A Prototype System For Static and Dynamic Program Understanding

David P. Olshefski and Alan Cole

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

We describe a tool we call PUNDIT (Program UNDERstanding Investigation Tool), a prototype intended to serve as a vehicle for exploring and testing ideas in the area of program understanding. It combines static analysis information with information collected at runtime. We describe the architecture of PUNDIT and its two main components: the C source analyzer and a graphical user interface. We explain several of the views provided by the tool, including a high level structure chart, a dynamic call graph, a control flow graph animated during program execution, a type definition window, and others. By integrating static and dynamic information, the tool provides a more comprehensive understanding of a program as the first step to re-engineering or maintaining the application than can be obtained by static analysis alone.

Program understanding

As an early step in the process of re-engineering, maintaining, or extending an application, whether automatic or manual, it is necessary to “understand” part or all of the program[7].

It has been our observation that the use of static analysis alone may give an imperfect understanding of the program, and that it is often important to comprehend how the program behaves at runtime. This is especially true with multi-threaded, distributed, or parallel programs, and also with object-oriented programs, for in all of these cases it becomes increasingly difficult to figure out what is going on from static analysis alone.

Since the goal is to help provide aid for the programmer in what is a highly intellectual activity requiring significant amounts of human cognitive activity, a reasonable question is “what is comprehension and how does it work?” Psychological studies[1-3, 9, 11, 12] have attempted to provide information that could lead to theories of comprehension. Due to the nebulous and complex nature of the problem, most studies have focused on one task or aspect of programming and have attempted to show how a programmer gathers and utilizes information to achieve a more meaningful understanding of a program. Although various theories of comprehension have arisen from these studies, there is as yet no generally accepted view.

Lacking a firm theoretical basis, program understanding is currently at a point where the best approach may be to “try it out and use what works.” In this spirit, PUNDIT is a tool intended to allow us to explore the pragmatic question of how best to actually provide useful information to a programmer. It has been implemented, and has been used at this point by a few tens of users, whose feedback has been invaluable in trying to construct a tool which is both practical and useful.

PUNDIT architecture

The PUNDIT architecture is depicted in Figure 1. The right portion of the figure represents the familiar compile/link process; the left portion shows that PUNDIT is simply an extension to that process.

PUNDIT is composed of two programs, a C source code analyzer and a graphical user interface. The C analyzer, named PCC, accepts a C source file as input and produces a binary output file. This binary output file can be thought of as an extension to the debug information contained in the object module file. The sole purpose of PCC is to gather semantic information not provided by the compiler in the object module debug tables. In a sense, the function of PCC parallels that of the compiler and the binary file parallels that of the object module.

The trend in compiler construction is to provide more sophisticated and richer debug and analysis information
and, to the extent that this trend continues, the need for a separate analysis component such as PCC may disappear.

The graphical user interface, named PUNDIT, functions somewhat like a linker and debugger. It gathers information from the binary files, object modules, and executable file and attempts to present that information to the user in a useful manner. Neither the binary files nor object modules contain inter-module information except for external references. PUNDIT resolves these references as a linker would. Furthermore, the program may be executed under control of PUNDIT, providing insight into the program's dynamic behavior.

If some of the files from which PUNDIT collects information do not exist (for example, code being ported from another system may not yet compile successfully under the new system's compiler), then the type of information which can be displayed will be accordingly restricted.

![Figure 1. PUNDIT architecture](image)

Our approach uses only the collection of files produced by the analyzer, compiler, and linker. The architecture does not include a database. This runs counter to most approaches, including our own earlier work. We trade generality for improvements in performance: our earlier version, which used a relational database to store analysis information, ran an order of magnitude more slowly, used more space, and made it more difficult to update the analysis information.

The use of PUNDIT also fits well into the typical programming process. For example, a make file can be modified in a straightforward fashion to run the PCC analyzer whenever the source is recompiled. The binary files produced by the analyzer can often be conveniently managed by the same mechanisms used to manage object modules. Compared with the previous version of our tool, which required users to acquire, install, and become familiar with administering and maintaining a database, the current version is more responsive, less resource-hungry, and easier to use.

![Figure 2. Future architecture](image)

A direction that many compilers are taking is to expose the interfaces between intermediate stages within the compilation pipeline. Figure 2 shows how the current architecture would be expanded to accommodate new information or function provided by such a compiler. The resulting environment would consist of a loosely coupled set of tools providing specialized views. Information could be shared between tools but also between tools and the compiler. For example, the debugger could control execution of the program and distribute run-time information to the static browser. The performance monitor could modify one of intermediate forms in the compilation pipeline to cause instru-
C semantic analysis

The C semantic analyzer, about 2,000 lines of C, is based on a language parser generator similar to YACC. It accepts a grammar as input and produces an LALR parser which consists of a set of parsing tables and a driver. The lexer (tokenizer) used in our analysis is written in assembler and C, and consists of about 500 lines of code. Given the name of a C source file, the lexer reads and tokenizes the file and performs all necessary C language preprocessing—the parser just sees a stream of tokens.

The C analyzer produces two types of information in the output binary file. The first is semantic information about control flow and data use and the second is information required by the user interface for display purposes (such as column position of a variable on a particular line). Although the current analyzer only supports C analysis, the format of the binary file is language independent.

There is, of course, a trade-off between the amount of information computed at analysis time and the size of the output file. Our approach has been to keep the amount of information stored in the output file to a minimum, instead computing as much information as possible on demand in response to user queries. For example, the static single assignment representation of a program[8] is used in computing program slices. But rather than computing the SSA tables at analysis time, they are computed on demand. For functions of hundreds of lines, this can be done in less than a second even on a personal computer, so it makes sense to perform this function interactively. As a result, the size of the binary file produced by the analyzer is modest, typically about the same size as the object file produced by the compiler.

The following table shows the performance characteristics of the analyzer. All performance figures reported here were obtained on an IBM PS/2 model 90-0KD (a 33 mHz i486 processor) running OS/2 version 2.0. We analyzed a number of the sample programs shipped with the IBM WorkFrame/2 and Toolkit products for OS/2 2.0. For comparison purposes, we also show compilation times.

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Statements</th>
<th>Compile Time</th>
<th>Analysis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample01</td>
<td>912</td>
<td>572</td>
<td>8.0</td>
<td>2.3</td>
</tr>
<tr>
<td>touch</td>
<td>14,822</td>
<td>8,304</td>
<td>12.7</td>
<td>5.0</td>
</tr>
<tr>
<td>grep</td>
<td>16,168</td>
<td>9,073</td>
<td>14.2</td>
<td>7.3</td>
</tr>
<tr>
<td>hanoi</td>
<td>18,393</td>
<td>9,965</td>
<td>16.8</td>
<td>9.9</td>
</tr>
<tr>
<td>mahjong</td>
<td>21,088</td>
<td>12,288</td>
<td>22.9</td>
<td>15.8</td>
</tr>
</tbody>
</table>

Although the analysis is not incremental, it is fast enough to run as often as compilation. As expected, the analysis times are faster than the compiler, since PCC does not perform code optimization or code generation. On the other hand, the time required by the analyzer grows quickly with code size than compile time. For a program which does nothing but include all the OS/2 2.0 toolkit and compiler header files, the analysis runs in 7.6 seconds. As mentioned earlier, we hope future compilers will be able to provide all the necessary semantic information. This would make it unnecessary to support a separate analyzer (which somehow never quite seems to agree with the compiler on the exact language definition).

User interface: integrated code browsing and debugging

The user interface is a multi-threaded, multi-windowed OS/2 Presentation Manager application which attempts to integrate the functions of code browsing (static program understanding) and debugging (dynamic program understanding). It presents both textual and graphical views. The layout and display of both text and graphs are handled by existing "PM controls" (similar to X widgets), greatly simplifying the PUNDIT code itself, which consists of only about 15,000 lines of C.

The user interface was designed to be as language independent as possible. Most of the views presented to the user are language independent. The code in the user interface which reads the debug information from object modules and executables is dependent on the format of this information, which is also language independent in principle (though sometimes less so in practice).
Major functions

The following is a list of some of the major functions supported by the user interface:

- Perform program slicing
- Allow the user to graphically display data structures during debugging
- Create a dynamic call graph during debugging
- Graphically display the high level structure chart
- Graphically display the control flow graph for a function
- Animate the control flow graph and source code during program execution
- Display a textual list of symbols used in a function and their cross-reference
- Textually display the type definitions for all types used in a source file
- Graphically display the static relationship between type structures
- Display the source code (*.c or *.h)
- Textually display the include file nesting
- Display a textual list of the functions contained in each source file
- Display register values, storage values, set/clear breakpoints, etc.

In addition, there are help functions and an edit window for cutting and pasting text. All the information presented by the PUNDIT user interface can be captured and used out of the PUNDIT context (for example, included in printable documents).

The above functions are realized in PUNDIT via a set of windows, or views. Each view has its own set of functions that operate on the data in its window. Each view is also tied to other views, usually by functions which link the data in one window to corresponding data in another.

Some design approaches taken for the user interface were based on the following:

- Provide only a few, full function interactive views to the user
- Be general rather than task specific
- Do not assume a particular strategy for investigation by the user
- Display the relationships between objects graphically or textually, where appropriate
- Allow the user to click on anything visible
- Provide short path lengths between user actions and display of information
- Tightly-couple views so that a user action in one view results in appropriate changes to other views

- Provide as much view editing and filtering as possible.

It was not our intention to provide support for arbitrary, user defined queries.

In some ways, the PUNDIT tool is similar to the FIELD environment[10]. Like FIELD, it provides both static and dynamic information to the user. Like FIELD, it uses no formal database. But, in contrast to FIELD, PUNDIT is more tightly integrated. Both static and dynamic information is presented by the same process, eliminating inter-process messaging, which gives PUNDIT a performance advantage at a cost in openness.

PUNDIT views

In the following sections, we discuss a few of the views which PUNDIT offers and the functions which they support. Notice that each view falls into one of three categories:

- View provides static information
- View provides dynamic information
- View provides both static and dynamic information

High level structure chart

The high level structure chart (HLS) is a graph window. In the HLS, there are two types of nodes: one node type represents a function, the other, a global variable. Function nodes are green and square. Global data nodes are cyan and six-sided. (PUNDIT attempts, where possible, to use color as a redundant cue, so that its views will still make sense on a monochrome display.) Figure 3 shows a sample HLS.

There are two types of arcs in the HLS: directed red arcs which flow from function nodes to global nodes (or vice versa) and directed yellow arcs which flow between functions. The latter type occurs when one function (statically) calls another function (or itself). The set of all function nodes and all yellow arcs represents the static call graph.

The red arcs represent the relationship between functions and the globals that they access. A red arc flowing from a function node to a global indicates that the function modifies that global. A red arc flowing from a global node to a function indicates that the function accesses the value of that global. Arcs are labeled with a number which represents the actual number of connections between nodes, so that only one arc is
drawn between any two nodes, reducing visual complexity.

A large HLS, displayed in its entirety, is generally unhelpful as an aid in comprehending the overall structure of an application. Unfortunately, most real applications have a large HLS. Our approach to simplify the HLS is to provide the user a rich set of filtering capabilities which can be interactively applied to the graph.

When the HLS appears for the first time during a PUNDIT session, the user is presented with only those function nodes in the graph which are not called by other functions. In other words, only the root nodes of the call graph are visible to the user. From this point, the user can choose a number of options to grow or shrink the graph based on various characteristics.

The graph manipulation capabilities are extensive. Examples include the following: display only those functions from a particular source file; display only the immediate children of a function; display all paths leading to a function; remove (or include) all library functions from the graph; include (or remove) all global data structures.

One filtering capability which can be valuable is to display all functions which modify a global data structure, plus the calling relationship between those functions and their ancestors. This answers the question "who modifies this global variable and how can those functions be reached?"

In addition, the user can click on a function node and bring up its source code, control flow graph, type definitions, symbol cross-reference, or header file list. The user can also re-layout the graph on the fly, to view the HLS in either a top-bottom layout or left-right layout.

Of all the views in PUNDIT, this may be the most useful. This global picture of how functions and global data relate is too difficult to create manually, yet, in our experience, many programmers seek information at this level in order to gain a top-down, hands-around-the-

Figure 3. Portion of a high level structure chart (zoomed in)

97
system understanding of how functions and global data interrelate.

Dynamic call graph

The dynamic call graph window animates the calling relationships among the functions for an actual execution of the program. The window is empty until execution begins. Then, each time a function is called, a node representing it is added to the graph (if not already there). As each function is called or returns, its node changes color. Each arc is annotated with the number of times the call has been made to the current point in execution.

The user can enable or disable the dynamic call graph at any point in time, which allows the user to begin building the dynamic call graph after a breakpoint has been hit. The user can either execute the target program and watch as the dynamic call graph is built and animated or the user can build the dynamic call graph one function call at a time by single stepping at the function call level.

This capability may be useful in understanding the behavior of a set of functions. The HLS provides a static view of who calls whom, whereas the dynamic call graph shows the order of function calls as well as the actual number of times a function calls another.
Control flow graph

In PUNDIT, a control flow graph can be presented for each function, each graph having its own window. Each node in the control flow graph represents one statement and the arcs represent flow of control between statements. Arcs are labeled with the type of flow, if appropriate. An arc may be labeled as true, false, return, break, continue, default, goto or combinations of these labels. This was done to eliminate the need for statement nodes of those types, and helps to keep the graph simpler.

The nodes are labelled either with the statement type, or with the actual text of the statement (truncated to fit), as a user option.

The control flow graph view also has filtering capabilities. For example, the user can select a node in the graph and color all paths leading to that node, or all paths leading from the node. In a similar way, the user can select a node in the graph and reduce the graph to show only those paths leading to or from a node; other paths and nodes are made invisible. The user may also locate the source line of code that a node represents.

During a debug session, the user can select which control flow graphs to animate. As the target program executes a line of code, the associated control flow node changes color from green to cyan—the previously executed node’s color is changed from cyan back to green. This gives an animation of the execution thru the control flow graph whenever code from that function is executing.

Another option allows the user to color, and leave colored, all nodes that have been executed. When execution is complete, the user is left with a visual picture of code coverage for the execution.

Although the control flow graph is an important abstraction in program analysis, we question whether it is beneficial to display the graph itself to the pro-
Several of the early users of PUNDIT have commented that for large functions, the control flow graph is too big to understand unless it is highly structured (such as a huge switch statement). And small functions are understood more easily by reading the source code than by looking at the control flow graph.

In spite of these comments, we believe that animation of the control flow graph during debugging or testing may be helpful and hope to gain additional user feedback in this scenario.

---

**Figure 6. Type definition window**

The type definition window displays the definitions for all types used in the associated source file. This information is derived from the debug tables in the object module, and contains details such as structure field names, offsets, and lengths (information not contained in source header files).

Although this text window simply collects existing type definitions into one place, it is a popular facility. The user does not have to open and traverse a number of header files to find the definition. Although debuggers use this information to display variable instance data, they do not typically show the user all this information at once.
Type graph window

The type graph is related to the type definition window. It is a graph showing static relationships between data structures. Each node in this graph represents the type definition of one structure. The text in each node is the list of fields within that structure as well as the offset of each.

There are several types of arcs in this graph, each representing a different relationship that can exist between two structures. A yellow arc between two structures means that the first structure contains a pointer to the second (as a field). A red arc between two structures means that the first structure contains the second structure as a field (nested structure). A cyan arc between two structures means that the first structure contains an array of the second structure.

This view shows the static relationship between type definitions used in the program. It provides to the programmer a diagram of how data types relate to one another: the sort of diagram we have often observed programmers drawing by hand. Here, the diagram is created automatically and the data structures represented span multiple header files.
Data layout graph

The data layout graph is created during program execution, and shows the actual instance data of a symbol or structure. For example, most debuggers allow the user to show the contents of a variable. This view will graphically display the contents of a variable plus the contents of all the structures attached to that variable, plus the contents of all structures attached to them, and on on. The user can refresh the view after any break-point in order to see how data structures grow in storage at various points during program execution.

As shown in Figure 8, each variable or block is displayed with its address in storage and its value. Fields within structures are displayed as well.

This view can be thought of as the dynamic version of the static type graph mentioned above. The function view automatically performs is basically what programmers must go thru when attempting to get a mental picture of the data structures their programs are creating in storage.
Figure 9. Symbol list view

Symbol list

The symbol list is a text window showing the symbols used by a function. Each function has its own symbol list window. The user may select a symbol and perform various functions. For example, the user can color all instances of a symbol within a function, or display the symbol's type. The user may also display each line of source code that references the symbol.

Figure 9 shows an example of this window. The symbols are listed against the left margin. The cross-reference for a symbol expands below the symbol name and is indented. The user may hide or show the cross-reference information on a symbol-by-symbol basis.

Source code

One source code window can be brought up for each source file analyzed. The user may select in the source window and perform various functions. Setting/clearing breakpoints, coloring all paths to a particular statement, displaying a symbol's cross-reference information, locating a statement in the control flow graph, identifying the reaching definitions for a variable or its following uses, are all examples.

In addition, "program slicing" has been implemented in the source code view. Program slicing is a method of reducing a program to a smaller subprogram which behaves in the same way as the original program for a given set of variables at a given statement[13]. This vertical-thread like partitioning of programs may aid comprehension by reducing the size of the program the user needs to understand.

Slicing has been implemented by allowing the user to click on a line of source code with the mouse, and then to specify the variable or variables referenced in that
statement to be used in building the slice. The resulting slice, shown in the source code by coloring each line of code in the slice, consists of all those lines of code in the original program which affect the value of those variables at the chosen statement.

The source code view is a read-only text window. Ideally, the system would allow the user to edit the source, re-analyze the changes and synchronize other views with the updated information. These capabilities were beyond what was intended for the PUNDIT prototype.

Function index and header file list

The function index is a list of all the functions contained in each source file—a table of contents of each source file. Like all other PUNDIT views, the user can gain access to the control flow graph, source code, symbol list and type information from the function list view.

The header file list is a list of the header files included during compilation (one window per source file). The fully qualified path for each header file is shown. The list is indented to reflect the nesting depth of each include file and the order of the list is the order in which each file was included. Access to the header file source code is provided by simply clicking on the filename within the list. Though this view is conceptually simple, it has been found to be very useful by programmers working on applications which pull in huge include file hierarchies (such as Presentation Manager or X Window System applications).

Figure 10. Function Index and header file list

104
Logging

In order to collect information about how PUNDIT is used, the tool was augmented to log all user actions. The result is a data file, an example of which is shown in Figure 11. The *time* column gives the time of day when the user performed an action, the *dif* column shows the time elapsed since the last action.

This data was used to analyze user interactions with an earlier version of the tool incorporating only static analysis[4-6], but has not yet been used to analyze the current version combining static and dynamic analysis.

<table>
<thead>
<tr>
<th>time</th>
<th>dif</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>16:16:25</td>
<td>3.44</td>
<td>window created: Program Understanding Environment</td>
</tr>
<tr>
<td>16:16:32</td>
<td>5.34</td>
<td>window created: Call Graph</td>
</tr>
<tr>
<td>16:16:43</td>
<td>15.04</td>
<td>window created: Control Flow Graph: ensureProjectListVisible in PUTILE.C</td>
</tr>
<tr>
<td>16:16:53</td>
<td>4.88</td>
<td>resized: Control Flow Graph: ensureProjectListVisible in PUTILE.C and...</td>
</tr>
<tr>
<td>16:16:55</td>
<td>2.44</td>
<td>Hide window: Control Flow Graph: ensureProjectListVisible in PUTILE.C...</td>
</tr>
<tr>
<td>16:16:57</td>
<td>2.25</td>
<td>window created: Control Flow Graph: processMenuSelection in PUTILE.C</td>
</tr>
<tr>
<td>16:17:02</td>
<td>4.31</td>
<td>Hide window: Control Flow Graph: processMenuSelection in PUTILE.C time.</td>
</tr>
<tr>
<td>16:19:22</td>
<td>2.47</td>
<td>Show window: Control Flow Graph: listWindowProc in PUTILE.C time hidden:</td>
</tr>
<tr>
<td>16:19:24</td>
<td>1.88</td>
<td>Hide window: Control Flow Graph: listWindowProc in PUTILE.C time visible:</td>
</tr>
<tr>
<td>16:19:25</td>
<td>1.31</td>
<td>scroll line down -&gt; Listing: PUTILE.C</td>
</tr>
<tr>
<td>16:19:27</td>
<td>2.19</td>
<td>scroll line down -&gt; Listing: PUTILE.C</td>
</tr>
<tr>
<td>16:19:37</td>
<td>9.06</td>
<td>Hide window: PUNDIT.EXEProgram Understanding Environment time visible: 189.88 secs</td>
</tr>
<tr>
<td>16:19:37</td>
<td></td>
<td>Close PUNDIT main window</td>
</tr>
</tbody>
</table>

Figure 11. Event log for user session

Conclusions

We have presented what we feel to be a good architecture for a program understanding system. It is performance oriented and has few dependencies. It fits well with the current build process and takes advantage of existing compiler capabilities. It also demonstrates our belief that a good way to make program understanding tools available to the programmer is by integrating them more closely with debugging capabilities and with the compiler itself.

We have also described a variety of program understanding functions which we feel can be of use to a programmer. Although our list is far from complete, we believe it indicates the usefulness of combining static and dynamic information into a single tool.

One of the problems still to be faced is learning more about what information helps most in the re-engineering and maintenance processes. Another big issue is in handling magnitude of scale questions. Beyond a certain size, a call graph more resembles the Milky Way as seen from Earth than it does a useful aid to program comprehension. The filtering and graph manipulation capabilities incorporated by PUNDIT are an attempt to address the scale issue, but we feel much still remains to be done in this area.

Acknowledgements

The work described here owes much to a number of our colleagues, past and present. The PUNDIT work was originated by Linore Cleveland and Tom Corbi. The parser (LPG) we used was developed by Philippe Charles and Gerry Fisher, and we thank Jim Christensen for the use of his lexical analyzer. We also acknowledge Vance Waddle for his graph layout and display package, and the IBM Lab at Cary, NC for their text control.

References


