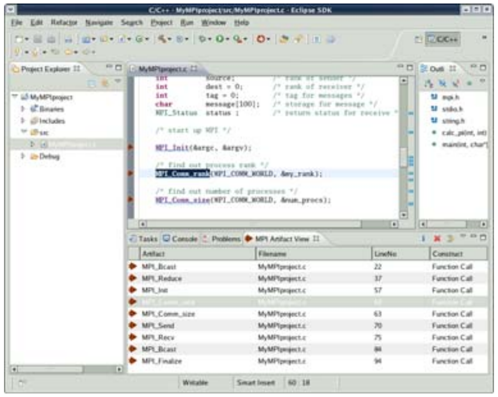
eclipseCON™ 2008



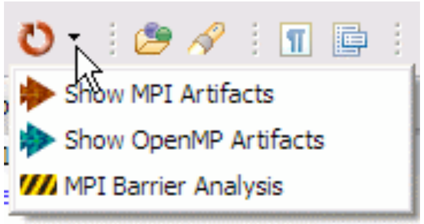
Static Analysis in PTP with CDT

Parallel Tools Platform eclipse.org/ptp

C/C++ Development Tools eclipse.org/cdt

What can I find out about my C/C++ program?

How do I do it? Why is it useful?



Beth R. Tibbitts IBM Research

tibbitts@us.ibm.com

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

Outline

- Basics of static analysis
- What CDT provides today:
  - AST: how to inspect it; how to walk it
- Additional info built by PTP/PLDT for analysis
  - Call graph (incl recursion)
  - Control flow graph
  - Data dependency (partial)
- Upcoming features
  - Refactoring & potential in CDT 5.0
  - Using external info for analysis: e.g. compiler info
  - Source Code instrumentation

PLDT = Parallel Language Development Tools: "the analysis part of PTP"

What is static analysis?

- **Static code analysis** is analysis of a computer program that is performed without actual execution - analysis performed on executing programs is known as dynamic analysis.
  - Usually performed on some intermediate representations of the source code.
  - Routinely done by compilers in order *to generate and optimize* object code
- Motivation:
  - Deriving properties of execution behavior or program structure
  - Various forms of analysis and refactoring
  - Lots more in JDT :)

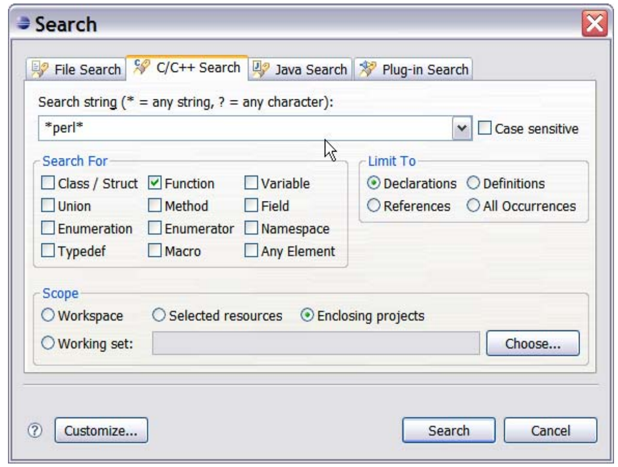
CDT Introspection Components

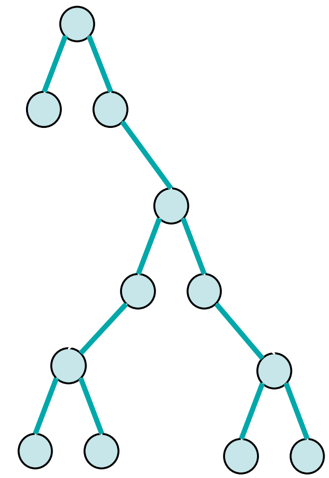
- Knowledge about the user's source code is stored in the CDT's DOM: Document Object Model
- Two components of DOM
  - DOM AST concentrate here
    - Abstract Syntax Tree that stores detailed structural information about the code
  - Index
    - Built from the AST
    - Provides the ability to perform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

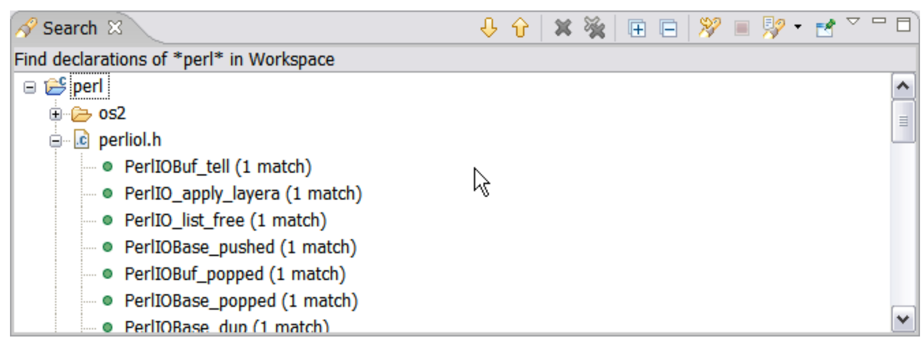
Ref: EclipseCon 2007, "C/C++ Source Code Introspection Using the CDT", Recoskie & Tibbitts

What is this information used for in CDT?

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

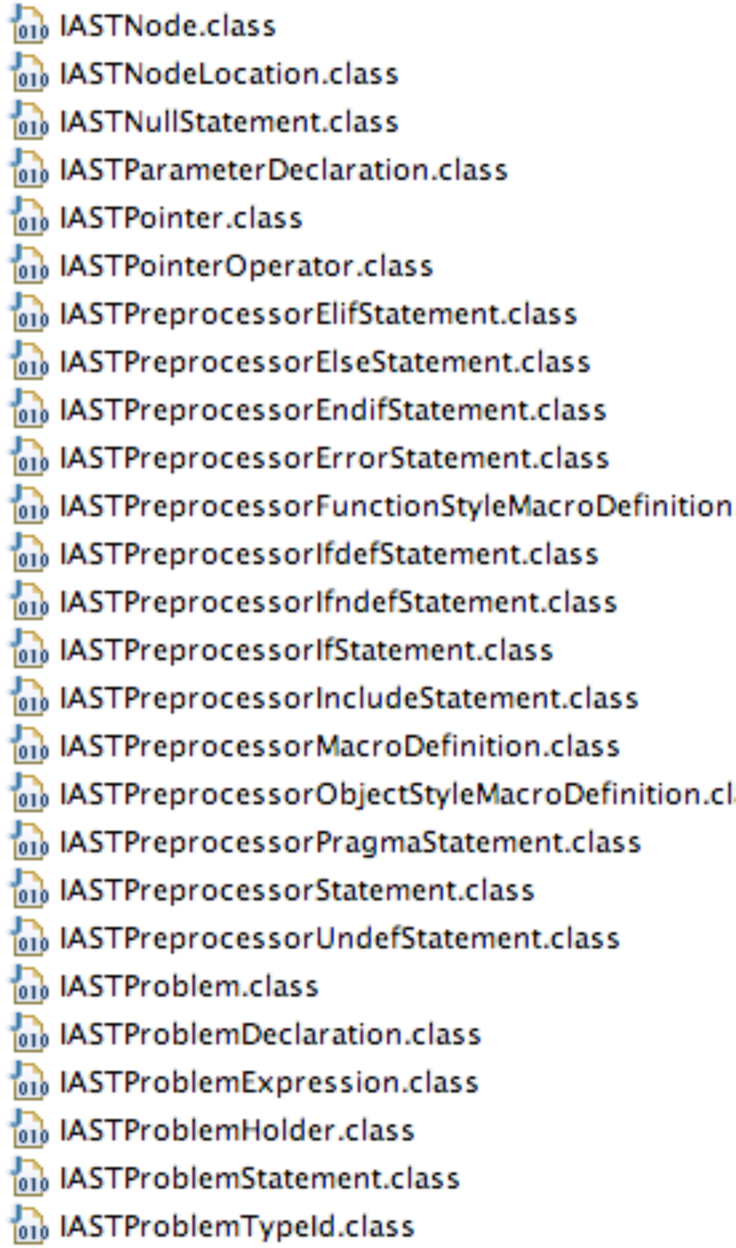






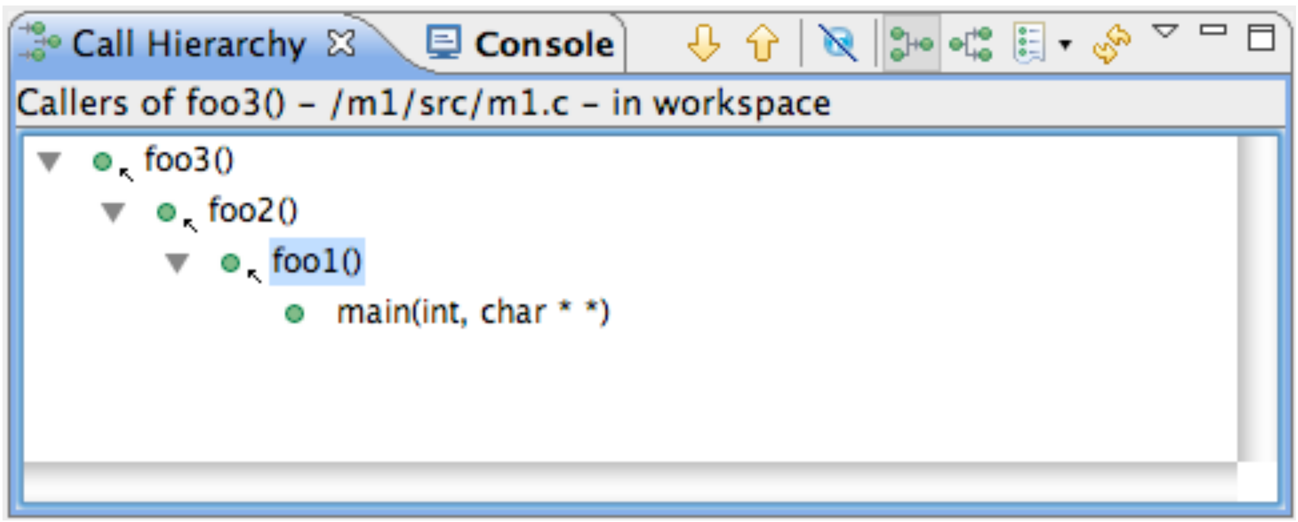
Abstract Syntax Tree: AST

- Maps C/C++ source code info onto a tree structure
  - A tree of nodes, all subclasses of
    - org.eclipse.cdt.core.dom.ast.IASTNode
  - Nodes for: functions, names, declarations, arrays, expressions, statements/compound statements, etc.
  - Src file root: IASTTranslationUnit
    - Correlates to a source file: myfile.c
  - Tree structure eases analysis
  - Knows relationships (parent/child)
  - Easy traversal (ASTVisitor)
  - … etc



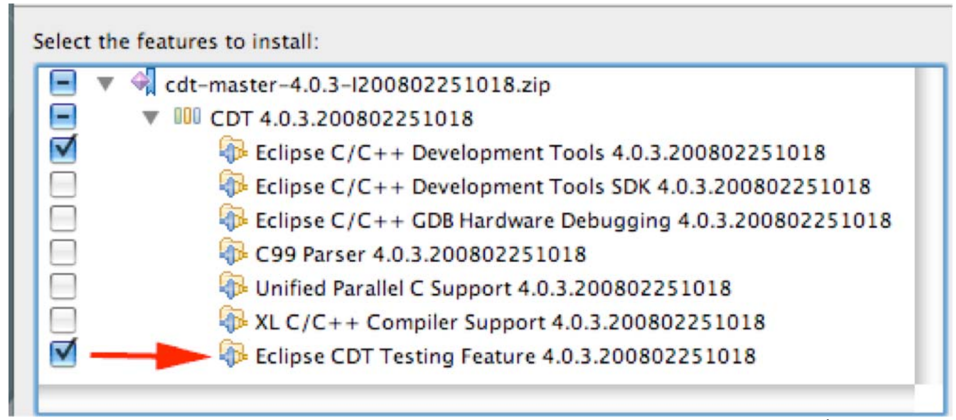
Existing CDT views that use structure include….

CDT Call Hierarchy view

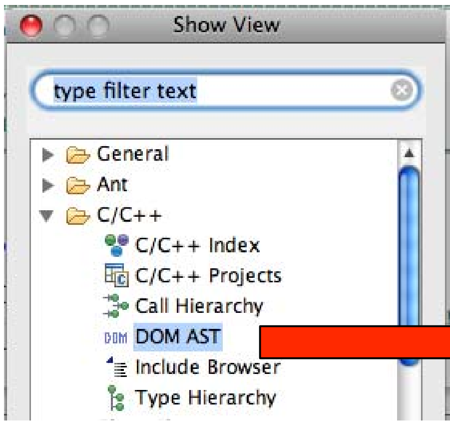


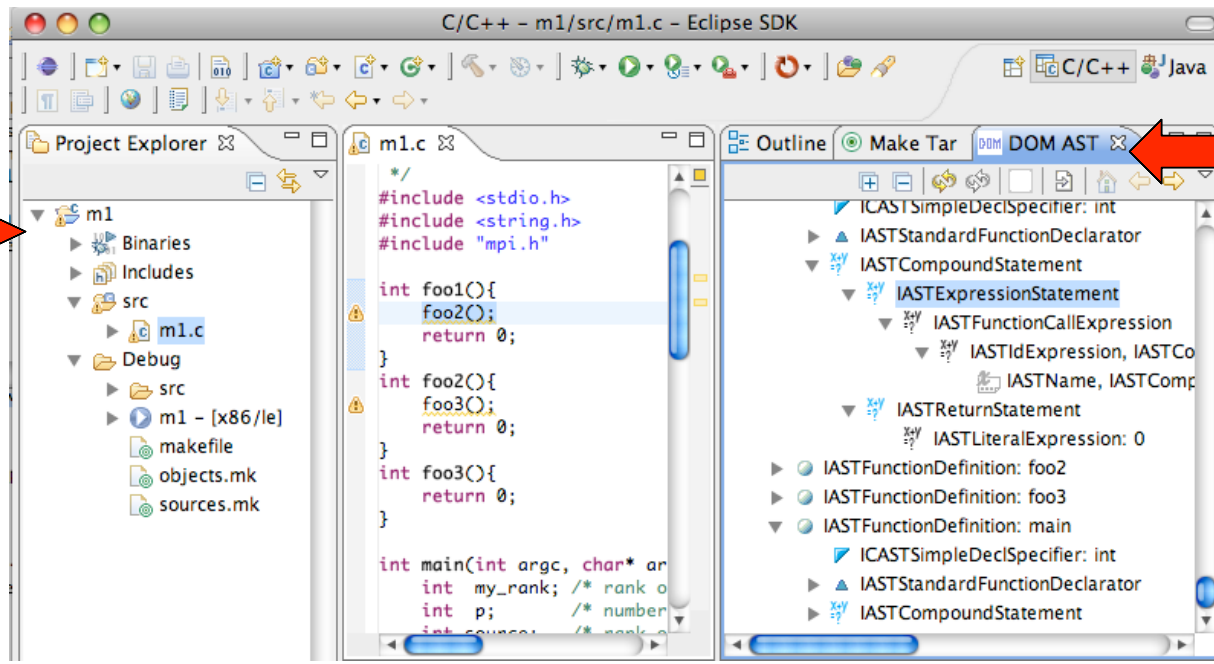
CDT DOM AST View

- Graphical inspection of AST



Available in CDT Testing feature:
org.eclipse.cdt.ui.tests package





Creating an AST : steps to create and use

1. Get ITranslationUnit - from e.g. source file “foo.c”
2. From ITranslationUnit, Show some “flat info” - lists preprocessor stmts, declarations, include dependencies
3. Walk the ITranslationUnit’s tree: CElements
4. Create AST and walk it

Code for tree walking is in sample plugin

On dev.eclipse.org: org.eclipse.ptp/tools/samples/

Project org.eclipse.ptp.pldt.sampleCDTstaticAnalysis

Walk AST

Show Call Graph

Show Control Flow Graph

Creating an AST (1): get ITranslationUnit

- From a source file in Projects view - Example: Plugin with an Action which gets current selection:

```
public void runSelectionExample(ISelection selection) {
  if(selection instanceof IStructuredSelection) {
    IStructuredSelection ss=(IStructuredSelection)selection;
    for (Iterator iter = ss.iterator(); iter.hasNext();) {
      Object obj = (Object) iter.next();
      // It can be a Project, Folder, File, etc...
      if (obj instanceof IAdaptable) {
      IAdaptable iad=(IAdaptable)obj;
      IResource res = (IResource) iad.getAdapter(IResource.class);
      System.out.println("     got resource: " + res);

      // ICElement covers folders and translation units
      ICElement ce = (ICElement) iad.getAdapter(ICElement.class);
      System.out.println("     got ICElement: "+ce);

      ITranslationUnit tu =
              (ITranslationUnit)iad.getAdapter(ITranslationUnit.class);
      System.out.println("     got ITranslationUnit: "+tu);
      listFlatInfo(tu);
      walkITU(tu);
}}}}
```

Creating an AST: (2) get AST and listFlatInfo()

```
void listFlatInfo(ITranslationUnit tu) throws CoreException {
 IASTTranslationUnit ast = tu.getAST();

 System.out.println("AST for: "+ast.getContainingFilename());
 IASTPreprocessorStatement[]ppss= ast.getAllPreprocessorStatements();
  System.out.println("PreprocessorStmts: (omit /usr/...)");
  for (int i = 0; i < ppss.length; i++) {
    IASTPreprocessorStatement pps = ppss[i];
    String fn = pps.getContainingFilename();
    if(!fn.startsWith("/usr")) System.out.println(i+"  PreprocessorStmt: "+fn+" "+pps.getRawSignature());
  }
  IASTDeclaration[] decls = ast.getDeclarations();
  System.out.println("Declarations: (omit /usr/...)");
  for (int i = 0; i < decls.length; i++) {
      IASTDeclaration decl = decls[i];
      String fn = decl.getContainingFilename();
      if(!fn.startsWith("/usr")) System.out.println(i+"  Declaration: "+fn+" "+decl.getRawSignature());
  }
  IDependencyTree dt=ast.getDependencyTree();
  IASTInclusionNode[] ins = dt.getInclusions();
  System.out.println("Include statements:");
  for (int i = 0; i < ins.length; i++) {
    IASTInclusionNode in = ins[i];
    IASTPreprocessorIncludeStatement is = in.getIncludeDirective();
    System.out.println(i+"  include stmt: "+is);
  }
}
```

```
ast.
getAllPreprocessorStatements();
```

```
ast.getDeclarations();
```

```
ast.getDependencyTree();//includes
```

Creating an AST: (2) show “flat info”

- Source and Results: flat info

```
ws = complete path to workspace
```

```
//walkast.c
#include <stdio.h>
#define MYVAR 42

int main(void) {
  int a,b;
  a=0;
  b=MYVAR; // use defined
  b = b + a;
  return b;
}
int foo(int bar){
  int z = bar;
  return z;
}
```

```
SampleAction.selectionChanged()
     got resource: L/Hello/src/Hello.c
     got ICElement: Hello.c
     got ITranslationUnit: Hello.c
AST for: ws/Hello/src/Hello.c
PreprocessorStmts: (omit /usr/...)
0  PreprocessorStmt: ws/Hello/src/Hello.c #include <stdio.h>
357  PreprocessorStmt: ws/Hello/src/Hello.c #define MYVAR 42
Declarations: (omit /usr/...)
154  Declaration: ws/Hello/src/Hello.c int main(void) ….
155  Declaration: ws/Hello/src/Hello.c int foo(int bar){
int z = bar;
return z;
}
Include statements:
0  include stmt: /usr/include/stdio.h
```

Demo: this plus DOMAST

AST: (3) walkITU: Walk ICElement tree

```
private void walkITU(ITranslationUnit tu) {
  String tuName = tu.getElementName();
  System.out.println("ITranslationUnit name: " + tuName);
  tu.accept(new ICElementVisitor() {
    public boolean visit(ICElement element) {
      boolean visitChildren = true;
        System.out.println("Visiting: " +
             element.getElementName());
      return visitChildren;
    }
   });
}
```

Visitor pattern:

Your *Visitor* code gets called *at each node*

tree.accept(visitor)

Exception handling omitted for brevity

Watch the walking: walkITU - ICElement Visitor

- Source and Results:
  walkITU()

```c
#include <stdio.h>
#define MYVAR 42

int main(void) {
  int a,b;
  a=0;
  b=MYVAR; // use defined
  b = b + a;
  return b;
}
int foo(int bar){
  int z = bar;
  return z;
}
```

```
ITranslationUnit name: Hello.c
Visiting: Hello.c
Visiting: stdio.h
Visiting: MYVAR
Visiting: main
Visiting: foo
```

AST: (4) Walk AST nodes : walkITU_AST()

```
private void walkITU_AST(ITranslationUnit tu) throws CoreException {
   System.out.println("AST visitor for "+tu.getElementName());
   IASTTranslationUnit ast = tu.getAST();
   ast.accept(new MyASTVisitor());
}
class MyASTVisitor extends ASTVisitor{
   MyASTVisitor(){
         this.shouldVisitStatements=true; // lots more
         this.shouldVisitDeclarations=true;
   }
   public int visit(IASTStatement stmt) { // lots more
         System.out.println("Visiting stmt: "+stmt.getRawSignature());
         return PROCESS_CONTINUE;
   }
   public int visit(IASTDeclaration decl) {
         System.out.println("Visiting decl: "+decl.getRawSignature());
         return PROCESS_CONTINUE;
   }
}
```

Visitor pattern:

Your *Visitor* code gets called *at each node*

tree.accept(visitor)

Note: simple visitor only visits statements and declarations

eclipseCON™ 2008

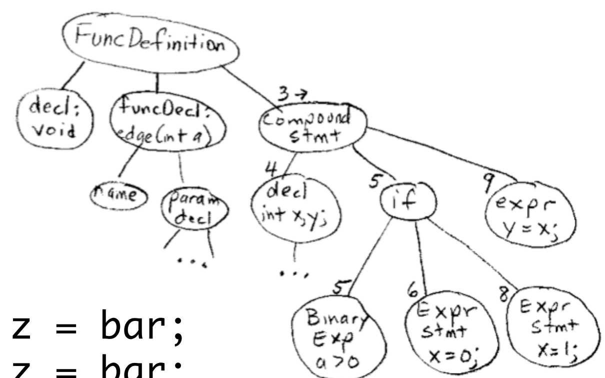
Watch AST walking: walkITU_AST()

- Source and Results:

```
// walkast_edge.c
1 #include <stdio.h>
2
3  void edge(int a) {
4    int x,y;
5    if(a>0)
6      x=0;
7    else
8      x=1;
9    y=x;
10  }
11  int foo(int bar){
12    int z = bar;
13    return z;
14  }
```

Note new example

```
AST visitor for walkast_edge.c
..Omit included stuff …
AST visitor for walkast_edge.c
Visiting decl: void edge(int a) {…}      3
Visiting stmt: { int x,y … y=x;}         3
Visiting stmt: int x,y;                  4
Visiting decl: int x,y;
Visiting stmt: if(a>0)  x=0;  else x=1;  5
Visiting stmt: x=0;                      5
Visiting stmt: x=1;                      8
Visiting stmt: y=x;                      9
Visiting decl: int foo(int bar){
  int z = bar;
  return z;
}
Visiting stmt: {
  int z = bar;
  return z;
}
Visiting stmt: int z = bar;
Visiting decl: int z = bar;
Visiting stmt: return z;
```



MyASTVisitor2

- More details in the visit() methods….
- Implements leave() methods too… showing depth
- See tree on next slide

Adds:

1. New type of construct visited (IASTName)
2. leave() methods as well as visit() methods

```
visit(IASTName stmt){…}
leave(IASTName stmt){…}
```

AST: More complex Visitor exposes more AST

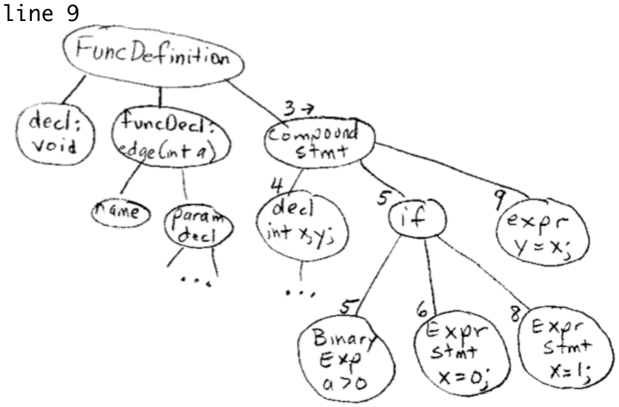
`leave()` indicates nesting

```
// walkast_edge.c
#include <stdio.h>

3 void edge(int a) {
4   int x,y;
5   if(a>0)
6     x=0;
7   else
8     x=1;
9   y=x;
  }
  int foo(int bar){
    int z = bar;
    return z;
  }
```

```
Visiting stmt: if(a>0)          line 5
                   x=0;
                 else
                   x=1;
Visiting name: a
 Leaving name: a
Visiting stmt: x=0;              line 6
    Visiting name: x
     Leaving name: x
     Leaving stmt: x=0;
      Visiting stmt: x=1;        line 8
      Visiting name: x
       Leaving name: x
       Leaving stmt: x=1;
Leaving stmt: if(a>0)
                x=0;
                else
                x=1;
Visiting stmt: y=x;                line 9
  Visiting name: y
   Leaving name: y
  Visiting name: x
   Leaving name: x
   Leaving stmt: y=x;
 Leaving stmt: {
    int x,y;
    if(a>0)
      x=0;
    else
      x=1;
    y=x;
    }
```

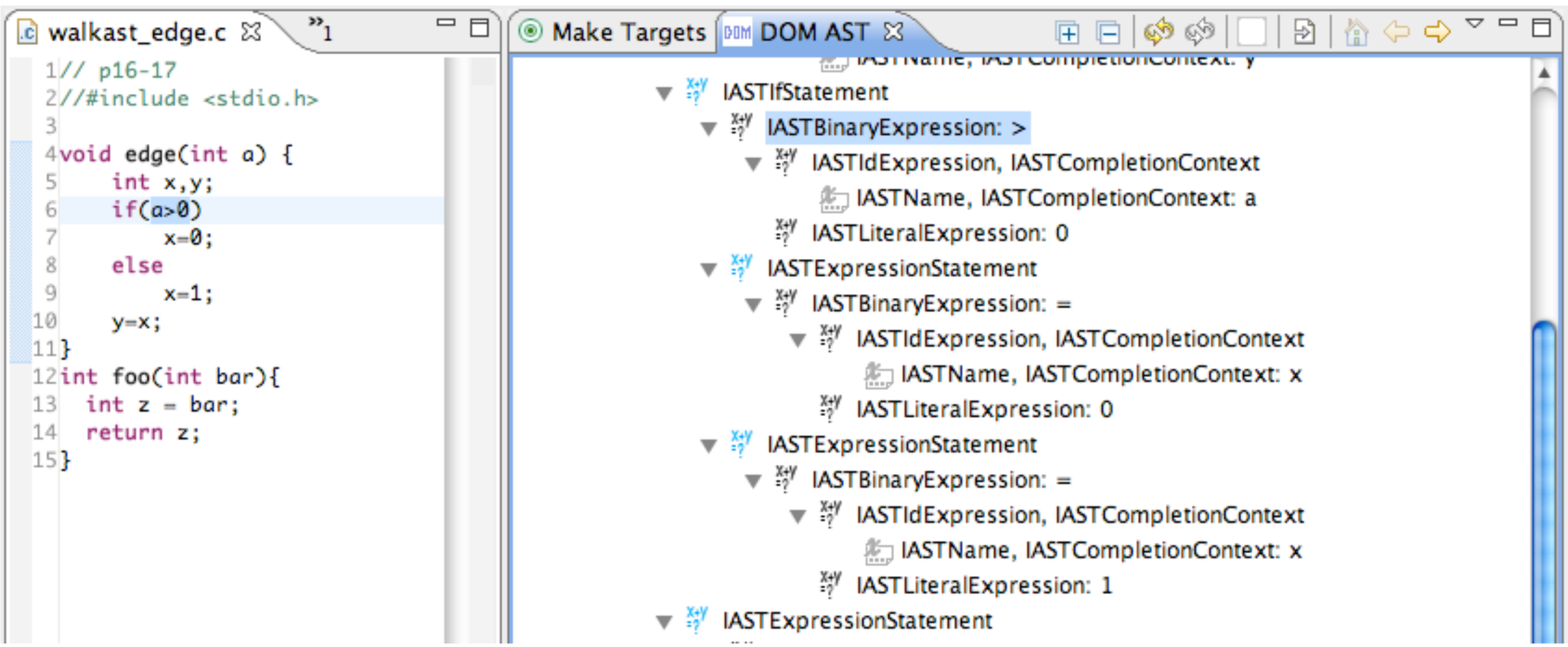
visit if stmt

leave if stmt



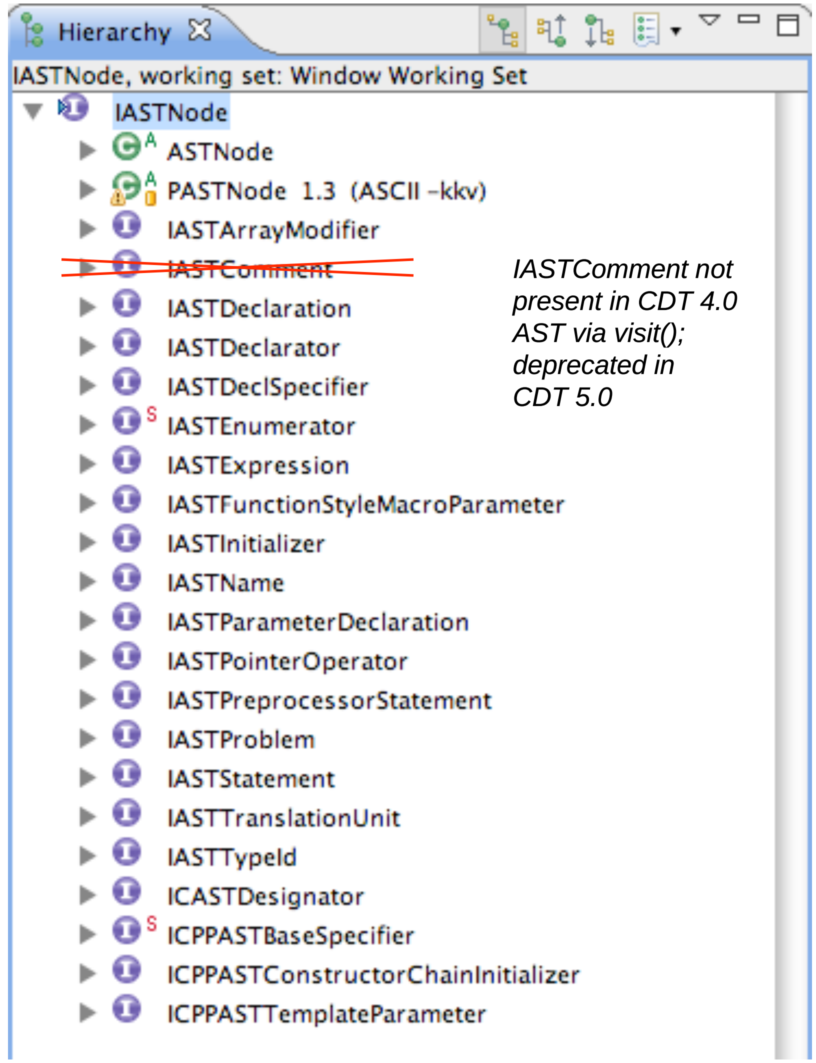
Relook at DOM AST View: see depth parsed

- Relook at DOM AST View: see “descending into structure”

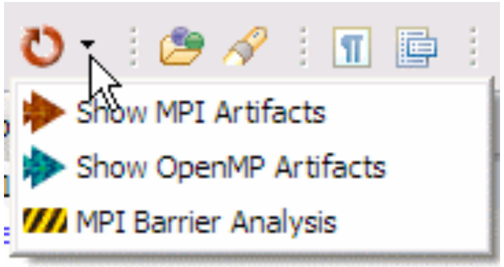


IASTNode hierarchy

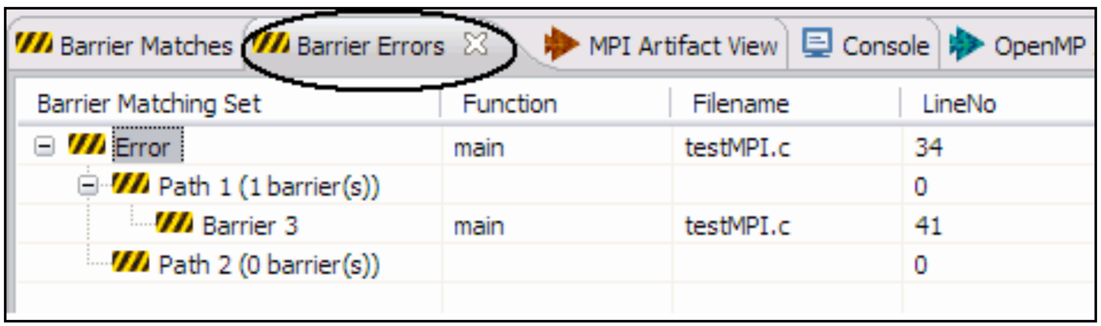
- Classes represent various language constructs

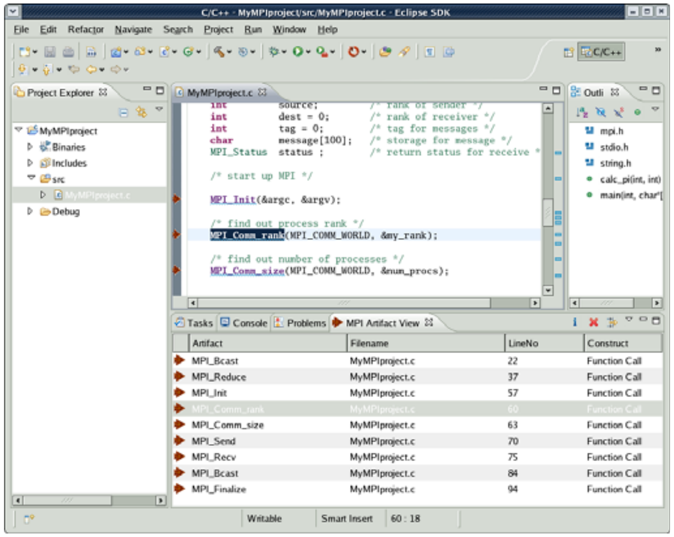


AST: what we do with it
PTP/PLDT provided structures



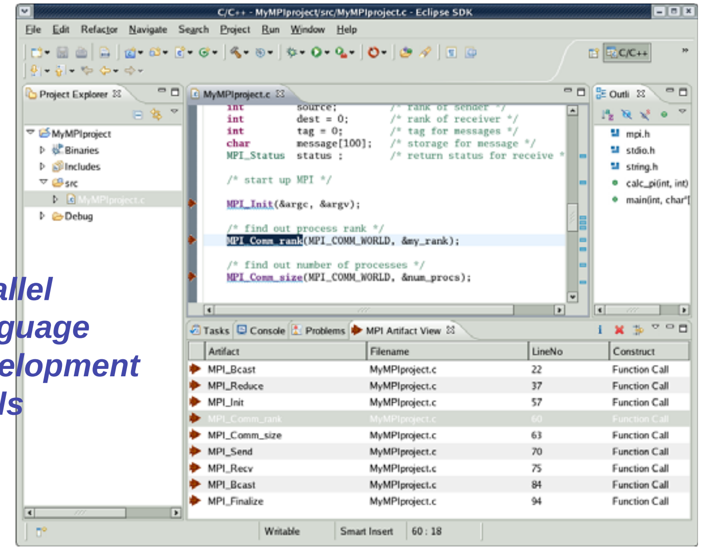
- Find Location of: MPI, OpenMP artifacts; MPI Barriers with AST walking
- Analysis: MPI barrier deadlock detection, OpenMP concurrency





eclipseCON™ 2008

PLDT’s AST walking



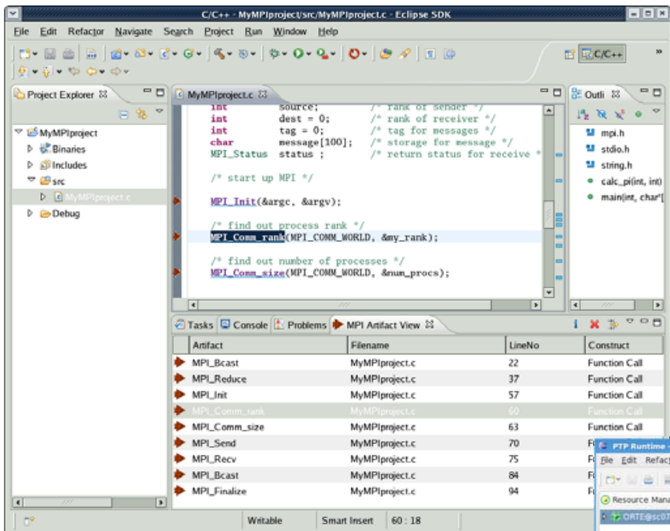
Parallel Language Development Tools

- Location of “MPI Artifacts”
- Not a simple text location
- During tree walking, expressions are located for function calls, and tested for viability:

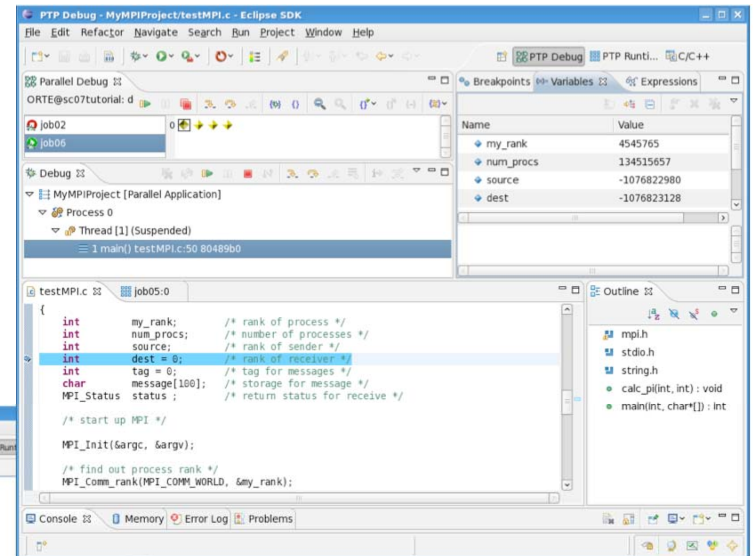
```
protected boolean isMPIArtifact(IASTName funcName) {
  IBinding binding = funcName.resolveBinding();
  String name=binding.getName();
  String rawSig=funcName.getRawSignature();
  //Sometimes names are empty (e.g. preprocessor change) or represented differently
  name = chooseName(name, rawSig);

  IASTName[] decls=funcName.getTranslationUnit().getDeclarationsInAST(binding);
  for (int i = 0; i < decls.length; ++i) {
    // IASTFileLocation is file and range of lineNos
    IPath includeFilePath = new Path(decls[i].getFileLocation().getFileName());
    // see if it's in the list of known (MPI) Include paths
    for (String knownMPIIncludePath : includes_) {
      IPath includePath = new Path(knownMPIIncludePath);
      if (includePath.isPrefixOf(includeFilePath))
        return true;
    }
  }
  return false;
}
```

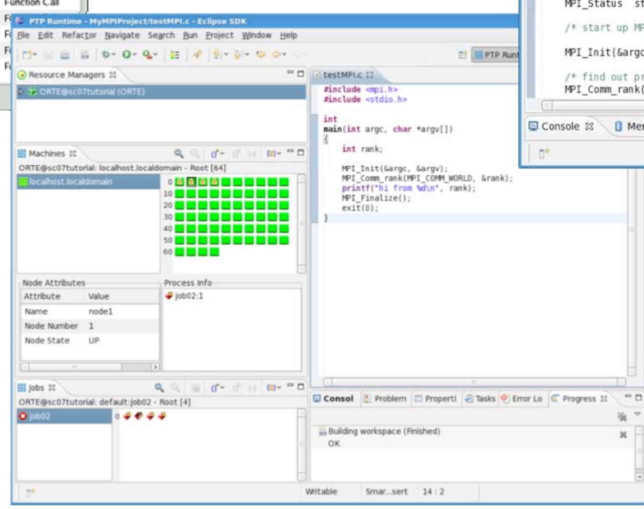
PTP from 30,000 feet
Parallel Tools Platform
http://eclipse.org/ptp

PTP BOF
Tues. 8:45 PM



Leverages
Eclipse CDT
& Photran
(eventually)





PLDT:
Parallel Lang.
Dev. Tools
(Analysis Tools)

PTP Debugger

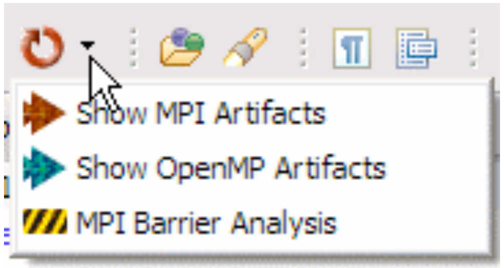
PTP Runtime

Constructed by PTP's PLDT:

- Call Graph
- Control Flow Graph
- Dependency Graph (Defined/Use Chain: partial)

In order to do:

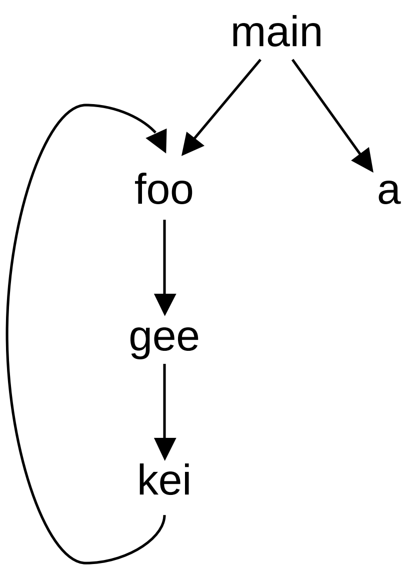
- MPI Barrier Analysis: detect deadlocks; find concurrently executed statements



Caveats:

- C only (not C++)
- No UI - structures used for analysis only

Call Graph - PLDT (example)



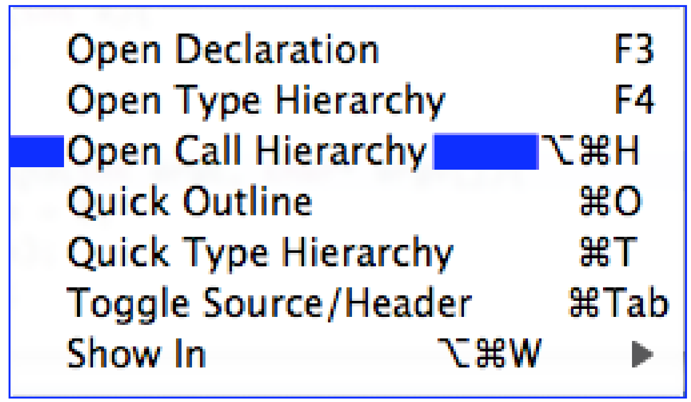
Recursive calls detected…

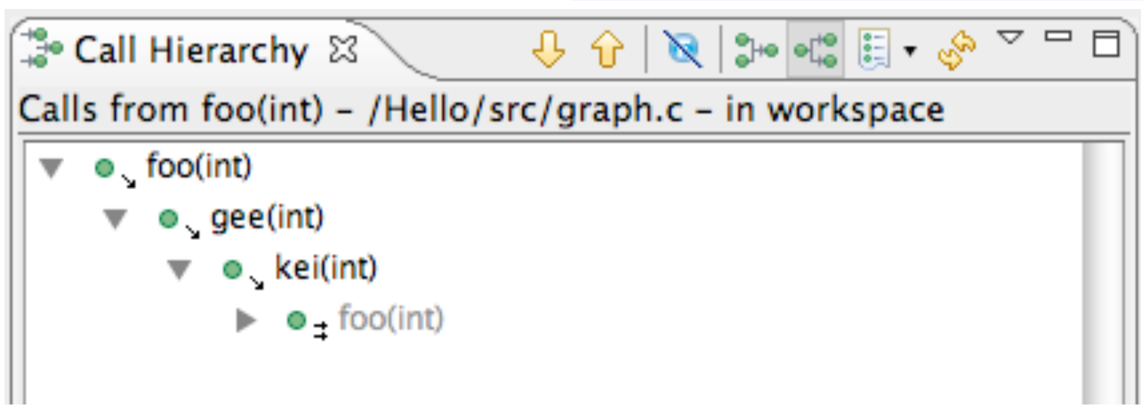
A cycle is detected on foo, gee and kei

```
#include "mpi.h"
#include "stdio.h"

void foo(int x);
void gee(int x);
void kei(int x);

void foo(int x){
  x ++;
  gee(x);
}

void gee(int x){
  x *= 3;
  kei(x);
}

void kei(int x){
  x = x % 10;
  foo(x);
}

void a(int x){
  x --;
}

int main3(int argc, char* argv[]){
  int x = 0;
  foo(x);
  a(x);
}
```

CDT’s Call Hierarchy view - partial call graph

- Call Hierarchy view shows info for a selected function
  - From context menu within CDT editor:





Call Graph - PLDT - how to

- As part of the PLDT analysis, call graphs are constructed.
- `org.eclipse.ptp.pldt.mpi.analysis.cdt`
  Project isolates generic analysis code
- `org.eclipse.ptp.pldt.mpi.analysis.cdt.graphs.GraphCreator`
  class has several convenience methods

```
GraphCreator graphCreator = new GraphCreator();
Iresource resource = …
// Initialize call graph with function info
ICallGraph cg = graphCreator.initCallGraph(resource);
// Compute callers & callees
graphCreator.computeCallGraph(callGraph);
// print call graph to console
graphCreator.showCallGraph(callGraph);
```

Sample plugin

Shows call graph

- text only

Control Flow Graph

- A **control flow graph (CFG)** is a representation of all paths that might be traversed through a program during its execution. Each node in the graph represents a basic block, i.e. a straight-line piece of code with a single point of entry and a single point of exit
- **A Statement Level CFG** is a CFG with individual statements instead of larger basic blocks.
  - *PLDT builds a statement level CFG as described here*

Control Flow Graph creation

```
// get the first node in the CallGraph
ICallGraphNode node = callGraph.topEntry();
IASTStatement funcBody = node.getFuncDef().getBody();
// create CFG from a statement
IControlFlowGraph cfg = new ControlFlowGraph(funcBody);
cfg.buildCFG();


// print CFG
IBlock entryBlock = cfg.getEntry();
for (IBlock block= cfg.getEntry(); block!=null;
               block = block.getTopNext()) {
     block.print();
}
```

Motivation: AST vs. CFG (Control Flow Graph)

- What can CFG provide that AST cannot?
- AST alone is not the right representation to do all static analysis

```
5  if(..)
6    x=0;
7  else
8    x=1;
9  y=x;
```

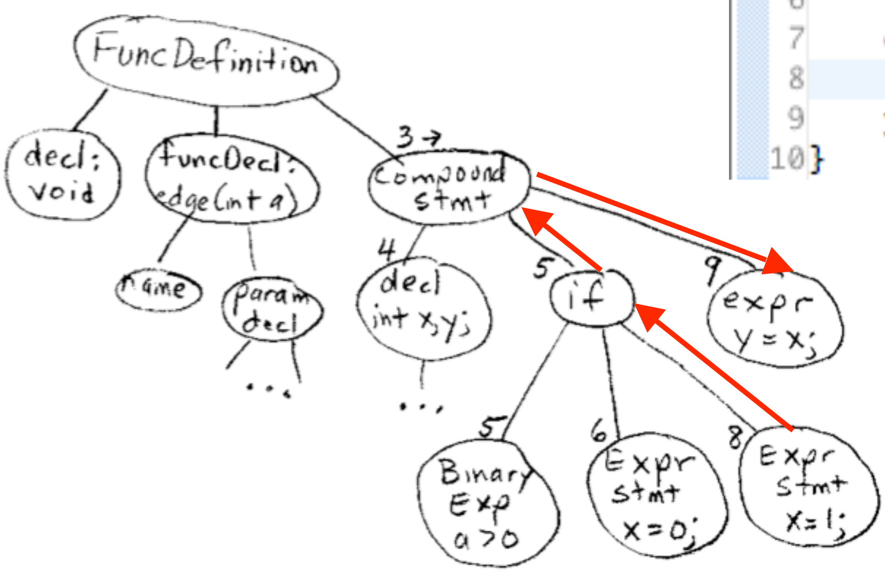
•*Live variables: still used in program (x)*

•*Data flow: need to determine what values could flow into x at 9*

•*CFG has direct edge from 8 to 9; using AST would have to walk tree backwards*

Motivation: AST vs. CFG (Control Flow Graph)

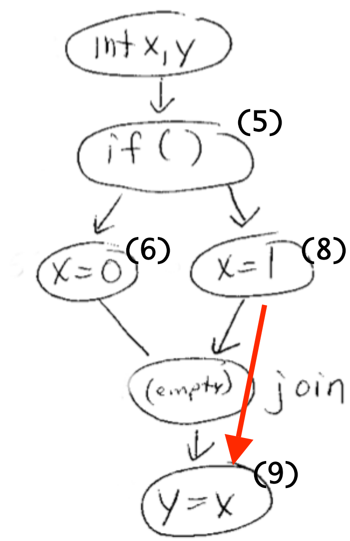
AST



CFGtest.c

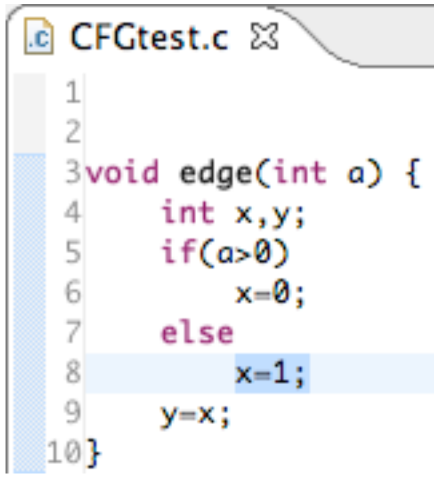
```
1
2
3 void edge(int a) {
4     int x,y;
5     if(a>0)
6         x=0;
7     else
8         x=1;
9     y=x;
10 }
```

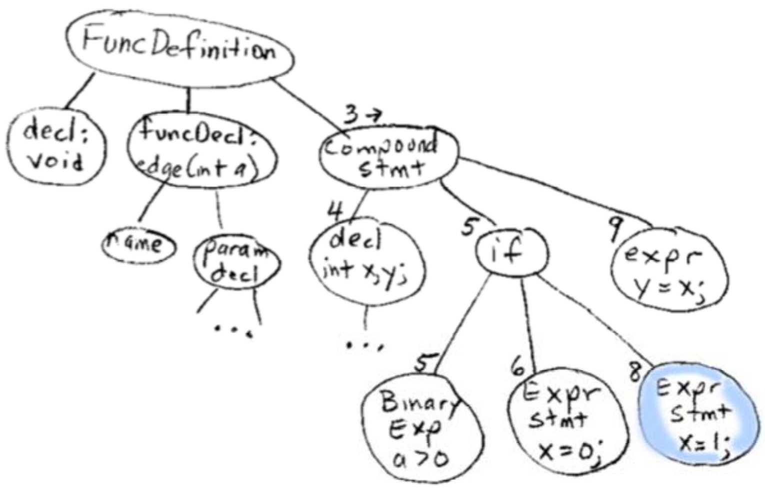
CFG

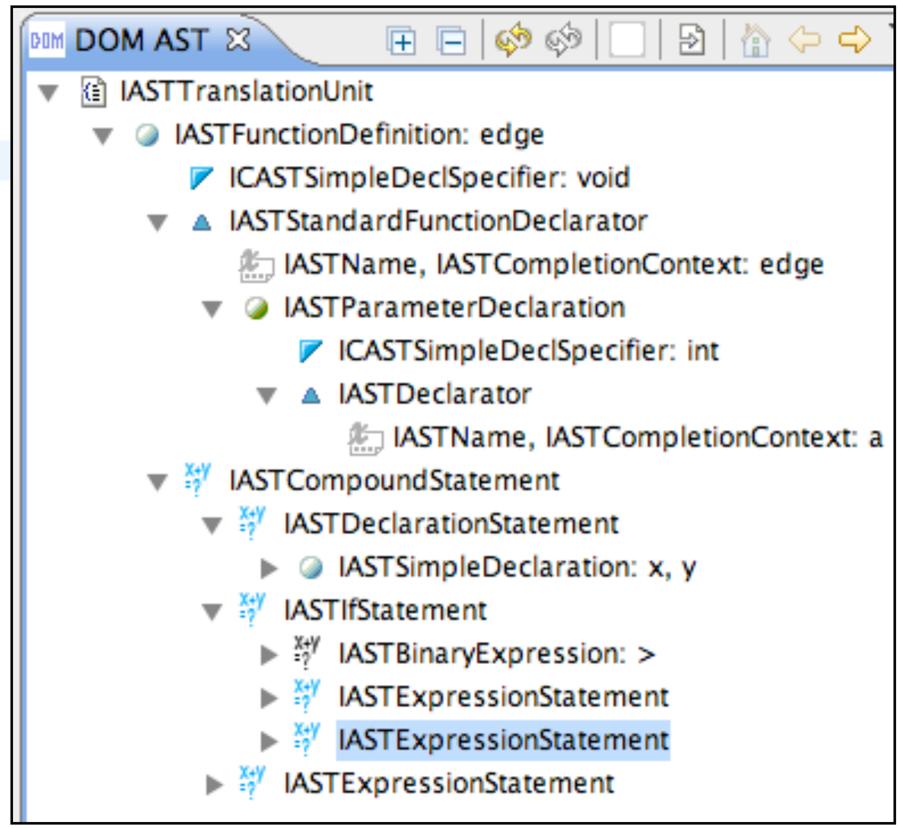


- To walk from line 8 to 9, in AST would have to walk backwards up tree to IASTIfStatement, then next to y=x expr on line 9
- Control Flow Graph (CFG) has direct edge from 8 to 9

AST Illustrated



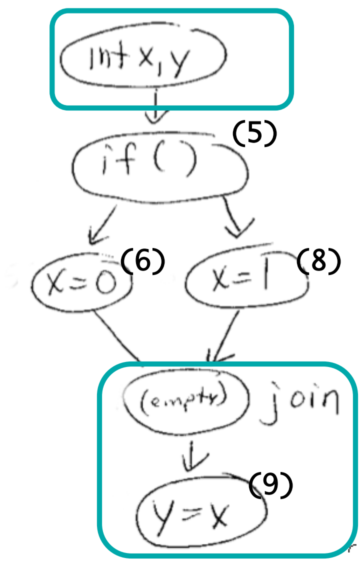


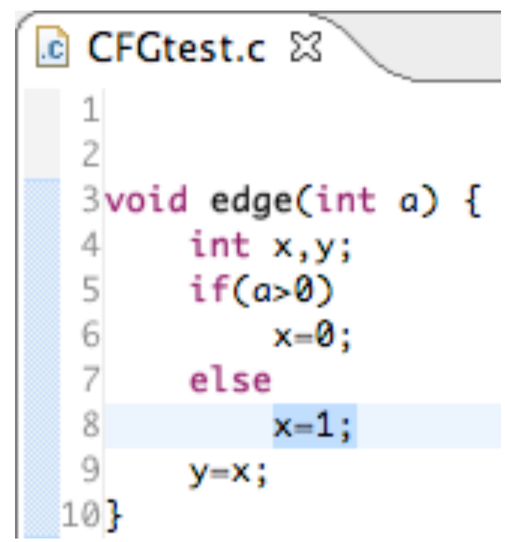


CFG illustrated

- To walk from line (8) to (9), in AST would have to walk backwards up tree to IASTIfStatement, then back down to y=x expr on line 9
- Control Flow Graph (CFG) has direct edge from (8) to (9)

```
Block 0:    Empty block                  (Line No)
  flows to: 2,
Block 2:    CASTDeclarationStatement  int x,y;
  flows to: 3,
Block 3:    CASTIdExpression  true          (5)
  flows to: 5, 6,
Block 6:    CASTExpressionStatement   x=1; (8)
  flows to: 4,
Block 5:    CASTExpressionStatement   x=0; (6)
  flows to: 4,
Block 4:    Empty block
  flows to: 7,
Block 7:    CASTExpressionStatement   y=x;  (9)
  flows to: 1,
Block 1:    Empty block
```



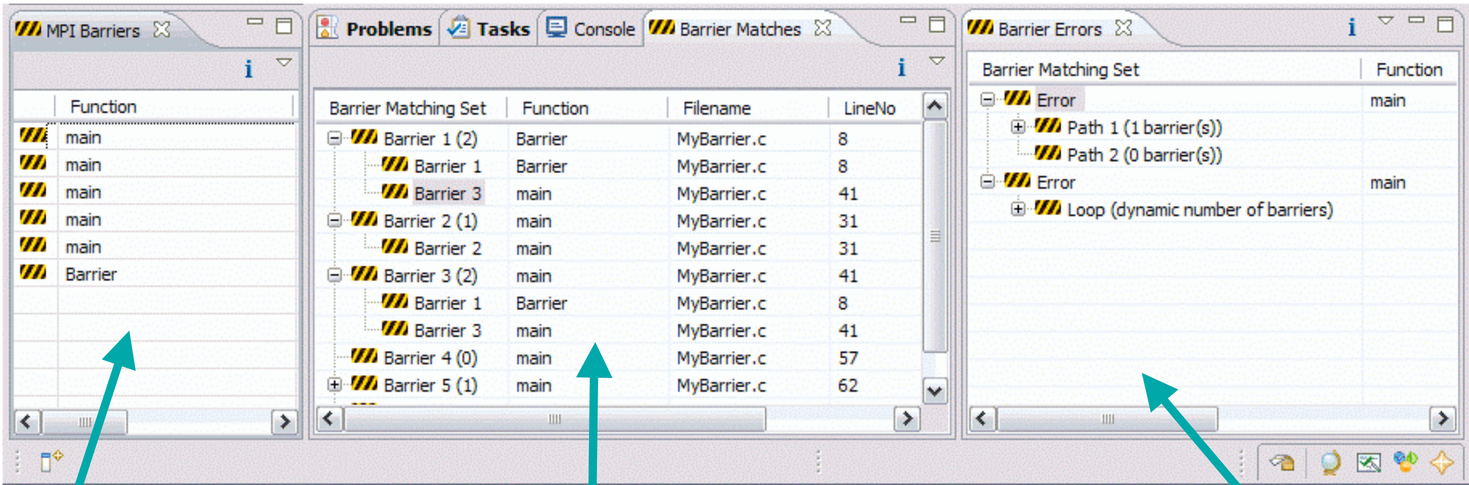


Basic Block

Direct flow from
line (8) --Block 6 to
line (9) --Blocks 4-7

Use of CFG: MPI Barrier Analysis

Finds potential deadlocks in MPI code due to mismatched MPI_Barrier statements



MPI Barriers view

Simply lists the barriers

Like MPI Artifacts view, double-click to navigate to source code line (all 3 views)

Barrier Matches view

Groups barriers that match together in a barrier set – all processes must go through a barrier in the set to prevent a deadlock

Barrier Errors view

If there are errors, a counter-example shows paths with mismatched number of barriers

Demo (if time)

Static Analysis and related features
A peek at things to come

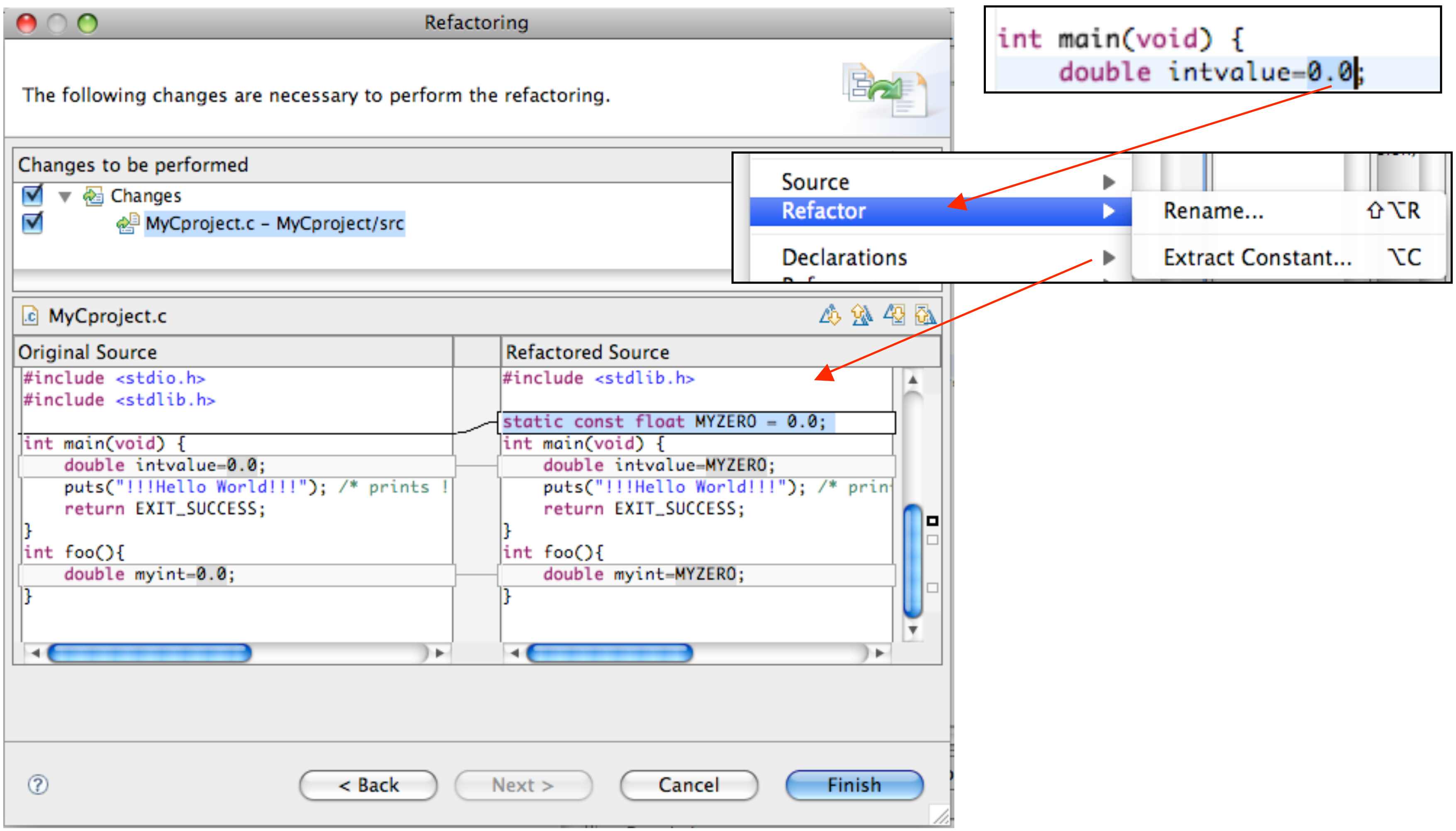
- Refactoring in CDT 5.0
  - New framework with AST Rewriter has potential for complex and useful refactorings
- PTP/PLDT integration of compiler information
- PTP/PLDT Source code instrumentation
  - Usable by Dynamic Instrumentation

Refactoring (CDT 5.0)

- New refactoring framework in CDT 5.0
  - Leverages Platform refactoring framework in LTK
- Current refactorings:
  - Rename (class, variable, etc)
  - Extract Constant - example of new framework use
  - In `org.eclipse.cdt.ui`: See `org.eclipse.cdt.internal.ui.refactoring.extractconstant.ExtractConstantRefactoring`

    1. `checkInitialConditions(..)`
    2. `createChange()…`
    3. `checkFinalConditions().`

CDT 5.0 Refactoring: Extract Constant



Future: Integrating Analysis information from Compilers

- Compilers have stronger tools for analysis than what we have in CDT! And more years of experience generating the info
- Compiler information offered to expose to PTP users:
  - IBM xlC/xlC++, xlF (C, C++, and Fortran)
  - HP compilers (C, C++, and Fortran)
  - U.Houston
- Compiler output supplied in various forms
- Exposed to use in Eclipse view, mapped to source line
- Expected types of output:
  - Parallelization attempts, hindrances
  - May assist user in manual parallelization of code, for example

Future: Source Code instrumentation

- Instrumentation of Java code is being done for IBM's TuningFork, and extensions to do this for C/C++ code as well are planned.
- Summary: use AST and pgm info to make decisions about how to instrument code (add statements for gathering information during dynamic analysis / performance tuning)
- Planned to be part of PTP Perf. Analysis Framework

A Performance Analysis Framework For C/C++ and Fortran - Wyatt Spear (University of Oregon)
EclipseCon Short Talk Wednesday, 15:40,
10 minutes | Room 209/210 |

Summary

- CDT has the basics for Static Analysis, including AST (Abstract Syntax Tree)
- Other useful graphs are built by PTP's PLDT
  - PLDT=Parallel Language Development Tools
  - Call Graph, Control Flow Graph, etc.
  - These graphs make analysis more straightforward
- Other features are in the works
  - CDT 5.0: Refactoring framework + refactorings
  - PLDT:
    - Integration of external tools' analysis findings (e.g. compilers)
    - Source Code Instrumentation uses AST to find instrumentation points

http://eclipse.org/ptp Parallel Tools Platform

Legal Notices

- Code samples in this presentation are Copyright 2008 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0.

- THE INFORMATION DISCUSSED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, SUCH INFORMATION. ANY INFORMATION CONCERNING IBM'S PRODUCT PLANS OR STRATEGY IS SUBJECT TO CHANGE BY IBM WITHOUT NOTICE.

- IBM and the IBM logo are trademarks or registered trademarks of IBM Corporation, in the United States, other countries or both.
- Java and all Java-based marks, among others, are trademarks or registered trademarks of Sun Microsystems in the United States, other countries or both.
- Eclipse and the Eclipse logo are trademarks of Eclipse Foundation, Inc.
- Windows is a registered trademark of Microsoft Corporation in the United States, other countries, or both.
- Mac is a trademark or registered trademark of Apple, Inc., registered in the U.S. and other countries.
- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Other company, product and service names may be trademarks or service marks of others.