CDT 2.0 Managed Build System FDS

This document describes the design of the new features for the managed build system.

Revision History

Date	Revision	Description	Author
1/15/2004	0.1	First draft to be reviewed internally	Sean Evoy
1/19/2004	0.2	Revised draft	Sean Evoy
1/27/2004	1.0	First version	Sean Evoy

Table of Contents

1 Introduction ... 2
  1.1 Release Goals ... 2
  1.2 Terms ... 3
2 Requirements ... 3
  2.1 Requirements ... 3
3 Design Overview ... 4
  3.1 UI Elements ... 4
    3.1.1 Tool Command Editor (C1) ... 4
    3.1.2 Setting Options for All Configurations (C2) ... 5
    3.1.3 Converting a Project from Managed to Standard (C3) ... 6
    3.1.4 Clone Existing Configuration (C8) ... 8
    3.1.5 Support Variables in Path Specifications (C11) ... 9
    3.1.6 New Project Wizard Modifications (C13) ... 11
    3.1.7 Changing Project Types (C15) ... 13
  3.2 Unicode Support and G11N ... 13
    3.2.1 Unicode in the Manifest and Project File ... 13
    3.2.2 Build Property Page ... 13
    3.2.2 Unicode in Generated Makefile ... 14
4 Design Discussion ... 14
  4.1 Modifying the Tool Command (C1) ... 14
    4.1.1 Use Cases ... 14
    4.1.2 Limitations ... 15
  4.2 Setting Build Options Simultaneously (C2) ... 15
    4.2.1 Use Case ... 16
    4.2.2 Limitations ... 16
  4.3 Converting a Project from Managed to Standard (C3) ... 16
    4.3.1 Use Case ... 16
    4.3.2 Limitations ... 17
  4.4 Refactor the Current Default Manifest (C4) ... 17
    4.4.1 Limitations ... 17
  4.5 Generic "GNU" Target for Default Toolchain (C5) ... 17

4.5.1 Use Case ... 17
4.5.2 Limitations ... 18
4.6 Automatically Update the Build Information (C6) ... 18
4.6.1 Use Case ... 18
4.6.2 Limitations ... 18
4.7 Update Target Definitions in Manifest (C7) ... 18
4.7.1 Use Case ... 18
4.7.2 Limitations ... 18
4.8 Cloning an Existing Configuration (C8) ... 19
4.8.1 Use Case ... 19
4.8.2 Limitations ... 19
4.9 Debug Configuration is the Default (C9) ... 19
4.9.1 Use Case ... 19
4.9.2 Limitations ... 20
4.10 Remove Gnu Extensions from Generated Makefiles (C10) ... 20
4.10.1 Use Case ... 20
4.10.2 Limitations ... 20
4.11 Support Variables in Path Specifications (C11) ... 20
4.11.1 Use Cases ... 20
4.11.2 Limitations ... 21
4.12 Supply VC++ Toolchain Implementation (C12) ... 21
4.12.1 Use Cases ... 22
4.12.2 Limitations ... 22
4.13 New Project Wizard Should Show All Targets (C13) ... 22
4.13.1 Use Case ... 22
4.13.2 Limitations ... 23
4.14 Implement Tool Inheritance (C14) ... 23
4.15 Automate Changing a Project's Build Goal (C15) ... 23
4.15.1 Use Case ... 23
4.15.2 Limitations ... 23
10 References ... 24

1 Introduction

This document is intended to capture the requirements for the managed build system for the 2.0 release of the CDT.

1.1 Release Goals

The focus of this release is to improve the out-of-the-box user experience. Features such as searching, content assist, and build are all central to the end-user's positive experience with the CDT on Eclipse. The build model will be updated to provide more useful information to these features without the user needing to intervene. We will be incorporating feedback from users on work-flows that are not supported in the current implementation of the user interface. We will also concentrate on fixing the deficiencies that have been identified with the builder.

The list of new features we want to implement is not complete. Many more requirements were identified at the start of the planning process. If other partners wish to take on the features from that list, then this document should be updated.

The long-term goals for the build systems in the CDT are not addressed in this document.

1.2 Terms

This section defines the terms commonly used in this document.

Term	Definition
Build goal	The main files that are produced as a result of a build, e.g. an executable, a shared library, or a static library.
Configuration	A configuration is a line up of tools and settings for the options for those tools as well as other information that configures the project to produce a build goal.
Target	A target in the build sense represents the execution environment for the build output. A target can be described using a number of different aspects including operating system, processor, system libraries, etc. Probably the best analogy would be the --target argument used to configure the gnu tools.
Tool	A utility of some sort that is used in the build process. A tool will generally process one or more resources to produce output resources. Most tools have a set of options that can be used to configure the functionality of the tool.
Toolchain	The main set of tools that produce a build goal for the project.
VC++	An acronym for the Microsoft Visual C/C++ compiler and toolchain.

2 Requirements

Many requirements were identified for the CDT 2.0 managed build feature. The requirements in the section below were determined to be the most important for reaching the goals of the release. If other partners have pressing requirements, it is hoped that they will contribute to the development effort.

2.1 Requirements

The table below shows the list of requirements that have been identified as helping us reach goals of the CDT 2.0 release. Higher priority requirements will be addressed first, but no order is implied by the list.

Number	Requirement	Priority
C1	The user shall be able to change the default tool command at the project and configuration level. The overridden tool command will be associated with a particular configuration for a project. The overridden setting will be stored between sessions for the configuration. The user will be able to reset the tool command back to the default command defined in the plug-in manifest.	P1
C2	The user shall be able to simultaneously change the settings for options in all the defined configurations for a project.	P1
C3	The user shall be allowed to convert a managed make project to a	P1

	standard build project.	
C4	The build model shall be refactored to allow a more compact definition of configurations that derive from a common root.	P1
C5	The UI shall support a "Gnu" target as the default for executable, library, and shared library build goals on all platforms.	P1
C6	The build system shall automatically update the include path, linker commands and defined symbols settings for a project when another project is added to, or deleted from, its list of referenced projects.	P1
C7	Default values for includes paths, defined symbols, and macros shall be specified for all targets to ensure a good out-of-box experience for new users of the CDT. The defaults shall also be sufficient to allow search, content assist, and indexing to work properly.	P1
C8	The user shall be able to add new configurations to a project by cloning an existing configuration.	P2
C9	The debug configuration of all the targets supplied in the plug-in manifest shall be treated as the "default" configuration.	P2
C10	The pattern of the generated makefile shall work with all versions of make. An extension point should be defined to allow a customizable pattern. The extension point would also allow for setting of the default build command.	P2
C11	The build model shall support the use of variables within any path specification.	P2
C12	A default implementation for the Visual C/C++ tools shall be provided along with the gnu implementation that currently ships with CDT.	P3
C13	The user shall be presented with the entire list of defined targets when creating a new project.	P3
C14	The build model shall support tool inheritance in the plug-in manifest.	P3
C15	The user shall be able to change the type of build goal for a project, for example from an executable to a library.	P3

3 Design Overview

3.1 UI Elements

This section contains some mock-ups of the UI elements that may be required for the new or enhanced features in the managed build system. These do not represent final design decisions.

3.1.1 Tool Command Editor (C1)

We can add the capability to change a tool command to the property page of a managed build project. At the moment, when a tool is selected nothing is displayed in the settings edit pane. We can add a label and text entry widget to allow the user to change the name of the command as shown below.

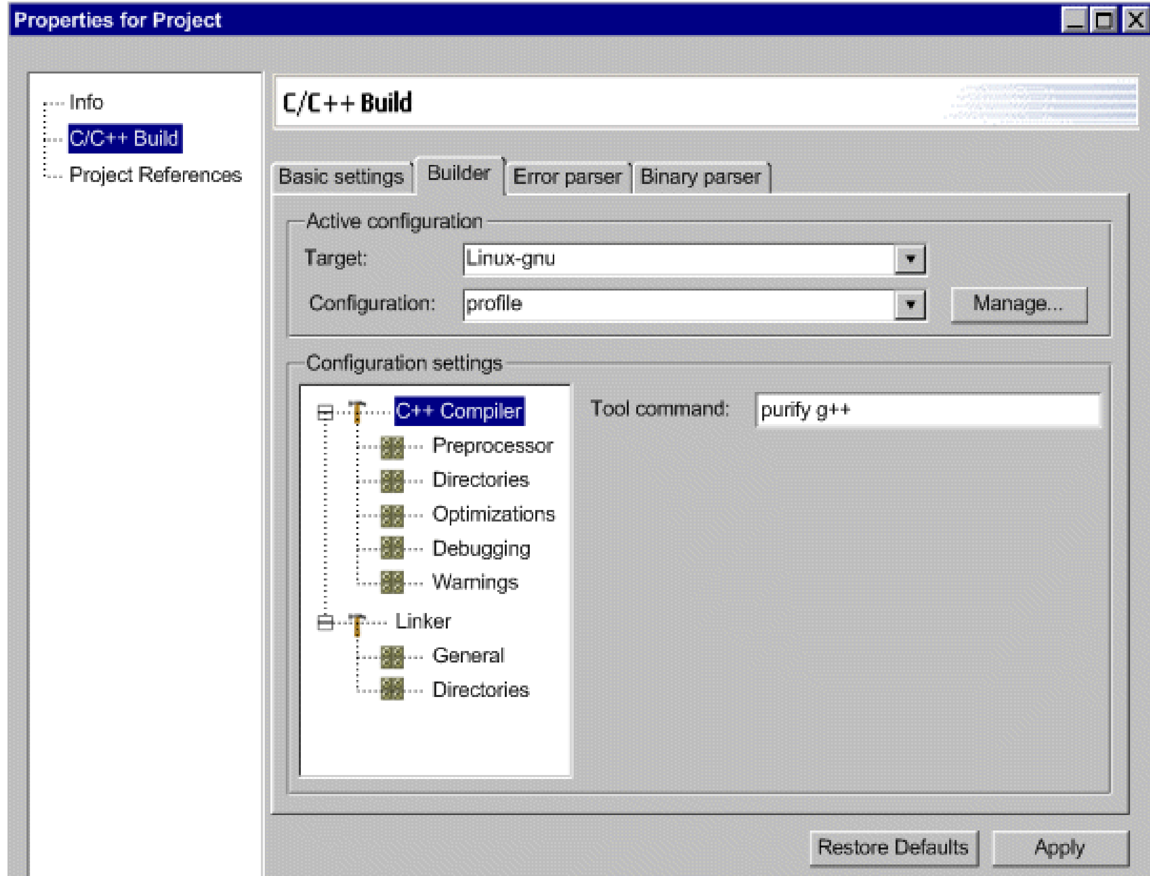

Figure 1 Edit tool command for project

3.1.2 Setting Options for All Configurations (C2)

The figure below shows one possible way to display options for an 'all' configuration. In the case where an option is overridden in one or more of the configurations, the option can be shown as disabled as is the case with the check-box in the example.

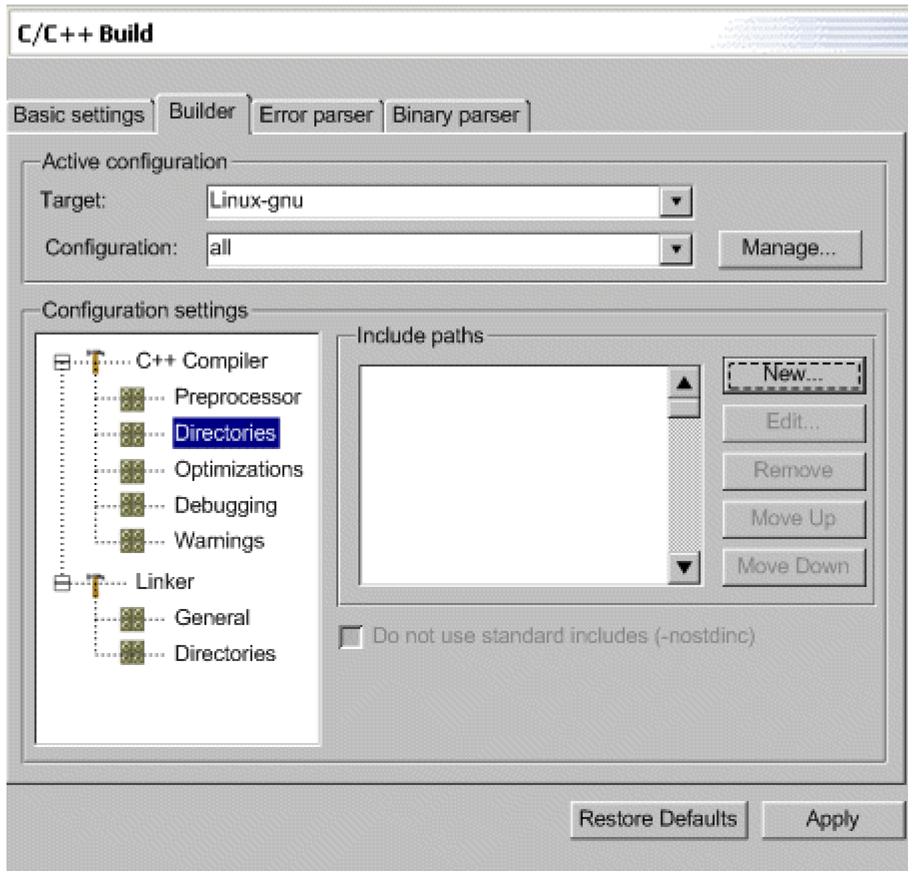

Figure 2 Edit all configuration options

3.1.3 Converting a Project from Managed to Standard (C3)

3.1.3.1 Wizard Option

We will need a conversion wizard to change a project from managed to standard build. A managed build project can have as many configurations as the user wants to add. Each can have different include path and defined symbol lists. The standard build system does not, at least for now. The first page of the wizard will ask the user which configuration they want to copy their settings from. If there is only one configuration, then this page will be displayed with that configuration selected.

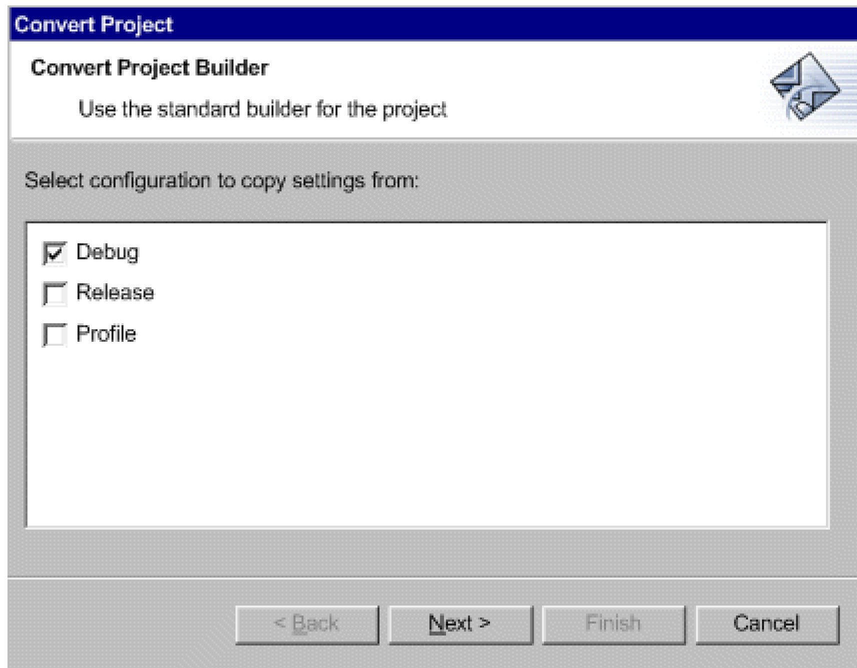

Figure 3 Configuration selection page of project conversion wizard

The second page of the wizard summarizes the settings that will be transferred to the project when the conversion takes place.

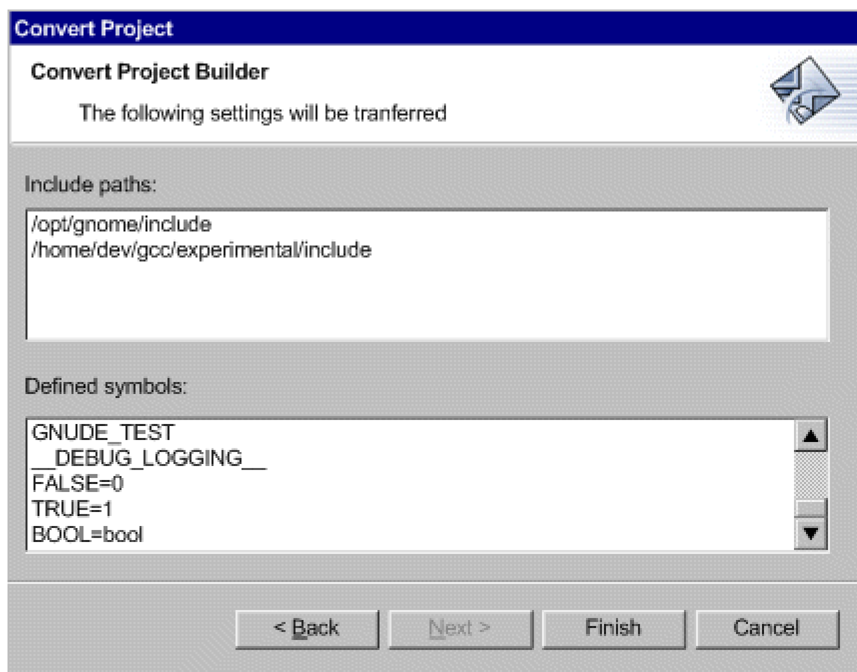

Figure 4 Summary page of project conversion wizard

3.1.3.2 Alternative

Rather than creating some sort of project conversion wizard, we could collapse the two project property pages into one. We could use a tabbed property page for both build

systems. On the first page we would have the common stuff like the command for make and the location of the project's build output. Another field could be selecting whether or not the CDT generates a makefile, since that is effectively the only difference between the two projects from the user's perspective. Of course, turning off makefile generation would not imply that the user's option settings would be lost, but they would still have to keep the includes paths up-to-date manually.

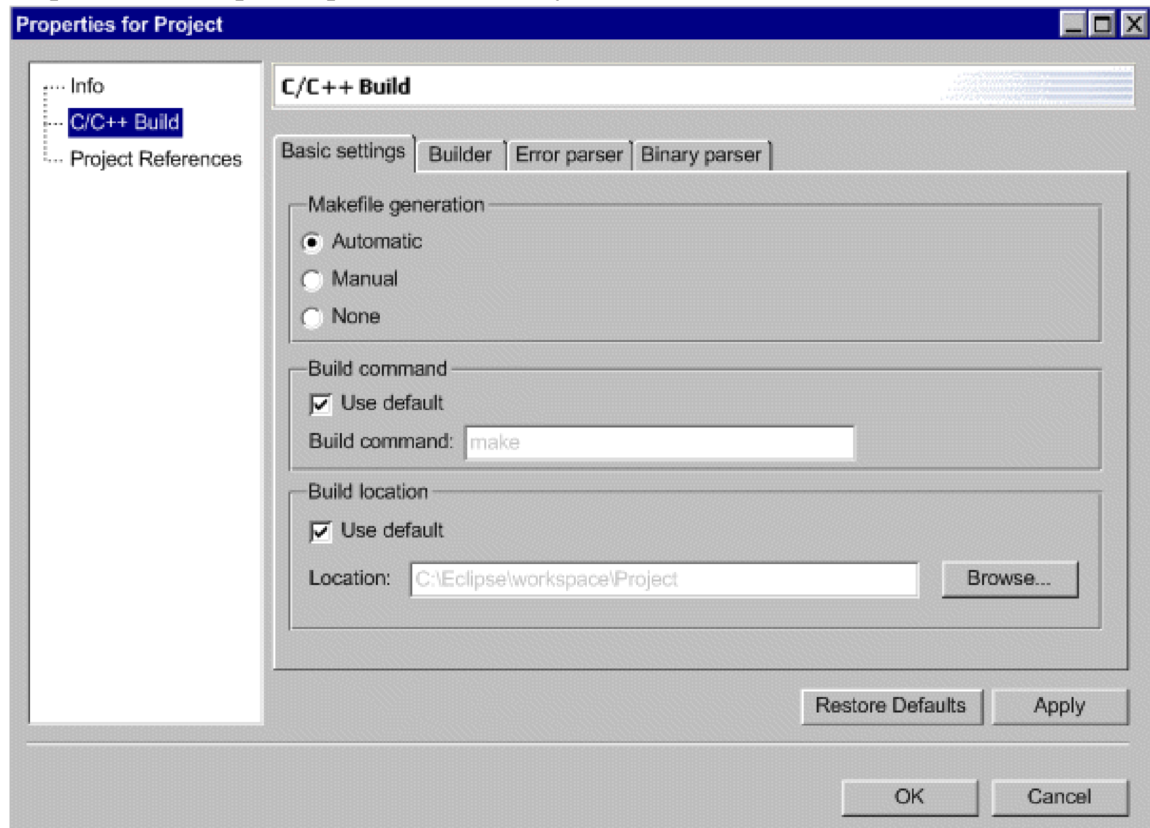

Figure 5 Alternative UI for converting projects

3.1.4 Clone Existing Configuration (C8)

There is already a dialog to add new configurations based on the default settings found in the plug-in manifest to a project. The clone gesture will create a new configuration and copy all the tool settings from a configuration defined in the project. For now, the UI impact will be minimal if we simply add a new button to the dialog that adds new configurations.

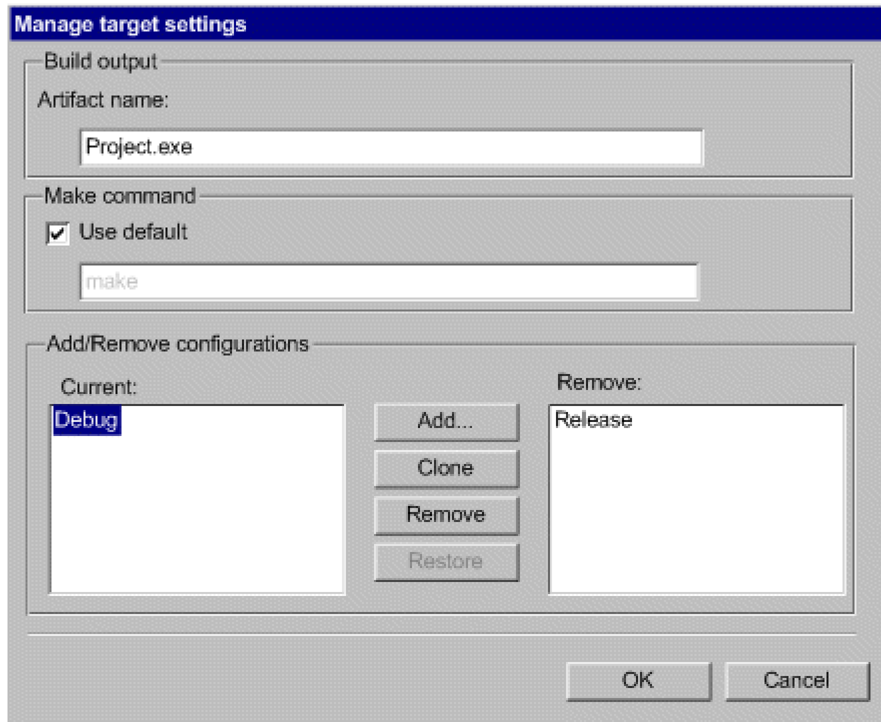

Figure 6 Dialog to support configuration cloning

3.1.5 Support Variables in Path Specifications (C11)

A number of elements need to be added to the UI to support the use of path variables specific to the CDT.

3.1.5.1 Build Property Page

Users can enter includes paths to the settings of a managed build project using a simple dialog. If we permit the user to supply a macro in the name of a path, then it would be a nice touch if allowed the user to see the defined macros. The dialog below has a "Macros" button that can be pressed to reveal the defined macros.

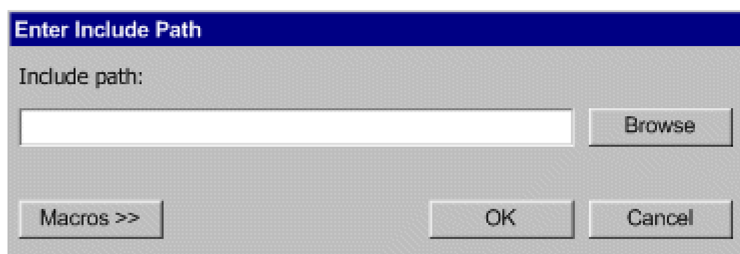

Figure 7 - Basic Include Path Entry Dialog

When the button is pressed, the dialog is resized to show the macro list control. The user will be able to drag a single macro onto the text widget at the cursor location. The "Macros" button can be clicked again to hide to list.

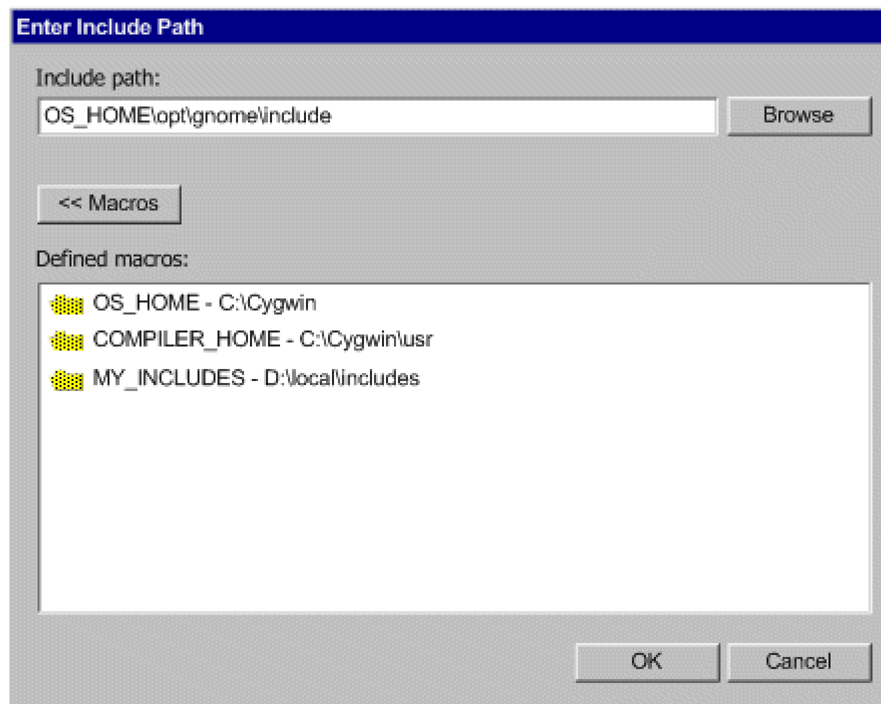

Figure 8 - Include path Dialog Expanded to Show Macros

3.1.5.2 Dedicated Preference Page

There will be a preference page for editing these variables at the workspace level. The analog for this feature is *classpath* variable in the JDT.

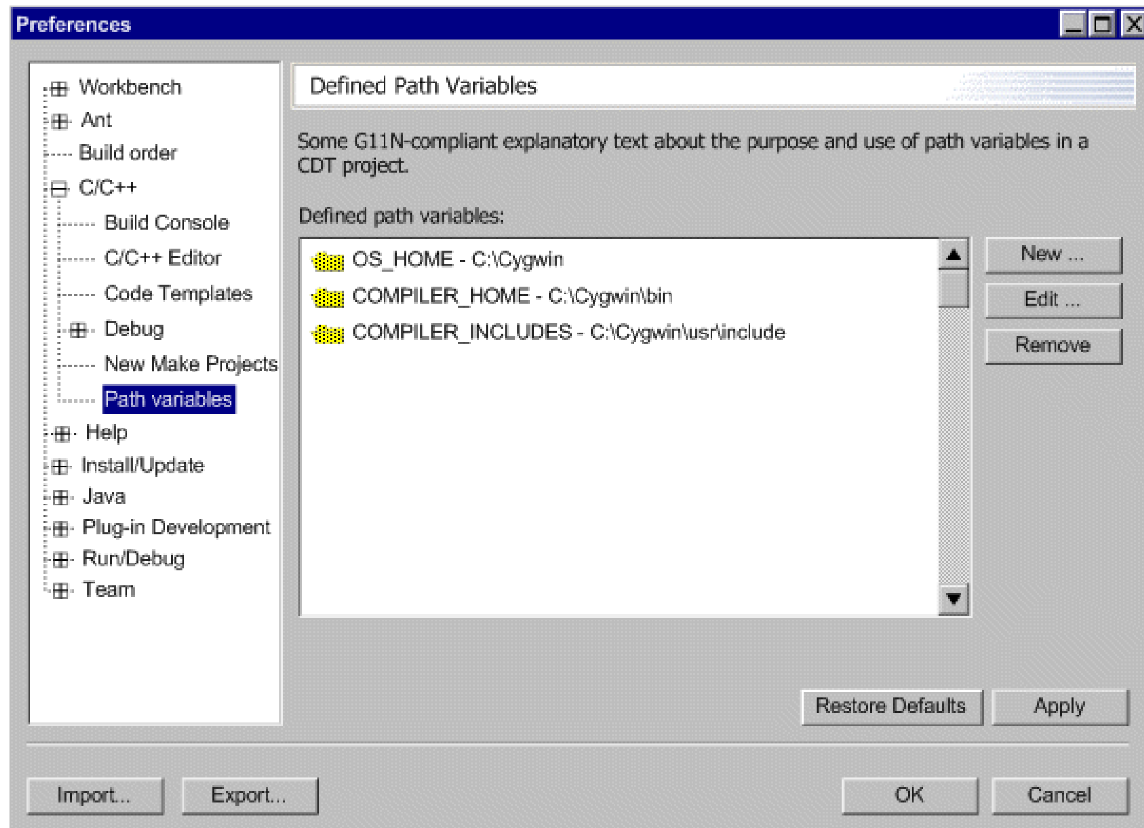

Figure 9 - Path Variable Preference Page

3.1.6 New Project Wizard Modifications (C13)

Imagine a scenario in which a user is running the CDT on Linux. There are two toolchains defined for Linux, one based on the GCC tools and another based on the Intel compiler. The build model also has definitions for the VC++ and Mingw toolchains that are hosted on Win32 and target Win32. There is also a definition for a Cygnus toolchain hosted on Solaris and targeting PowerPC 603.

The figure below shows a target list when filtering is turned on. Note that the user can only choose one of the two targets with tools hosted on Linux.

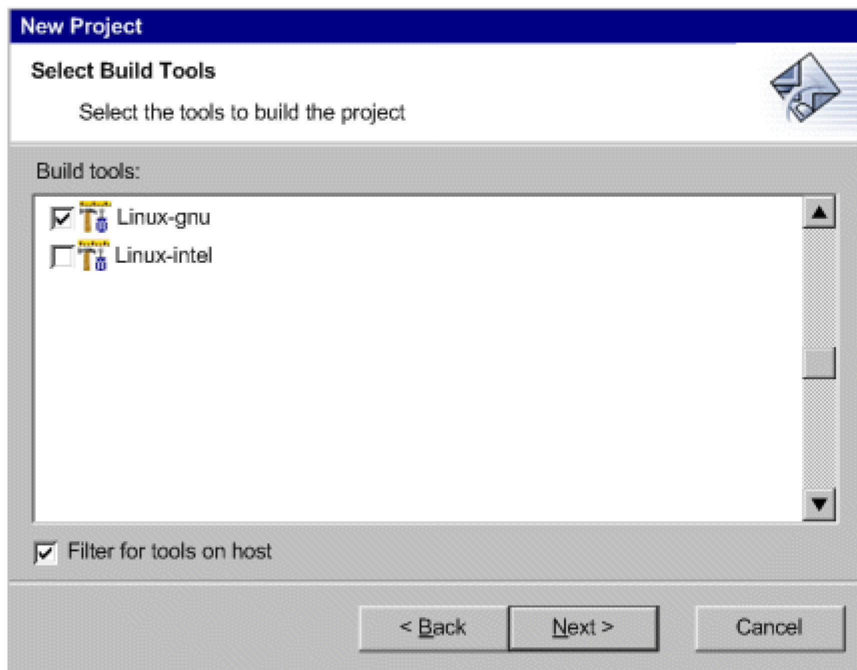

Figure 10 – Filtered Target List

The figure below shows the same dialog when filtering is turned off. The target platform for the build goal is shown inside the parentheses. As this might be different than the platform that the tools are hosted on, we might want to consider how to show that information.

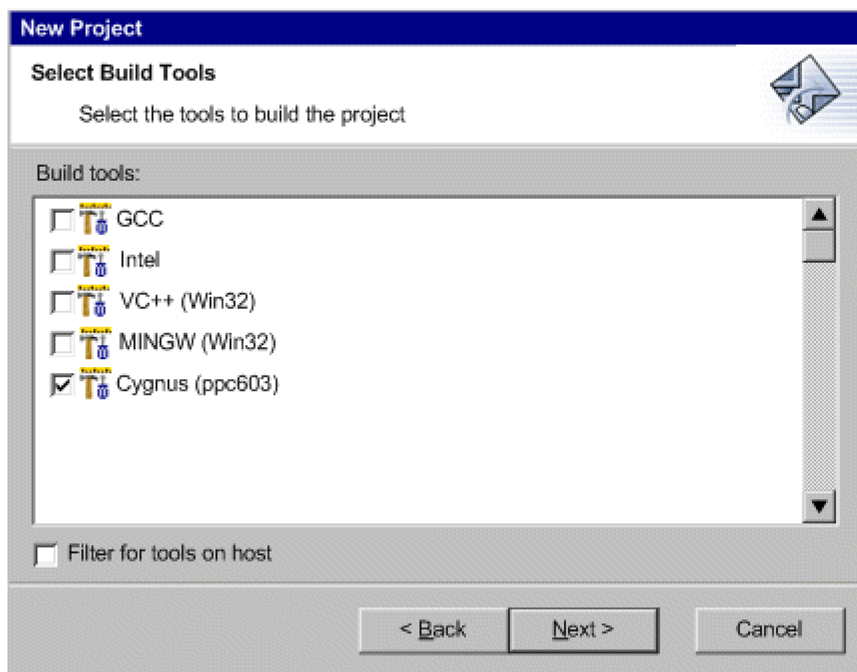

Figure 11 - Unfiltered Target List

3.1.7 Changing Project Types (C15)

We want to support the work flow where the user decides to change the build goal of the project from one 'type' to another. In order to do that, the user interface will need a bit of tweaking. One possible solution is shown below. This is a modified version of the current manage target/configuration dialog that currently exists in CDT 1.2. This mock-up is based on the assumption that the managed make property page will have a tabbed component and that the make command editor will be moved to the 'Basic settings' tab to make room for the type selector widget. If that assumption is incorrect, the existing dialog will have to be stretched to accommodate the extra widget.

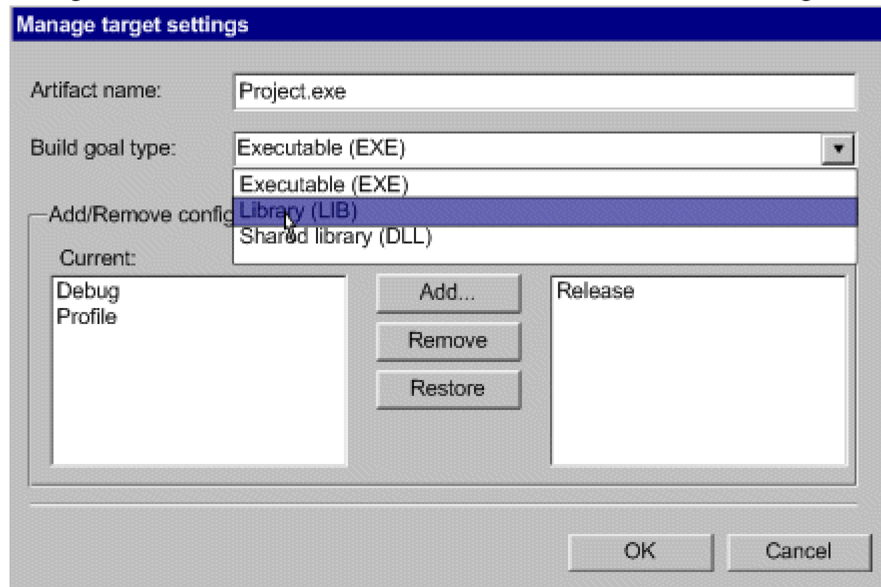

Figure 12 UI support for changing build goal of a project

3.2 Unicode Support and G11N

There are two areas of the managed build system that may require some work to be G11N compliant. The first is the UI, which is based on the standard SWT widget set and layout managers. We have to assume that these widgets and layout managers will behave properly in whatever locale they are created in, otherwise we will not be able to meet all of the requirements described in [2]. However, this section will discuss any work that still needs to be done to make the UI fully G11N compliant.

The managed build system also interacts with other tools that may or may not conform to these requirements. This section will discuss those issues.

3.2.1 Unicode in the Manifest and Project File

The plug-in manifest and the special file the build system uses to store user settings are encoded in UTF-8, so we should have no problem storing and retrieving Unicode characters in either.

3.2.2 Build Property Page

The build property page is generated dynamically depending on what options are defined for a toolchain. The labels for the widgets are all stored in the plug-in properties file.

Based on how the workbench handles translation, we can safely assume they are ready for translation and that aspect of the UI will be L10N compliant. Controls that take a value populate their default settings from the plug-in manifest, so we will have to make sure the default values are externalized in the plug-in properties file too.

For the most part, the widgets on the build property page will not be used to supply locale-specific information like dates or time. The entry widget that accepts a string does no translation, so if the user should happen to enter a number with a decimal place, it would simply pass it, as found, to the compiler or linker.

The widgets on the property page do no sorting, since the order entered by the user or discovered in the manifest is assumed to be correct. However, in order that we support bi-directional languages, the property page will have to be updated to display input correctly based on how it was encoded.

3.2.2 Unicode in Generated Makefile

The builder must generate a makefile with the information from the build model. This will include path locations with Unicode characters in them, and toolchain options. If we stick to the principle that the compiler toolchain is UTF-8 only, then the Unicode characters will be encoded properly. But what happens at this point depends on how the make utility interprets this input. If the make variant the user has installed on the system does not support UTF-8, then we will have the same problem we had for projects with spaces in the path names. Eclipse will work just fine, but building will fail because of the limitations of an external tool.

4 Design Discussion

4.1 Modifying the Tool Command (C1)

One important work flow that has been identified for the managed build system is changing the command line invocation of a tool. The use case identifies a user who wants to invoke a special variant of a tool without having to create a new target specification in a plug-in manifest. Typically, this involves running something like `purify <your_compiler_name>`, or one of the target-specific GCC variants, like `cc68k`, in place of the compiler command specified in the tool definition.

4.1.1 Use Cases

UC 1

The first use case begins when the user wants to change the tool invocation for a particular configuration in their project.

1. The user opens managed build settings on the property page for the project.
2. The user selects the configuration to edit. Alternatively, if the user wants to change the invocation for the entire project, they can select the 'all' configuration.
3. The user selects the tool from the list of tools in the configuration settings.

4. The user will be shown a widget in the settings edit area that contains the current tool command. The user will edit the command and hit the **Apply** or **OK** button to make the change.
5. If the user selects **Apply**, the property page will remain open and the build should not occur, even if the auto-build feature is turned on.
6. The user may also hit the **Restore Defaults** button to reset the change before they hit **OK**. This gesture should reset the tool command to the default in the plug-in manifest.
7. The managed build system will flag the project as out-of-date so that all files will be rebuilt using the new command when the project is built again.
8. The managed build system will change the setting for the tool in the file it uses to store settings between sessions.

UC 2

The second use case begins when the user wants to change the command line invocation for a tool for all projects in the workspace.

1. The user opens the preference page for managed build settings.
2. The user selects the toolchain from the list of defined toolchains the managed build system contains.
3. The user selects the tool from the list of tools defined for the toolchain.
4. The edit area will display the command line invocation in an edit widget.
5. The user will change the tool command and hit **Apply** or **OK**.
6. The managed build system will flag the project as out-of-date so that all files will be rebuilt using the new command when the project is built again.
7. The managed build system will change the setting for the tool in a special section of the file it uses to store settings between sessions.

4.1.2 Limitations

There has to be an arbitrary ordering of the levels of overrides when the managed builder generates a makefile for the project. By default, the build model will use the command in the manifest. If the user has set the command at the project level, that setting will be used. If the user has overridden the setting for a configuration, that setting will be used.

I am concerned that this feature might lead to false expectations about how to create new managed build targets. The work-flow that someone might try is to create a new project based on one of the defined tool chains, and then modify the name of the compiler and linker for each configuration. This will clearly fail if they try to use the options shown in the UI for their new compiler or linker unless their tools have identical command line syntax.

4.2 Setting Build Options Simultaneously (C2)

Users have pointed out that it is tedious and error prone to set options individually for each configuration of a given target. The build system should supply an *all* configuration by default where any changes to the options would be applied to all defined configurations.

4.2.1 Use Case

1. The user opens the build property page for the project.
2. The user selects the *all* configuration from the list of available configurations.
3. The UI will display entry widgets for options that are common to both.
4. The widgets for options that contain default values or values that are identical shall be populated with that option value. Widgets for options that are not the same in every configuration shall be empty.
5. The user enters the option value.
6. When the user clicks OK, the option value will be set for all the configurations in the project.

4.2.2 Limitations

Only options that are common to both configurations will be displayed.

Assume the user has set (overridden from the default) the value of option *O1* in configuration *C1* but not configuration *C2*. At some later time, the user switches to the *all* configuration. Since *O1* is overridden in *C1*, it can no longer be treated as part of the *all* configuration and is displayed as blank. If the user changes the value of *O1* in the *all* configuration and applies the change, the new value will appear when the next time the user looks at *C1*.

When the all configuration is selected, the 'Restore Defaults' command will reset every option in every configuration to the defaults defined in the manifest. Therefore hitting this button with the all target selected may have fairly major consequences for the project settings. This may be a good candidate for a prompt asking the user to confirm this gesture.

If the toolchain implementer supplies an *all* configuration, it will be ignored by the build model.

4.3 Converting a Project from Managed to Standard (C3)

4.3.1 Use Case

1. The user creates a managed project.
2. The user adds all of the relevant classes and files to that project and the CDT generates a makefile.
3. At some point the user decides that they want to manage the makefile themselves and run the project conversion wizard to switch the project from a managed to a standard project.
4. If there is more than one configuration defined for the project and if the path and symbol settings are different, the user will be prompted to select the configuration settings they wish to transfer.
5. The wizard shall insure that the include paths and defined symbols are transferred.
6. The make command for the standard project will be set to the same value that was used in the managed project.
7. The makefiles will not be deleted. If they were not available for the selected configurations, they will be generated before the wizard finishes.

4.3.2 Limitations

After the conversion, the same limitations for standard projects apply. Every new file that is added to the project will have to be manually added to the list of source files in the makefile, all of its dependencies captured, as will any custom build commands.

A project can be converted to a standard build project, but the wizard will not convert a standard project to a managed project.

The wizard requires that the add-ins for both build systems be installed. If the standard build system is not available, the wizard should show a dialog to the user explaining why the conversion can not start.

4.4 Refactor the Current Default Manifest (C4)

The current default manifest that defines the Gnu toolchain is bulky because options are duplicated for each tool on each host. The default toolchain definition should be made more compact by better defining inheritance of tools and options. This will make it easier to correct problems and maintain the definitions going forward.

4.4.1 Limitations

This will require changes the build model itself. In order to support backwards compatibility, a versioning scheme will be needed to read in older variants of the build model.

4.5 Generic "GNU" Target for Default Toolchain (C5)

Given that the GCC tools have the same command line invocation and flags for every platform they run on, the user can reasonably expect their managed build projects to build anywhere the GCC has been ported to. Currently this is not the case because each target in the default manifest is defined in terms of their platform and toolchain. This means that while it is probably sufficient to have a "Gnu Executable" target, we now have a "Linux Executable", a "Cygwin Executable" target and so on. If the user wants to add a new target for a new platform, even if the tool settings for that target do not differ in any meaningful way, they have to edit the plug-in manifest. This needlessly complicates porting projects and porting the managed build to new platforms.

4.5.1 Use Case

1. The user starts Eclipse on *PlatformA*.
2. The user creates a project, *ProjectA*, using one of the supplied default targets based on Gnu. The user sets the required paths and tool settings to build the project correctly on *PlatformA*.
3. The user runs the CDT on *PlatformB*. Note that GCC must be available and installed on *PlatformB*.
4. The user imports *ProjectA* into their workspace on *PlatformB*. After modifying any settings that may not be identical (like include paths), the project will build with no additional input from the user.

4.5.2 Limitations

Unfortunately, this is one of the requirements that would have been easier to implement in the first release. The UI displays the platform during project creation and on the property page.

Note that this feature is not intended to support remote building on another platform. The user will be importing and building their projects in the Eclipse UI.

4.6 Automatically Update the Build Information (C6)

4.6.1 Use Case

1. The user specifies that a managed project references another managed project in the workspace.
2. The build system shall update the build information for the managed project automatically.
3. If the referenced project is a library, the library and library search path will be added to the linker information.
4. Header file search paths should be updated in the referencing project, if possible.
5. If the referenced project is a standard project, the user will be warned that they will have to add the build information manually.

4.6.2 Limitations

The library and library search path information can only be added automatically if the referenced project is a managed project. Standard projects do not know what type of build goal they produce.

Adding header file search paths to the project's build information may prove more difficult. While both managed and standard projects will have includes path information, it is not clear what should be added to the referencing project's build information. Consider the case where the referenced project is a library. The build information in the referenced project contains the information needed to build that project, but only the header files that define the interface should be added by the referencing project. Blindly appending the path information to the referencing project is not only overkill, it may also lead to errors if there are header files in the referenced project with the same name. For now, I am not sure this is feasible.

4.7 Update Target Definitions in Manifest (C7)

4.7.1 Use Case

1. The user creates a new project with a build goal based on an execution target.
2. The user builds the project.
3. The project builds correctly with no input from the user.

4.7.2 Limitations

The goal of providing a reasonable set of default build settings is easier to say than do. We cannot be sure that user has installed their build tools or cross-platform development environment in a default location. In order to accommodate this, we will have to define

defaults using environment variables or macros of some sort. The user will still have to intervene to set these up correctly.

Basically, the problem boils down to defining what is reasonable. The sticking point for the clients of the build information seems to be the location of header files. Using variables in path specifications should get us part of the way there, absent an installer or bundled toolchain.

In the case of compiler settings, we could try to determine them automatically using the same scheme being proposed for the standard make system. If the tools are not in the path, this will not work either.

4.8 Cloning an Existing Configuration (C8)

4.8.1 Use Case

1. The use case begins when the user selects a target and configuration from the C/C++ Build property page for their project and clicks Manage.
2. The build system will display a dialog with a list of defined configurations.
3. The user selects one of the configurations in the list and the Clone button is enabled.
4. The user clicks the Clone button and is prompted for a configuration name.
5. The name field will be populated with the name of the selected configuration with the string "_clone" appended to it.
6. The user accepts the default name or enters a new name and clicks OK.
7. The build model shall create a new configuration with the name selected by the user and all of the *current* settings of the configuration selected in the list.

4.8.2 Limitations

The UI that supports managing configurations was assembled rather quickly. There are now two gestures the user can perform; creating a new configuration based on the default settings defined in the plug-in manifest (the currently supported gesture), and creating one based on a modified configuration (the gesture proposed here). We should probably consider how to make this distinction clear, which could simplify the dialog.

4.9 Debug Configuration is the Default (C9)

To improve the out-of-box experience with creating and building a managed build project, the debug configuration should be the default. That way, a first-time user can get the project up and running in the debugger without having to open a single settings dialog.

4.9.1 Use Case

1. The use case begins when the user creates a new managed build project.
2. The project shall be created so that the debug configuration is selected by default when the user builds.

4.9.2 Limitations

A Target does not know which configuration is the default. Currently, the build model treats the plug-in manifest as an ordered list and supplies this list to the UI. The only way to make the debug configuration the "default" is to define it before any other configuration in the manifest.

4.10 Remove Gnu Extensions from Generated Makefiles (C10)

A key theme for this release is to improve the experience for new users. One problem that experienced and new users alike have encountered is that the makefiles generated by the managed build system rely on Gnu-specific extensions to build correctly. This problem is compounded in systems where there is more than one make utility available, or where the make utility supported by a toolchain does not accept the extensions.

4.10.1 Use Case

1. The use case begins when the user adds a resource to their project and builds the project.
2. The build system will update the makefiles correctly and call the `make` utility specified for the project.
3. If the `make` utility fails to run, the build system will present a reasonable message to the user with as much information as possible.
4. The build system will report any errors encountered during the build to the user or report that the build completed successfully if there were no errors.

4.10.2 Limitations

The user has to have a make utility in their path.

4.11 Support Variables in Path Specifications (C11)

This feature is difficult to scope. The goal is to be able to specify a location in a file system using variables instead of fixed locations that can vary from user to user.

4.11.1 Use Cases

UC 1

1. The first use case begins when a toolchain implementer creates a new toolchain specification.
2. The implementer defines a compiler that searches a number of paths by default, so an end user will not explicitly set these paths. If the implementer does not specify these paths, the built-in parser will be unable to completely parse a file and core features of the CDT that rely on the parser will not work correctly.
3. The implementer defines the search paths using a variable instead of a fixed location knowing that users may install their compilers in different locations.
4. The implementer specifies a default value for the variable, but expects the end-user to supply another value if this default is incorrect.

5. The implementer shall specify a unique variable name or live with the best effort of the build system to resolve duplicates.

UC 2

1. The second use case begins when a team leader or build engineer creates a new project for a team of developers based on a defined toolchain.
2. The user supplies a default value for the variables and shares the project with other developers through their CM system.
3. When a developer checks out the project into their workspace, the build system will insure that the variable is defined for the workspace and populate it with the default value.

UC 3

1. The third use case begins when a developer adds a new project to their workspace.
2. The default location of the compiler is incorrect.
3. The developer changes the value of the variable in the workspace so that the search paths are correct.

4.11.2 Limitations

Other IDEs allow users to use macros in any text entry widget. Allowing that degree of control may be valuable to users, but a feature that ambitious is probably better implemented in Eclipse. For now, this feature is intended to allow the use of a linked resource in any entry widget on a project build property page that expects a path. In future releases, we may want to widen the scope to all entry widgets that accept a path.

Path variables are associated with a workspace. Users edit path variable by manipulating Linked Resources in the workspace preferences. This has two major implications. The first is that a user may only want to change the setting for a particular project, not every project and there is currently no way to support that. The second is that workspace preferences cannot be shared, so there is no clean way of implementing the second use case.

Even if we resolve the limitations inherent in path variables, we have to face the fact that this feature will not improve the out-of-the-box experience for naïve users. The variable itself will have to be set and only the end user will be able to do this reliably, since we do not provide build tools like other commercial IDEs and we do not have an install program that would prompt a user to set this up manually. Unlike Eclipse, which has the built-in failsafe that if there is no JRE, you cannot even launch, a user can get all the way to the compile step before they notice that no tools are in the path. The only way they would notice that includes paths are not available is if they try to use content assist.

4.12 Supply VC++ Toolchain Implementation (C12)

This is a very complex feature that requires a lot of thought before it is undertaken. It will require a lot of work to scope this feature properly, and to determine the needs and

expectations of the users who would take advantage of this capability if we do add it to the CDT. There is a partial list of use cases below. If this feature is committed, then the use cases can be elaborated. I am including an initial discussion of the feature for the sake of completeness.

4.12.1 Use Cases

1. Allow existing VC++ users to import and build existing VC++ projects in Eclipse as managed build projects.
2. Allow developers to create new managed VC++ projects in Eclipse.
3. Allow the user to import an existing VC++ project into the CDT as a standard project.

4.12.2 Limitations

There are several important technical problems no matter how we scope the work. The current CDT parser is not expected to support any VC++ extensions in the near term. Content assist will either be unreliable or unavailable if the parser fails. Indexing and search will also have similar problem. As the managed builder relies on the indexer to track resource-level dependencies, we will have to find a reliable replacement since there is no VC++ equivalent to `makedepend`.

Another critical problem is that there is no integration with the VC++ debugger. It is hard to imagine anyone taking advantage of this integration without that basic capability.

If we do decide to continue, VC++ projects often use many different tools. For example there is an IDL compiler, a resource compiler, integration with .NET interpreted language tools, and an installer utility. If we are trying to support VC++ project imports, we will only be able to define tools with command line interfaces. Even if we do support every critical build tool, there will be no way for the user to edit GUI resources other than as text files.

If we want to limit the feature to creating new Eclipse projects that use the VC++ toolchain, then the work is almost as complex as the first use case. In addition to all the tools the VC++ user might reasonably expect to use, there are many different "types" of VC++ projects; MFC, ATL, Win32, and .NET.

Importing an existing VC++ project as a standard CDT project may be a more attainable goal. In this case, the CDT becomes a replacement editor but the VC++ tools are used to build the project. While this pushes the problem of managing the build settings back onto the user's plate, VC++ users are probably unwilling to sacrifice the convenience of the build system of their current IDE.

4.13 New Project Wizard Should Show All Targets (C13)

4.13.1 Use Case

1. The use case begins when the user runs the New project wizard.
2. After choosing a name for the project and its location, the user is prompted to select one or more execution targets for the build goal of the project.
3. The list of choice shall be populated with all the defined execution targets.

4. The user will select one or more targets from the list and continue.

4.13.2 Limitations

The user will be able to create a project that cannot be built on their host platform. This will undoubtedly lead to confusion for naïve users.

4.14 Implement Tool Inheritance (C14)

This requirement is not user driven, although it will make it easier to maintain the plug-in manifest and it will facilitate the C4 and C5 requirements. Currently there is no way for a tool to inherit its settings from a parent tool. So if the user wants to create a tool that differs from another in a minor way, they still have to create an entire specification leading to needless duplication, a large manifest, and more maintenance headaches.

4.15 Automate Changing a Project's Build Goal (C15)

The use case to be supported is a user that wants to change the type of project build goal, for example from an executable to a library. This is not an activity that occurs that frequently, so automating the process seems like a lot of work for very little reward. I am including a full discussion of the feature for the sake of completeness but I do not think there will be time in this release to complete this feature.

4.15.1 Use Case

1. The user selects a project in the navigator or project explorer.
2. The user selects the **New > Project > Conversion** wizard.
3. The build system will present the user with a list of valid build goals and the user will select one.
4. The build system shall do its best to preserve build settings between build goals.
5. The project will be rebuilt the next time the user does a build.
6. If auto-build is turned on, the rebuild will be triggered when the user selects **Finish**.

4.15.2 Limitations

I really don't know what the frequency of this type of gesture will be after a project is initially created. In other IDEs, you chose the type and live with it. If you made a bad choice at project creation time, you start again and import your source files into the new project. Frankly, as a developer, I think that is a reasonable limitation.

It will likely be difficult or impossible to transfer all the build settings when a project's 'type' is changed. For example, include paths and preprocessor symbols should be straight forward, but if the user switches an executable project to a library, the build model will have to discard the linker options. We can attempt to make the build system work on 'best-effort' basis and warn the user that they should manually verify that all the settings are properly configured for the new type.

We will have to decide how much automation to provide. Consider the user who converts a project from an executable to a shared library. One possibility would be to use the quick fix feature to highlight the `int main (...)` function and suggest the user change it to `BOOL WINAPI DllMain (...)`. This is probably sufficient, but it will require that the

build system understand what type of method to look for and what to replace it with. At the moment we do not have this capability, so the user will have this pain-point anyway.

10 References

1. *CDT 2.0 Managed Build System SRS.*

2. *Globalization FDS in CDT 2.0.*

3. *FDS C/C++ Parser for CDT 2.0*