An Evaluation of Rational PurifyPlus

Jameel Gbajabiamila
Animesh Kejriwal
Yash Patodia

Table of Contents

1 Introduction

IBM Rational® PurifyPlus is a runtime analysis solution designed to help developers write reliable code faster. Runtime analysis in this package includes four basic functions: memory corruption detection, memory leak detection, application performance profiling, and code coverage analysis. In this report we will evaluate Rational Purify, Quantify and PureCoverage from a usability point of view.

2 Installation

2.1 On Windows

The evaluation version of Purify for windows was found easily on the website. The installation was in the form of the "installation wizard" and was straightforward. However, Purify comes with a license administrator, which requires an evaluation license for Purify to work. The Rational website mentioned that a 30 day evaluation license is available for download. However, we could only find a 15 day license for PurifyPlus on the website, and were forced to use that. After importing the license, Purify gave a cryptic error with a short message saying that the license was not found. Understanding the cause of the error took a couple of hours, and a small tweak in License Administrator solved the problem. The tweak that had to be made was to ask the license administrator to launch PurifyPlus instead of Purify, since the key was for PurifyPlus. This was somewhat unexpected since PurifyPlus was not installed on the computer. We installed PurifyPlus after performing our experiments on Purify. The installation of PurifyPlus was hassle-free.

2.2 On Linux/Unix

We were unable to install PurifyPlus on Linux and UNIX. This was primarily because of problems with the software. The IBM Rational website had broken links and therefore we could not download the 30 day version of Rational Purify for Linux/Unix. We were forced to use the PurifyPlus package which is very large and has a shorter license. Moreover, the Linux version the tool supported was incompatible with the latest version of Linux available on-campus. Having spent a lot of time and effort on trying to install the tool on both Linux and UNIX, we finally decided to stick to the Windows version.

3 Rational Purify

IBM Rational Purify ensures reliability via two crucial functions: memory corruption detection and memory leak detection. Rational Purify packages support for these two runtime analysis capabilities in a single product with a common install and licensing system.

3.1 Expectations

Large development projects have multiple versions of code, written by a large number of developers, often using a variety of third-party tools. Debugging such complex programs for memory bugs is a very difficult task. It is expected that Rational Purify will help rectify this problem by allowing a user of the tool to quickly locate memory related bugs which exist in the code. It is hoped this will provide clear information to aid the developer in identifying memory related bugs in the code.

3.2 Memory leaks and Other memory bugs

Memory access errors, such as **array-bounds errors**, dangling pointers, uninitialized memory reads, and memory allocation errors, are among the most difficult to detect. The symptoms of incorrect memory use typically occur far from the cause of the error and are unpredictable, so that a program that appears to work correctly really works only by accident.

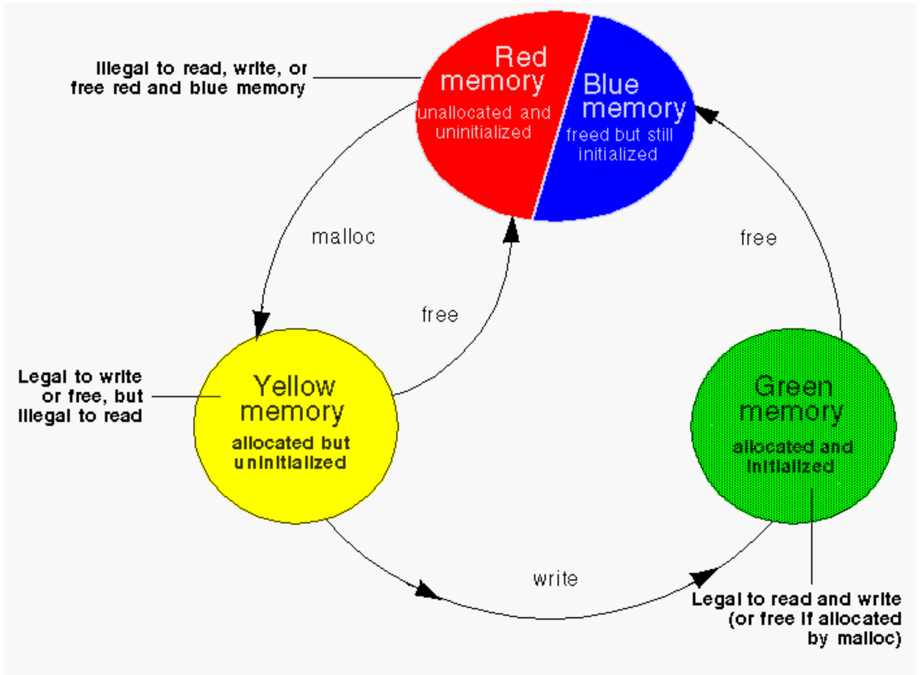
Array bounds errors - Purify inserts guard zones around statically and dynamically allocated memory to catch this type of access error. Purify reports an **array bounds read (ABR)** or an array bounds write (ABW) message at the time it detects the error.

Accessing through dangling pointers - Purify tracks freed memory and reports invalid memory accesses as **free memory read (FMR)** or free memory write (FMW) errors at the time the errors occur.

Uninitialized memory reads - Purify tracks new memory blocks as they are allocated and reports any attempt to read or use a value from the block before it's initialized as an **uninitialized memory read (UMR)** error.

Memory allocation errors - Purify intercepts all calls to memory allocation API functions such as malloc, new, new[], calloc, realloc and related functions, to warn you about their incorrect use. For example, when you use an incorrect function to free memory, such as calling free on memory obtained from new, Purify generates a **freeing mismatched memory (FMM)** message.

3.3 How Purify Works


Illegal to read, write, or free red and blue memory
Red memory
unallocated and uninitialized
Blue memory
freed but still initialized
malloc
free
free
Legal to write or free, but illegal to read
Yellow memory
allocated but uninitialized
Green memory
allocated and initialized
write
Legal to read and write (or free if allocated by malloc)

Purify labels memory states by color.

Purify monitors every memory operation in a program, determining whether it is legal. It keeps track of memory that is not allocated to your program, memory that is allocated but uninitialized, memory that is both allocated and initialized, and memory that has been freed after use but is still initialized.

Purify uses a table to track the status of each byte of memory used by your program. The table contains two bits that represent each byte of memory. The first bit records whether the corresponding byte has been allocated and the second bit records whether the memory has been initialized. Purify uses these two bits to describe four states of memory: red, yellow, green, and blue.

Purify checks each memory operation against the color state of the memory block to determine whether the operation is valid. If the program accesses memory illegally, Purify reports an error.

- *Red:* Purify labels heap memory and stack memory red initially. This memory is unallocated and uninitialized. Either it has never been allocated, or it has been allocated and subsequently freed. In addition, Purify inserts guard zones around each allocated block and each statically allocated data item, in order to detect array bounds errors. Purify colors these guard zones red and refers to them as *red zones*. It is illegal to read, write, or free red memory because it is not owned by the program.

- *Yellow:* Memory returned by malloc or new is yellow. This memory has been allocated, so the program owns it, but it is uninitialized. You can write yellow memory, or free it if it is allocated by malloc, but it is illegal to read it because it is uninitialized. Purify sets stack frames to yellow on function entry.

- *Green:* When you write to yellow memory, Purify labels it green. This means that the memory is allocated and initialized. It is legal to read or write green memory, or free it if it was allocated by malloc or new. Purify initializes the *data* and bss sections of memory to green.

- *Blue*: When you free memory after it is initialized and used, Purify labels it blue. This means that the memory is initialized, but is no longer valid for access. It is illegal to read, write, or free blue memory.

Since Purify keeps track of memory at the byte level, it catches all memory-access errors. For example, it reports an uninitialized memory read (UMR) if an int or long (4 bytes) is read from a location previously initialized by storing a short (2 bytes).

3.4 Experiments

Our main goal was to understand the effectiveness of memory leak detection by Purify. Purify can be run on programs with or without source code. Having the source code helps the tool pinpoint the location of the error. When run on programs where the source code is unavailable, Purify tells the user which libraries or files have problems. This is very useful when creating applications that use closed source libraries or third party applications.

Purify integrated seamlessly with Microsoft Visual Studio 6.0, by providing a toolbar in the IDE that lets one run Purify directly from Visual Studio. Our first step was to run Purify on a simple application and understand how the tool worked. Then we tested Purify on a more complex program, and compared the time taken by the tool to find the bug to the average time taken by our classmates to identify the same bug.

3.4.1 Simple Experiment

We conducted this test to get familiar with the tool, and to determine the user-friendliness of the results. We created a simple "Hello World" program, where we allocated memory and did not free it, as shown in the code below.

```
1.    int main(int argc, char* argv[])
2.    {
3.    char word[] = "Purify";
4.    char *x = (char*)malloc(strlen(word)*sizeof(char));
5.    strcpy(x,word);
6.    printf("Hello World! %s ",x);
7.    return 0;
8.    }
```

Purify successfully recognized two errors:

1. Array bounds read: This error is seen in line 5, since the memory required in variable 'x' to store "Purify" is one more than the length of the string to accommodate for the '\0' character denoting the end of string. In the example above, variable 'x' has only 6 bytes, but the word needs 7 bytes to be stored. Hence reading the last byte gives an array bounds read error.
2. Memory leak: Purify also identified the memory leak caused by not freeing variable 'x'. Purify pointed to line 4, to indicate that the malloc at line 4 was not freed.

The “Error View” window provided a summary of the errors found in the program. This was in the form a tree, and the error of interest could be quickly expanded to find the details of the error. We had absolutely no difficulty understanding the nature and source of the error.

3.4.2 Complex Experiment

We conducted a study by comparing the time taken by Purify to instrument the program and find the error, to the average time taken by a fellow Software Engineering student to find the same. It was necessary to choose a relatively simple but long program for this study, because we wanted to avoid having the subjects spend time trying to understand complex code, and at the same time did not want to test them on a trivial program.

For this test, we used a C program from a previous freshman level course. This was a simple 500 line program that allowed operations on a dictionary. We were fairly confident that we could find some genuine memory errors in that program. We ran Purify on the program, and discovered that there was indeed a memory leak. The nature of the leak can be seen by looking at the following snippets of code:

```
struct Node
{
  char *st;
  struct Node *next;
}NodeT;
```

The struct declared above is a data structure for a simple linked list, which holds a string value inside it. When inserting strings into this data structure, the “char *st” is allocated memory by calling “calloc”.

```
while(temp!=NULL)
{
      curr = temp->next;
      free(temp);
      temp = curr;
}
```

The snippet of code shown above is executed when the program ends. It tries to free all the memory that is allocated to the program. We can see that though the loop successfully frees the memory allocated to the node, it does not free the memory allocated to the string within the node. This causes a memory leak.

We asked a total of 6 people to participate in this experiment, each having approximately two to three years of experience with C/C++ programming. The students were explained the basic functionality of the program, to ensure that they spend time analyzing the code rather than understanding it. Out of these, one student did not find the error, while the other students took seven minutes on average to find the error. This highlights the usefulness of the tool. In fact, for larger applications, analyzing code manually would be a lot more cumbersome and the increase in time required would be exponential.
One must note that Purify pointed at the program line where the string that was not freed is allocated memory, while the classmates identified the area where the string *should* be freed. Thus, identifying the area where the memory leak could be fixed took another 45 seconds for this program.

Time taken to debug using Purify	Avg. time taken by subject (MSE Classmate)
12 seconds (Purify overhead) + 45 seconds (to fix error)	7 minutes

Overall, running this program with Purify was extremely easy. Since Purify integrated with the IDE, running Purify on the program was just a click of the button. Purify takes a few seconds to instrument the necessary files and libraries, and then the program executes normally, but relatively slowly, since Purify running in the background. Purify tracks the errors caused by the program as the program is running. Purify provides details of the error such as the line number (if source code is available), the actual program line that caused the error, the functions in the call stack for that program line, and details about the type of error. Our overall experience with Purify was great, and we feel it is one of the most helpful tools available in the industry to find memory errors effectively.

4 Rational Quantify

Rational Quantify provides information about the performance of code during execution. It offers functionalities to look at performance statistics in various ways, and makes it easier to pinpoint areas of the code that are most likely to increase performance.

Quantify runs in the background once the program starts, and provides performance statistics of the program. It allows the user to customize the way in which data is collected – whether it is for a particular portion of the code or for the entire code execution. It provides ways for the user to compare the change in performance before and after a change is made. This allows user to analyze the impact of their changes. The most useful part of Quantify is that it allows the user to locate areas which have the highest potential for improving performance. Moreover, the statistics provided by Quantify do not include the overhead of running the program with it. It represents the time taken by the program without Quantify. Quantify instruments all the code of a program, whether the code is available or not. This makes it possible to use Quantify on programs that use closed source libraries and applications.

4.1 Expectations

Performance is often a quality attribute of high importance in many industry applications. Lots of servers, mission critical software and business applications require a guarantee of performance. For products that require such high performance goals, it is necessary to analyze the program and identify areas have the maximum potential to increase performance. The performance data should be available in various different metrics, such as number of function calls, time spent in functions, functions which have source code available (and hence can be optimized), etc. This data should be available in user-friendly representations such as charts and graphs. The goal of using such a tool is to maximize productivity by minimizing the time spent in identifying areas that can be optimized and maximizing the time spent in making the actual optimizations.

4.2 Learning Curve

The learning curve for using Quantify was very low. Quantify integrated with Microsoft Visual Studio 6.0 in the form of a toolbar. To run Quantify on a program, we simply had to click on a button that engages Quantify, and then run the program. Quantify produces various tables and call graphs that can be used immediately to analyze the performance of the program.

4.3 Features

Quantify offers various ways to check the performance of an executed program. We found the following features provided by Quantify particularly helpful:

1. Call graph: Quantify provides a call graph of the various functions in a program. It connects the vertices of these graphs by edges that indicate function calls. The edges are of varying degrees of thickness and represent the time spent in that area by the program. Furthermore, Quantify presents some useful features that help analyze various properties of the program. For instance, it has a feature that highlights the top ten functions where the program spends most of its time or the functions which have their source available, and so on. All these are in an extremely intuitive user interface and the visual representations are very useful.

2. Function list: This view provides a list of all the functions in the program, along with various performance statistics. The columns we found especially useful were:

   - The time spent only in a function
   - The time spent in a function and its descendants
   - The number of calls made to a function
   - The module and source file of a function

   Also, this view provides an easy way of sorting the columns, changing precision and restricting the number of functions shown, which makes determining areas for optimization very easy.

3. Annotated source: We found this view to be extremely user-friendly. Users can look at the source code, and see the amount of time spent at each line of code. Quantify provides a summary of each function at the top of the function. Then it provides performance data for each line in the code. This helps a lot to pinpoint areas of potential optimization. Moreover, it makes comparison of performance after and before optimizations very easy.

Quantify also allows one to see the difference in performance once an optimization is made. It calculates how much faster a program becomes due to a particular optimization.

4.4 Measurements

Our test program was the same dictionary program that we used in our Rational Purify experiment. We used Quantify to identify areas for optimization and included modifications that made the program perform about 20% faster than the original program (optimizations were turned off). The optimizations we included were binary arithmetic, loop unrolling, increasing cache hits and inlining functions that were called very frequently. The features mentioned above were the ones that helped us in determining areas where optimization would be most effective.

4.5 Possible Improvements

We thought of some possible improvements that could be made for the tool:

1. Optimization suggestions: The tool should provide suggestions on types of optimizations that would be possible in particular areas. For instance, the tool can identify a loop which is called very frequently, and could suggest that one should consider loop unrolling, and maybe even provide an estimate of the performance increase expected by the modification. This is probably hard to integrate into the tool, but would certainly make the tool indispensable for the industry.

2. Cache hits and misses: The tool should provide information about the number of cache hits and misses in a program, especially in loops, since managing the cache effective can result in significant performance improvements. It should flag the user to think about modifying the loop construct to make cache hits more frequent.

5 Rational PureCoverage

The Rational PureCoverage application is packaged with the PurifyPlus software and provides an automatic way for a user to identify tested and untested code throughout an application which is in development. It works with applications that are developed using Visual C/C++, Java, and Visual Basic. The general purpose of this feature is to allow a tester to pinpoint exactly which sections of code have or have not been tested in the application. It is hoped to be use along side of a test suite, which test sections of code that have been executed, and provide a means to highlighting parts of ones code that may have been unintentionally overlooked by the test suite.

Its key features are that it (1) allows a quick analysis of executables without requiring recompilation, (2) performs an analysis of an entire application including components, with or without source code, (3) presents coverage statistics for each run of the executable, (4) provides features such as diff and merge that allow PureCoverage to compare coverage data from multiple runs of the same executable, (5) integrates with Microsoft Visual Studio 6.0, (6) allows one to control the level of code coverage data collected per module.

5.1 Expectations

In the industry development happens at a rapid speed, and many developers are work on the same application at the same time. In such cases, hasty programming is often committed and can cause a developer to forget to test the code thoroughly. It is expected that Rational PureCoverage will help rectify this problem by allowing a user of the tool to quickly locate sections of an application which have possibly gone untested during test runs and execution. It is hoped this will provide clear information to aid the developer in locating regions of code that need further examination and testing. It is hoped that PureCoverage will identify weak spots inside an application and allow developers to conduct further testing in these regions to provide increased quality to the end user, while decreasing overhead and lowering cost.

5.2 Learning Curve

The tool was tested using Microsoft Visual Studio 6.0 and the command line interface in Cygwin. The PureCoverage system integrates seamlessly with the MVS 6.0 GUI and is extremely user friendly in helping new users locate setup and preference information associated with the tool. Following the manual step-by-step, allows a user to have PureCoverage running and delivering coverage information in less than 5 minutes. The tool is sufficiently robust and allows other features of the tool to be turned on and off. The Windows GUI is extremely clean and there is never a need under MVS 6.0 to result to using the command line, because the GUI is well built and provides an articulate interface to gain easy access to all the features of the tool. However, personally, the command line has the advantage of creating scripts to conduct automatic testing of certain applications and other functions in which scripting has an advantage.

The command line interface, too, is relatively simple to become acquainted with, especially with the manual at your side. The manual has a complete listing of all the argument types and configuration parameters associated with the tool and how to use each of the features. More information and examples could have been provided in the manual. But overall, we were impressed by the documentation provided. Using the command line is a bit more tedious in that it increases the number of steps necessary to complete a coverage run, compared to the GUI which allows the tool to be run by a single click.

Overall, while the PureCoverage functionality is simple and limited and not much is needed to utilize the features of the tools, we would still have to say that IBM made a strong effort to allow the learning curve of the tool to be painless and as quick as possible. The learning curve is small and worth the time, since the benefits of the tool far overshadow the time necessary to start providing results.

This tool can overall save a developer days and weeks of testing code, because the tool specifically pinpoints areas which have most likely been overlooked and not covered during testing of the application. Before this would require a developer to pay very close attention in making sure that all of the code was tested throughout. Moreover, even if a developer does pay close attention, they could still not usually guarantee that they covered all of the code in the application, or report how much they covered.

5.3 Advanced Usage

The tool was further tested in order to gain the complete picture of its advantages and disadvantages. The tool has a beautiful GUI for displaying result information to the user. The tool provides precise information, providing statistical information regarding the extent of the application that was covered by the current and previous test runs. It provides a clear line by line process of indicating which parts of the code was possibly tested during the current run as well as the lifetime of the application. It displays the lines of code that were altered and identifies from these lines the ones that were tested or not.

Advance usage of the tool allows a user to compare coverage data of an application during different runs. This can be extremely useful in determining how changes affected the testing of the application and when and where certain pieces of code were covered. Another feature allows PureCoverage to produce an annotated source text file indicating unused code, untested code, altered code, etc. The tool can be configured to report when the coverage is below a certain threshold. This can be used as a warning mechanism to inform a tester that the application is possibly not being tested thoroughly. Features provided by the tool include ways to:

- Annotate the output of a diff for a modified source code
- List files for which coverage has changed
- Mail a report to the last person who modified insufficiently covered files
- Identify the subset of tests required to exercise modified source code
- Produce a summary in a spreadsheet format or table format

Each of these options is extremely easily configured in the GUI and on the command line. A tester could actually become extremely comfortable with this tool after a few runs of the tool and our team was actually able to start using the advance functionality within 20-30mins of launching the tool.

5.4 Experiments

The tool on average offered an overhead of about 5 seconds on each run of the program. The time added appeared insignificant and ran along side the application at runtime, taking note of what areas of the code where viewed and not viewed. This test was conducted on a version of the simple “Hello World” example that was used for the Purify test above.

The most time consuming task associated with this tool, was the feature that allowed a user to generate an annotated source text file indicating unused and uncovered portions of code that may have been untested code or newly altered code. For the “Hello World” example it took approximately 10secs to generate this output. On a larger application this time would surely increase, increasing overall overhead associated with using this feature. However, this time associated with obtaining the annotated text file is worth the benefit associated with the results. During a simple run of the program it took a user approximately one minute to determine what sections of code would be covered in a specific run, the tool was able to do it and provide results within 10-15secs. This is a relatively large difference on a very simple program. Clearly, as programs get larger and more complicated, the benefits will be more apparent. Often, large applications make it almost impossible to manually determine which sections of code are covered during execution and testing.

6 Comparison with other tools

When comparing Rational PurifyPlus with other tools used in the class, one must keep in mind that PurifyPlus is an industrial tool, whereas the ones used in class – Magic and Fluid are research tools. When comparing the tools, we focus on the usability aspect, since the functionality of the tools differs tremendously.

6.1 Fluid

“The **Fluid Project** is focused on creating practicable tools for programmers to assure and evolve real programs. We focus on "mechanical" program properties that tend to defy traditional testing and inspection regimes. These are properties with a non-local character, in that there may be no single place in the code where they are manifest, and they may involve non-determinism.”

The Fluid Project is similar to Rational PurifyPlus in that both can be used as a plug-in for an IDE. The learning curves however, differ drastically. Rational PurifyPlus is very easy to use and has a very low overhead in terms of time spent on using the tool. However, Fluid requires a significant amount of effort and time to understand the functionalities and the output and to setup the environment.

6.2 Magic

The aim of Magic is to analyze and reason about software components written in the C programming language. The overall goal of MAGIC is to check *conformance* between component *specifications* and their *implementations*. One of the most important restrictions of Magic is the fact that it can be used only on C implementations. Rational PurifyPlus on the other hand, can be used on Java, C, C# and C++ implementations, which covers most of the popular languages used in the industry today.

Moreover, the output provided by Magic is very difficult to understand and use for a developer. The counterexamples tend to be convoluted and difficult to read and therefore do not provide much utility to the user. On the other hand, the error messages provided by Rational PurifyPlus are very clear, simple and easily understandable by a developer. This improves the utility of the tool tremendously.

The learning curve associated with Magic is extremely high. Setting up the tool, and getting a fairly basic, Hello World example, can be a tedious task. On the other hand, Rational PurifyPlus is fairly easy to use and does not have a high learning curve associated with it.

7 Conclusions

We found the Rational PurifyPlus suite to be a very useful tool with a lot of potential. However, we were surprised at the difficulty in setting up the tool on the various platforms, given that it is an industrial application.

Having seen the results that PurifyPlus produces, coupled with its efficiency and user-friendly interface, we feel that it could become an asset for any compatible software development project. Each of us strongly feels that in all industry application, running Purify on the source code during development would be an excellent idea. We believe it would significantly improve developer productivity and help ensure quality of a product.

8 References

PurifyPlus Manual http://publibfp.boulder.ibm.com/epubs/pdf/12653120.pdf

Rational Quantify Support http://www-306.ibm.com/software/awdtools/quantify/support/