SpokenC: Automated Print Debugging in a Visual Environment

Master's Project by Andrew Pellegrini

Abstract

Debugging is a very broad field in computer science and many different methods and tools have been developed to facilitate this process. This paper proposes a novel method that extends from print debugging and provides an easy to use tool and a more complete debugging experience. SpokenC accomplishes this by providing an automation tool for generating debug information and a visualization tool that provides an intuitive environment for debugging. The need for this tool is explained, along with its design, empirical results and potential future direction.

Introduction

Debugging is the procedure of finding and reducing the number of "bugs" or defects in a piece of software, to make it behave as expected. While debugging derives its name from the location and removal of literal bugs from a mainframe computer, it has long since moved away from that procedure and evolved along with the rest of computers. It now takes on many different forms, each with their own methods and specific types of bugs they detect, but all of which are dedicated to reducing the number of bugs.

Since the beginning of Computer Science, debugging code has always been an integral part of any development process. As soon as any code is written it needs to be tested to ensure proper functionality and in the case of errors, debugging needs to be performed to rectify the problem. As the field of Computer Science advanced, various formalisms for development were proposed and have evolved, but even among the many different procedures, testing and debugging are always included as integral components.[7][3][10] Even though other components of the process may be removed and others added, debugging always remains, because of its importance in producing code and products of high quality.

Just as software development has evolved, and new methods were developed, debugging has also gone through a similar process. One method of debugging is print debugging. This method involves using print statements to produce information that reveals the state and the execution path of the program. Print debugging is a very useful and lightweight method of debugging, but is often difficult to implement well. In addition, the results of this method are text based and can often be hard to decipher. Because of these, and other limitations to be discussed later, this paper proposes a variant of print debugging that aims to fix these problems and create a more productive debugging method.

Debugging Techniques

Categories

To get an understanding of why this specific method of debugging was proposed, a general understanding of the existing different types, methods and tools of debugging is required. Most methods of debugging can fit into two general categories, static debugging and dynamic debugging. The methods of debugging in these two categories take drastically different approaches to finding bugs, and as such have significantly different strengths and weaknesses.

Static debugging involves some form of analysis of the code base, without it being executed, either at a high level or in a compiled state. A typical example of this type of debugging is type checking for a typed language such as C.[4] In this case, the compiler checks the types of each of the variables in the program, during the compilation process. It is then able to determine if any variable is used in a location where its type would be incompatible, and reports any such cases. This whole procedure happens without the code being run. Working with the same type checking example, some languages are dynamically typed, meaning that the types of variables are determined at runtime, when a variable is actually used. In such a case the dynamic type checker performs analysis on the running code and reports any error it encounters.

The strengths of static debugging focus primarily on the fact that since the entire code-base can be analyzed, all the errors of the type that are being checked for can be found and reported. This is significantly different from dynamic debugging where only errors that are encountered during execution can be reported. This makes static debugging a very powerful tool, as it can perform complete checking, where dynamic debugging cannot.

On the other hand, there are many types of errors that are infeasible to check statically. One such type of error is locating infinite loops.[6] While this can be performed statically in some cases, it is very often an intractable problem. Dynamic debugging, can however, detect this type of error much more easily, and is far more efficient in doing so. The fact that some types of errors are significantly easier to detect at runtime is the primary advantage of dynamic debugging.

Another point to consider is the relative burden that the debugging process places on the user. This burden can be considered to be either the amount of time and effort the user needs to devote to the debugging tool. Or also in terms of resources, either CPU or memory, that are devoted to finding the bugs. This property of the debugging tool, depends highly on both the type of bug that is being detected and the method of detection. As such, this is something that should be considered, both while developing a debugging tool and when choosing an appropriate tool for use.

Existing Tools and Methods

There are many existing tools and methods for debugging, each with their own niche, strengths and weaknesses. By understanding what already exists, the shortcomings and room for improvement become immediately evident.

The most commonplace tool for debugging is the standard debugger, built in with most modern IDEs. This tool does not focus on a specific type of bug, but instead allows the user to investigate whatever they deem necessary. To facilitate this, the standard debugger provides a way to stop and resume the execution of a program. While the program is stopped, the user can inspect the state of the program, including declared variables and associated values, call stacks and threads. Typically, this type of debugger integrates with the compiler and source and is able to provide information during runtime that would be otherwise unavailable if only the executable were available.

Another similar tool is Valgrind. Valgrind provides runtime analysis of an executable. It does not provide the ability to pause the execution of the program, but upon encountering a memory error, will present the user with program state information, including a memory map and a stack trace.[8]

A different approach entirely is automatic code correction. This method relies on the source code and automatically generates fixes for bugs, by randomly choosing a small edit, based on some heuristics, and determining if this has fixed the problem. This is a very powerful method of debugging, since it does not rely on user intervention at all, but it does require that the user put significant faith in the fixes it generates.[9]

Another very different method, somewhat similar to print debugging, is error report logging. This technique intends for a separate piece of software to be packaged with the distributed software. This extra software monitors the distributed software and will report any errors back to the developer.[1] This is similar to print debugging in that information is collected during the execution of the program and then is reported.

Print debugging is a form of error report logging, but instead of being packaged with a distributed system it is intended to work on a piece of software currently in development. Also it has the flexibility to report whatever information is deemed to be pertinent to the bug, instead of only reporting fixed information.

Of the few representative types of debugging described above, there are a few clear shortcomings. The first and most significant is that none of the above methods provide information about the entire execution of the program, but only provide snapshots at certain points in execution. These snapshots are either taken at breakpoints, like in standard debuggers, or at the time of error, like Valgrind and error report logging. By collecting information from the entire execution of the program, not only will the likelihood of collecting the relevant information be significantly increased, but also the root cause of the problem can be traced instead of just detecting the resultant problem. Also, these tools provide information in a primarily textual manner. Text is great for telling a story with a linear progression, but is not well suited for allowing the user to explore. If these tools were able to provide a graphical

representation of the debugging information or even a structured textual layout, comprehension of the essential information would be that much easier for the user.

There are also shortcomings of print debugging that this new method seeks to resolve. The first shortcoming is the amount of user intervention required to set up just a single run of print debugging. The user must determine the location of important events and then insert meaningful print statements at these locations. On every subsequent run, the user needs to add more print statements for newly discovered areas of interest and remove others that now only serve to clutter the output. After making these adjustments, the program must be compiled and run again to get the new information. All of this occurs in the code itself and must be cleaned up before the final product is shipped. The other shortcoming is that the information printed is usually very rudimentary and is either just simple flags to show the location of execution or the value of a variable of interest. The user has no way of determining what other events are occurring at the same time or what the state of the program is without making further modifications to the print statements.

SpokenC Print Debugging

Taking all the above information about the current state of debugging techniques into consideration, this paper presents a new variant of print debugging, which is packaged as a tool called SpokenC. SpokenC contains two components, one for the automatic addition of logging information and the other to present the log information in a visual manner. The automation tool takes a C file as input and performs a source to source translation on it, adding the logging statements. This produces a new C file that when compiled and run will produce logging output with information about variable declarations and assignments, as well as function calls and other information about program scopes. This tool can be run on any subset of files within a project, allowing the selection by the user of files of interest.

The visualization tool is then able parse this log information and present it in several different ways to the user, each designed to highlight different information. The program flow area shows the memory of the program and how it changes during execution. This component, shows variables as they are created and changed along a time scale with grouping on a per function basis. The call stack view integrates with the program flow view and allows the user to get more details about the program at a particular point in time. This detailed information contains call stack information as well as the current values of the variables from each of those scopes. The visualization tool also provides a more traditional text view where the user can view the log file side by side with the source code, and perform various filtering operations.

SpokenC provides an exploratory environment, where all the log information is collected in a single run of the program. This allows the user to then explore different parts of interest in the program without the need for either changing print statements, recompilation or rerunning the program. This provides the user with flexibility, as they are free to inspect the debugging information without having to leave and return to debugging several times.

Design

As mentioned just above, SpokenC consists of two tools, both with a very specific task. The details of what each component is capable of, and how this functionality is provided, is described below.

Automation

The automation tool's purpose is to automatically add meaningful debug statements in the source code. This is done to remove the step of manually adding debug statements from the user's debugging process. In addition, creating a version of the source code with the debug statements will result in a faster runtime than if the debugging was added at runtime. In order to perform this source to source translation, a large amount of details about the source code is required. To accomplish this, a C parser is used. The automation tool is written in Python, and as a result, the pycparser library was chosen, because of both its simplicity and flexibility.[2] pycparser specifically allows a single C file to be preprocessed and then parsed to an AST. This accommodates the functionality of the user being able to select which files they are interested in. Once the AST is generated, it can be recursively inspected, and upon encountering an interesting piece of code, pertinent debug information can be inserted at that location in the source code.

The events of interest, that debug information is being added for, include function calls and returns, variable declarations and assignments, and changes in program scope. For each function call, a debug log is inserted that indicates which function was called and what the arguments, and values of them, are. This provides the user with a good overview of the environment before any function code is executed. Function returns are also logged, and the information included is the name of the function that is returning and what the return value, if any, is. Variable declarations produce log information about the name and type of the variable and variable assignments produce log information about the name, type and current value. The last type of log information that is generated, scope changes, is not directly useful to the user, but when combined with the visualization tool, a detailed structure of variables and their scopes can be presented. Having the ability to view just the variables in a particular scope, is particularly useful during debugging, and is the reason why this information was chosen to be logged. All log information also includes file name and line numbers, allowing the user to determine the location in the source code of each logged event.

A sample of the before and after source code along with the log output is shown below.

(a)

```c
#include <stdio.h>

int factorial(int i) {
    if (i <= 1) {
        return 1;
    } else {
        return i * factorial(i - 1);
    }
```

```
}

int main(int argc, char **argv) {
    int x = 0;
    const char *string = "The factorial of %d is: %d\n";
    for (int i = 1; i <= 2; i++) {
        x = factorial(i);
        printf(string, i, x);
    }
    x = 10;
    return 0;
}
```

(b)

```
#include <stdio.h>

int factorial(int i) {
  fprintf(stderr, "fact.c:3:call(factorial)\n");
  fprintf(stderr, "fact.c:3:decl(int,i)\n");
  fprintf(stderr, "fact.c:3:assign(int,i,%d)\n", i);
  if (i <= 1)
  {
    fprintf(stderr, "fact.c:4:scope_in\n");
    int ____0 = 1;
    fprintf(stderr, "fact.c:5:return(factorial,int,ret,%d)\n", ____0);
    return ____0;
    fprintf(stderr, "fact.c:4:scope_out\n");
  }
  else
  {
    fprintf(stderr, "fact.c:4:scope_in\n");
    int ____1 = i * factorial(i - 1);
    fprintf(stderr, "fact.c:7:return(factorial,int,ret,%d)\n", ____1);
    return ____1;
    fprintf(stderr, "fact.c:4:scope_out\n");
  }

  fprintf(stderr, "fact.c:9:return(factorial,int,ret,undef)\n");
}


int main(int argc, char **argv) {
  fprintf(stderr, "fact.c:11:call(main)\n");
  fprintf(stderr, "fact.c:11:decl(int,argc)\n");
  fprintf(stderr, "fact.c:11:assign(int,argc,%d)\n", argc);
  fprintf(stderr, "fact.c:11:decl(char**,argv)\n");
  fprintf(stderr, "fact.c:11:assign(char**,argv,%p)\n", argv);
  int x = 0;
  fprintf(stderr, "fact.c:12:decl(int,x)\n");
  fprintf(stderr, "fact.c:12:assign(int,x,%d)\n", x);
  const char *string = "The factorial of %d is: %d\n";
  fprintf(stderr, "fact.c:13:decl(char*,string)\n");
  fprintf(stderr, "fact.c:13:assign(char*,string,%s)\n", string);
  fprintf(stderr, "fact.c:14:scope_in\n");
  int ____2 = 1;
  fprintf(stderr, "fact.c:14:decl(int,i)\n");
```

```
  fprintf(stderr, "fact.c:14:assign(int,i,%d)\n", ____2);
  for (int i = ____2; i <= 2; (i++) | (fprintf(stderr,
       "fact.c:14:assign(int,i,%d)\n", i + 1) == 0))
  {
    x = factorial(i);
    fprintf(stderr, "fact.c:15:assign(int,x,%d)\n", x);
    printf(string, i, x);
  }

  fprintf(stderr, "fact.c:14:scope_out\n");
  x = 10;
  fprintf(stderr, "fact.c:18:assign(int,x,%d)\n", x);
  int ____3 = 0;
  fprintf(stderr, "fact.c:19:return(main,int,ret,%d)\n", ____3);
  return ____3;
  fprintf(stderr, "fact.c:20:return(main,int,ret,undef)\n");
}
```

(c)

```
fact.c:11:call(main)
fact.c:11:decl(int,argc)
fact.c:11:assign(int,argc,1)
fact.c:11:decl(char**,argv)
fact.c:11:assign(char**,argv,0xbfae95e4)
fact.c:12:decl(int,x)
fact.c:12:assign(int,x,0)
fact.c:13:decl(char*,string)
fact.c:13:assign(char*,string,The factorial of %d is: %d)
fact.c:14:scope_in
fact.c:14:decl(int,i)
fact.c:14:assign(int,i,1)
fact.c:3:call(factorial)
fact.c:3:decl(int,i)
fact.c:3:assign(int,i,1)
fact.c:4:scope_in
fact.c:5:return(factorial,int,ret,1)
fact.c:15:assign(int,x,1)
fact.c:14:assign(int,i,2)
fact.c:3:call(factorial)
fact.c:3:decl(int,i)
fact.c:3:assign(int,i,2)
fact.c:4:scope_in
fact.c:3:call(factorial)
fact.c:3:decl(int,i)
fact.c:3:assign(int,i,1)
fact.c:4:scope_in
fact.c:5:return(factorial,int,ret,1)
fact.c:7:return(factorial,int,ret,2)
fact.c:15:assign(int,x,2)
fact.c:14:assign(int,i,3)
fact.c:14:scope_out
fact.c:18:assign(int,x,10)
fact.c:19:return(main,int,ret,0)
```

Figure 1(a, b, c): (a) pre-translation source code. (b) post-translation source code. (c) log output.

Limitations

Although there is no theoretical limit on what can be done in this source translation step, there are some practical limitations on what can be done currently. There is no support for array member or struct variables. In the case of array members, a more advanced parsing must be done to allow logging of individual members. As for struct variables, this would require developing a type parser, to manage the types of members of the struct to allow for proper logging. While the automation tool does not currently generate any debug information for these assignments, a program with this type of operation can still be run through the tool. It also does not support global variables and will fail when trying to generate debug output for these.

A physical limitation that is imposed by the source to source translation nature of this process, is that logging can only be added to file for which the source is available. This means that no precompiled libraries can be logged. However, since the library is not the code being debugged, this is a less significant limitation.

Visualization

The other component of SpokenC is the visualization tool. As can be seen above, the log output by itself is of limited value, because of its dense text nature. It was designed to be parsed by the visualization tool, as having a graphical debugging environment is one of the original goals of SpokenC. The graphical environment provided by this tool has three main components, the program flow view, the call stack view and the log view.

The program flow view is the centerpiece of the visualization tool and is designed to be the central focus of debugging tasks. This view consists of several graphs. Each graph contains information for a single variable. The graphs are linear, consisting of a head node for the declaration of the variable and then several following nodes for the assignments to that variable. The nodes contain the value of the variable after the assignment and the edges are labeled with the location in the source where the assignment occurs. The graphs are laid out horizontally, each column representing a different variable, and the vertical spacing indicating the time during execution of each assignment. The nodes are also color coded by function, allowing the user to quickly identify different sections of the code and finding where each variable is modified.

The call stack view presents detailed information about the entire state of the program at a particular point in execution. By clicking on a node in the program flow graph, the user is presented with this detailed information. The information is presented as a fairly standard call stack, with each currently active function call listed, with the most recent one on top. In addition to this information each function call has a nested list of its own variables and their current values. The function scopes are also colored coded and are consistent with the coloring of the program flow view.

The third view is the log view. This view consists of two side by side text areas, one containing the log file and the other containing the source files. The log file is color coded by log event type and has filtering capabilities based on log event type.

Below are some sample views from the visualization tool of the above log file.

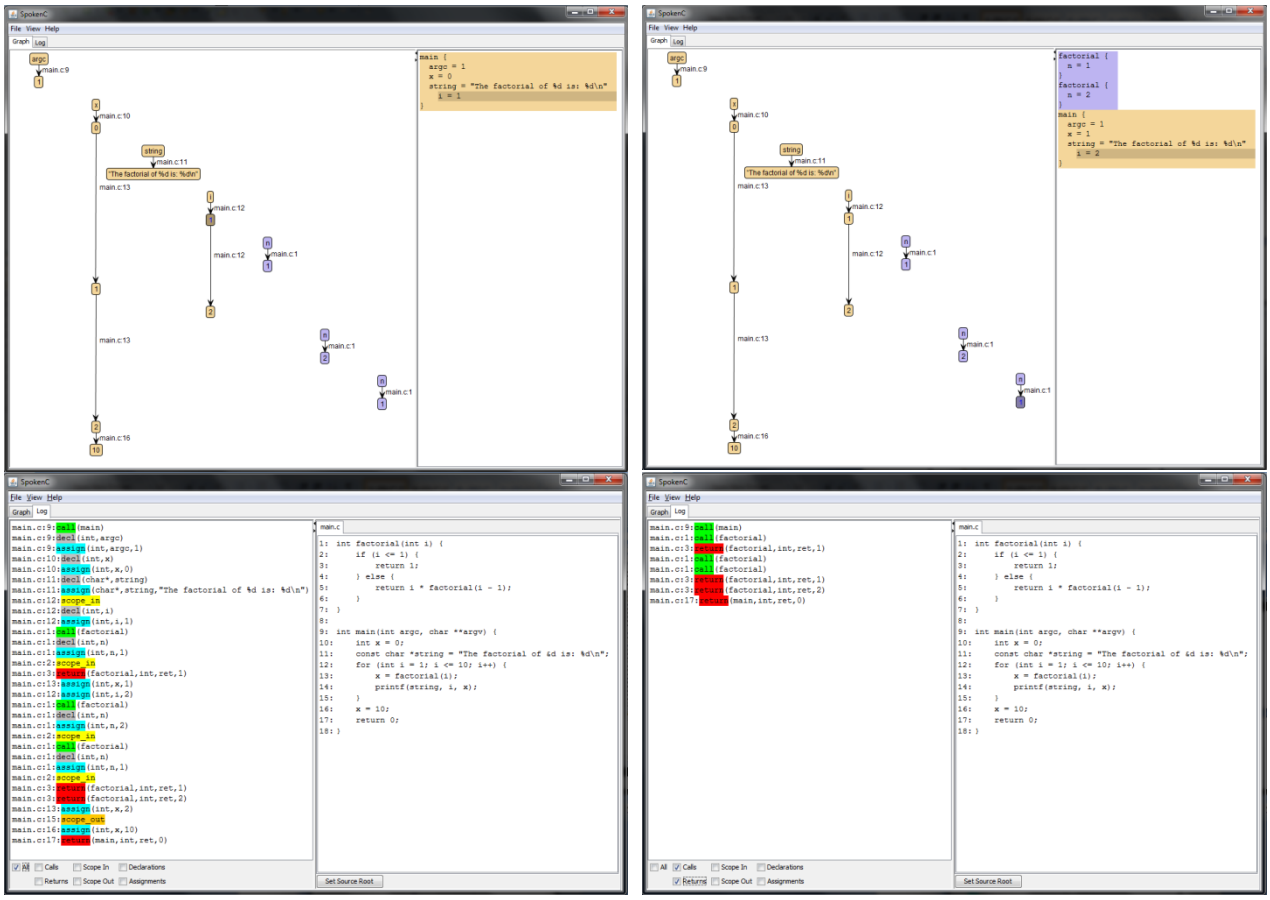

Figure 2: sample views of the visualization tool

Testing

To determine the usefulness of the SpokenC debugging tool some testing must be performed. The tests performed contain both objective and subjective measures and focus on both the automation tool and the visualization tool.

Automation

Tests for the automation tool focus on correctness, completeness and runtime. The automation tool was run on a sampling of simple programs and real world programs. Correctness is a boolean value of whether or not the resultant program compiles and produces the same results. The time metric is the time required to perform the translation in seconds. Relative runtime is the multiple of time required to run the translated program over the original. The results are listed below.

Program Name	Description	Correct	Time	Relative Runtime
factorial	Calculates and prints the value of several factorials.	Y	.103s	42.3
merge	Merges two ordered lists of integers	Y	.124s	10.5
compact	Prompts for an input string and then reduces sequences of whitespace characters into a single space	Y	.125s	18.2

Table 1: automation tool test results

Visualization

The visualization tool was tested mainly on likability, using an exit survey to collect impressions of the tool. The metrics were all created using the Likert scale and are shown below along with aggregated results from five experienced programmers.

Prompt	Result
I enjoyed using this application	4
I would prefer using this program over other debugging programs	3
I would use this program in conjunction with other debugging programs	5
I was able to determine which sections of the program correspond with parts of the visualization	4.5
The visualization provided useful information	4
I was able to interpret the call stack	5
The call stack provided useful information	4.5
I was able to interpret the log file	4
The log filters were useful in inspecting the log file	5

Table 2: visualization tool survey results

Analysis

Looking first at the results of the automation tool results, we can see that it was correct on all test cases. This does not mean that it can be used on all C programs, but instead that if the previously stated limitations are kept in mind then it will be successful. Looking at the translation time, we see that none of the translations take a significant amount of time and as a result do not impose a burden on the user. The relative runtime of the test programs is in the range of 10 to 50 times the runtime of the pre-translation code. This may seem to be very high, but keeping in mind other runtime debugging tools such as Valgrind, this is right on par, with other tools of the same nature. In addition, other runtime methods like print debugging require several runs of a program to locate the source of a bug. In this sense, SpokenC provides an advantage in that the program will only need to be run once, considering the code coverage that is provided.

The results of the visualization tool survey are very interesting. They indicated that there is strong likability of each of the different views, however the tool as a whole is rated somewhat lower. Of interest is metric 2 and 3. These indicated that the visualization tool would not be used as the sole method of debugging, but would be a welcome addition to any debugging environment. The things that this tool does well pertain to analysis of a program after it has been run. It, by design, does not provide any runtime debugging and that may be part of why the test

users felt that it couldn't replace other debugging programs. What this program does do well is allowing the user to explore program execution and determine where the problem arose, and this is strongly shown by the results.

Future Direction

The SpokenC debugging tool is only a first attempt at this type of debugging and as such there is plenty of room for future development. An immediate improvement that can be made is adding support for struct and array types. This would significantly increase the range of programs for which this tool can be used. A user convenience feature that could be added is to allow user control of what variables and functions produce debug information. This would increase the granularity of user control beyond file specification, and would allow for a better runtime speed at the cost of increased user intervention. Another step, would be to implement the automation tool as part of the compiler. This would reduce the number of steps the user needs to take to get a debug executable, and could be as simple as including a flag to the compiler. This would take significant effort to implement as most compilers are not very modification friendly. One such compiler that provides some modification support is Clang.[5] However, at the moment, Clang does not support code generation from a modified AST. A significant undertaking, would be to extend support to the C++ language. This would require handling of other types, such as classes and dealing with new language constructs such as templates.

Conclusion

As shown in this paper, SpokenC provides a new method of debugging, similar to, but still radically different from other current types of debugging tools. The current implementation provides a novel and well liked tool. As seen in the testing section, this tool is both efficient and liked by users. This leaves the door open for future development and new innovations to further improve this already useful tool.

References

1. Arnold, M., Vechev, M., & Yahav, E. (2008). QVM: an efficient runtime for detecting defects in deployed systems. *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, *43*(10), 143-162. Retrieved from http://dl.acm.org/citation.cfm?id=1449776&bnc=1
2. Bendersky, E. (2013, January 24). *pycparser*. Retrieved May 1, 2013, from https://bitbucket.org/eliben/pycparser
3. Boehm, B. B. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, *11*(4), 14-24. Retrieved from http://dl.acm.org/citation.cfm?id=12948&bnc=1
4. Cardelli, L. (2004, February 25). *Type systems*. Retrieved May 1, 2013, from http://lucacardelli.name/Papers/TypeSystems.pdf
5. *"clang" C Language Family Frontend for LLVM*. (n.d.). Retrieved May 1, 2013, from http://clang.llvm.org/
6. Cook, B. (n.d.). *Principles of program termination*. Retrieved May 1, 2013, from http://www.cs.tufts.edu/comp/150BUGS/terminator-principles.pdf
7. Dr. Royce, W. W. (1970). *Managing the development of large software systems*. Retrieved May 1, 1970, from http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf
8. Nethercote , N., & Seward, J. (2007). How to shadow every byte of memory used by a program. *Proceedings of the 3rd international conference on Virtual execution environments*, 65-74. Retrieved from http://dl.acm.org/citation.cfm?id=1254820&bnc=1

9. Novark, G., Berger , E. D., & Zorn, B. G. (2008). Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM - Surviving the data deluge*, *51*(12), 87-95. Retrieved from http://dl.acm.org/citation.cfm?id=1409382&bnc=1
10. Schwaber, K., & Sutherland, J. (2011, October). *The scrum guide The definitive guide to scrum: The rules of the game*. Retrieved May 1, 2013, from http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf