

Photran Debugging

Feanil Patel

`fpatel@uiuc.edu`

Ken Schultz

`kschultz@uiuc.edu`

Shawn Temming

`temming@uiuc.edu`

Table of Contents

Introduction.....	4
Background.....	5
Eclipse.....	5
CDT	5
Extension Points for other Languages	6
Photran Debugger from CDT debugger	6
What the project is about?	6
How the problem was handled before	7
Requirements.....	8
Remove U Variables.....	8
Replace brackets with parentheses.....	8
Correct Lower bound and Upper bound	9
Arrays and Structures in the Expressions View.....	9
Strings.....	10
Pointers	10
Conditional Breakpoints.....	10
Multi-Dimensional Array Display	11
Stop at "main".....	11
Watchpoints	11
Host Association Variable Visibility.....	11
Include File Variable Visibility.....	12
Variables View Type Derivation.....	12
Multidimensional Array Display.....	12
Mouse-Over Array Inspection.....	12
Fortran Watchpoints.....	12
Fortran Expressions View.....	13
Variable Change Tracking.....	13
Set Variables to NaN.....	13
OpenMP Support.....	14
Features.....	15
Remove U/SC Variables.....	15
Conditional Breakpoints.....	16
Strings.....	17
Replace Brackets with Parentheses.....	18
Array Upper bound Lower bound.....	19
Conditional Breakpoints.....	22
Moving to Plugin.....	25
Alternatives.....	27
Testing.....	30
Problems with Automated Testing.....	30
Manual Testing.....	32
Non-technical Problems.....	33
Client Location.....	33

Lack of Documentation.....	33
Future Development.....	35
Source lookup on Windows.....	35
Variables View Icons.....	36
Things We wish we had known.....	38
Fortran	38
Eclipse Framework	38
JUnit UI testing	39
Conclusion.....	41
Appendix: Developer Docs.....	42
Setup Eclipse to Develop for Photran.....	42
Install Cygwin.....	42
Installing g95.....	42
Adding Cygwin to your Classpath.....	43
Install JRE/JDK.....	44
Setting up Eclipse and Obtaining the Sourcecode.....	44
Windows Source Lookup with Cygwin.....	48
Manual Test.....	50
Files of Interest.....	56
CValue.java	56
MISession.java.....	57

Introduction

Over the last couple of years, a Fortran IDE was developed at the university of Illinois using the Eclipse and CDT frameworks. However the IDE, which was named Photran, had a debugger which was quite underdeveloped. With the guidance of the senior project staff and our customers Brian Smith and Walt Brainerd, we established a set of requirements to improve the usability of the debugger. In this paper we will outline the changes we made, how we made them as well as general information about Photran which we hope will serve as documentation for future developers who would like to contribute to this project.

Background

We will begin by introducing the technologies that we dealt with during the course of this project as well as a little background about the project.

Eclipse

Eclipse is an open-source development platform. It was developed at IBM and released in 2001. It is currently a widely used open source platform followed by a large open-source community. Eclipse is meant to be a Rich Client Platform (RCP), which allows the interface of eclipse to not only be very rich, but also very extensible. This combines with the use of plug ins allows Eclipse to be a very powerful software development application.

CDT

One such set of plug ins is the C/C++ Development Toolkit. CDT is meant to be a fully functional, powerful C/C++ IDE built upon the Eclipse framework. It uses extension points in the Eclipse framework to allow it to write and parse C code. It also provides many tools that aid in C/C++ development such as a debugger.

Extension Points for other Languages

Originally in CDT you could only write C/C++ code, but, thanks to recent additions to the CDT framework by Jeff Overbey, extension points have been placed in CDT so that the CDT framework itself can be extended even further to allow for the development of other languages. Extension points allow for better code reuse because code need not be copied in order to make changes.

Photran Debugger from CDT debugger

The extension points were added to CDT because of the Photran IDE, which is a Fortran development environment. Photran originally was a copy of CDT that was modified to work with Fortran as opposed to C/C++. However, the Photran development team soon realized that a lot of the code from CDT was simply being re-used and not modified. Realizing this, they moved Photran to its own plugin that built upon the CDT framework. However due to problems in CDT they had to release a patch along with Photran that would patch CDT to have extension points that Photran could hook into. Currently the patch is no longer required because the extension points have officially become a part of CDT.

What the project is about?

However, the fact that Photran is built on top of CDT has also caused a few problems, particularly during debugging. Currently Photran does not have a debugger of its own. As a result when one invokes the Photran debugger, the CDT debugger is run. The problem with this is that the information provided by the CDT debugger does not make sense in a Fortran environment. This makes debugging a Fortran project much

more difficult. The goal of this project is to improve the Photran debugger so that it reports pertinent information clearly to the user and allows the user to efficiently interact with the debugger.

How the problem was handled before

Before this project there was no other project that tried to address this problem. We are the first to tackle the debugging section of Photran, as they have been using the CDT debugger straight out of the box previously.

Requirements

After talking with our customers we established a number requirements for what changes needed to be made to the debugger to improve its usability. Below we enumerate the requirements, give a brief description about each of them, as well as our initial assessment as to why the problem was occurring. Later on we will explain in detail what the actual solutions were for the features that we chose to implement.

Remove U Variables

In the Variables View the user sees many variables with the name $U[number]$. These variables are not user defined and do not provide any useful information so they should not be displayed.

GDB/MI is a line based machine orientated text interface that Eclipse uses to interact with GDB. Eclipse has a class just for interpretation of MI. Since GDB does not natively understand g95 Fortran it most likely misinterprets some internal variables and outputs them as what we know as the U variables.

Replace brackets with parentheses

When displaying a multidimensional variable in the variables view the syntax $[length]$ is used instead of $(length)$ both for subscripts and size. This conflicts with the Fortran

syntax standards because in Fortran parentheses are used when referring to array subscripts. This is different from C/C++ syntax where brackets are used.

GDB/MI outputs syntax similar to C/C++ as that was the initial use for GDB. It has now been adapted for many other languages. The brackets were used to indicate the size of an array.

Correct Lower bound and Upper bound

When multidimensional variables are displayed in the variables view the lower bound is always set to 0 instead of the lower bound set in the Fortran code which can be any integer. Similarly, the upper bound is set to the length of the variable instead of lower bound plus length (i.e. A length 10 array with a lower bound of -2 should have an upper bound of 8, not 10).

When the variable is populated we need to keep track of the offset to account for it when the array index is displayed. The offset will be from 0 in order to account for the C/C++ syntax that is enforced now.

Arrays and Structures in the Expressions View

GDB does not accept Fortran array sections, structure constructor, array constructor, array element, or structure component syntax as a valid part of an expression when using the expressions view.

This is a problem with the language. It seems that GDB is using C/C++ syntax instead of Fortran syntax. The proper syntax for an array in Fortran is (/ 1, 2, 3, 4, 5, 6 /) however the expressions view does not currently recognize it. This problem can most

likely be solved by changing the language that GDB accepts.

Strings

The arrays should be displayed as a whole word not an array of characters. The syntax corresponding to an array of characters is uniform with C syntax where in Fortran a string is a single element.

This is a problem with the language. It seems that GDB is using C/C++ syntax instead of Fortran syntax. We will have to set the language for GDB in order for it to behave properly.

Pointers

Pointers should show their target or null rather than the memory location. In Fortran after a point is defined, once it is set to point to an object it can be used just like the object without dereferencing.

GDB is not set up to interpret g95 and most likely does not have support for pointers currently. Perhaps patching GDB for g95 will fix this problem.

Conditional Breakpoints

Conditional breakpoints will only accept the “!=” inequality syntax as in C/C++ and not “.ne.” or “/=” that is used in Fortran.

This is a problem with the set language in GDB. It is interpreting the expressions in conditional breakpoints as C/C++ instead of Fortran. Once GDB is correctly set to Fortran this problem should be fixed.

Multi-Dimensional Array Display

Multidimensional Arrays are only displayed as a single array in the Variables view.

All other data is missing from the variable.

This is a problem with the language. It seems that GDB is using C/C++ syntax instead of Fortran syntax.

Stop at "main"

Change the stop at main() so that it stops at the first executable statement of the main program. This will allow breakpoints to be set at that time and will allow the use of “Step Over” and other debug perspective buttons without having to set a breakpoint prior to execution.

Watchpoints

Currently watchpoints do not function as one would expect them to. For some reason GDB/MI does not track changes on a variable.

The Expressions View is another place where the syntax should be Fortran, not C.

Host Association Variable Visibility

Variables accessed by use and host association (module variables and host variables for internal/module procedures) should be visible. Currently, when you are inside a module procedure, you can see variables that are inside that procedure but you cannot see variables that are outside the procedure but inside the host module.

Include File Variable Visibility

Variables defined in an include file should be visible. This problem seems to have been fixed already but should be kept in mind so that we do not break it.

Variables View Type Derivation

The (x) symbol in the variables view provides no information. This symbol is a placeholder when a variables type cannot be determined.

Jeff Overbey is going to create something similar to a hash lookup table that we can use to determine the type of a variable.

Multidimensional Array Display

The same lower bound and upper bound error as with linear arrays reoccurs with multi-dimensional arrays, but at the same time with a multidimensional array the array is flattened into a single dimensional array.

Mouse-Over Array Inspection

When the mouse is over a variable in code it should inspect the variable and display the value appropriately.

Fortran Watchpoints

Since the language for GDB is not correctly set it does not accept Fortran syntax.

We need to correctly set the language of GDB to Fortran so that it will accept Fortran syntax. We also need to provide user documentation to explain the features of the

debugger.

Fortran Expressions View

The expressions view does in fact work, but since the language for GDB is not correctly set it does not interpret the syntax correctly and does not seem to function.

Variable Change Tracking

When the program pauses, it should provide the capability of determining where (in the source code) a particular variable last changed (i.e., got its current value). Clicking on something might take you to the relevant line of source code. Being able to go back three times would be even more useful. Currently, this feature does not exist in any form.

We would have to create “silent” watchpoints for every variable and update an internal tracking variable. Then we would have to provide access to this tracking variable as well as click able source-lookup.

Set Variables to NaN

In the variables view the user should have the capability to set all real and complex variables to NaN (the compiler should do this, but many do not). Currently NaN is not accepted as a valid value, and so variables cannot be set to NaN via the variables view interface.

We do not have any approaches to fix this problem will most likely not have time to work on this feature.

NaN Watch

Have an option where whenever a real or complex is set to NaN by execution of the program, the source line is also stored as part of its value. When a program pauses, part of the information relating to the value of that variable is the source line; clicking on it takes you to the source line. However, NaN is not accepted as a valid value as stated in the previous issue.

This problem is related to the previous problem.

OpenMP Support

OpenMP is a library for using with threading when working on a multiprocessor system. When in such programs the debugger should provide an easy way to interact with the different threads of your program.

This is a major issue and would require vast more resources. This itself could be another project on its own.

Features

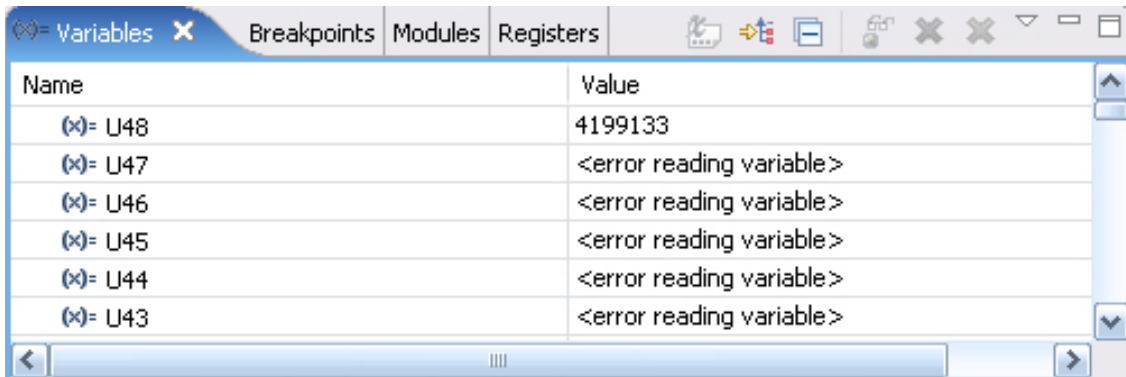
Of the requirements that we have laid out, we implemented the removal of the U variables, allowing for conditional breakpoints, displaying strings, changing brackets to parentheses when referring to array indices, and displaying array bounds properly. Once we did all this we moved our changes into a set of PlugIns that together created a new debugger for Photran that worked with CDT but was separate from the CDT C/C++ debugger.

Remove U/SC Variables

One of the largest problems that kept the Photran debugger from being very useful was the fact that as the debugger was running, the variables view would be filled with erroneous variable that were not part of the program. We refer to these as the U/SC variable because the variable names begin with either U followed by a numeric or SC followed by a numeric. In order to solve this problem we found where Photran updated the variables in the Variables View and filtered out the U variables. We did not want to completely remove them as the function of the variables is still not entirely clear, but we have come to understand that they are internal variables that GDB uses to evaluate Fortran.

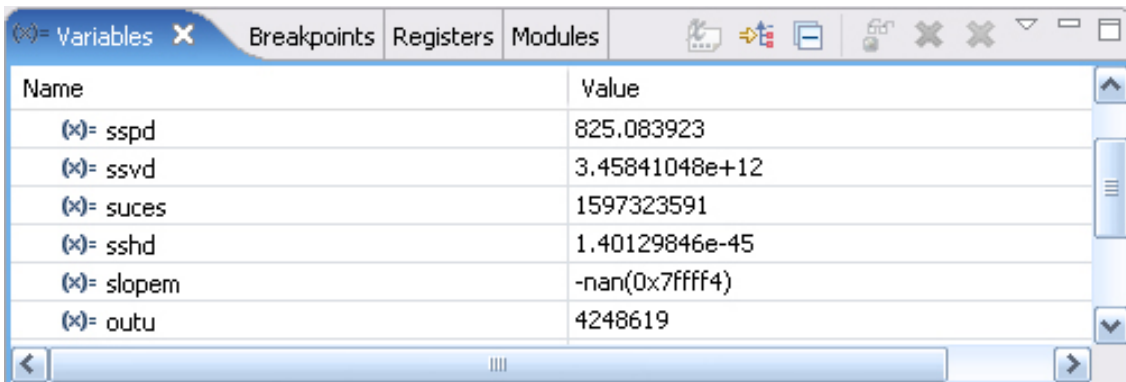
As we continued development over the next semester we discovered that the U

variables were not the only erroneous variables displayed by the Variables View. With the discovery of the SC variables we expanded our filter to not only remove U variables but all variables that had any uppercase letters. We are able to do this safely because Fortran is a case insensitive language and GDB down cases the names of all variables that are a part of our compiled program.



Name	Value
(x)= U48	4199133
(x)= U47	<error reading variable>
(x)= U46	<error reading variable>
(x)= U45	<error reading variable>
(x)= U44	<error reading variable>
(x)= U43	<error reading variable>

Illustration 1: U/SC Variables Before



Name	Value
(x)= sspd	825.083923
(x)= ssvd	3.45841048e+12
(x)= suces	1597323591
(x)= sshd	1.40129846e-45
(x)= slopem	-nan(0x7ffff4)
(x)= outu	4248619

Illustration 2: U/SC Variables After

Conditional Breakpoints

We noticed that conditional breakpoints were working if you used C/C++ syntax. From this we figured out that GDB did not recognize the Fortran .90 file extension and

was setting the language for GDB to the default C/C++. In order to ensure that proper Fortran syntax could be used in the conditional breakpoints we were forced to manually set the language to Fortran.

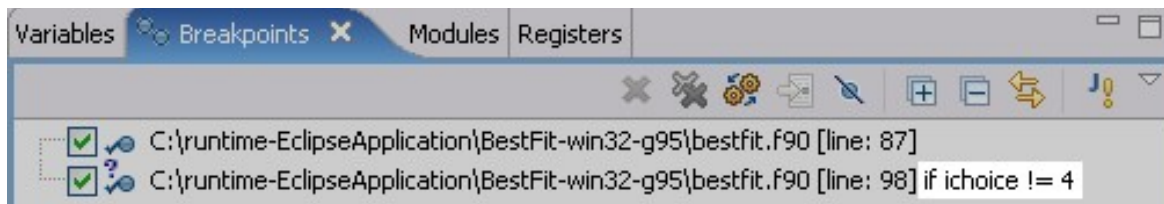


Illustration 3: Conditional Breakpoints Before

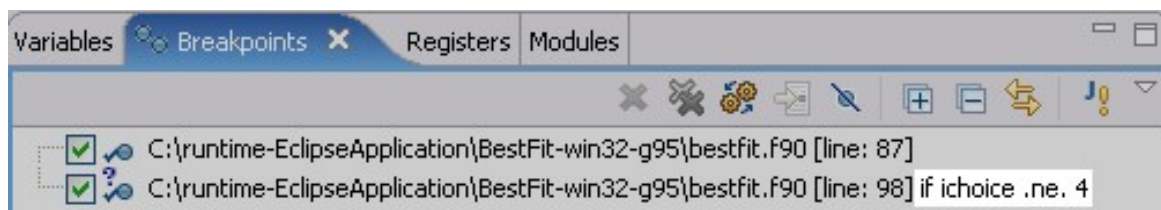


Illustration 4: Conditional Breakpoints After

Strings

When we changed the language for GDB, strings were also displayed properly. This problem was a byproduct of changing the language of GDB specifically to Fortran.

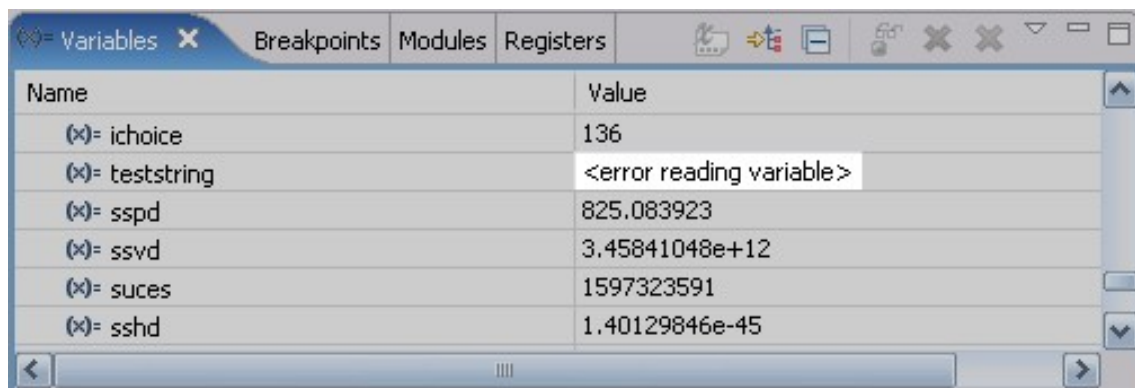


Illustration 5: Strings Before

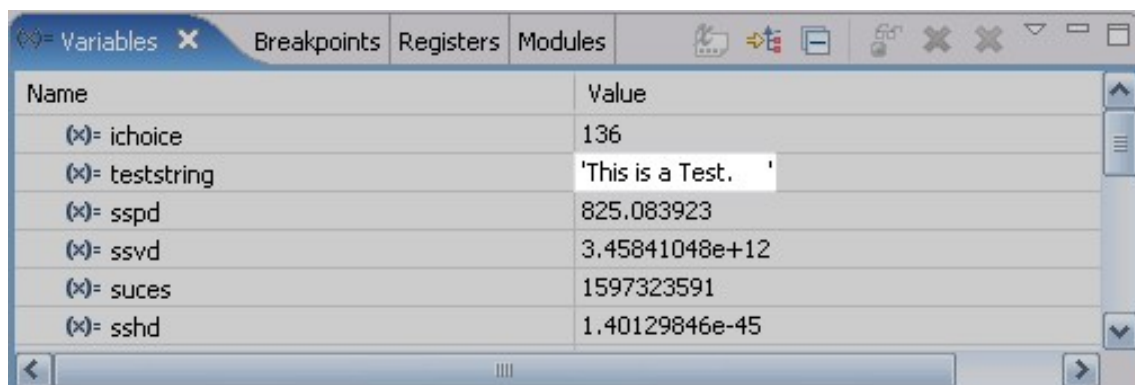


Illustration 6: Strings After

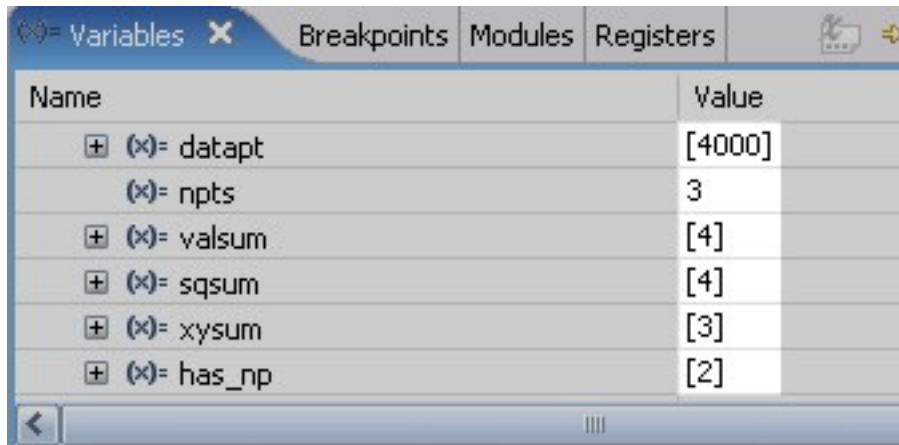
Replace Brackets with Parentheses

One of the first issues that we decided to tackle was changing brackets to parentheses when referring to array indices. Unlike C/C++, in Fortran, arrays are indexed using parentheses instead of brackets (e.g. `myArray(2)` instead of `myArray[2]`). Our initial guess at the problem was that GDB was using C/C++ and was just returning the wrong syntax. Our first step was to change the language GDB was using. After that change was implemented, GDB was still returning parentheses and we had to take a different route.

Our next, and final, step towards a solution was to capture the output before it was sent to the Variables View and change the brackets to parentheses where applicable. In order to do this, we had to understand the flow that the information took after being received from GDB but before being output to the screen. After much research, we were able to determine the location where the information was first being received from GDB. At first we were reluctant to change the information immediately in case it was used later by a different part of the program. However, after running some tests, we decided that this information was not used anywhere else and thus it was safe to change it immediately.

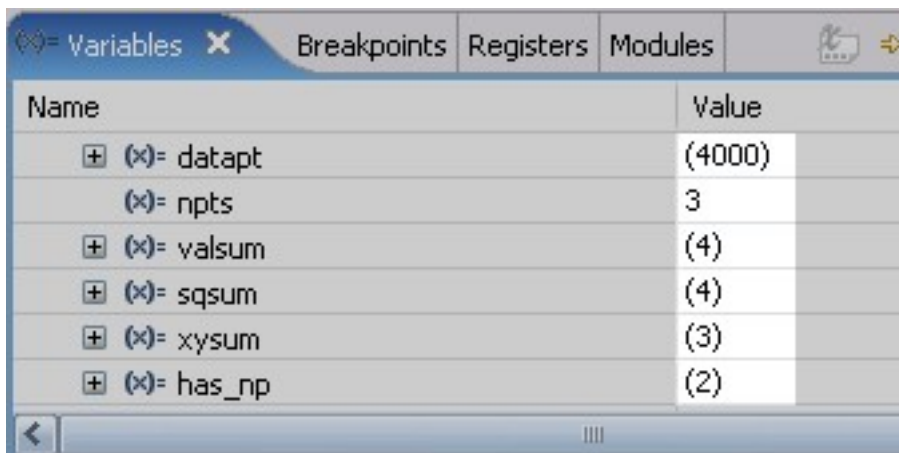
To implement the change, we decided that a sufficient and necessary condition for when we had to change the value returned by GDB was if the value started and ended with brackets. We then did a string replacement on these values to change the enclosing brackets to parentheses. These changed values then propagated to the

Variables View in the normal method and resulted in a display to our client's specification.



Name	Value
+ (X)= datapt	[4000]
(X)= npts	3
+ (X)= valsum	[4]
+ (X)= sqsum	[4]
+ (X)= xysum	[3]
+ (X)= has_np	[2]

Illustration 7: Array Index Before



Name	Value
+ (X)= datapt	(4000)
(X)= npts	3
+ (X)= valsum	(4)
+ (X)= sqsum	(4)
+ (X)= xysum	(3)
+ (X)= has_np	(2)

Illustration 8: Array Index After

Array Upper bound Lower bound

The first problem we encountered when working on this issue was that the underlying debugger that we are using (GDB) does not support Fortran multidimensional arrays. It flattens all arrays in column-major order and does not retain any information about the original array dimensions. It also does not store any information about upper and lower bounds, even for one-dimensional array, but rather reverts everything to a zero-

indexed array. Therefore, when we ask GDB for the information about an array, one or more dimensional, we can only get the total size of the array and a flat, zero-indexed array representation of the data. To actually get the dimensional information we had to resort to using a section of Photran that Jeff was developing.

For his refactoring tools, Jeff was developing a symbol table that would hold all the variable information such as name, type, scope, etc. Most importantly, the symbol table held the array information we needed such as number of dimensions and bounds. At the time, there was no public interface into this information however. Luckily, Jeff was extremely helpful and offered to implement a public interface to our specifications.

Once the interface was done, we thought that was going to be the end of this problem, but it wasn't. In order to use Jeff's code, we had to provide a filename and line number in order to get the current scope. This turned out to be a non-trivial task that involved taking a roundabout path through the eclipse framework and then back to Photran. Although roundabout, the path worked and we were able to line number and file information. The next problem that we encountered was that the file information we could retrieve was not exactly what we needed.

The first problem was that it was an absolute path when we needed a path relative to the project folder. The second problem was with the format of the path. In order to get the file information for a file, CDT uses Linux commands such as `pwd` to get the information from the system. Since we were operating on a Cygwin base, the path that was returned was of the form `"/cygdrive/c/ ... "`, but the look-up command for the file expected the windows form of `"c:/ ... "`. A quick string replacement to fix the path

form, and a new function call to a get a relative path, and we were able to get the array information. Fortunately, we have become experts at string replacement from previous work on this project so this setback was minor.

At this point, we are able to display the upper and lower bound information for every dimension. However, our current implementation still does not support getting information about arrays that do not have their dimensions explicitly defined with literals (e.g. not assumed shape and not using a variable for the bounds). Also, it cannot determine the information about arrays that are part of a derived type. These limitations are either inherent limitations of the symbol table (i.e. assumed shape arrays can never be determined by this method) or limitations in the current implementation of the symbol table (i.e. access to derived types is not implemented).

We also have several suggestions for future improvements that we would have liked to implement but could not due to time constraints. The first is to display each dimension of the multi-dimensional array as a child element of the main array. Then, if each dimension were loaded on demand only, handling many-dimensional arrays would be much easier and lightweight. For example, say you have an array with five dimension and eighty elements per dimension. The current implement loads all 4000 elements at once while the proposed implementation would only load eighty elements at a time. Decreasing the number of elements loaded at once would help to increase the responsiveness of the Variables View.

Another suggestion involves taking a direction perpendicular to the previous suggestion. Instead of creating child elements for each dimension, still load the entire array at once. However instead of showing the index as an offset from zero in a flat

column-major array, show the index as its position in the multi-dimensional array (e.g. instead of showing “15” show “(5,3)”). This new indexing system would make searching for a particular value much easier and increase usability.

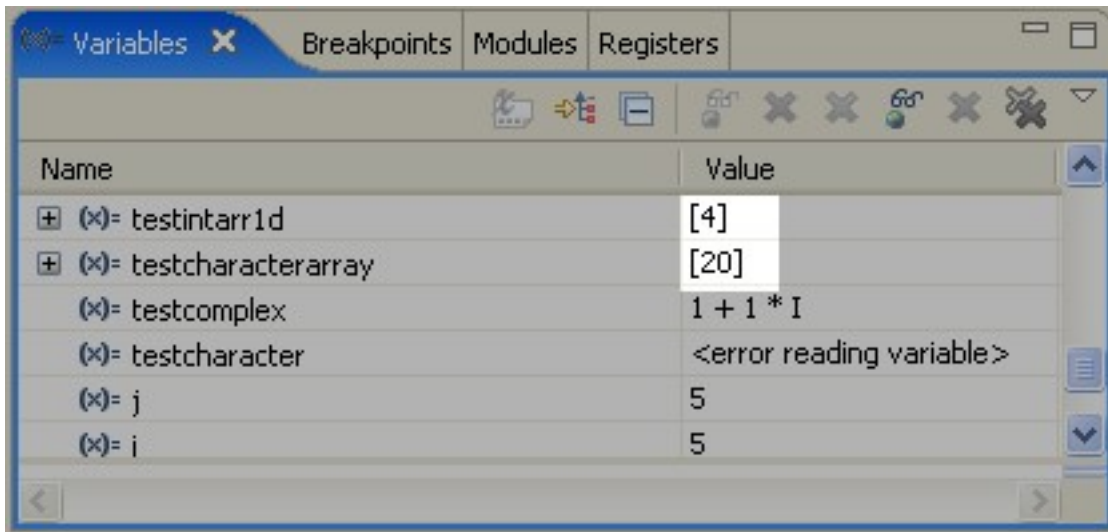


Illustration 9: Array Bounds Before

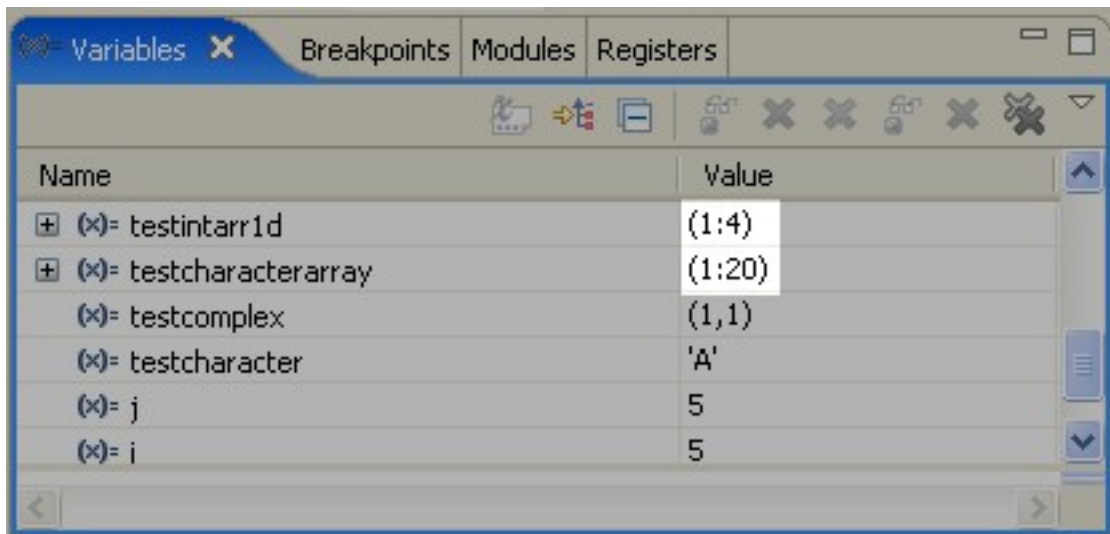


Illustration 10: Array Bounds After

Conditional Breakpoints

While first attempting to tackle the problem of conditional breakpoints, we started by looking at the breakpoints themselves. Our original line of thought was that the “\=” or

“ne.” was breaking the parsing of the condition. For “\=” we thought the slash might be escaping the equal sign and for “.ne.” we thought the characters were not being interpreted as an operator. After much research into this line of thought, we realized that our initial guess was probably incorrect and we switched gears. Taking into account some other problems that we were having at the time we realized that the problem might be that the wrong language was being used for some of the operations. We suspected that since we were building off CDT, the underlying debugger, GDB, might not be automatically detecting that we were using Fortran. Theoretically, GDB was supposed to use the file extension to automatically discover which language was in use. However, this appeared not to be happening.

The communication between the Photran debugger and GDB is done through GDB/MI, or MI for short. Within CDT there are many classes representing the different commands that MI allows. We also discovered that there was a class representing an debugging session using MI. At the beginning of a debugging session, an aptly name MISession variable is initialized. It was during this initialization that we were able to solve our problem.

We first inserted some diagnostic commands to determine whether GDB was actually getting the correct language or not. As suspected, although the language was set to “auto,” it was still using C/C++. The reason for this behavior was that GDB only supports Fortran77 which has a .f extension while we were using Fortran95 which has an unrecognizable .f90 extension. Lucky for us, GDB has a command to explicit set the language, and the CDT MI interface supported this command. By explicitly setting the language to “Fortran” during the session initialization we were able to kill several

birds with one stone, so to speak.

The first problem that was fixed by this command was conditional breakpoints. It turns out that the problem was all on GDB's end, so fixing GDB fixed everything. The “=” and “.ne.” syntax now worked properly for conditional breakpoints. Another problem that was fixed with this command was strings. Before this fix, strings did not display at all – there was an error message in their place. After this fix, they displayed properly in the Variables View. The problem was that the string representation was different between C and Fortran, so trying to interpret the data as a C string didn't make sense, but interpreting it as a Fortran string did.

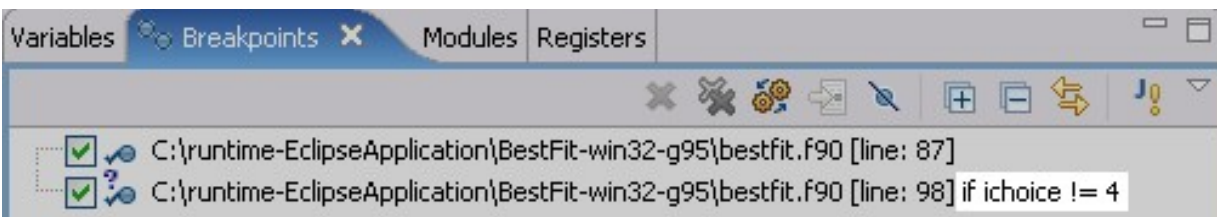


Illustration 11: Conditional Breakpoints Before

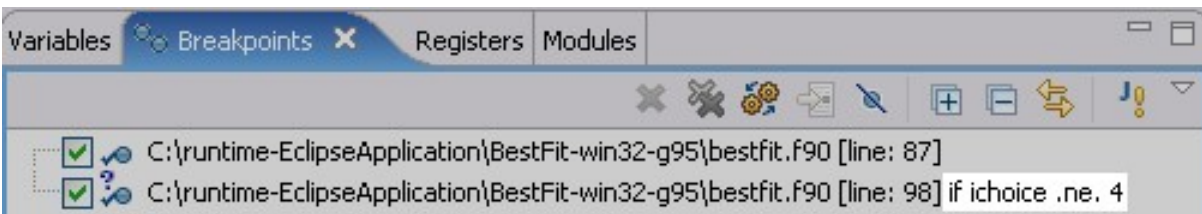


Illustration 12: Conditional Breakpoints After

Moving to Plugin

When we started this project our basic design principle consisted of two stages. The first stage occurred while we were still learning code and testing methods of correction. In this stage we are just focused on functionality. This functionality is normally accomplished by simply adding or changing the current CDT code. While this method of coding allows us to learn the given code and get a rough prototype, it is not a sustainable final solution. The second stage involves converting our modified CDT debugger into its own plugin that is part of Photran and can work alongside CDT if desired. The functionality of the first stage made into the plugin of the second stage will be the product we deliver to our clients.

There are two different options for creating a new plugin. The first option uses extension points that are built into CDT. We would then provide implementations for these extension points that allow Fortran to work in the debugger. Extension points would help Photran because the underlying implementation of CDT could change and as long as the extension points were maintained Photran would still function as intended. Extension points would also help CDT by making it a more flexible framework with more features that could be customized by plugins built on it. Unfortunately, if we add extension points we would have to modify CDT. It is developed and managed by a group that we are not a part of. This is the ideal solution,

but may not be practical due to time constraints and CDT bureaucracy.

The second option is to extend certain classes in CDT to provide our own implementation. We now have two different ways to integrate the new classes into the existing project. The first method would change CDT to properly use the correct classes, but has the downside of changing CDT. The second method is based on the fact that each plugin has a configuration controlling which classes are loaded for select components of the Eclipse framework. We could create our own configuration that loads CDT classes for most of the components and loads our classes for certain components. The reason we chose this method is that we do not have to modify CDT. If we change CDT classes we may cause unexpected behavior. A major source of concern in this project is being subject to volatility in a project we do not control. Ideally, we would want create extension points for CDT, but we cannot wait for permission to modify CDT before having a working product.

After careful consideration the option that we chose and implemented was to copy a fairly good amount of the CDT debugger code and change it to make it a debugger for Photran. This choice was made in part due to CDT's very long development cycle and bureaucracy. Integrating our changes into CDT's code would have taken far longer than the time we had to work on this project. At this time we began making the new plugin by copying the CDT debugger code and string replacing CDT with Photran in order to make the Photran debugger code distinct from the original CDT code. Then we had to make sure all the plugin configuration files such as plugin.xml reflected the naming configuration we made for the new Photran debugger code. After all this was done we had to copy the original CDT debugger launcher and modify it to make it a

Photran debugger launcher by string replacing CDT with Photran once again.

As the project continued, the plugin we made specifically for the Photran debugger set up the project for future development. Since the Photran debugger is now completely separate from the CDT debugger we do not have to worry about changes to the CDT debugger that could break the newly made Photran debugger plugin. Although, this ceases the automatic updates that CDT would provide.

During the development of the separate Photran debugger plugin we noticed that when we copied only the sections of code we thought we needed the debugger would not function the way we thought it should. An instance of this would be when you would try to create a debugger configuration for Photran and it would register as a CDT C/C++ debug configuration. To fix this we had to duplicate more code from the CDT debugger plugin for the Photran debugger plugin.

After all the configuration files for the plugin were changed to reflect Photran and the launcher was created for the plugin we were ready to move our changes from the hacked version of CDT to our new Photran debugger plugin. Because of the way we chose to make the plugin this was as simple as replacing the original files with the modified ones we had worked on. The next thing to do was to integrate Jeff Overbey's array code into our plugin. The code he wrote would give the array bounds for all arrays but derived type and assumed shape arrays.

Alternatives

As noted in a previous section, our current implementation of the Photran debugger is mostly a copy and paste from the CDT debugger with our changes inserted where

necessary. There are advantages and disadvantages to this approach, but ultimately having extension points in the CDT debugger would be a more robust solution. With extension points, we could be guaranteed that our implementation would not be broken by CDT changes while still getting the benefit of those changes. However, not all of our changes would make appropriate extension points. Theoretically, extension points in CDT would be used by many different plugins built off of CDT. Any proposed extension points need to have a universal application and not a specific Fortran/Photran application.

The first proposed extension point revolves around the change to MI`Session` that allowed us to change the language that GDB was using. Any plugin that builds off of CDT and uses GDB will most likely have additional configurations that they would like to perform on GDB. Therefore, creating an extension point in the initialization of an MI session that allows these plugins to provide customized initialization would make CDT much more robust and extensible. The current system hides too much of the functionality of GDB by provided default initialization.

The second proposed extension point stems from our work fixing the display of the Variables View. The current system displays variables only from a C/C++ point of view. Any language with different syntax or variable types does not work properly. Adding an extension point that allows for final customization of the data being output to the Variables View would have universal application to any plugin building off CDT.

With these extension points, and others, CDT would become much more of a generic framework for building language specific plugins rather than being so intimately tied

to C/C++. We were unable to implement these changes due to several reasons largely related to time constraints and focusing on functionality as a top priority.

Testing

Problems with Automated Testing

A major concern for this project was to establish a foundation for future development. We are the first group working on the Photran debugger but there will be many others. Therefore, we need to establish a solid testing plan to ensure a quality product. The solution we originally chose involved creating Junit tests to make sure certain features are working as intended. Ideally, we want a suite of tests that can be run and, if they succeed, give the developer confidence that everything is working.

The Eclipse framework provides an API for all of its components that we can use for testing. Theoretically, anything that is shown in the UI and any interaction with the UI can be simulated in code. Within our tests, we needed to take several steps before we could get to a place where we could test the debugger. First, we would have spawned a new instance of Eclipse with the Photran plugin loaded. Next, create a project in the workspace and import a test Fortran program into that project. Then we insert a breakpoint and build the program. After building the program, we run until it reaches the breakpoint. Once it suspends on the breakpoint, we switch the perspective of our new instance of Eclipse to the Debug Perspective. With the Debug Perspective activated, we have a new instance of Eclipse that has the correct state for our tests to

be run.

For our cosmetic changes, we had to retrieve what is being displayed in the Variables View and compare it to what we expect to be there. To retrieve what is being displayed, we go through several layers of the Eclipse API to get to the actual content. The content itself is normally stored in a tree that we will traverse to do comparisons. With Junit, we thought we would be able to create tests that automatically check to make sure the comparisons are accurate. Also, since Junit is integrated with the Eclipse Java IDE, we could quickly run the tests and see visual results.

With the help of Danny Dig we tried to navigate the Eclipse GUI to find the variables view in order to make sure the changes we had made were there. Danny provided us with some code that theoretically would provided us access the current variables view for Photran. This proved to be much more convoluted than first thought. We were able to get to the variables view but in our case there were no variables there for use to check against. The reason we thought that the variables we needed were not there was due to an improper initialization of the Eclipse. There was a lot we still did not understand about the Eclipse/CDT/Photran framework. Also, in order to test certain functions of Eclipse/Photran we needed other variables/objects to test them requiring us to built almost the complete program to test it instead of being able to test it piecemeal. Furthermore, we had already spent too much time on automated testing to spend more time on it. Because the framework was not fully understood due to its vast size, we would have had to spend much more time to delve into the framework before we could retrieve the correct variables view information for our automated testing.

Manual Testing

While we were unable to do automated testing we understood that some form of testing is still necessary. In order to maintain a sanity check for the changes we made in the debugger we have written instructions to manually test the system. The test is provided with this document in the appendix. It is our hope that in the future test can be automated as this would be a more efficient way to track changes and test the system. To this end the test Fortran program that is provided with this report will still be useful as it is an attempt at creating all known Fortran variable types to make sure that their information is being displayed properly in the debug perspective.

Non-technical Problems

Client Location

Since our clients were not in state certain parts of the project were quite difficult. Unlike other teams, we were not able to talk face to face with our clients and had to rely on e-mail and phone calls to communicate. The distance made the project more difficult because we could not ask questions of the clients in a timely manner. Since our project has a strong visual component to it, the visual problems and intended fixes were difficult to convey through email or on the phone. Also, our clients were a smaller part of the decision making process since our interactions with them were infrequent. A lack of constant interaction due to physical separation may lead to miscommunication or problems later if we are not careful to get detailed instructions.

Lack of Documentation

In the beginning of this project, we were provided with a large code base to work on, however, we were not directly given any documentation to go with this code. As we began to search for documentation we realized it was hard to find any. It was because there wasn't documentation dealing directly with what we were attempting to do. However, there was a lot of documentation on subjects related to what we wanted to

accomplish, such as documentation on debuggers in general or on eclipse plugins. Also, the CDT debugger lacked documentation that explained its structure and organization. This made learning the frameworks slower and much more difficult.

Future Development

Source lookup on Windows

When assessing the usability of the Photran debugger at the beginning of this project, we ran through some typical use cases to try and find places where improvements were necessary. Many of the deficits we found were already laid out in the requirements requested by our clients. Surprisingly, we found one seemingly glaring omission. This omission was that the source-lookup did not work. The source-lookup utility is used to highlight the actual line of code you are at while debugging a program's execution. Without source lookup, breakpoints and line-by-line stepping only tell you a line number, which makes it much harder to determine useful information.

After talking to Jeff, and doing some deductive reasoning about why the client's hadn't reported this problem, we decided that this problem only occurred when running Photran on Windows. Later, through an accidental discovery, we found out that the problem with source-lookup in Windows was actually in CDT, and not Photran. There is a workaround available using some built-in eclipse functionality, but actually integrating a solution into the Photran debugger would be very helpful.

During the course of our project, we discovered some quirks in the way that file

locations were handled. The Photran debugger uses Linux commands such as `pwd` to get current file information. On Windows with a Cygwin base, this returns a path such as `“/cygdrive/c/workspace/...”`, but then we try to use this path with Windows functions to actually retrieve the file. These retrieval functions expect the Windows form of `“c:/workspace/...”` for a file path and thus don’t work with the form we give it. Our solution in a different section of our project, that is applicable to this situation, was to do a string replacement whenever `“/cygdrive/”` was detected at the start of a file path to change it into the canonical Windows form.

Unfortunately, due to more pressing issues, we were unable to allocate sufficient time to implement this fix. This problem was bumped down our list once the temporary fix was discovered. Also, the clients list of issues was given precedence.

Variables View Icons

Currently, the variables have a small blue `“(x)”` image next to their name. This icon could be any image and is often used in other development environments to signify the type or scope of a variable. In the future, it would be helpful to have type information reflected in the variable icon. To do this, the origin of image must be determined and, at that point, type checking must be done with the appropriate image being returned based on type.

At first, we did not implement this change because we did not have a reliable way of getting type information. Getting type information became easier when we received Jeff’s symbol table code, but due to time constraints at that point, we were still not able to use it for this purpose. In the future, a developer could use Jeff’s symbol table

to easily derive the type for a variable and then assign it an image. Our upper and lower bound derivation code could be used as a guide for getting type from Jeff's symbol table.

Things We wish we had known

Fortran

Originally, we thought we would not have to learn Fortran. As we worked on the project, we learned that we would have to learn a little just to understand our clients and their requirements. In particular, we had to learn the differences between C and Fortran so we could address how GDB was failing Fortran development. Initially, we did not understand the requirements because GDB worked fine for C development. After we learned some Fortran we understood how GDB needed to be changed for Photran.

Eclipse Framework

The framework for Eclipse is very large and robust which makes it very powerful, but at the same time hard to learn. It is difficult to know when to call objects directly from Eclipse due to all the plugins. There are times when they should be called from the classes that inherit them within the plugins. This is a difficult problem that we have faced a few times and is most relevant to testing.

Junit UI testing

A major concern for this project is to establish a foundation for future development. We are the first group working on the Photran debugger but there will be many others. Therefore, we need to establish a solid testing plan to ensure a quality product. The solution we originally chose involved creating Junit tests to make sure certain features are working as intended. Ideally, we wanted a suite of tests that can be run and, if they succeed, give the developer confidence that everything is working.

The Eclipse framework provides an API for all of its components that we wanted to use for testing. Theoretically, anything that is shown in the UI and any interaction with the UI can be simulated in code. In theory, several steps need to be taken before we get to a place where we can test the debugger. First, we need to spawn a new instance of Eclipse with the Photran plugin loaded. Next, create a project in the workspace and import a test Fortran program into that project. Then we insert a breakpoint and build the program. After building the program, we run until it reaches the breakpoint. Once it suspends on the breakpoint, we switch the perspective of our new instance of Eclipse to the Debug Perspective. With the Debug Perspective activated, we have a new instance of Eclipse that has the correct state for our tests to be run. This however did not work because we did not have a full understanding of the Eclipse framework and Junit UI testing. We had worked with Danny Dig on this scenario. While he did help us, our problem was too complex with this time constraints and we were not able to access a populated variables view to perform our testing.

For our cosmetic changes, we need to retrieve what is being displayed in the Variables View and compare it to what we expect to be there. To retrieve what is being

displayed, we wanted go through several layers of the Eclipse API to get to the actual content. The content itself is normally stored in a tree that we will traverse to do comparisons. With Junit, we would have been able to create tests that automatically check to make sure the comparisons were accurate. Also, since Junit is integrated with the Eclipse Java IDE, we could have quickly run the tests and seen visual results.

Conclusion

While this senior project semester is ending this project is not yet over. There is much work left to be done and many features left unimplemented or unfixed due to time constraints or knowledge constraints. This project has shown us the difficulties of working with a framework that has very little documentation. We very clearly see the need for clear and concise documentation. We hope that we have performed to our clients satisfaction and that they are pleased with our accomplishments. We have paved the road for future work to be done. The problems we have faced in terms of learning a new framework, figuring out how we want to make our changes, figuring out where we want to make our changes and tackling the problems we face in order to implement our features have been a immensely rewarding learning experience that we can take beyond this class. The most rewarding thought about having worked on this project is how necessary to its customers, some of which are us. When we started, the debugger was almost useless because you could not obtain information about certain variables, nor did the debugger provide an elegant way in order for the user to eliminate errors in their code.

Appendix: Developer Docs

Setup Eclipse to Develop for Photran

Install Cygwin

Before you can install Photran on windows you will need to do a few things. The first thing you will need to do is install Cygwin. Cygwin is a Linux-like environment for Windows. It consists of two parts. The first is a DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux API functionality and the second is a collection of tools which provide Linux look and feel. You can obtain Cygwin on their website which is <http://www.cygwin.com> Installation instructions for Cygwin can also be found here. The most important Cygwin package to install is the make package which is under the "Devel" category.

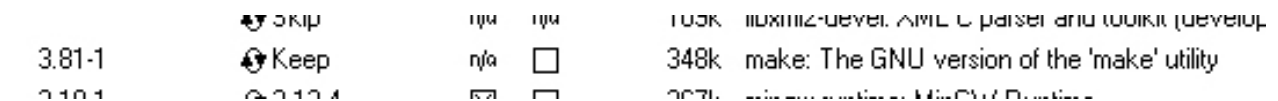


Illustration 13: Selecting make in Cygwin

Installing g95

Once Cygwin is installed we need to go get a Fortran compiler. The one that we developed with was the g95 compiler although alternatively the gFortran or other Fortran compilers could be used. G95 is a stable, production Fortran 95 compiler

available for multiple CPU architectures and operating systems. This package can be obtained on the g95 website which is located at <http://g95.sourceforge.net> Once you have installed g95 onto your system the next thing you need to do is add Cygwin to your class path.

Adding Cygwin to your Classpath

Right click on "My Computer" and go to properties. Click on the "Advanced" and then click on the "Environment Variables" button at the bottom. Highlight "PATH" in

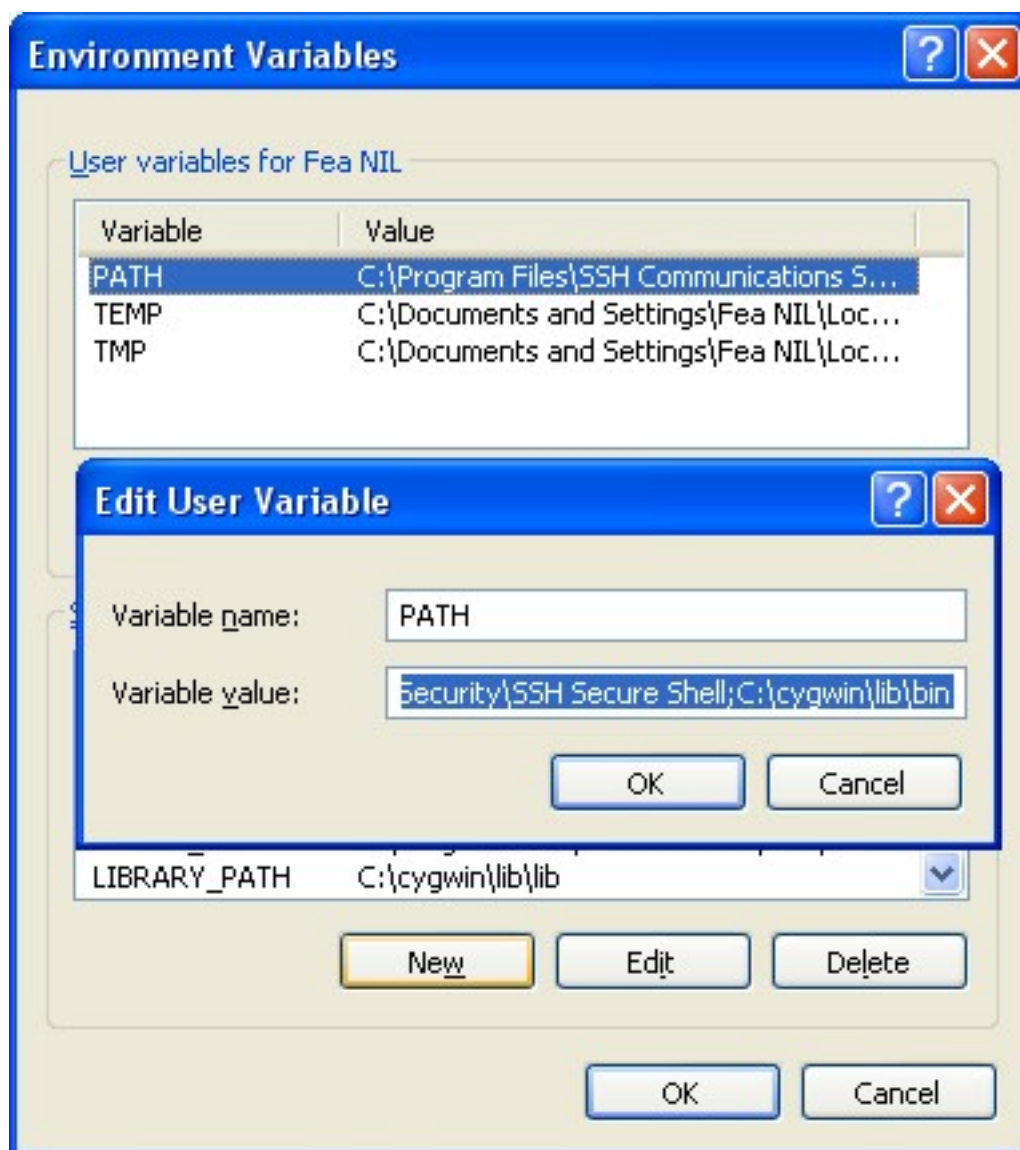


Illustration 14: Editing Windows Classpath

the top window and click the "edit" button. You want to add C:\Cygwin\lib\bin to the end of the text field with a space separating the previous text from what you are adding. This will break GAIM(Pidgin IM) if you use it due to dependency problems involving GTK.

Install JRE/JDK

Finally you need to make sure you have Java installed on your windows machine. Eclipse does not include a Java runtime environment (JRE). You will need a 1.4.2 level or higher Java runtime or Java development kit (JDK) installed on your machine in order to run Eclipse. After this is all done you are ready to install Eclipse.

Setting up Eclipse and Obtaining the Sourcecode

You can obtain eclipse from <http://www.eclipse.org/downloads> . The most current installation for Linux is listed there so you will need to click on "Other downloads for (current version number)". This will bring you to a page where you can download the appropriate windows installation package. Once you have downloaded and installed Eclipse it is time to get all the source for CDT and Photran. To get the source for CDT you need to open up Eclipse. In Eclipse, switch to the CVS Repository Exploring perspective. Right-click the CVS Repositories view and choose New, Repository Location. Enter the following information, then click Finish:

Host name:	dev.eclipse.org
Repository path:	/cvsroot/tools
Connection type:	pserver
Username:	anonymous

Password: (no password)

Expand :pserver:anonymous@dev.eclipse.org:/cvsroot/tools, and then expand HEAD in the CVS Repositories view. Expand org.eclipse.cdt-build. Under org.eclipse.cdt-build, right click and

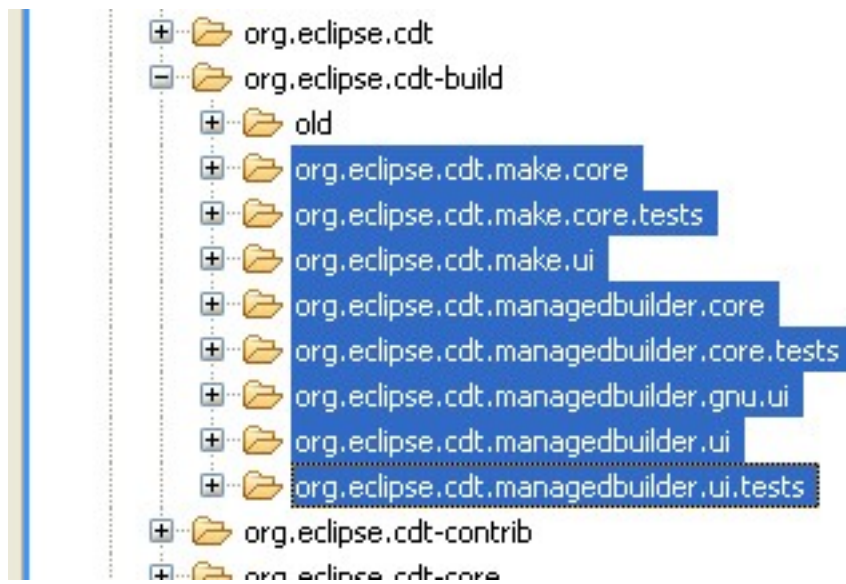


Illustration 15: Obtaining cdt-build Source

checkout all the org.eclipse.cdt.* packages. Do the same with org.eclipse.cdt-core, org.eclipse.cdt-debug, org.eclipse.cdt-doc, and org.eclipse.cdt-launch. org.eclipse.cdt-doc is a large download and is not necessary to run CDT. Feel free to download it when you have the time. After the checkouts are complete, you will have the CDT source code. Make sure it compiles successfully. There will be a lot of warnings, but no errors. Now it is time to get the Photran source.

To obtain Photran go to <http://slappy.cs.uiuc.edu/fall06/cs492/Group1/> and download the source for Photran with our latest code. We developed this code using Eclipse 3.2.0, CDT 3.1.2 and JRE 1.5. After you have downloaded the Photran tar file you

need to import it into Eclipse. Open the Eclipse workspace where you want to import the code, then go to File, Import. In this window expand the "General" node and select "Existing Projects into Workspace". Hit "Next" and select the "Archive File" radio button. Then click browse and find the Photran tar file you just downloaded. Hit "OK" when you find the file and then Hit "Finish" to complete. Make sure everything compiles and you are ready to use Photran.

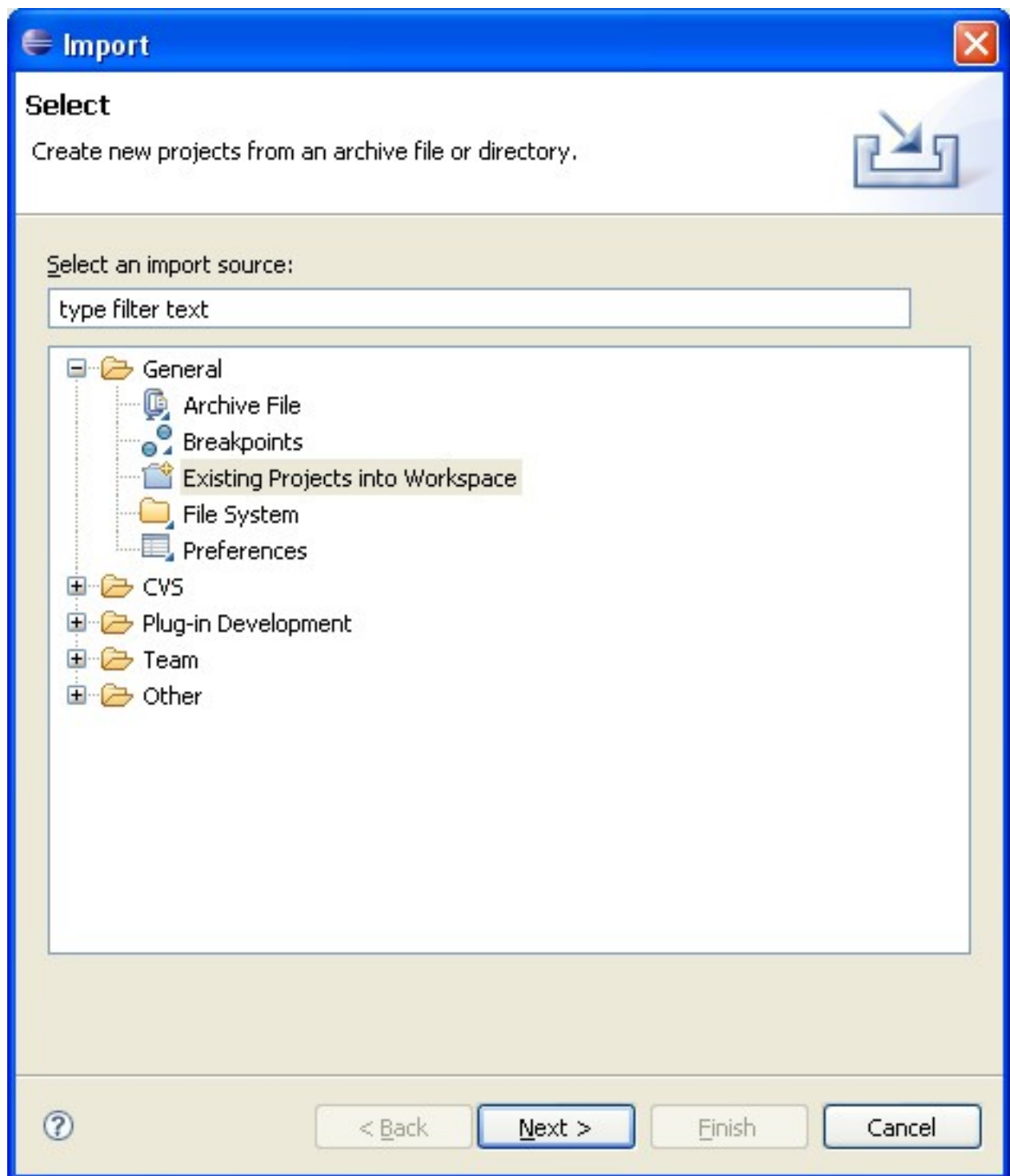


Illustration 16: Importing Photran Source

Windows Source Lookup with Cygwin

One of the problems that occurs when developing with Photran or CDT in windows is that source lookup in Photran does not work. This is due to a path mapping problem and can be fixed if and when the problem arises. On the screen when the source usually appears in the Photran debug mode, you will see a button named "Edit Source Lookup Path". Click "Edit Source Lookup Path". Then click "Add". Select "Path Mapping" and press OK. Select the new Path Mapping

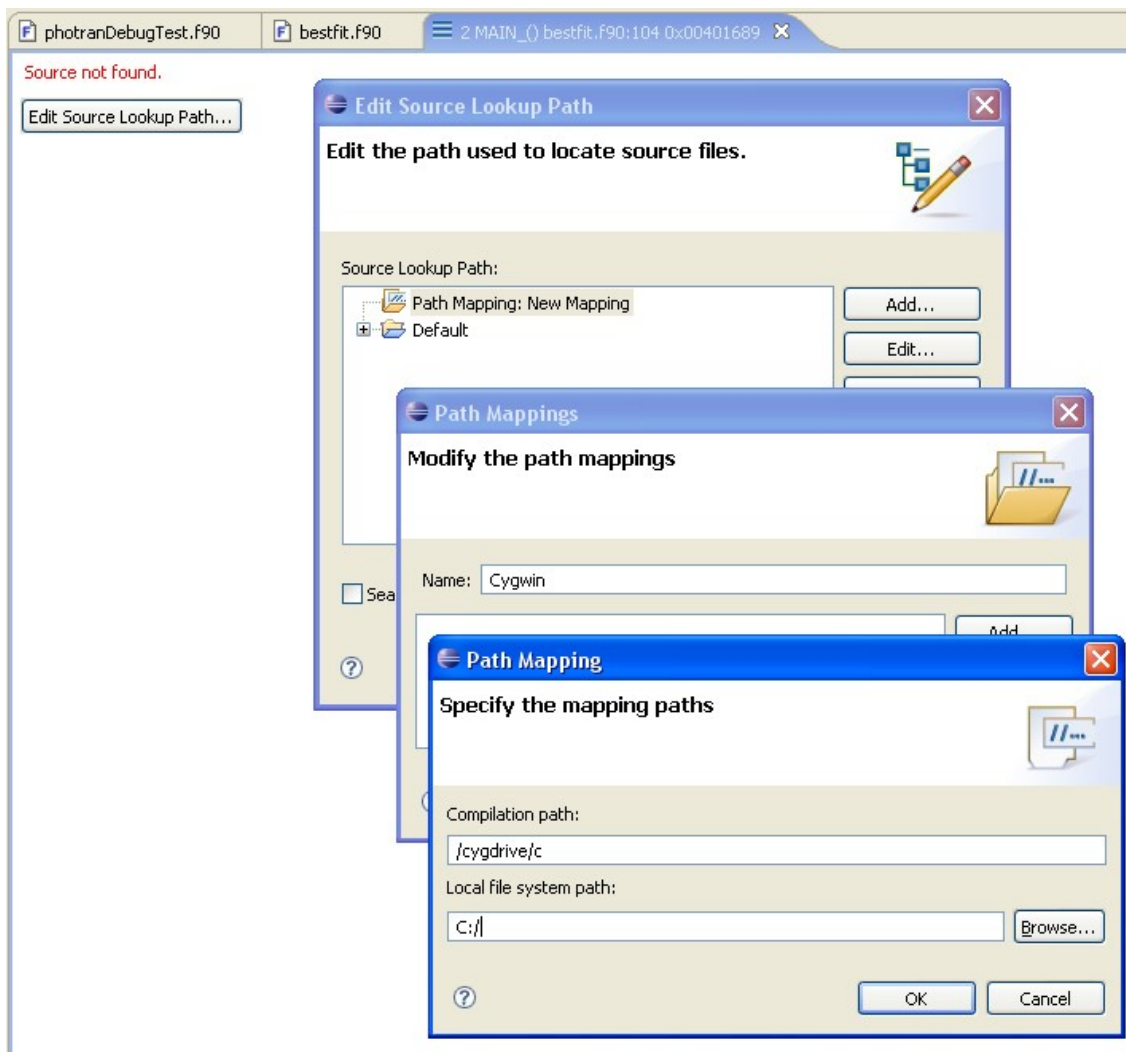


Illustration 17: Source Lookup Path Mapping

you just created and hit "Edit". This will bring up a new window for the Path

Mapping. Name it something relevant such as "cygdrive fix" and click "Add". Under Compilation Path enter "/cygdrive/c/" and under Local file system path enter "C:/". Once everything is entered hit OK. Hit OK two more times to close everything and it is fixed.

Manual Test

First get the source code that will be used for the debug testing. This can be found on our group website (<http://slappy.cs.uiuc.edu/fall06/cs492/Group1>) under the heading Photran Test Project. Open up Eclipse into the workspace which contains all your CDT and Photran source code. Check the Package Explorer view in the Java perspective to be sure that each of the following plugins are present and have compiled without errors. There will be many warnings, but do not worry about them.

```
org.eclipse.cdt.core
org.eclipse.cdt.core.aix
org.eclipse.cdt.core.linux
org.eclipse.cdt.core.linux.ia64
org.eclipse.cdt.core.linux.ppc
org.eclipse.cdt.core.linux.x86
org.eclipse.cdt.core.linux.x86_64
org.eclipse.cdt.core.macosx
org.eclipse.cdt.core.qnx
org.eclipse.cdt.core.solaris
org.eclipse.cdt.core.tests
org.eclipse.cdt.core.win32
org.eclipse.cdt.debug.core
org.eclipse.cdt.debug.core.tests
org.eclipse.cdt.debug.mi.core
org.eclipse.cdt.debug.mi.ui
org.eclipse.cdt.debug.ui
org.eclipse.cdt.debug.ui.tests
org.eclipse.cdt.launch
org.eclipse.cdt.make.core
org.eclipse.cdt.make.core.tests
org.eclipse.cdt.make.ui
org.eclipse.cdt.managedbuilder.core
org.eclipse.cdt.managedbuilder.core.tests
org.eclipse.cdt.managedbuilder.gnu.ui
org.eclipse.cdt.managedbuilder.ui
org.eclipse.cdt.managedbuilder.ui.tests
org.eclipse.cdt.refactoring
org.eclipse.cdt.refactoring.tests
org.eclipse.cdt.ui
org.eclipse.cdt.ui.tests
org.eclipse.photran-feature
org.eclipse.photran.debug-feature
org.eclipse.photran.intel-feature
org.eclipse.photran.refactoring-feature
org.eclipse.photran.cdtinterface
org.eclipse.photran.cdtinterface.tests
```

```

org.eclipse.photran.core
org.eclipse.photran.core.analysis
org.eclipse.photran.core.intel
org.eclipse.photran.core.tests
org.eclipse.photran.debug.core
org.eclipse.photran.debug.mi.core
org.eclipse.photran.debug.mi.ui
org.eclipse.photran.debug.ui
org.eclipse.photran.errorparsers.xlf
org.eclipse.photran.launch
org.eclipse.photran.managedbuilder.core
org.eclipse.photran.managedbuilder.gnu.ui
org.eclipse.photran.managedbuilder.intel.ui
org.eclipse.photran.managedbuilder.ui
org.eclipse.photran.managedbuilder.xlf.ui
org.eclipse.photran.refactoring.core
org.eclipse.photran.refactoring.tests
org.eclipse.photran.refactoring.ui
org.eclipse.photran.ui
org.eclipse.photran.ui.analysis

```

Right click on org.eclipse.photran.ui and select the "Run As" item and then select "1 Eclipse Application". Then wait. What you have done is started a runtime Eclipse application which has the CDT and Photran plugins built into it. Import the Photran Debug Test source code, by going to "File", "Import". Under the "General" node select "Existing Projects into Workspace". Chose the "Select archive file" radio

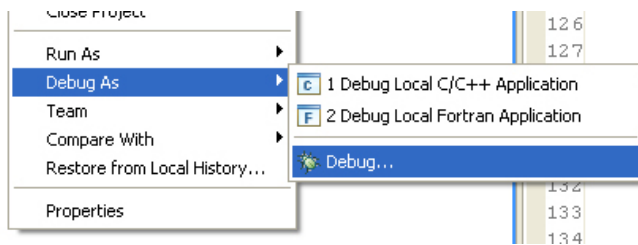


Illustration 18: Debug Menu

button and click browse to find the Photran Debug Test file you downloaded from our site. After this click "Finish".

Now that you have the Photran

Debug Test we are now ready to test the Photran Debug View using the Fortran test application. Right click on line 117 and select "Toggle Fortran Breakpoint" Right click on the project name ("photranTest") and go to "Debug As" sub-menu and select

"Debug...".

Under the "Fortran Local Application" node select "photranDebugTest". If it does not exist double click on "Fortran Local Application" and it should be created.

Be sure that the settings in the "Debugger" tab as the same as those displayed in the reference image below. All the fields in the "Shared Libraries" tab in the "Debugger" tab should be empty. Then click "Debug". If you are asked to confirm the perspective switch click "Yes". The source code for the photranDebugTest.f90 file displays

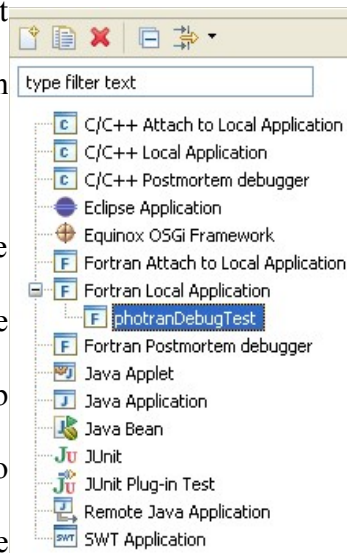


Illustration 19: Debug Launcher

highlighting line 117. Select the "Variables View" in the "Debug" perspective.

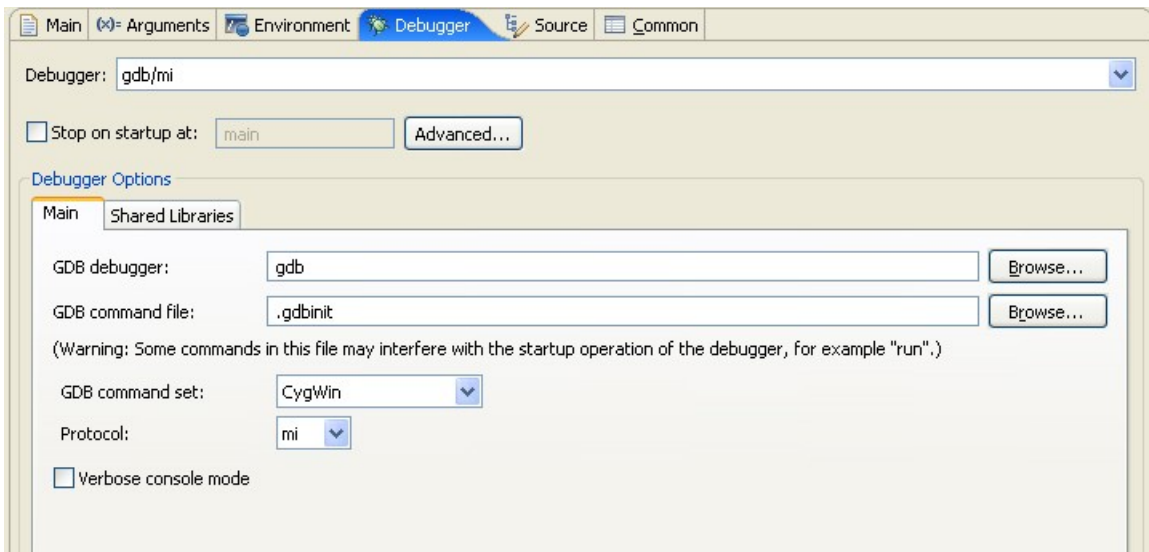


Illustration 20: Debugger Tab

Expand the "testmat2d" node. Check to see if there are any variables that have upper case letters in them anywhere in the variables view. Specifically U variables or SC variables. These should not exist. Then compare the results with that in the reference

image for the variables view results. Specifically make sure that the array bounds are the same, "teststring" displays as a string, the "testcharacter" holds the capital letter A, and that "testmat2d" displays the bounds for both dimensions, but a flattened version of the actual information.

Variables		Breakpoints	Modules	Registers
Name	Value			
(x)= testcomplexarr1d	(1:4)			
(x)= testublb	(3:5)			
(x)= testreal	4.5			
(x)= teststring	'This is a Test. '			
(x)= testrealarr1d	(1:4)			
(x)= testlogical	.TRUE.			
(x)= testmat2d	(1:4, 1:4)			
(x)= 0	1			
(x)= 1	2			
(x)= 2	3			
(x)= 3	4			
(x)= 4	2			
(x)= 5	4			
(x)= 6	6			
(x)= 7	8			
(x)= 8	3			
(x)= 9	6			
(x)= 10	9			
(x)= 11	12			
(x)= 12	4			
(x)= 13	8			
(x)= 14	12			
(x)= 15	16			
(x)= testlogicalarr1d	(1:4)			
(x)= testint	4			
(x)= testintarr1d	(1:4)			
(x)= testcharacterarray	(1:20)			
(x)= testcomplex	(1,1)			
(x)= testcharacter	'A'			
(x)= j	5			
(x)= i	5			

Illustration 21: Variables View

Right click on line 95 and select "Toggle Fortran Breakpoint" and then right click line 95 again and select "Fortran Breakpoint Properties". In the condition area enter the condition "i .eq. 3" and set ignore count to 0. Press "OK" to complete the conditional breakpoint. In the "Breakpoints" view you should see two breakpoints.

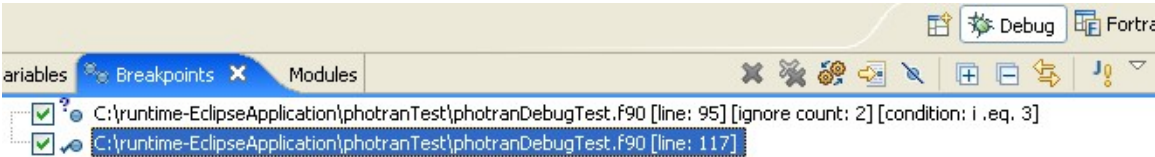


Illustration 22: Breakpoints

One of which has a question mark on it for the conditional breakpoint icon. Then rerun the debugger by clicking on the bug in the quick launch menu at the top of the screen. The debugger should

stop at line 95 and the variables view should match the reference image below. Particularly, i should be equal to 3 and "testintarr1d" should be half filled with index 2 and 3 containing garbage values.

Variables		Breakpoints	Modules	Registers
Name	Value			
(x)= testcomplexarr1d	(1:4)			
(x)= testublb	(3:5)			
(x)= testreal	4.5			
(x)= teststring	'This is a Test.'			
(x)= testrealarr1d	(1:4)			
(x)= testlogical	.TRUE.			
(x)= testmat2d	(1:4, 1:4)			
(x)= testlogicalarr1d	(1:4)			
(x)= testint	4			
(x)= testintarr1d	(1:4)			
(x)= 0	1			
(x)= 1	2			
(x)= 2	4007184			
(x)= 3	4007184			
(x)= testcharacterarray	(1:20)			
(x)= testcomplex	(1,1)			
(x)= testcharacter	'A'			
(x)= j	1			
(x)= i	3			

Illustration 23: Variables View During Conditional Breakpoint

Files of Interest

The framework for Eclipse/CDT/Photran is enormous and convoluted. There is very little documentation dealing with classes, their interactions in the system, and the flow of information. This made it very hard to try to learn the framework. Our primary mean of exploring the code was to use the Eclipse debugger to debug Photran's debugger. This way we could follow the information flow through the code and find when certain fields were populated. This eventually led us to find certain files such as CValue and MISession where the majority of our changes were made. We would watch the variables view as we stepped through code waiting for it to be populated. Then once we found a function that populated it, we would step into that function with the Eclipse debugger to figure out where it got its information and where it was being sent. This also led us to MISession and learning how CDT used MI to interact with GDB to debug. There are still a lot of things we do not know about the frame work. We mainly understand bits and pieces of it in which information pertinent to our work was being passed and manipulated.

CValue.java

The CValue.java file is located in the org.eclipse.photran.debug.core plugin in the org.eclipse.photran.debug.internal.core.model package. This file is where a lot of our changes were made. This is because each object in the variables view has an

associated CValue instance. This made making changes inside this file ideal in order to change what was being displayed in the variables view. This is where, we replaced brackets with parentheses, where we removed unnecessary U/SC variables, as well as where we used Jeff Overbey's symbol table in order to determine the array bound of any array that was explicitly created. We found that in terms of execution methods in this class were the last to be executed before data was actually displayed to the screen. Due to this fact, it was an ideal place to implement the features that affected the Debug perspective superficially.

MISession.java

The MISession.java file is located in the org.eclipse.photran.debug.mi.core plugin in the org.eclipse.photran.debug.mi.core package. An instance of the MISession is what is used in order to interact with GDB and translate Photran debugger commands into equivalent GDB commands and translate GDB data into useful objects from which Java can read data. As we had mentioned earlier, we had suspected that the underlying GDB did not have its interpreter language set to Fortran but rather to its default C/C++ in order to fix this we have an explicit gdb-set-language invocation that occurs during the creation of a Photran debug MI Session. The MISession is also where data is being retrieved from GDB and for a short period of time this is where we considered making the changes that eventually were made in the CValue.java file. This was because we thought that it would be a lot cleaner to delete unnecessary data such as U/SC variables before variable objects were created for them. This would also be a good place to reformat the variables so that they can contain proper type and bound information.