Instant IDEs: Supporting New Languages in the CDT

Jeffrey Overbey
University of Illinois at Urbana-Champaign
MC 258
201 North Goodwin
Urbana, IL 61801
overbey2@cs.uiuc.edu

Craig Rasmussen
Los Alamos National Laboratory
P.O. Box 1663
Los Alamos, NM 87545
rasmussn@lanl.gov

ABSTRACT

While Eclipse has greatly simplified the task of creating integrated development environments, creating a full-featured IDE can still take years. Fortunately, for a large category of languages—those that can be compiled with `make` and debugged with `gdb`—the task can be simplified greatly. By leveraging proposed multilingual extensions to the Eclipse C/C++ Development Tool (CDT), a modest IDE can be created in far less time. As a proof of concept, we have extended the CDT to support a `gcc`-based toy language (Eightbol); with the multilingual extensions in place, the Eightbol support code can be rewritten in its entirety in less than an hour. The same extensions have been used to implement version 3.0 of Photran, a full-featured IDE for Fortran.

This work is being funded by IBM under the PERCS project.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*program editors*

General Terms

Languages, Language Tools, Integrated Development Environments, Eclipse, CDT.

1. INTRODUCTION

The convenience of integrated development environments (IDEs) is virtually unquestioned. Although a few programmers prefer to use `emacs` and a plethora of independent command line tools (such as `gdb`, `cvs`, `diff`, and `find`), many prefer a graphical environment where similar tools are seamlessly integrated. Until very recently, IDEs have typically been the creations of large corporations—Borland, IBM, Sun, and Microsoft, to name a few—simply because they are very large, typically requiring several man-years to create.

Eclipse has simplified the creation of IDEs by creating a framework of common components—a customizable editor, source control, a debugger framework, and project support, among others. Even with all of this in place, creating a full-featured IDE remains an arduous task.

Fortunately, though, for many languages, this does not have to be the case. A large number of languages can be compiled with `make` and debugged with `gdb`. For these languages, it is preferable to have a narrower framework where a `make`-based build system, binary executable launcher, and `gdb`-based debugger already exist.

The Eclipse C/C++ Development Tool (CDT) [1] is an obvious starting point for such a framework: Its projects are compiled with `make`, it can execute native executables, and it contains a GUI interface to `gdb`. As part of our work on Photran [5], an Eclipse-based IDE for Fortran, we have identified a small number of modifications that allow the CDT to support other languages in addition to C and C++. We introduce a conceptual change in the CDT user interface and then describe an extension point which allows other plug-ins to add languages to the CDT with a surprisingly little amount of work.

2. CHANGES TO THE CDT

Before describing how to integrate a new language into the CDT, we will give a brief overview of how the current version of the CDT must be modified to support additional languages.

2.1 User Interface Changes

Figure 1 illustrates a typical editing session in CDT 3.0. Three things stand out as being C/C++-specific: (1) the editor and Outline view are customized for C/C++, (2) there is a C/C++ perspective, and (3) there is a C/C++ Projects view. The following are not visible in the screenshot but are also C/C++-specific: (4) there are C and C++ project types and New Project wizards, (5) the launcher menu item reads "Run Local C Application," and (6) many icons and images are decorated for C and C++.

Consider how these can be made more language-neutral. Regarding (1), we will return to the issue of editors and Outline views later, as these are necessarily language-specific. The remaining items (2–6), however, do *not* need to be C/C++-specific. Figure 2 illustrates two superficial changes: (2) the C/C++ perspective has been renamed to the Make perspective, and (3) the C/C++ Projects view

OOPSLA'05, October 16–20, San Diego, California, USA.

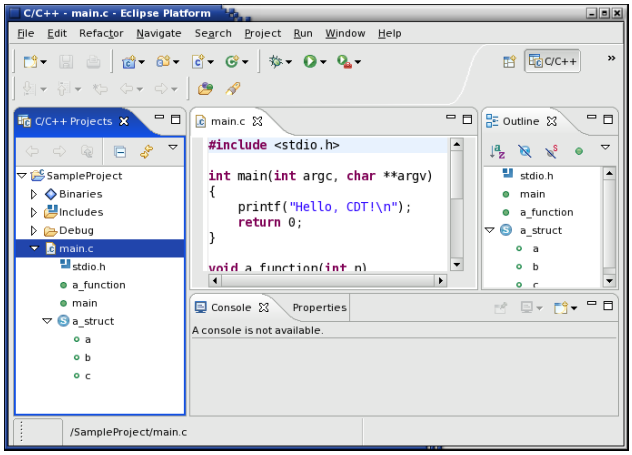

Figure 1: CDT 3.0 (Unmodified)

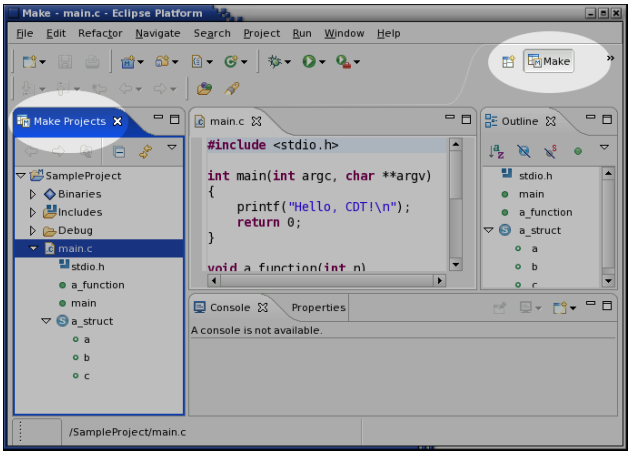

Figure 2: UI Modifications for Language Neutrality

is now the Make Projects view. Similarly, (4) the C and C++ project types and New Project wizards can be replaced with generic Make project types and a New Make Project wizard, (5) the launcher menu item can read "Run Local Application," and (6) icons and images can easily be made language-neutral. Most importantly, all of these changes are superficial: They involve changing labels and icons, but the underlying CDT code can remain untouched.

From the user's perspective, these changes turn the CDT into a generic IDE for projects that are built using `make`. Rather than creating a C/C++ project, the user creates a Make project; rather than running a C Local Application, he runs a (generic) Local Application. In this environment, it makes just as much sense to develop a Fortran or Ada application as it does to develop a C/C++ application.

2.2 Core Changes

Observe the C/C++ Projects view in Figure 1 or, equivalently, the Make Projects view in Figure 2. Notice that the CDT has identified `main.c` as being C source code and is able to provide a high-level outline of its contents. We would like it to do the same for other languages.

The Make Projects view is essentially a visualization of the CDT *model*, a tree data structure which records the contents of all Make Projects in the workspace. The highest levels of the model mirror the resource tree, i.e., the model includes all of the folders and files in each project. More importantly, though, when a file is identified as source code, that file is parsed, and its high-level contents are added to the model as well.

To integrate additional languages into the model, we must do two things. First, we must allow the CDT to identify non-C/C++ files as valid source code. Second, we must provide a means for parsing languages other than C and C++ so that the model can include functions, classes, etc. in non-C/C++ source files.

We can accomplish both of these by adding an extension point to the CDT Core plug-in. Not surprisingly, extensions will be asked to (1) identify the types of files they can parse, and (2) provide a *model builder* which actually does the parsing and populates the model. The next section discusses this in more detail.

3. INTEGRATING A NEW LANGUAGE

Amid the discussion in the last section, we identified three things that must be supplied before a language is truly integrated into the CDT:

- an editor,
- a list of (source code) filename extensions, and
- a model builder.

There are two aspects of the CDT that did not have to be modified to support additional languages but will need to be considered when integrating a new language into the CDT:

- error parsers, and
- the managed build system.

3.1 Building an Editor, Part 1

The first step in integrating a new language is to create a new plug-in and start building an editor for the language. Currently, this means building an ordinary Eclipse editor, usually based on JFace Text.[1] Editors are described well in several books, including [2].

The editor will be modified in Section 3.6 to integrate with the CDT. Specifically, we will replace its document provider and ruler context menu, and we will allow CDT to provide the contents of the Outline view.

3.2 Registering Content Types

Rather than looking directly at file extensions, the CDT uses *content types* to identify what files correspond to source code in each supported language. When a language is added, a content type for its source files needs to be registered with the Eclipse runtime. Each source filename extension for the language should be associated with this content type. For example:

[1]Can it be easier than this? Maybe. However, nearly all of the customizable aspects of an editor—syntax highlighting schemes, auto indent strategies, double-click strategies, formatting strategies, folding, and content assist schemes—tend to be language-specific. At the same time, one could argue that each language should not have to supply its own preference pages for syntax coloring. So while a CDT editor framework does not exist now, it might in the future.

```
<extension point="org.eclipse.core.runtime.contentTypes">
   <content-type
         id="xyzSource"
         name="XYZ Language Source File"
         base-type="org.eclipse.core.runtime.text"
         priority="high"/>
   <file-association
         content-type="xyzSource"
         file-extensions="xyz"/>
</extension>
```

It is important to make sure that the list of filename extensions declared here matches the list of filename extensions associated with the editor created in the previous step.

3.3 Building an `IAdditionalLanguage`

Once an editor has been created and a content type declared, we can begin to integrate with the aforementioned CDT extension point.

First, we need to create a class, say `XYZLanguage`, which implements the `IAdditionalLanguage` interface declared in the `org.eclipse.cdt.core.addl_langs` package:

```
public interface IAdditionalLanguage {
  public String getName();
  public Collection<String> getRegisteredContentTypeIds();
  public IModelBuilder createModelBuilder(
    org.eclipse.cdt.internal.core.model.TranslationUnit tu,
    Map newElements);
}
```

The `getName` method should return the name of the language being added, e.g., "Fortran" or "XYZ Sample Language." The method `getRegisteredContentTypeIds` simply lists the content type(s) declared in the previous step. Each content type name should be fully qualified, i.e., the name of the plug-in followed by the name of the content type (`XyzLanguagePlugIn.xyzSource`). We will postpone our discussion of `createModelBuilder` until the next section.

After the `XYZLanguage` class has been created, we can tie it into the `AdditionalLanguages` extension point in the CDT Core plug-in. First, we must specify `org.eclipse.cdt.core` as a dependency of our new language plug-in. Then we can extend the `AdditionalLanguages` extension point, supplying the fully-qualified name of our class:

```
<extension point="org.eclipse.cdt.core.AdditionalLanguages">
   <language class="com.mycompany.XYZLanguage"/>
</extension>
```

3.4 Building a Model Builder

As described previously, each additional language needs to supply a model builder which can parse files in that language and add structural information to the CDT model. This is done via the `createModelBuilder` method.

A model builder is simply an implementation of the interface `org.eclipse.cdt.core.addl_langs.IModelBuilder`. Its constructor takes a `TranslationUnit`, i.e., a file for which a model needs to be created. Its `parse` method returns a `Map` containing the elements that should appear in the Outline view for that file.

A trivial `parse` method would look something like this (the Outline produced is shown in Figure 3):

```
public Map parse(boolean quickParseMode) throws Exception {
  Map newElements = new Map();

  // Create a namespace as a child of the translation unit
  Namespace ns = new Namespace(translationUnit, "A Namespace");
  translationUnit.addChild(ns);
  newElements.put(ns, ns.getElementInfo());

  // Create a typedef as a child of the namespace
  TypeDef td = new TypeDef(ns, "A Typedef");
  ns.addChild(td);
  newElements.put(td, td.getElementInfo());

  // No parse errors were encountered
  translationUnit.getElementInfo().setIsStructureKnown(true);
  return newElements;
}
```

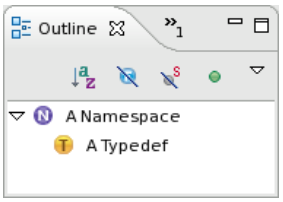

Figure 3: Outline produced by the trivial model builder

Naturally, a model builder for a "real" language will be more complicated than this, but the model is constructed in the same way. When a parser for the language is available, typically it will construct an abstract syntax tree for the translation unit, and the model can be built by a Visitor [4] on that tree. When a parser is not available, a more *ad hoc* method may be used. For example, in C++, classes can be identified by tokenizing the input and scanning for the keyword `class` followed by an identifier.

3.5 Building New Model Elements

In the sample `parse` method above, we reused two model classes from the CDT: `Namespace` and `TypeDef`. There are also model classes for function declarations, structs, classes, and a number of other entities. (For a complete list, browse the `org.eclipse.cdt.core.model.ICElement` type hierarchy.) For many languages, however, the existing model classes are not sufficient. For example, Fortran has modules, namelists, and block data, none of which have counterparts in C or C++.

To create a new model class, we simply need to implement `IAdditionalLanguageElement`, another interface defined in the `org.eclipse.cdt.core.addl_langs` package. `IAdditionalLanguageElement` extends `ICElement`, adding a method `getBaseImageDescriptor` which returns an icon for that element type, suitable for display in the Make Projects and Outline views.

Although one is free to implement all of the `ICElement` methods directly, it is far easier to subclass from a CDT class. Photran's model elements all subclass from the CDT's `SourceManipulation` class, for example.

```
public abstract class SampleElement
  extends SourceManipulation
  implements ICElement, IParent, ISourceReference,
             IAdditionalLanguageElement {

  public SampleElement(Parent parent, String identifier) {
    super(parent, identifier, -1);

    // To set position information within the file:
    // setIdPos(offset, length);
    // setPos(offset, length);
    // setLines(startLine, endLine);
  }
```

```
  public Object getBaseImageDescriptor() {
    return XyzLanguagePlugIn.
                getImageDescriptor("icons/sample.gif");
  }
}
```

3.6 Building an Editor, Part 2

After a model builder is functioning, source files for our new language will have their functions, subroutines, etc. displayed in the Make Projects view, just like C/C++ files. Now we will finish integrating our editor with the CDT. First, we will integrate the CDT outline page, which uses the model to display an outline of the file being edited. Then we will make a slight change so that the user can set breakpoints in our editor.

Before we can use the CDT outline page, we need to replace our editor's document provider with the CDT's. After `org.eclipse.cdt.ui` has been added as a dependency, this can be done in the editor's constructor with the following line:

```
setDocumentProvider(CUIPlugin.getDefault().getDocumentProvider());
```

Typically, the document provider is where we set up our editor's partitioner, which controls syntax highlighting. Since we do not have our own document provider anymore, we need to do this elsewhere. We recommend overriding

```
protected void doSetInput(IEditorInput input)
```

in the editor class and setting the partitioner there.

To actually integrate the Outline page, we need to implement `getAdapter` in the editor class as follows.[2]

```
protected CContentOutlinePage fOutlinePage;

public Object getAdapter(Class required) {
  if (IContentOutlinePage.class.equals(required)) {
    return getOutlinePage();
  }
  if (required == IShowInTargetList.class) {
    return new IShowInTargetList() {
      public String[] getShowInTargetIds() {
        return new String[] { CUIPlugin.CVIEW_ID,
          IPageLayout.ID_OUTLINE,
          IPageLayout.ID_RES_NAV };
      }

    };
  }
  return super.getAdapter(required);
}

public CContentOutlinePage getOutlinePage() {
  if (fOutlinePage == null) {
    // For now, at least, the editor parameter can be null
    fOutlinePage = new CContentOutlinePage(null);
    fOutlinePage.addSelectionChangedListener(this);
  }
  setOutlinePageInput(fOutlinePage, getEditorInput());
  return fOutlinePage;
}

public static void setOutlinePageInput(
    CContentOutlinePage page,
    IEditorInput input) {
  if (page != null) {
    IWorkingCopyManager manager =
      CUIPlugin.getDefault().getWorkingCopyManager();
    page.setInput(manager.getWorkingCopy(input));
  }
}
```

[2] All of the methods in this section are identical between languages, and nearly all are copied from `CEditor`. In the future, we plan to factor these methods into a common superclass shared among the `CEditor` and editors for other languages.

After the outline page is functioning, we may want the editor to jump to the corresponding location when the user selects an item in the Outline view. To do this, the editor needs to implement `ISelectionChangedListener`. (If this behavior is not needed, remove the line

```
fOutlinePage.addSelectionChangedListener(this);
```

from the code copied above.) For the time being, the easiest way to do this is to copy the following methods from `CEditor`:

```
public void selectionChanged(SelectionChangedEvent event)
private boolean isActivePart()
public void setSelection(ISourceRange element,
                         boolean moveCursor)
```

The only thing missing at this point is that the user cannot set debug breakpoints in our editor. The CDT Debugger plug-in contributes a Toggle Breakpoint menu item to the context menu for the CDT editor's ruler. The easiest way to get the same actions in our editor is to "borrow" the CDT editor's ruler context menu. This can by done by appending the following line to our editor's constructor:

```
setRulerContextMenuId("#CEditorRulerContext");
```

Now we have a fully functioning editor. We can create Make projects containing files for our language, edit those files, and see them outlined in the Make Projects and Outline views. If the underlying compiler for our language produces binaries that can be debugged with `gdb`, we can do that too—the CDT debugger will recognize breakpoints set in our editor, and when those breakpoints are hit, Eclipse will open our editor and allow the user to step through the code as expected.

3.7 Building an Error Parser

When a user builds a program and `make` is run, the output appears in the Console view. When the compiler produces error messages, it is helpful for them to appear in the Problems view as well, with corresponding red markers appearing in the editor. This is done by *error parsers*.

Error parsers scan the output of `make` for error messages from their associated compiler. When they see an error message they can recognize, they extract the filename, line number, and error description, and use this information to populate the Problems view.

The CDT includes error parsers for several C/C++ compilers, including `gcc` and Visual C++. However, if a compiler for a new language does not produce error messages in one of those formats, a new error parser will need to be written.

Fortunately, error parsers are straightforward to write. The following is Photran's error parser for Intel Fortran 8.1:

```
/**
 * Error parser for Intel Fortran 8.1: Extracts file , line number,
 * and error message from lines of the form
 *     fortcom: Error: test.f90, line 3: Message
 */
public class IntelFortranErrorParser implements IErrorParser {
  public boolean processLine(String line, ErrorParserManager mgr)
  {
    String fortcom, severitystr , filestr , linestr , message;

    StringTokenizer tokenizer = new StringTokenizer(line, ":");
    if (line.startsWith("fortcom: ")) {
      try {
        fortcom = tokenizer.nextToken();
        severitystr = tokenizer.nextToken().trim();
        filestr = tokenizer.nextToken(",").substring(2).trim();
        linestr = tokenizer.nextToken(":").substring(2).trim();
```

```
        message = tokenizer.nextToken("\r\n").substring(2).trim();

        int severity = (severitystr.equals("Error")
            ? IMarkerGenerator.SEVERITY_ERROR_RESOURCE
            : IMarkerGenerator.SEVERITY_WARNING);
        int lineno = Integer.parseInt(linestr.substring(5));
        IFile file = mgr.findFilePath(filestr);

        mgr.generateMarker(file, lineno, message, severity, null);
      }
      catch (Throwable x) {;}
    }
    return false;
  }
}
```

Essentially, we (1) try to tokenize the error message, (2) map the filename to an `IFile` in the workspace, and then (3) call `generateMarker` to add a marker which will display in the editor and Problems view.

New error parsers can be added to the CDT by extending the `ErrorParser` extension point in its Core plug-in as follows.

```
<extension id="IntelFortranErrorParser"
     name="Photran Error Parser for Intel Fortran 8.1"
     point="org.eclipse.cdt.core.ErrorParser">
  <errorparser class="org.photran.IntelFortranErrorParser" />
</extension>
```

3.8 Managed Build System Integeration

One final aspect of language integration remains to be considered. On some projects, users prefer not to maintain their own makefiles; they would rather have a makefile be automatically generated and automatically updated as source files are added to and removed from their project.

The CDT provides *managed make* projects for this purpose. The *Managed Build System* (MBS) is responsible for maintaining makefiles in these projects, tracking dependencies and updating the makefile as the project changes.

We will not discuss the details of MBS integration in detail; this is already done very thoroughly in the *Managed Build System Extensibility Document,* which is available from the "Reference Documentation" link on the CDT home page [1].

4. EIGHTBOL: A PROOF OF CONCEPT

To illustrate how quickly a new language can be integrated into the CDT, we used the technique described here to build an IDE for a toy language called Eightbol. Moreover, we have reworked the creation of this IDE as a tutorial, which can typically be completed in less than an hour.

Eightbol is a trivial compiled language with an XML-like syntax.[3] (A sample program exploiting all of Eightbol's language features is shown in Figure 4.) An `gcc`-based front end for Eightbol is available, which allows it to be compiled in a makefile and debugged with `gdb`.

```
<?xml version="1.0"?>
<program>
  <section name="Sections are just for code readability">
    <print string="We believe that Eightbol's lack of" />
    <print string="Turing-completeness is overshadowed by" />
    <print string="its fashionable XML-based syntax and" />
    <print string="Eclipse-based IDE." />
  </section>
</program>
```

Figure 4: Sample Eightbol program

The Eightbol IDE tutorial uses Eclipse's sample XML editor as a starting point and generally follows the discussion in the previous section. The editor is configured with a content type and associated with the filename extension "8bl." An `IAdditionalLanguage` is created and fitted with a model builder based on SAX (thanks to Eightbol's XML-based syntax). Finally, the CDT Outline is tied into the editor, at which point the IDE is complete, and CVS support, debugging, and error parsing can all be illustrated. Due to space constraints, the Eightbol tutorial cannot be reproduced here; however, the Eightbol compiler, IDE, and tutorial are available from the Eightbol Web site [3].

5. FUTURE WORK

Currently, the described user interface changes and the `AdditionalLanguages` extension point do not exist in the official release of the CDT. However, Photran 3.0 [5] includes a copy of CDT with these modifications. We are actively working with the CDT committers to integrate these changes into the CDT proper.

The CDT changes described here represent a significant step toward multi-language support. However, they are not complete. For example, the current changes offer no means of integrating refactoring support, and integration with the C/C^{++} indexer has not been investigated at length.

The ultimate goal is to provide a compiled language IDE framework. To fully realize this, all languages (including C and C^{++}) should be implemented via an extension point, effectively separating the current CDT into a language-independent framework and a C/C^{++} plug-in.

As these changes take place, we can expect the CDT (and the API for adding additional languages) to undergo significant changes. The version discussed here is preliminary; however, it is still extremely useful. Photran 3.0 is built entirely as described, and was done so in a matter of weeks. The aforementioned user interface changes and `AdditionalLanguages` extension point provide an effective way to build IDEs for compiled languages in surprisingly little time.

6. REFERENCES

[1] *Eclipse C/C++ Development Tools* http://www.eclipse.org/cdt/

[2] D'Anjou, J., S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse,* 2nd edition. Addison-Wesley, 2005.

[3] *The Eightbol Project.* http://www.eightbol.org/

[4] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software.* Addison-Wesley, 1995.

[5] *Photran, an Eclipse Tool for Fortran Developement* http://www.photran.org/

[3] XML syntax was chosen so that Eclipse's sample XML editor could be used in the tutorial without modification of its syntax highlighting code. Under other circumstances, we would have preferred a good syntax instead.