preCICE

Developer's Guide

February 7, 2013

Contents

1 Introduction

This guide is meant for all people aiming at working on the source code of preCICE. In section 2 a development environment based on Eclipse is configured. It is not necessary to work with Eclipse, but Eclipse has proven to be a valuable tool, especially with regards to its plugin capabilities. One very important plugin simplifying SVN operations is Subclipse, whose basics are explained in section 3. section 4 lists basic C++ programming and software pattern knowledge that should be clear before starting programming on preCICE, otherwise no source code of good quality can result. All coding conventions and the coding style is given in section 5. Every programmer working on preCICE should stay as close as possible to these rules, unless there are serious reasons not to do so.

Back to Contents.

2 Setup

This section describes how to setup a development environment for preCICE using Eclipse for coding and version control (with Subversion, SVN), and using Eclipse or SCons for building libraries and executables of preCICE.

Back to Contents.

2.1 Preparing Eclipse

Eclipse can be downloaded for free under `http://www.eclipse.org/`. In order to program C++ in Eclipse, you should use the CDT plugin. You can get Eclipse as bundle with CDT, or get CDT afterwards as plugin. It is highly recommended to work with the plugin Subclipse that interfaces Subversion (see section 3). When working with Subclipse under Windows, make sure you also have downloaded and selected the SVNKit as client in the SVN interface to SVN (in preferences→team→SVN), in order to avoid unnecessary repeated password typing.

Plugins are installed into Eclipse as follows:

1. Click on the "Help" menu in Eclipse
2. Choose the "Install new software" entry
3. Enter the address of the new plugin into the "Work with:" field
4. Wait until the available modules of the plugin are displayed
5. Select the modules to be installed (usually all)
6. Continue by clicking "Next" and perform remaining installation steps

The plugin address of the CDT can be found here `http://www.eclipse.org/cdt/downloads.php`. The subclipse plugin address can be found at `http://subclipse.tigris.org/` under downloads.

Back to Contents.

2.2 Eclipse Basics

Working with Eclipse requires to define one (or several) *workspaces*, where the source code of the C++ projects (and other projects) is located. When starting Eclipse for the first time, the user needs to declare a directory to be a workspace. Manipulations of files within that workspace should be done through Eclipse only, in order to avoid problems. Eclipse creates special workspace and project files starting with a dot, that contain information about the projects and files contained in a workspace. These files are adapted automatically when files are manipulated through the Eclipse workbench. Nevertheless, in some cases it becomes necessary to do changes in the workspace outside of Eclipse. In order to make Eclipse recognize these changes, you have to refresh the ProjectExplorer view (by selecting the corresponding workspace folders in the ProjectExplorer and pressing F5, e.g.).

The content of the Eclipse workbench is configured through *perspectives*, which are tailored to the different tasks a user wants to do. Important perspectives are the C/C++, SVN Repository Exploring, and Team Synchronization perspectives. A perspective can be customized to show different *views*, which are subtags offering different functionalities.

A new C++ project can be created by clicking on the "File" menu, then at the entry "New" and selecting "C++ Project".

Back to Contents.

2.3 Retrieving Source Code from Subversion Repository with Eclipse

To retrieve the sources of preCICE (and the related project Peano) the Subclipse plugin of Eclipse can be used. The source codes are kept in a *version control system* called Subversion (SVN), which enables to retrieve any old version of the sources and lets several people work on the same sources concurrently in a copy-change-merge workflow. Detailed descriptions of SVN can be found under `http://svnbook.red-bean.com/nightly/en/index.html`. Subclipse is a graphical interface to SVN in Eclipse and enables simple and fast SVN operations. Documentation for Subclipse can be found at `http://help.collab.net/index.jsp?topic=/org.tigris.subclipse.doc/topics/toc.html`. The first step to get the sources of preCICE is to setup the location information of the SVN repository in the "SVN Repository Exploring" perspective. Before performing the steps listed below, make sure that you are in the user groups enabling the work with the preCICE and Peano repositories (i.e. `svn_precice` and `svn_peano`) by using the command

```
$ groups
```

and checking whether the required groups are listed.

ATTENTION: Be careful in the "SVN Repository Exploring" perspective, it is possible to delete sources from the SVN repository and all its old versions.

The repository locations of preCICE and Peano are added as follows:

1. Choose the "SVN Repository Exploring" perspective in Eclipse
2. Activate the "SVN Repositories" view
3. Rightclick at an empty spot in the "SVN Repositories" view
4. Select "New" and then "Repository Location..."
5. Enter the URL (you have to adapt the USERNAME, the machine number XY, and replace {at} by its symbol): `svn+ssh://USERNAME{at}atsccsXY.informatik.tu-muenchen.de/home_local/repositories/svn/precice`
   → an entry for the preCICE repository appears in the "SVN Repositories" view
6. Add the repository location for peano, which has the same location but ends with "peano" instead of "precice"

When expanding the repository location of preCICE or Peano, three subfolders appear:

- branches
- tags
- trunk

All of the three subfolders do contain complete versions of a project, but only the *trunk* has the most current version of the project. Thus, only the resources contained in this folder are necessary for development work on preCICE. Development does not take place directly on the resources in the SVN repository, but requires a *checkout* of the resources to a different place, which will be the before created workspace of Eclipse. Perform the following procedure for the preCICE **and** Peano repository:

1. Expand the repository location in the "SVN Repositories" view in Eclipse
2. Rightclick on the trunk folder
3. Select "Checkout..."
4. Make sure that the radio button "Check out as a project using the New Project Wizard" is selected
5. Make sure that the checkbox "Check out HEAD revision" is selected
6. Click on the "Finish" button and create a C++ project in the workspace

The checkout may take several minutes, depending on your network connection. The Peano repository is quite big, it is sufficient and recommended to checkout only the directory `src` of Peano.

Back to Contents.

2.4 Retrieving external tools and libraries needed to build preCICE

preCICE needs other external sources and libraries to be built succesfully. In addition to the preCICE and Peano sources, the following projects have to be available

- Boost C++ libraries (no installation is necessary) from `http://www.boost.org/`. Check which version is compatible in Downloads.
- MPI (recommended and mandatory for MPI socket communication is the newest MPICH2 version) from `http://www.mcs.anl.gov/research/projects/mpich2/` (a version of MPI2 is probably installed already)

Back to Contents.

2.5 Building preCICE

preCICE is built using the software construction tool SCons, which is based on Python. Hence, Python (version smaller than 3.0) is required to install SCons. Python can be downloaded from http://www.python.org/download/. SCons can be downloaded from http://www.scons.org/. For installation instructions look in the documentation of Python and SCons respectively. Please note, Python is probably already installed on the machine you are working on as may be SCons. If you do not have root rights on the machine you want to install Python or SCons, you have to do use the `prefix` argument on configuration of the programs to install them at a location you have full rights. SConstruct files are the root build files, and describe what has to be built and how. The preCICE project containes an SConstruct file as well as the Peano project. Actually, in the preCICE project there is one file for building under Windows, `SConstruct-windows`, and one for building under Linux, `SConstruct-linux`.

Before building preCICE, a couple of environment variables have to be exported:

- Export the environment variable `TARCH_SRC` to be the absolute path to the Peano `src` directory.
- Export the environment variable `BOOST_ROOT` to be the absolute path to the Boost project root directory.

To export an environment variable the following command can be used:

```
$ export VARIABLE_NAME=VARIABLE_VALUE
```

The content of a variable can be checked by

```
$ echo $VARIABLE_NAME
```

In order to save the time of repeated exports, the export commands should be added to the `.bashrc` (for bash command shell) file in the home directory of a user. After adding the commands to the `.bashrc` file, a command shell must be reopened or the command

```
$ source .bashrc
```

has to be issued to register the changes made to `.bashrc`.

ATTENTION: Be careful with changing the `.bashrc` file, if you introduce errors you might not be able to login to your user account any more.

In Windows, environment variables can be set in the extended system control menu.

Back to Contents.

2.5.1 Building from Command Line

For building preCICE on Linux change to its project root directory and type

```
$ scons -f SConstruct-linux
```

For building preCICE on Windows type

```
$ scons -f SConstruct-windows
```

On a machine with several cores, you can use more than one core by adding the `-j2` option (for 2 cores, e.g.) when invoking the build. An executable and a library of preCICE are then built into the folder
`precice_root/build/debug-dim2-mpi-serial/`.
Back to Contents.

2.5.2 Building from Eclipse

Under Linux, SCons builds can be integrated into Eclipse CDT, which prevents one from using the command line and allows to highlight compile errors directly in the source files. In order to enable the use of SCons in Eclipse, the following steps have to be done:

1. Open the project properties (by rightclicking at the project root folder in the "Project Explorer" view and selecting the corresponding menu point, e.g.)
2. Select the "C/C++ Build" entry from the left menu pane
3. Select the tab "Builder Settings" on the right window frame, which shows the settings for the currently activated build configuration (which is displayed in the top of the window)
4. Switch off the checkbox "Use default build command" and type instead the desired SCons command (`scons -f SConstruct-linux build=debug`, e.g.); one Eclipse build configuration can trigger one SCons build configuration, thus.
5. Switch off the checkbox "Generate makefiles automatically"
6. Select the root path of the project as build location
7. Switch to the tab "Behaviour" then, to further adjust build settings
8. Delete the string in the textbox "Build (incremental build)"
9. Replace the string in the "clean" textbox by `-c`, which is the SCons clean argument

Now, you should be able to build with SCons in Eclipse. Errors, warnings and info messages appearing as build output in the "Console" view are collected in the "Problems" view and additionally highlighted in the source code files and in the "Project Explorer" view of Eclipse.
Back to Contents.

3 SVN with Subclipse

3.1 How to operate SVN with Subclipse

As described in section 2, Subclipse is a graphical frontend for SVN. This section describes the basic operations and terms needed to operate with SVN in Eclipse.

Operations and terms discussed are

- Checkout
- Update
- Commit
- Conflicts

- Team synchronization
- Compare view

Back to Contents.

3.2 Checkout

A checkout is done to retrieve resources from an SVN repository. The procedure is described in detail in subsection 2.3. The checked out resources are a copy of the resources in the repository, i.e. changing the checked out resources does not affect the resources in the repository. This is the intended way to work on the resources of the repository.

ATTENTION: Do *never* work directly on the resources of a repository!!

Back to Contents.

3.3 Commit

In order to apply the local changes of the repository resources to the resources of the repository itself, a *commit* has to be done. Before the commit, the local changes are not seen by any other developers. Afterwards the other developers can retrieve the modified resources via an update (see subsection 3.4). The commit operation should always be done in the "Team Synchronizing" perspective, which is described in subsection 3.6. When performing a commit, a non-empty message **has to be** attached, otherwise the commit fails. This message should contain a brief description of the changes done to the resources to be committed.

Back to Contents.

3.4 Update

Since several people can work concurrently on the resources of a repository. The changes done by other developers must be somehow transported into the own local copy. The operation performed to get these changes is called *update*. In Eclipse, an update should always be performed in the "Team Synchronizing" perspective, which is described in subsection 3.6.

Back to Contents.

3.5 Conflicts

SVN allows several users to work on the same resource (file) of a repository. Thus, it is possible that two developers do change exactly the same part of a file in their local copy. The first developer commiting his changes will not run into any difficulties. However, the second developer encounters a special situation when trying to update or commit the very same resource, which is called a *conflict*. A conflict can only be resolved by comparing the version of the resource in the repository with the local version of the

resource. The "Team Synchronization" pesrpective provides functionalities to resolve such conflicts and is described in subsection 3.6.

Back to Contents.

3.6 Team Synchronization

Subclipse allows a direct commit of resources in the "Project Explorer" view. However, if a commit of a resource having conflicts is done, the conflicting parts will be overwritten by the local changes without any notice. This means, the work of another developer will be overwritten! Thus, never directly commit your local changes into the repository.

A similar situation can occur when performing a direct update of resources. Conflicting parts of resources are then enhanced with the version from the repository and marked by special character sequences (>>>>>>>>>>>>>>, e.g.). The developer has then to go through the resources and find the places where conflicts have occured. This can be time consuming and possibly introduce errors in the code. Hence, it should be avoided whenever possible.

To improve this situation, the "Team Synchronization" perspective offers a solution. The general procedure for performing updates and commits should be as follows:

1. Select the resources that sould be updated/commited in the "Project Explorer" view of Eclipse.
2. Rightclick on one of the selected resources and select "Team" and "Synchronize with Repository".
   → the perspective changes to "Team Synchronization"
3. Wait until the progress bar (in the lower right) has completed.
4. Open (if not opened) the "Synchronize" view.
5. Select the button "Conflicts Mode" within the "Synchronize" view
   → all conflicting resources are shown.
6. Resolve all conflicts of files that should be updated or commited.
7. Click on the button "Incoming Mode" to see possible updates of resources.
8. Perform all desired updates.
9. Click on the button "Outgoing Mode" to see possible commits of resources.
10. Before doing any commits, *make sure that the version of the code you posess is functional*. This is especially important, when you have retrieved updates before or if you have resolved conflicts. Other developers might use different compile flags and possibly have overlooked errors in the code that you will discover when compiling or running test cases. Resolving conflicts is error-prone, even with the help of Subclipse, and makes checking the code a necessary task before any commit. After you have checked the correctness of the code, perform all desired commits.

Updates and commits can be simply done by selecting the resources in the "Synchronize" view, rightclicking on one of the selected resources and selecting "Update" or "Commit...". How to solve conflicts and to check what parts of the resources will be updated or commited is explained in the next subsection.

Back to Contents.

3.7 Compare view

In order to see the conflicts, updates and commits of a resource, double click on the resource. A compare view opens, which shows the local version of the resource on the left hand side, and the repository version on the right hand side. Conflicting parts are highlighted in red, updates in blue, and commits in grey. By clicking the button "Copy all non-conflicting parts from right to left", all updates will be retrieved. Conflicts and commits are remaining. When clicking on the box of a connecting line of two highlighted code snippets (for updates and conflicts), the version of the repository will replace the local one in case of an update. In case of a conflict, the version of the repository will be *added* to that of the local version. This means, that in case of a conflict, careful by-hand copying of the parts to be kept is necessary. After you have modified your local version of a resource in the compare view, don't forget to save the resource. If you have resolved all conflicts of a file to your satisfaction (this does not necessarily mean, that the compare view does not show any more conflicts), rightclick on the resource in the "Synchronize" view and select "Mark as Merged". The file will then go into the category "commit" or "update".

Back to Contents.

4 Programming Prerequesits

The following concepts are employed in the source code of preCICE, and should be clear to the developer before starting to modify and extend the source code of it.

Back to Contents.

4.1 C++ programming concepts

- Classes
- public, protected and private class scope modifiers
- Derived classes by public inheritance
- Abstract classes
- Virtual methods
- Pure virtual methods
- Copy and assignment constructor
- Memory allocation and deallocation with new and delete
- The keyword const for methods and variables
- Poiners and references
- Generic programming with templates (classes and methods)
- Template specialization
- Typedefs
- Enumerations
- Static method variables
- Namespaces
- Definition, Declaration and Predeclaration of classes and methods

- Shared pointers (smart pointers)

Back to Contents.

4.2 Software Pattern

- Visitor
- Factory method
- Template method
- Proxy

Back to Contents.

4.3 Concepts taken from Peano (see Peano tutorials)

- Configuration concept
- Unit and integration tests
- Logging
- Assertions

Back to Contents.

4.4 Other used language features

- C++ standard template library containers (vector, map, list)
- Boost C++ library (smart pointers, tuples, variant)
- XML files

Back to Contents.

5 Coding Rules and Style

The coding rules describe the set of rules on how to write source code and limit the application of the C++ language features. The coding style describes the layout of the source code. Both sets of rules are very important in a multi developer project to create a high quality source code.

Most important: The following subsections do not give many examples of how the actual source code can look like. For examples check the source code of preCICE.

Back to Contents.

5.1 Rules for C++ language features

Many of the rules are taken from the book "Effective C++" and "More Effective C++" from Jack Meyers, Addison Wesley. These two books should be definitely read by anyone who wants to program in C++. While the following rules do only describe the "how" to code, the books do also explain why these rules make sense.

- All sources are nested into the namespace "precice".
- Sources which are not meant to be in the API of preCICE are nested into further namespaces.
- All class variables have to be made private, with the exception of static const variables, which can be protected or public.
- Functions returning a class variable do return const references to them, if the variable is of user-defined type (objects). They return the variable by value if it is of builtin type (int, double, char).
- The variable modifiers `short`, `long` and `unsigned` are used scarcly.
- Builtin type parameters (int, double, char) are passed by-value to a function. The `const` modifier is not used for pass-by-value parameters.
- User-defined type parameters (objects) are passed as `const` reference to functions, if their state is not intended to be changed, and as reference parameters, if their state is changed by the function.
- Member functions that do not alter the state of an Object are made const
- Use of references is preferred over that of pointers.
- If a function returns one result, it should do so via a return value.

```
void computeResult ( int * result )
{
  *result = /* computation */;  // WRONG!! Pass variable by return.
}
int computeResult ( void )
{
  return /* computation */;   // RIGHT
}
```

- If a function returns several results:
  - A `struct` or `std::tr1::tuple` can be returned holding the results.
  - The results can be written into non-const reference parameters given to the function.
  - Pointers can be used only for objects (not for built-in types), and only if the return value is created with the `new` operator inside of the function. The class defining the function must take care of deleting the memory allocated with `new` at some suitable point (Destructor, e.g.). A better way is to use smart pointers instead.
- Non-virtual functions are not overwritten in subclasses.
- Preprocessor macros are used only if it is not possible to use other language features to achieve the same functionality.
- Include guards are used to ensure unique includes of header (.hpp) files.
- Pre-declarations of user-defined types should be used in header files, if possible.

Back to Contents.

5.2 Rules for code layout

- The source code is placed into files ending with .hpp and .cpp. The .hpp files contain declarations, template and inline definitions. The .cpp files contain all other definitions.
- Coding and documentation is done in English.
- No tabulators are used in the source code. Instead, only whitespaces are used (Eclipse allows to configure this).
- The indentation depth for code is 2 whitespaces per level.
- The upper limit of characters (including whitespaces) in a line is 80 (Eclipse allows to display a vertical line at a given character number).
- The hungarian naming convention for variables is **not** followed, i.e. variables do not start with the description of its type (dVariable for doubles, or iIndex for an int, e.g.). However, starting letters might be used to signal special purposes of variables, such as loop counters starting with an i.
- Virtual function declarations in derived classes must always be enhanced by the `virtual` keyword.
- Preprocessor defines and macros are written in capital letters and start with `PRECICE_`. Words are separated by underscores (`PRECICE_TWO_POWER_DIM`, e.g.).
- Variable names should give an idea about the meaning of the variable. Abbreviations should be avoided. The same variable names should be used for the same tasks. In precice, for loop variables are usually named `i`, `j`, `k` or `l`, iterator names start with `iter`, counters end with `Count` and so on.
- Local Variables are defined as close to their point of use as possible.
- Non-const variable names start with a lower case letter. They do not use undersorces, but connect words directly with captial first letters (`anExampleVariable`, e.g.).
- Class member variable names start with an underscore (`_memberVariable`, e.g.).
- Struct member variables do not start with underscore.
- Names of userdefined types, i.e. classes, structs, unions, enums start with Uppercase letters.
- If a class is abstract and does contain no definitions, its name starts with a capital `I` for interface.
- If a class is abstract and does contain definitions, its name starts with `Abstract`.
- Method names start with a lower case letter.
- The void keyword is not used to highlight empty method parameters.
- Namespaces consist of one word, written in lower case letters.
- Single (or few) line class methods can be defined directly in the class' body.
- A class defining virtual methods has to define a virtual destructor, even if the destructor does nothing.

Back to Contents.