Project configuration guide for OsiriX plugins using the ITK framework

Author: Brian Jensen

Systems Development Project (SEP)
Chair for Computer Aided Medical Procedures & Augmented Reality,
Department of Informatics, TU München
Supervisor: Ralph Bundschuh
Advisor: Prof. Dr. Nassir Navab

This document covers the steps necessary to enable a plugin for the popular OsiriX dicom viewer to safely make use of the ITK framework. This process involves some source level modification of the ITK source code, although this happens on a very superficial level and should have absolutely no functional difference to an unmodified version of the framework. Only the aspects of Osirix plugin development or objective-c++ programming that are relevant for ITK integration are discussed, for more general information on these subjects see the OsiriX plugin guide. As a reference the source code for the plugins *PetSpect-Fusion* (http://campar.in.tum.de/Students/IdpPetSpectRegistration) or *NM_Segmentation* (http://campar.in.tum.de/Students/SepOsiriXSegmentation) can be used.

1 Prerequisites

The foremost requirement is that you have a current version of Apple's developer tools installed on your system, this should be version 3.1.0 or newer. You should also have already created a valid OsiriX plugin Xcode project. If you have not done this, you can create a new plugin project using the helper script located under *plugins/_help/Osirix Plugin Generator.zip* of the main OsiriX svn directory. This script automatically creates a new Xcode project with the necessary settings for creating an OsiriX plugin. Although it is technically not required, it is highly recommended that you check out a copy of the OsiriX source code from the svn repository. This is accomplished by entering the following command in the terminal

```
svn co https://osirix.svn.sourceforge.net/svnroot/osirix osirix
```
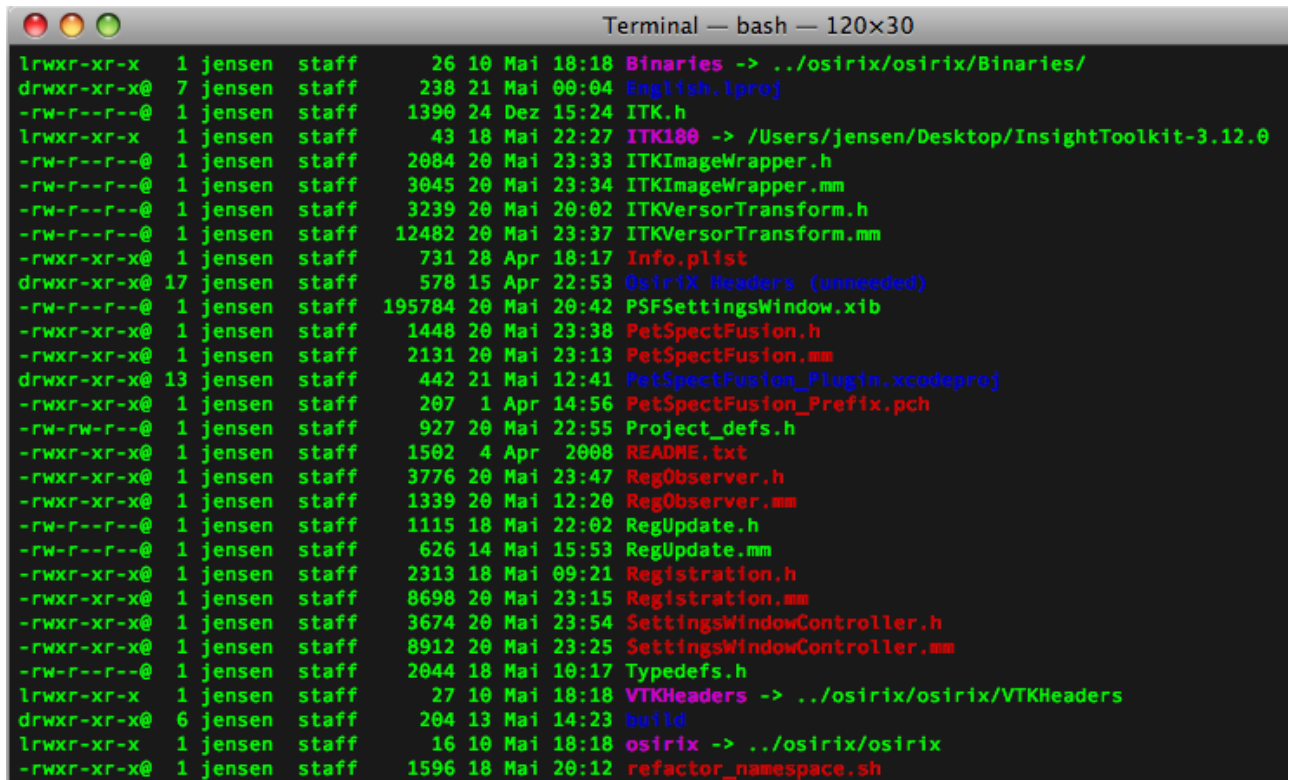
This will download a copy of the newest source code revision into the current directory. The OsiriX Xcode project is located under the subdirectory *osirix*, and many of the standard plugins included with OsiriX are located under *plugins*. When compiling the OsiriX source code for the first time, don't forget to set the active target to **Unzip Binaries** and build the project. This step unzips the ITK and VTK libraries necessary to build OsiriX and only needs to be run once, afterwards you should set the active target to **OsiriX**.

You will also need to download a copy of the ITK source code, which can be obtained from http://www.itk.org. In order to configure ITK you will need to install CMake located at http://www.cmake.org.

2 Xcode configuration

There are two options available for configuring your Xcode project. You can either let your project use the modified Osirix headers that included with project when it is created, or you can use the Osirix headers straight from the source code. The latter naturally offers more flexibility, but it requires that you have the source code on your hard drive. You also have the choice of using the ITK libraries included with OsiriX, or as is highly recommended using your own copy of ITK, which will be explained in section 3.

2.1 Configuring paths

Figure 1: The path listing of the PetSpectFusion plugin project folder with the four required symbolic links

Regardless of whether the original OsiriX headers are used, the search paths for the ITK headers have to be setup. This can be accomplished either by using project wide settings or by specifying the search path for each individual file that uses ITK classes, although I will only cover the project wide settings. First you will need to set a few symbolic links.

Figure 1 shows the four symbolic links used by the example project PetSpectFusion. Each symbolic link needs to be created in your plugin's project root, which should be the directory

where the Xcode project file is located. The only real required symbolic link is *ITK180*, which either points to *osirix/ITK180* in the OsiriX source code directory, or to your own copy of ITK. If you want to have access to all the OsiriX header files, then you will need a symlink osirix that points to the Xcode project directory inside the OsiriX source code. If your project uses VTK, then you will need the symbolic links *Binaries*, which points the directory *osirix/Binaries*, and *VTKHeaders* which points to *osirix/VTKHeaders*.

Next you need to configure the user header and library search paths in Xcode. For this select your plugin's name under **Targets**, right click and select **Get Info**. This opens the project build settings dialog as can be seen in figure 5.

Make sure that **All Configurations** is selected in combo drop down box, so that the settings are modified for all configurations. Under the section **Search Paths** double click **User Header Search Paths**. This will bring up a configuration sheet like in figure 2. Copy the paths to the style sheet so that they match those present in figure 2. The path *ITK180/Code/Review* only needs to be entered if ITK was configured to use the review directory, see section 3. If you are not using the headers from the OsiriX source code, then the path *osirix* is unnecessary.
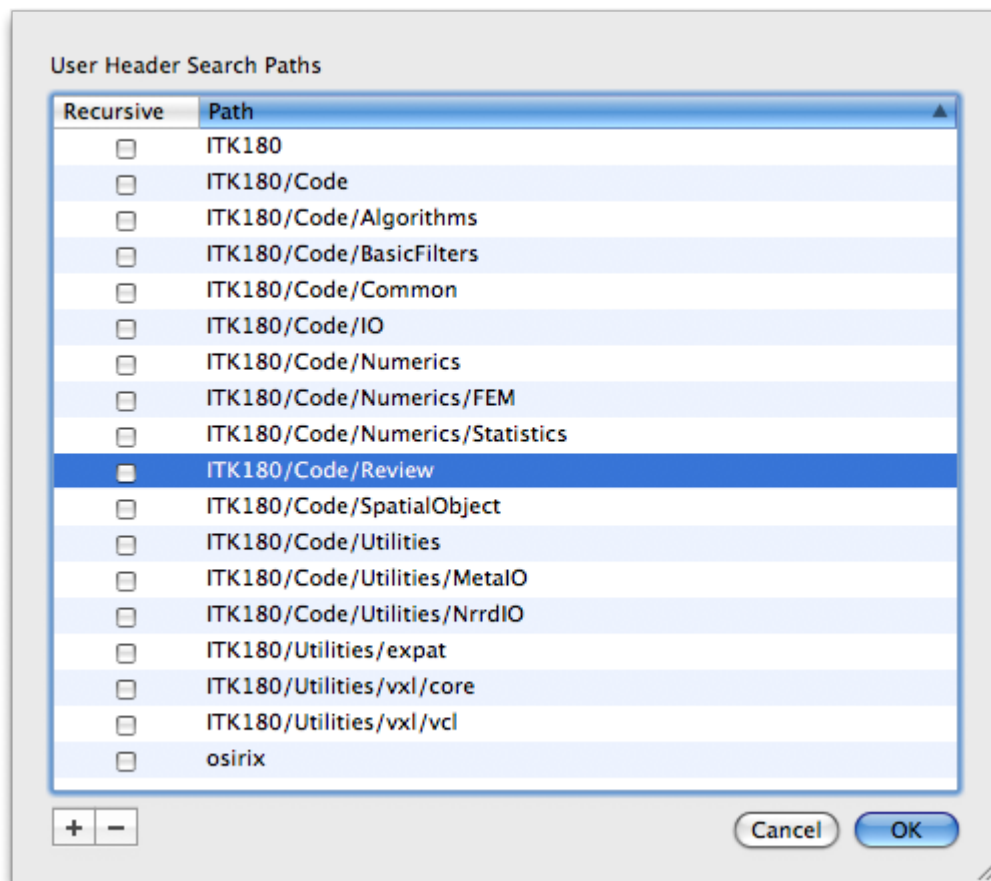

Figure 2: The user header search paths

Next you need to modify the setting **Library Search Paths**. You should modify the settings to match those present in figure 3.
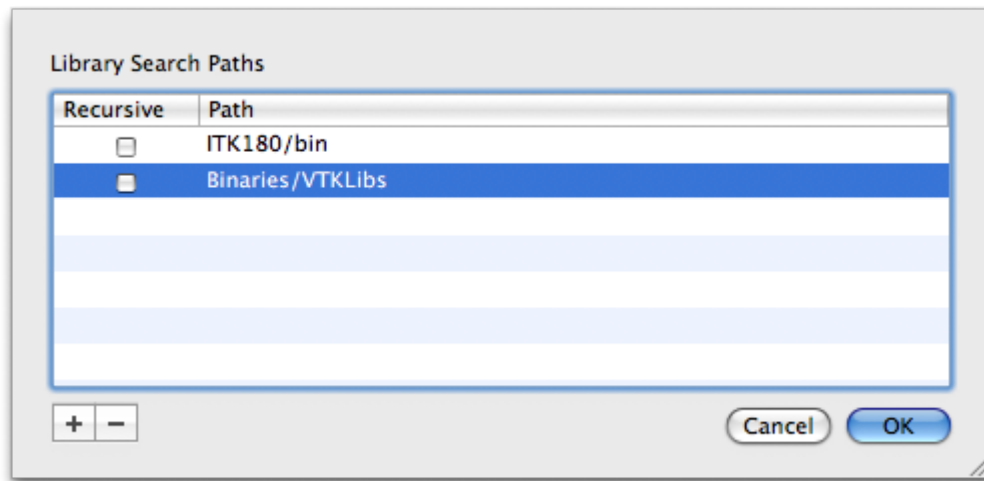
Figure 3: The user library search paths

2.2 Configuring build settings

Once the symbolic links and the search paths have been configured the build settings need to be modified. The first thing that you might want to modify is the **Architectures** setting. This setting has to match or expand the settings present in the OsiriX executable you wish to target. The publicly released OsiriX version supports the **i386** and **ppc7400** architectures. The 64 bit extension also supports the **x86_64** architecture, so I recommend that you target all three architectures.

Under the section **Compiler Version** you should select **GCC 4.2**. This version of gcc generates more optimized code compared to the default of gcc 4.0. Under the section **Linking** you can make two optional changes that help to avoid runtime linking problems. In **Other Linker Flags** remove both the entries **-undefined** and **dynamic_lookup**. Then modify the **Bundle Loader** property to point to the OsiriX executable you are targeting like in figure 4. This option enables the static linker to verify that any symbols used by the plugin are actually present in the host executable, but this also means the target executable has to support all the architectures your plugin targets, otherwise the linker will generate errors.

The final mandatory step is to tell the static linker which ITK libraries to link against and where they are located. This is accomplished in the main viewer window by adding a new group under **Frameworks and Libraries**. You can name the new group as you please, but I recommend you choose a new name that is descriptive of its purpose. Then right click on the new group and select **Add** then **Existing Files...**. In the dialog navigate to the *bin* folder of the ITK distribution you are using, and select all files ending in **.a**. Your new group should look something like figure 6.

An optional step for easing the transition between development and deployment builds is to include a script that automatically changes a link in the OsiriX plugins directory to point to the current build. This is accomplished by simply adding a new build phase to the plugin target by right clicking the target and selecting **Add** then **New Build Phase** then New Run Script Build Phase. Simply copy the example script in figure 7 and a symbolic link under *USER_HOME/Library/Application Support/Osirix/Plugins/* to current plugin bundle will be automatically generated every time you build the project. The scrip should be added at the
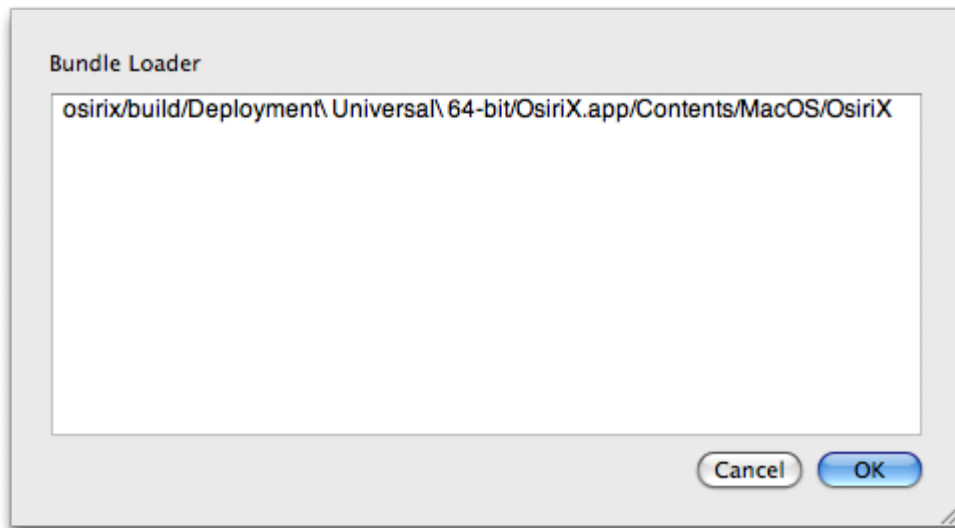
Figure 4: The bundle loader path used to verify presence of symbols in the target application

end of build flow, if not make sure it is the last task run.

Note: I highly recommend that you set all of the symbolic links mentioned in this section using the command line **ln** command and not using aliases. It has been my experience that using aliases leads to random errors were certain files were not always found by the build and plugin loading process, leading to very difficult errors to debug.

3 ITK configuration

This section deals with creating your own ITK libraries to link with your plugin. Ideally you should be able to freely chose between using the ITK libraries included with the OsiriX source code or using your own version. In practice this is only true under certain conditions due to the way cocoa loads plugins and the dynamic linker resolves symbols. Without any modifications you can only safely use your own ITK copy if it is the same version and compiled with the same options as the one in your targeted version of OsiriX.

The main reason for this condition is due to the way the static and dynamic linkers handle symbol management for Mach-O files, the executable format used on OS X. When you statically link against a library, even though any referenced symbols (functions, classes, global variables) are copied into your executable, the references to these symbols don't point directly to the symbols themselves, but instead into the symbol table of the executable. The symbol table holds the offset in the executable for each statically linked symbol, and is empty for symbols that are loaded at runtime by dynamic linker (unless prebinding is activated). When the executable is run and a piece of code accesses a statically linked symbol such a function, instead of calling the function directly, it first calls the dynamic linker. The dynamic linker determines the proper address of the function from the symbol table, and replaces the reference to the function with the correct runtime time address of the function in the calling code. Control returns to the calling code just before the function call, so that the function call this time won't be to the dynamic linker, but to the actual function.

When OsiriX loads your plugin, Cocoa's bundle loading facilities copy your plugin's exe-
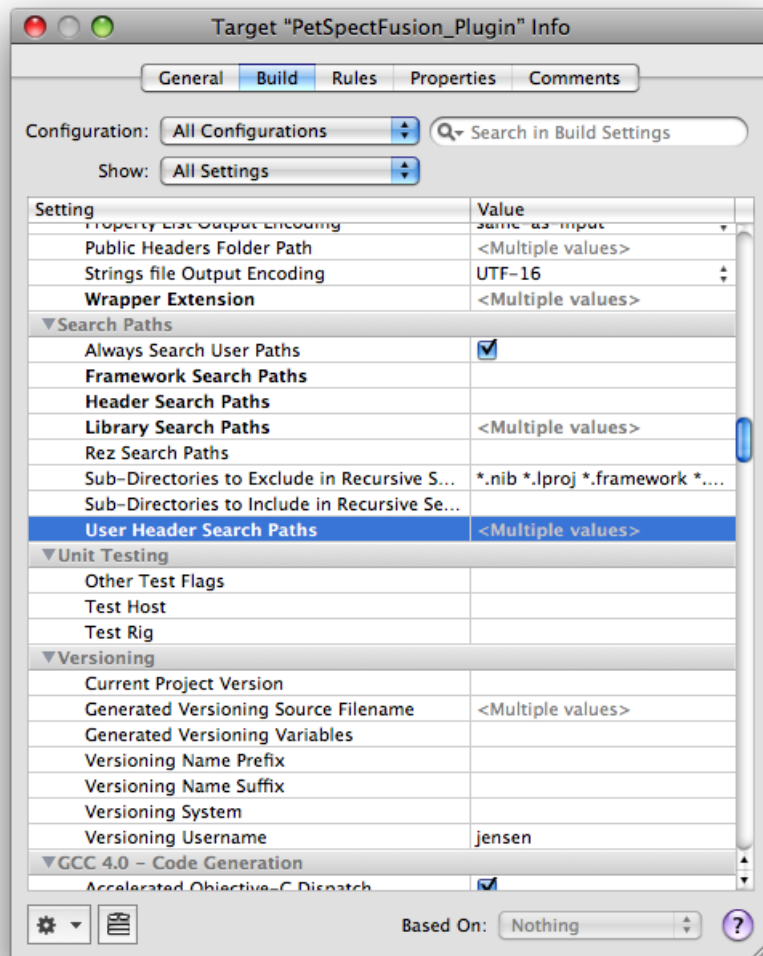
Figure 5: The project build settings dialog

cutable code into OsiriX's executable section in memory. One effect of this behavior is that the symbol table in your plugin's executable is merged with the OsiriX executable's symbol table. The catch is that only the symbols that are not already defined in OsiriX's symbol table are copied, meaing that if your plugin and OsiriX have some of the same symbol names defined, only the OsiriX version of those symbols will be available to your code, even if they are implemented completely differently.

That means the only safe way to use ITK in your plugin is to either use the exact same version as is present in your targeted version of OsiriX, or use your own copy of ITK with symbol names different from those present in OsiriX. The former option is highly problematic from a maintenance perspective, because you would have to release a version of your plugin for each version OsiriX in order to be sure there are no unexpected symbol collisions.

In comparison creating a version of ITK with unique symbols has far less risks of unexpected behavior and is much more maintainable. The easiest way to accomplish this is to use a
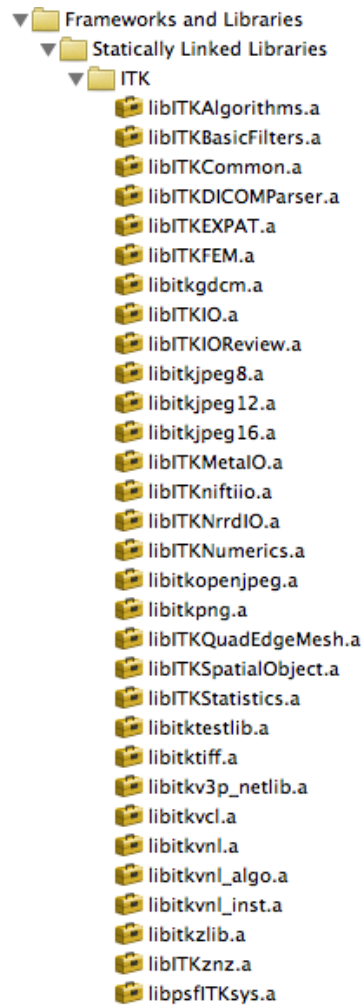
Figure 6: The list of ITK libraries the PetSpectFusion plugin is linked against. Because Xcode always stores these links as absolute pathnames, it is important that the libraries are read added each time you change your ITK location.

script included with the PetSpectFusion and NM Segmentation plugins' source code that automatically refactors the ITK namespace used in the libraries to a user specified name before they are compiled. Technically an alternative solution would to be use the low level bundle loading facilities to manually resolve the ITK symbols in your plugin, but such a method would likely involve manually resolving tens to hundreds of symbols and would quickly become infeasible.

3.1 CMake build settings

In order to refactor the ITK namespace you first have to download and configure the ITK source code. The configuration is taken care of using CMake. When configuring ITK make sure you set the build directory to the source directory, so that the ITK build will have the same structure as the ITK build present in the OsiriX source code. This can be convenient if you want to easily switch between the OsiriX ITK version and your ITK version, all you

would need to do is change location pointed to by the *ITK180* symbolic link in your probject directory. It is important that you turn off **BUILD_TESTING** in the CMake configuration options, because the automatic namespace refactoring script has to deactivate the ITK test driver. You may also wish to disable **BUILD_EXAMPLES** in order to speed up the build time and reduce the hard drive space required. For the **CMAKE_BUILD_TYPE** setting, both **Release** and **MinSizeRel** are appropriate options. Make sure all of the architectures targeted by your plugin are listed in under **CMAKE_OSX_ARCHITECTURES**. The rest of the default settings should not need to be changed.

3.2 Namespace refactoring

After you have configured your copy of ITK and generated the build files, you need to run the namespace refactoring script named *refactor_namespace.sh*, which can be obtained from the source of the *PetSpectFusion* (http://campar.in.tum.de/Students/IdpPetSpectRegistration) or *NM_Segmentation* (http://campar.in.tum.de/Students/SepOsiriXSegmentation) plugins. The script makes changes to the source code of the ITK libraries, so that the ITK namespace is changed to a namespace specific for your plugin. The script is used as follows:

```
refactor_namespace.sh path/to/itk yourNamespace
```

Once the script completes you can simply build itk by starting *make* in the itk build directory. After building completes you need to change the **ITK180** symbolic link in your project directory to point to your newly created ITK build. Then final step involves modifying all the references to the ITK namespace in your source code and replacing them with the new namespace. The easiest solution is to run a global find / replace on all instances of *itk*, but this makes switching between ITK versions a pain. A better approach is to set a **#define** directive in a header file included in all your source code files that use ITK, so that the namespace used can be easily changed at compile time. As an example:

```
#define ITKNS myITKNamespace

...

ITKNS::ImageType::Pointer image = ....
```
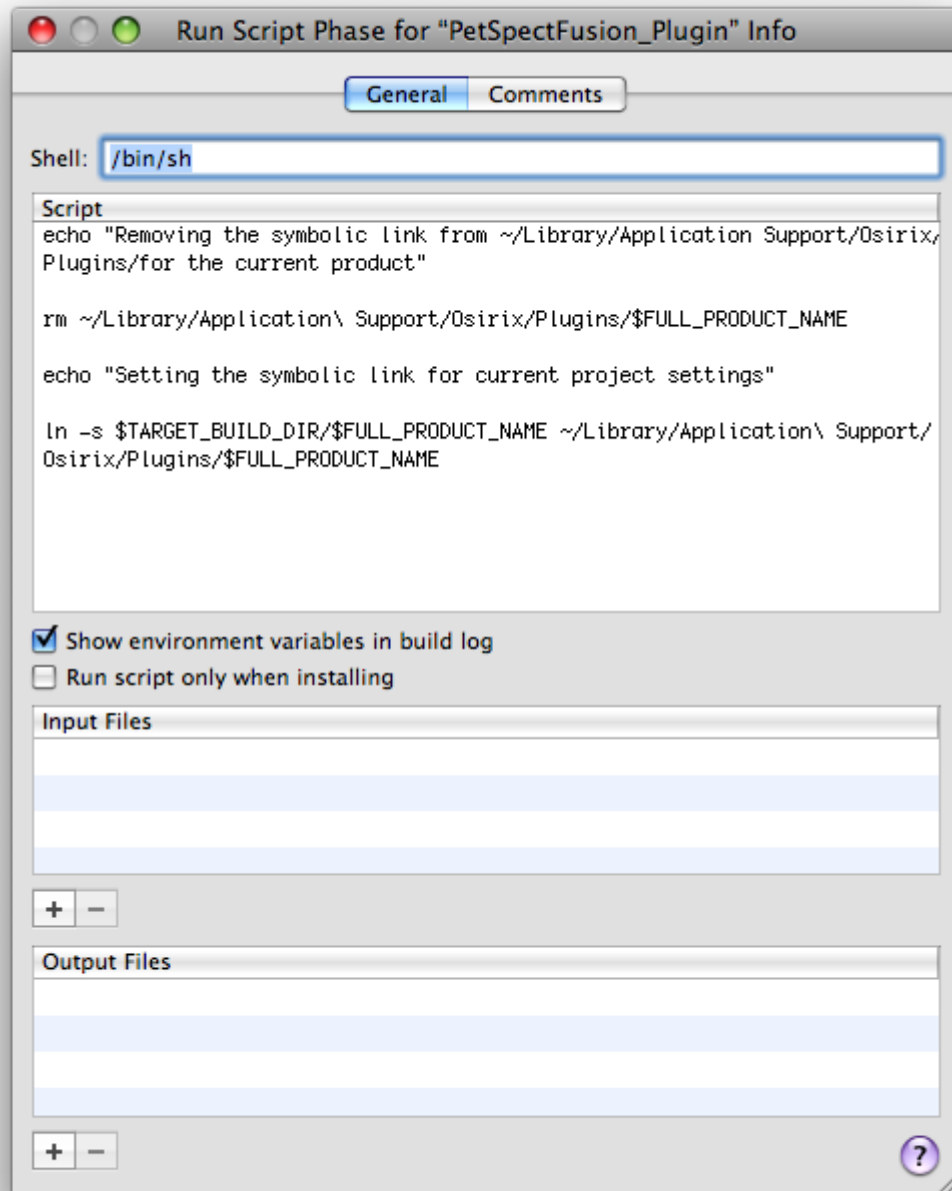
Figure 7: A custom script that automatically creates a symbolic link to the plugin bundle for current configuration.