned in the “resources” section of the configuration file.
There are two configuration files, both defined in XML format. The first one is SalomeApp.xml file, called “global” and situated in the ${GUI_ROOT_DIR}/share/salome/resources folder. The user can not change this file – it defines the default settings of the application. The user configuration file named .SalomeApprc.3.x.x (where 3.x.x is a SALOME version number, e.g. 3.0.0) is created during the first application launch in the user’s home directory.

All modules and packages which need some resource files to be loaded by the application should be mentioned in the “resources” section of the configuration file(s). For example:

```
…
<section name="resources">
    <parameter name="SUIT" value="${SUITRoot}/resources"/>
    <parameter name="STD" value="${SUITRoot}/resources"/>
    <parameter name="GEOM" value="${GEOM_ROOT_DIR}/share/salome/resources"/>
    …
</section>
…
```

The value of each parameter defines the directory where Resource manager should look for resource files of some module. It is possible to use any environment variables to define search path. Resource files are the *.qm Qt internationalization files (compiled from *.po files by msg2qm tool), image files (*.png, *.bmp, etc) and other user-defined files. By default Resource manager looks for <MODULE>_msg_<LANG>.qm, <MODULE>_icons.qm and <MODULE>_images.qm files, where <MODULE> is a module name and <LANG> is a used language (defined in "language" section of the configuration file).

The access to the Resource manager is provided by the `SalomeApp_Application` class:

```
SUIT_ResourceMgr*  resourceMgr() const;
```

To create the pixmap from the icon file use `loadPixmap()` method of the Resource manager:

```
QPixmap loadPixmap( const QString&, const QString& ) const;
```

To find some other resource file use `path()` method:

```
QString path( const QString&, const QString&, const QString& ) const;
```

Sometimes might be necessary to load some internationalization files manually. It can be done by calling `loadTranslator()` method:

```
void loadTranslator( const QString&, const QString& );
```

Access to the user preferences is also provided via the Resource manager. The preferences are automatically loaded when application is started and saved when application is closed normally (i.e. not killed by killSalome.py script). Each user preference is defined by its section and name. The Resource manager provides a lot of methods to read/write preferences:

```
bool    hasSection( const QString& ) const;
bool    hasValue( const QString&, const QString& ) const;

bool    value( const QString&, const QString&, int& ) const;
bool    value( const QString&, const QString&, double& ) const;
bool    value( const QString&, const QString&, bool& ) const;
bool    value( const QString&, const QString&, QColor& ) const;
bool    value( const QString&, const QString&, QString&, const bool = true ) const;
int     integerValue( const QString&, const QString&, const int = 0 ) const;
double  doubleValue( const QString&, const QString&, const double = 0 ) const;
bool    booleanValue( const QString&, const QString&, const bool = false ) const;
QColor  colorValue(const QString&, const QString&, const QColor& = QColor()) const;
QString stringValue( const QString&, const QString&, const char* = 0 ) const;

void    setValue( const QString&, const QString&, const int );
void    setValue( const QString&, const QString&, const double );
void    setValue( const QString&, const QString&, const bool );
void    setValue( const QString&, const QString&, const QColor& );
void    setValue( const QString&, const QString&, const QString& );

void    remove( const QString&, const QString& );
void    removeSection( const QString& );
```

The first parameter of all these methods is a resource section name, and second is a name of the setting.

In addition SALOME 3 GUI provides the common preferences edition dialog box. This dialog box has the separated tab page for each module. The module which wants to export some preferences to the common dialog may redefine createPreferences() method and then use addPreference() and setPreferenceProperty() methods:

```
void GeometryGUI::createPreferences()
{
  int tabId = addPreference( tr( "PREF_TAB_SETTINGS" ) );

  int genGroup = addPreference( tr( "PREF_GROUP_GENERAL" ), tabId );
  addPreference( tr( "PREF_SHADING_COLOR" ), genGroup,
                 SalomeApp_Preferences::Color, "Geometry", "shading_color" );
  int step = addPreference( tr( "PREF_STEP_VALUE" ), genGroup,
                 SalomeApp_Preferences::IntSpin, "Geometry", "SettingsGeomStep" );

  setPreferenceProperty( genGroup, "columns", 1 );

  setPreferenceProperty( step, "min", 0.001 );
  setPreferenceProperty( step, "max", 10000 );
  setPreferenceProperty( step, "precision", 3 );
}
```

This dialog box allows editing of most used types of settings, like strings, color values, integer and floating point values, boolean values, etc. Refer to `SalomeApp_Preferences` class for more details.

3.2.6. View management

To operate with viewers the mechanism of view managers is provided by the SALOME 3 GUI.
To get the view manager use `getViewManager()` method of `SalomeApp_Application` class:

```
SUIT_ViewManager* getViewManager( const QString&, const bool );
```

The first parameter defines the type of the view manager being requested, e.g. “OCCViewer”, “VTKViewer” or “Plot2d”. The second parameter says if it is necessary to create new view manager if there is no yet appropriate one. The new view window is created automatically when necessary.

To create new view window use its `createView()` method. To get active view, use `getActiveView()`.

Active view manager can be also get from the `SalomeApp_Application` by `activeViewManager()` method.

3.2.7. Object Browser, python console, etc.

The access to the Object browser is provided via `SalomeApp_Application` class:

```
OB_Browser* objectBrowser();
```

You unlekely need to have direct access to the Object Browser if you do not implement your own selector class.
To get the selected objects from the Object browser use `getSelected()` methods; to set the selection, use `setSelected()` methods.
To get the root entry (in order to iterate through the children) you may use `getRootObject()` method, and to update its contents call `updateTree( SUIT_DataObject* )`.

In addition, the `SalomeApp_Application` class provides method `updateObjectBrowser( const bool = true )` to update the Object browser contents. The optional parameter says if it is necessary also to update data models for all loaded modules.

The access to the embedded Python console is possible via `SalomeApp_Application` class:

```
PythonConsole* pythonConsole();
```

To execute a Python command in the embedded Python console, use its `exec(const QString&)` method.

To send the text message to the Log output window use logWindow() method of the `SalomeApp_Application` class:

```
LogWindow*  log = logWindow();
log->putMessage( “simple message” );
```

3.3. Launching

To launch SALOME with custom module it is necessary to define environment variable <MODULE>_ROOT_DIR to point to you module’s binaries distribution and then modify configuration file (see 3.2.5 above for more details about Resource manager). Here and below <MODULE> is a module symbolic name, e.g. for Geometry module’s symbolic name is GEOM.

```
…
  <section name="launch">
    <parameter name="modules" value="MODULE"/>
  </section>
  <section name="resources">
    <parameter name="MODULE" value="${MODULE_ROOT_DIR}/share/salome/resources"/>
  </section>
  <section name="MODULE">
    <parameter name="name" value="Module"/>
    <parameter name="icon" value="Module.png"/>
    <parameter name="library" value="libModuleGUI.so"/>
  </section>
…
```

Section “launch” defines the modules which should be taken into account when launching SALOME application.
Section “resources” defines for each module the directory or list of directories where SALOME application should look for resource files. Usually it is a module’s share/salome/resources folder.
The module configuration section may contain any module-specific preferences, but several parameters in this section are obligatory:

- “name” parameter defines the module user name (that one displayed in the “Components” toolbar);
- “icon” parameter defines the icon for the module;
- **optional** “library” parameter defines the name of the library which should be used as GUI library for the module; by default **lib<MODULE>.so** file name is used.

4. PYTHON MODULES MIGRATING

In order to minimize expenses concerned with migrating of existing Python modules on new GUI architecture SALOME 3 tries to keep as much as possible the compatibility with SALOME 2.x Python API. This concern SalomePyQt, libSALOME_Swig and SalomePy libraries which are used from the python code to access SALOME GUI functionality, and SalomePyQtGUI library which represents a base GUI library for all Python modules.

This paragraph refers to the pure Python modules, i.e. which contain no C++ code. Modules which include both C++ and Python code may need to perform some migrating steps which are descried in the previous section of this document.

4.1. Python API

As it is mentioned above, the Python API of SALOME GUI (wrapped by SIP and SWIG tools) is fully compatible with the previous versions of SALOME. The functions which became obsolete in the new GUI are still supported. This significantly reduces the migrating expenses for the Python modules.

The new Python modules should use the new API included in the SalomePyQt, libSALOME_Swig and SalomePy libraries, but old ones still may use old API with some reserves.

4.2. Configuration and building

Minimum modifications are needed to make the Python module package to be built with new GUI.

4.2.1. msg2qm

For *.po resource files to be compiled it is necessary to add check step to the configure script (see 3.1.1).

4.2.2. Menu resources

As it was mentioned in 3.2.4 menu resource files (XML-based) are not supported in SALOME 3 GUI. All menu/toolbars actions should be hard-coded in the module sources. The only exception to this rule is made to the Python modules. The support of XML-based menu resource files is kept as before. The SalomePyQtGUI library automatically tries to find menu definition file and parses it to create main menus, toolbars and popup menus. The format of menu definition files is not changed too.

Note: new Python modules should avoid using of XML-defined menus, because this way of menu definition is obsolete and can be removed in future versions of SALOME. New Python API should be used instead in order to create menu, toolbars, etc.

Some modules can use direct access to main menu via SalomePyQt module:

```
static QMenuBar*   getMainMenuBar();
static QPopupMenu* getPopupMenu( const MenuName );
```

Such modules add menu items to the main menu manually, usually by using `insertItem()` functions. Since application does not clear automatically all “alien” menu items the module is obliged to remove these menu items by its own. Note that if XML-based files are used for creation of menus the clearing is performed automatically.

4.3. Launching

The paragraph 3.3 describes the actions which should be done in order to launch SALOME with custom module(s). The same concerns the pure Python modules. The “library” parameter of the module preferences section should contain “libSalomePyQtGUI.so” value. It means that for the Python modules which do not have own GUI libraries libSalomePyQtGUI.so should be used.

4.4. Caveats

4.4.1. Multi-desktop environment

Since SALOME 3 GUI introduces multi-desktop interface, Python module GUI should take it into attention. It means that if any desktop-referenced variable is used in the code this variable should be reinitialized in `setSetting()` and `activeStudyChanged()` methods. It is recommended to re-get desktop each time when it is needed by calling of the corresponding SALOME Python API methods.

4.4.2. Workspace

SALOME 3 GUI uses tabbed widget to stack all viewer windows. Moreover, SUIT library provides different types of desktop, and the default desktop type can be changed in future or even become customizable. So, the Python module GUI should not rely on the value passed by `setWorkspace(ws)` method – it may have `0` (zero) value. This method seems not to have significant utility. It is considered as obsolete.

APPENDIX

Check SALOME GUI procedure

```
AC_DEFUN([CHECK_SALOME_GUI],[

AC_CHECKING(for Salome GUI)

SalomeGUI_ok=no

AC_ARG_WITH(gui,
          --with-salome_gui=DIR   root   directory   path   of   SALOME   GUI
installation,
          SALOME_GUI_DIR="$withval",SALOME_GUI_DIR="")

if test "x$SALOME_GUI_DIR" = "x" ; then
  if test "x$GUI_ROOT_DIR" != "x" ; then
    SALOME_GUI_DIR=$GUI_ROOT_DIR
  else
    # search Salome binaries in PATH variable
    AC_PATH_PROG(TEMP, SUITApp)
    if test "x$TEMP" != "x" ; then
      SALOME_GUI_BIN_DIR=`dirname $TEMP`
      SALOME_GUI_DIR=`dirname $SALOME_GUI_BIN_DIR`
    fi
  fi
fi

if test "x$SALOME_GUI_DIR" != "x" ; then
  if test -f ${SALOME_GUI_DIR}/bin/salome/SUITApp  ; then
    SalomeGUI_ok=yes
    AC_MSG_RESULT(Using SALOME GUI distribution in ${SALOME_GUI_DIR})
    GUI_ROOT_DIR=${SALOME_GUI_DIR}
  fi
fi
if test "x$SalomeGUI_ok" == "xno" ; then
  AC_MSG_WARN("Cannot find compiled SALOME GUI distribution")
fi

AC_SUBST(GUI_ROOT_DIR)

AC_MSG_RESULT(for SALOME GUI: $SalomeGUI_ok)

])dnl
```