# l'architecture à plug-ins de SALOMÉ TMF

Mikaël Marche

26 avril 2006

# Table des matières

# Table des figures

# Chapitre 1

# Framework "JPF" (Java Plugin Framework)

Ce chapitre présente le framework JPF (Java Plugin Framework) sur lequel s'appuie l'architecture à plugins de Salomé TMF.

## 1.1 Introduction

JPF est le fruit d'un projet Open Source, sous licence LGPL, portant le même nom et accessible dans le portail Source Forge sous le lien http ://jpf.sourceforge.net.

## 1.2 Architecture

Le schéma en Figure 1.1 présente l'architecture générale du framework JPF.

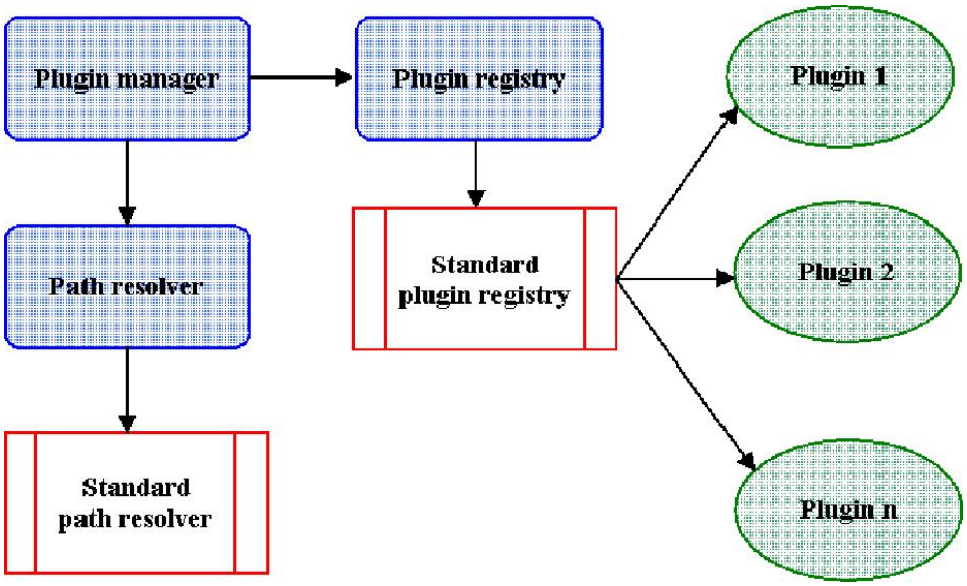



FIG. 1.1 – Architecture du framework JPF

- **Plugin registry** : ensemble d'interfaces qui enregistrent les plugins présents à partir des méta-information présentes dans le fichier manifest (XML) de chaque plugin. JPF en propose une implémentation standard (standard plugin registry).
- **Path resolver** : composant qui répertorie pour chaque plugin la localisation physique de ses ressources. JPF en propose également une implémentation standard (standard path resolver).
- **Plugin manager** : classe principale du framework qui charge et active les plugins. Une instance "standard" du plugin manager peut être créée en utilisant les implémentations standards du plugin regitry et du path resolver

# Chapitre 2

# Architecture à plugins de Salomé TMF

Ce chapitre présente l'architecture à plugins de Salomé TMF suivant le framework JPF.

## 2.1 Initialisation de JPF

Cette initialisation est effectuée par la méthode *"startJPF"* de la classe JPFManager se trouvant dans le package *org.objectweb.salome_tmf.plugins*. Afin d'initialiser les paramètres nécessaires à JFP, un fichier "properties" est utilisé. Ce fichier est nommé *"CfgPlugins.properties"* et se trouve dans le sous répertoire "plugins" du répertoire d'installation de Salomé TMF. Il contient le nom du répertoire qui abrite les plugins, la liste des plugins présents ainsi que d'autres paramètres utilisés par le système de trace (*log4j*).

Une fois ces paramètres initialisés, on crée une table de hachage qui contient autant d'éléments que de plugins, chaque élément ayant la forme suivante :

*Clé* : fichier manifest → *Valeur* : URL du répertoire contenant les ressources du plugin.

Cette table de hachage permet alors de créer un plugin manager comme le montre le code suivant :

```
// Pamameters for PluginManager
StringTokenizer PluginsFolders = new StringTokenizer(
                                        props.getProperty(PARAM_PLUGINS_FOLDERS), ",", false);

Map pluginLocations = new HashMap();
for (; PluginsFolders.hasMoreTokens();) {
        String currentPlgFolder = appliRoot + PluginsFolders.nextToken().trim();
        StringTokenizer PluginsList = new StringTokenizer(
                                        props.getProperty(PARAM_PLUGINS_LIST), ",", false);
        for (; PluginsList.hasMoreTokens();) {
                String currentPlg =  currentPlgFolder + "/" + PluginsList.nextToken().trim();
                try {
                        pluginLocations.put(new URL(currentPlg + "/plugin.xml"), new URL(currentPlg));
                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

```
// New instance of PluginManager
PluginManager pluginManager = PluginManager.createStandardManager(pluginLocations);
```

## 2.2 Le plugin *"core"* et les points d'extension

Le plugin *core* est un plugin particulier. En effet ; c'est un plugin qui doit être obligatoirement présent, et qui est activé par défaut.

### 2.2.1 Fichier manifest du plugin *core*

Chaque plugin possède un fichier manifest. C'est un fichier XML qui respecte la DTD spécifiée dans le framework JPF. Pour le plugin *"core"*, Ce fichier est composé de trois parties. La première partie est commune à tous les plugins et se présente de la façon suivante :

```
<plugin id="core" version="0.0.1" class="org.objectweb.salome_tmf.plugins.core.CorePlugin">
```

Cette entête spécifie l'identificateur du plugin, sa version ainsi que sa classe principale. La deuxième partie du fichier manifest est sous la forme suivante :

```
<runtime>
        <library id="core" path="core/core.jar" type="code">
                <export prefix="*"/>
        </library>
</runtime>
```

Cette partie spécifie que le plugin utilise une librairie *(core.jar)* qui se trouve dans le répertoire *"core"* (lui-même se trouvant dans le répertoire qui contient les plugins), et que cette librairie est du type *"code"*. Il est également précisé dans cette partie que toutes les classes et packages (*) de ce plugin sont visibles pour les autres plugins qui peuvent par conséquent les utiliser.

La troisième et dernière partie de ce fichier est celle qui est la plus intéressante car elle offre à l'application la possibilité d'être extensible. Il s'agit de la définition des points d'extension du plugin core, c'est-à-dire la manière avec laquelle d'autres plugins peuvent étendre celui-ci. Il n'y pas de restriction sur le nombre de points d'extension pour un plugin.

Cette partie se présente ainsi :

```
 <extension-point id="Common">
        <parameter-def  id="class"/>
        <parameter-def  id="name"/>
        <parameter-def  id="description" multiplicity="none-or-one"/>
</extension-point>
```

Ici, un point d'extension nommé *"Common"* est défini. Il est également précisé que tout plugin qui est extension de ce point doit valoriser trois paramètres :

- Le paramètre *"class"* : c'est le nom de la classe principale du plugin.
- Le paramètre *"name"* : il s'agit du nom du plugin.
- Le paramètre *"description"* : c'est la description du plugin.

Comme pour les points d'extension, il n'y a pas de restriction sur le nombre de paramètres que l'on peut associer à un point d'extension.

### 2.2.2 Activation du plugin core

L'activation du plugin core est déclenchée après l'initialisation de JPF et la création d'une instance du plugin manager.

Ensuite, pour chaque point d'extension, une méthode implémentée dans la classe principale du plugin core (*org.objectweb.salome_tmf.plugins.core.CorePlugin*) permet de récupérer ce point d'extension sous forme d'un objet de type **"ExtensionPoint"**. Voici le code qui effectue ces tâches (source : méthode **"startJPF"** de la classe **org.objectweb.salome_tmf.plugins.JPFManager**) :

```
// Running the "core" plugin
Plugin corePlugin = pluginManager.getPlugin("core");
if (corePlugin == null) {
        throw new Exception("can't get plug-in core");
}
// Extension points initialization
ExtensionPoint Common = (ExtensionPoint)corePlugin.getClass().getMethod (
        "getCommonExtPoint", new Class[] {}).invoke(corePlugin, new Object[] {});
```

## 2.3 Le point d'extension "Common"

Pour chaque point d'extension défini dans le fichier manifest du plugin core, une interface portant le même nom lui est associée dans le package *org.objectweb.salome_tmf.plugins.core*. Parmi ces interfaces figure l'interface *Common*.

L'architecture à plugins de *Salomé TMF* propose ce point d'extension dans le but de fournir une interface générique qui peut être utilisée par la plupart des plugins (sauf dans le cas de plugins spécifiques ; plugins pour l'exécution automatique des tests par exemple).

### 2.3.1 L'interface "Common"

Tout plugin qui déclare dans son fichier manifest qu'il est une extension du point Common doit implémenter l'interface *org.objectweb.salome_tmf.plugins.core.Common* proposée dans Salomé TMF. Cette interface facilite l'accès pour un plugin aux composants graphiques de Salomé TMF, afin qu'il puisse y ajouter ses fonctionnalités.

Cette interface est divisée en deux parties. La première a pour objet les fonctionnalités du plugin qui seront mises dans les menus "Outils" pour les suites de test, les campagnes de test et la gestion de données. La capture d'écran en Figure 2.1 montre un exemple de fonctionnalités de plugins dans un de ces trois menus.

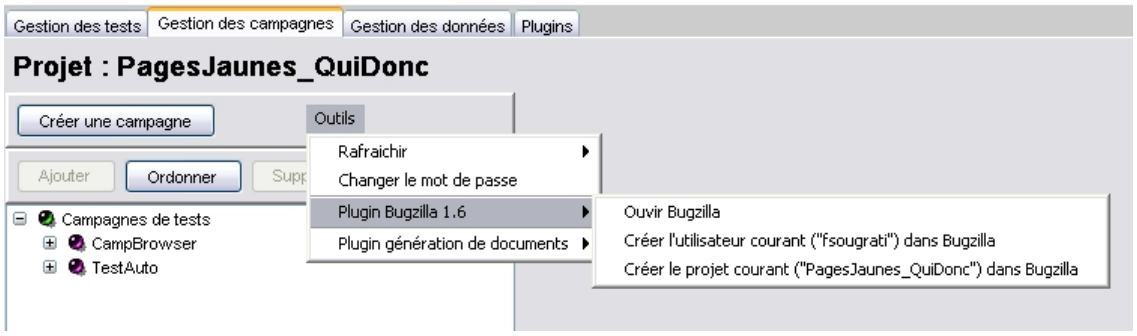



FIG. 2.1 – Menu outils des plug-ins

La deuxième partie de l'interface Common concerne le reste des composants graphiques de Salomé TMF utilisés par le plugin. En effet, le plugin doit spécifier les composants Salomé qu'il

utilise ou dans lesquels il rajoute des fonctionnalités, et dans le dernier cas il doit implémenter la manière avec laquelle ses fonctionnalités doivent être intégrées à Salomé.

Salomé TMF propose une multitude de ses composants graphiques exploitables par des plugins, c'est l'objet de la section 2.3.3.

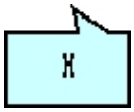
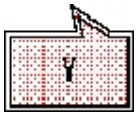
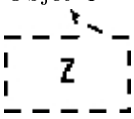
### 2.3.2 Utilisation des composants graphiques de Salomé TMF par les plug-ins

Si un plugin de type Common utilise des composants graphiques de Salomé TMF (autres que les menus outils), il doit implémenter la fonction **"getUsedUIComponents()"** de l'interface *Common* qui retourne la liste de ces composants. Afin que l'utilisation de ces composants graphiques soit transparente pour les plugins, les composants de Salomé TMF qui peuvent être utilisés par des plugins ou qui sont susceptibles d'abriter leurs fonctionnalités sont répertoriés.

A chacun de ces composants est associée une constante [1].Ainsi, le plugin fournit à l'application Salomé TMF la liste des constantes correspondant aux composants graphiques utilisés, et dans Salomé TMF on active le plugin dans ces objets graphiques selon leur type (statique ou dynamique) via les fonctions : **activatePluginInStaticComponent()** et **activatePluginInDynamicComponent()**.

### 2.3.3 Composants graphiques de Salomé-TMF

Les Figures de cet section dont la légende est donnée par le Tableau 2.1, illustrent les correspondances entre les objets graphiques de Salomé TMF et les constantes pour l'accès des plug-ins. des composants graphiques de Salomé TMF) :

| | |
|---|---|
|  | Composant graphiques utilisables par les plugins de type *Common* |
| Objet A   | La constante X correspond à l'objet graphique A qui est de type **statique** |
| Objet B   | La constante Y correspond à l'objet graphique B qui est de type **dynamique** |
| Objet C   | La constante Z correspond à l'objet graphique C<br>qui peut être de type **statique** ou **dynamique** selon les cas |

TAB. 2.1 – Légende

[1] voir la classe **org.objectweb.salome_tmf.ihm.UICompCst**

**Correspondances entre les objets graphiques de Salomé TMF et les constantes de plug-ins**

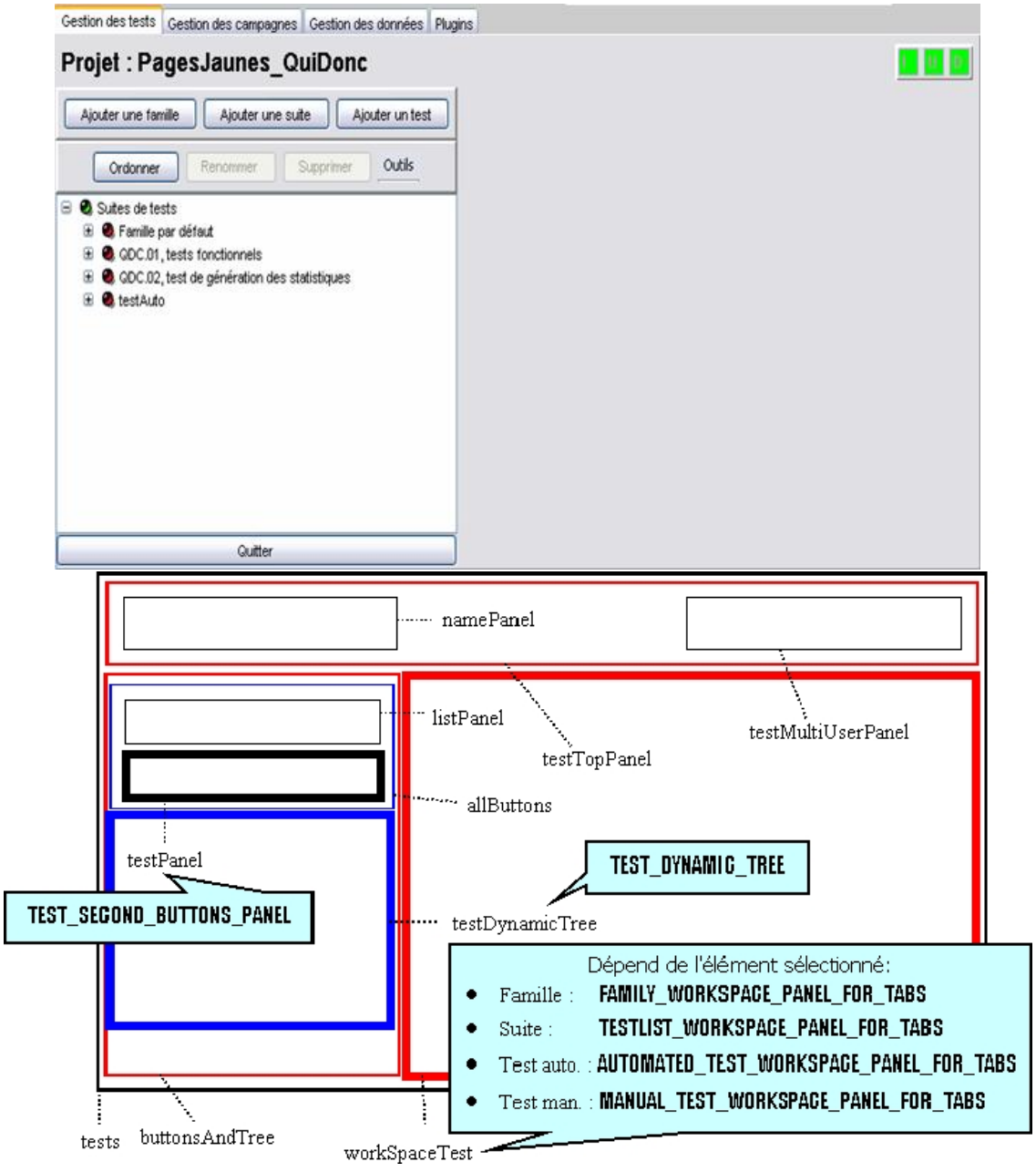



FIG. 2.2 – Onglet plan de test

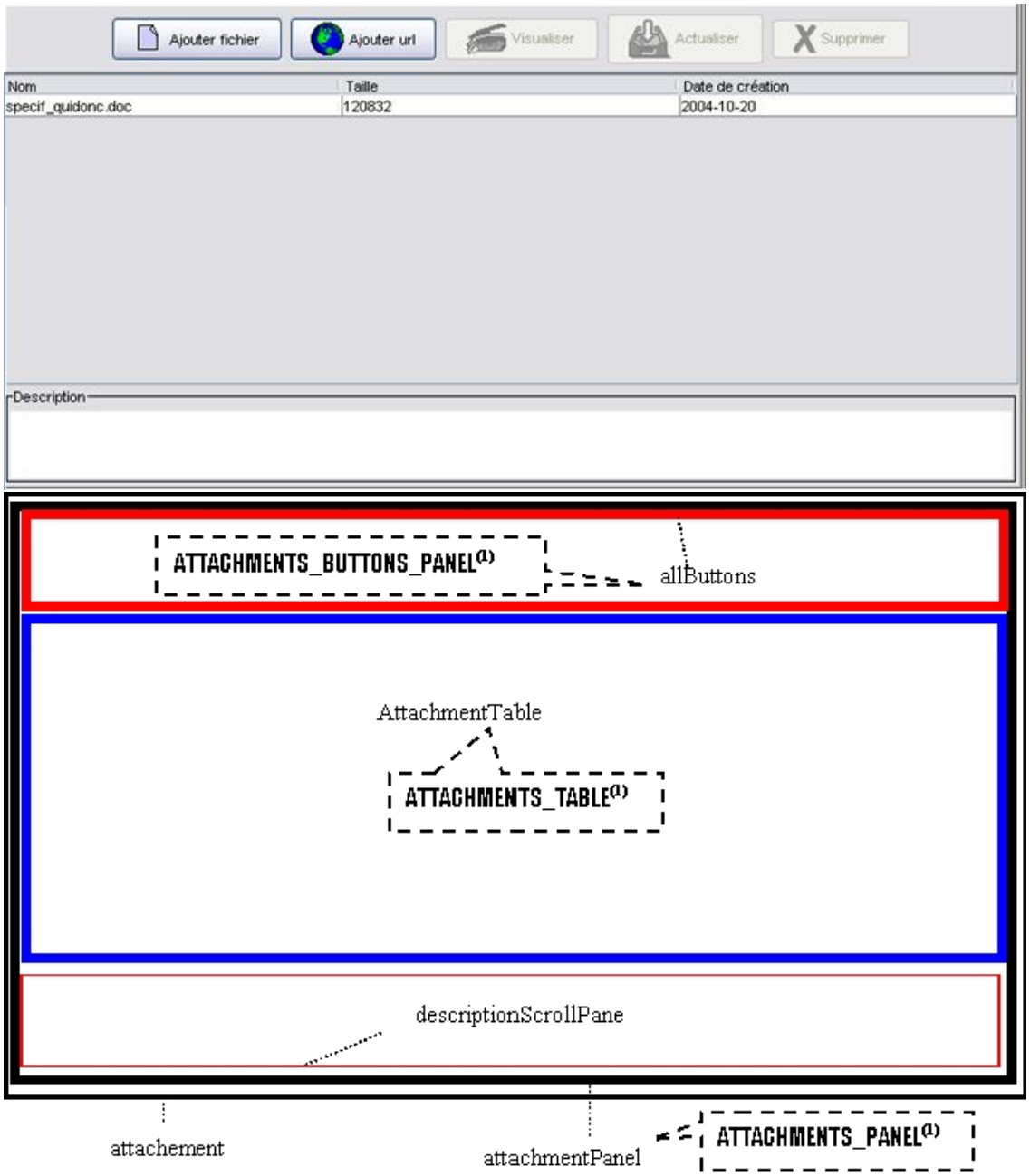



FIG. 2.3 – Vue "attachements" (Commune à tous les éléments pouvant avoir des attachements)

(1) Ces composants graphiques sont statiques lorsqu'il s'agit des attachements liés à une suite de test, un test ou une campagne. Ils sont dynamiques dans les autres cas.

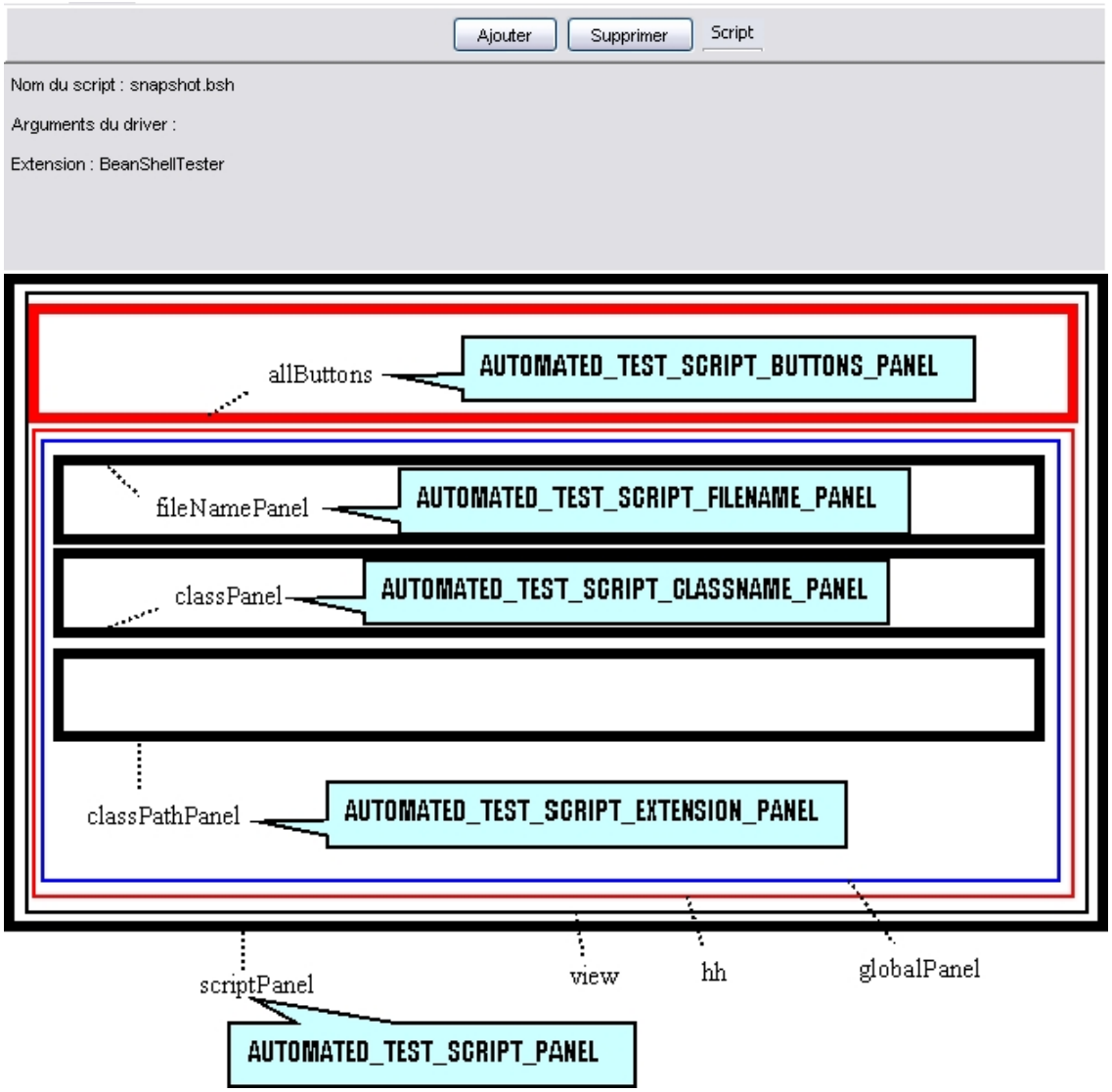



FIG. 2.4 – Vue "Script" pour un test automatique

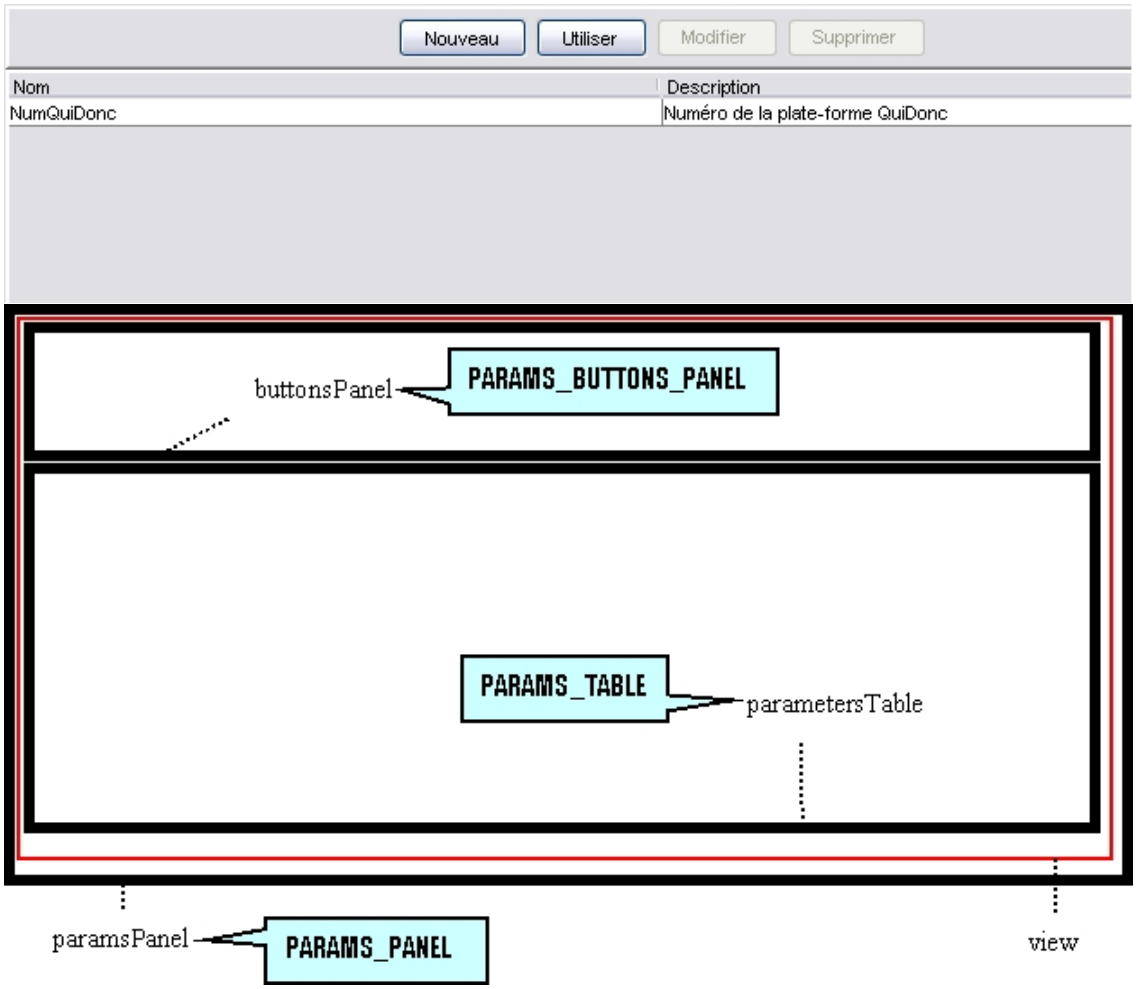



FIG. 2.5 – Vue "Paramètres" pour les tests ainsi que dans le panel de gestion des données

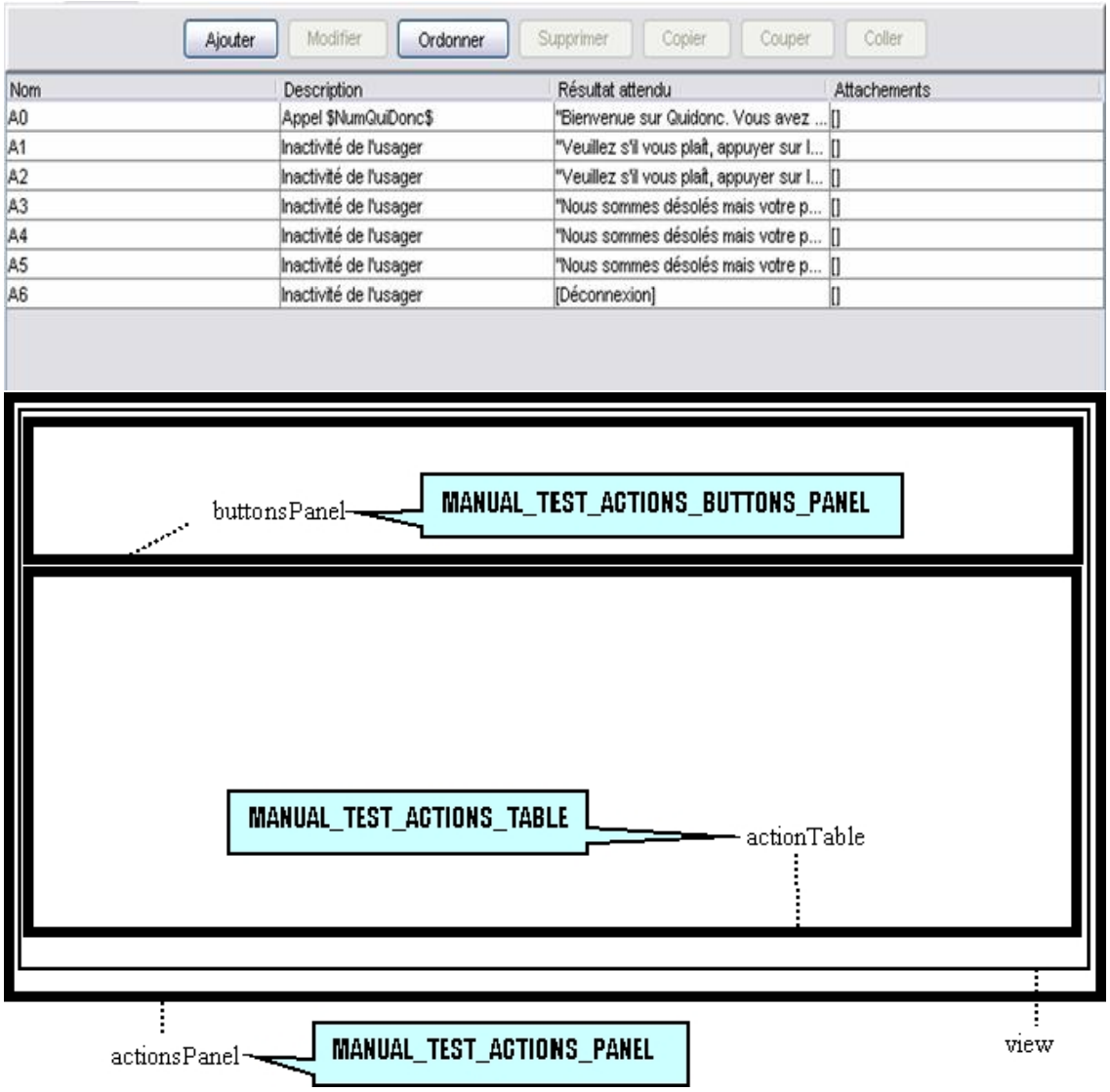



FIG. 2.6 – Vue "Actions" pour un test manuel

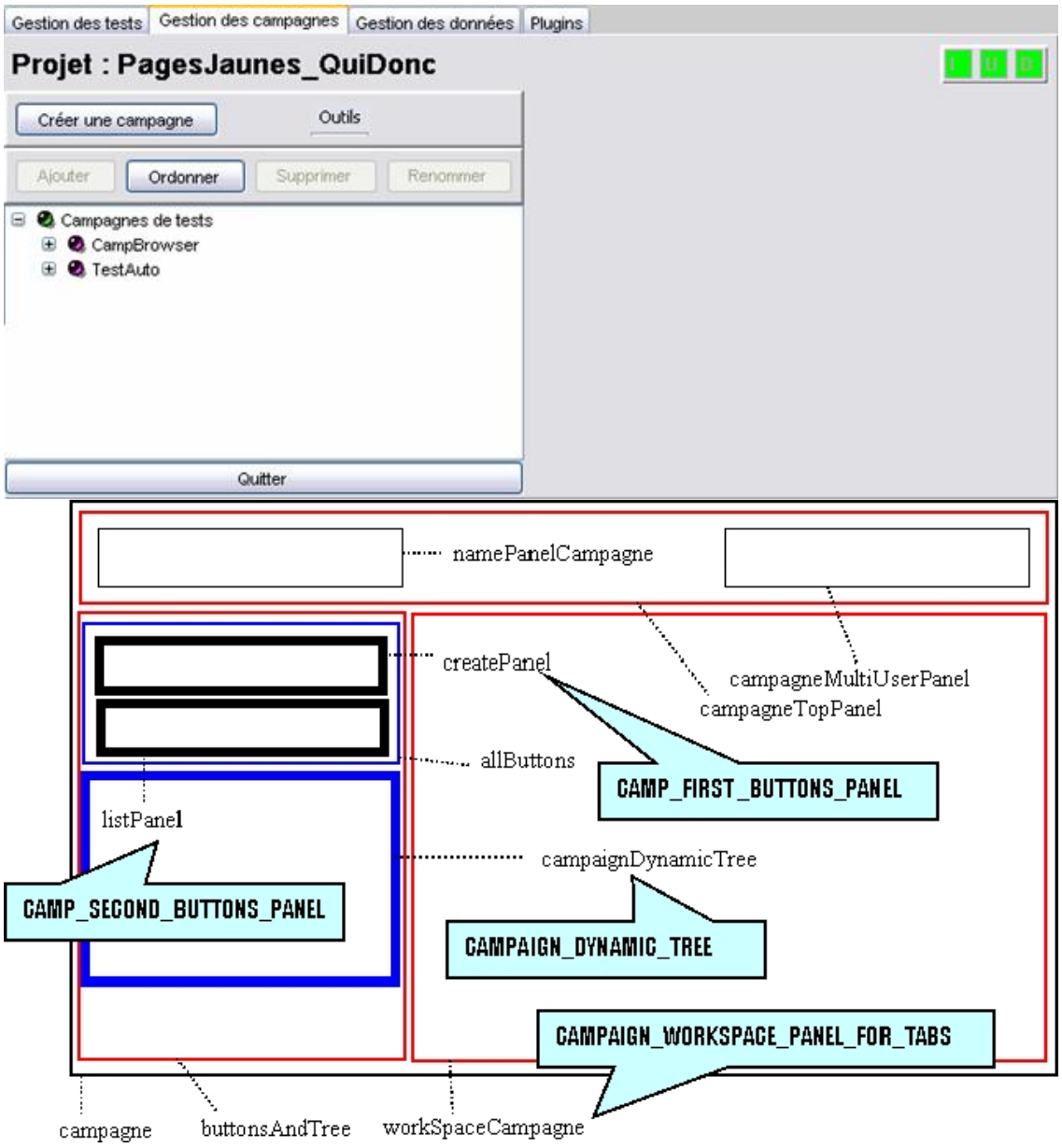



FIG. 2.7 – Onglet campagnes de tests

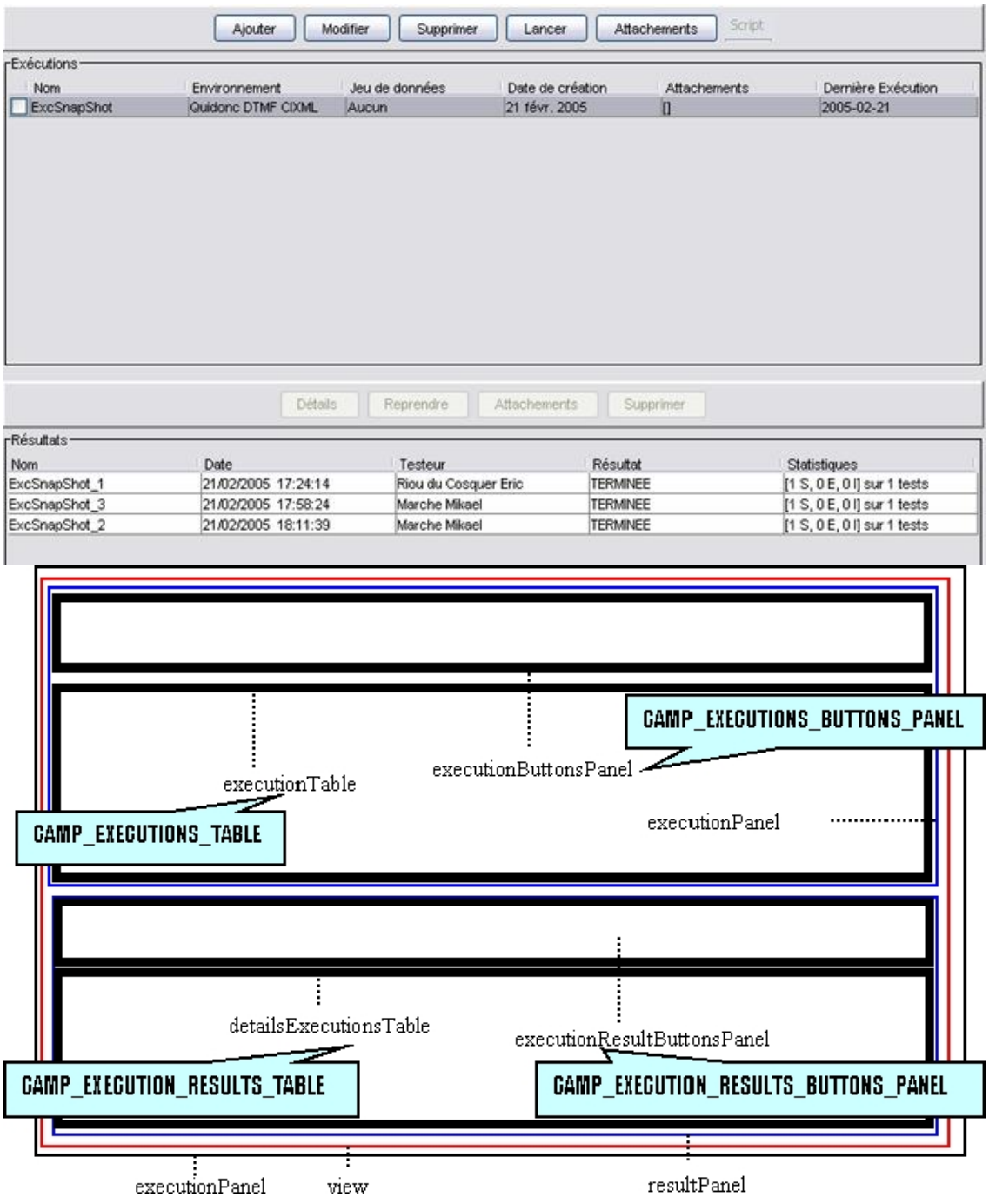



FIG. 2.8 – Vue "Exécutions" pour une campagne de test

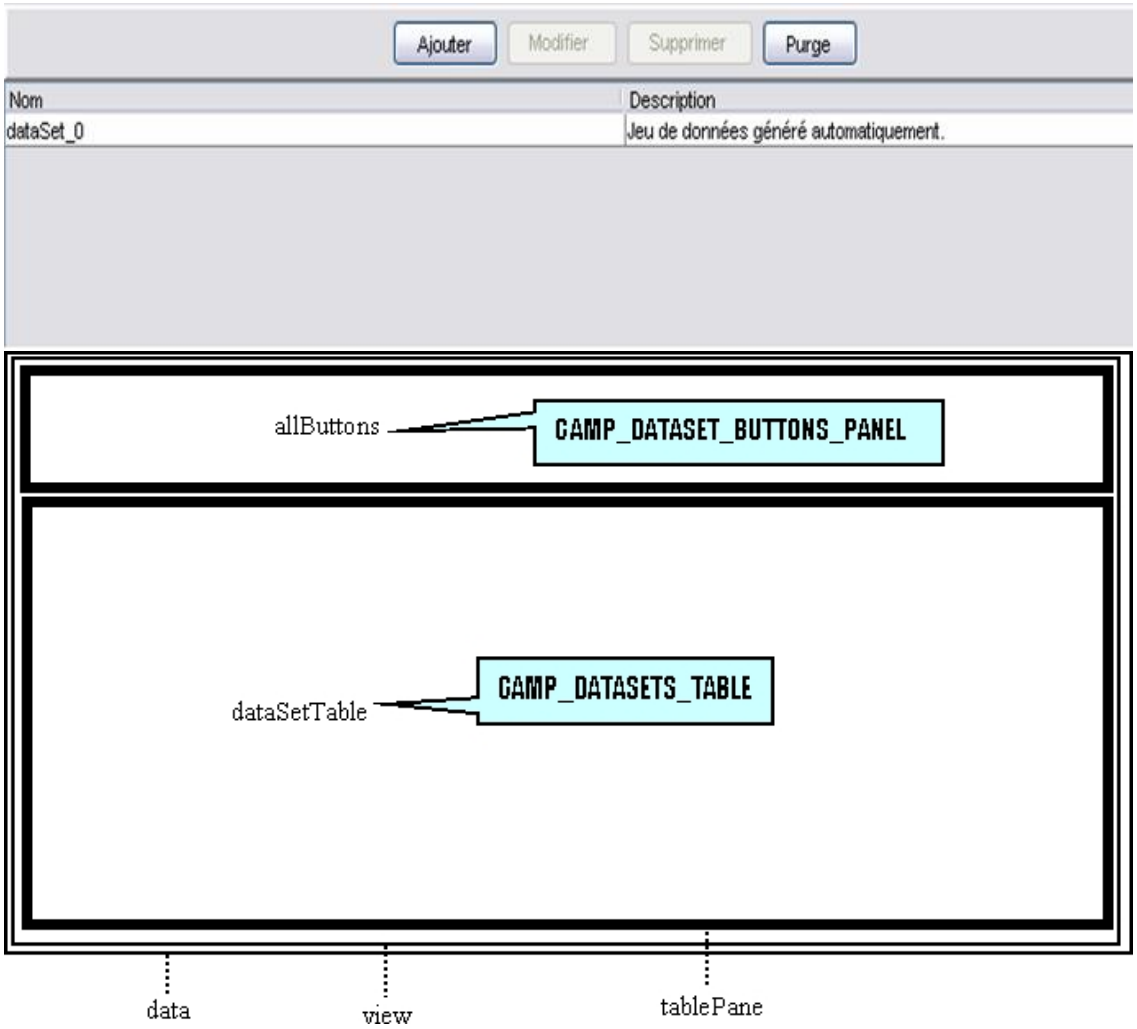



FIG. 2.9 – Vue "Jeux de données" pour une campagne de test

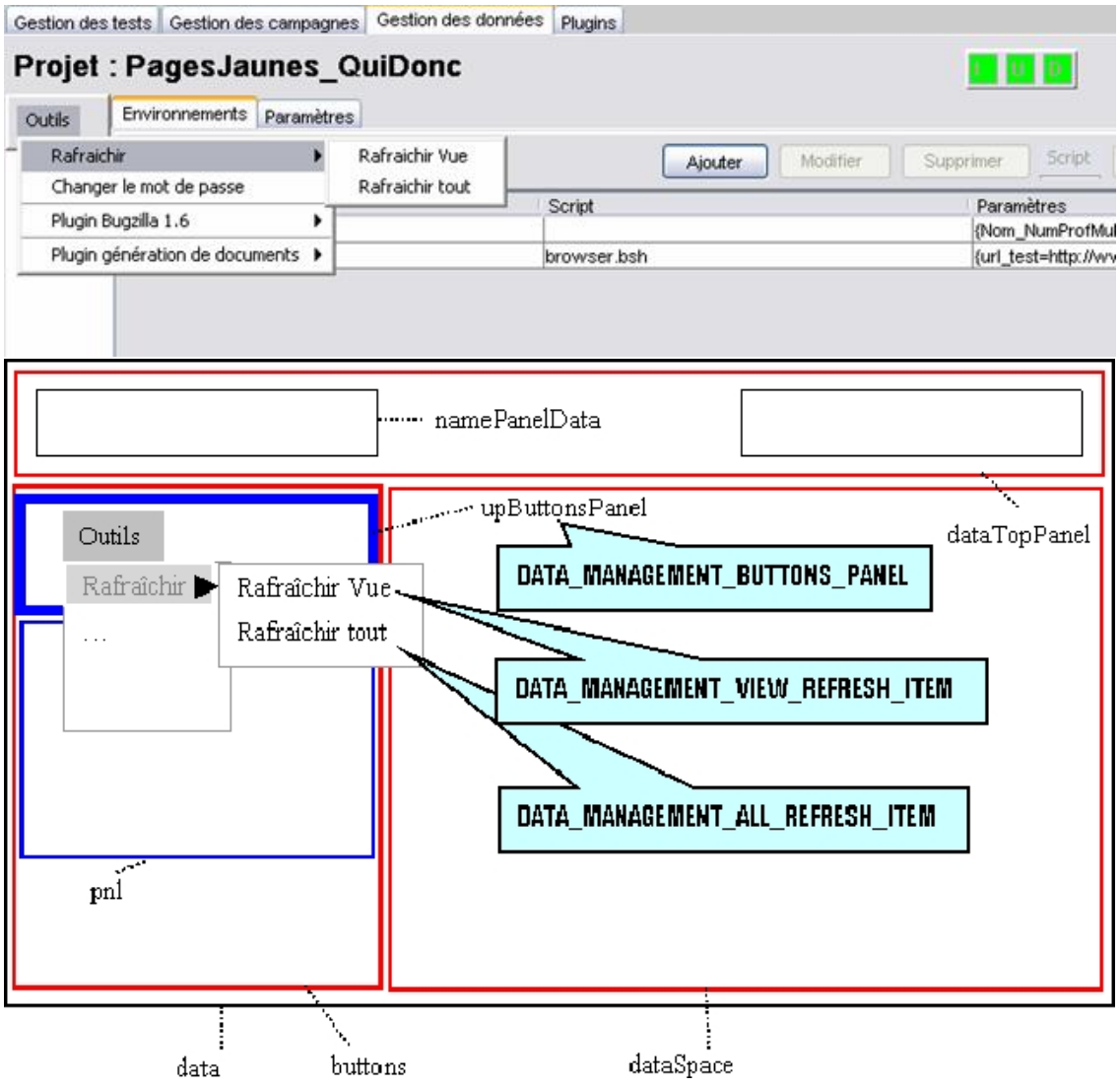



FIG. 2.10 – Fenêtre principale pour la gestion des données

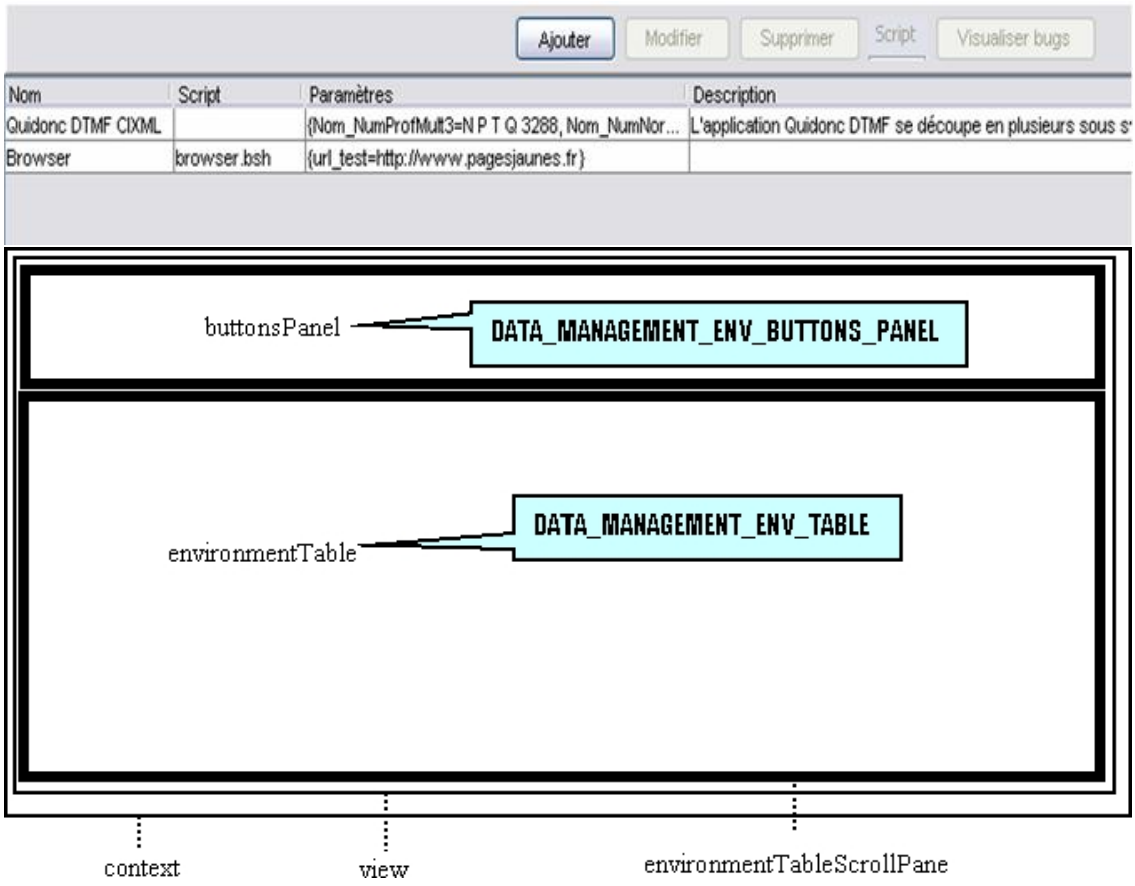



FIG. 2.11 – Vue "Environnements" dans le panel de gestion des données

NB : La vue "Paramètre" dans le panel de gestion des données est la même que la vue "Paramètres" pour les tests

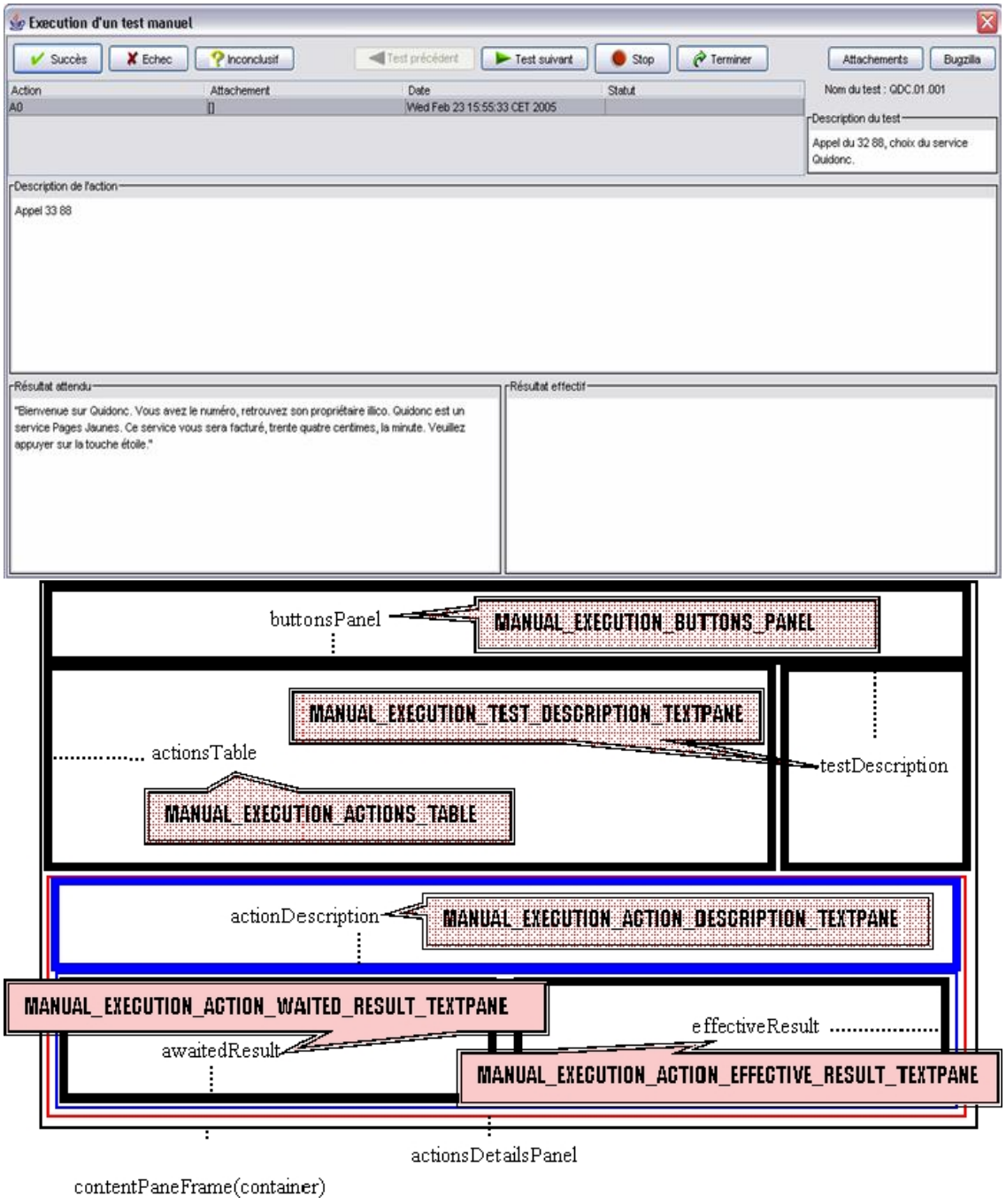



FIG. 2.12 – Fenêtre d'exécution d'un test manuel

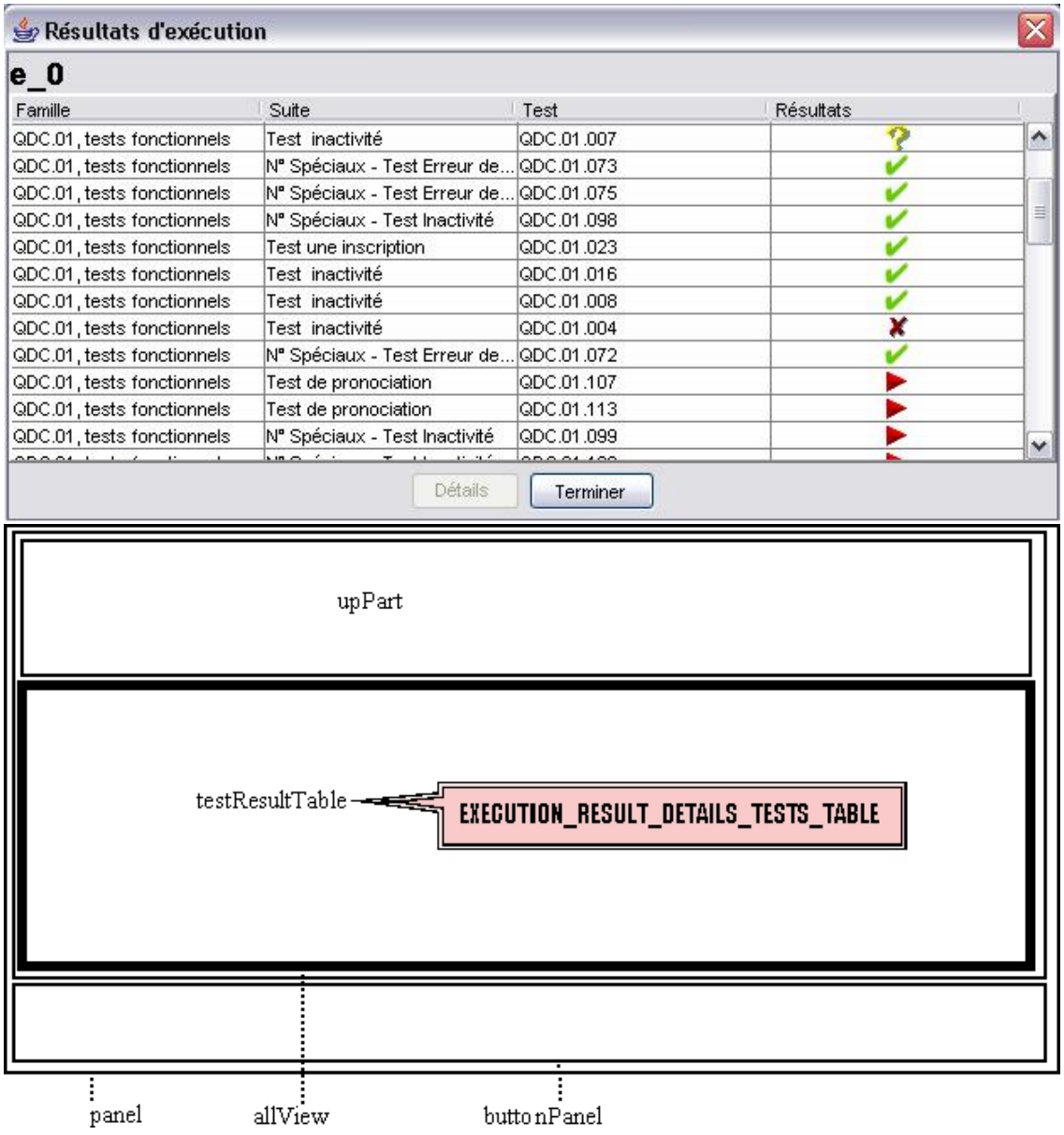



FIG. 2.13 – Fenêtre pour le détail d'un résultat d'exécution d'une campagne de test

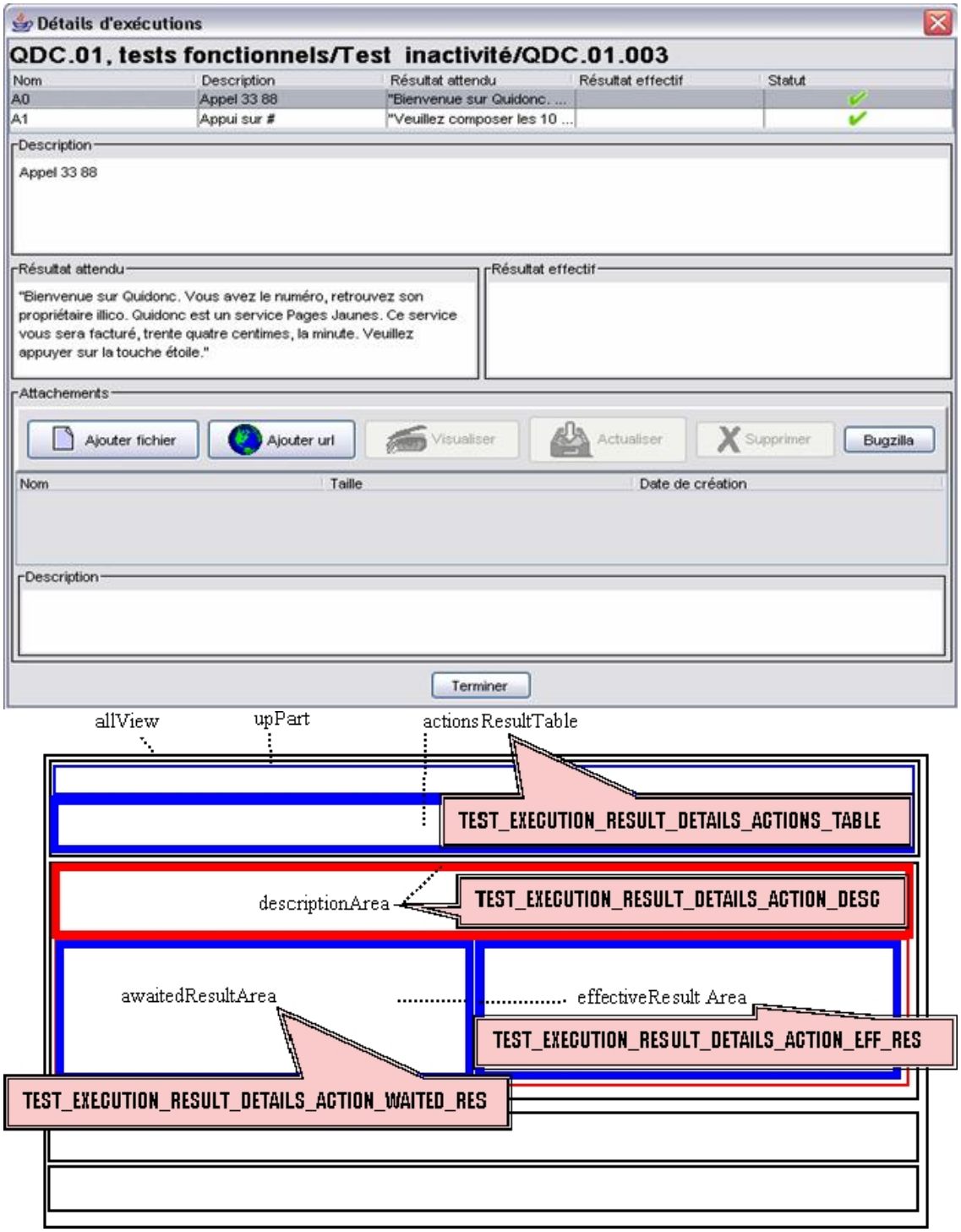



FIG. 2.14 – Fenêtre pour le détail d'un résultat d'exécution d'un test manuel

NB. : Les objets graphiques de la partie pour les attachements de la précédente fenêtre sont les mêmes que pour tous les éléments ayant des attachements.

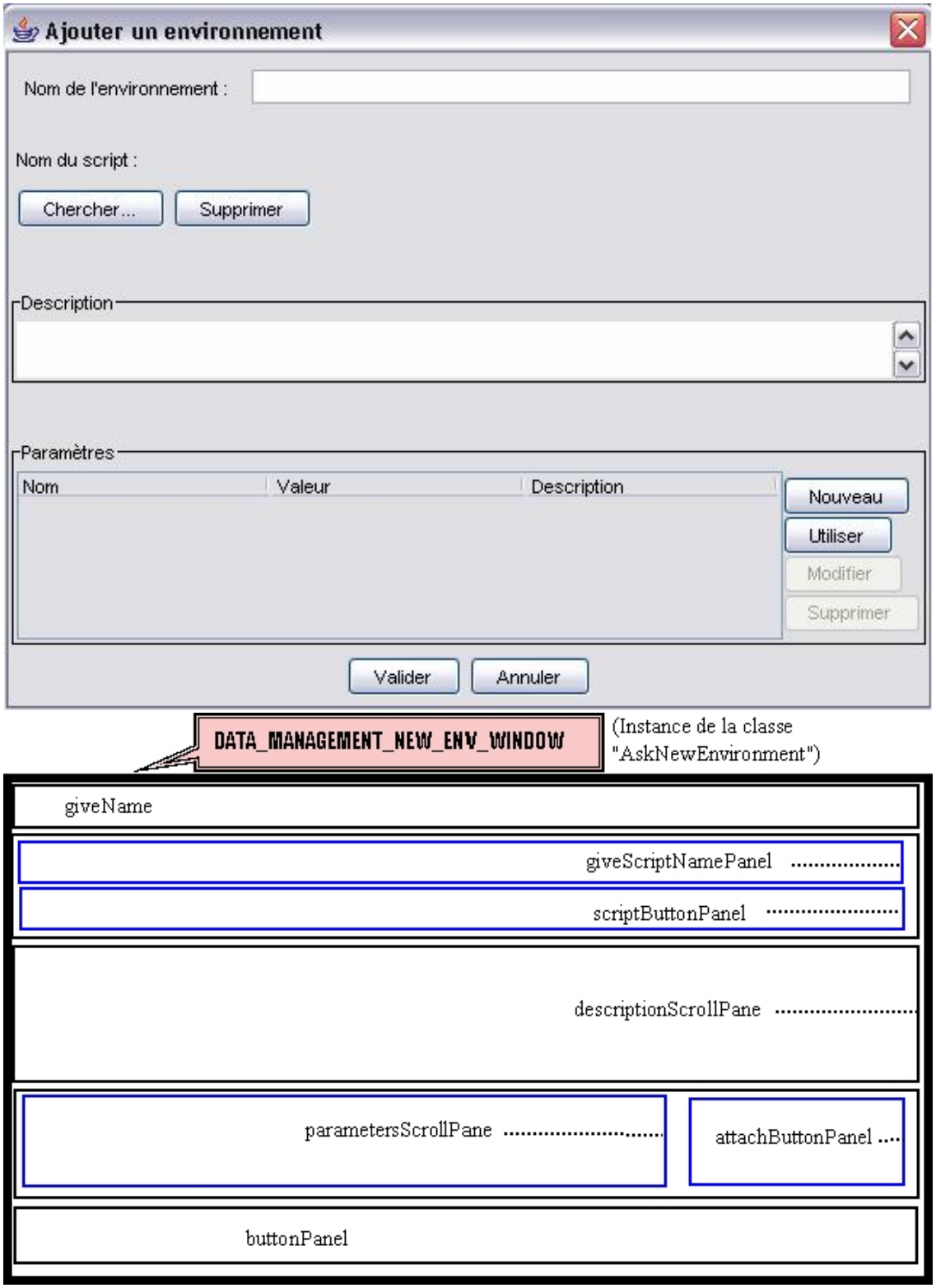



FIG. 2.15 – Fenêtre pour l'ajout/modification d'un nouvel environnement

# Chapitre 3

# Tutoriel : développement du plugin *"bugzilla"*

Afin d'illustrer le développement d'un plugin de type *Common* par un exemple, nous allons détailler le développement du plugin *"Bugzilla"*, plugin ayant pour but d'offrir des fonctionnalités de gestion de bug dans *Salomé TMF* en utilisant le bugtracker *Bugzilla2.16*.

## 3.1 Le fichier Manifest

La première partie du développement d'un plugin de Salomé TMF consiste en l'écriture du fichier manifest : *plugin.xml*. Ci-dessous le contenu du fichier manifest du plugin Bugzilla :

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Manifest 0.2" "http://jpf.sourceforge.net/plugin_0_2.dtd">
<plugin id="bugzilla" version="0.0.1"  class="salomeTMF_plug.bugzilla.BugzillaPlugin">
<requires>
        <import plugin-id="core"/>
</requires>

<runtime>
        <library id="Bugzilla" path="bugzilla/bugzilla.jar" type="code"/>
</runtime>

<extension plugin-id="core" point-id="Common" id="bugzilla.Common">
        <parameter id="class" value="salomeTMF_plug.bugzilla.BugzillaPlugin"/>
        <parameter id="name" value="Bugzilla"/>
        <parameter id="description" value="Plugin Bugzilla"/>
</extension>
</plugin
```

La première partie du fichier (après l'entête) précise le nom du plugin, sa version et la classe principale qui étend la classe Plugin.

Ensuite, dans la balise *"requires"*, le plugin déclare le ou les plugins dont la présence est nécessaire pour son activation. Ici, il s'agit du plugin core, puisque le pluing Bugzilla utilise le point d'extension Common.

Dans la balise *"runtime"*, le plugin déclare la liste des librairies utilisées, ainsi que leurs types.

La dernière partie du fichier (balise *"extension"*) précise les informations relatives aux points d'extensions utilisés : plugin fournissant le point d'extension, nom du point d'extension, ainsi que les paramètres spécifiques au point d'extension.

## 3.2 Implémentation de l'interface *"Common"*

La classe d'entrée du plugin, en l'occurrence la classe *BugzillaPlugin* du package *salomeTMF_plug.bugzilla* pour le plugin Bugzilla, doit implémenter l'interface *Common* puisque le plugin utilise le point d'extension Common.

Les fonctions principales de cette classe se composent en deux parties :
- fonctions relatives aux menus "Outils";
- fonctions relatives aux autres composants graphiques de Salomé TMF.

En ce qui concerne les menue "Outils" (présents dans les parties "Gestion des tests", "gestion des campagnes" et "Gestion des données"), il existe deux méthodes par menu. Pour la partie "Gestion des tests" par exemple, ces deux méthodes sont les suivantes :

- **isActivableInTestToolsMenu()** : méthode qui renvoie un booléen précisant l'utilisation du menu par le plugin.
- **activatePluginInTestToolsMenu()** : méthode permettant au plugin d'ajouter ces fonctionnalités dans le menu (le menu est passé en paramètre à cette méthode).

Pour ce qui est des fonctionnalités du plugin utilisant d'autres composants graphiques de Salomé TMF (définies en section 2.3.2), il existe quatre méthodes à implémenter par le plugin :

- **usesOtherUIComponents()** : méthode qui retourne un booléen précisant l'utilisation d'autres composants graphiques de Salomé TMF par le plugin.
- **getUsedUIComponents()** : méthode qui retourne la liste (de type java.util.Vector) des composants graphiques utilisés pas le plugin. Ci-dessous un extrait de l'implémentation de cette méthode pour le plugin Bugzilla :

```
public Vector getUsedUIComponents() {
        Vector uiComponentsUsed = new Vector();
        uiComponentsUsed.add(0,UICompCst.MANUAL_EXECUTION_BUTTONS_PANEL);
        uiComponentsUsed.add(1,UICompCst.ATTACHMENTS_BUTTONS_PANEL);
    [?]
        return uiComponentsUsed;
}
```

- **activatePluginInStaticComponent ()** : méthode permettant d'activer le plugin dans les composants graphiques statiques parmi ceux renseignés par la méthode précédente. Le plugin doit préciser dans cette méthode la manière avec laquelle il sera activé selon les composants graphiques. Voici un extrait de cette méthode pour le plugin Bugzilla :

```
public void activatePluginInStaticComponent(Integer uiCompCst) {
        if (uiCompCst == UICompCst.DATA_MANAGEMENT_ENV_TABLE) {
                environmentTable = (JTable)SalomeTMF.getUIComponent(uiCompCst);
                return;
        }
        if (uiCompCst == [?]) {
                [?]
        }
        [?]
}
```

- **activatePluginInDynamicComponent()** : idem que la méthode précédente pour les composants graphiques dynamiques de Salomé TMF.