Functional Setup With TSL: Attaching GREP To Eclipse/ExDT

Preface

In this paper we consider a sample task of attaching a new tool to the Eclipse/ExDT environment and solve it illustrating the logic of functional setup with TSL. As a sample, we select GREP – a great powerful command-line tool to search for a static text or for a basic or an extended regular expression, which came from the UNIX world.

Describing GREP Input Language In TSL

To attach a tool to Eclipse/ExDT, we need to establish relationships between tool's parameters, its graphic setup dialogs and its command line options must be established. In Eclipse/ExDT these relationships ares described in custom TSL – Tools Specification Language.

Getting Started

Well, let's start writing a TSL specification for GREP. First, create a file named **grep.xml** in the Eclipse subdirectory `plugins/com.elphel.vdt_1.1.0/tools` and write down the standard TSL frame:

```
<?xml version="1.0" encoding="UTF-8"?>
<elphel-project>
</elphel-project>
```

GREP is a simple utility. It is not a part of any package that would need spaical setup, neither it takes any configuration or command files on input – only a single command line. Thus we do not need to describe any `package` ot `project-template` contexts – the only `tool` context with a setup dialog and a single control line will suffice:

```
<elphel-project>
  <interface name = "GREPInterface">
  </interface>
  <tool name      = "grep"
        label     = "GREP tool"
        exe       = "grep.exe"
        interface = "GREPInterface">
  </tool>
</elphel-project>
```

In the header of the `tool` context section we specify the tool's internal name, the setup dialog's title, the executable program's name and the internal name of the interface describing the general syntax of the tool's control language. Also we insert this `interface` section with the same name. Internal names serve for references between objects only, thus we may invent them arbitrarily.

GREP's Control Language

If we run GREP with the single parameter '`--help`', we'll see a long list of its input parameters. First of all, notice that most of parameters are specified either in a form of '*`--parameter-name`*' or '*`--parameter-name=value`*'. This is what we call a *general syntax* of GREP's options. The only parameters that do not fall into these two cases are the pattern to match (which is specified as '`'value'`') and the list of input files to search in.

Parameters of the first sort are of logical nature – they serve only to enable or disable some program's behavior, or operation mode: when such a parameter is present in the command line, the behavior is enabled, otherwise it is disabled.

Parameters of the second sort serve to pass some value associated with some behavior to GREP. As we can observe, there is the only parameter (pattern to match) that can be an arbitrary text string, se values are either some arbitrary texts, two parameters that are numbers (of lines) and some parameters whose values are some predefined texts from a fixed set (e.g. `output-only`).

So, we need to define some *data types* for three purposes: to represent the values of these parameters internally, to know how to input them and how to check their correctness, to know how to write these values in the command line. Since we see that all the variety of parameters values falls in some common forms (logical, string, numeric and enumeration types), we can also speak of some *general semantics* of GREP's parameters.

Now the rule to learn and follow is: all general rules and forms (syntax and semantics) are defined in the `interface` section, as they may be shared among several tools (not in the case of GREP, however); and all concrete forms for certain parameters are defined in the `tool` section (as tool-specific).

Let's do this in turns.

Interface Definition

We've found out that general forms of GREP's options are not numerous. Let's write in their *output formats*:

```
<interface ... >
  <syntax name = "EquationParamSyntax"
          format = "--%%ParamName=%%ParamValue"/>
  <syntax name = "BoolParamSyntax"
          format = "--%%ParamName"/>
  <syntax name = "InputPatternSyntax"
          format = "'%%ParamValue'"/>
</interface>
```

Format names are internal names invented for reference purposes only. Format strings are patterns consisting of fixed text parts and variable parts. Variable parts are denoted by predefined *patterns-generators* which will be replaced by certain texts. E.g., the `%%ParamName` will be replaced by the parameter name and `%%ParamValue` – by the parameter's value.

The output format `EquationParamSyntax` is purposed for value parameters, `BoolParamSyntax` is for logical parameters, and `InputPatternSyntax` is for the pattern to match, which is just a string value in apostrophes.

Now turn to types. Usually, their definitions come at the beginning of the `interface` section.

We need string and numeric types for describing GREP's parameters. However, as they are commonly used data, there are predefined types `String` and `Cardinal` in the basic interface (called `BasicInterface`) which we can use without special notes.

Things become a bit less obvious with a logical parameter type. In GREP, the output form of any logical parameter is either the parameter name prefixed by '`--`', or an empty string. It is easy to see that we do not have a format which allows to generate both forms, neither can build it using two above generators. Quite an elegant solution of this problem will be given later, using a special attribute. Right now, we just point out that TSL boolean types are of no use with GREP.

At last, to represent fixed values set types we will use TSL enumeration types that need be defined. Let's define such a type for parameter '`--binary-files`':

```
<interface ... >
  <typedef name = "BinaryFilesType">
    <paramtype kind = "enum" base = "String">
      <item label = "Text"     value = "text"/>
      <item label = "Binary"   value = "binary"/>
      <item label = "No match" value = "without-match"/>
    </paramtype>
  </typedef>
  ...
</interface>
```

The internal type name is used for references only and is invented arbitrarily. Labels specify how the values of this type will be shown in the input drop-down list. `Value` attributes specify how values will be substituted in the output formats, replacing the `%%ParamValue` pattern.

All other necessary enumeration types may be declared in similar ways.

Parameters Definition

Now we turn to the tool specification, to define all GREP parameters.

Each parameter is to be described inside the `tool` section. Let's add, for example, a definition of parameter `ignore-case`:

```
<tool ... >
  ...
  <parameter id      = "ignore_case"
             outid   = "ignore-case"
             label   = "Ignore case distinctions"
             type    = "BoolOnOff"
             format  = "BoolParamSyntax"
             default = "false"
             omit    = "false"/>
  ...
</tool>
```

Internal identifier `id` serves for references to the parameter from other parts of the specification. Note that the value of this attribute must contain only Latin letters, digits and underscore characters. Therefore there is the optional attribute `outid`, which is free of such restrictions. It contains the output name of the parameter, as substituted in the format for `%%ParamName`. We need not to specify `outid` if it equal to `id`.

Attribute `label` defines a text shown in the tool's setup dialog to explain this parameter.

Atttibute `default` contains the default value of the parameter; the optional attribute `omit` contains the value that suppresses outputting this parameter into the command line.

This parameter's type `BoolOnOff` is defined in the basic interface. Actually, we could use any boolean type here, because its output forms `formatTrue` and `formatFalse` will never get into the command line. Why? The point is that there is no the pattern `%%ParamValue` in the output format `BoolParamSyntax` of this parameter.

Together with attribute `omit` this gives the promised solution of the problem with logical parameters: indeed, when the parameter has the `true` value, its output form will be '`--ignore-case`' (see format `BoolParamSyntax`); but when the value is `false`, `omit` forces to output nothing into the command line.

Almost all parameters may be defined in this way.

Note now that some GREP's parameters are mutually exclusive. E.g., consider options '`--extended-regexp`', '`--fixed-strings`', and '`--basic-regexp`'. If we introduce three different parameters for them, it will be possible to pass two or more of them together. Well, we can do this anyway when calling GREP, and it will be its problem how to interpret this, but we better find a way to avoid such collisions at the specification level.

The solution, in fact, is quite simple: let's consider all these parameters as different states of some operation mode and let's define a parameter of an enumeration type for this mode, that has *names* of these options as *values* (or, at least, as their output forms). For the above three parameters we define this type and an additional output format as follows:

```
<interface ... >
  ...
  <typedef name = "PatternTypeType">
    <paramtype kind = "enum" base = "String">
      <item label = "Fixed"          value = "fixed-strings"/>
      <item label = "Basic regexp"   value = "basic-regexp"/>
      <item label = "Extended regexp" value = "extended-regexp"/>
    </paramtype>
  </typedef>
  ...
  <syntax name = "EnumParamSyntax"
          format = "--%%ParamValue"/>
</interface>
```

The mode parameter itself looks as:

```
<tool ... >
  ...
  <parameter id      = "pattern_type"
             outid   = "pattern-type"
             type    = "PatternTypeType"
             format  = "EnumParamSyntax"
             default = "fixed-strings"
             omit    = "fixed-strings"
             label   = "Pattern type"/>
</tool>
```

Setup Dialog Definition

After all parameters are defined, it is time to design the appearance of the GREP setup dialog to input parameters values in the Eclipse/ExDT shell.

We design to segregate all parameters into two groups: one group of parameters to control the search mode, and another group of parameters to control results output mode.

All we need to do is to fill the `input` section with parameters names by groups:

```
<tool ... >
  ...
  <input label="GREP Preferences">
```

```
    <group label="Match">
      "pattern"
      "pattern_type"
      "word_regexp"
      "line_regexp"
      "ignore_case"
      "invert_match"
    </group>
    <group label="Output">
      "no_messages"
      "with_filename"
      "binary_files"
      "byte_offset"
      "line_number"
      "directories"
      "output_only"
      "count"
      "before_context"
      "after_context"
    </group>
  </input>
</tool>
```

Note that here we write internal parameters names (`id`s, with underscores) not output options names (`outid`s, with minuses)!

`Label` attributes specify the dialog's and tabs' titles. The tabs and parameters in tabs will apeear in the order of enumeration here. Each parameter will contain an explanatory text defined for it and an input control: a checkbox for a boolean parameter, a drop-down list for an tnumeration parameter and an input field for text and numeric parameters.

Command Line Format Definition

Finally, let's create the `output` section and fill it with names of all introduced parameters:

```
<tool ... >
  ...
  <output>
    <line name = "command_line" sep = " ">
      "%pattern_type"
      "%word_regexp"
      "%line_regexp"
      "%ignore_case"
      "%invert_match"
      "%no_messages"
      "%with_filename"
      "%binary_files"
      "%byte_offset"
      "%line_number"
      "%directories"
      "%output_only"
      "%count"
      "%before_context"
      "%after_context"
      "%pattern"
      "%(%%FileList%| %)"
    </line>
  </output>
</tool>
```

The only `line` section with `name="command_line"` means that there is only one control line for GREP utility and this line is passed as a command line to it. Attribute `sep` says that options will be separated by a single blank. The list of output elements specifies the format of the command line. Their order is only important if it makes sense for the utility.

Each element in the list is just a string. All these strings are concatenated via the separator to meke the whole format. The format may contain any fixed texts, if you need them (not in our case, however). But typically, the list looks as above: internal parameters names are prefixed by single percent characters to make patterns-options, which expand to parameters format strings according to their definitions. E.g., for the `pattern_type` parameter the defined format is `EnumParamSyntax` which is `"--%%ParamValue"`, hence the output option may look like '`--basic-regexp`'.

The last element in the list, "`%(%%FileList%| %)`" is a so-called pattern-repetitor. When GREP is launched, this pattern will be replaced with the full list of files in the user's project separated by blanks.

The final GREP definition may be found in file `tools\Simple Samples\`**`grep.xml`**.

Attaching GREP To Eclipse/ExDT

Now, when the description of GREP's control language is over, we must think of making it available from Eclipse/ExDT shell.

All tools in Eclipse/ExDT are available from the design menu. Let's select a menu where we wish to put GREP tool in – let it be the sample MainDesignMenu3. All we need to do,to add the new tool is to define a new menu item in the right place of the menu (see this in `plugins/com.elphel.vdt_1.1.0/tools/`**`DesignMenu.xml`** file):

```
<menu name="MainDesignMenu3" ...>
  ...
  <menuitem name="GREP"
            label="Run GREP"
            call="grep"/>
</menu>
```

The `name` attribute here is the menu item name. `Label` is a tool name, as it will be seen in the menu. `Call` refers to the `tool` context that describes all of the GREP's stuff that we have done above.

Now, to make GREP available, you only need to set the MainDesignMenu3 menu as the working menu for your project.

Installation Setup Of GREP

Attachment is done, but we still need to point to Eclipse where to find GREP program.

This is done in the installation setup dialog available from the menu. Since GREP is a stand-alone utility, you will find it in the list of available package and stand-alone utilities. Open its setup dialog and specify (by browsing) the absolute location of the program. If you set this path to empty, the call line generated for this utility will not contain any path at all. This may work, if your system `PATH` environment contains the path to GREP already.

Well, we did it!