Bachelor's Thesis

Information Technology

Embedded Software

2011

Petri Tuononen

LLVM TOOLCHAIN SUPPORT AS A PLUG-IN FOR ECLIPSE CDT

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Embedded Software

Completion of the thesis: 3/2011| Total number of pages: 59

Instructor: Lic.Tech. Jari-Pekka Paalassalo

Author: Petri Tuononen

TURUN AMMATTIKORKEAKOULU THESIS

The mission of the thesis was to develop LLVM toolchain support as a plug-in for Eclipse CDT. There are multiple reasons why such a plug-in was worth to make. First, it is proven that LLVM with Clang front-end can compile C/C++ and Objective C program code faster than GCC. Secondly, currently LLVM tools are mostly run on command prompt and the commands tend to need option flags. The newly created plug-in integrates the LLVM toolchain tools with readily configured default option flags. The plug-in allows users to create C/C++ projects in Eclipse which use LLVM toolchain with Clang or LLVM-GCC compiler, among others. Building a project becomes a one click operation via graphical user interface.

The main point of the plug-in is the pure easiness of building C/C++ projects with one of the industry's most efficient C/C++ compiler. Eclipse was chosen as IDE, because it is one of the most popular open-source IDEs available. Eclipse provides a full-blown software development environment for even the most conscious developers and now it is available in LLVM based development. Although the LLVM tools are pre-configured by default, they can be configured easily in Eclipse.

Using LLVM from Eclipse is efficient and user-friendly which saves valuable time and thus money. The LLVM toolchain and Eclipse IDE complement each other by creating a coherent C/C++ development environment with advanced and modern features. The environment is fully open-source and operating system independent.

Another research topic was to find out how to contribute to Eclipse Foundation.

KEYWORDS:

LLVM, Clang, llvm-gcc, Eclipse, CDT, plug-in, toolchain, open-source compiler, contribution to Eclipse, open-source contribution.

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ
TURUN AMMATTIKORKEAKOULU

Tietotekniikan koulutusohjelma | Sulautetut ohjelmistot

Opinnäytetyön valmistumisajankohta: 3.2011 | Sivumäärä: 59

Ohjaaja: TkL Jari-Pekka Paalassalo

Tekijä: Petri Tuononen

LLVM TYÖKALUKETJUN TUKI ECLIPSE CDT:IIN LISÄOSANA

Opinnäytetyön tavoitteena oli kehittää LLVM työkaluketjun tuki Eclipse CDT:iin lisäosana. Syitä lisäosan tekemiseen löytyy monia. Ensiksi on todistettu, että LLVM työkaluketju yhdessä Clang kääntäjän kanssa kääntää C/C++ ja Objective C ohjelmakoodia nopeammin kuin GCC. Toiseksi nykyisin LLVM työkaluja käytetään pääasiassa komentorivin kautta ja komennot vaativat yleensä valintalippuja. Lisäosa integroi LLVM työkaluketjun työkalut ennalta konfiguroiduilla valintalipuilla. Lisäosa luo käyttäjille mahdollisuuden luoda C/C++ projekteja Eclipsessä, mitkä käyttävät LLVM työkaluketjua yhdessä esim. Clang tai LLVM-GCC kääntäjän kanssa. Sovellusprojektin ajokelpoinen ohjelma voidaan saada aikaan vain yhdellä käyttöliittymän painikkeen napsautuksella.

Pääajatus lisäosan takana on C/C++ projektien erittäin helppo rakentaminen yhdellä alan tehokkaimmalla kääntäjällä. Eclipse valittiin ohjelmointikehitysympäristöksi, koska se on yksi suosituimmista ohjelmointikehitysympäristöistä. Eclipse tarjoaa kokonaisvaltaisen ympäristön ohjelmistokehitykselle jopa kaikista vaativimmille käyttäjille ja nyt se saadaan käyttöön myös LLVM pohjaisessa kehityksessä. Vaikka LLVM työkalut ovatkin ennalta konfiguroituja, voidaan ne konfiguroida Eclipsestä käsin vaivatta.

LLVM:n käyttäminen Eclipsessä on tehokasta ja helppokäyttöistä, mikä säästää aikaa ja siten rahaa. LLVM työkaluketju ja Eclipse ohjelmointikehitysympäristö täydentävät toisiaan luoden koherentin ympäristön C/C++ ohjelmistokehitykselle kehittynein ja modernein toiminnallisuuksin. Ympäristö on täysin avointa lähdekoodia ja on käyttöjärjestelmäriippumaton.

Tutkimuksen kohteena oli myös ottaa selvää kuinka Eclipse Foundationiin toimitetaan kontribuutio.

ASIASANAT:

LLVM, Clang, llvm-gcc, Eclipse, CDT, lisäosa, työkaluketju, avoimen lähdekoodin kääntäjä, Eclipse kontribuutio, lahjoitus avoimen lähdekoodin yhteisölle.

CONTENT

APPENDICES

FIGURES

TABLES

LIST OF ABBREVATIONS

API	Application Programming Interface.
Back-end	Back-end uses intermediate representation to produce code in a computer output language.
Bug	Flaw in software.
Bugzilla	Bug tracking software.
Bytecode	Binary file format used by LLVM.
CDT	C/C++ Development Tools plug-in for Eclipse IDE.
Clang	‘LLVM native’ C/C++/Objective-C compiler front-end.
Committer	Committer is an individual who has been given write access to codebase hosted by someone.
Cross-platform	Platform (operating system) independent.
Cygwin	Linux-like environment for Windows.
Eclipse	Popular open source software tool platform featuring a GUI and IDE.
ELF	Executable and Linkable Format.
Front-end	Initial stages of a process i.e. interface between the user and the back-end. Translates source code into an intermediate representation.
GCC	GNU Compiler Collection.
GNU	GNU's Not Unix!
GUI	Graphical User Interface.
IDE	Integrated Development Environment.
IR	Intermediate Representation.

Java	Computer language.
JIT	Just-In-Time.
JUnit	Unit testing framework for the Java programming language.
LLVM	Low-Level Virtual Machine.
LLVM-GCC	LLVM C front-end which compiles C/ObjC programs into native objects, LLVM bytecode or LLVM assembly language.
Mach-O	Mach object file format.
MBS	CDT Managed Build System.
MinGW	“Minimalist GNU for Windows”, is a minimalistic implementation of essential GNU software development programs as native Windows applications.
Patch	Piece of software which fixes software problems e.g. bugs.
PDE	Plug-in Development Environment.
PE	Portable Executable.
Plug-in	In this context, a plug-in is software aimed to provide additional functionality on top of Eclipse platform.
Subversion/SVN	Revision Control System.
Toolchain	Chain of programming tools used to build a computer program.
UML	Unified Modelling Language.
URL	Uniform Resource Locator.

1 INTRODUCTION

The initial work started in spring by Leo Hippeläinen, a Senior Software Technology Specialist at Nokia Siemens Networks. After I joined Nokia Siemens Networks in June as a summer trainee I continued the development of the plug-in and collaborated with another summer trainee and the Senior Software Architect. I decided that the work I started in Nokia Siemens Networks would become the topic of my thesis. In the end I was the only Nokia Siemens Networks employee working on the project on day-to-day basis. By the end of August the plug-in still needed further development and I planned to continue on winter 2011.

Nokia Siemens Networks was interested in making a plug-in for Eclipse which allows using LLVM toolchain within Eclipse IDE. This allows C/C++ developers to work in fully open-source environment without any dependencies on commercially licensed software.

LLVM toolchain plug-in for Eclipse CDT provides a cross-platform development environment which uses one of the most efficient and modern compiler architectures available for C and C++ languages at the moment.

The plug-in is released under Eclipse Public License 1.0 and is freely available for anyone to download. Nokia Siemens Networks wanted to contribute to open-source community by releasing it with such a license that allows anyone to contribute i.e. modify source code of the plug-in. It was planned that the plug-in would be donated to Eclipse Foundation which would ensure the further development of the plug-in.

2 BACKGROUND INFORMATION

2.1 What is LLVM?

LLVM (Low Level Virtual Machine) is a collection of advanced cross-platform compiler technology i.e. infrastructure which consists of libraries, toolchain and compiler tools. It started as a research project at the University of Illinois in 2000 by Chris Lattner and Vikram Adve. The initial release was in 2003. Currently it is developed by LLVM Developer Group, numerous individual contributors and industry and research groups. LLVM uses University of Illinois Open Source License which allows individuals to see and modify the project’s source code. [1]

Although LLVM was originally implemented for C/C++ and also written with C++, its language-independent virtual instruction set and type system allows creation of front-ends for other computer languages. [1] [2]

The strengths of the LLVM infrastructure are its extremely simple IR (Intermediate Representation) design which is easy to learn and use, source-language independence, powerful and modular mid-level optimizer, clean and modular code generator, automated compiler debugging support, extensibility, and its stability, reliability and performance of the generated code. [3][4] LLVM also supports a so-called life-long compilation model which includes link-time, install-time, run-time, and offline optimization. [5]

The LLVM provides reusable modular components (libraries and tools) that allow building compilers, optimizers, JIT (Just-In-Time) code generators, and many other compiler-related programs easily and with reduced time and cost. [4][6] Tools that LLVM contains e.g. assemblers, automatic debugger, linker, code generator and modular optimizers are also made by using the LLVM libraries and are shared across different compilers. [2] [4]

“The core of LLVM is the intermediate representation (IR). Front ends compile code from a source language to the IR, optimization passes transform the IR, and code generators turn the IR into native code.” [6] LLVM IR design allows LLVM to analyze and optimize code as early as possible and compile-time optimizations can be run also at link-time. [4] Low-level instruction set (the virtual object code) enables powerful program analysis and transformation capabilities at link-time and run-time. [7]

“The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. The three different forms of LLVM are all equivalent.” [8]

LLVM is nowadays a project with multiple contributors which include members from industry, research groups and individuals. [2]

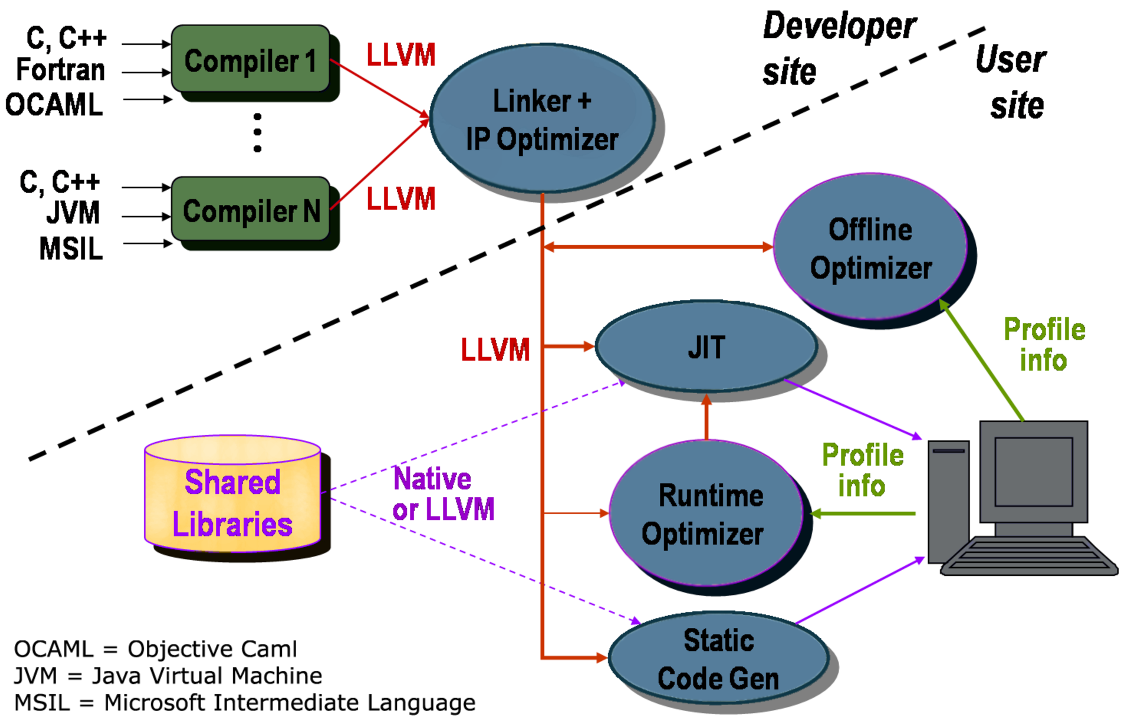


Figure 1. LLVM Architecture [9]

The above figure shows how source program code is fed to Clang or LLVM-GCC compiler and the files produced are then linked together and optimized at link-time. This is the phase where developer site ends and user site begins. After the linking process the file might be optimized by Runtime Optimizer or

Offline Optimizer before ending up to either Static Code Generator or JIT. Both Static Code Generator and JIT share the same shared LLVM libraries.

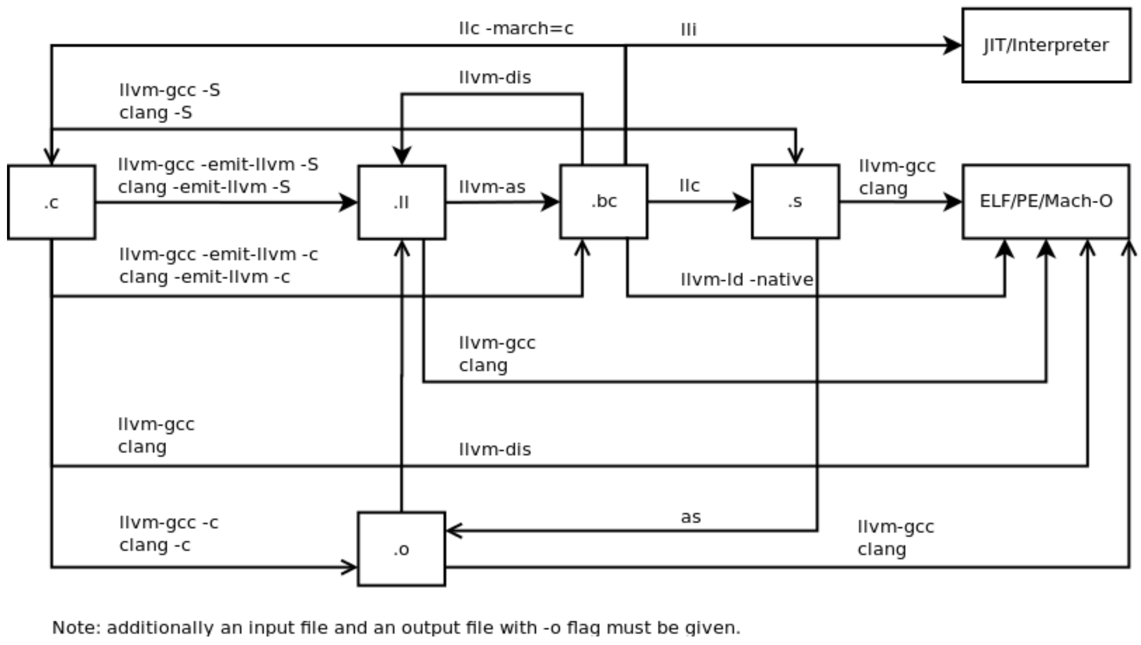


Figure 2. The transformation of file formats with different commands

The above figure shows how Clang and LLVM-GCC front-ends can create .ll, .bc, .s, .o and ELF files. Assembler (llvm-as) can also create .bc files from .ll files. From .bc files linker (llvm-ld) can create platform specific executable file, static compiler (llc) can create .s file and JIT compiler can fast load them.

2.1.1 LLVM toolchain tools

LLVM toolchain contains multiple tools for multiple purposes and that is one reason what makes it modular.

2.1.1.1 LLVM C/C++ compiler (llvm-gcc/llvm-g++)

GCC consists of three major parts which are front-end, optimizer and code generator. The LLVM C/C++ compiler replaces optimizer and code generator, but still uses GCC C/C++ parser and runtime libraries. [2] LLVM C/C++ compiler also differs from GCC in a way that .o files (that are created after compilation of C/C++ file) contain LLVM IR/bytecode, not machine code and executable can be bytecode or machine code. [5] Linking LLVM and GCC compiled code stays safe despite the differences in compilers. It is also safe to call into libraries built with other compilers. LLVM C/C++ compiler features a link-time optimizer which further improves performance. Compile time and code execution time is reduced by LLVM's optimizations and code generation. [2]

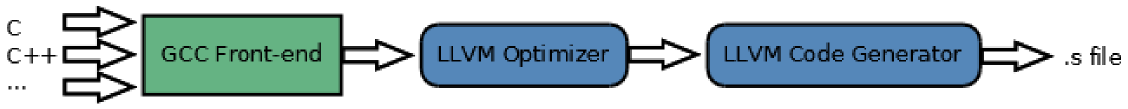


Figure 3. LLVM GCC design

2.1.1.2 Clang (LLVM native compiler) (clang, clang++)

Clang is a GCC compatible compiler front-end for the C, C++, Objective-C and Objective-C++ programming languages and uses the Low Level Virtual Machine as its back-end. [10]

Clang on top of LLVM is aimed as a replacement of the GCC stack. Clang is designed to reduce memory footprint, disk space and compilation time compared to GCC. Clang also increases program execution speed. Clang is highly modularized with codebase significantly simpler than GCC's which makes it more flexible than GCC and thus developers can extend it with less effort and coding skills. [10][11]

Clang features expressive diagnostics and that was one of the main reasons for its development along with the fact that GCC libraries had become more and more complex during the decades due to open-source nature of its license and also the development had become stagnated. [12][2] Although Clang is also developed under open source license it has not suffered from large or complex codebase issues due to its modular architecture.

Clang Static Analyzer is a notable feature of Clang that uses algorithms and techniques to analyze source code in order to find bugs. This feature is useful even as a standalone tool. [13]

The downside of the Clang is the fact that compilers tend to take time to mature as in case of GCC and being such a new compiler it has to gain approval of the mainstream. [14]

2.1.1.3 Assembler (llvm-as)

The assembler transforms the human readable LLVM assembly to LLVM bytecode and finally writes the result into a file or to standard output. [15]

2.1.1.4 Disassembler (llvm-dis)

The disassembler transforms the LLVM bytecode to human readable LLVM assembly code. [15]

2.1.1.5 Linker (llvm-ld)

Links multiple .bc (LLVM bytecode) files together into a single bytecode file. [16]

2.1.1.6 Archiver (llvm-ar)

The archiver produces an archive containing the given LLVM bytecode files which can then be linked into an LLVM program. [15]

2.1.1.7 Optimizer (opt)

LLVM optimizer features standard scalar optimizations, loop optimizations and interprocedural optimizations. [17]

2.1.1.8 LLVM Static Compiler (llc)

The llc tool compiles LLVM source inputs into assembly language for a specified architecture. [18]

2.1.1.9 LLVM Execution Engine (lli)

The lli tool uses a Just-In-Time compiler if it is available and otherwise LLVM interpreter. JIT emits machine code into memory instead of ".s" file and uses same code generator as Static Code Generator. Interpreter is simple and very slow but portable. [19]

2.1.2 LLVM file formats

File format	Description
.ll	LLVM human-readable assembly language in text format.
.bc	LLVM bytecode in binary format.
.s	LLVM Assembler code.
.o	Machine code generated by a compiler from source code module. Used by linker to form a completed program from multiple object files.

Table 1. LLVM file formats

2.2 Advantages of LLVM

2.2.1 Efficiency

Selecting LLVM toolchain can lead to faster compilation and code execution time in most cases. There are huge amount of debates going on with multiple architectures, compiler versions and software to be compiled with. It is not that straightforward which compiler takes the overall speed champion trophy after all. There are still some cases where some compiler performs better than the other considering the equal and fair environment for all contenders. Efficiency is one of the reasons to choose LLVM to compile programs.

2.2.2 GCC replacement in C/C++ development

The main reason for LLVM project was to provide an alternative to stagnated and old GCC compiler with modular and better performing compiler, which is easy to use. LLVM provides modern features and a new code base which is easy to learn and the modular architecture of LLVM infrastructure makes possible to create new compilers using LLVM libraries and tools. [2]

2.2.3 Future possibilities provided by the LLVM toolchain

LLVM provides a new compiler system for the current generations which consist of multiple separate projects nicely integrated together and the organizational status on behalf of Apple Inc. provides a nice playing ground for new experiments.

2.2.3.1 Further modularization

Modular architecture of LLVM allows the improvement of one tool at a time (without compromising others) e.g. optimizer which is used during link-time and compile-time that makes optimizations even better and faster which in turn makes the whole compilation process to perform better.

2.2.3.2 New projects using LLVM libraries

LLVM could become an integral part of new compiler designs aimed for embedded system architecture. For example compiler front-ends for multiple target embedded system architecture and multi-OS support which use LLVM libraries as back-end. [20]

2.3 Companies using LLVM

Apple Inc. is probably a company using LLVM most widely and part of the reason is that Apple hired the main author of LLVM. Apple uses LLVM on Xcode IDE and Mac OS X operating system. Adobe Systems Incorporated uses LLVM Optimizer and JIT codegen for the Hydra language. NVIDIA uses LLVM on OpenCL runtime compiler. Cray Inc. uses LLVM as a back-end for the Cray x86 compiler on their supercomputers. [21]

2.4 Companies using Eclipse CDT

Companies using Eclipse CDT include e.g. ARM, Freescale Semiconductors, IBM, Intel and Nokia. [22]

2.5 What is Eclipse platform?

The Eclipse platform is a development framework that was donated to open source community by IBM. This platform allows anyone to build tools that integrate seamlessly with the environment and other tools. The method to integrate tools seamlessly is by plug-ins. In fact everything except a small runtime kernel is a plug-in in Eclipse. [23] Eclipse platform on the other hand is a sub-project of Eclipse which provides the core frameworks and services upon which all plug-in extensions are created. [24] Eclipse platform is the core component of Eclipse IDE and everything else i.e. subprojects are built on top of it.

The platform is defined by components whose development is handled as their own projects. These projects include Ant integration, platform runtime and resource management, CVS integration, generic execution debug framework, release engineering, integrated search facility, Standard Widget Toolkit, generic team and compare support frameworks, text editor framework, help system, initial user experience, cheat sheets etc., platform user interface and dynamic update/install/field service. [23] These projects form a generic development base for new extensions.

2.6 What is Eclipse CDT?

The CDT (C/C++ Development Tools) project goal is to provide C and C++ Integrated Development Environment (IDE) for the Eclipse platform. [25] That implies that Eclipse CDT is a plug-in itself.

2.7 What is Eclipse plug-in?

Plug-in provides additional functionality to Eclipse IDE platform. Plug-ins can be made by using Plug-in Development Environment and utilizing Eclipse libraries. A plug-in requires an extension point to plug into in order to function. [26] Furthermore an update site project can be generated and uploaded to a web server and anyone can then install the plug-in by downloading it from online within Eclipse UI. Plug-ins are an easy way to add functionality as PDE provides tools for developers to help their work and the installation of the plug-ins is easy for the end-users.

Eclipse workbench and workspace provide an essential support for plug-ins. They contain extension points that can extend user interface with views, dialogs and events. Workspace's extension points allow to interact with resources e.g. projects and files. [26] On other words Eclipse components are extended by other plug-ins in order to achieve additional functionality. Debug and Release components for example are used to launch programs. Help component is useful for creating end-user documentation.

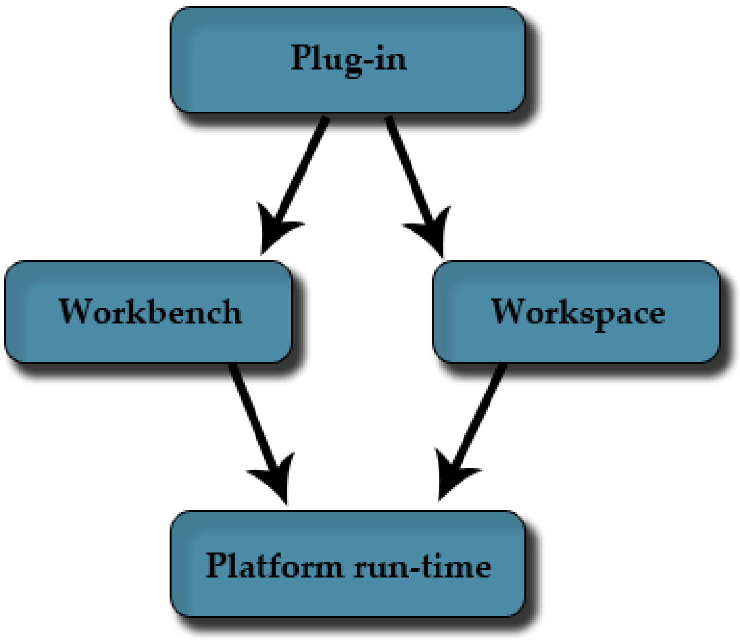


Figure 4. Eclipse extension point scheme

2.8 Initial preparations

Let's look at the initial preparations that are necessary in order to develop LLVM plug-in or just use it as an additional toolchain with Eclipse CDT.

2.8.1 Installing/compiling LLVM with front-ends

Preparations differ depending on the operating system in use. The most time-consuming method is to compile LLVM libraries along with front-ends especially on Windows. LLVM with Clang must always be built from sources on Windows. LLVM-GCC front-end binaries are provided for MinGW and are usually sufficient. Compiling LLVM-GCC can be tricky and very time-consuming thus recommended only if necessary.

2.8.1.1 Linux environment

In case of major Debian based Linux distribution like Ubuntu, all necessary LLVM related components can be installed via package manager. This is by far the fastest and easiest method to install LLVM with Clang and LLVM-GCC compilers. There are also two alternative options for Linux. Either to compile just LLVM libraries if distributions package manager does not include such a package and download front-end Linux binaries separately or compile front-ends from sources. Compilation takes quite a lot of time (even hours) hence it is recommended to compile only those packages that are necessary.

2.8.1.1.1 Compiling LLVM with Clang and LLVM-GCC front-ends on Linux

Note: These instructions are Ubuntu specific and may or may not work with other Linux distributions. Administrator rights are more than likely needed to run these commands.

Step	Instructions

1.	Download LLVM:
	`mkdir /usr/llvm`
	`cd /usr/src` (create if does not exist)
	`svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`

2.	Download Clang :
	`cd /usr/src/llvm/tools`
	`svn co http://llvm.org/svn/llvm-project/cfe/trunk clang`

3.	Configure and build LLVM with Clang:
	`mkdir /usr/build`
	`mkdir /usr/build/llvm`
	`cd /usr/build/llvm`
	`/usr/src/llvm/configure --prefix=/usr/llvm`
	`make -j# ENABLE_OPTIMIZED=1` (#=number of CPU cores)

4.	Download LLVM-GCC binaries: Download matching build of the LLVM-GCC binaries from http://llvm.org/releases.

5.	Install LLVM-GCC binaries:
	`extract llvm-gcc to /usr/build/llvm-gcc`
	`mkdir /usr/build/llvm-gcc`
	Copy extracted llvm-gcc files to `/usr/build/llvm-gcc`
	For example: `cp -r /../llvm-gcc4.2-2.8-x86_64-linux/*`
	`/usr/build/llvm-gcc` (replace .. with a correct subdir)

6.	Add paths permanently to PATH environment variable:
	`export PATH=/usr/build/llvm/Release+Asserts/bin:${PATH}`
	`export PATH=/usr/build/llvm-gcc/bin:${PATH}`
	For example in Ubuntu the above lines should be added to `.bashrc` file:
	`gedit ~/.bashrc`
	and enable/update PATH variable: `source ~/.bashrc`

Table 2. Compiling LLVM with Clang and LLVM-GCC front-ends on Linux

2.8.1.2 Windows environment

There is no native way to install LLVM for Windows. Either Cygwin or MinGW must be installed which provide a Linux-like-environment. This complicates installation quite a lot, because Cygwin/MinGW needs additional software to be installed and thus takes more time.

LLVM libraries and Clang front-end must be compiled with MinGW. LLVM-GCC front-end is available as binary format. During MinGW installation MSYS and g++ compiler need to be installed as well. MSYS version of Perl and libcrypt

(Perl dependency) must be uncompressed into MSYS directory. Also binutils package needs to be uncompressed into directory where LLVM-GCC MinGW binaries were uncompressed if they were not compiled from sources. MinGW bin directory, LLVM with Clang and LLVM-GCC front-end binary paths must be added to PATH environment variable.

Cygwin needs more software to be installed and LLVM libraries with all front-ends must be compiled. Getting LLVM compiler system working under Windows takes more time with Cygwin than MinGW. Cygwin bin directory and LLVM binary directory paths must be added to PATH environment path. Also CYGWIN environment variable with value nodosfilewarning=0 must be added, because by default Cygwin expects POSIX style path which results in error in Eclipse without this solution.

2.8.1.2.1 LLVM installation instructions for Windows using MinGW compiler suite

This might be the fastest, easiest and most of all the most likely to work solution to get LLVM suite and front-ends working on Windows.

These instructions are tested to work with LLVM 2.8, Clang 2.8, LLVM-GCC 4.2 and newest MinGW installation (as of February 2011). These instructions are not version dependent but download links are given to the newest versions available at the moment of writing.

Step	**Instructions**
1.	Download LLVM and Clang source code and MinGW binary of LLVM-GCC. Download site: http://llvm.org/releases/download.html#2.8 or download the newest version of LLVM and Clang sources and LLVM MinGW binaries from http://llvm.org/releases/.
2.	Uncompress LLVM sources e.g. `C:/llvm-2.8`
3.	Uncompress Clang sources to tools directory inside LLVM source directory e.g. `C:/llvm-2.8/tools` and check/rename that the added

	directory is simple called 'clang' with no suffix e.g. version number.
4.	Download MinGW compiler suite. Download site: http://sourceforge.net/projects/mingw/
5.	Install MinGW (Download latest repository catalogues) with MSYS and C & C++ compiler which are checked in Wizard type installation.
6.	Add Ming’s "bin" directory to the PATH environment variable. e.g. `C:\MinGW\bin` (Run sysdm.cpl and click Environment Variables... button).
7.	Download Perl MSYS binaries. Download link: http://sourceforge.net/projects/mingw/files/MSYS/perl/perl-5.6.1_2-2/perl-5.6.1_2-2-msys-1.0.13-bin.tar.lzma/download or download newest binary package from http://sourceforge.net/projects/mingw/files/MSYS/perl/.
8.	Uncompress Perl binary package into MSYS/1.0 directory (merge bin & lib folders).
9.	Download libcrypt MSYS binaries (Perl requires it). Download newest version from http://sourceforge.net/projects/mingw/files/MSYS/crypt/.
10.	Uncompress libcrypt binary package into MSYS/1.0 directory. (merge bin folders).
11.	Open MSYS command prompt (e.g. `C:/MinGW/msys/1.0/msys.bat`) and navigate to directory where your LLVM sources are located e.g. `cd C:/llvm-2.8`
12.	Create folder for build files. e.g. `mkdir C:/llvm-2.8/BUILD`
13.	Go to BUILD directory. e.g. `cd BUILD`
14.	Run configure script: `../configure`
15.	Build the LLVM suite: `make -j# ENABLE_OPTIMIZED=1` (-j number_of_processor_cores for parallel compilation) (`ENABLE_OPTIMIZED=1` to perform a Release (Optimized) build and `ENABLE_OPTIMIZED=0` to perform a Debug build). If building fails for some reason, try building it again or alternatively run '`make clean`' command first. If building gets stuck at some point (as it does quite often) close the shell and run the make command again.

16.	Add `C:\llvm-2.8\BUILD\Release+Asserts\bin` (or `../Debug+Asserts../bin`) directory to your `PATH`.
17.	Uncompress MinGW LLVM-GCC binary files to some directory e.g. `C:/llvm-gcc.`
18.	Uncompress the binary binutils MinGW package into your LLVM-GCC binary directory. Download link: http://sourceforge.net/projects/mingw/files/MinGW/BaseSystem/GNU-Binutils/binutils-2.21/binutils-2.21-2-mingw32-bin.tar.lzma/download or download the newest binary package from http://sourceforge.net/projects/mingw/files/MinGW/BaseSystem/GNU-Binutils.
19	Add LLVM-GCC's "bin" directory to your PATH environment variable. e.g. `C:/llvm-gcc/bin`

Table 3. LLVM installation instructions for Windows using MinGW compiler suite

2.8.1.2.2 LLVM installation instructions for Windows using Cygwin compiler suite

Step	**Instructions**
1.	Download Cygwin from http://www.cygwin.com.
2.	Install Cygwin with following packages: Development gcc4-core gcc4-g++ make subversion Interpreters perl
3.	Open Cygwin bash shell.

4.	Create directories.
	`mkdir /usr/build`
	`mkdir /usr/build/llvm`
	`mkdir /usr/build/llvm-gcc`

5.	Get LLVM, Clang and LLVM-GCC sources.
	`cd /usr/src`
	`svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
	`svn co http://llvm.org/svn/llvm-project/llvm-gcc-4.2/trunk llvm-gcc`
	`cd /usr/src/llvm/tools`
	`svn co http://llvm.org/svn/llvm-project/cfe/trunk clang`

6.	Build LLVM with Clang (release build).
	`cd /usr/build/llvm`
	`/usr/src/llvm/configure --prefix=/usr/llvm`
	`make -j# ENABLE_OPTIMIZED=1 (#=number of cpu cores for parallel compilation)`
	`PATH=/usr/build/llvm/Release+Asserts/bin:${PATH}`

7.	Build and install the front-end.
	`cd /usr/build/llvm-gcc`
	`/usr/src/llvm-gcc/configure --enable-languages=c,c++- --enable-llvm=/usr/build/llvm`
	`make -j#  ENABLE_OPTIMIZED=1 (#=number of cpu cores)`
	`make install`
	`PATH=path_to_llvm-gcc_bin_dir:${PATH}`

8.	Rebuild and install the LLVM.
	`cd /usr/build/llvm`
	`make -j# ENABLE_OPTIMIZED=1`
	`make install`

9.	Check build. `make check`

Table 4. LLVM installation instructions for Windows using Cygwin

2.8.2 Development configuration: Installing Eclipse PDE with CDT SDK

In order to create Eclipse plug-ins, Plug-in Development Tools must be included in Eclipse installation along with CDT SDK. CDT packages must be imported into workspace as source projects. LLVM plug-in source codes can be checked out from SVN using subversion/subversive SVN client within Eclipse. CDT SDK is needed, because LLVM plug-in has many dependencies to CDT sources. CDT SDK should be newest possible, because along the process many CDT source files had to be modified to suit LLVM plug-in's needs and those patched versions of files are included in the newer versions of CDT (>8.0.0).

2.8.3 End-user configuration: Installing Eclipse CDT

End-user who does not plan to develop LLVM plug-in and just use it as a toolchain to compile C/C++ program code needs only to install Eclipse IDE for C/C++ Developers or Eclipse IDE with CDT plug-in and LLVM plug-in from Eclipse update site. Similar to development environment LLVM with front-ends must be built and set, because LLVM plug-in is dependent on those binaries.

2.8.4 System Environment variable settings

Regardless of the operating system LLVM back-end and front-end binaries shall be added to the PATH environment variable. In UNIX derived operating systems LD_LIBRARY_PATH environment variable should be appended with C++ Standard Library path. These paths can also be added via plug-in's LLVM preference page.

2.9 Creating an Eclipse plug-in

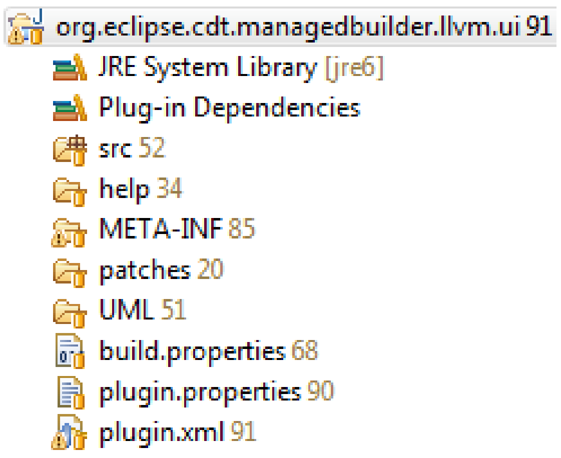
Creating working versions of LLVM-plug-in for Eclipse was a multiple step process. First PDE (Plug-in-Development) environment advices to create plug-in project, feature project and update site for just one project after all. Sooner you will notice that update site must be transferred to a web-server in order that anybody has access to download the plug-in you just made. This plug-in can be fully open-source, but Eclipse allows using licenses for commercial use too.

2.9.1 Creating a plug-in wizard template

Creating a plug-in template goes through many internal features of Eclipse platform. Eclipse has come up with an easy to use wizard, but an overall plan for the plugin must be made first. Project wizard asks for project name, version, vendor and some other settings.

2.9.2 Structure of the Eclipse plug-in

Plug-in project in Eclipse workspace contains at least plug-in project based dependencies, Java libraries, plugin.xml file, manifest file, some property files and project source files.



```
org.eclipse.cdt.managedbuilder.llvm.ui 91
    JRE System Library [jre6]
    Plug-in Dependencies
    src 52
    help 34
    META-INF 85
    patches 20
    UML 51
    build.properties 68
    plugin.properties 90
    plugin.xml 91
```

Figure 5. Eclipse project tree

2.10 Creating feature project

An Eclipse Feature project must be created in to the workspace in order to create an Eclipse update site project. At least ID, version, name and provider along with supported operating systems are necessary to be set. Feature description, copyright notice and license agreement were also written in LLVM feature project. Finally a plug-in must be added to the feature and dependencies checked and added if necessary. All of these configurations are done in feature.xml file.

2.11 Creating Eclipse update site

An update site project shall be created into the workspace after an Eclipse feature project is created. Category is first created and then feature added to that category. ID, name and description are written in category properties. Clicking ‘build all’ command in site.xml file’s site tab generates files that are necessary for the update site.

The files contained in the update site project must be uploaded to a web server preferably under a directory named “update”. Now update site can be added in Eclipse by navigating Help -> Install New Software… Add. After selecting the added update site URL the plug-in can be installed. During the installation process plug-in description, version etc. are shown and license must be accepted.

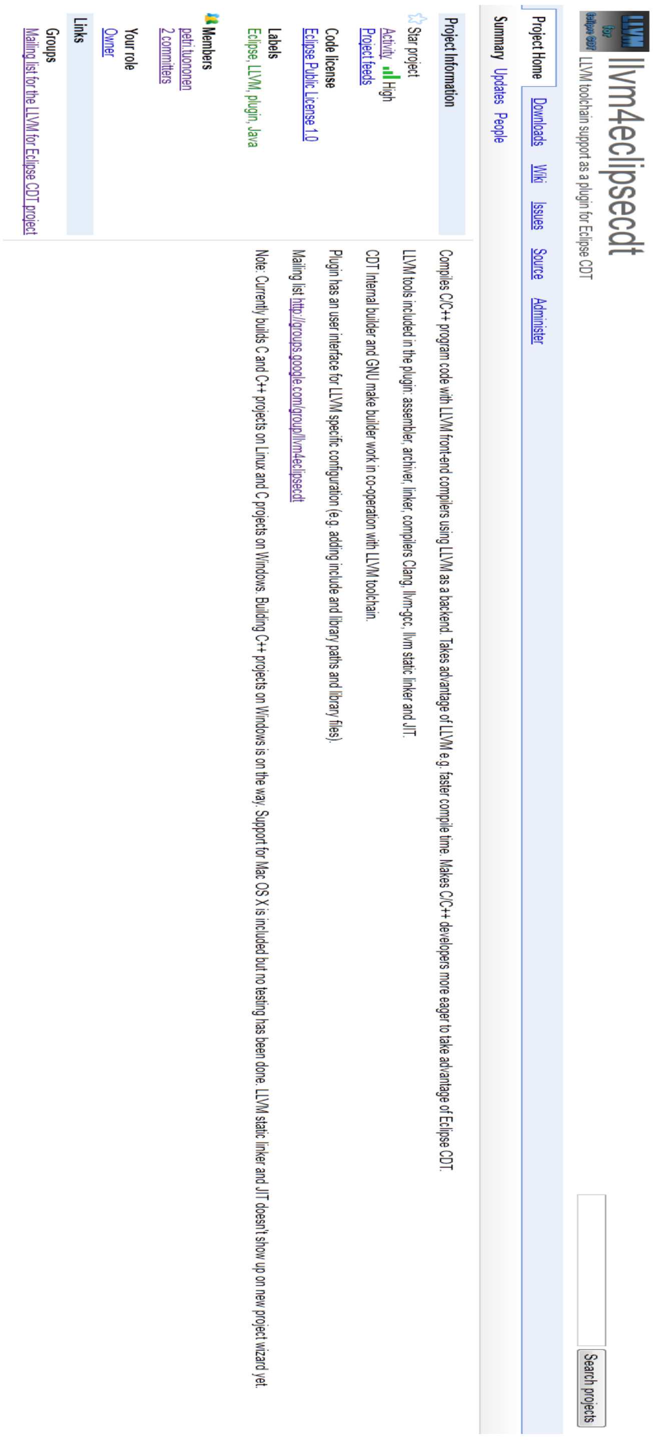
2.11.1 Hosting for the project update site

A web server is needed to host the contents of the update site project. Initially my personal domain and hosting server is used to host the plug-in update site but later on when the plug-in is integrated into the CDT release the Eclipse Foundation will host the plug-in source files along with the update site.

2.12 Project website

Project website contains general information about the project as well as wiki pages, downloads, source code browsing, commits and other updates, issue tracker and project members. Anyone can apply to become a committer or a contributor for the project by contacting project owner and specifying the reason how he/she would like to contribute towards the project.

Link to the project website: http://code.google.com/p/llvm4eclipsecdt/.



Project Information

Star project

Activity High

Project feeds

Code license

Eclipse Public License 1.0

Labels

Eclipse, LLVM, plugin, Java

Members

petri.tuononen

2 committers

Your role

Owner

Links

Groups

Mailing list for the LLVM for Eclipse CDT project

Compiles C/C++ program code with LLVM front-end compilers using LLVM as a backend. Takes advantage of LLVM e.g. faster compile time. Makes C/C++ developers more eager to take advantage of Eclipse CDT.

LLVM tools included in the plugin: assembler, archiver, linker, compilers Clang, llvm-gcc, llvm static linker and JIT.

CDT Internal builder and GNU make builder work in co-operation with LLVM toolchain.

Plugin has an user interface for LLVM specific configuration (e.g. adding include and library paths and library files).

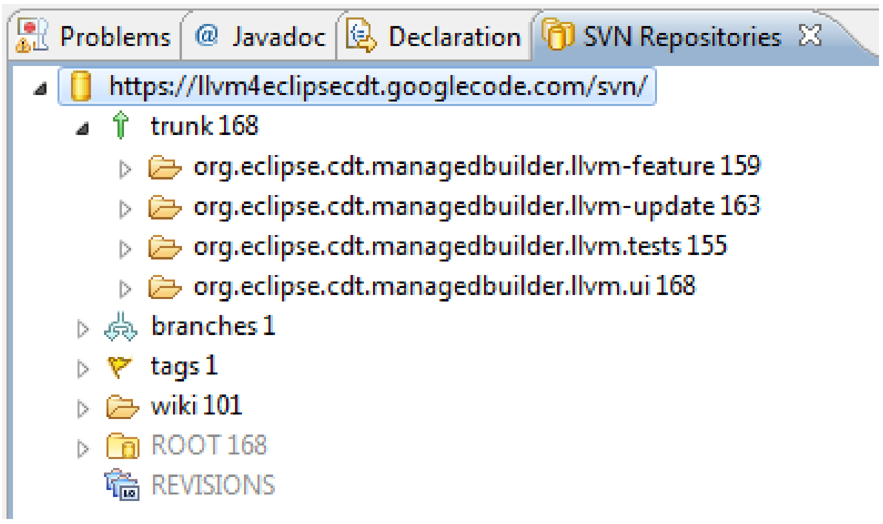
Mailing list http://groups.google.com/group/llvm4eclipsecdt

Note: Currently builds C and C++ projects on Linux and C projects on Windows. Building C++ projects on Windows is on the way. Support for Mac OS X is included but no testing has been done. LLVM static linker and JIT doesn't show up on new project wizard yet.

Figure 6. Project website

2.12.1 Version tracking

Subversion version tracker was used to keep track of new revisions and to be able to revert back and compare different revisions. Version tracker provides a secure way to manage project and allows others to participate too by creating new access rights to the SVN repository.



```
Problems | Javadoc | Declaration | SVN Repositories
◢ https://llvm4eclipsecdt.googlecode.com/svn/
    ◢ trunk 168
        ▷ org.eclipse.cdt.managedbuilder.llvm-feature 159
        ▷ org.eclipse.cdt.managedbuilder.llvm-update 163
        ▷ org.eclipse.cdt.managedbuilder.llvm.tests 155
        ▷ org.eclipse.cdt.managedbuilder.llvm.ui 168
    ▷ branches 1
    ▷ tags 1
    ▷ wiki 101
    ▷ ROOT 168
      REVISIONS
```

Figure 7. Subversion repository tree

2.12.2 Issue tracking

An issue tracker was maintained as a source of keeping list of bugs and creating new entries of issues that may need to be fixed. Anonymous users were given rights to write new entries and some people came up with a decent and helpful manner to find defects in the project.

2.12.3 Wiki

Project wiki is a place for documentation aimed for developers. It contains instructions on how to setup the development environment suitable for LLVM plug-in. It also informs about necessary dependencies.

2.13 Project mailing list

Anyone can subscribe to LLVM plug-in for Eclipse CDT project development mailing list to get all development specific updates directly to his/her email address. Mailing list contains discussions and notices of project commits, new issues and wiki changes. Person can also view the Google group site in regular basis if he/she does not want email messages. However if one wants to be aware of the latest changes what happens in the project development then subscription to the mailing list is recommended. Everyone can subscribe to the list.

Link to project mailing list: http://groups.google.com/group/llvm4eclipsecdt/.

2.14 Contributing to the Eclipse Foundation

The company Nokia Siemens Networks where I worked for and started developing this plug-in allowed me to publish this project under an open-source license (EPL) and contribute towards Eclipse Foundation. I immediately liked the idea for multiple reasons. The number one reason is that the open-source license allows everyone to enjoy the achievements of the work which ultimately leads to LLVM’s wider use in C/C++ project development. Contributing to the CDT on the other hand forms an integration that makes sense, because LLVM plug-in is also aimed to compile C/C++ code. CDT community might also become useful by providing development help and pretty good guarantee that development is continued.

There are specific rules how to communicate with CDT developers and how to contribute to Eclipse Foundation. CDT developers can be contacted by posting to CDT developers’ mailing list. It is worth to remind though that this is not a CDT support mailing list. Instead its purpose is to function as a channel between CDT developers. I posted a thread in order to notify developers that this is coming to CDT on some point and also to spark interest and gain their approval. I found out that the proper way to include this kind of project into the CDT would be to create an Eclipse Bugzilla enhancement entry under CDT tools. Contribution instructions are written to CDT Wiki. I obtained a fair share of encouragement by kind feedback and some development ideas from the CDT-DEV mailing list responders.

I was told that it was too late for CDT 8.0 release as it might take some months for IP (Intelligence Property) reviewers to take a look and approve the code. The CDT developers also want to test the new major release carefully and thus they build multiple milestone versions and nightly builds. A major addition like this might compromise the overall stability and reliability of the CDT. I completely agree with them as I want to make sure that the plug-in works as well as possible before listing an enhancement entry to Bugzilla. There were still multiple issues that had to be fixed and further testing to be done on other operating systems than Linux distributions. This gave me more time to finish the plug-in the way I wanted, because I did not need to rush in order to make it to the next CDT release.

3 REQUIREMENT ANALYSIS

The mission was to integrate LLVM toolchain with Eclipse CDT. The middleman in this composition was the CDT plug-in for Eclipse which provides C/C++ developing environment on top of Eclipse platform. Libraries from the CDT project had to be referenced especially from org.eclipse.cdt.managedbuilder.core package, which can be seen from Appendix 2.

3.1 Functional specifications

3.1.1 Integration with a popular graphical IDE

LLVM plug-in for Eclipse IDE does not already exist even though Eclipse is one of the most popular open-source IDEs. Clang main page mentions “Allow tight integration with IDEs”. Eclipse has a great functionality to help produce plug-ins and its open-source library makes the creations legally possible.

IDE has a visibility across the entire project thus sharing an address space across multiple files which provides intelligent caching and other techniques that reduce analysis/compilation time. [11]

3.1.2 Supported platforms

The plug-in can be used on all operating systems which support Eclipse and have some sort of Unix-like emulator which enables to execute LLVM binaries. Top priority in the beginning is to support Linux, Mac OS X and Windows.

3.1.3 LLVM tools

Tools that shall be implemented in the LLVM plug-in are assembler, archiver, linker, static compiler, execution engine aka JIT, Clang and LLVM-GCC compiler.

3.1.4 Tool options

Every tool has its specific options and some of them have to be included and some are optional. This had to be taken into account by thoroughly reading documentation and configuring all the options (most of which no value was given). User must be able to change tool parameters by using a user-friendly user Interface.

3.1.5 LLVM front-ends

Clang and LLVM-GCC front-ends needs to be added to the plug-in, because they are the front-ends that compile C/C++ code.

3.1.6 CDT Internal builder and GNU make builder support for LLVM toolchain

Eclipse features two builders by default and the plan is to provide support for both of them.

3.1.7 User interface for LLVM specific configuration

There shall be a GUI dialog where specific workspace-wide LLVM configurations can be set.

3.1.8 Additional system environment path variables

Some additional system environment path variables need to be set in order to find all the dependent LLVM binaries and C++ Standard Library.

3.2 Contributing project as part of Eclipse CDT for Eclipse Foundation

It is advantageous to release the plug-in with EPL (Eclipse Public License) and donate it to Eclipse Foundation to ensure further development of the plug-in. As a result everyone can benefit from the plug-in and wider audience promotes it and thus drives the development onward.

3.3 LLVM integration should not disturb initial functionality of CDT

In order to integrate LLVM plug-in as a part of the official CDT release it is highly important that the plug-in does not affect the initial functionality of CDT and that license is compatible. LLVM plug-in cannot ship with LLVM tools to ensure that it meets all CDT project licensing rights.

4 IMPLEMENTATION

4.1 Overview of the LLVM plug-in architecture

The LLVM plug-in architecture currently consists of 5 packages. As can be seen from the diagram below the plug-in has major dependencies to CDT project packages including org.eclipse.cdt.make.internal.core.scannerconfig, org.eclipse.cdt.make.internal.core.scannerconfig.gnu, org.eclipse.cdt.build.internal.core.scannerconfig2, org.eclipse.cdt.build.core.scannerconfig, org.eclipse.cdt.core.settings.model, org.eclipse.cdt.core.model, org.eclipse.cdt.managedbuilder.makegen.gnu and org.eclipse.cdt.core.

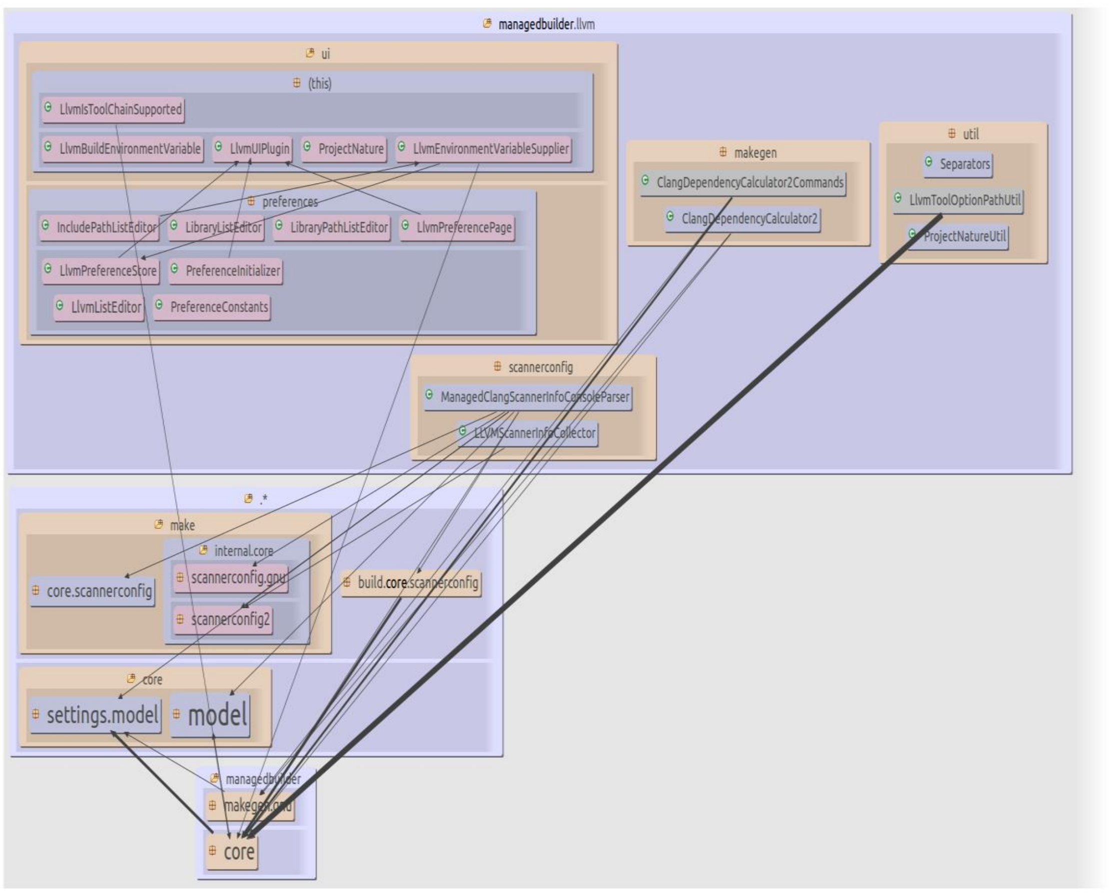


Figure 8. LLVM project with Eclipse package dependencies

For a better view, see Appendix 2.

The diagram below shows relations between LLVM plug-in packages and classes.

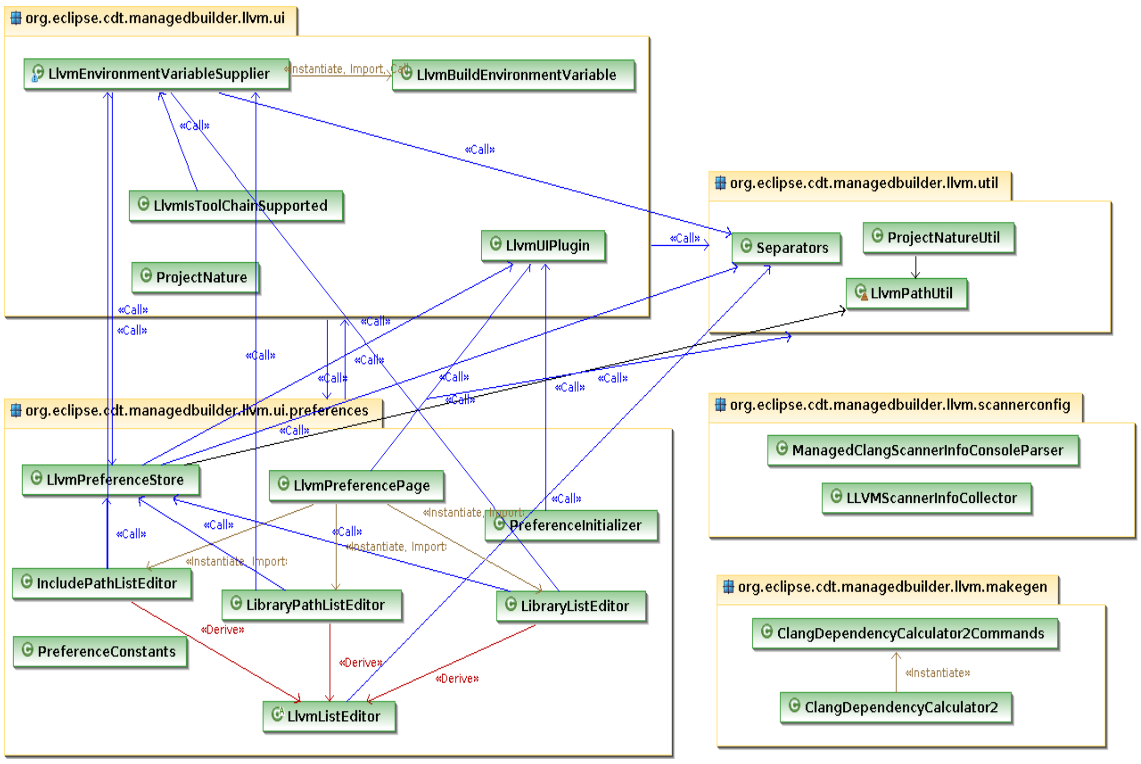

org.eclipse.cdt.managedbuilder.llvm.ui
LlvmEnvironmentVariableSupplier
«Instantiate, Import, Call»
LlvmBuildEnvironmentVariable
«Call»
LlvmIsToolChainSupported
LlvmUIPlugin
ProjectNature
org.eclipse.cdt.managedbuilder.llvm.util
Separators
ProjectNatureUtil
LlvmPathUtil
org.eclipse.cdt.managedbuilder.llvm.scannerconfig
ManagedClangScannerInfoConsoleParser
LLVMScannerInfoCollector
org.eclipse.cdt.managedbuilder.llvm.ui.preferences
LlvmPreferenceStore
LlvmPreferencePage
PreferenceInitializer
IncludePathListEditor
LibraryPathListEditor
LibraryListEditor
PreferenceConstants
LlvmListEditor
«Instantiate, Import»
«Derive»
org.eclipse.cdt.managedbuilder.llvm.makegen
ClangDependencyCalculator2Commands
«Instantiate»
ClangDependencyCalculator2

Figure 9. LVM project class dependencies and inheritances

4.2 CDT bug/feature patches

CDT contained bugs that came on the way and had to be corrected first. Necessary functionality missing from the released CDT 7.x had to be added. Patches were submitted via Eclipse Bugzilla for CDT project and all of the patches are included in the next major release of CDT (8.0). This means that at least certain functionality may not work with older versions of CDT, but backward compatibility may be provided by creating multiple versions of the plug-in.

One of the added functionalities were getting library search paths from tool's option in managed build system (MBS) which can be seen in figure 'Managed build model elements'. This is listed in Eclipse Bugzilla as Bug 321040.

A rare API change that affects plug-in's backward compatibility was also created. This was not really necessary but as the deprecated class org.eclipse.core.runtime.PluginVersionIdentifier showed as a warning on IDE I decided to change it to org.osgi.framework.Version. The class was tagged as deprecated meaning that it would be replaced at some point anyway. I thought that better sooner than later so I created a patch and attached it to a bug entry I created. This API change is from 7.1.0 onwards. This is listed in Eclipse Bugzilla as Bug 318581.

4.3 Extending the Eclipse CDT Managed Build System

Eclipse CDT architecture consists of external and internal components. Internal part consists of build file generator, CDT parser and UI component. External part consists of UI elements and tool integrator. In order to add LLVM toolchain support for CDT, additions through tool integrator and UI interfaces had to be made.

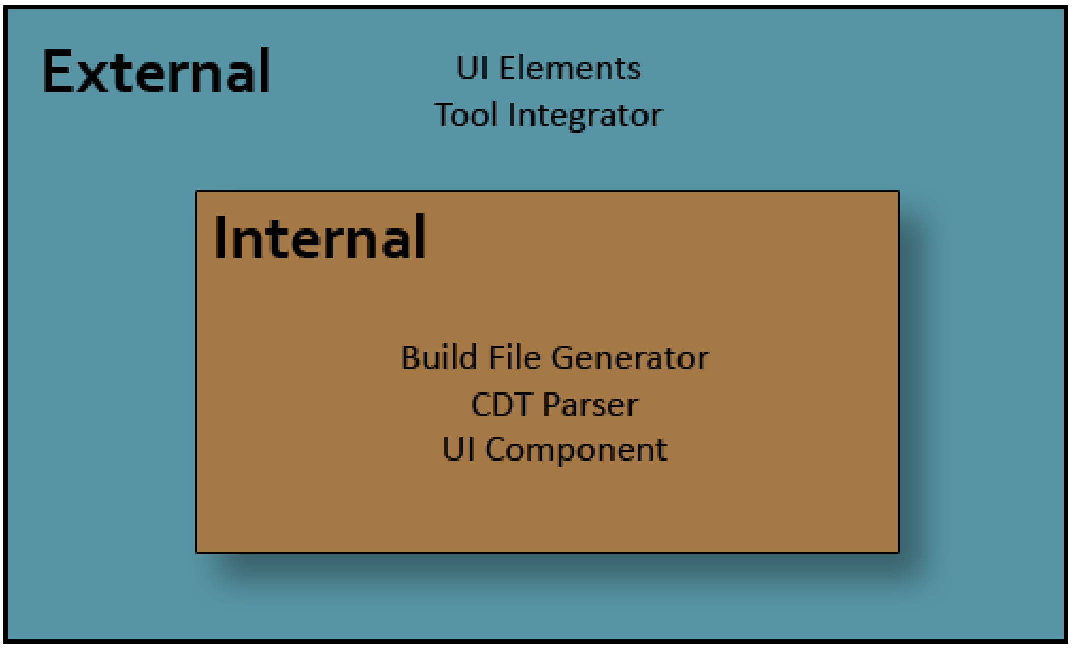


Figure 10. CDT Managed Build System Architecture

Managed build model schema shows the architecture behind build configurations. The figure below instructs how different elements are linked together. This model came very handy in order to program functions which e.g.

add library path to debug/release configuration’s ‘LLVM with Clang’ toolchain’s linker tool where library path is an option.

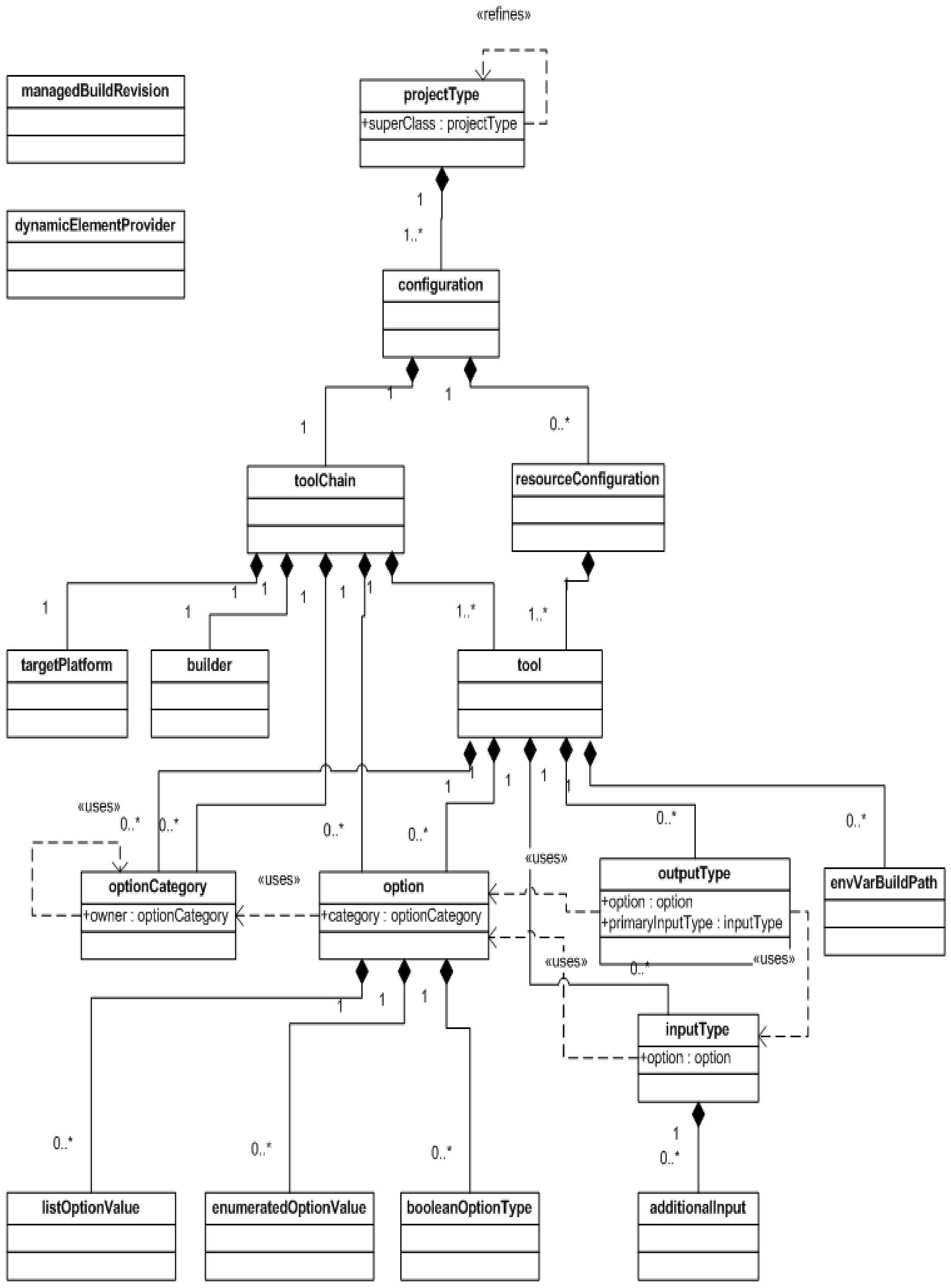


Figure 11. Managed build model elements [27]

LLVM plug-in is basically a new toolchain added to Eclipse CDT. LLVM toolchain definitions were added by plugging into the MBS tool definition extension point, org.eclipse.cdt.managedbuilder.core.buildDefinitions. The extension point defines an XML schema that lets describe tools in the toolchain.

4.3.1 Configuring LLVM tools for Eclipse

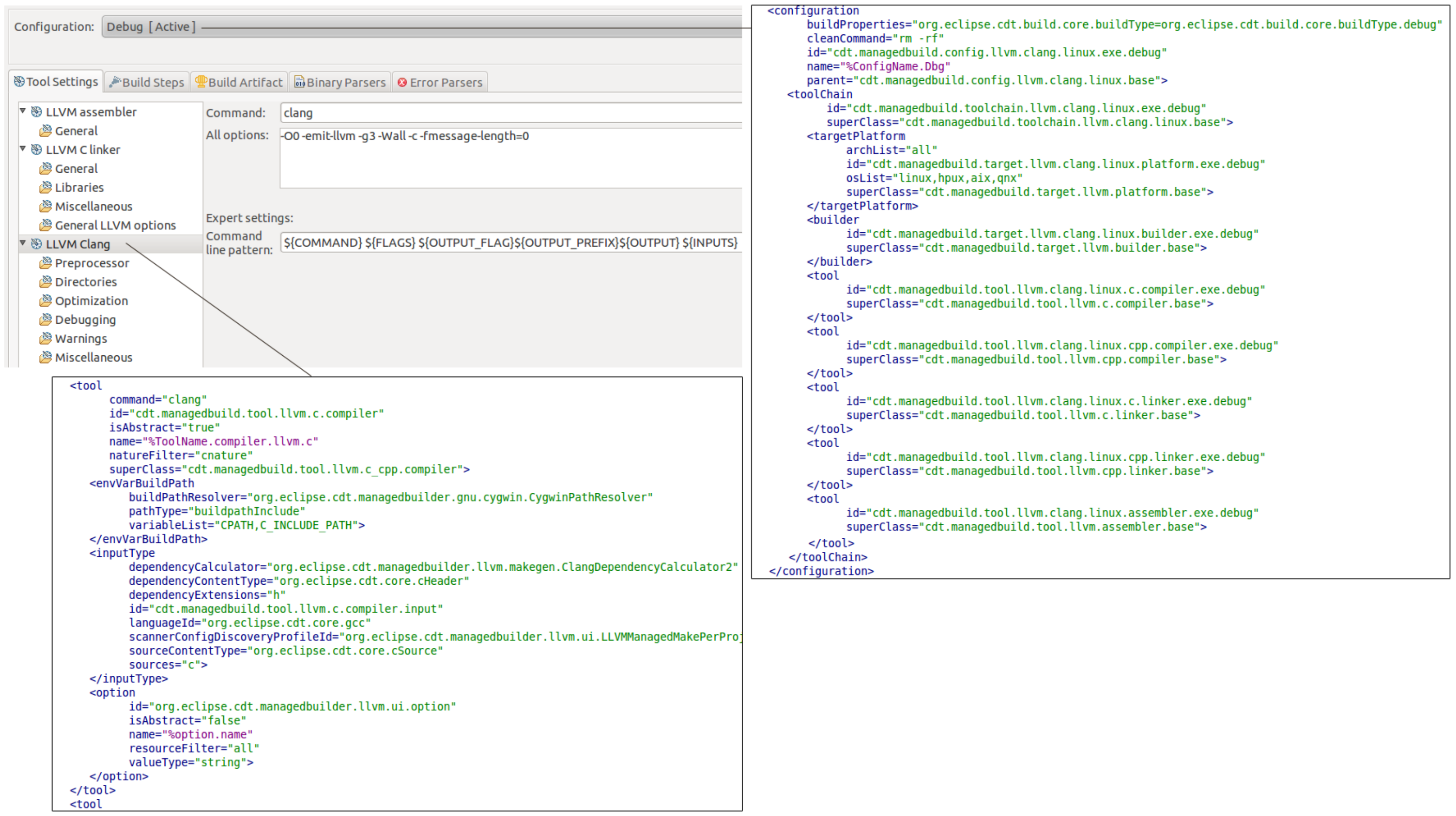
Tools that needed to be implemented in the LLVM plug-in were assembler, archiver, linker, static compiler, execution engine aka JIT and Clang and LLVM-GCC compilers.

LLVM tools were configured for Eclipse by managing plug-in extensions in plugin.xml file. This was one of the most crucial steps in creating the LLVM plug-in for Eclipse CDT. Build definitions were created in such a way that toolchains configured for each platform and compiler (in case of Windows platform also Unix-like environment) inherit as much as possible from abstract toolchains that share similar options. Additional dissimilarity between platforms is the executable format (output type).

Every toolchain configuration includes LLVM versions of archiver, assembler, linker and CDT internal builder or Make builder which are provided by abstract LLVM toolchain which acts as the most upper-level container. Every toolchain must also include one of the four front-ends which are Clang, LLVM-GCC, LLVM static compiler and JIT compiler. Platform specific Clang with LLVM and LLVM-GCC with LLVM toolchains are themselves inherited from LLVM with C/C++ Linker which is further inherited from highly abstract LLVM toolchain which includes similar options for all LLVM based toolchains (builder, assembler and archiver).

See Appendix 1. to get a clear view of the architectural design of toolchain, tool and platform configurations.

The configuration of tools can be done through plugin.xml file's extensions tab in Eclipse's GUI or optionally manually by writing pure xml code. The figure below shows C/C++ build settings when LLVM plug-in is installed. The xml code demonstrates how build configuration (xml code on East side) and tool (xml code on South) settings have been formed. Note that some Clang tool settings are inherited from Abstract LLVM C/C++ compiler tool as can be seen from Appendix 1. This means that some settings are not visible on the provided xml listing. The plugin.xml file alone contains ~5000 lines of code.



Configuration: Debug [Active]

Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

- LLVM assembler
  - General
- LLVM C linker
  - General
  - Libraries
  - Miscellaneous
  - General LLVM options
- LLVM Clang
  - Preprocessor
  - Directories
  - Optimization
  - Debugging
  - Warnings
  - Miscellaneous

Command: clang

All options: -O0 -emit-llvm -g3 -Wall -c -fmessage-length=0

Expert settings:

Command line pattern: ${COMMAND} ${FLAGS} ${OUTPUT_FLAG}${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}

```
<tool
      command="clang"
      id="cdt.managedbuild.tool.llvm.c.compiler"
      isAbstract="true"
      name="%ToolName.compiler.llvm.c"
      natureFilter="cnature"
      superClass="cdt.managedbuild.tool.llvm.c_cpp.compiler">
   <envVarBuildPath
         buildPathResolver="org.eclipse.cdt.managedbuilder.gnu.cygwin.CygwinPathResolver"
         pathType="buildpathInclude"
         variableList="CPATH,C_INCLUDE_PATH">
   </envVarBuildPath>
   <inputType
         dependencyCalculator="org.eclipse.cdt.managedbuilder.llvm.makegen.ClangDependencyCalculator2"
         dependencyContentType="org.eclipse.cdt.core.cHeader"
         dependencyExtensions="h"
         id="cdt.managedbuild.tool.llvm.c.compiler.input"
         languageId="org.eclipse.cdt.core.gcc"
         scannerConfigDiscoveryProfileId="org.eclipse.cdt.managedbuilder.llvm.ui.LLVMManagedMakePerProj
         sourceContentType="org.eclipse.cdt.core.cSource"
         sources="c">
   </inputType>
   <option
         id="org.eclipse.cdt.managedbuilder.llvm.ui.option"
         isAbstract="false"
         name="%option.name"
         resourceFilter="all"
         valueType="string">
   </option>
</tool>
<tool
```

```
<configuration
      buildProperties="org.eclipse.cdt.build.core.buildType=org.eclipse.cdt.build.core.buildType.debug"
      cleanCommand="rm -rf"
      id="cdt.managedbuild.config.llvm.clang.linux.exe.debug"
      name="%ConfigName.Dbg"
      parent="cdt.managedbuild.config.llvm.clang.linux.base">
   <toolChain
         id="cdt.managedbuild.toolchain.llvm.clang.linux.exe.debug"
         superClass="cdt.managedbuild.toolchain.llvm.clang.linux.base">
      <targetPlatform
            archList="all"
            id="cdt.managedbuild.target.llvm.clang.linux.platform.exe.debug"
            osList="linux,hpux,aix,qnx"
            superClass="cdt.managedbuild.target.llvm.platform.base">
      </targetPlatform>
      <builder
            id="cdt.managedbuild.target.llvm.clang.linux.builder.exe.debug"
            superClass="cdt.managedbuild.target.llvm.builder.base">
      </builder>
      <tool
            id="cdt.managedbuild.tool.llvm.clang.linux.c.compiler.exe.debug"
            superClass="cdt.managedbuild.tool.llvm.c.compiler.base">
      </tool>
      <tool
            id="cdt.managedbuild.tool.llvm.clang.linux.cpp.compiler.exe.debug"
            superClass="cdt.managedbuild.tool.llvm.cpp.compiler.base">
      </tool>
      <tool
            id="cdt.managedbuild.tool.llvm.clang.linux.c.linker.exe.debug"
            superClass="cdt.managedbuild.tool.llvm.c.linker.base">
      </tool>
      <tool
            id="cdt.managedbuild.tool.llvm.clang.linux.cpp.linker.exe.debug"
            superClass="cdt.managedbuild.tool.llvm.cpp.linker.base">
      </tool>
      <tool
            id="cdt.managedbuild.tool.llvm.clang.linux.assembler.exe.debug"
            superClass="cdt.managedbuild.tool.llvm.assembler.base">
      </tool>
   </toolChain>
</configuration>
```

Figure 12. C/C++ Build settings

4.3.2 Configuring template associations

Eclipse's project wizard defines few types of templates. LLVM based toolchains are added to EmptyProject, HelloWorldCAnsiProject and HelloWorldCCProject templates which are defined in org.eclipse.cdt.managedbuilder.gnu.ui package. Every toolchain configuration is defined as an id for those three templates. In other words templates are associated with toolchains.

4.3.3 Configuring content types

By extending org.eclipse.core.contenttype.contentTypes content types can be created which define id, name, base type (such as text) and file extension for a file type. File types can be associated with file extensions.

4.3.4 Configuring scanner discovery profiles

Scanner info collector, scanner info provider and scanner info console parser are needed in order to add built-in include paths, library paths and preprocessor definitions automatically to the paths and symbols preference page. Among these three classes scanner info console parser and scanner info collector are implemented from the GCC ones with minor modifications and scanner info provider points directly to GCC scanner info provider (org.eclipse.cdt.make.internal.core.scannerconfig2.GCCSpecsRunSIProvider).

Scanner info provider's function is to add correct information to the scanner info collector when it is being called. Plugin.xml extension org.eclipse.cdt.make.core.ScannerConfigurationDiscoveryProfile is added separately for clang and clang++.

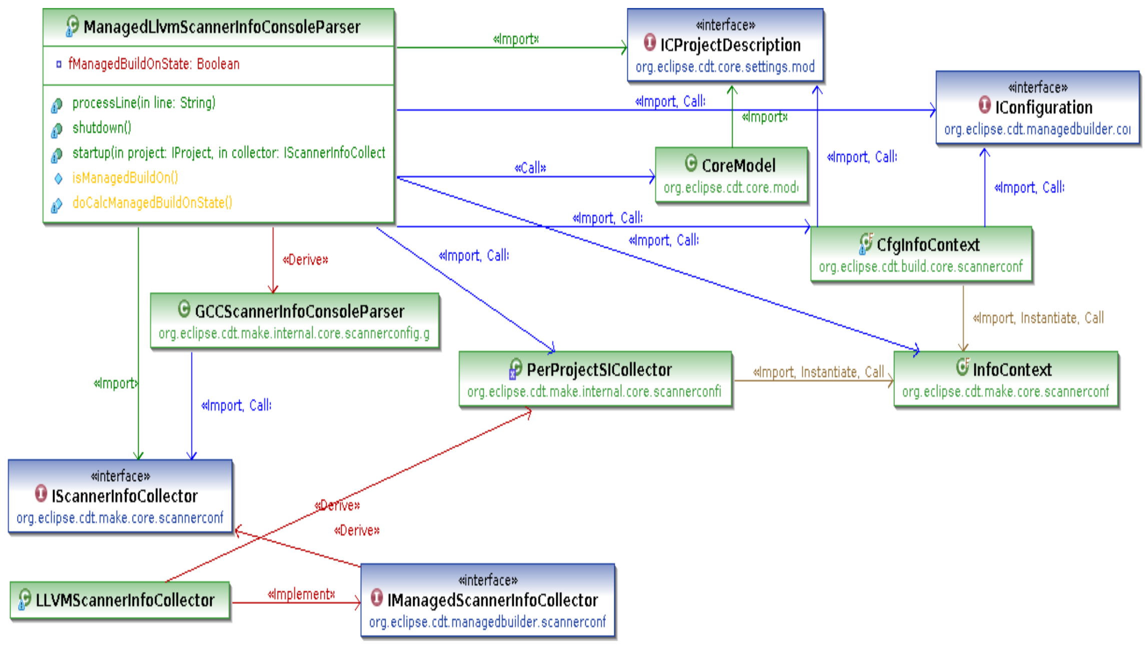

ManagedLlvmScannerInfoConsoleParser
fManagedBuildOnState: Boolean
processLine(in line: String)
shutdown()
startup(in project: IProject, in collector: IScannerInfoCollect
isManagedBuildOn()
doCalcManagedBuildOnState()
«interface»
ICProjectDescription
org.eclipse.cdt.core.settings.mod
«interface»
IConfiguration
org.eclipse.cdt.managedbuilder.co
CoreModel
org.eclipse.cdt.core.mode
CfgInfoContext
org.eclipse.cdt.build.core.scannerconf
GCCScannerInfoConsoleParser
org.eclipse.cdt.make.internal.core.scannerconfig.g
PerProjectSICollector
org.eclipse.cdt.make.internal.core.scannerconfi
InfoContext
org.eclipse.cdt.make.core.scannerconf
«interface»
IScannerInfoCollector
org.eclipse.cdt.make.core.scannerconf
LLVMScannerInfoCollector
«interface»
IManagedScannerInfoCollector
org.eclipse.cdt.managedbuilder.scannerconf
«Import»
«Import, Call»
«Call»
«Derive»
«Import, Instantiate, Call»
«Implement»

Figure 13. llvm.scannerconfig package

4.3.5 Configuring dependency calculators

Dependency calculator provides the dependency calculation for a given tool. LLVM specific dependency calculator was created by extending DefaultGCCDependencyCalculator2 class. Implemented dependency calculator’s two overloaded methods with same name getDependencySourceInfo return LLVM specific dependency calculator commands class which is extended from DefaultGCCDependencyCalculator2Commands.

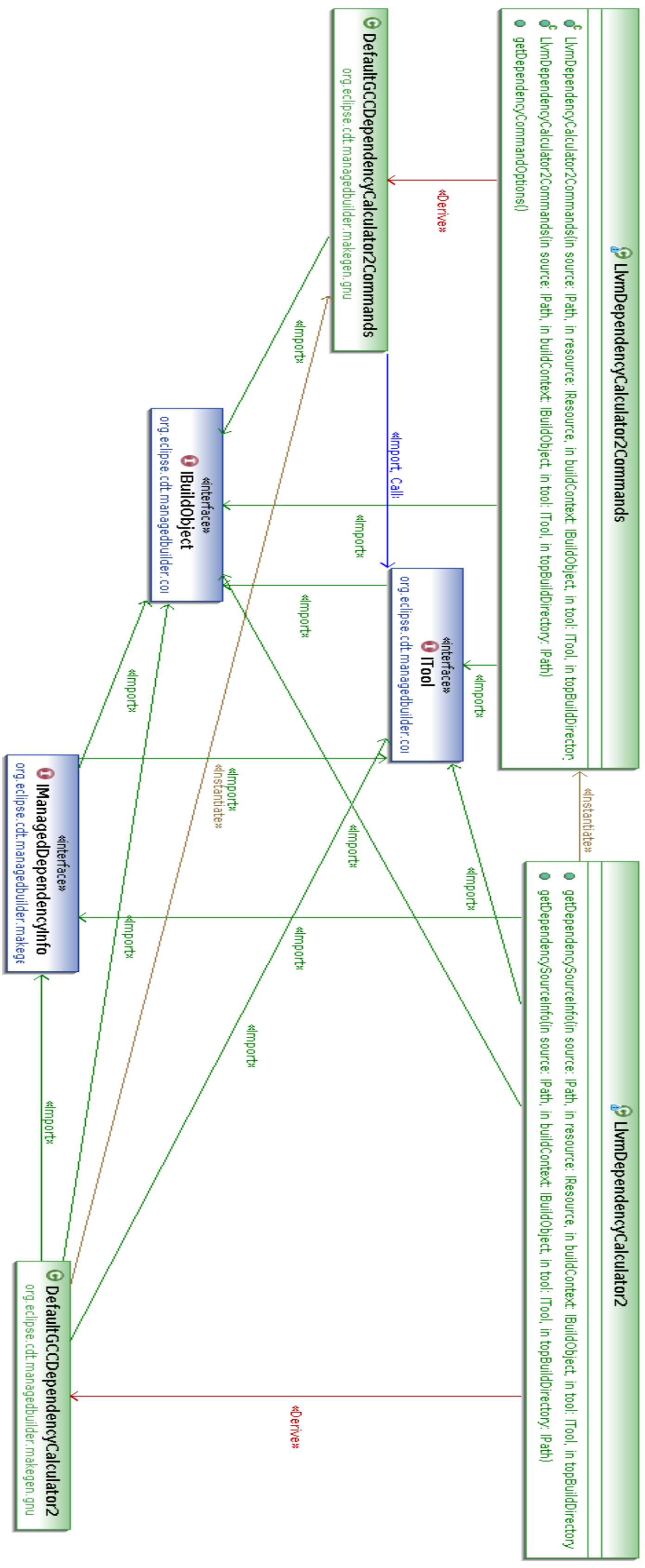

LlvmDependencyCalculator2Commands
LlvmDependencyCalculator2Commands(in source: IPath, in resource: IResource, in buildContext: IBuildObject, in tool: ITool, in topBuildDirectory
LlvmDependencyCalculator2Commands(in source: IPath, in buildContext: IBuildObject, in tool: ITool, in topBuildDirectory: IPath)
getDependencyCommandOptions()
DefaultGCCDependencyCalculator2Commands
org.eclipse.cdt.managedbuilder.makegen.gnu
«Derive»
«Import»
«Import, Call»
«interface»
IBuildObject
org.eclipse.cdt.managedbuilder.cor
«interface»
ITool
org.eclipse.cdt.managedbuilder.cor
«interface»
IManagedDependencyInfo
org.eclipse.cdt.managedbuilder.makege
«Instantiate»
«Import»
«Instantiate»
LlvmDependencyCalculator2
getDependencySourceInfo(in source: IPath, in resource: IResource, in buildContext: IBuildObject, in tool: ITool, in topBuildDirectory
getDependencySourceInfo(in source: IPath, in buildContext: IBuildObject, in tool: ITool, in topBuildDirectory: IPath)
DefaultGCCDependencyCalculator2
org.eclipse.cdt.managedbuilder.makegen.gnu
«Derive»

Figure 14. makegen.llvm package dependencies

4.3.6 Environment variable supplier: discovering LLVM, MinGW, Cygwin and C++ Standard Library paths

LLVM environment variable supplier’s function is to provide and add LLVM specific environment variable paths defined in LLVM preference page. Environment variables are also added and visible in Project -> Properties -> C/C++ Build -> Environment.

LLVM, MinGW, Cygwin and C++ Standard Library paths have to be set automatically whenever a user creates a new project with corresponding toolchain. Otherwise plug-in cannot find the files it is dependent on. User is only expected to add LLVM, LLVM-GCC, MinGW and Cygwin build paths to system environment path.

LLVM specific toolchains are only displayed in Eclipse’s new project wizard if the plug-in is able to find LLVM binaries. Class LlvmIsToolChainSupported is used to notify if LLVM binary path can be found from LLVM environment supplier by using getBinPath method inside LlvmEnvironmentVariableSupplier class. In buildDefinition extension under LLVM toolchain the class LlvmIsToolChainSupported is given as a parameter for isToolChainSupported option. Class LlvmEnvironmentVariableSupplier is given as a parameter for configurationEnvironmentSupplier option. These configurations make sure that LLVM toolchain is only available for the user if LLVM binaries are found and that LLVM specific environment variables are initialized when creating a new LLVM specific project.

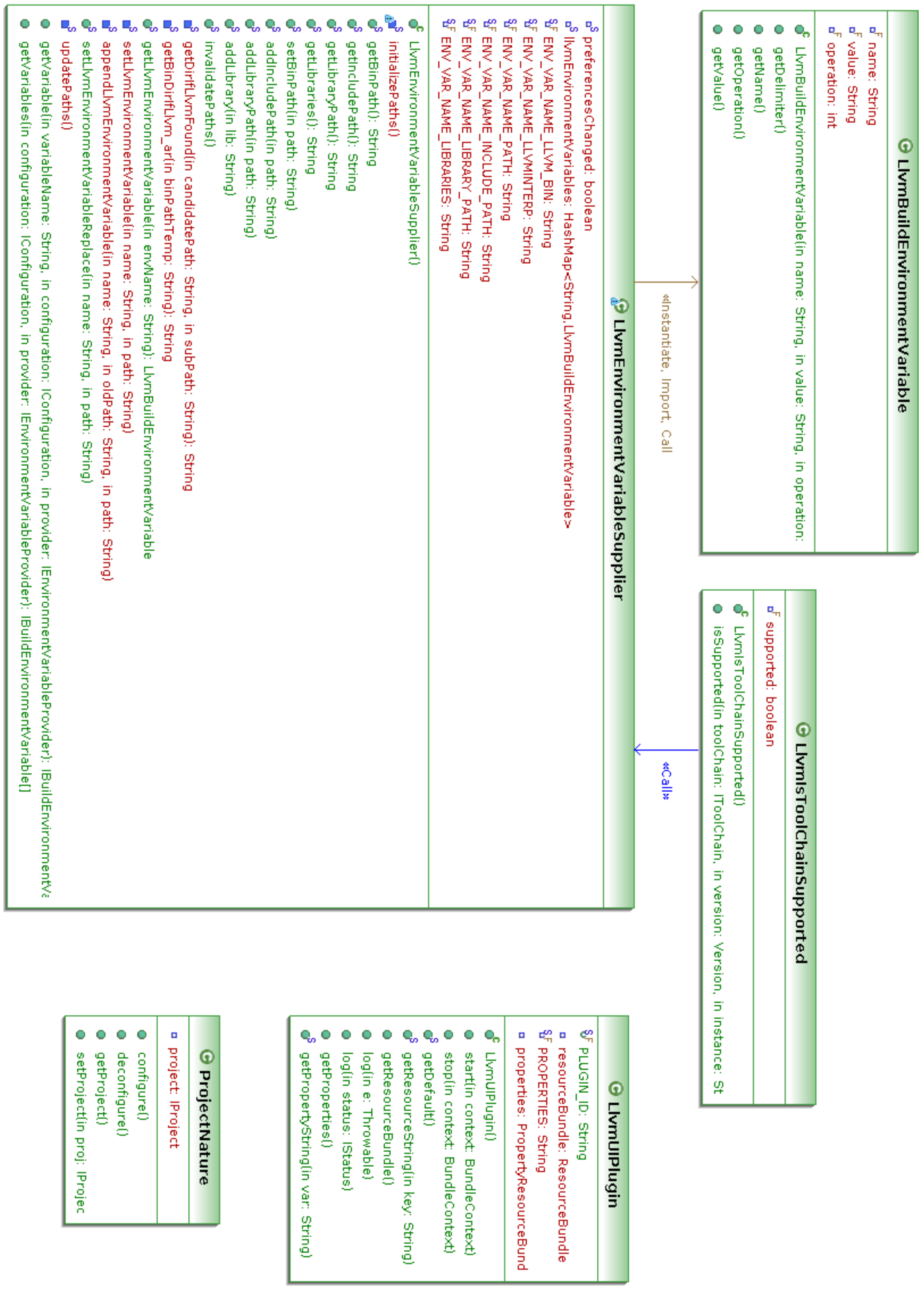
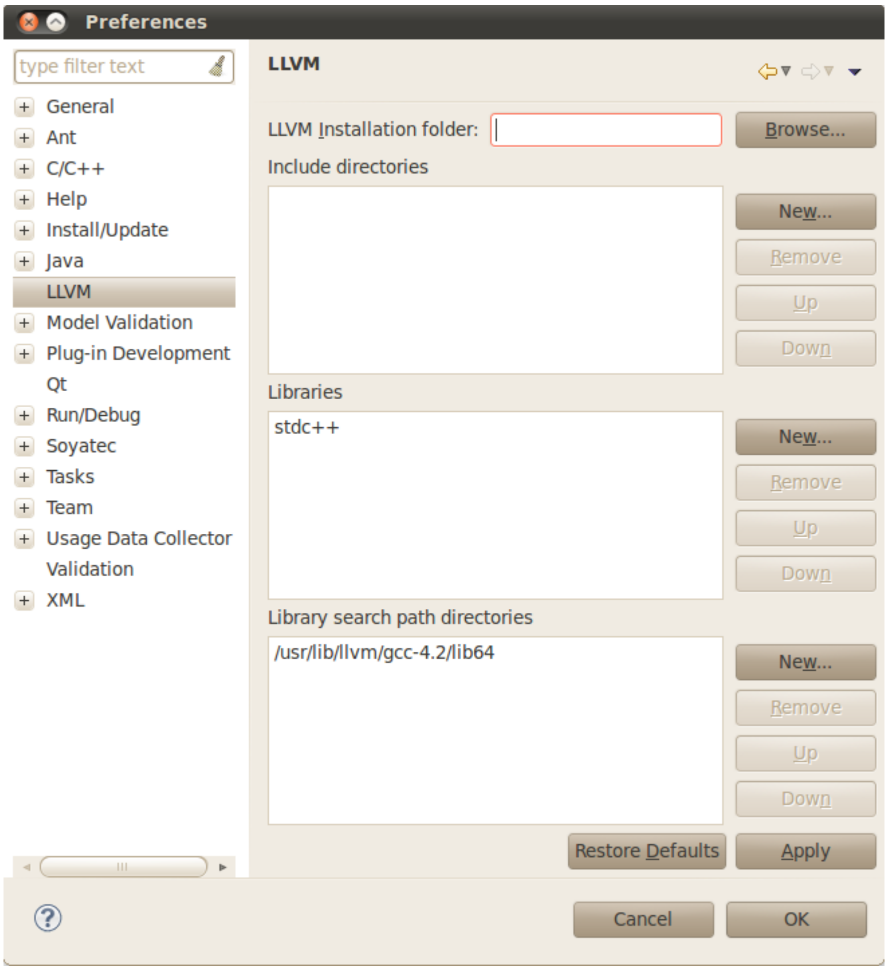

LlvmBuildEnvironmentVariable
name: String
value: String
operation: int
LlvmBuildEnvironmentVariable(in name: String, in value: String, in operation:
getDelimiter()
getName()
getOperation()
getValue()
«Instantiate, Import, Call
LlvmIsToolChainSupported
supported: boolean
LlvmIsToolChainSupported()
isSupported(in toolChain: IToolChain, in version: Version, in instance: St
«Call»
LlvmEnvironmentVariableSupplier
preferencesChanged: boolean
llvmEnvironmentVariables: HashMap<String,LlvmBuildEnvironmentVariable>
ENV_VAR_NAME_LLVM_BIN: String
ENV_VAR_NAME_LLVMINTERP: String
ENV_VAR_NAME_PATH: String
ENV_VAR_NAME_INCLUDE_PATH: String
ENV_VAR_NAME_LIBRARY_PATH: String
ENV_VAR_NAME_LIBRARIES: String
LlvmEnvironmentVariableSupplier()
initializePaths()
getBinPath(): String
getIncludePath(): String
getLibraryPath(): String
getLibraries(): String
setBinPath(in path: String)
addIncludePath(in path: String)
addLibraryPath(in path: String)
addLibrary(in lib: String)
invalidatePaths()
getDirIfLlvmFound(in candidatePath: String, in subPath: String): String
getBinDirIfLlvm_ar(in binPathTemp: String): String
getLlvmEnvironmentVariable(in envName: String): LlvmBuildEnvironmentVariable
setLlvmEnvironmentVariable(in name: String, in path: String)
appendLlvmEnvironmentVariable(in name: String, in oldPath: String, in path: String)
setLlvmEnvironmentVariableReplace(in name: String, in path: String)
updatePaths()
getVariable(in variableName: String, in configuration: IConfiguration, in provider: IEnvironmentVariableProvider): IBuildEnvironmentV
getVariables(in configuration: IConfiguration, in provider: IEnvironmentVariableProvider): IBuildEnvironmentVariable[]
LlvmUIPlugin
PLUGIN_ID: String
resourceBundle: ResourceBundle
PROPERTIES: String
properties: PropertyResourceBund
LlvmUIPlugin()
start(in context: BundleContext)
stop(in context: BundleContext)
getDefault()
getResourceString(in key: String)
getResourceBundle()
log(in e: Throwable)
log(in status: IStatus)
getProperties()
getPropertyString(in var: String)
ProjectNature
project: IProject
configure()
deconfigure()
getProject()
setProject(in proj: IProjec

Figure 15. llvm package dependencies

4.3.7 Creating preference page for LLVM specific settings



LLVM Installation folder:

Include directories

Libraries

stdc++

Library search path directories

/usr/lib/llvm/gcc-4.2/lib64

Figure 16. LLVM Preferences page

LLVM preference page gives users an option to point location to LLVM installation directory and add libraries, include paths and library search paths e.g. for C++ Standard Library.

The starting point programmatically for the preference page was to create a parent class (LlvmListEditor) for all lists. LlvmListEditor contains methods that create buttons and selection listeners and actions for them. Then three classes for different lists were implemented by extending LlvmListEditor (IncludePathListEditor, LibraryListEditor and LibraryPathListEditor). Different classes for lists were implemented, because they differ from the way what type of value is added (file or path) and how they are added or removed from the preference store. LlvmPreferenceStore is like a memory bank where all preferences are recorded. Preferences can be added, listed and removed. PreferenceInitializer initializes the values from the preference store to the workspace. PreferenceConstants features String definitions that cannot be changed. LlvmPreferencePage implements the actual page which adds fields for LLVM installation directory, include paths, libraries and library paths.

The LLVM preference page is added to Eclipse's Window –> Preferences dialog by extending org.eclipse.ui.preferencePages. Also org.eclipse.core.runtime.preferences shall be extended and PreferenceInitializer defined as the initializer for the preference page.

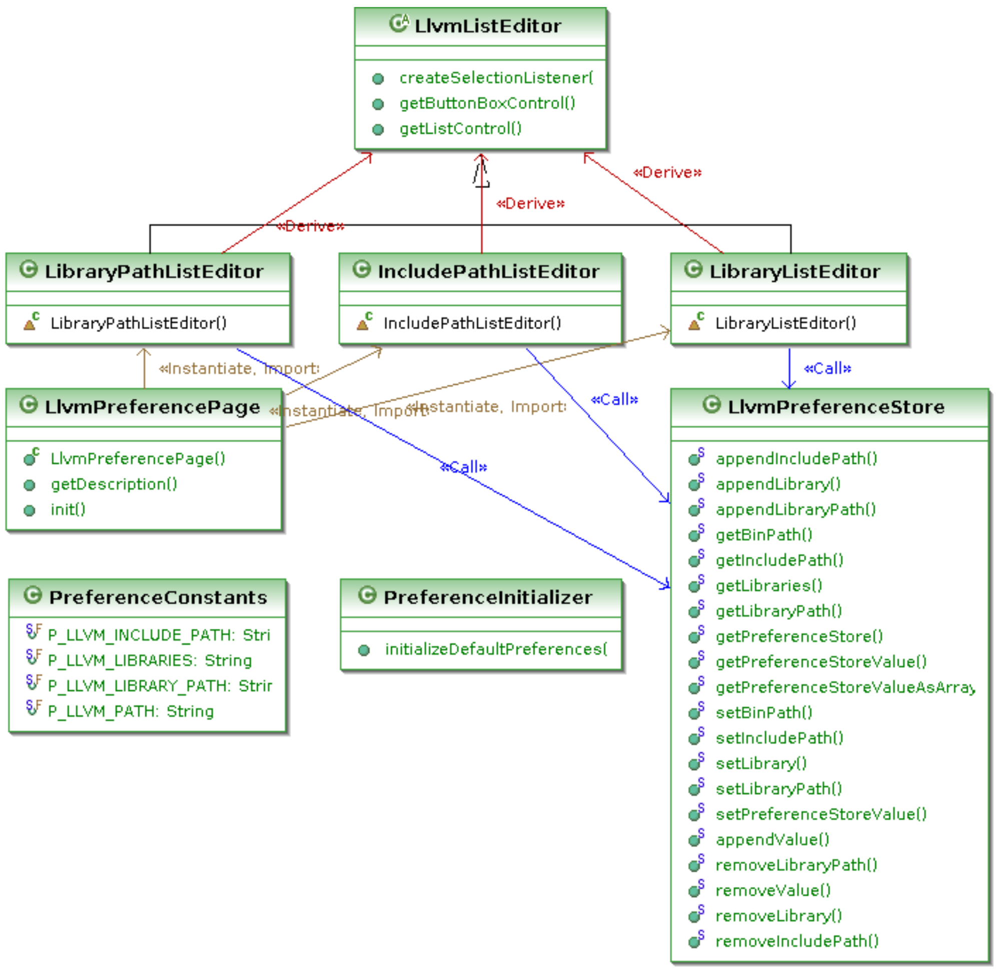


Figure 17. preferences package dependencies and inheritances

4.3.8 Creating and configuring extension for help files

Help extension is useful way to provide informative documents that end-users can read. Documents itself are written in html format. In order to access documents from Eclipse’s Help -> Help Contents, we must extend org.eclipse.help.toc and specify the path for table of contents xml file or index file. In this project the table of contents xml file was created which is used as a starting point to access other documents.

4.3.9 Adding include paths, libraries and library paths to Tool’s options

In order to show files and paths in paths and symbols dialog (Project -> Properties -> C/C++ General -> Paths and Symbols) that are added to LLVM preference page it is not enough to only add them to the LLVM preference store. The added files and paths might work even though they would not be visible in paths and symbols dialog. However this might be really confusing for the end-user, because paths and symbols dialog is the standard place to check what dependencies are added.

Adding include paths, library files and library search paths to paths and symbols dialog is not as straightforward as adding them to the LLVM preference store. The reason is that everything set in LLVM preference page is aimed to be workspace-wide such as all Eclipse Window -> Preferences configurations are. First we should get all projects from the workspace. Then all build configurations from every project. The file or path must be added independently to every build configuration that is done by looping through every project’s every build configuration. We must get the right option from the right tool and append the new value on top of the existing values. For that we need to know the option id by giving tool and option value type as enumeration to getOptionId() method’s parameters. Next we set the tool’s option by calling ManagedBuildManager.setOption(IConfiguration config, IHoldsOptions holder, IOption option, String[] value) method and saving the build configuration by calling ManagedBuildManager.saveBuildInfo(final IProject project, final boolean force) method. We must also be able to remove the added entries which is done by removing the specific entry in question from the existing list and setting a new option list similarly as we would add a new entry. LlvmToolOptionPathUtil class contains all necessary algorithms to make this all work.

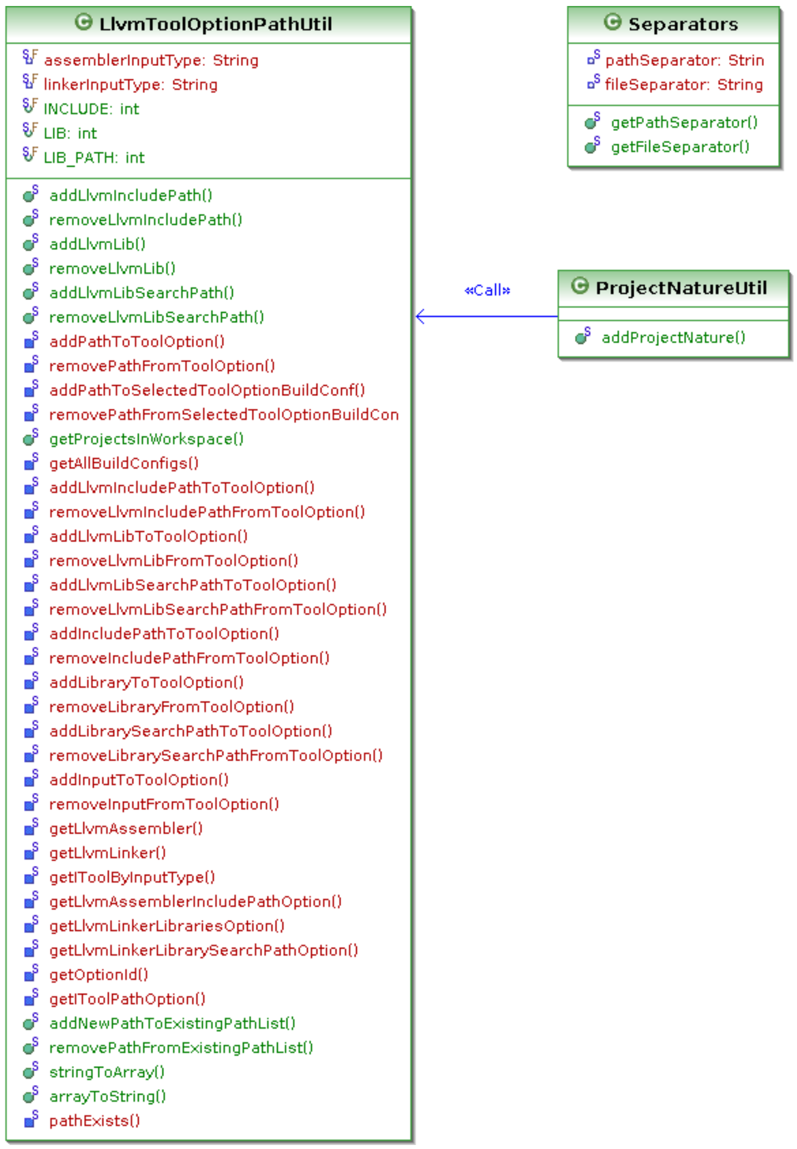
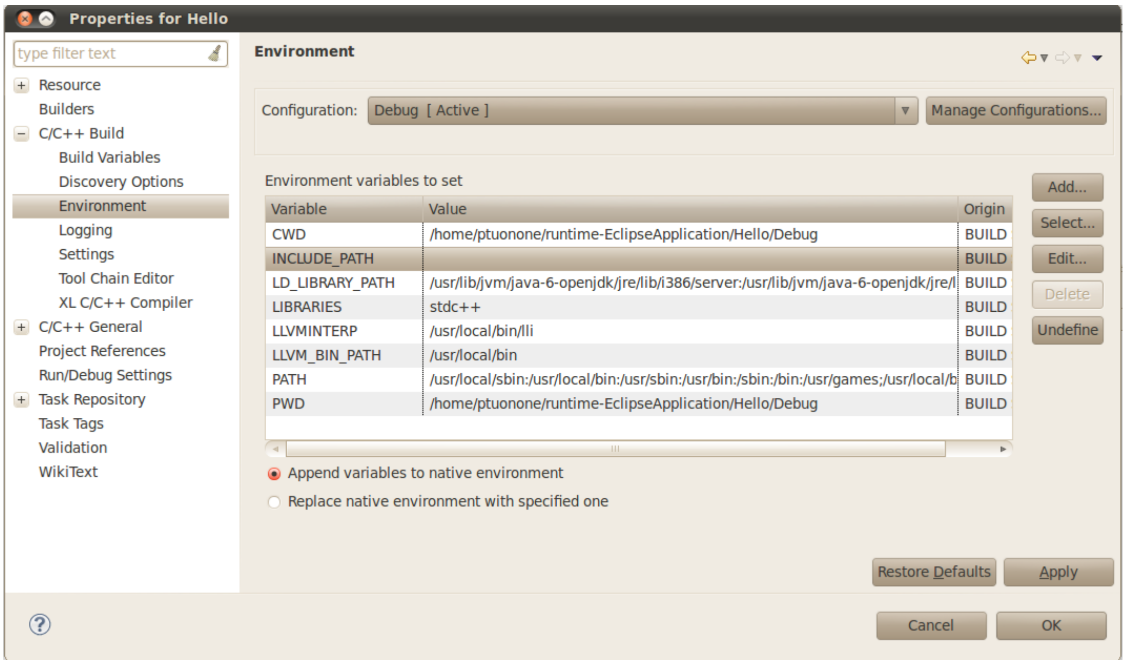

LlvmToolOptionPathUtil
assemblerInputType: String
linkerInputType: String
INCLUDE: int
LIB: int
LIB_PATH: int
addLlvmIncludePath()
removeLlvmIncludePath()
addLlvmLib()
removeLlvmLib()
addLlvmLibSearchPath()
removeLlvmLibSearchPath()
addPathToToolOption()
removePathFromToolOption()
addPathToSelectedToolOptionBuildConf()
removePathFromSelectedToolOptionBuildCon
getProjectsInWorkspace()
getAllBuildConfigs()
addLlvmIncludePathToToolOption()
removeLlvmIncludePathFromToolOption()
addLlvmLibToToolOption()
removeLlvmLibFromToolOption()
addLlvmLibSearchPathToToolOption()
removeLlvmLibSearchPathFromToolOption()
addIncludePathToToolOption()
removeIncludePathFromToolOption()
addLibraryToToolOption()
removeLibraryFromToolOption()
addLibrarySearchPathToToolOption()
removeLibrarySearchPathFromToolOption()
addInputToToolOption()
removeInputFromToolOption()
getLlvmAssembler()
getLlvmLinker()
getIToolByInputType()
getLlvmAssemblerIncludePathOption()
getLlvmLinkerLibrariesOption()
getLlvmLinkerLibrarySearchPathOption()
getOptionId()
getIToolPathOption()
addNewPathToExistingPathList()
removePathFromExistingPathList()
stringToArray()
arrayToString()
pathExists()
Separators
pathSeparator: Strin
fileSeparator: String
getPathSeparator()
getFileSeparator()
«Call»
ProjectNatureUtil
addProjectNature()

Figure 18. llvm.util package dependencies

4.4 Testing

4.4.1 Testing LLVM environment supplier



Environment

Configuration: Debug [Active] Manage Configurations...

Environment variables to set

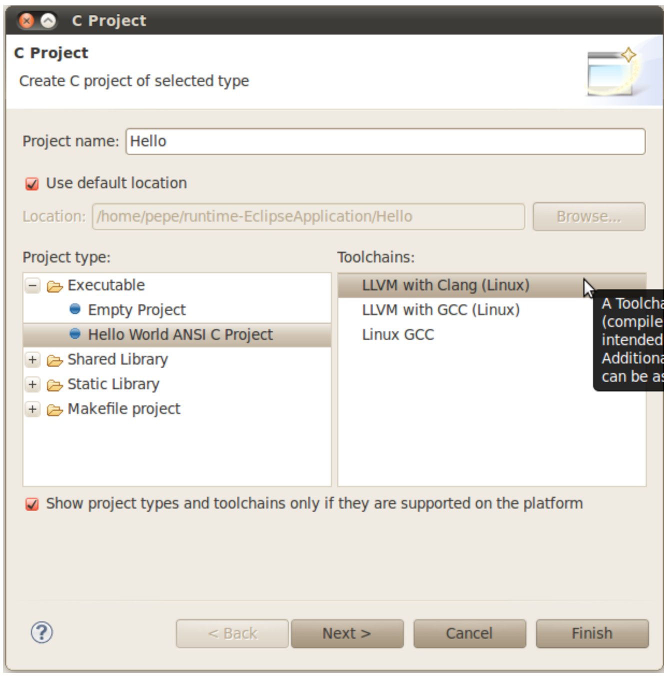
Variable	Value	Origin
CWD	/home/ptuonone/runtime-EclipseApplication/Hello/Debug	BUILD
INCLUDE_PATH		BUILD
LD_LIBRARY_PATH	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/server:/usr/lib/jvm/java-6-openjdk/jre/l	BUILD
LIBRARIES	stdc++	BUILD
LLVMINTERP	/usr/local/bin/lli	BUILD
LLVM_BIN_PATH	/usr/local/bin	BUILD
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/b	BUILD
PWD	/home/ptuonone/runtime-EclipseApplication/Hello/Debug	BUILD

[✓ Append variables to native environment]
Replace native environment with specified one

Figure 19. Build environment variables

Notice how INCLUDE_PATH, LD_LIBRARY_PATH, LIBRARIES, LLVMINTERP, LLVM_BIN_PATH are new C/C++ build environment variables. These variables have values if values are detected automatically via LlvmEnvironmentVariableSupplier class or added via OS to these variables or added via LLVM preference page.

4.4.2 Creating a new project



C Project

C Project

Create C project of selected type

Project name: Hello

☑ Use default location

Location: /home/pepe/runtime-EclipseApplication/Hello

Project type:
- Executable
  - Empty Project
  - Hello World ANSI C Project
- Shared Library
- Static Library
- Makefile project

Toolchains:
- LLVM with Clang (Linux)
- LLVM with GCC (Linux)
- Linux GCC

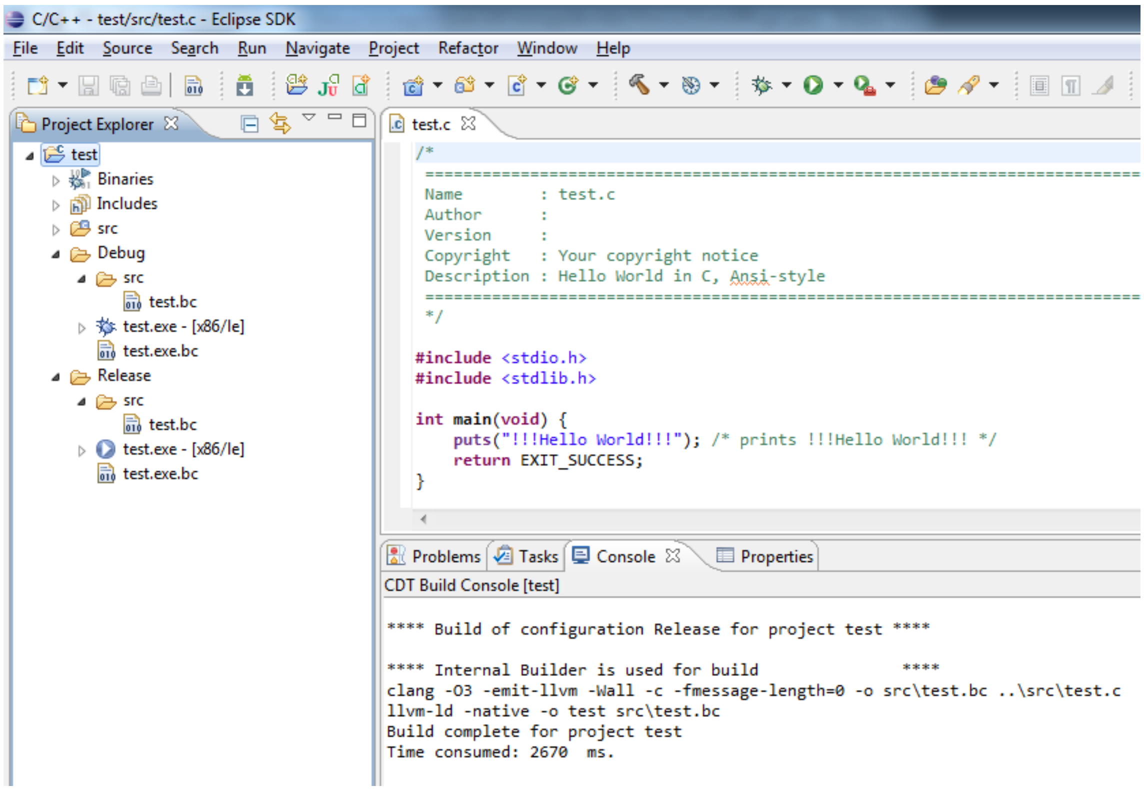
A Toolcha
(compile
intended
Additiona
can be as

☑ Show project types and toolchains only if they are supported on the platform

Figure 20. C Project creation in Linux

Notice that toolchains featuring LLVM exist in toolchains window. This is true only if LLVM binaries are found (from PATH or automatically via LlvmEnvironmentVariableSupplier class).

4.4.3 Building LLVM toolchain projects

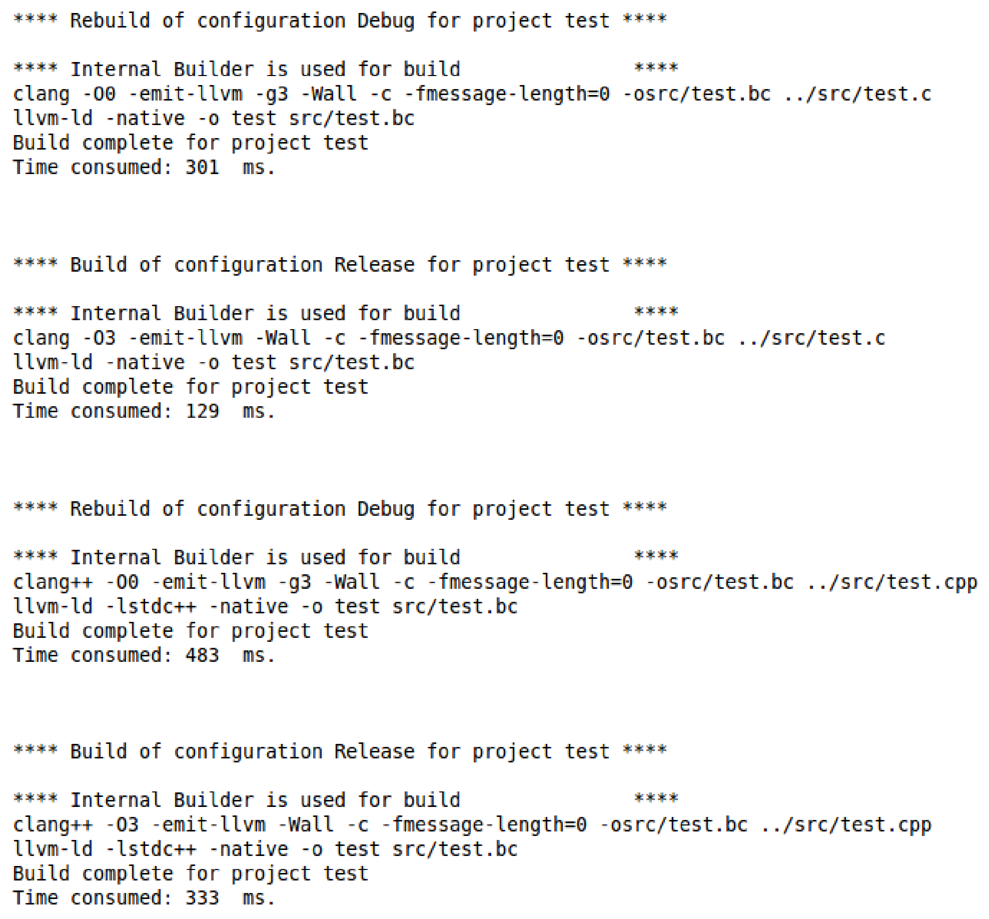


```
/*
 ============================================================================
 Name        : test.c
 Author      :
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
    return EXIT_SUCCESS;
}
```

```
**** Build of configuration Release for project test ****

**** Internal Builder is used for build               ****
clang -O3 -emit-llvm -Wall -c -fmessage-length=0 -o src\test.bc ..\src\test.c
llvm-ld -native -o test src\test.bc
Build complete for project test
Time consumed: 2670  ms.
```

Figure 21. IDE view of C project release build with Clang on Windows

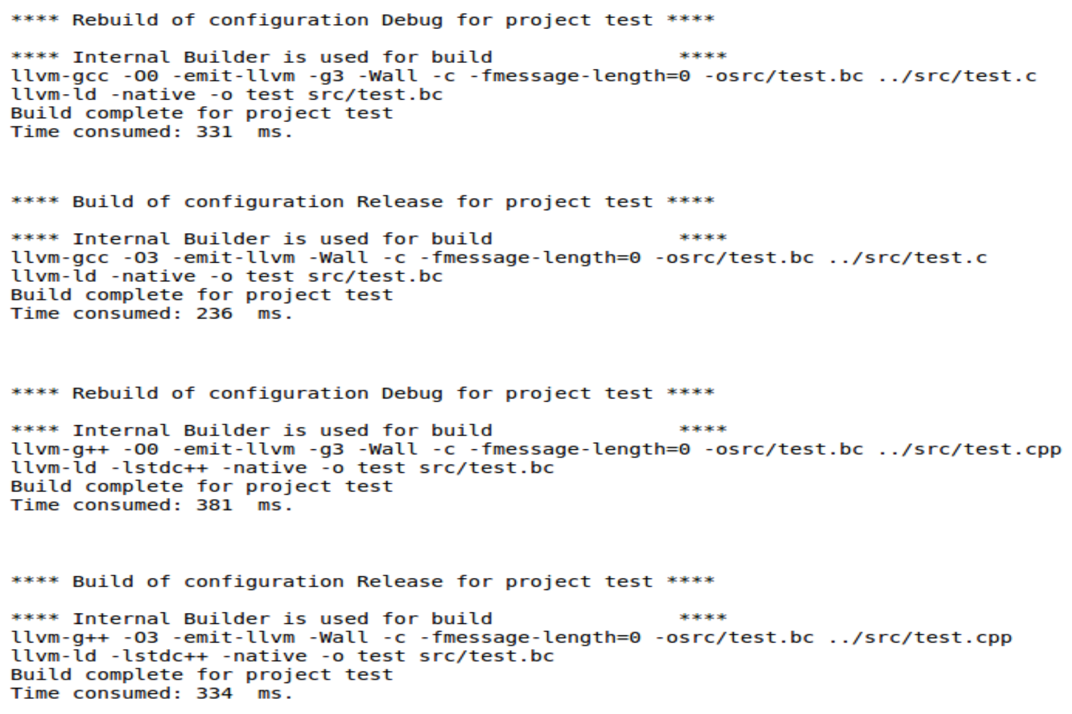
Notice that src folder includes .bc file which is created by Clang compiler.



```
**** Rebuild of configuration Debug for project test ****

**** Internal Builder is used for build               ****
clang -O0 -emit-llvm -g3 -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.c
llvm-ld -native -o test src/test.bc
Build complete for project test
Time consumed: 301  ms.


**** Build of configuration Release for project test ****

**** Internal Builder is used for build               ****
clang -O3 -emit-llvm -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.c
llvm-ld -native -o test src/test.bc
Build complete for project test
Time consumed: 129  ms.


**** Rebuild of configuration Debug for project test ****

**** Internal Builder is used for build               ****
clang++ -O0 -emit-llvm -g3 -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.cpp
llvm-ld -lstdc++ -native -o test src/test.bc
Build complete for project test
Time consumed: 483  ms.


**** Build of configuration Release for project test ****

**** Internal Builder is used for build               ****
clang++ -O3 -emit-llvm -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.cpp
llvm-ld -lstdc++ -native -o test src/test.bc
Build complete for project test
Time consumed: 333  ms.
```

Figure 22. C and C++ debug and release builds with Clang

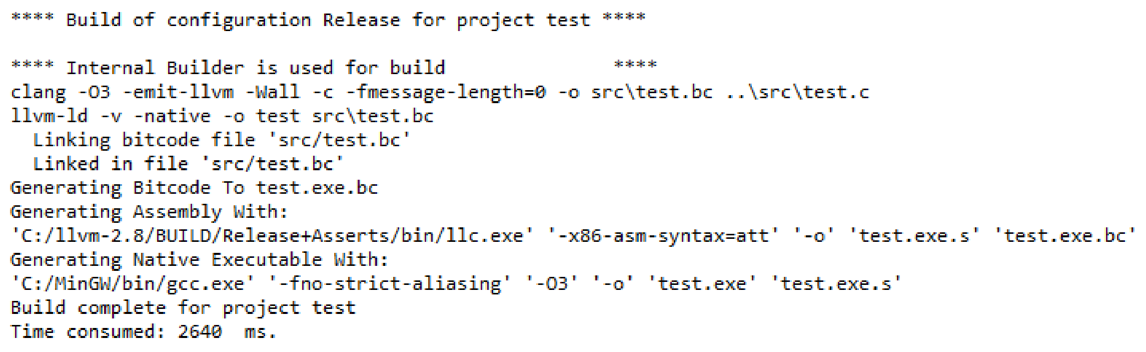
These are Eclipse console views of Clang builds with C/C++ and debug/release configurations.



```
**** Rebuild of configuration Debug for project test ****

**** Internal Builder is used for build               ****
llvm-gcc -O0 -emit-llvm -g3 -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.c
llvm-ld -native -o test src/test.bc
Build complete for project test
Time consumed: 331  ms.


**** Build of configuration Release for project test ****

**** Internal Builder is used for build               ****
llvm-gcc -O3 -emit-llvm -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.c
llvm-ld -native -o test src/test.bc
Build complete for project test
Time consumed: 236  ms.


**** Rebuild of configuration Debug for project test ****

**** Internal Builder is used for build               ****
llvm-g++ -O0 -emit-llvm -g3 -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.cpp
llvm-ld -lstdc++ -native -o test src/test.bc
Build complete for project test
Time consumed: 381  ms.


**** Build of configuration Release for project test ****

**** Internal Builder is used for build               ****
llvm-g++ -O3 -emit-llvm -Wall -c -fmessage-length=0 -osrc/test.bc ../src/test.cpp
llvm-ld -lstdc++ -native -o test src/test.bc
Build complete for project test
Time consumed: 334  ms.
```

Figure 23. C and C++ debug and release builds LLVM-GCC

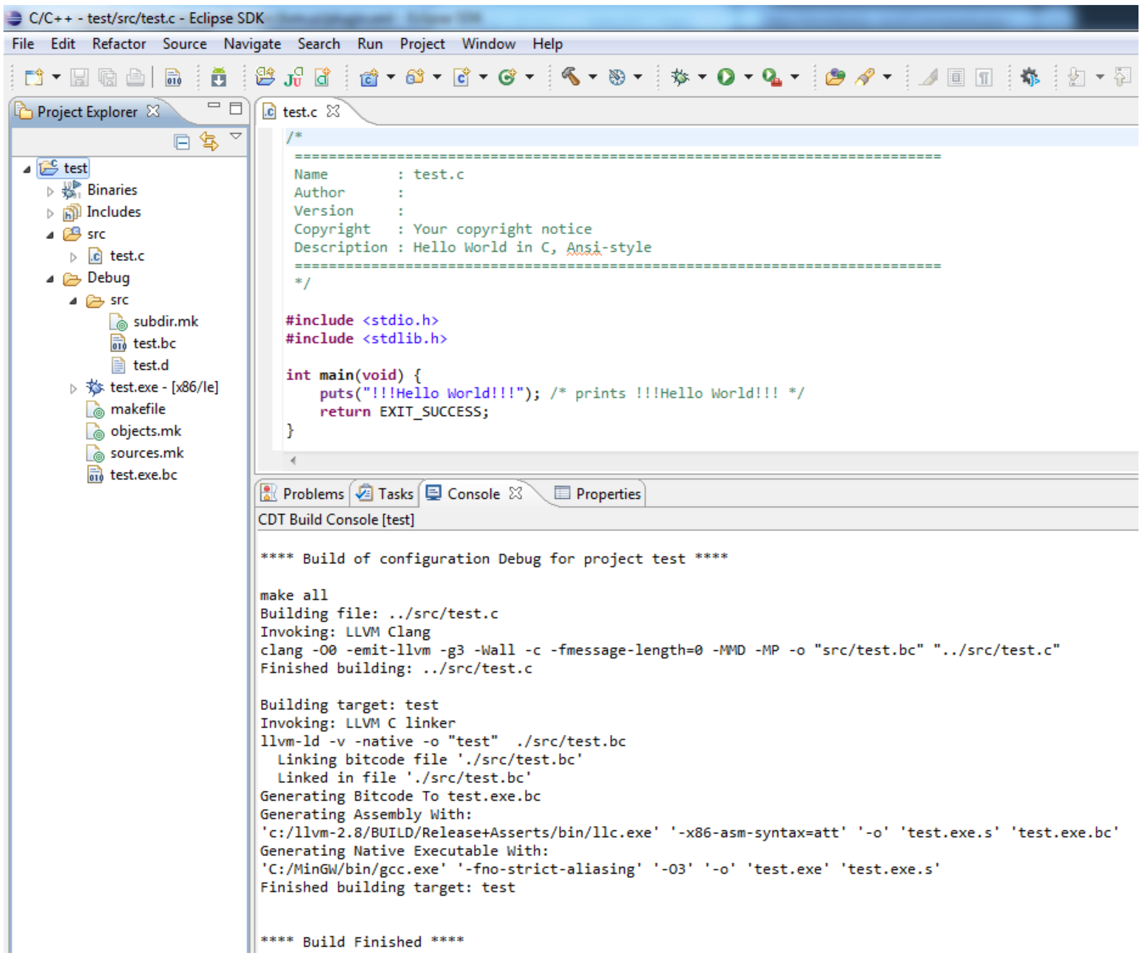
These are Eclipse console views of LLVM-GCC builds with C/C++ and debug/release configurations.



```
**** Build of configuration Release for project test ****

**** Internal Builder is used for build               ****
clang -O3 -emit-llvm -Wall -c -fmessage-length=0 -o src\test.bc ..\src\test.c
llvm-ld -v -native -o test src\test.bc
  Linking bitcode file 'src/test.bc'
  Linked in file 'src/test.bc'
Generating Bitcode To test.exe.bc
Generating Assembly With:
'C:/llvm-2.8/BUILD/Release+Asserts/bin/llc.exe' '-x86-asm-syntax=att' '-o' 'test.exe.s' 'test.exe.bc'
Generating Native Executable With:
'C:/MinGW/bin/gcc.exe' '-fno-strict-aliasing' '-O3' '-o' 'test.exe' 'test.exe.s'
Build complete for project test
Time consumed: 2640  ms.
```

Figure 24. Clang release build with linker’s verbose mode enabled on Windows

With verbose mode option checked the user gets more descriptive console output.



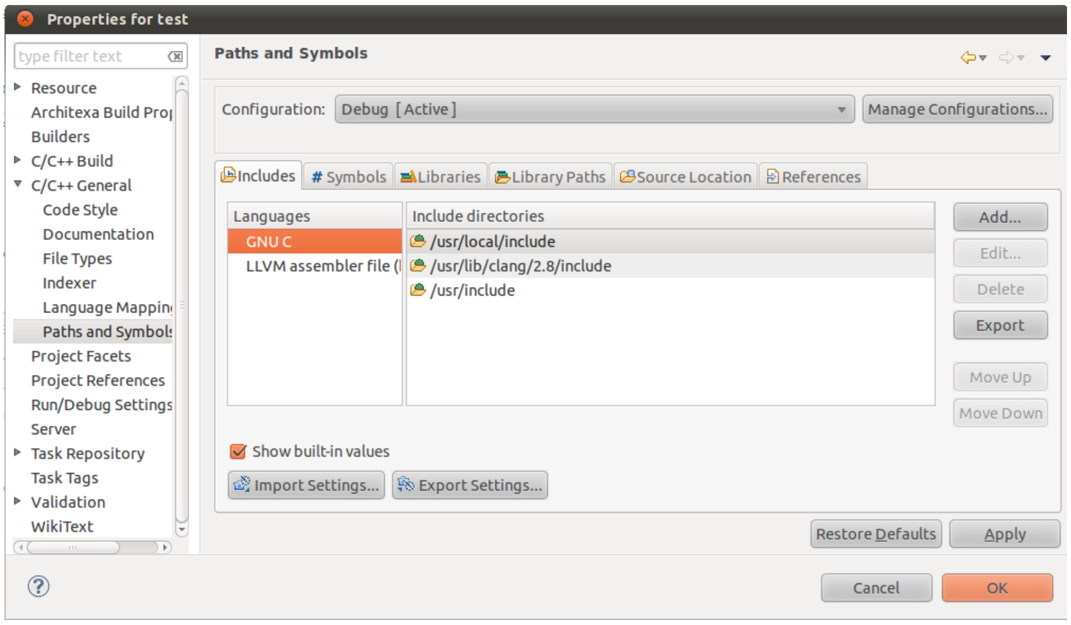
```
/*
 ============================================================================
 Name        : test.c
 Author      :
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
	puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
	return EXIT_SUCCESS;
}
```

```
**** Build of configuration Debug for project test ****

make all
Building file: ../src/test.c
Invoking: LLVM Clang
clang -O0 -emit-llvm -g3 -Wall -c -fmessage-length=0 -MMD -MP -o "src/test.bc" "../src/test.c"
Finished building: ../src/test.c

Building target: test
Invoking: LLVM C linker
llvm-ld -v -native -o "test"  ./src/test.bc
  Linking bitcode file './src/test.bc'
  Linked in file './src/test.bc'
Generating Bitcode To test.exe.bc
Generating Assembly With:
'c:/llvm-2.8/BUILD/Release+Asserts/bin/llc.exe' '-x86-asm-syntax=att' '-o' 'test.exe.s' 'test.exe.bc'
Generating Native Executable With:
'C:/MinGW/bin/gcc.exe' '-fno-strict-aliasing' '-O3' '-o' 'test.exe' 'test.exe.s'
Finished building target: test


**** Build Finished ****
```

Figure 25. Clang release build with Make builder on Windows

LLVM plug-in supports CDT internal builder and Make builder as seen in the above figure.

4.4.4 Testing scanner configuration discovery profiles



Configuration: Debug [Active]

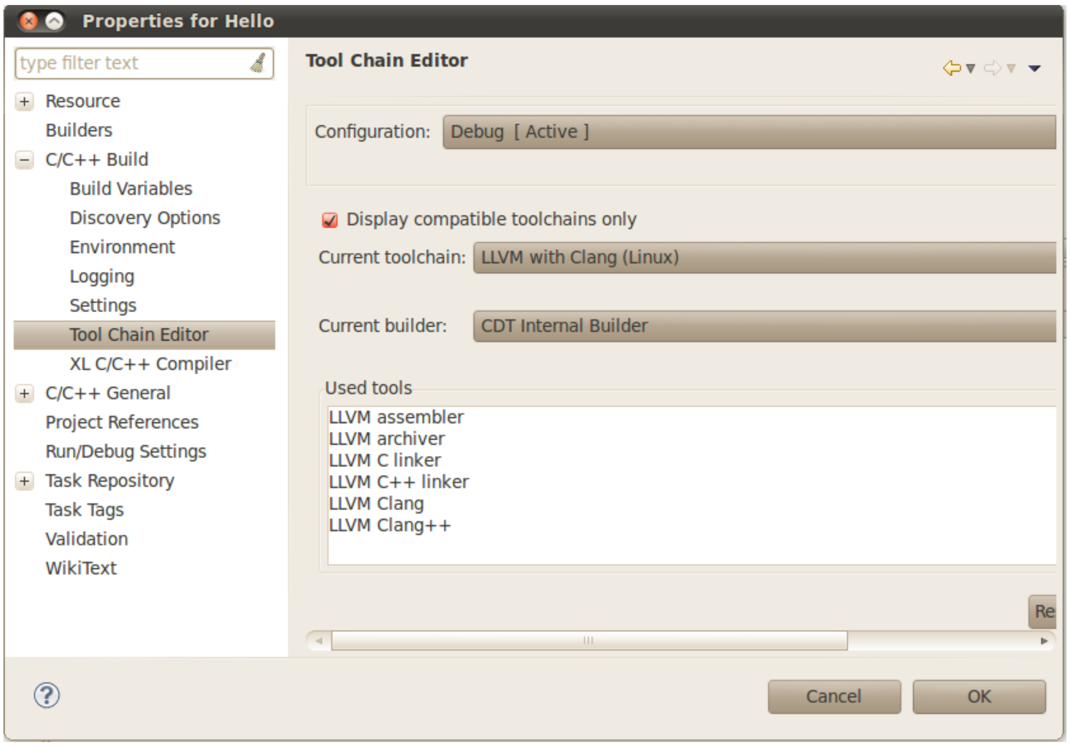
Languages	Include directories
GNU C	/usr/local/include
LLVM assembler file (I	/usr/lib/clang/2.8/include
	/usr/include

✓ Show built-in values

Figure 26. Paths and symbols for project’s build configuration

If scanner configuration discovery profiles are set properly and scanner info collector and scanner info console parser are programmatically correct then paths and symbols dialog should show all LLVM/Cygwin/MinGW dependent paths. The figure above shows C project includes.

4.4.5 Testing toolchain configuration



Configuration: Debug [Active]

☑ Display compatible toolchains only

Current toolchain: LLVM with Clang (Linux)

Current builder: CDT Internal Builder

Used tools
LLVM assembler
LLVM archiver
LLVM C linker
LLVM C++ linker
LLVM Clang
LLVM Clang++

Figure 27. Toolchain editor for project’s build configuration

Shows available tools for the current toolchain and provides an option to add/remove and replace tools. The important aspect here is to notice that all necessary tools for the current toolchain are included.

4.5 Test cases

JUnit project was created to test several functionalities of the plug-in.

The first test finds out the current operating system in use and creates projects compatible with that OS and compiles the projects.

The second test adds and removes values from LLVM preference page.

5 VALIDATION

5.1 User feedback

Eclipse Marketplace provides user reviews and metrics of e.g. how many clicks the plug-in page has received. This feature is helpful and allows developers to follow how much popularity their plug-in has gained. The marketplace also informs the amount of unsuccessful installs but does not seem to contain information about all installations. The marketplace contains a bit over 1000 plug-ins at the moment and LLVM plug-in has received approximately 50 clicks daily (even 85 on best days) and zero unsuccessful installations so far. I have not received any user feedback from Eclipse marketplace yet, but one user has marked the plug-in as his/her favorite.

5.2 CDT developers feedback

From the beginning the CDT developers have been encouraging to develop the plug-in and interested in integration proposal. Many of them have promised to support the development. The plug-in has gained approval from the CDT project lead and one CDT committer has already submitted a patch for the plug-in.

6 DISSEMINATION

6.1 Contacting Chief Architect of the LLVM Compiler Infrastructure

I decided to contact the primary author of LLVM in order to get llvm4eclipsecdt plug-in as part of the http://llvm.org/ProjectsWithLLVM/ web site that lists LLVM related projects. By listing the project on LLVM site, the project would get more attention and at the same time the project is introduced to the main author of LLVM. The original plan was to gain publicity before the plug-in finds its way into official CDT release. However after a month or so the plug-in is not yet listed on the site and it seems that it never will.

6.2 Contacting Eclipse CDT lead developer

Contacting the lead developer of CDT is understandable, because the plug-in only works if CDT is installed to Eclipse IDE. Plug-in would be a great addition to CDT and make the LLVM toolchain immediately available for Eclipse users (if their system meets the plug-in requirements). The lead developer may help by providing development help towards the process of donating the project to Eclipse Foundation. He suggested posting to CDT developers' mailing list to gain everybody's approval and interest and keep on working from that. He also told that the proper way to include this kind of project into CDT would be to create an Eclipse Bugzilla enhancement entry under CDT tools project.

6.3 Posting to cdt-dev mailing list

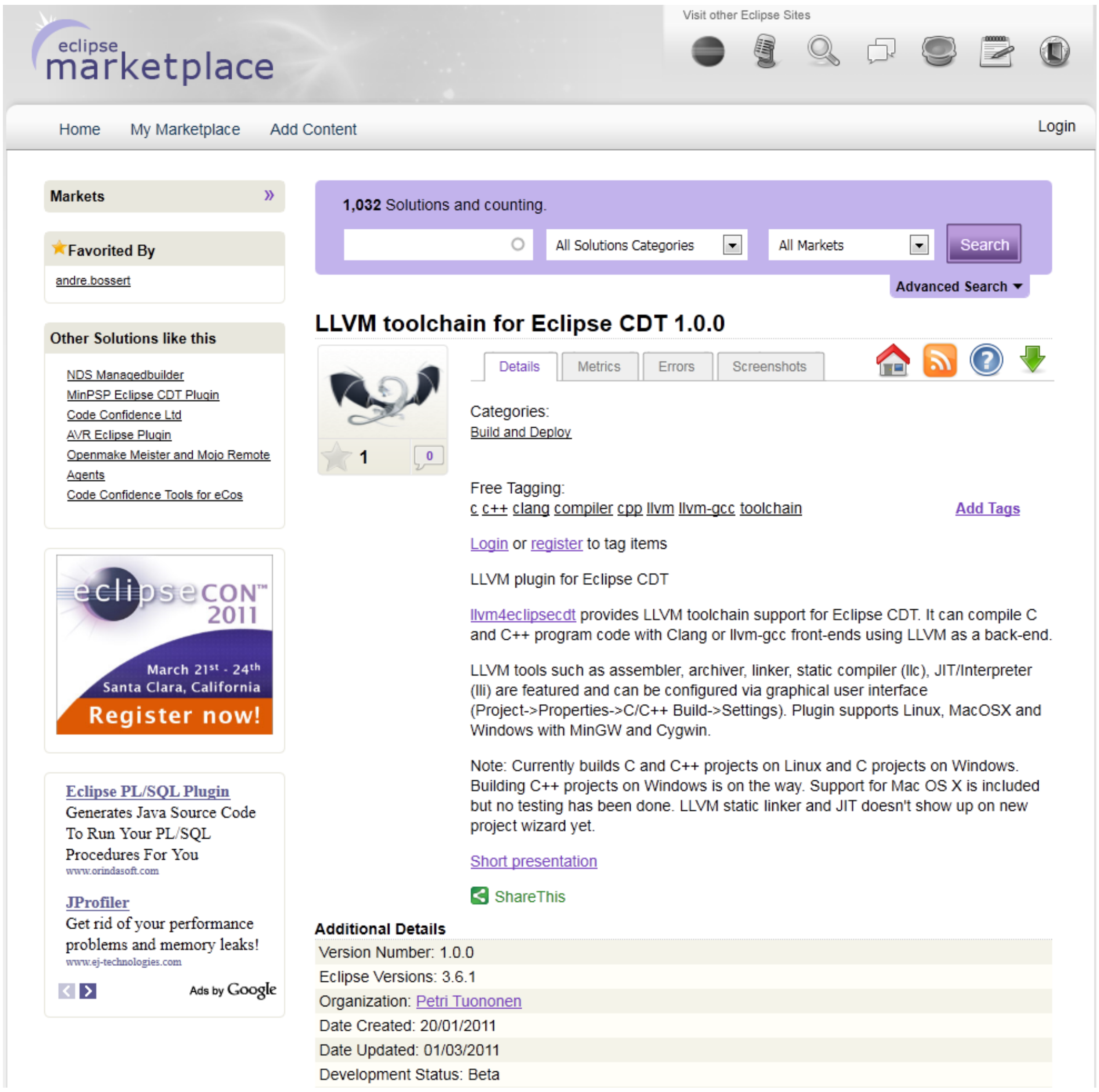
Cdt-dev is Eclipse CDT developers' mailing list and thus posting to that forum brings the most attention amongst CDT developers. With my post I made clear what I have been working on and the plan would be to integrate it to CDT and make it an official sub-project. For this I need the help of the whole CDT community and their approval. I promised to provide as much documentation as I can and if needed to participate on conference calls held at Ottawa. The teleconferences are only held once a month and only last about an hour.

The first message posted on cdt-dev mailing list (http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg21163.html):

> I feel like I have a great proposition to make. I have been working on LLVM plugin for Eclipse CDT for some time now. This allows compilation of C/C ++ code with Clang or llvm-gcc front-ends using LLVM as a backend. LLVM static compiler and JIT practically also implemented. Linux and Windows platforms are targeted, but would work on Mac OS X too. This is the first ever LLVM plugin targeted for Eclipse and development is in quite good shape regardless of the man power. The plugin uses EPL open-source license and would be highly appreciated if this project would get part of the 'official' CDT sub-projects in order to ensure its further development.
>
> The new LLVM toolchain would be just perfect addition to CDT don't you think?
>
> I would like to be part of the following teleconferences etc. what is necessary and introduce it further. Also documentation is widely available on request. I'm also willing to help to work with this project further and would appreciate a committer status as I have been fixing multiple amounts of CDT bugs lately. This committer status could be targeted only to LLVM-CDT integration plug-in, because a proper web hosting for update site would be appropriate.
>
> Currently this is a plugin project hosted on Google code
> http://code.google.com/p/llvm4eclipsecdt
>
> SVN repository: https://llvm4eclipsecdt.googlecode.com/svn/
>
> The plan would be to make this project as an integral part of CDT and for that I need all the help the CDT community can provide.
>
> I hope you all fellow developers take this as an option and think of the amount of users it would affect and all the limitless options it may bring. Let's make CDT even better!
>
> Best Regards,
> Petri Tuononen

6.4 Publishing plug-in to Eclipse marketplace

Eclipse marketplace is an easy to use method to search and download Eclipse plug-ins directly from Eclipse IDE. LLVM plug-in for Eclipse CDT entry contains a short description, info about author, license, version etc. By publishing LLVM plug-in to Eclipse marketplace provides exposure to millions of Eclipse users worldwide.



LLVM toolchain for Eclipse CDT 1.0.0

Categories:
Build and Deploy

Free Tagging:
c c++ clang compiler cpp llvm llvm-gcc toolchain Add Tags

Login or register to tag items

LLVM plugin for Eclipse CDT

llvm4eclipsecdt provides LLVM toolchain support for Eclipse CDT. It can compile C and C++ program code with Clang or llvm-gcc front-ends using LLVM as a back-end.

LLVM tools such as assembler, archiver, linker, static compiler (llc), JIT/Interpreter (lli) are featured and can be configured via graphical user interface (Project->Properties->C/C++ Build->Settings). Plugin supports Linux, MacOSX and Windows with MinGW and Cygwin.

Note: Currently builds C and C++ projects on Linux and C projects on Windows. Building C++ projects on Windows is on the way. Support for Mac OS X is included but no testing has been done. LLVM static linker and JIT doesn't show up on new project wizard yet.

Short presentation

Additional Details

Version Number: 1.0.0
Eclipse Versions: 3.6.1
Organization: Petri Tuonenen
Date Created: 20/01/2011
Date Updated: 01/03/2011
Development Status: Beta

Figure 28. LLVM plug-in on the Eclipse marketplace

6.5 Becoming Eclipse CDT committer

Along the project matures it will be more and more managed by Eclipse Foundation. I try to maintain a steady developer status on the project and therefore I try to get a committer access to at least parts concerning LLVM. Committers can make immediate revision updates in contrast to patch submitters who need to wait until their patch is approved and committed. I would be able to review new patches and commit them thus speeding up the LLVM plug-in development. The only downside is that individual can not just apply committer status. One must be nominated by a current CDT committer and get enough votes from other CDT committers. The voting must also be approved by Eclipse PMC and project lead must fill in CVS package and employer information. Individuals who have made multiple or large contributions to CDT are able to be nominated as committers.

7 FUTURE PLANS

7.1 Integrating plug-in to CDT and contributing to Eclipse Foundation.

In order to contribute to the CDT project an official enhancement bug entry was filed. A project of this size needs to be reviewed by Eclipse lawyers and tested properly before publishing which leads to the fact that it was too late for the CDT 8.0 release. Eventually the plug-in will be part of the CDT as I participate in the migrating process in autumn.

7.2 Additional features

New upcoming tools that may bring interesting opportunities to known compiler technology may be added later on.

7.3 Activating open-source developers to contribute towards the project

New opportunities to promote LLVM plug-in are constantly looked for. Main mission is to get as much people to use LLVM toolchain as possible. This is especially good for testing in the initial phase to get wider audience's feedback how the plug-in suits their needs.

7.4 Keeping the project development continuous

It is my interest to co-operate and communicate with fellow developers if they show interest towards developing the LLVM plug-in further. I will make sure that the plug-in sees a smooth transition as a part of the CDT release.

8 SUMMARY

The LLVM plug-in was developed according to specifications and is currently released on the Eclipse Marketplace. While the plug-in was just a few weeks late from the CDT 8.0 release it will eventually be released in the CDT 9.0 release and before that in CDT HEAD. When LLVM plug-in will get added to the CDT it will be polished thus meeting user expectations. Overall the project was a great success and will surely be a long waited feature for the CDT.

REFERENCES

[1] Low Level Virtual Machine. Wikipedia article. Available from: http://en.wikipedia.org/wiki/Llvm [cited 1.2011]

[2] Introduction to the LLVM Compiler System. PDF document. Available from: http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf [cited 1.2011]

[3] The LLVM Compiler Framework and Infrastructure Tutorial. WWW document. Available from: http://llvm.org/pubs/2004-09-22-LCPCLLVMTutorial.html [cited 1.2011]

[4] The LLVM Compiler Framework and Infrastructure. PDF document. Available from: http://llvm.org/pubs/2004-09-22-LCPCLLVMTutorial.pdf [cited 1.2011]

[5] The LLVM Compiler Infrastructure. WWW document. Available from: http://llvm.org/Features.html [cited 1.2011]

[6] How the LLVM Compiler Infrastructure Works. WWW article. Available from: http://www.informit.com/articles/article.aspx?p=1215438&seqNum=2 [cited 1.2011]

[7] Vikram Adve. WWW document. Available from: http://cs.illinois.edu/people/faculty/vikram-adve [1.2011]

[8] LLVM Language Reference Manual. WWW document. Available from: http://llvm.org/docs/LangRef.html [1.2011]

[9] LLVM(Low Level Virtual Machine) - New Compiler Technology. WWW document. Available from: http://iluvcompiler.springnote.com/pages/5381257 [cited 2.2011]

[10] Clang. Wikipedia article. Available from: http://en.wikipedia.org/wiki/Clang [cited 1.2011]

[11] Clang - Features and Goals. WWW document. Available from: http://clang.llvm.org/features.html [cited 1.2011]

[12] Expressive Diagnostics (Clang). WWW document. Available from: http://clang.llvm.org/diagnostics.html [cited 1.2011]

[13] Clang Static Analyzer. WWW document. Available from: http://clang-analyzer.llvm.org [cited 1.2011]

[14] The Short History of GCC development. WWW article. Available from: http://www.softpanorama.org/People/Stallman/history_of_gcc_development.shtml [cited 1.2011]

[15] Getting Started with the LLVM System. WWW document. Available from: http://llvm.org/releases/1.1/docs/GettingStarted.html [cited 1.2011]

[16] LLVM linker. WWW document. Available from: http://llvm.org/cmds/llvm-ld.html [cited 1.2011]

[17] Introduction to the LLVM Compiler Infrastructure. PDF document. Available from: http://llvm.org/pubs/2006-04-25-GelatoLLVMIntro.pdf [cited 1.2011]

[18] LLVM static compiler. WWW document. Available from: http://llvm.org/cmds/llc.html [cited 1.2011]

[19] The LLVM Compiler Framework and Infrastructure. PDF document. Available from: http://www.cs.cmu.edu/afs/cs/academic/class/15745-s09/www/lectures/lect3-llvm.pdf [cited 1.2011]

[20] ELLCC - The Embedded LLVM Compiler Collection. WWW document. Available from: http://ellcc.org [cited 1.2011]

[21] The LLVM Compiler Infrastructure: LLVM Users. WWW document. Available from: http://llvm.org/Users.html [cited 2.2011]

[22] Members contributing to, or shipping products based on CDT. WWW document. Available from: http://www.eclipse.org/membership/showMembersWithTag.php?TagID=13 [cited 2.2011]

[23] Eclipse Platform. WWW document. Available from: http://www.eclipse.org/platform [cited 1.2011]

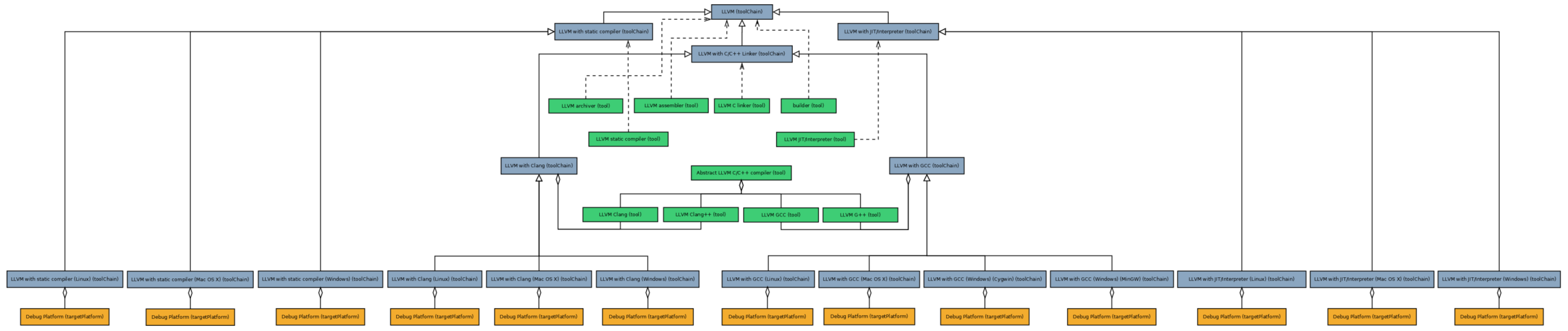
[24] Eclipse Platform Overview. WWW document. Available from: http://www.eclipse.org/platform/overview.php [cited 1.2011]

[25] CDT/User/FAQ: What is the CDT? Available from: http://wiki.eclipse.org/CDT/User/FAQ#What_is_the_CDT.3F [cited 1.2011]

[26] Developing Eclipse plug-ins. WWW article. Available from: http://www.ibm.com/developerworks/library/os-ecplug [2.2011]

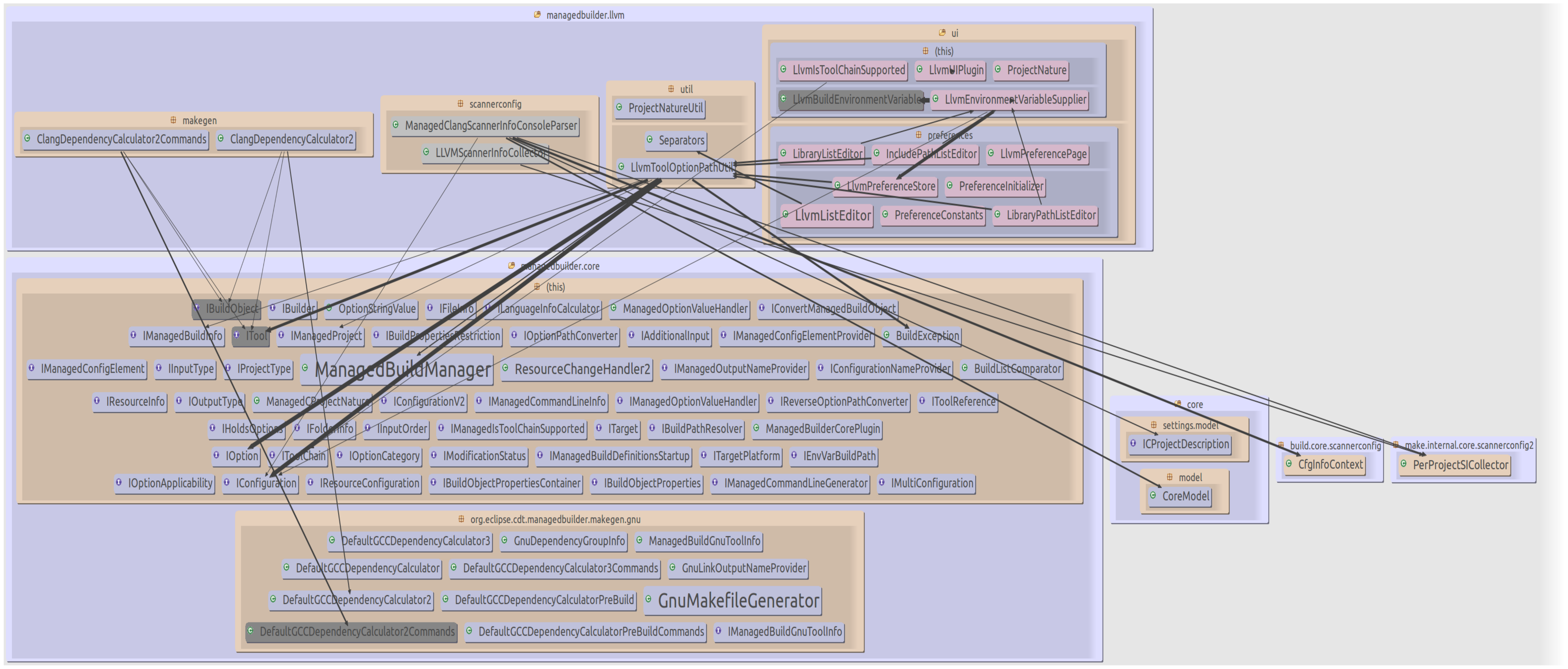
[27] Eclipse Platform Overview. WWW document. Available from: http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.cdt.doc.isv/guide/mbs/extensibilityGuide/Managed_Build_Extensibility.html [cited 2.2011]

Plugin.xml configuration schema


LLVM (toolChain)
LLVM with static compiler (toolChain)
LLVM with JIT/Interpreter (toolChain)
LLVM with C/C++ Linker (toolChain)
LLVM archiver (tool)
LLVM assembler (tool)
LLVM C linker (tool)
builder (tool)
LLVM static compiler (tool)
LLVM JIT/Interpreter (tool)
LLVM with Clang (toolChain)
Abstract LLVM C/C++ compiler (tool)
LLVM with GCC (toolChain)
LLVM Clang (tool)
LLVM Clang++ (tool)
LLVM GCC (tool)
LLVM G++ (tool)
LLVM with static compiler (Linux) (toolChain)
LLVM with static compiler (Mac OS X) (toolChain)
LLVM with static compiler (Windows) (toolChain)
LLVM with Clang (Linux) (toolChain)
LLVM with Clang (Mac OS X) (toolChain)
LLVM with Clang (Windows) (toolChain)
LLVM with GCC (Linux) (toolChain)
LLVM with GCC (Mac OS X) (toolChain)
LLVM with GCC (Windows) (Cygwin) (toolChain)
LLVM with GCC (Windows) (MinGW) (toolChain)
LLVM with JIT/Interpreter (Linux) (toolChain)
LLVM with JIT/Interpreter (Mac OS X) (toolChain)
LLVM with JIT/Interpreter (Windows) (toolChain)
Debug Platform (targetPlatform)

LLVM toolchain/tool/targetPlatform configuration architecture for LLVM Eclipse CDT plug-in.

Layered package diagram



The diagram represents various packages such as managedbuilder.llvm (North) which contains all LLVM plug-in related packages. On the middle is managedbuilder.core and below it managedbuilder.makegen.gnu and some packages on the East side which all act as extension points for LLVM plug-in source code dependencies.