Photran-CDT Compatibility Feature

This material is partly based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

Table of Contents

Jeffrey Overbey
(overbey2@illinois.edu)
Summer Intern, IBM Research
August, 2008

Summary

This document describes a prototype CDT-compatible program representation and infrastructure for Fortran. The program representation consists of a heavyweight abstract syntax tree (AST) based on the CDT DOM, a lightweight AST ("model") based on the CModel, and a cross-reference database based on the CDT PDOM. The prototype uses the Open Fortran Parser to build both the DOM and the model, although a DOM builder based on Photran's existing AST and program database is included as well, as is a small sample illustrating how to use the PDOM to support loading and storing information about Fortran modules.

Architecture

The prototype CDT Compatibility feature is based on Photran 4.0 beta 4, CDT 5.0, and Eclipse 3.4 and requires a Java 1.5 compiler.

The CDT Compatibility Feature consists of the following Eclipse projects.

- **org.eclipse.photran.cdtcompatibility-feature**
  Eclipse feature comprised of the following plug-ins.
- **org.eclipse.photran.cdtcompatibility-dev-docs**
  Contains this document and the notes/progress reports from which it was compiled.
- **org.eclipse.photran.cdtcompatibility.core**
  Contains DOM, PDOM, and CModel classes for Fortran. (These do not and should not depend on any particular parser.)
- **org.eclipse.photran.cdtcompatibility.core.ofp**
  Contains the Open Fortran Parser (OFP) and contributes a DOM builder and model builder based on OFP.
- **org.eclipse.photran.cdtcompatibility.core.vpg**
  Contributes a DOM builder which uses Photran's existing AST and indexer to build a (CDT-compatible) DOM. This was developed as a proof-of-concept and is not intended to be completed.
- **org.eclipse.photran.cdtcompatibility.core.tests**

  JUnit tests for the Open Fortran Parser. Eventually, this should contain DOM tests as well.

- **org.eclipse.photran.cdtcompatibility.ui**
  Contributes the Fortran DOM AST view and a Find Module action.

Implementation Notes

Extending Photran

Mechanism

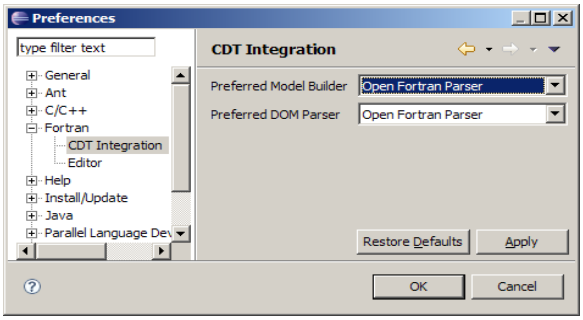
The CDT Core (essentially the model, AST, and indexer) can be extended to support new languages via an extension point. Extensions provide (1) a list of platform content types supported by that language, (2) a model builder, and (3) a DOM parser. Photran contributes to this extension point via the cdtinterface plug-in; however, this plug-in, in turn, provides an extension point through which a Fortran model builder and DOM parser may be contributed. Photran's extension point also allows a reconciler to be contributed to the Fortran editor.[1]

Photran's refactoring-centric language infrastructure (the VPG, or Virtual Program Graph) is included in the core.vpg and ui.vpg plug-ins; these contribute a model builder and a reconciler but not a DOM parser. (The reconciler provides the Fortran editor with content assist and synchronization with the Fortran Declaration view.)

The CDT Compatibility Feature provides a model builder and DOM parser via the language extension point but currently does not provide a reconciler.[2] (See plugin.xml in the core.ofp plug-in.)

Usage

When both the VPG and CDT Compatibility features are installed, Photran allows the user to choose which model builder and DOM parser



1 Photran's architecture is described in detail in the *Photran Developer's Guide,* which is linked from the Contributor Info page on Photran's Web site.

2 Photran's existing parser and program representation , as well as UI contributions and editor features based on these, are entirely contained in two plug-ins which are separate from the rest of Photran. Although it is desirable for the existing representation and the CDT-compatible representation to coexist (as they do now), it is possible to build Photran with either omitted.

to use via the Fortran > CDT Integration category in the workspace preferences.

DOM (Heavyweight AST)

CDT's full abstract syntax tree (AST) is also known as the DOM (à la XML). The structure of a DOM is implicitly defined by the interfaces in the org.eclipse.cdt.core.dom.ast package. All DOM nodes are expected to implement IASTNode; IASTTranslationUnit is the root of the DOM; there are several interfaces such as IASTDeclaration, IASTStatement, and IASTExpression which are particularly useful for defining Fortran constructs that have no analog in C or C++; and there are many other node types (e.g., IASTDoStatement, IASTCastExpression) which are much more C/C++-specific.

DOM Design

There are two competing forces in developing a CDT-compatible DOM for Fortran.

On one hand, code already written for CDT generally expects a DOM to represent a C/C++ program; this is implicit in the assumptions that are made about the syntactic and lexical structure of the program (e.g., functions are not nested, identifiers are case-sensitive and space-free) as well as its semantics (e.g., all identifiers are declared before use). Superficially, it appears that this code will be reusable by building a C/C++ DOM which approximates a Fortran program. For example, CDT is able to semantically highlight UPC *forall* loops because they are represented as *for* loops in the DOM.

On the other hand, the *purpose* of an AST (DOM) is to represent the syntactic structure of the program. A proper AST for Fortran will provide Fortran-specific constructs (like modules, common blocks, and I/O statements) their own node types.

The DOM in the CDT Compatibility feature generally follows the latter approach. Fortran and C/C++ are very different languages. Many Fortran constructs have no reasonable analog in C or C++ (e.g., *implicit* statements, implied do loops, nested procedures, intrinsic declarations, variable kinds, common blocks, block data subprograms, computed gotos, keyword parameters, and *where* statements, among many others). In these cases, building a 100% C/C++-compatible DOM would

essentially amount to building a Fortran-to-C/C++ compiler, and even then the DOM would contain nodes with no corresponding source code while some regions of source code would be mapped to several DOM nodes. The utility of this data structure for static analysis would be limited (e.g., consider computing flow information from a serial C/C++ approximation of Fortran's *where* statement or array sections or computing binding information based on an approximation of nested procedures), while the DOM would be utterly useless as a Fortran-specific program representation due to all of the syntactic and semantic information that would be lost in translation.

In general, the Fortran DOM implements CDT DOM interfaces (and occasionally subclasses CDT DOM node implementations) at the lowest level where it is appropriate. For example, IFortranSubprogram implements IASTFunctionDefinition since a Fortran subprogram definition directly obeys the the syntactic contract of a C/C++ function definition,[3] while IFortranPrintStmt implements only the high-level IASTStatement interface since there is nothing equivalent to a *print* statement in C/C++.

Fortran DOM Nodes

A very incomplete set of (demonstration) Fortran DOM nodes is located in the cdtcompatibility.core plug-in. The externally-visible interfaces of Fortran DOM nodes are defined in the org.eclipse.photran.cdtcompatibility.core.dom package; implementations not intended for public consumption are contained in the org.eclipse.photran.cdtcompatibility.internal.core.dom package. Every Fortran DOM node is expected to implement IFortranASTNode (which, of course, subclasses CDT's IASTNode).

Currently, DOM nodes have only been implemented for some high-level organizational/scoping constructs (main programs, subprograms, modules, and block data subprograms), *call* and *print* statements, and some expressions (unary, binary, identifiers, and literal expressions).

3 This may change. In every case – the DOM, the PDOM, and model – the Fortran implementation started by reusing CDT classes frequently. However, as more pieces were developed and more Fortran specifics were needed, it became necessary to either subclass CDT classes or simply build entirely new objects which implemented the same interfaces. Whether a C/C++ node is a "good enough" approximate representation of a Fortran construct depends on what that node is being used for. This is true of AST design in general: An AST for a Fortran compiler could omit many declaration statements, for example, while a Fortran AST for an IDE or refactoring tool probably would not.

DOM Builders

Two DOM builders have been prototyped. (Again, the one to use may be selected in the workspace preferences.) These are implemented in plug-ins separate from the DOM nodes themselves; note that parser-specific classes (e.g., Token) are used only in the DOM builders, *not* in the DOM node interfaces or implementations.

A first implementation of a Fortran DOM builder is contained in the cdtcompatibility.core.vpg plug-in: It uses Photran's existing AST and indexer to construct a DOM. This was used for prototyping the DOM and is not intended to be completed or maintained.

The main implementation (in the cdtcompatibility.core.ofp plug-in) uses the Open Fortran Parser to construct a DOM. Its implementation is not immediately obvious and is described in more detail in the next section ("Open Fortran Parser").

Visitors

All DOM nodes (i.e., objects implementing IASTNode) implement an #accept(IASTVisitor) method which allows them to be traversed using a variant of the Visitor design pattern.

The cdtcompatibility.core plug-in contains a utility class (UniformCDTASTVisitor) which can be used to implement a Visitor that treats every type of DOM node identically.

When traversing a Fortran DOM, it is often preferable to subclass FortranASTVisitor: This class contains a callback method for each type of node in a Fortran DOM.

Prettyprinting and the Fortran DOM AST View

Two means are provided to assist with debugging and testing the Fortran DOM.

The Fortran DOM AST View (in the cdtcompatibility.ui plug-in) is copied and from the CDT's DOM AST View and modified slightly to support the Fortran DOM. However, *it is not very reliable:* CDT's DOM AST View code is not particularly well-written and is very specific to CDT's C/C++ DOM, so, consequently, the Fortran DOM AST view does not always produce the "correct" display.

A more reliable depiction of the Fortran DOM is provided by the FortranPrettyPrinter class, which prettyprints source code from the DOM. (Note that the FortranPrettyPrinter is intended to be a simplest-possible implementation that can be used for testing and debugging only; it is not intended to be a production prettyprinter or code formatter.)

DOM Testing

The prototype CDT Compatibility feature contains almost no code for testing the DOM, although comprehensive testing will be critical for a full-scale DOM implementation. Fully testing the DOM will involve

1. identifying the contexts in which a particular node should appear, constructing programs accordingly, and ensuring that the appropriate node exists in the DOM;

2. testing accessor methods for DOM nodes, ensuring that all necessary information about a construct is publicly-visible;

3. testing connectivity, ensuring that parent-child relationships are 1-1 among DOM nodes; and

4. testing the Visitor mechanism, ensuring that every node is visited in the correct order and that the CDT's extensive set of visitor controls (e.g., boolean returns and the many shouldVisitXYZ fields) are observed appropriately.

Open Fortran Parser (OFP)

JAR

The Open Fortran Parser is contained in ofp.jar in the root of the cdtcompatibility.core.ofp plug-in. Inside the *build* folder, there is an Ant script (create-ofp-jar.xml) which will build this jar by checking out the OFP sources from the SourceForge repository, compiling them, and then bundling the result.

Note that OFP is continually under development, and so the jar must be rebuilt using the Ant script for modifications in OFP to be reflected in the CDT Compatibility feature.

Tests

A large suite of JUnit tests for OFP (adapted from similar tests for Photran's parser) is located in the cdtcompatibility.core.tests plug-in. These tests simply parse Fortran programs and ensure that a non-null AST is returned.

The base set of test Fortran codes is located in the org.eclipse.photran.core.vpg.tests plug-in. Several other codes (including POP, IBEAM, FMLIB, LAPACK, and the IBM XL Fortran 12.1 test suite) are not available in public CVS; tests for these codes will fail silently if they are not present.[4]

The four base classes in org.eclipse.photran.internal.core.tests are copied from identically-named classes for Photran's parser and modified to use OFP and return a DOM instead. The test classes in the org.eclipse.photran.internal.core.tests.a_parser package are all *copied verbatim* from the same package in Photran's core.vpg.tests plug-in; however, since these copied are located in the cdtcompatibility.core.tests plug-in, they use the four base classes described above and so function as tests for OFP rather than Photran's parser.

DOM Building

The Open Fortran Parser was designed to support arbitrary semantic actions. The IFortranParserAction interface has one callback method corresponding to each rule in the grammar; OFP can be given an arbitrary object implementing this interface, and the corresponding actions will be called as grammar rules are matched.

However, these actions must be implemented using a stack-based model. Each callback method is expected to push objects onto a stack; subsequent method calls should pop these objects based on parameters supplied to the the callback methods. For example, suppose the callback for expressions ("expr") pushes an ExpressionNode object on the stack. Now, suppose that this expression is used in the context of a named constant definition. According to OFP's ANTLR grammar, FortranParser.g,

4 All of these except for the XLF test suite are available in a private CVS repository; contact the author for access information.

```
// R539
named_constant_def
      :     T_IDENT T_EQUALS expr
      {action.named_constant_def($T_IDENT);}
      ;
```

the named_constant_def callback method will be invoked, and the identifier token will be passed to it as a parameter. This ExpressionNode object must be popped from the stack.

Unfortunately, IFortranParserAction contains 486 methods, and it is difficult (if not impossible) to maintain a "correct" stack when some methods have been implemented but others have not (e.g., if one method pushes an object onto the stack but the corresponding method that should pop it has not been implemented).

To remedy this, the OFP DOM builder attempts to use a more familiar attribute grammar model, but it does so by building on top of OFP's stack-based model. Here, there is one stack *for each nonterminal in the grammar*. The callback method for a particular grammar rule should push a DOM node for the nonterminal on its left-hand side (or some object, such as a String or Integer, that will later be incorporated into a DOM node). For example, the grammar rule

```
// R313
// ERR_CHK 313 five characters or less
label returns [Token tk]
    : T_DIGIT_STRING
    { tk = $T_DIGIT_STRING;
      action.label($T_DIGIT_STRING); };
```

is given the following callback method.

```
/** R313 */
public void label(Token lbl) {
    attr.pushFragment(AttrKey.label,
        Integer.parseInt(lbl.getText()));
}
```

Scoping, Binding Resolution, and Module Loading

Like many IDEs, rather than using symbol tables (as one would find in a compiler), CDT's scoping/binding resolution is based the AST: Some AST nodes (e.g., function definitions) have an associated *scope* (an object implementing IScope), and each scope contains a collection of *bindings*. Every scope has a name and a parent scope (except

the AST root – a translation unit – whose parent scope is null).

CDT's code for scoping and binding resolution is contained mostly in static methods in the CVisitor class. This code is very dependent on the structure of the DOM (note the number of instanceof tests) and so is not reusable for Fortran. (The scoping/binding rules for C/C++ and Fortran are different anyway.)

A sample scope and three sample bindings for Fortran have been implemented. The public interfaces are IFortranScope and IFortranBinding; the implementations are FortranScope, FortranImplicitBinding, FortranModuleBinding, and FortranSubprogramBinding. Scopes are associated with translation units, block data subprograms, modules, main programs, and subprograms (functions and subroutines). Bindings are assigned to the identifiers in module and subprogram definitions when the corresponding DOM nodes are created. Currently, every *use* of an identifier is assigned a FortranImplicitBinding (although this is incorrect – it was done simply to make the PLDT MPI artifact visitor work, cf. below); a full implementation would obviously need to resolve identifiers to the subprogram/variable/etc. that they actually reference in a manner conceptually similar to CVisitor.

PDOM

The CDT's PDOM (Persistable DOM) maintains a collection of scopes and definitions in each file and (when used as a C/C++ indexer) is updated incrementally in response to changes to workspace resources.

Linkage Association

Each language supported by CDT is associated with a PDOM *linkage;* this association is currently specified in four places:

1. FortranASTTranslationUnit#getLinkage
2. FortranASTName#getLinkage
3. FortranSubprogramBinding#getLinkage
4. FortranLanguage#getLinkageID

As additional bindings (other than FortranSubprogramBinding) are added to the PDOM, the linkage will be specified there as well.

The linkage is contributed to CDT via the org.eclipse.cdt.core.language extension point.

Linkage Implementation

Fortran's PDOM linkage is defined in the class PDOMFortranLinkage, which is based on CDT's PDOMCLinkage class.

The types of symbols that may be stored in the PDOM are

1. given an integer constant in the IIndexFortranBindingConstants interface,

2. implemented as a class in the org.eclipse.photran.cdtcompatibility.internal.core.pdom package (see PDOMFortranModule and PDOMFortranVariable for examples), and

3. serialized/deserialized in PDOMFortranLinkage (#addBinding serializes, #getNode deserializes).

The method PDOMFortranLinkage#adaptBinding is not necessary for the small Fortran PDOM prototype will likely be more useful in a full implementation: It should convert between DOM bindings and PDOM bindings. See PDOMCLinkage for example code.

PDOM Binding Implementation

Two sample Fortran PDOM bindings have been implemented. These are supposed to be *illustrative* only: Much of their code has been copied from similar CDT classes, although this should be avoided in a full implementation.

PDOMFortranVariable is a simple binding largely copied from PDOMCVariable.

PDOMFortranModule is more complex, as a module is a binding which also doubles as a scope for other bindings.

Note that every binding has a scope, and the “outermost” bindings in a

translation unit have the PDOMFortranLinkage object itself as their scope. Every binding also has a name and may also have additional fields (representing the type or other attributes of the symbol being defined).

Viewing the PDOM

The Fortran PDOM (treated as a C/C++ Index) may be viewed in the C/C++ Index View.

Model (Lightweight AST)

The CDT Compatibility feature also contains an (incomplete) OFP-based model builder and model elements for Fortran. These are contained in the *model* folders in the core and core.ofp plug-ins; new (contributed) model elements are mapped to their corresponding images via an array at the top of the class org.eclipse.photran.cdtcompatibility.internal.ui.Activator. The model builder resembles the DOM builder, and the model elements resemble DOM nodes, although their type hierarchies (and purposes) are different. These will not be described in detail here; more information about building CDT model builders and model elements is availabe in the following paper:

J. Overbey and C. Rasmussen, “Instant IDEs: Supporting New Languages in the CDT,” Eclipse Technology eXchange Workshop at OOPSLA 2005, San Diego, CA, October 17, 2005.
http://jeff.over.bz/papers/2005/instant-ides.pdf

PLDT Integration

An additional project (org.eclipse.ptp.pldt.mpi.core.photran) contains two implementations of PLDT's *Find MPI Artifacts* action. One (PhotranVPGMPIAnalysis) is based on Photran's existing program representation; the other (PhotranDOMMPIAnalysis) is based on the CDT-compatible DOM. Currently, the VPG-based analysis is more accurate, since the VPG is a complete AST and contains binding information, but the DOM-based analysis is a useful demonstration of what can be done with the CDT-compatible DOM. In particular, the inner class in PhotranDOMMPIAnalysis (MpiFortranASTVisitor) is nearly identical to MpiCASTVisitor.

If the user selected a Fortran DOM builder in the workspace preferences

(see above), the DOM-based analysis will be used; if no DOM builder was selected, or if the DOM parser fails, the VPG-based analysis will be used instead. (See the class PhotranMPIAnalyzer.)

Problems and Future Work

Open Fortran Parser

- **Parser Bugs.** As of 6/6/08, OFP fails 65 of Photran's 3485 unit tests, 72 of 1936 tests in a sample of the XLF 12.1 test suite, and crashes the JUnit test runner when attempting to run the full test suite. (The last attempt completed 14660/62860 runs with 9659 failures.)

- **Design.** OFP, in particular its FrontEnd (the parser entrypoint), is intended for command line use and needs to be modified for use in Eclipse. In particular,

    - Errors should be handled by a callback, not printed to stderr.

    - INCLUDE line processing should be handled by a callback, and include directories should be configurable from the project properties (perhaps they should be the same as the include directories configured for CDT's C preprocessor).

    - The fixed/free-form test should be based on Eclipse content types, not filename extensions.

    - OFP is very tied to Java File objects and subclasses ANTLRFileStream. However, its input should be an arbitrary Reader or InputStream, not necessarily a File. This is critical for parsing the active editor's (unsaved) content (e.g., for the Outline view) and for parsing files that do not exist on the local filesystem. Furthermore, INCLUDEd files may not exist on the local filesystem and may need to be treated as Eclipse IFile objects instead.

    - ANTLR Tokens do not contain line/column information.

    - As discussed earlier, the stack-based programming model can be somewhat confusing and error-prone, even with an attribute grammar model built on it, due to the size of the Fortran

grammar and the number of actions involved. Whether, and how, this can be improved should be open for discussion.

- **Action Bugs.** In the course of implementing a few DOM nodes, the following bugs were found and posted to the fortran-parser-devel list at SourceForge.
  - Need a callback method for R1231 (subroutine_subprogram)
  - R208 (execution_part) needs count parameter for execution_part_construct
  - R429 (derived_type_def) needs count for private_or_sequence, component_def_stmt, type_bound_procedure_part
  - R705 (add_operand) needs flag for initial add_op
  - R1107 (module_subprogram_part) needs count for module_subprogram
- **Efficiency.** FMLIB/FM.F90 (37 KLOC) takes 2563 ms to parse with no parser action and 3563 ms with a DOM building action. Whether this is acceptable, and how to improve the speed if it is not, should be discussed.

C Preprocessor

OFP will eventually need to be integrated with a C preprocessor, perhaps the one in CDT. This effort is currently (8/2008) underway at UIUC, but it is not complete.

CDT's preprocessor operates quite directly as defined in the C Language Spec, which means that it doubles as a tokenizer for CDT. Its output is (essentially) a stream of Token objects which are consumed by the C/C++ parser. On the contrary, most Fortran programs assume that the preprocessor outputs a stream of text (similar to gcc -E). The only real difference between the two – and perhaps the biggest problem for Fortran – is that all information about spacing and line endings is disposed of by CDT's preprocessor. Losing newlines is a problem since they are used to terminate statements in Fortran. Losing spaces is a problem particularly in cases like ".true.", which is a single token in Fortran but three tokens in C.

Based on a fairly brief and superficial scan of the CDT preprocessor code,

modifying it to be able to retain spaces and newlines will likely require

1. modifying CDT's Token class to include preceding whitespace (or something like this),
2. modifying the Lexer class to track these tokens, and
3. modifying MacroDefinitionParser, ObjectStyleMacro, and MacroExpander to include whitespace in their output.

DOM (Heavyweight AST)

- **Remaining DOM nodes, semantic actions, and tests.** 376 semantic actions remain to be filled in, along with the corresponding DOM nodes, interfaces, prettyprinter methods, and tests. Again, proper testing will be critical.

- **Semantic Checks.** It should be noted that some amount of semantic analysis will likely be necessarily to have a "useful" Fortran front end. For example, whether F(3) indicates a function call or an array reference must be determined semantically, as must the body comprising a loop of the form "DO 10 I..." which references a statement label.

- **DOM AST View.** A separate Fortran DOM AST View was created because CDT's DOM AST view is not compatible with the Fortran DOM: There are many instanceof tests, the DOM is expected to have a C-language structure (e.g., no nested subroutines), and the search to populate this view uses offset/length information to determine parenting rather than the actual pointer structure of the tree. The problems discovered while attempting to populate the DOM AST View with a Fortran DOM may be predictive of more problems integrating a non-C/C++ DOM with existing CDT code.

Scoping, Binding Resolution, and Module Loading

- **Implement binding resolution.** As mentioned above, every identifier use is current assigned a FortranImplicitBinding (although this is incorrect); a full implementation obviously needs to resolve identifiers to the subprogram/variable/etc. that they actually reference in a manner conceptually similar to CVisitor (but specific to the Fortran DOM and Fortran's scoping rules).

Note that several constructs will need careful design consideration, including implicitly-declared variables, interface blocks, module uses, and common blocks. Photran's code for collecting and resolving bindings on its existing AST is contained in the core.vpg plug-in and follows a model similar to CDT's (i.e., certain AST nodes double as scopes and contain a collection of bindings); this code may be used for reference when implementing binding resolution on the DOM.

- **Module loading.** One unique and challenging aspect of binding resolution in Fortran is modules. Modules can be imported by name; moreover, only a subset of a module's contents can be imported, and imported entities can be renamed in the importing scope. In a compiler, it is the user's responsibility to ensure that files are compiled in a correct order based on these dependencies among modules; as each file is compiled, .mod files are generated, which store the contents of modules in a persistable form. In an IDE, the user generally would not be expected to provide such a list of dependencies; it would need to be computed automatically. One possibility is the following. As each file is indexed, modules are added to the PDOM by name (analogous to what a compiler would store in a .mod file). Binding resolution is done on demand; that is, when the indexer runs, only definitions are collected, and identifier uses are not resolved to their corresponding definitions until this is explicitly requested. When a binding resolution is requested (presumably after indexing has completed), USE statements can be processed and the required modules loaded from the PDOM to complete the binding resolution. (See the FindModuleAction class for the basic structure of code for locking and querying the PDOM.) Based on Photran's existing implementation and similar code in gfortran, this is likely to require 5000-8000 LOC.

PDOM

- **CDT does not parse Fortran linkage by default.** This was posted as a CDT bug (Eclipse Bugzilla 242607). Although there is a Fortran linkage declared in CDT, AbstractIndexerTask#runTask does not process it (one line must be added to do so). Without this change, Fortran files do not get indexed, and thus the Fortran PDOM is never populated.

Editor/UI Integration

- **Outine view does not appear to update.** This is due to the fact that OFP can only process Java File objects and therefore cannot handle the editor's (unsaved) buffer; see above.

- **CDT views and DOM/PDOM-based editor features need to be integrated.** We looked carefully at the possibility of subclassing the CEditor to get these features "for free," but that does not appear to be feasible at this time: The C editor is very much tied to C/C++ and would need significant changes to support languages substantially different from C/C++.

  From Chris Recoskie (regarding how UPC is supported in the editor):

  Syntax highlighting for UPC is a combination of CDT 5's semantic highlighting support and the keyword map; "partitions he more or less got for free as the new AST node types extend from existing ones" ... "so a upc_forall just looks like a for loop as far as most of CDT is concerned"

  From Mike Kucera:

  I agree, I don't think the C editor is ready to support Fortran. In some cases it doesn't even differentiate between C and C++. For example if you open a plain C file and type "class" then invoke content assist you will get template proposals for C++ classes. C partition types are used directly all over the place. It would probably be necessary to add extension points for a lot of things like the semantic highlighter.

Conclusions

Although it will require a significant amount of work, it appears feasible to implement a language infrastructure for Fortran based on the DOM, PDOM, and the Open Fortran Parser. Although this requires CDT internal classes, no significant changes to CDT are required.