AN INTERACTIVE DEBUGGING TOOL FOR C++
BASED ON DYNAMIC SLICING AND DICING
PART II: IMPLEMENTATION, TESTING, AND EVALUATION

ABSTRACT
The main objective of this work was to develop an interactive debugging tool for C++ programs. The tool that was developed is called C++Debug and it uses program slicing and dicing techniques. The design started by including simple statements first and then expanded to pointers, structures, functions, and classes. In order for C++Debug to be more powerful, dynamic slicing rather than static slicing was chosen. The work includes new algorithms that handle Class, Function, and Pointer in C++.

Slicers
C++Debug is comprised of four parts: Cpptrace, Database, Slicer, and Dicer (as shown in Figure 1).

1. INTRODUCTION
C++Debug is an interactive debugging tool designed to function as a utility program of the UNIX system. C++Debug was developed based on slicing and dicing techniques. In order for C++Debug to be more powerful, dynamic slicing rather than static slicing was chosen for implementation. C++Debug was designed in a way to allow ease and convenience on the part of the user. Using C++Debug, the user can interact directly with the computer in locating errors in a program. Menus are provided to allow the user to select any one of a number of functions (Slice, Dice, Help, etc.) supported by C++Debug.

To produce the C++Debug tool, three activities of a software process are introduced: software specification, software development, and software validation. Some parts of the waterfall approach are used to take those three activities and represent them as separate process phases: requirements specifications, software design, implementation, testing, and valuation. In order to make C++Debug a good piece of software, essential attributes such as maintainability, dependability, efficiency, and usability were considered.

2. IMPLEMENTATION AND RESULTS
2.1 Implementation
2.1.1 C++Debug Block Diagram
C++Debug is comprised of four parts: Cpptrace, Database, Slicer, and Dicer (as shown in Figure 1).

1. Cpptrace was designed as a tool allowing one to follow the execution of a C++ program, statement-by-statement. Cpptrace reads the C++ source program in a file, inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to generate two major parts: (1) a trajectory of the program and (2) some databases, where a trajectory is a feasible path that has actually been executed for some input and the databases are a list of reserved words, a list of basic types, identifier information, types, symbol tables, and scope of influence. flex and bison are tools used to implement Cpptrace. flex reads a specification file containing regular expressions for pattern matching and generates a C or C++ routine that performs lexical analysis [3]. This routine reads a stream of characters and matches sequences

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’05, March 13–17, 2005, Santa Fe, New Mexico, USA.
Copyright 2005 ACM 1-58113-000-0/00/0004...$5.00.
that identify tokens. Bison reads a specification file that codifies
the grammar of a language and generates a parsing routine [1].
This routine groups tokens into meaningful sequences and
invokes action routines to act upon them. C++ grammar from
Stroustrup’s textbook was used in this implementation [8].

2. Database stores ordered sets of data such as a list of reserved
words, a list of basic types, identifier information, types, symbol
tables, and scope of influence, etc. All data are created by
Cpptest as a database. The D and U ordered sets of data are
computed from the trajectory path. This database is used by Slicer to compute a program slice(s).

3. Slicer was created to obtain a new program of generally smaller size that still maintains all aspects of the original program's behavior with respect to the criterion variable. The number of program slices is dependent on the slicing criterion.

4. Dicer can then be used to compare two or more slices resulting from the program slicing technique in order to identify the set of statements that are likely to contain an error.

2.1.2 Datastructures

The datastructures of a source program, functions, a trajectory, sets such as D, UDU, DCL, etc. were implemented based on datastructures shown in Figure 2.

2.1.3 Symbol Tables

Symbol tables were designed by following the concepts of symbol tables that are used in cool, the Classroom Object-Oriented Language [2]. cool is a small language designed for use in an undergraduate compiler course project at the University of California at Berkeley [2]. The key is two functions: enterscope() and exitscope(). Function enterscope() makes the table point to a new scope whose parent is the scope it pointed to previously, while function exitscope() makes the table point to the parent scope.

3. LIMITATION

C++Debug has some limitations as listed below.

1. Limitation of OS: UNIX
2. Limitation of language: GNU G++
3. Limitations of algorithm: worst-case O(N^2V), average-case O(N log N), best-case O(N), where N is the #LOC of the trajectory part, and V is the maximum number of variables in each line in a debugged program.
4. In the current implementation, limitation of #LOC of the executable part: 1,000.

4. RESULT

Based on the results of the experimentation, C++Debug could generate a new slicing program that is of smaller size than the original source program. The new slicing program still preserves part of the program's original behavior for a specific input. In addition, C++Debug can be used as a tool like ctrace under UNIX. C++Debug can work on both C and C++.

By using the -g option, C++Debug supports the generation of

---

<table>
<thead>
<tr>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxLine</td>
</tr>
<tr>
<td>MaxTraj</td>
</tr>
<tr>
<td>VarLength</td>
</tr>
<tr>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>Action</td>
</tr>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable = string</th>
</tr>
</thead>
<tbody>
<tr>
<td>SliceCriterion, LastDef</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>Fname</td>
</tr>
<tr>
<td>TFrame</td>
</tr>
<tr>
<td>MT</td>
</tr>
<tr>
<td>MTFname</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>DFrame</td>
</tr>
<tr>
<td>UFrame</td>
</tr>
<tr>
<td>LFrame</td>
</tr>
<tr>
<td>DUName</td>
</tr>
<tr>
<td>LDFName</td>
</tr>
<tr>
<td>TCFname</td>
</tr>
<tr>
<td>IFname</td>
</tr>
</tbody>
</table>

| A         | = array[1..N] of set[Action] |
| S         | = array[1..N] of set[Action] |
| LastDef   | = set[action] // a set of last definition |
| LT        | = set[Action] // a set of test actions |
| VS        | = set[X] // Variable-Scope |
| CS        | = set[X] // Control-Scope |
| Slice     | = array[1..N] of set[X] // Slices |
| Dice      | = set[X] // a final dice |

---

**Figure 2. Slicing data structures**
grammar derivation trees. A user can study how the parser checks the syntax of a program. By using the -1 option, all information about C++Debug can be displayed. One who is interesting in the dynamic slicing area can use the information provided by C++Debug, such as D, U, DU, symbol tables, etc., to investigate the process of slicing, dicing, or compiling in general.

5. PROBLEMS AND SITUATIONS IN C++ THAT WERE TAKEN INTO ACCOUNT IN THE DESIGN

There are eight major problems and situations in C++ that were taken into account in the design of C++Debug. They are discussed bellow.

1. Problems and situations with classes and objects such as classes, structures, unions, anonymous unions, friend functions, friend classes, inline functions, defining inline functions within a class, parameterized constructors, static class members, static data members, static member functions, the scope resolution operator, nested classes, local classes, passing objects to functions, returning objects, and object assignment.

2. Problems and situations with arrays, pointers, references, and the dynamic allocation operators such as arrays of objects, uninitialized arrays, pointers to objects, type checking C++ pointers, the *this pointer, pointers to derived types, pointers to class members, reference parameters, passing references to objects, returning references, independent references, references to derived types, restrictions to references, dynamic allocation operators (i.e., the new operator in C++), initializing allocated memory, allocating arrays, allocating objects, the noexcept alternative, and the placement forms of new and delete.

3. Problems and situations with function overloading, copy constructors, and default arguments such as function overloading, overloading constructor functions, overloading a constructor to gain flexibility, initialized and uninitialized objects, copy constructors, finding the address of an overloaded function, the overload anonymism, default function arguments, default arguments vs. overloading, using default arguments correctly, and function overloading and ambiguity.

4. Problems and situations with operator overloading such as operator overloading using a friend function, using a friend to overload += or -=, friend operator functions adding flexibility, overloading new and delete, overloading new and delete for arrays, overloading the noexcept version of new and delete, overloading some special operators, overloading [], overloading (), overloading ->, and overloading the comma operator.

5. Problems and situations with inheritance such as base-class access control, inheritance and protected members, protected base-class inheritance, inheriting multiple base classes, constructors, destructors, inheritance, passing parameters to base-class constructors, granting access, and virtual base classes.

6. Problems and situations with virtual functions and polymorphism such as virtual functions, calling a virtual function through a base class reference, the inherited virtual attribute, hierarchical virtual functions, pure virtual functions abstract classes, and late binding.

7. Problems and situations with templates such as generic functions, a function with two generic types, explicitly overloading a generic function, loading a function template, using standard parameters with template functions, generic function restrictions, applying generic functions, a generic sort, compacting an array, generic classes, a generic array class, using non-type arguments with generic classes, using default arguments with template classes, explicit class specializations, and the typename and export keywords.

8. Problems and situations with exception handling such as exception handling fundamentals, catching class types, using multiple catch statements, handling derived-class exceptions, exception handling captions, catching all exceptions, restricting exceptions, rethrowing an exception, terminate() and unexpected(), the uncaught_exception() function, and the exception and bad_exception classes.

6. TESTING

Testing is the primary means for showing that the implementation has the requisite functionality and satisfies other non-functional properties [7]. Testing and other forms of verification and validation are important at all stages of the software development process. A number of standard testing techniques were used to test C++Debug. These techniques include random testing, assertion testing, grammar-based testing, functional testing, black and white box testing, requirements analysis testing, and integration testing.

7. EVALUATION

This section defines the evaluation approach adopted for C++Debug. The evaluation procedure is explained along with the details of the experimental design and the results obtained.

7.1 Introduction

C++Debug was evaluated based on Lyle [5] [6] and Gallagher’s [4] approach by training several Computer Sciences graduate students at Oklahoma State University in its operation and by collecting data on how the students used C++Debug to locate faults in C++ programs. The main objective of the evaluation was how can C++Debug be used to enhance the debugging process and localize errors.

7.2 Evaluation Procedure

The debugging process was studied by allowing each student to debug one program with and without C++Debug. There were four steps as listed below.

7.2.1 Step I: Familiarization

Let each student answer a questionnaire covering background information, read an overview of the evaluation, and finally read the manual on how to use C++Debug.

7.2.2 Step II: First Treatment

Let each student debug C++ programs without using the C++Debug tool. Each student can use other tools such as DBX, GDB, etc.
7.2.3 Step III: Second Treatment

In this step, the C++ programs in step II were debugged by using the C++Debug tool.

7.2.4 Step IV: Subject Remarks

All information from Step I, Step II, and Step III were collected and analyzed based on Lyle’s [6] approach to find out:

1. Is C++Debug useful?
2. Are there some negative and positive comments?
3. What do they like about C++Debug?
4. What don’t they like about C++Debug?

The students involved in the evaluation of C++Debug were asked to fill out a questionnaire based on Lyle’s [6] approach as follows.

Questionnaire
(1) How long have you been programming (Years/Months)?
(2) How many CS, (Computer Science), classes in your BS/BA?
(3) How many CS classes taken so far in grad school?
(4) How many other CS classes have you taken?
(5) Which programming languages are you familiar with? Familiar means you used the language for at least a semester’s work.
(6) On a scale from 0 to 10, how familiar are you with C++?

where
0 = I’ve never used C++
2 = I know some C++
5 = I know C++ about average
7 = I am comfortable with C++
10 = I know C++ well

0 2 4 6 8 10
++-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

(put a check mark on the scale)

(7) On the same scale from 0 to 10, how familiar are you with the VI or EMACS text editor?
0 2 4 6 8 10
++-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

(put a check mark on the scale)

(8) Do you know about program slicing?

The subjects involved in the evaluation of C++Debug were ten graduate students at the Computer Science Department of Oklahoma State University. The student responses to the questions are summarized in Table 1 and 2. The number of changes made to the tested programs by each student, and the number of slices each student computed are shown in Table 3. And finally, edit times, compile times, and execution times are presented in Tables 4 and 5.

7.3 Comments on C++Debug

Seven of the ten subjects reported that in the slicing mode C++Debug was very useful. In the dicing mode, four subjects reported that C++Debug can help them to locate errors in a program. Five subjects felt surprised that C++Debug could
eliminate irrelevant statements. Three subjects said that in the –t mode the trajectory path generated by C++Debug worked like the cptrace tool in C, in an effective and useful manner.

On the negative side, one subject felt that C++Debug was not more powerful than other debugging tools like GDB. Two subjects mentioned that the dicing process is quite complicated because of the process of selecting the appropriate slicing criteria (variables and positions for dicing). One subject mentioned that in the –g mode, C++Debug generated derivation tree that were too long, and that it was difficult to understand all of them.

8. CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

C++Debug was designed to allow ease and convenience on the part of the user. Using C++Debug, a user can interact directly with the computer in locating errors in a certain program. For convenience, the program provides menus to allow the user to select any one of the functions contained therein. Based on the results of the experimentation, C++Debug could generate a new slicing program that is of smaller size than the original source program. The new slicing program still preserves part of the program's original behavior for a specific input. In addition, C++Debug can be used as a tool like cptrace under UNIX [9]. C++Debug can work on both C and C++.

8.2 Future Work

Based on the initial experiments with C++Debug, we found that improvements and additions can be made to C++Debug in the following aspects.

8.2.1 Improvements

The size of C++Debug after compiling by an optimized compiler is 2,088,720 bytes. It appears that it should be smaller if some algorithms and memory uses are managed better. Time and space complexities are dependent on the size of the trajectory (and not necessarily the size of the source code). To avoid running out of disk space (which is needed to store the trajectory path), the user must know how far the trajectory must go and how much disk space is required. It would be better if C++Debug can automatically check and tell the user about the sufficiency of the disk space. And it should also estimate the time that C++Debug is going to take to obtain the slices and the dices.

Table 5. Time measures for debugging without using the tool

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>mean</th>
<th>sd</th>
<th>min</th>
<th>max</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit user time</td>
<td>10</td>
<td>1224</td>
<td>1012</td>
<td>371</td>
<td>2501</td>
<td>902</td>
</tr>
<tr>
<td>Edit system time</td>
<td>10</td>
<td>713</td>
<td>633</td>
<td>286</td>
<td>1522</td>
<td>411</td>
</tr>
<tr>
<td>Compile user time</td>
<td>10</td>
<td>12501</td>
<td>492</td>
<td>11903</td>
<td>12833</td>
<td>12532</td>
</tr>
<tr>
<td>Compile system time</td>
<td>10</td>
<td>3962</td>
<td>557</td>
<td>3255</td>
<td>4482</td>
<td>3921</td>
</tr>
<tr>
<td>Execute user time</td>
<td>10</td>
<td>588</td>
<td>113</td>
<td>464</td>
<td>621</td>
<td>521</td>
</tr>
<tr>
<td>Execute system time</td>
<td>10</td>
<td>730</td>
<td>248</td>
<td>492</td>
<td>919</td>
<td>627</td>
</tr>
</tbody>
</table>

8.2.2 Additions

Instead of just menus, some windows should be supported so that a user can view the source code, the trajectory path, the program slice, etc. on the screen. Using a mouse can help a user probably better than using the keyboard in selecting which function to use, or selecting the variables and positions required to compute a slice.

8.2.3 Future Work

For a tested C++ program that has pointers, global variables, and static declarations in classes, the algorithm that was used to implement C++Debug yields an output slice larger than it should be (however, it still gives the correct output and its size is smaller than the original source program). Some lines that should be eliminated are not eliminated. If a better algorithm to manage pointers, global variables, and static declarations in classes is implemented, the size of the resulting slice will be smaller.

It will be desirable if C++Debug can be made a multi-user-tool. However, in the current implementation, since C++Debug saves specific files in a local directory, it cannot be used in the multi-user mode.

Because of the complexities of the C++ symbol table and the time constraint, the current version of C++Debug cannot treat array elements and fields in dynamic records as individual variables.

9. REFERENCES