Perfect Developer User Guide

Version 4.0, May 2009

Disclaimer

The information in this publication is given in good faith but may contain errors and omissions. The contents of this document and the specifications of Perfect Developer are subject to change without notice. Where third-party products are mentioned, no endorsement of those products is implied.

Contents

Introduction

1. Summary
   1.1 Overview
   1.2 Hardware
   1.3 Source and Output Files

2. Compatible tools
   2.1 Source code editor
   2.2 Source code control system with project build facilities
   2.3 C++ compiler
   2.4 Java SDK

3. Using Perfect Developer
   3.1 Overview of Perfect Developer projects
   3.2 Adding and removing files from a project
   3.3 Using the context sensitive right-click files menu
   3.4 Changing project settings
   3.5 Using multiple configurations
   3.6 Building and verifying a project
   3.7 Error output
   3.8 Importing UML class diagrams
   3.9 Changing Perfect developer global settings

4. Building a console application
   4.1 Main program
   4.2 Compiling and linking the generated code

5. Hints and Tips

6. Command Syntax
   6.1 General
   6.2 Environment variables
   6.3 Form of command line options
   6.4 Error message options
   6.5 Code generation options
   6.6 Source code options
   6.7 Verification options
   6.8 Cross-reference options
   6.9 Miscellaneous options

7. Return Codes

8. Compiler Files
   8.1 Compiler program file
   8.2 Built-in declarations file
   8.3 Math rules file
   8.4 Library and run-time files

Introduction

The principles

By expressing the user's requirements and system specification in a high-level description language, called Perfect, the Perfect Developer tool is able to use automated reasoning technology to understand the descriptions and to verify that the user requirements will be met. Perfect Developer goes on either to generate code from the specification automatically, or to verify that developer-supplied code implements the specification perfectly.

Intelligent verification

Perfect Developer uses a powerful automatic inference engine and theorem prover to reason about the requirements, specifications, and code. It is not necessary for developers to have advanced mathematical knowledge.

Fast prototyping and refinement
Perfect Developer will generate most of the code directly from the specification, facilitating the rapid construction of a fully-functional prototype. This prototype can then be evaluated to ensure that requirements have been correctly captured. Typically, only a fraction of the prototype code will have to be replaced to obtain the performance required of the final product.

Publications available

The intention of this User Guide is to provide guidance on the operation of the Perfect Developer Project Manager. For an outline of the Perfect language itself, please refer to Getting Started with Perfect. The full definition of Perfect is available in the Perfect Developer Language Reference Manual.

1. Summary

1.1 Overview

Perfect Developer from Escher Technologies Limited does the following:

- Generates ANSI C++, Java, or C# source code from Perfect Developer notation (Perfect) specifications and refinements
- Generates verification conditions for Perfect software systems
- Automatically proves the vast majority of true verification conditions
- Produces output files outlining the proofs

It is intended to be used in conjunction with a third-party editor and a C++, Java or Ada 95 compiler. Perfect Developer runs on industry-standard PC hardware under both Linux and Windows, including Windows 2000 and Windows XP.

1.2 Hardware

The recommended hardware is a PC to the following specification:

- Intel Pentium or AMD Athlon or compatible processor at 1GHz or higher clock speed
- 256Mb or more RAM
- At least 100Mb free hard disk space

Verification of large projects requires additional memory. Error-correcting memory should be fitted if Perfect Developer is used to build critical systems.

1.3 Source and Output Files

Source files may be prepared with a standard text editor (see section 2 on Compatible Tools). The language recognized is defined in the Perfect Developer Language Reference Manual from Escher Technologies Limited.


2. Compatible tools

The following is for guidance only. Escher Technologies accepts no responsibility for the performance of third-party products.

2.1 Source code editor

Perfect Developer does not have its own source code editor. Instead, the Project Manager works with a source code editor of your choosing.

The source code editor should have provision for displaying and printing Perfect source code with syntax highlighting. Many source code editors have configurable syntax highlighting and some also provide for the definition of language structure templates to assist in the rapid entry of program text.

Evaluation versions of some popular editors are supplied with Windows versions of Perfect Developer. Also included are customization files for these and other Windows and Linux editors.

2.2 Source code control system

The facilities required of the source code control system will depend on user requirements. However, most source code control systems include project build facilities, usually in the form of a make or build program. Some are also supplied with a program for determining file interdependencies and updating the make file appropriately. Such a program should ideally be configurable to recognize the import directives of Perfect.

2.3 C++ compiler

When the chosen target language is C++, an ISO-compatible C++ compiler for the target machine is required. Unfortunately, not all present-day compilers are
sufficiently compliant with the standard. The following compilers have been successfully used with large Perfect Developer projects:

- Microsoft Visual C++ 2005
- Microsoft Visual C++ 2008
- GNU gcc version 4.3

When compiling C++ code produced by Perfect Developer, ensure that your compiler is configured to the following settings:

- Strict ISO or ANSI compatibility (i.e. disable non-ISO extensions)
- Enable exception processing
- Enable run-time type information
- The directory containing the file Ertsys.hpp must be included in the compiler search path for include files

Further information on C++ compiler compatibility may be found on the support site of the Escher Technologies web site.

2.4 Java SDK

If compiling to Java, the Java 6 SE SDK is required. This can be downloaded from Sun Microsystems, Inc.

3. Using Perfect Developer

The simplest way to use Perfect Developer is via the Project Manager, installed automatically with the compiler itself. To start the Project Manager under Windows, go to the Start menu and select the Perfect Developer item from the Programs -> Perfect Developer menu. Under Linux, the Project Manager can be started by executing the file /opt/escher/bin/perfectdeveloper.

When started, the application will appear as shown below (the Project Manager is shown here running under Windows; under Linux the appearance will be different, but the functionality is identical):

The most important parts of the application window are the files area and the results area.

The files area displays the full paths to the files that are contained in the current project.

The results area is used to capture the output from the tool which is invoked in response to the user selecting one of the Build, or Check items from the Project menu. Following a failed build, some lines in the output window will normally contain error messages generated by the tool. Double-clicking on such lines results in the configured editor being requested to open the offending source file at the relevant position.

The toolbar provides instant access to the most important commands from the menu.

3.1 Overview of Perfect Developer projects

The Project Manager allows the user to create Perfect Developer ‘projects’ that contain a record of all the Perfect source files for an application, and the tool settings that will be used when the user wishes to compile, verify or generate a cross-reference listing for the project. Perfect Developer project menu options appear on the File menu, as detailed below:
New project...

This item prompts the user for the name and location of a new Perfect Developer project file, and creates it with an empty file list and some default tool settings. See the next section (3.4) for details about how to change the project settings.

Open project...

This item prompts the user for the name and location of an existing Perfect Developer project file, and opens it in the application. If the user wishes to guard against the risk of making unwanted changes to the project, the 'Open as read-only' check box at the bottom of the file-open dialog may be checked, and the save items detailed below will not be available for the duration of the project session (this feature is not available under Linux).

Save project

This item prompts saves the currently open project to its existing name. Note that the item is only available if the project has actually been changed, i.e. if files have been added or removed or if project settings (as distinct from application settings, under the Options menu) have been altered.

Save project as...

This item saves the currently open project to a new name, prompting the user for the new name.

Open common proof file

This item provides a mechanism to open a configured common proof output file in an appropriate external viewer. Note that if the project settings are such that the verifier is not configured to produce a common proof file or if it is, but the file does not exist, then this menu item will not be available.

Open common unproven file

This item is essentially functionally the same as the Open common proof file menu item, except that it operates on any common unproven file instead of a proof file.

Open error file

This item provides a mechanism to open a configured error output file in the configured editor. Note that if the project settings are such that an error file is not generated, or if it is, but the file does not exist, then this menu item will not be available.

Exit

This item quits the application, prompting the user to save the current project if it has been changed since it was last saved or opened.

3.2 Adding and removing files from a project

Once a project is open, existing Perfect source files may be added, new source files created, and unneeded files removed as required, using either the toolbar buttons provided or the corresponding items in the Project menu. When creating a new file, the following dialog is displayed, allowing you to choose whether to create skeleton code and/or open the file for editing:

When creating a new file, a file name, directory and author name must be provided. If the Create with skeleton code option is checked, the file will be generated with a generic class skeleton. If the Automatically open option is checked, the file will be opened for editing upon pressing OK. Alternatively, by selecting Perfect main entry point, a file 'main.pd' will be created in the chosen directory containing a schema with the correct signature for the entry point into the Perfect code.

3.3 Using the context sensitive right-click files menu

It is often convenient to be able to perform operations upon a chosen set of source files. Perfect Developer provides a mechanism for doing this, via the context-sensitive files menu, activated by right-clicking on a file selection in the files list. The selection is made using the mouse left-click (multiple files may be
selected by using the 'Control' or 'Shift' keys, as is usual with Windows applications). The image below shows the right-click menu in action:

![Right-click menu](image)

The functionality provided by the items on the right-click menu are described below (note that the Open proof file and Open unproven file items are only available when a single file has been selected):

**Edit**
This feature allow a selection of files to be opened in the currently configured editor.

**Build**
This feature provides a mechanism that allows a selection of source files to be built. Note that the files built using this mechanism will be built unconditionally, i.e. no check will be made to see whether they actually require building.

**Check**
This feature allows a dummy build to take place, where all syntactical and semantic checks are made, but no code is generated. Users may find it useful during the development of large systems.

**Verify**
This feature provides a way to verify only a selection of source files instead of verifying the entire project, which is the default behavior when choosing the Verify item from the Build menu. This feature is invaluable when developing a particular file in a project containing a large number of source files.

**Remove**
This feature allows a selection of files to be removed from a Perfect Developer project. This is an alternative to removing files using the mechanism described in the section 3.2 Adding and removing files from a project.

**Open proof file**
This item provides a quick mechanism for accessing proof output files generated by the verifier. The feature is very useful because the project settings provide a number of ways to specify where proof output files are generated and what their names are. Selecting this menu item automatically determines the name and location of the generated proof file associated with the source file selected, and opens it in an appropriate external viewer. Note that this viewer is not the configured editor, but is either a web browser (for HTML format proof files), a text editor, or a TeX viewer. Note also that if the project settings are configured such that separate proof output files are not being generated, this menu item will not be available (it will be grayed out).

**Open unproven file**
This item uses the same logic as the Open proof file item described above, but applies to unproven output files instead of proof output files.

**Properties**
This item is not currently implemented.

### 3.4 Changing project settings

The current settings for a project may be viewed or changed by selecting the Settings item from the Project menu, or by clicking on the Settings tool button as shown below:
Either of these actions will open a tabbed dialog box (also known as a 'property sheet') with a tab for each of the following classes of settings:

**Error reporting**

This tab contains general settings that tell the system what to do with error messages; what should be included in the messages; the maximum number of errors to report before terminating, etc. The tab is shown below:

Checking the Show secondary errors box affects the treatment of errors when a Build, Verify, Check or Cross Reference operation is performed. Normally, error messages that are likely to have been caused by other errors are suppressed, making it easier to understand the error messages. These secondary errors will not be suppressed if the box is checked. We advise you to check this box only if a message is generated stating that errors were detected but all of them were suppressed.

**Verification**

This tab contains settings pertaining to the verification of the project.
The current options selected for any proof or unproven files generated are displayed. These may be changed by using the Change buttons, which bring up a dialog similar to this:

**Code generation**

This tab contains settings pertaining to code generation, and includes settings for which runtime checks should be generated in the output code.
Some standard combinations of settings are available from the code generation dialog. These are selected using the radio buttons in the 'runtime checks' section.

**Build**

This tab allows the user to specify operations that should be performed before and after building the Perfect source files. For example a compiler for the generated code may be invoked as a post-build step. The tab is shown below:

Prior to executing pre- and post-build commands, Perfect Developer sets up the following environment variables:

- **PD_CPP_INCLUDE_PATH** - The path to the directory in which the C++ header files describing the Perfect Developer run-time library were installed
- **PD_CPP_LIB_PATH** - The path to the directory in which the C++ run-time library files were installed
PD_JAVA_LIB_PATH  The path to the directory in which the Java run-time library files were installed
PD_JDK_BIN_PATH  The path that you have indicated contains Java tools (see Changing Perfect Developer global settings)
PD_PACKAGE  The Java package name as configured on the Code generation tab

Miscellaneous

This tab contains various settings and is shown below:

The Quiet level option selects the extent to which messages are displayed in the Results window indicating the progress of the tool. The progress of the verifier is always output on the status line of the Project Manager, regardless of this setting.

The Source file tab width setting is used to calculate column numbers when generating diagnostic messages. This should be set to the same value that your editor assumes for .pd files.

3.5 Using multiple configurations

The Project Manager allows for multiple configurations of a project to be defined. Each configuration has its own settings. The configuration manager is accessed via the Configurations item on the Project menu as follows:

This opens the configuration manager:

From the configuration manager the following operations can be performed:
Copy
This copies the currently-highlighted configuration, prompting the user for a
name for the new configuration, and then opening the ‘settings’ property sheet to
allow settings to be defined in the new configuration.

Rename
This prompts the user for a new name for the selected configuration.

Remove
This removes the currently-highlighted configuration (this is only available if
more than one configuration is present).

Select
This selects the currently-highlighted configuration as the active configuration
for subsequent operations.

Close
This closes the configuration manager.

The currently-active configuration is shown below the menu bar in the Project
Manager main window. Clicking on this allows the user to select a new active
configuration.

3.6 Building and verifying a project

The Build menu is shown below, and contains all the options for building and
verifying the current project:
These items perform the following actions when chosen:

**Build**

This causes any files in the project which are determined to have been changed since the last build to be rebuilt (i.e. for code to be generated in the language specified in the code generation tab of the project settings).

**Re-build all...**

This builds all files in the project, regardless of whether they have been changed or not. This is useful, for example, if a change has been made to the project settings which affects code generation.

**Batch-build...**

This allows the project to be build using several different configurations without any further user intervention. See section 3.5 for information on setting up multiple configurations.

**Check**

This performs syntax checking, type checking and basic static analysis on all the files in the project, without generating any code. This can be used to quickly check for simple errors, or to see whether a change to one file in the project has consequences for other files.

**Verify**

This will attempt to verify all the files in the project using the options set in the Verification tab of the project settings. Progress will be displayed on the status line during verification, as this can take some time for a large or complex project.

**Cross-reference...**

This will generate a text file listing all declarations in the current project and the locations at which they are referenced. The dialog displayed after choosing this option offers the choice of listing just those declarations that are not referenced within the current project, and the option to automatically open the listing file in the editor when the cross-reference is complete.

### 3.7 Error and warning messages

After performing a build, verify, check or cross-reference, errors, warnings and informational messages may be displayed in the Results pane. Double clicking on a message will open the file for which the message was produced in the editor, at the relevant line and column (if the editor supports this - see section 3.8 for information on configuring the editor). Right-clicking on the message will bring up a menu as shown below:
Depending on the error message, some of the options may be unavailable (grayed out). The options have the following meanings:

**Go to error location**
This performs the same action as double clicking, and opens the editor at the location of the error.

**Go to referenced declaration**
This will open the editor at the location of a declaration that is relevant to the source of the error. For example, it may be the location of the declaration of a precondition that cannot be proved to be satisfied.

**Go to error context**
This option is available when a method defined in a class has failed to be verified in the context of one of its descendents. In this case it opens the editor at the declaration of the class in which the method is being verified.

**Go to proof information**
This option is available when the verifier has produced proof or unproven output which contains information about why the verification condition could not be proved, provided that a viewer can be found that is appropriate to the chosen output format (e.g. a browser when the format is HTML). Selecting the item will cause the relevant proof or unproven file to be opened at the appropriate location.

### 3.8 Importing UML class diagrams

*Perfect Developer* contains the facility to generate class skeletons from a UML diagram described in XMI format. This is performed by selecting *Import UML...* from the *Project* menu, or by clicking on toolbar button:

The user will then be asked to select an XMI file to import. If the selected file does indeed include a UML class diagram, the following dialog will be displayed:
From this dialog the user can choose which classes should be imported, which directory the generated *Perfect* files should be placed in, and whether to generate a separate file for each class (in which case the class name is used as the file name), or if all the classes should be placed in a single specified file.

### 3.9 Changing *Perfect Developer* global settings

The Project Manager has some global settings that are independent of any particular project. These settings are accessed through the *Options* menu:

**Editor...**

This item allows you to configure *Perfect Developer* to use your preferred editor when opening *Perfect* source files.
As well as setting the name of the executable file that starts your editor, you must select the editor type so that Perfect Developer knows how to pass it the correct command-line parameters. If your editor is not one of the types listed, select other. You will then need to manually enter the templates for the editor command line parameters.

The Open specified file(s) box allows you to define an editor command line tail that instructs the editor to open one or more files. Use the string '$f' is used as a place marker for the filename(s), or '$F' if double quotation marks should always be used around each filename.

The Open specified file and set cursor at line/column box allows you to define an editor command line tail that instructs the editor to open a single file and move the cursor to the specified position. As well as using '$f' or '$F' to stand for the filename, you can use 'sl' to stand for the line number and 'sc' to stand for the column number. If you find that the cursor seems to move to the wrong column, check that the tab character width configured in the editor tab Source file tab width to one. If your editor does not support moving to a particular column at start up, just have it move to the correct line.

Job priority...

This item allows the user to configure Perfect Developer to run chosen tasks at low priority, i.e. in the background, so that the user's machine remains responsive to other tasks. The dialog is shown below:

A checked box indicates that the associated task will be run at low priority.

Build management...

This item allows the user to change the logic used by Perfect Developer when carrying out pre- and post-processing during a build.

The first field of this dialog relates to the date stamp placed on the generated .cpp or .java code files, which affects which files will be included in a Build operation. If this is set to Source file date/time then Perfect Developer can detect when it needs to generate new code from Perfect source files even when the time stamp on the source files moves backwards in time (e.g. when regressing to an earlier version). Otherwise, new code will be generated only if the source file time stamp is later than the generated code file time stamp.

The first check box specifies whether Perfect Developer should retain the date/time on declaration files (the .hpp files, if the target language is C++) that are observed not to have changed following the building of a Perfect file. This prevents the external compiler used to compile the generated code from needlessly re-compiling source files that include that declaration file.

Checking the second box will mean that if the user requests a Build but there are no out-of-date files, the user will be asked if the post-build step should be
performed anyway. If the box is not checked, the post-build step will only be performed if some files have been rebuilt.

**Miscellaneous...**

This item allows the user to change other features of the operation of the Project Manager.

Checking the 'Enable audible alerts' box causes a beep to be emitted at the end of every build.

The path to the Java tools is used only for setting up the environment variable PD_JDK_BIN_PATH prior to running pre- and post-build commands. Its value is of no consequence unless you are using pre- or post-build scripts that use this variable, such as the ones supplied with the examples.

4. Building a console application

4.1 Main program

To build a console-mode application, the program must define a main schema with the following signature (if the target language is Java this schema must appear in a file called `Main.pd`):

```
schema main(env!: limited Environment, args: seq of string, ret!: out nat)
```

The environment parameter represents the outside world and is required for input/output. The argument list passed to the schema contains the program name and command line arguments (as for a C++ main program). The schema should set the return code `ret` within the postcondition.

Some sample Perfect source files may be found in the `Examples` directory (unless you installed under Windows and chose not to install the examples).

An example 'hello world' project is available in `Examples\HelloWorld\Hello.pdp`. It included configurations to generate both C++ and Java code. Under Windows, the generated C++ can then be compiled using the Microsoft Visual C++ version 6.0 project file `Examples\HelloWorld\HelloWorld.dsp`. Under Linux, you can compile and run the generated C++ code using the Gcc compiler. The generated Java can be compiled using the Java 1.4 SDK from Sun Microsystems.

4.2 Compiling and linking the generated code

If generating C++, it is necessary to compile and link together all the `.cpp` files generated from the relevant Perfect source files, and link with the relevant runtime library file, which can be found in the directory described in the section on Library and run-time files (§4). If using gcc under Linux this can be achieved by using the options `-L/opt/escher/perfectdeveloper/lib/perfectdeveloper` and `-lperfectruntime` for a release build, or `-lperfectruntimeM` for a debug build. If using Microsoft Visual C++ 6.0 under Windows, ensure that library 'PerfectRuntime.lib' for a release build, or 'PerfectRuntimeD.lib' for a debug build, is present in the list of libraries in the link 'input' settings, together with the path in which the library is located (normally C:\Program Files\Escher Technologies\Perfect Developer\Runtime\Lib\Cppli). If generating Java, the file Entry.java in Examples\HelloWorld must be compiled along with the generated files. See the document Creating a Java application using Perfect Developer and JDK 1.4 for more information.

4.3 Runtime checks

Ideally, the program will be completely verified by Perfect Developer and no debugging of the generated code will be necessary. However, options to generate runtime checks are provided in the Code generation tab of the Settings property sheet. Note that some runtime checks may substantially increase execution time (especially loop invariant checks).

If a runtime check fails then the following action is taken:

- If the generated code was C++, a debug breakpoint instruction is executed, so that if the program is being run under a debugger, control will be transferred to the debugger.
- If the generated code is C++ but no debugger is active, an exception will be thrown. The default implementation of the C++ `main` function provided in the runtime library will catch this exception and proceed to display the nature of the failed runtime check and the call stack before aborting the program.
• If the generated code was Java, an runtime error exception will be thrown. The version Entry.java supplied does not catch this exception; the Java virtual machine will typically produce a stack trace before terminating the program.

All generated and runtime C++ files must be compiled in “debug mode” (i.e. without the NDEBUG macro defined) for this support to be enabled, and should be linked with the debug version of the runtime library (PerfectRuntimeD instead of PerfectRuntime).

5. Hints and Tips

• Projects involving a large number of interdependent files are best built by building all necessary files in a single command. Each file will only be parsed and bound once, saving time. However, if a single file has been heavily modified and is liable to contain parsing errors, it may be quickest to try building it by itself.

• When performing verification, the theorem prover may spend a great deal of time trying to prove a difficult verification condition (if the condition is in reality unprovable). Use the time limit and boremom limit settings to limit the amount of effort spent on each verification condition.

• When running PTool from a batch file under Windows NT, command line options passed as batch file parameters are liable to have "=" characters replaced by spaces. To avoid this problem, enclose the command line option in double quotation marks.

• It has been observed that Windows NT may behave unexpectedly if it runs out of virtual memory. For this reason, and because the compiler may use large amounts of memory during verification, the total paging file size should be configured to 1000Mb or more (this setting may be accessed from the desktop via "Start - Settings - Control Panel - System - Performance").

• Windows NT Workstation and Windows 2000 Professional are not effective at balancing CPU time between multiple tasks (typically, one computation and memory-intensive task will virtually monopolize the processor). When using the computer for concurrent verification and editing or other interactive tasks, either use a multiprocessor system, or change the Project Manager options to run the verification task at low priority.

• Suggestions on how to track down the reasons for verification failures may be found in the tutorial on our web site.

6. Command syntax

6.1 General

As well as running the code generator and verifier from the Project Manager, you can also run it from the command line, or from a batch file or shell script, or from your editor (it is able to invoke external programs).

The command line tail following the command name PDTtool may comprise one or more items. Items beginning with "-" are parsed as command line options to control the operation of the compiler. All other items are treated as the names of source files to be compiled.

Any command line item that contains embedded spaces must be enclosed in double quotation marks.

Each command line option overrides any other options earlier on the command line with which it conflicts.

6.2 Environment variables

If the environment variable PERFECTOPTIONS is defined, items in the value of this variable are parsed before the command line tail is processed.

The environment variable PERFECTPATH is registered at installation time; it holds the path (including trailing backslash) to the base directory in which Perfect Developer was installed.

6.3 Form of command line options

Following the leading "-" character, a command line option comprises one, two or three characters, sometimes followed by an equals-sign and a value. The equals sign is optional if the value is numeric.

Multiple options beginning with the same letter may be combined by factoring. The initial letter is followed by "(" and a comma-separated list of options (leaving out the initial "," and the first letter) and finally closed by ")". For example, 
-eme=1000,a) is equivalent to "-em=1000 -ea".

In the following descriptions, # represents a digit or digit string and $ represents a character string.

6.4 Error message options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ea</td>
<td>Display all error messages (normally, errors that are believed to be caused by other errors are not reported)</td>
</tr>
<tr>
<td>-ef=$</td>
<td>Write information, warning and error messages to file $</td>
</tr>
</tbody>
</table>
6.5 Code generation options

- **-g** Enable code generation
- **-gd** Enable generation of all run-time checks - equivalent to \(gd(A=1,a=1,c=4,i=1,l=3,o=1,p=1,r=1,t=1)\)
- **-gdA** Select generation of post assertion checks (0 = disabled, 1 = enabled; default 0)
- **-gdA** Select generation of embedded assertion checks (0 = disabled, 1 = enabled; default 0)
- **-gdc** Select generation of class invariant checks (0 = disabled, 1 = generate empty class invariant body for use by other files, 2 = generate full class invariant body but no checks, 3 = generate full body plus checks when returning from a constructor or modifying schema, 4 = generate full body and full checks; default 0)
- **-gdi** Select generation of checks for no guard true in conditional expression, conditional postcondition or "if..fi" statement (0 = disabled, 1 = enabled; default 0)
- **-gdl** Select generation of loop checks (0 = disabled, 1 = check variants, 2 = check invariants, 3 = check both; default 0)
- **-gdo** Select generation of postcondition checks (0 = disabled, 1 = enabled; default 0)
- **-gdp** Select generation of precondition checks (0 = disabled, 1 = enabled; default 0)
- **-gdr** Select generation of recursion variant checks (0 = disabled, 1 = enabled; default 0)
- **-gdt** Select generation of type constraint checks (0 = disabled, 1 = enabled; default 0)

- **-gi** Set indentation of nested blocks in generated code to # characters (default 4)
- **-gk** Use package name $ when generating Java
- **-gp** Generate output file in path $, relative to the source file path (default is the path of the source file)
- **-gr** Treat the string $ as a semicolon-separated list of reserved words in the target language (note that each use of this option adds words to the reserved word list, i.e. previous instances of this option are not overridden)
- **-gs** Generate output in layout style # (0 = newline before ';', 1 = Kernighan & Ritchie style; default 0)
- **-gv** Generate code for target language compiler type $ ($ = "ISO", "Borland", "Gcc" or "Microsoft"; default "Microsoft")
- **-gw** Set maximum width of output lines to # (default 100)
- **-gx** Specify executable type (0 = generic, 1 = Win32 console; default 1)

6.6 Source code options

- **-sh** Read built in declarations from file $ (default builtin.pd in the directory from which PDTool.exe is loaded)
- **-sp** Enables use of precompiled imports (this option is not available in the education / evaluation edition)
- **-sr** Read rules from file $ (default rubric.pd in the directory from which PDTool.exe is loaded)
- **-st** Assume tab stops every # character positions when reporting the location of an error (default 4)

6.7 Verification options

- **-v** Enable verification
- **-vb** Set prover boredom limit (in seconds) per verification condition (0 = no limit; default 15)
- **-vl** Set verification method (0 = use rewriter only, 1 = use rewriter and prover; default 1)
7. Return codes

The compiler terminates with one of the following return codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors or warnings were reported</td>
</tr>
<tr>
<td>1</td>
<td>No errors were reported but one or more warnings were reported</td>
</tr>
<tr>
<td>2</td>
<td>One or more errors were reported</td>
</tr>
<tr>
<td>3 or higher</td>
<td>A fatal or internal error was detected</td>
</tr>
</tbody>
</table>

8. Compiler files

The paths given for the following files under Windows are relative to the directory into which Perfect Developer is installed. The paths for the files under Linux are absolute.

8.1 Compiler program files

The file bin/PDtool.exe under Windows, or /opt/escher/perfectdeveloper/bin/pdtool under Linux, holds the compiler & verification tool.

8.2 Built-in declarations file

The file bin/builtin.pdc under Windows, or /opt/escher/perfectdeveloper/builtin
/builtin.pdc under Linux, holds declarations needed for the correct functioning of the compiler.

8.3 Math rules file

The file binrubric.pdc under Windows, or /opt/escher/perfectdeveloper/builtin/rubric.pdc holds information used when performing verification.

8.4 Library and run-time files

Runtime libraries are supplied for both C++ and Java target languages.

8.4.1 C++

The file runtime/include/cppErtsys.hpp under Windows, or /opt/escher/perfectdeveloper/include/Ertsys.hpp under Linux, contains a directive to include a number of C++ header files. The Perfect Developer tool generates a directive to include Ertsys.hpp in the generated C++ files.

The runtime library files for release and debug builds respectively are supplied in the files runtime/lib/cppiPerfectRuntime.lib and runtime/lib/cppiPerfectRuntimeD.lib under Windows, or /opt/escher/perfectdeveloper/lib/perfectruntime.a and /opt/escher/perfectdeveloper/lib/perfectruntime-debug.a under Linux. One of these files must be included in the linker library list.

8.4.2 Java

The file runtime/lib/java/PerfectRuntime.jar under Windows, or /opt/escher/perfectdeveloper/java/PerfectRuntime.jar under Linux, contains the runtime classes required to build and run generated Java files. The alternative files runtime/lib/java/PerfectRuntimeD.jar (Windows) and /opt/escher/perfectdeveloper/java/PerfectRuntimeD.jar (Linux) perform run-time precondition checking within library methods.

© 2001-2009 Escher Technologies Limited. All rights reserved.