Aristotle: A System for Development of Program Analysis Based Tools

Mary Jean Harrold, Loren Larsen, John Lloyd, David Nedved, Melanie Page, Gregg Rothermel, Manvinder Singh, Michael Smith
Department of Computer Science
Clemson University
Clemson, South Carolina USA 29634-1906
harrold@cs.clemson.edu

Abstract- Aristotle provides program analysis information, and supports the development of software engineering tools. Aristotle's front end consists of parsers that gather control flow, local dataflow and symbol table information for procedural language programs. We implemented a parser for C by incorporating analysis routines into the GNU C parser; a C++ parser is being implemented using similar techniques. Aristotle tools use the data provided by the parsers to perform a variety of tasks, such as dataflow and control dependence analysis, dataflow testing, graph construction and graph viewing. Most of Aristotle's components function on single procedures and entire programs. Parsers and tools use database handler routines to store information in, and retrieve it from, a central database. A user interface provides interactive menu-driven access to tools, and users can view results textually or graphically. Many tools can also be invoked directly from applications programs, which facilitates the development of new tools. To assist with system development and maintenance, we are also creating support tools for managing bug and test suite databases.

1 Introduction

The availability of program analysis systems that can be used to build prototype software engineering tools for experimentation on large software systems written in C and C++ is limited. Some development systems [16] operate on a subset of the C language, which restricts the class of programs that can be input to the system. Others, such as ProDAG [23], provide some of the analysis information required but currently process neither C nor C++ programs.

To alleviate this problem, we have developed a program analysis system, called Aristotle. Aristotle operates on C programs; we are currently extending it to handle programs written in C++. Aristotle provides an environment that supports a variety of tools for static and dynamic analysis of programs. Aristotle parsers collect control flow, local dataflow, and symbol table information on source code files, and store it in a central database. Aristotle tools use information from this database to perform further analysis, or provide views of analysis data. For example, one tool uses database information to build control dependence graphs. Aristotle tools can be accessed in two ways. First, users can invoke Aristotle tools interactively through a menu interface, or from the command line. Second, programmers can invoke many Aristotle tools from applications programs, by linking their programs with libraries that contain the tools as callable subroutines.

We have implemented Aristotle's C parser using the front end of the GNU C compiler\footnote{Copyright (C) 1987, 1989 Free Software Foundation, Inc. 675 Mass Avenue, Cambridge, MA 02139.} [24]. We have also implemented a set of program analysis tools that perform further analysis, and support the development of additional tools. For example, our instrumented-code generator creates an instrumented version of a program that, when executed with input data, provides execution trace information. This execution trace information is used by a dataflow tester, and will be used by other testers and dynamic program slicers. We have also implemented a control dependence analyzer, intraprocedural and inter-
2 System Overview

Figure 1 overviews the functional requirements for Aristotle using a dataflow diagram. The diagram depicts the system in terms of six groups of functions:

- A user interface
- Handlers that write information to a database
- Handlers that read information from a database
- A family of parsers
- A collection of analysis tools
- A group of viewing tools

Aristotle’s user interface transforms user input into commands that invoke Aristotle parsers, analysis tools, and viewing tools. We discuss this interface in Section 4.4.

Aristotle stores program analysis information in a central database. The database is accessed through database handlers. Database write handlers receive information from parsers or analysis tools, and deposit the data into the database. Database read handlers retrieve data from the database and pass it to analysis or viewing tools. In Section 3.2, we describe the Aristotle database and database handlers.

Aristotle parsers parse programs written in procedural languages, gather analysis information about these programs, and pass information to database write handlers. In Section 3.1, we discuss the Aristotle C parser that is currently in use.

Aristotle analysis tools perform several types of program analysis. Figure 2 expands the “analysis tools” bubble of Figure 1, to depict the requirements for these tools in greater detail. The figure depicts the following types of tools:

- Dataflow analyzers
- Control dependence analyzer
- Graph builders
- Code instrumenter
- Dataflow tester
Aristotle analysis tools use analysis data obtained from database read handlers, and parameters obtained through the user interface, to perform various types of program analysis. The tools output data to database write handlers. We describe the five groups of analysis tools listed above in Section 3.3.

Aristotle viewing tools let users view program analysis data. Figure 3 expands the “viewing tools” bubble of Figure 1, to depict the requirements for viewing tools in greater detail. As the figure illustrates, the system supports two classes of tools: table viewers and graph viewers. These tools input analysis information from database read handlers, and then format that data for viewing. Data formatted by graph viewers is input to a graph viewing tool, XVCG. We discuss viewing tools further in Section 3.4.

We are developing Aristotle incrementally. We expect that many program analysis based tools will use information gathered by Aristotle, and that the system will be used for diverse purposes. Since we cannot fully predict the uses Aristotle will support, our primary design objectives for the system are that it be modular and extensible, and provide easy access to analysis data.

It is easy to integrate new program analysis tools with Aristotle: a tool designer need only link tools with database handler routines that provide database
information. When tools generate data that is useful to other tools, the tool designer uses handler routine templates and common handler library routines to build new handlers that pass the new data to, and retrieve it from, the central database.

Our work with Aristotle has also motivated the development of software engineering support tools. We are implementing tools that manage bug and test suite databases. We describe these tools in Section 5.

3 System Components

3.1 Parsers

To perform different types of analysis on a program, we need both control flow and dataflow information about the program. A control flow graph contains one node for each statement in a program; edges between nodes represent flow of control between statements. Figure 4 shows a program segment and its control flow graph; simple statements in the program segment are labeled “Sn”, while predicate statements are labeled “Pn”. We discuss dataflow information in Section 3.3.1.

Although program analysis information can be gathered at many different stages of translation, we gather information during the parse. Aristotle’s parser for C, CParse, analyzes a C procedure, a group of C procedures or an entire C program to obtain information required by other Aristotle modules. CParse accepts a C source file as input, and outputs control flow, dataflow and symbol table information about each procedure in the file.

We created CParse by inserting actions into the GNU C parser. As the parser recognizes each language construct, it passes (through calls embedded in the inserted actions) the portion of the abstract syntax tree that corresponds to that construct to an analysis routine. Given a portion of the abstract syntax tree, this analysis routine first constructs the portion of the control flow graph that corresponds to that portion of the tree, and then uses GNU C macro routines to retrieve and collect symbol table information that identifies the variables defined and used in that portion of the tree. When CParse recognizes the end of a procedure, it uses database write handler routines to write analysis information collected for that procedure to the database. Unlike previous algorithms for computing control flow during the parse of the program, our algorithm handles structured transfers of control, such as break and continue statements, and unstructured transfers of control (goto statements).

We are currently transferring CParse technology to the GNU C++ parser to obtain analysis information for C++ programs.

3.2 Central Database and Database Handler Routine Library

Aristotle collects program analysis data from parsers and program analysis based tools, and stores the data in a database. In designing our database, we had two major goals. First, we wished to create a flexible interface between the central database and the tools. Second, we wanted a simple database model that could be easily enhanced or replaced by a more sophisticated database. To meet these goals, we developed a library of database handler routines that serve as the sole means of access to the central database.

Database read handler routines retrieve data from the database and return it to calling routines. Database write handler routines receive data from calling routines, and deposit the data into the database. Programmers can access the database by making calls to the appropriate handler routines from their applications. To facilitate creation of new database handlers, we have created common library routines and templates. Aristotle can be converted to work with a new database system by creating a new library of handler routines and linking tools with that new library; parsers and tools do not need to be modified to facilitate such a conversion.

At present, Aristotle’s central database is implemented as a collection of files. For a given C source file, file.c, handlers create assorted files named file.c.suffix, where suffix distinguishes the type of information contained in the file. Database files contain information on a per-function basis; thus for any function in file.c, handler routines can return information pertaining to just that function. An environment variable directs parsers and tools to a directory that contains a user’s database files. Each Aristotle user may have his/her own database directory, or users may share a database directory. At present, however, database handlers do not provide concurrency control.

3.3 Analysis Tools

Aristotle analysis tools perform analysis beyond that performed by Aristotle parsers. Most tools operate on single procedures and on groups of interacting procedures. Users can invoke analysis tools via menu selections or from the command line; programmers can also invoke most tools from applications programs, by linking their programs with libraries that contain the tools as callable subroutines. In the following sections, we describe the functionality and design of each of our tools, and give some implementation details.
S1. read i  
S2. sum = 0  
S3. done = 0  
P4. while i <= 5 and !done do  
S5. read j  
P6. if j >= 0 then  
S7. sum = sum + j  
P8. if sum > 100 then  
S9. done = 1  
else  
S10. i = i + 1  
endif  
endif  
endwhile  
S11. print sum

Figure 4: Example program segment (upper left), its control flow graph (lower left), its control dependence graph (upper right) and its data dependence graph (lower right).

3.3.1 Dataflow analyzers

Dataflow analysis gathers information about the definitions and uses of variables in a program. A definition is a variable access in which a variable’s value is changed; a use is a variable access in which a variable's value is fetched without being changed. For example, in Figure 4, statements S1 and S10 contain definitions of i, and statements S2 and S7 contain definitions of sum. Similarly, statements P4 and S10 contain uses of i, and statements P6 and S7 contain uses of j.

Aristotle gathers dataflow information at three levels. Local dataflow analysis gathers information about definitions and uses in single statements. For example, given the program of Figure 4, local dataflow analysis determines that i is defined in statement S1, and used in statement P4. Global or intraprocedural dataflow analysis computes the reachability of definitions and uses within single procedures. For example, given the program of Figure 4, reaching definitions analysis determines that the definition of i in statement S1 reaches statement P4 and thus, is used in P4. Interprocedural
dataflow analysis gathers information about definitions and uses of variables that reach across procedure boundaries, and accounts for global variables and procedure parameters.

Aristotle parsers collect local dataflow information. Aristotle also provides two dataflow analysis tools: (1) an intraprocedural dataflow analyzer (IntraDFA) and (2) an interprocedural dataflow analyzer (InterDFA). IntraDFA uses control flow graphs and local dataflow information to compute reaching definition and reachable use information for each procedure in a program, using an iterative dataflow framework [3]. InterDFA uses local and intraprocedural data and control flow information to create an interprocedural representation of the interacting procedures; it uses an iterative dataflow framework [8] to compute interprocedural dataflow information.

3.3.2 Control dependence analyzer

Control dependence information is useful for both compiler optimizations and program testing. For nodes X and Y in a control flow graph, X is control dependent on Y if Y has two (or more) exits where one of the exits always causes X to be reached, and the other exits may result in X not being reached. For example, in the program in Figure 4, S5 is control dependent on the true branch of the while loop in P4, and S10 is control dependent on the false branch of the if statement in P8.

Aristotle's control dependence analysis tool, CDAnalyze, accepts a control flow graph as input, and produces control dependence information using the algorithm presented in reference [4]. This algorithm uses the control flow graph to compute postdominator information, which is then used in the computation of the control dependences for the program. CDAnalyze uses database write handler routines to write control dependence information to the central database.

3.3.3 Graph builders

Aristotle's graph building tools read analysis information from the central database using read handlers, and use that information to construct data dependence graphs, control dependence graphs and program dependence graphs.

Data dependence graph builder

We often require information about the data dependences among definitions and uses of variables in a program. One type of data dependence is flow dependence. A flow dependence exists between statements S1 and S2 if S1 defines variable v and S2 uses v, and there is a path in the program from S1 to S2 on which v is not redefined, i.e., the definition of v in S1 reaches the use of v in S2. If statement S2 is flow dependent on statement S1, then (S1,S2) is a definition-use pair. For example, in the program segment shown in Figure 4, since the definition of i in S1 reaches the use of i in P4, (S1,P4) is a definition-use pair for i.

A data dependence graph contains nodes that represent program statements, and edges that represent data dependences between statements. Figure 4 shows a program segment and its data dependence graph representing flow dependence.

Aristotle's data dependence graph builder, BuildDDG, accepts a program as input, and constructs data dependence graphs for the procedures in that program. BuildDDG requires dataflow information produced by the dataflow analyzer, InterDFA, which it accesses through the central database using database handler routines. BuildDDG uses this information to compute definition-use pairs. It then uses these definition-use pairs to create data dependence edges between program statements, and uses database handlers to write the data dependence graph to the central database.

Control dependence graph builder

Control dependence information is frequently encoded in a control dependence graph [4], such as the graph shown in the upper right in Figure 4. A control dependence graph summarizes the control conditions necessary for a program statement to execute, and identifies the statements that are guaranteed to execute if a given statement has executed. Figure 4 shows a control dependence graph for a program segment.

A control dependence graph contains several types of nodes. Statement nodes, shown as ellipses in Figure 4, represent simple statements in the program. Predicate nodes, from which two labeled edges may originate, are represented as boxes in the figure. Circles represent region nodes, which summarize the control dependences for statements in the program. For example, in Figure 4, S5, P6, and P4 are control dependent on P4 being true. Without region node R2, there would be three edges labeled “T” from node P4.

Aristotle's control dependence graph builder, BuildCDG, uses read handlers to get control dependence information from the central database, and uses this information to produce a control dependence graph. BuildCDG produces control dependence graphs that contain region nodes, which summarize the control dependence conditions necessary to execute code associated with their successor nodes. The algorithm in reference [4]

2 Other useful types of data dependence are anti dependence and output dependence; we do not discuss them here.
creates region nodes in two phases. The first phase creates region nodes by traversing the postdominator tree for the routine whose control dependence graph is being built, and the second phase ensures that predicate nodes have at most a single region node successor for each truth value outcome of the predicate. Upon implementing this algorithm, we discovered that the algorithm occasionally produced graphs containing unnecessary region nodes: nodes that had only single control dependence predecessors that were themselves region nodes. We thus implemented an additional phase in our region node creation routine that finds and eliminates such unnecessary region nodes.

**Program dependence graph builder**

Another important program representation is a *program dependence graph*, which combines control and data dependence graphs. Our *Aristotle* tool, *BuildPDG*, constructs program dependence graphs. *BuildPDG* accepts as input a program, and constructs program dependence graphs for the procedures in that program, using the control and data dependence graphs produced by *BuildCDG* and *BuildDDG*, respectively.

### 3.3.4 Code instrumenter

Dynamic program-based testing often requires an execution trace of a program. An *execution trace* of a program, recorded for a particular test, may simply list the statements in the program that were executed by that test. Alternatively, an execution trace may list the entire path traversed through the program for that test. For example, consider the program in Figure 4. A test with inputs \(<i = 1, j = 3, 4, 5, 7, 8>\) has an execution trace that consists of statements \((S1, S2, S3, P4, S5, P6, S7, P8, S10, S11)\), while an execution trace of the path traversed consists of \((S1, S2, S3, P4, (S5, P6, S7, P8, S10, P4)S11)\). In either case, we must instrument the program by inserting probes that will produce the trace at runtime.

To generate instrumented code, we have developed a code generator, *CGen* [25]. Users of *CGen* specify the type of probes they want to insert into a program, and *CGen* generates an instrumented C program, with the appropriate probes, that can be compiled by any C compiler. *CGen* uses source code mapping information produced by *CParse*, and control dependence graphs produced by *BuildCDG*, which it obtains from the central database by calling database handler routines.

*CGen* instruments code for each procedure in a program by walking the control dependence graph for that procedure, producing a “walk sequence” in which each node occurs exactly once. Next, *CGen* uses this walk sequence to visit the nodes in the control dependence graph a second time. For each node it visits, *CGen* invokes a user-supplied routine, *UserTrace*, that may be used to generate probes or any other source code desired. Users can easily build variants of *CGen* by supplying different *UserTrace* routines. We have implemented versions of *UserTrace* that generate execution traces of statements and execution traces of complete paths.

### 3.3.5 Dataflow tester

A *dataflow tester* measures the dataflow coverage induced by a set of tests. The tester analyzes the path taken through the program on execution of a set of tests to determine which definition-use pairs are satisfied by those tests. A definition-use pair is *satisfied* if a test causes the program to execute a subpath from the statement containing the definition to the statement containing the use without executing any other statements containing definitions of the variable.

Our dataflow tester, *FATE* [17], provides intraprocedural and interprocedural dataflow testing. *FATE* also runs in parallel on one test, several tests or an entire test suite to decrease the time required to compute dataflow coverage.

*FATE* uses *CGen* to create two new versions of the program. The first is an instrumented version of the program that, when run with some input data, produces an execution trace. The second is an acceptor, which is a variant of the program that reads the execution trace, reconstructs the path taken through the program, and determines the satisfied definition-use pairs. *FATE* also uses *Parallel Virtual Machine* (PVM) [5] to spawn multiple client processes. If *FATE* is executing one test in parallel, each of these client processes finds the satisfied definition-use pairs for one variable. If *FATE* is executing several tests at a time in parallel, each of these client processes finds all the satisfied definition-use pairs for one test.

### 3.4 Viewing Tools

*Aristotle* displays analysis information through table viewers and graph viewers.

#### 3.4.1 Table viewers

Table viewers display program analysis information in tabular form. Table viewers currently provide tabular views of control flow graphs, control dependence graphs, data dependence graphs, program dependence graphs, symbol table information, local definition-use information, reachable use information and reaching definition information. Viewers display results to the screen, or
write them to files named by the user. Table viewers are accessible from the menu interface, or from the command line.

3.4.2 Graph viewers

Aristotle users view representations of program analysis based graphs using our graph display tool, Visigraph. Visigraph displays control flow graphs, control dependence graphs, data dependence graphs, and program dependence graphs for individual procedures. Visigraph will be extended to display interprocedural graphs for interacting procedures and entire programs.

Users of Visigraph can specify several options when they view graphs. For example, users can request that node types be shown with the nodes in the graph, or they can request that source code be displayed in the nodes. After a user specifies a graph type and viewing options, Visigraph uses database handlers to obtain the specified graph from the central database, and creates a graph representation that can be read by XGVG [15]. XGVG is a graph viewing tool that displays graphs using X-windows, and prints graphs using PostScript.

4 Future Components

This section describes additional tools that we are developing that use the Aristotle system.

4.1 Regression test selector

When a program is modified, we perform regression testing on the modified program to gain confidence that it is correct. To reduce the cost of regression testing, we prefer to reuse tests from existing test suites where possible. Selective retest algorithms select, from existing test suites, tests that should be rerun on a modified program. We use a selective retest algorithm [20, 21] that uses control and data dependence information about a program \( P \) and a modified version, \( P' \), to select all tests in the test suite for \( P \) that may produce different output in \( P' \), and hence expose errors in \( P' \).

We are implementing our selective retest algorithm as an Aristotle tool, called DejaVu. To use DejaVu, users create a test database that contains tests of a software product. For each test, the user invokes \( \text{CGen} \) to generate traces that track the code executed by the test. The user then uses Aristotle's control dependence graph builder and dataflow analyzers to analyze both the old and new versions of the software. DejaVu retrieves this control and data dependence information from the central database, retrieves test history information from the test database, and uses this data to select all tests that execute code modified for the new version of the software.

4.2 Static and dynamic program slicers

A slice of a program reveals the statements that affect a particular program point [27]. Slicing has many important uses such as debugging [26], integrating program variants [10], and testing [6]. A static slice of a program with respect to program point \( p \) and variable \( x \) consists of all statements and predicates of the program that might affect the value of \( x \) at point \( p \). For example, in Figure 4, the static slice on variable \( i \) at \( P_4 \) consists of statements \((S_1,S_2,S_3,P_4,S_5,P_6,S_7,P_8,S_9,S_{10})\). A dynamic slice consists of the statements necessary to preserve a program's behavior given a specific program's input [12]. In Figure 4, the dynamic slice on input \( < i = 5, j = 3 > \), at program point \( S_{11} \), and with respect to variable \( \text{sum} \) contains statements \((S_1,S_2,S_3,P_4,S_5,P_6,S_7,P_8,S_9,S_{10})\).

To provide slicing for users of Aristotle, we have designed both a static slicer, SSlice, and a dynamic slicer, DSlice. Users of these slicers can either access slice information for use in their programs or view the resulting slice. For a static slice, the user inputs a program, a program point and a variable; for a dynamic slice, the user inputs a program, a program point, a variable and a test.

Since both SSlice and DSlice require control and data dependence information about the program, they use BuildPDG. Further, since DSlice requires an execution trace of the program, it uses \( \text{CGen} \). SSlice computes a static slice for a program \( P \), a variable \( x \), and point \( p \) by walking backward along the control and data dependence edges of \( P \)'s program dependence graph; any statements that are visited may affect the value of \( x \) at \( p \), and are added to the slice. DSlice computes a dynamic slice by first executing a program \( P \) to generate an execution trace for a test \( T \). Then, DSlice uses the execution trace to compute the dynamic slice for \( P \) and \( x \) at \( p \).

4.3 Simulation translator

Simulation is an important application of computer science that uses a model to mimic the behavior of a system. One way to facilitate simulation is to extend a general purpose language to include simulation primitives. A new system for object-oriented process simulation, called SimPOL [18], consists of both a language and a translator. The language extends C++ to include simulation primitives like those in SIMULA [13]; the translator transforms SimPOL programs into C++ programs, which can be compiled on any existing C++ compiler.
We are developing a SimPOL translator SimTrans as part of the Aristotle system. Users of SimTrans write programs in SimPOL, and SimTrans produces a C++ program that implements the control required for the simulation. SimTrans requires control flow graphs and mapping information produced by the C++ parser. Modifications in the parser will allow the parser to recognize simulation primitives during the parse.

### 4.4 Menu Interface

Currently, Aristotle users can make use of a text-based menu interface. Using this interface, users can run each of Aristotle's analysis and viewing tools. However, we are also using the Motif toolkit [9] to implement a graphical user interface to the system.

### 5 Support Tools

To assist with the development and maintenance of Aristotle, we have implemented, or are currently implementing, a pair of support tools.

#### 5.1 Bug reporter and database manager

As we developed the Aristotle system, we needed a method of documenting bugs found in Aristotle and keeping track of the code modifications required to fix those bugs. To facilitate this process, we have developed a bug reporting and bug database management tool, called BugSy, that lets Aristotle developers report bugs and maintain a bug database. BugSy is accessed through a text-driven menu system that lets a user report a bug, investigate a bug, browse existing bug reports, or obtain a copy of a bug. BugSy's database management system contains a bug report for each bug listing the description of the failure, the steps required to recreate the failure, and the method used to fix the bug. All files needed to recreate the bug are kept with the bug report, and the status of each bug is tracked at all times.

#### 5.2 Test suite database manager

Test set development requires substantial human effort, both in generation of tests, and verification of test results. It makes economic sense to retain the work performed in test set development, for use in retesting software after it is modified. To facilitate test set reuse, we must save information about tests, such as inputs, expected results, and history information that describes the code coverage attained by the tests. This information helps us automate regression test selection, test execution, and verification of test results.

Aristotle's test suite database management system will provide a method for archiving tests, test execution histories, and test results. DejaVu will use information on tests and their test histories to select tests for regression testing. Test information will be available for use by other testing tools, such as capture/playback facilities and test suite reduction tools.

### 6 Conclusions

We have presented an overview of our program analysis system, Aristotle. Aristotle analyzes single procedures or entire programs, and places analysis data in a database. Tools access this database to perform a variety of program analysis and software engineering tasks. We are also implementing software engineering tools that support Aristotle development and maintenance.

Aristotle simplifies the task of constructing program analysis and software engineering tools. We have successfully used Aristotle to create several such tools; additional tool development is planned.

### Acknowledgement

We wish to thank Angela Holland, Nandukumar Sankaran, Kanupriya Tewary, Rama Tumula and Amar Yalavarthy who contributed to the design and implementation of Aristotle. We would also like to thank Ka-Wing Wong for his careful review of the paper and helpful suggestions that improved its presentation.

### References


---

Copyright (C) 1991, 1994 O'Reilly & Associates, Inc.
Languages and Systems, vol. 9, no. 3, July 1987, pp. 319-349.


