C/C++ Source Code Introspection Using the CDT

Chris Recoskie, IBM CDT Team Lead
Beth Tibbitts, IBM Research

 March 7, 2007

Acknowledgements

- Thanks to Doug Schaefer for “loaning” me some of the material

What Is CDT?

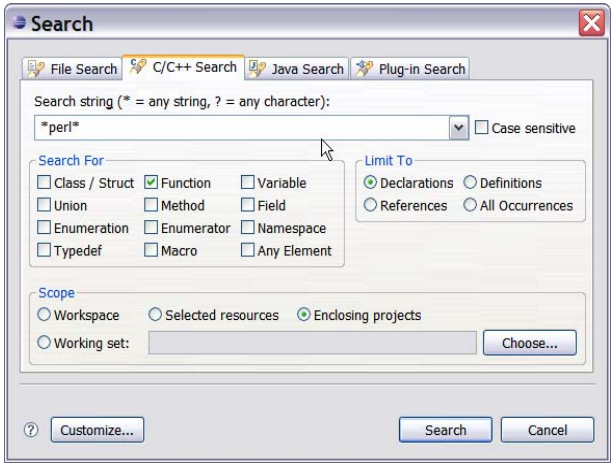
- Open source project hosted by Eclipse.org as a part of the Eclipse Tools top-level project (http://www.eclipse.org/cdt)
- Current stable release is 3.1.2, Europa-based release (CDT 4.0) due out June 30
- Unless otherwise specified, we will be focusing on CDT 3.1.2 in this tutorial. APIs may change in 4.0
- Two main purposes of CDT:
  - Provide a framework for C/C++ development tools integration into Eclipse IDE
    - Entire edit/build/debug cycle
  - Supplied implementations of APIs implement support for GNU toolchain
    - GNU make-based build systems
    - GCC/G++ compilers
    - GDB debugger

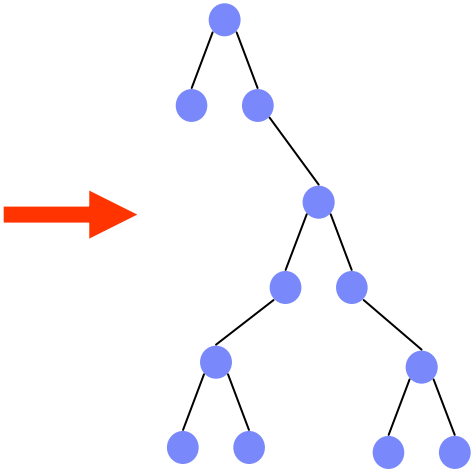
Introspection Components

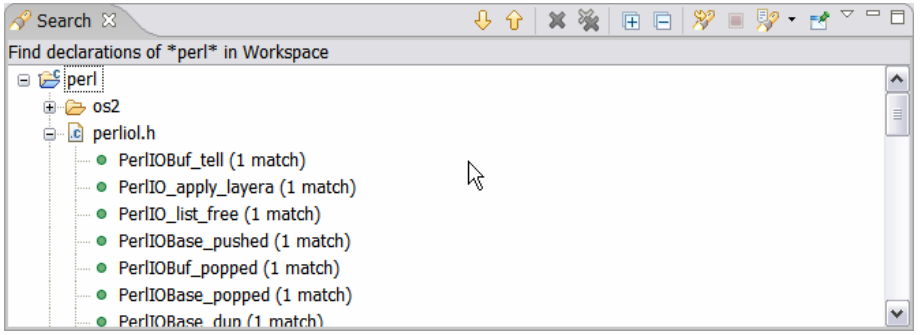
- Knowledge about the user's source code is stored in the CDT's DOM
  - Document Object Model
  - The name is a bit confusing ☺
- Two components
  - DOM AST
    - Abstract Syntax Tree that stores detailed structural information about the code
  - Index
    - Built from the AST
    - Provides the ability to perform fast lookups by name on elements
    - Persistent index called the PDOM (persistent DOM)

What is this information used for?

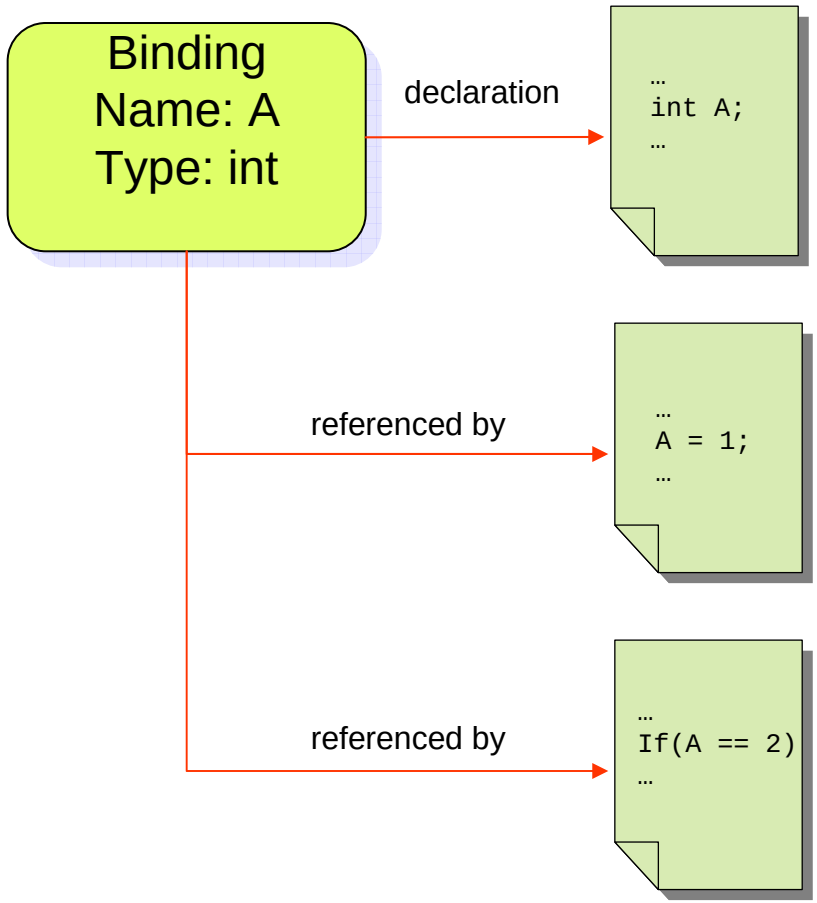
- Search
- Navigation
- Content Assist
- Call Hierarchy
- Type Hierarchy
- Include browsing
- Dependency scanning
- Syntax highlighting
- Refactoring







Data Components of DOM



- Bindings
  - Semantic link between names in AST
- Types
  - Type information for Bindings that have types
- Locations
  - Text locations of AST Nodes
  - Used for navigation

Actors in the DOM

- Scanner
  - Tokenizer for the parser
- Parser
  - Parses the output from the scanner and builds the AST
- AST Visitors
  - Visits nodes in the AST to mine data
- Indexer
  - Builds the index from the AST
- Index Visitors
  - Visit nodes in the index to mine data

AST

- Structure of AST generally follows grammar from the language spec
- Key elements – IASTTranslationUnit, IASTName
  - Minimal elements
  - IASTTranslationUnit is entry point (corresponds to a source file)
  - IASTName represents pathway to Bindings and Types
- IASTNode is root node type
  - Parent/Child hierarchy of AST
  - Map to Locations

AST Continued

- ASTVisitor
  - Walks the tree
  - You can go top down or bottom up
  - Separate visit() methods for common member types
- Common AST node types prefixed with IAST
- Languages extend the interfaces
  - ICPPAST, ICAST
  - Starting with IASTTranslationUnit

Locations

- Life in preprocessor land ☹
  - Nodes do not necessarily map directly to source due to macro expansions and conditional compilation
- Maintains a map of text expansions including includes, macros
- AST Nodes store the post preprocessing offset into the translation unit
- IASTNode.getNodeLocations() returns where the node is in the map
- IASTNode.getFileLocation() gives the file location where the text was generated
  - Will return header file of macro call

Bindings (IBinding)

- “Bind” references and declarations
  - Definition is C/C++ specialization of declaration
- IASTName models identifiers used in bindings
- IASTName.resolveBinding() to find Binding
  - Be careful not to do getBinding() before you do resolveBinding()!
- To find refs and decls for a binding, need context
  - IASTTranslationUnit
    - Translation unit wide, searches DOM
  - IPDOMResolver
    - Project wide, searches PDOM

Types (IType)

- Bindings often have types
  - E.g. variables have types
    - IVariable.getType()
- Types are not always Bindings
  - Sometimes they are
    - e.g. ICPPClassType
  - Sometimes they are not
    - e.g. IBasicType encompasses most built-in types such as int, char, etc.
- Expressions also have types
  - IASTExpression.getExpressionType()

Scanner (IScanner)

- Used by the GNU C and C++ parsers
- Tokenizes translation units
  - IToken IScanner.nextToken()
- Handles preprocessor on the fly
  - Creates the location map
  - *IASTPreprocessorFunctionStyleMacroDefinition*
- Built for speed
  - Minimize creation of String objects
- Not very reusable for other languages
  - Need to factor out preprocessor from tokenizer

Parsers (ISourceCodeParser)

- Produces AST for a given file
- One parser per language (ILanguage)
- In CDT 4.0 LanguageManager manages mappings of languages to individual files
  - User can override choice of language
- How to get AST?
  - ITranslationUnit.getLanguage()
  - ILanguage.getASTTranslationUnit()
- Also used to get an ASTCompletionNode for content assist
  - Returns all names that fit at a given offset in a given working copy

Code Reader Factory

- Provides flexible mechanism for finding include files
  - Hides details of resource management system
  - Does not hide details of include path
- Some modes provide caching of file buffers so that files only need to be read once.

GNU C and C++ Parsers

- Recursive Descent with backtracking, * lookahead
  - i.e. LL(*) – give or take
- Error recovery attempts to march ahead to next sane point in source and continue
- Does not use semantic information to resolve ambiguities
  - Stick “ambiguity nodes” in AST where this occurs
    - E.g. IASTAmbiguousStatement
  - Resolved at binding resolution time
  - Binding resolution slow, only resolved when someone cares

Language Extensibility

- Work is currently ongoing in 4.0 to provide a framework for easily adding other languages and language variants
- Using LPG parser generator to generate parsers based on grammars derived from the language spec
  - LALR(k) with backtracking to resolve ambiguities
  - C99 grammar exists now
  - C Preprocessor currently being written that will be used by all the new LPG-based parsers (possibly reusable by others?)
  - ISO C++ coming
- Grammars have actions that build the AST
- Grammars can be reused to create parsers for other C/C++ language variants
  - Inherit from our grammar, or cut & paste if you prefer

Index (IIndex, IPDOM)

- PDOM persists parts of AST, Bindings, and Types
- Binary flat file that is memory mapped (Database class)
  - Variable size records (16 byte blocks)
  - Divided into 16K chunks, mapped into memory separately
- Utility records to organize collections
  - Linked List
  - B-Tree
- IIndex abstracts away from PDOM details
  - Operates on IName

Indexers

- Full and Fast (or none)
  - Fast caches headers previously seen (not always 100% correct but probably good enough if you're not doing refactoring)
- Both use DOM as starting point
  - Walk the AST finding IASTNames and resolve their bindings
  - If not already, create PDOM object for Binding
  - If not already, create PDOM object file containing name
  - Create PDOM object for Name and hook up to Binding and File
- Re-index command available on project

Searching The Index

- Interfaces currently in IIndex
  - Use regular expressions in findBindings()
  - findDefinitions()
  - findReferences()
  - findIncludedBy()
  - findIncludes()
- Make sure you acquireReadLock() first to avoid synchronization issues with other index clients (such as the indexer)
- Make sure you release the lock on all paths through your code!

Example: Open Declaration

```
IIndex index=
CCorePlugin.getIndexManager().getIndex(workingCopy.g
etCProject(),
IIndexManager.ADD_DEPENDENCIES |
IIndexManager.ADD_DEPENDENT);
try {
index.acquireReadLock();
} catch (InterruptedException e1) {
return Status.CANCEL_STATUS;
}
try {
IASTTranslationUnit ast = workingCopy.getAST(index,
ITranslationUnit.AST_SKIP_ALL_HEADERS);
IASTName[] selectedNames =
workingCopy.getLanguage().getSelectedNames(ast,
selectionStart, selectionLength);

if (selectedNames.length > 0 && selectedNames[0] !=
null) { // just right, only one name selected
IASTName searchName = selectedNames[0];

searchName.resolveBinding();
if (binding != null) {
final IName[] declNames =
ast.getDefinitions(binding);
for (int i = 0; i < declNames.length; i++) {

IBinding binding = IASTFileLocation fileloc =
declNames[i].getFileLocation();

// no source location - TODO spit out an error in
the status bar
    if (fileloc != null) {
    final IPath path = new
Path(fileloc.getFileName());
    final int offset = fileloc.getNodeOffset();
    final int length = fileloc.getNodeLength();

    runInUIThread(new Runnable() {
    public void run() {
    try {
    open(path, offset, length);
    } catch (CoreException e) {
    CUIPlugin.getDefault().log(e);
    }
    }
    });
    break;
    }
}
}
}
}
finally {
index.releaseReadLock();
}
```

Example: Call Hierarchy

```
private static void findCalledBy(IIndex index, IBinding
  callee, ICProject project, CalledByResult result) throws
  CoreException, DOMException {
    if (callee != null) {
        IIndexName[] names= index.findReferences(callee);
        for (int i = 0; i < names.length; i++) {
            IIndexName rname = names[i];
            IIndexName caller= rname.getEnclosingDefinition();
            if (caller != null) {
                ICElement elem= IndexUI.getCElementForName(project,
                  index, caller);
                if (elem != null) {
                  result.add(elem, rname);
                }
            }
        }
    }
}
```

CDT static analysis examples

- PTP: Parallel Tools Platform (http://eclipse.org/ptp)
  - PLDT: Parallel Language Development Tools:
    - PLDT is a subset of PTP, and can stand alone (with CDT)
    - Static analysis of MPI and OpenMP code in C/C++/Fortran: general framework extensible for other parallel tools and languages
- Some things PTP 1.1 (PLDT) does with CDT:
  1. Walk AST with visitor to find artifacts (e.g. MPI)
  2. Visitor inspects information at each AST node (e.g. is it an MPI function?)
  3. MPI barrier analysis: walk ast (new bottom-up); build call graph, control flow graph, data dependency graph; analyze to find potential deadlocks (* Not yet available in PTP 1.1)
  4. OpenMP Common problems: semantic analysis to find
  5. OpenMP Concurrency analysis
- For each item above, we show the UI that uses the info, then the code that generates the information (code uses CDT 3.1.2 APIs)

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

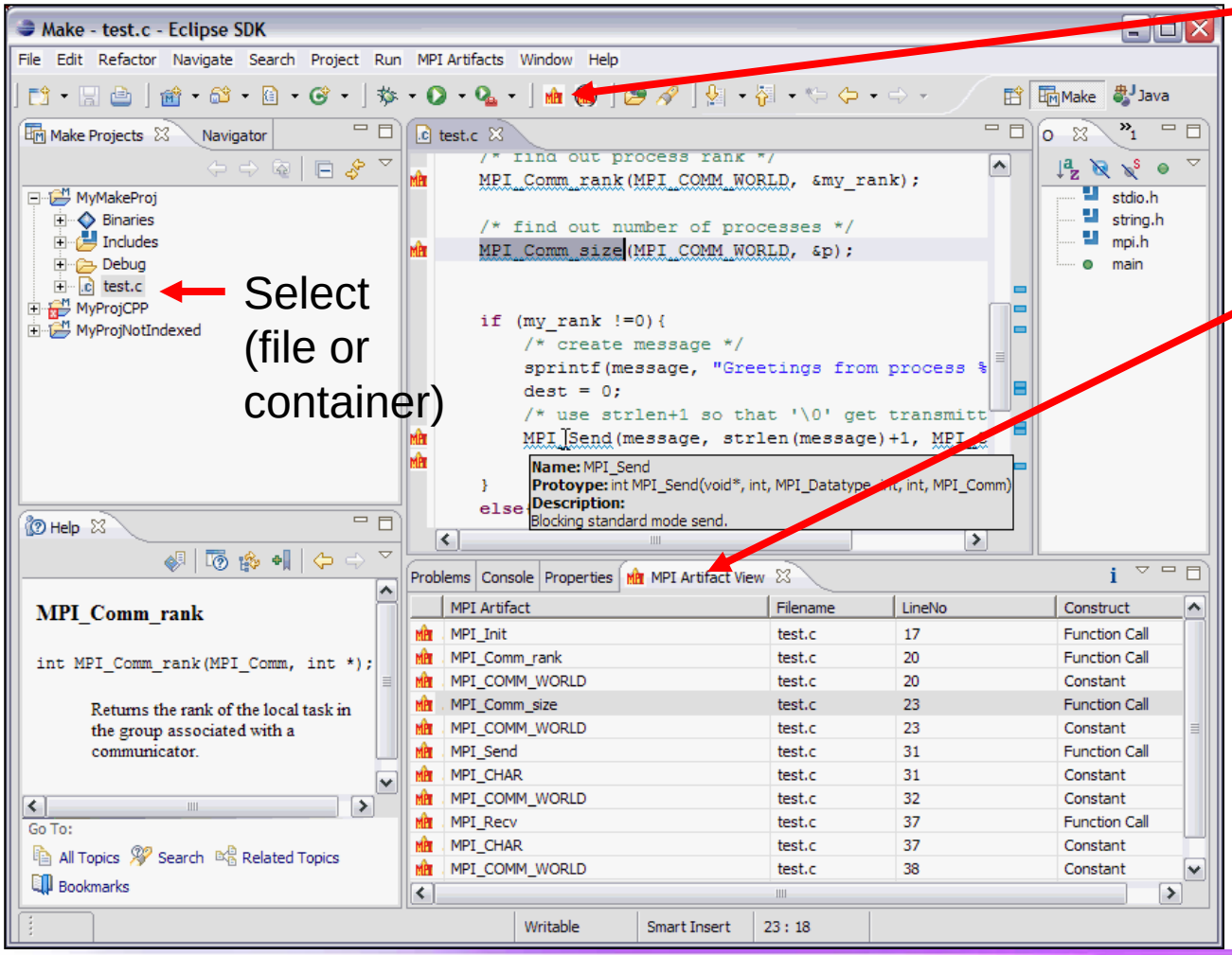
Parallel Programming Languages

Two languages supported by PLDT

- MPI – Messing Passing Interface
  - Distributed Memory / Clusters
- OpenMP
  - Shared Memory / Multi-Core

PTP (PLDT) Syntactic and Static Analysis with CDT

1. Walk AST with visitor to find artifacts



PTP PLDT example: Kick it off via this action

This view is populated from source files

1. Walk AST with visitor to find *artifacts**

- Walk resource tree: For each file (C source code), do this:

```
// walking the AST Tree
ITranslationUnit itu=(ITranslationUnit)selection;
IASTTranslationUnit tu =
   tu.getLanguage().getASTTranslationUnit(itu, 0);
tu.accept(new MpiCASTVisitor(file,…));
// visitor gets called at each node
```

* Artifact information is saved in markers, shown in view

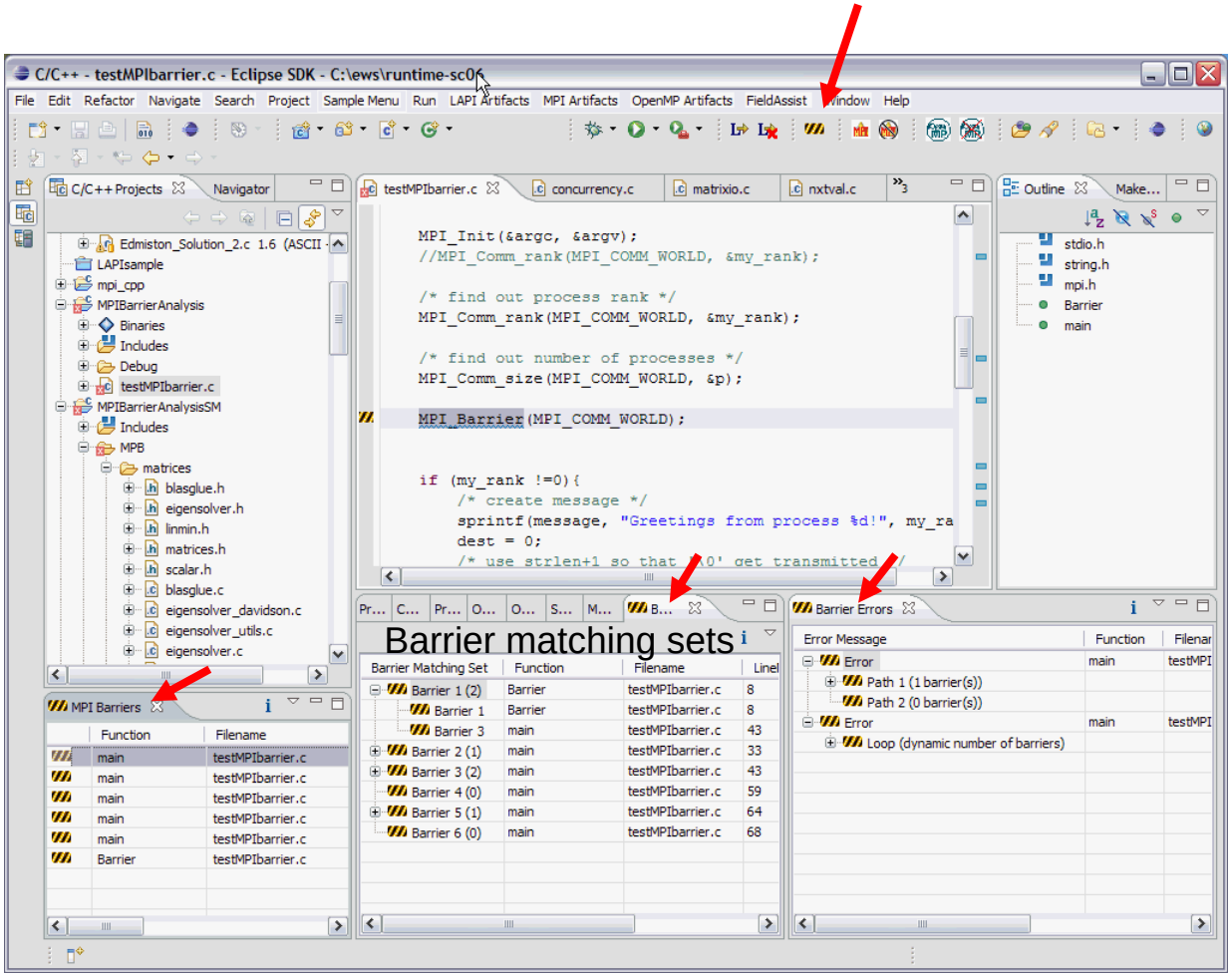
2. At AST nodes visitor is called during tree walk:

```
public class MpiCASTVisitor ( …

public int visit(IASTExpression expression) {
  // if it's a function call
  if (expression instanceof IASTFunctionCallExpression) {
    IASTFunctionCallExpression fce = (IASTFunctionCallExpression)
                                                        expression;
    IASTExpression astExpr = fce.getFunctionNameExpression();
    // raw signature ok if no macro changes it
    String fnSig = astExpr.getRawSignature();

    if (astExpr instanceof IASTIdExpression) {
      IASTName fnName = ((IASTIdExpression) astExpr).getName();
      IBinding binding = fnName.resolveBinding();
      String tempNAME = binding.getName();
      boolean preProcUsed = !rawSig.equals(tempNAME);
      if (preProcUsed) {
         fnSig = tempNAME;
      }
      if (fnSig.startsWith(PREFIX)) {  //PREFIX is "MPI_"
        IASTName funcName = ((IASTIdExpression) astExpr).getName();
        // decide if we add this name to our view
        processFuncName(funcName, astExpr); // cache artifact in
                                                  marker here;
                    this populates view via resource chg event}
    }
  }
  return PROCESS_CONTINUE;
}
```

PTP (PLDT) Syntactic and Static Analysis with CDT

3. MPI barrier analysis:

not yet available in PTP 1.1



Analyses MPI C source code to find potential deadlocks

3. MPI barrier analysis: walk ast (new bottom-up); graphs we build (call graph, control flow graph, data flow dependence graph)

```
public int visit(IASTDeclaration decl)
{
  String filename =
   declaration.getContainingFilename()
  if (decl instanceof IASTFunctionDefinition) {
    depth ++;
    IASTFunctionDefinition fd =
   (IASTFunctionDefinition)declaration;
    ICallGraphNode node =
            new CallGraphNode(file_, filename, fd);
    CG_.addNode(node);
    return PROCESS_SKIP;
  }
  else if (decl instanceof IASTSimpleDeclaration){
    if(depth > 0) return PROCESS_SKIP; //not global
    IASTSimpleDeclaration sdecl =
   (IASTSimpleDeclaration)declaration;
      /* if the declarator is null,
           then it is a structure specifier*/
    if(sdecl.getDeclarators() == null)
          return PROCESS_CONTINUE;
    IASTDeclSpecifier spec = sdecl.getDeclSpecifier();
    if(spec instanceof IASTCompositeTypeSpecifier ||
       spec instanceof IASTElaboratedTypeSpecifier ||
       spec instanceof IASTEnumerationSpecifier)
      return PROCESS_SKIP;
```

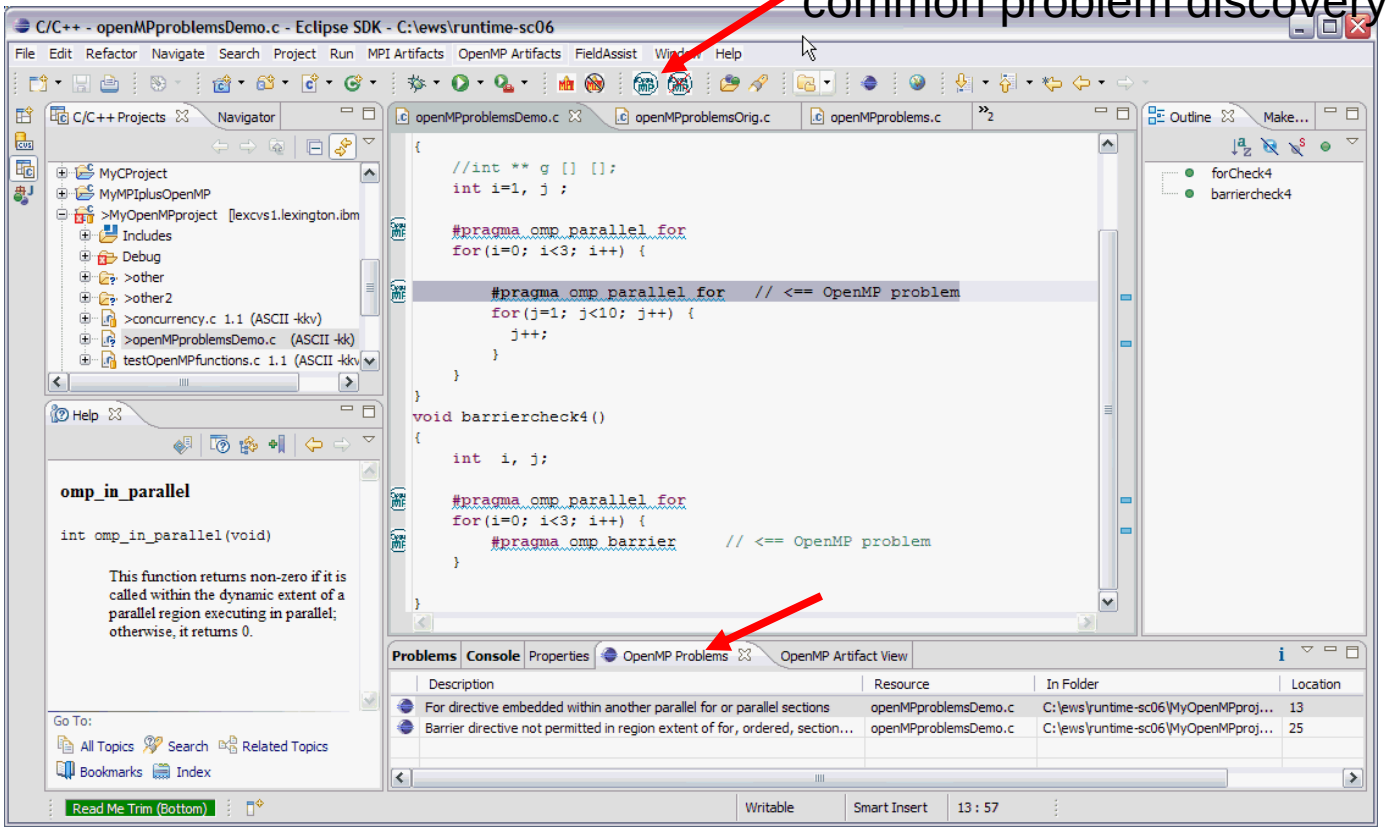
```
// collect global variable declarations
    List<String> env = CG_.getEnv();
    IASTDeclarator[] declarators =
sdecl.getDeclarators();
    for(int j=0; j<declarators.length; j++){
      if(declarators[j] instanceof
IASTFunctionDeclarator)
        continue;
      IASTName n = declarators[j].getName();
      String var = n.toString();
      if(!env.contains(var))
        env.add(var); // add global variable
    }
  }
  return PROCESS_CONTINUE;
}
```

leave() enables bottom-up traversal (CDT 3.1.2)

```
public int leave(IASTDeclaration declaration)
{
  if (declaration instanceof
           IASTFunctionDefinition) {
        depth --;
        return PROCESS_SKIP;
        }
   return PROCESS_CONTINUE;
}
```

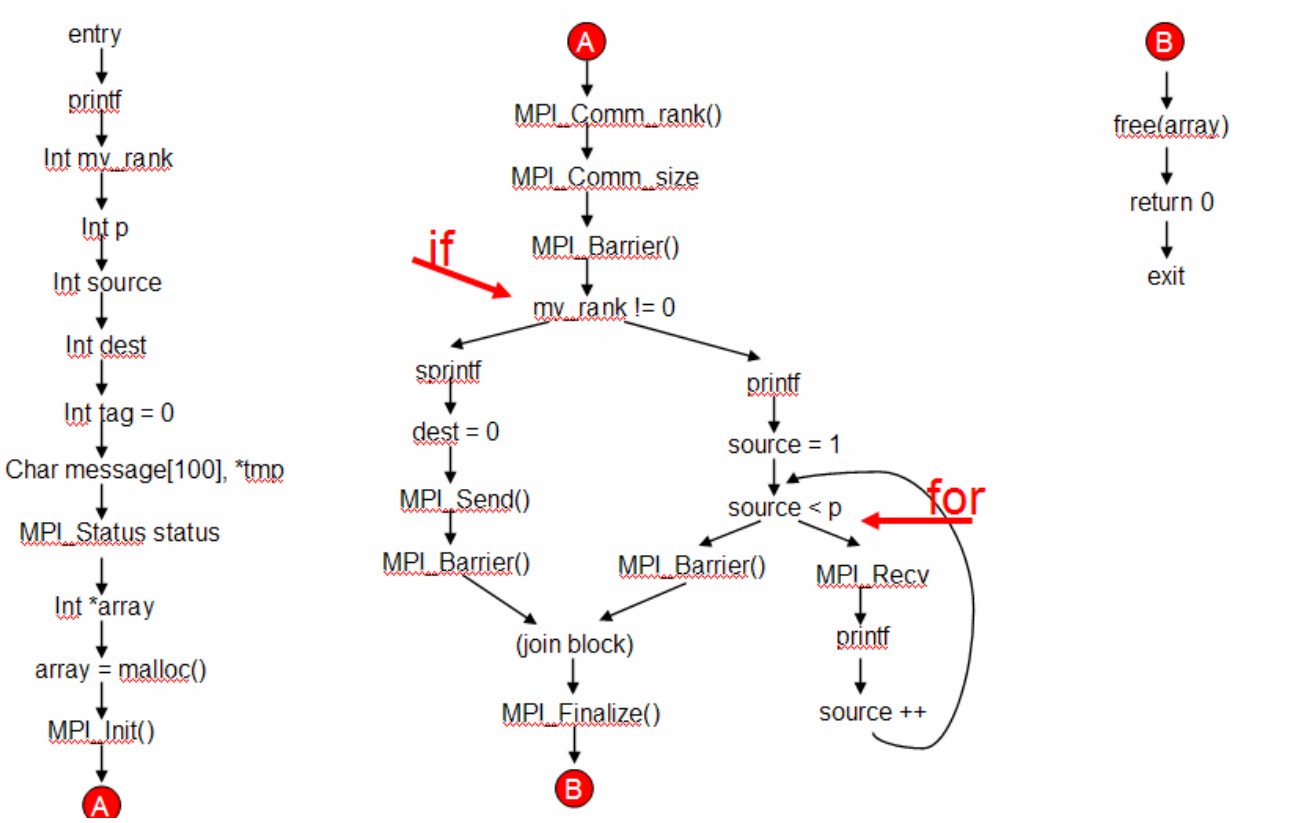
5. OpenMP common problems

OpenMP analysis includes common problem discovery



- We will construct a control flow graph

Control Flow Graph



5. OpenMP common problems: how/code

- During artifact analysis, a Control Flow Graph is constructed.
- ASTNodes for #pragmas are created.
- After analysis is complete, pragma nodes are inspected to make sure they don't try to implement disallowed features
- For example: don't allow a #pragma parallel for within another.

```
package org.eclipse.ptp.pldt.openmp.analysis…
class PASTSemanticCheck { …
private void forCheck(OMPPragmaNode pnode)
{
    PASTOMPPragma pragma = pnode.getPragma();
    OMPPragmaNode parent = pnode.getParent();//cfg
    boolean       errorFound=false;
    while(parent!=null) {
        int type = parent.getPragma().getType();
           if (type==PASTOMPPragma.ParallelFor){
               handleProblem(pnode.getPragma(),
     "For directive embedded within another
        parallel for or parallel sections",
                        OpenMPError.ERROR);
                   errorFound=true;  break;
           }
        parent = parent.getContextPredecessor();
    }
    return;
}
```

Links

- CDT homepage
  - http://www.eclipse.org/cdt
- CDT newsgroup
  - eclipse.tools.cdt on new news.eclipse.org
- EclipseCon C Development Track
  - http://www.eclipsecon.org/2007/index.php?page=sub/&area=c-devel
- Parallel Tools Platform homepage
  - http://www.eclipse.org/ptp
- Shameless plug
  - Go see Beth's short talk "Developing Parallel Programs - PTP's PLDT" at 2:00 PM in Room 210