Aristotle: A System for Research on and Development of Program-Analysis-Based Tools

Mary Jean Harrold
Department of Computer and Information Science, The Ohio State University.
and
Gregg Rothermel
Department of Computer Science, Oregon State University.

SUMMARY

`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 data-flow, and symbol table information for C and Java programs. `Aristotle` tools use the data provided by the front end to perform a variety of tasks, such as data-flow and control-dependence analysis, data-flow testing, regression test selection, graph construction and graph viewing. Parsers and tools use database access routines to store information in, and retrieve it from, a data repository. Users can view analysis data textually or graphically. A user interface provides menu-driven access to tools; many tools can also be invoked directly from applications programs. Most of `Aristotle`'s components function on single procedures and entire programs. We use `Aristotle` as a platform for developing and experimenting with program analysis, maintenance, and testing tools.

KEY WORDS: program analysis, software engineering, CASE, experimental systems

INTRODUCTION

Many techniques for automating software engineering tasks rely on program analysis information – information that is obtained by analyzing software. For example, several regression testing techniques [1, 4, 5, 9, 22, 23] use control- and data-dependence information to determine the retesting required after a program is modified. Techniques such as data-flow testing [8, 14, 18, 20], test-case generation [13], and dynamic execution profiling [3] require control-flow information. Tools for program understanding and impact analysis use program-slicing techniques [11, 19, 26], which, in turn, require various forms of dependence and flow information. Before researchers can implement and experiment with program-analysis-based testing and maintenance techniques such as these, they must acquire the prerequisite program-analysis tools. In general, it is difficult and time-consuming to construct such tools. An alternative is to obtain analysis tools from other sources; however, there are currently no program analysis systems that gather a sufficient range of information, are available with source code to researchers, and are documented and designed to facilitate extension and substitution of experimental components. In the absence of implementations and empirical results, it is difficult to motivate the use of program-analysis-based testing and maintenance techniques, and difficult to argue that they could usefully be transferred into practice.

A number of systems provide, or could support the development of, program analysis tools. The Arcadia project and its ProDAG subsystem [12, 21] provide program-analysis tools for use with Ada programs; these include tools for constructing control-flow graphs and computing data- and control-dependencies. Sage [16] creates abstract syntax trees for C and C++ programs, and provides several utilities that can be used to create new analysis tools. Aria [6] generates testing and analysis tools for C and C++ programs from abstract-syntax-graph representations (an abstract syntax tree with semantic information) of the programs, and can be used to generate many different types of dependence graphs. Rigi [17] provides a toolkit for program understanding that can be used for activities such as reverse engineering of large systems. SUIF [27], an infrastructure for support of compiler research for high-performance systems, provides a front end that generates abstract syntax trees for C programs. Each of these systems supports, to some degree, research on and development of program-analysis-based tools; however, each also has drawbacks. Of the systems described, only Arcadia (through ProDAG) and Aria provide data-flow and dependence analysis tools. However, Arcadia operates only on Ada programs, and although Aria can generate tools to process C and C++ programs, it is accessible only through license agreement and requires the Cfront C++ compiler. Moreover, to the best of our knowledge, none of these systems currently provides tools for interprocedural analysis, that are required to facilitate analysis of interacting procedures and complete programs. Finally, some of the systems, such as the Cfront C++ compiler, which is required for Aria, are not publicly available.

To provide support for research on, and development of, program-analysis-based software engineering tools and techniques, we have developed a system that we call `Aristotle`. We have identified several requirements for this system:

- The system must provide *core functionality* to users. Core functionality includes both basic program-analysis tools, such as tools for constructing control-flow graphs, and more complex tools, such as tools for calculating interprocedural data-flow analysis information. Core functionality also includes tools for viewing analysis information, and user interface access to tools and viewers.
- The system must be capable of serving as *an extensible platform* for developing and integrating new tools and techniques. The system architecture must facilitate tool access to analysis data produced by core tools, and minimize the complexity of interactions between tools, so that new components can be "plugged in" to the system. The architecture should also support, to the degree possible, the provision of language-independence in tools.
- The system must be *general and robust* enough to accommodate experimentation on a significant class of software artifacts. The system must support analysis of programs written in real and practical languages, and not be restricted to impractical subsets of the features of those languages. The system must perform intraprocedural analysis (analysis of procedures considered individually), and also interprocedural analysis (analysis of interacting procedures). The system must operate in a useful range of development environments, suitable to its intended user base (primarily, university researchers and students).
- The system must be *available to and accessible by* other researchers. In this context, accessibility means that the system must be equipped with documentation, installation procedures, source code, and a collection of sample software artifacts that can be used for experimentation.

`Aristotle` fully or partially meets the foregoing requirements. `Aristotle` operates on C and Java programs; its core functionality is composed of several program analysis tools, including a control-flow graph constructor, intraprocedural data-flow and control-dependence analyzers, dominator calculation routines, code instrumenters, test information management routines, and intraprocedural and interprocedural graph builders. `Aristotle` also provides a menu-based user interface, and tools for viewing and retrieving analysis information. `Aristotle` employs a data-centered architecture that can eliminate direct coupling of analysis tools, and provides language independence at the tool level. Furthermore, by restricting database access to specific routines, `Aristotle` encapsulates data storage details, and decouples analysis tools from those details. Within this architecture, tool designers can build new tools, and link them with database access routines, leaving other tools unaffected. However, we also design our tools as callable modules; tool designers can directly link modules together or with their tools at the call level if desired. `Aristotle` currently handles most C and Java constructs; work is ongoing to extend its range of applicability. `Aristotle` is publically available, with source code, extensive documentation, installation instructions, and sample artifacts, through our Web site.[1]

We are developing `Aristotle` incrementally. We have released three versions of the system, and, as of this writing, are constructing our fourth release and planning future releases. We have used `Aristotle` as a platform from which to construct several program-analysis-based software engineering tools, including a regression test selector, a metrics analyzer, and a data-flow tester. Additional analysis tools, including procedure-level dependence analyzers and program slicing tools, are currently under development. These tools, in turn, will support the development of tools for impact analysis and program understanding. Other tools, including generic code-instrumentation tools and class testers, are in the design stage.

In the process of developing `Aristotle`, we have made numerous decisions about requirements, design, and implementation details. Many of these decisions have been fortuitous; others have not. In this paper, we first describe the current `Aristotle` system, including its overall architecture and the tools and components that constitute its forthcoming release. We then discuss lessons that we have learned about system

[1] See http://www.cis.ohio-state.edu/~harrold/projects/aristotle.html or contact harrold@cis.ohio-state.edu.

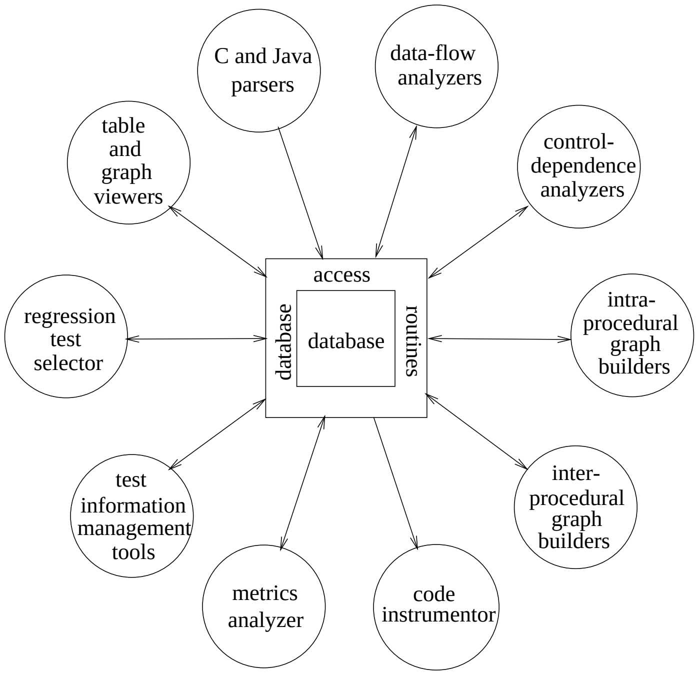


Figure 1: `Aristotle`'s data-centered architecture.

requirements, design decisions, and implementation decisions, that are shaping future work on `Aristotle`, and that may assist system developers whose goals are similar to those of ours. Finally, we discuss our ongoing development efforts with the system.

SYSTEM OVERVIEW

`Aristotle` employs a data-centered architecture, as depicted in Figure 1. With the exception of parsers that take source code as input, all system components use *database access routines* to retrieve data from a database. Most system components also use database access routines to deposit new data into the database. We further discuss individual components depicted in the figure in this and the next section.

From a functional standpoint, `Aristotle` can be viewed either as a stand-alone system that provides program-analysis information to the user or as a platform for the development of software engineering tools; Figures 2 and 3 use dataflow diagrams[2] to represent the flow of information through `Aristotle` for these two functional views of the system. Both views contain the following components:

- **Parsers** parse source code, gather basic analysis information about that code, and pass that information to database access routines.
- **Analysis tools** use analysis data obtained from database access routines, and parameters obtained through the user interface, to perform various types of program analysis. The tools output data to database access routines.
- **Database access routines** manage access to the `Aristotle` database.

The stand-alone view (Figure 2) also contains the following components:

[2] In a *dataflow diagram*, circles (bubbles) represent processing elements, parallel lines represent data stores, rectangles represent external entities, and directed edges represent the flow of data between system components.

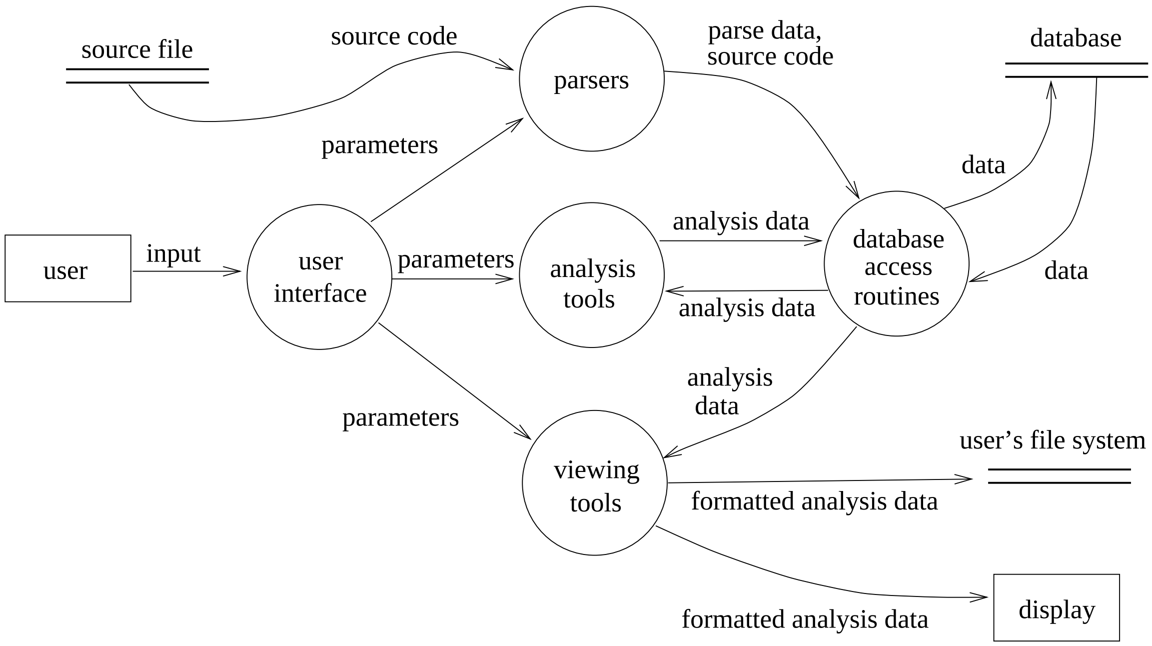


Figure 2: Dataflow diagram depicting `Aristotle` as a stand-alone system.

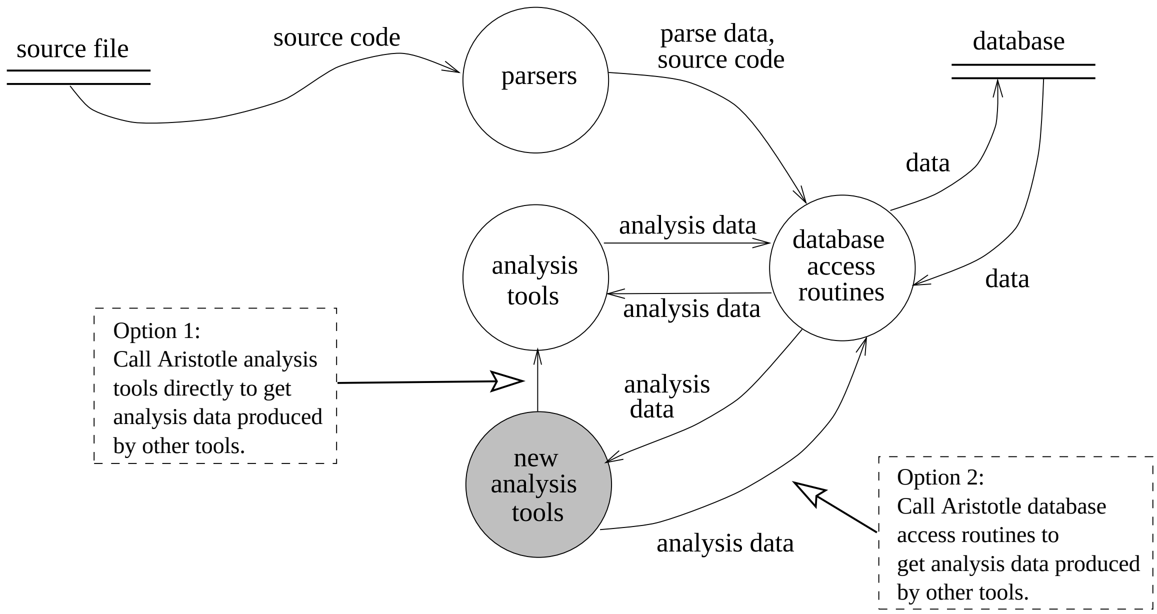


Figure 3: Dataflow diagram depicting `Aristotle` as a platform for development of new analysis tools.

- **A user interface** transforms user input into commands that invoke Aristotle parsers, analysis tools, and viewing tools.
- **Viewing tools** let users view program analysis data. The system supports two classes of tools: table viewers and graph viewers. These tools use access routines to retrieve analysis information from the database; they then format that data for viewing, and display the formatted data.

Tool builders who use Aristotle design their tools to invoke database access routines, thus obtaining analysis information from the database.

Aristotle is written primarily in C; we are currently implementing some new tools in C++. The system contains approximately 400,000 lines of source code excluding blank lines and comments. Aristotle operates and has been tested on SUN and HP workstations running Unix. System documentation includes an installation manual, system overview, user manual, and programmer manual.

SYSTEM COMPONENTS

Parsers

Aristotle's parser for C accepts a C source file as input, and outputs control-flow, local data-flow, and symbol table information about each procedure in the file. We created Aristotle's C parser by inserting actions into the GNU C parser. As the parser recognizes each language construct, these actions pass 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 symbol table information that identifies the variables defined and used in that portion of the tree. When the parser recognizes the end of a procedure, it uses database access routines to write analysis information collected for that procedure to the database. Our algorithm for computing control-flow information handles both structured transfers of control, such as **break** and **continue** statements, and unstructured transfers of control, such as **goto** statements.

Aristotle's parser for Java analyzes Java methods, classes, and applications programs. We used the Unix Yacc tool to create this parser by inserting actions that call our control-flow graph construction routines. We are extending the parser so that it will collect symbol table and local data-flow information.

Analysis Tools

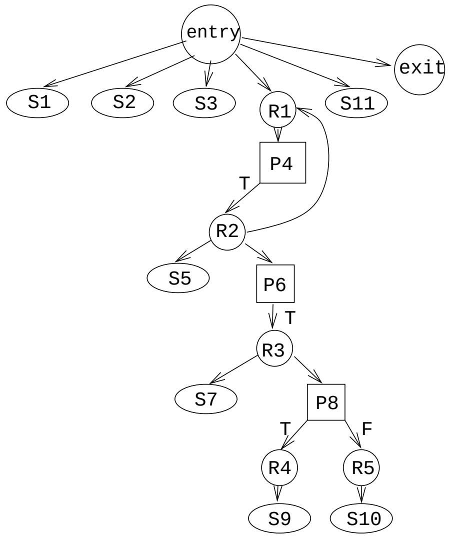
Aristotle analysis tools perform analysis beyond that performed by Aristotle parsers. Most tools operate on single procedures and groups of interacting procedures. Users invoke analysis tools by menu selections or from the command line; programmers can also invoke many tools from applications programs by linking their programs with libraries that contain the tools as callable subroutines.

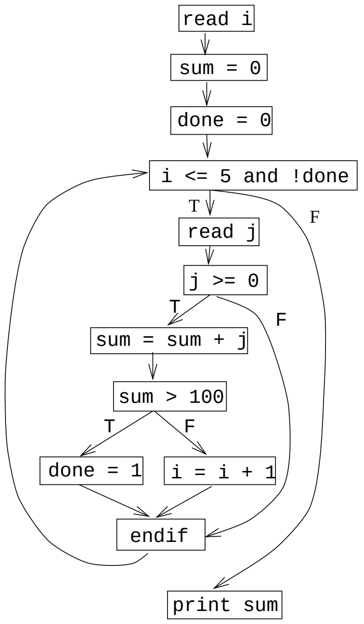
Data-flow analyzers

Data-flow analyzers produce information that is useful for many software maintenance and testing tools. Data-flow 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.

At present, Aristotle gathers data-flow information at two levels. Aristotle's parsers identify *local* data-flow information about definitions and uses in single statements. For example, given the program of Figure 4, local data-flow analysis determines that i is defined in statement S1, and used in statement P4. Aristotle's intraprocedural data-flow analyzers perform *intraprocedural* data-flow analysis, which computes the reachability of definitions and uses within single procedures using an iterative data-flow framework. 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.

```
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
```





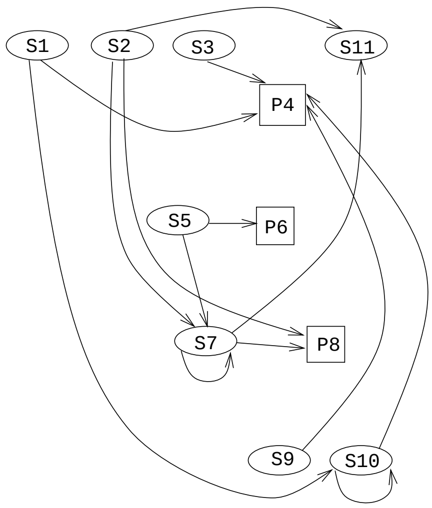


Figure 4: Example program segment (upper left), its control-flow graph (lower left), its control-dependence graph (upper right) and its data dependence graph (lower right).

Control-dependence analyzers

Control-dependence information is useful for both compiler optimizations and software engineering. 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 exit 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 intraprocedural control-dependence analysis tool accepts a control-flow graph as input, and produces intraprocedural control-dependence information using the algorithm presented in Reference [7]. This algorithm uses the control-flow graph to compute postdominator information, which is then used to compute control-dependencies for the program. The control-dependence analyzer uses database access routines to write control-dependence information to the database.

Intraprocedural graph builders

`Aristotle`'s intraprocedural graph building tools use access routines to read analysis information from the database, and use that information to construct a variety of intraprocedural graphs.

A *data-dependence graph* contains nodes that represent program statements, and edges that represent data dependencies between statements. Figure 4 shows a program segment and its data-dependence graph representing flow dependence.[3] 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, that is, 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, because the definition of i in S1 reaches the use of i in P4, (S1,P4) is a definition-use pair for i.

`Aristotle`'s data-dependence graph builder uses access routines to retrieve intraprocedural data-flow information from the database, and uses this information to calculate definition-use pairs and produce data dependence graphs for procedures in a source file. The tool uses database access routines to write the data-dependence graph to the database.

Control dependence information is frequently encoded in a control-dependence graph [7], such as the graph shown in the upper right in Figure 4. A *control-dependence graph* summarizes the control conditions necessary for a statement to execute, and identifies statements that are guaranteed to execute if a given statement has executed. Figure 4 shows a control-dependence graph for the program segment given in the figure. A control-dependence graph contains several types of nodes. Statement nodes, shown as ellipses in Figure 4, represent simple statements. Predicate nodes, from which labeled edges originate, are represented as rectangles. Circles represent region nodes, which summarize the control-dependencies for statements in the program. For example, 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 uses access routines to retrieve control dependence information from the database, and uses this information to produce control-dependence graphs. That tool uses the algorithm of [7] to create region nodes in two phases. The first phase creates the 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 that contained unnecessary region nodes: nodes that had only single control-dependence predecessors that were themselves region nodes. We implemented an additional phase in our region node creation routine to find and eliminate these nodes.

Because some algorithms that use control dependence graphs do not require region nodes, we recently implemented another graph builder that creates control-dependence graphs without region nodes.

Another important program representation is a *program-dependence graph*, which combines control- and data-dependence graphs. `Aristotle`'s program-dependence graph builder constructs these graphs. The program-dependence graph builder uses access routines to retrieve both control- and data-dependence information from the database, constructs the program-dependence graph by merging this data, and uses access routines to write the graph out to the database.

Interprocedural graph builders

A *call graph* models the calling relationships between entities in a program, and is useful for flow-insensitive analyses and program understanding. `Aristotle`'s call graph extractor uses access routines to retrieve data on calling relationships from the database, builds a call graph that models those relationships, and uses access routines to write the resulting graph to the database. Source files may contain calls to external routines, including standard library routines, that cannot themselves be analyzed for calling relationships. The call graph builder includes such calls, and nodes for the called routines, but marks them as unexpanded; thus, routines that use the call graph can distinguish functions for which further information is available from those for which it is not.

An *interprocedural control-flow graph* is useful for analyses such as interprocedural data-flow analysis [10] and program slicing [2]. To provide support for these techniques, we incorporated an interprocedural control-flow graph builder into `Aristotle`. This builder uses access routines to retrieve control-flow graphs

[3] Other useful types of data dependence are *anti dependence* and *output dependence*; we do not discuss them here.

for functions from the database, connects them to form the interprocedural control-flow graph, and uses access routines to write the resulting graph to the database.

Similarly, the *interprocedural control-dependence graph* builder retrieves control-dependence graphs for individual functions from the database, performs analysis on the control-flow graphs for those functions to obtain prerequisite information, and uses that information and the control-dependence graphs to construct and output an interprocedural control-dependence graph.

Code instrumenters

Dynamic program-based testing often requires the ability to collect execution traces of programs. An *execution trace* of a program, 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 depicted in Figure 4. A test of that program with inputs $< i = 1,\ j = 3, 4, 5, 7, 8 >$ has an execution trace that lists 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)^5,S11⟩. In either case, to obtain the trace we must instrument the program by inserting probes that will produce the trace at runtime.

To generate instrumented code, we developed a code instrumenter [25]. Users of this tool specify the type of information they wish to collect about program execution, and the tool generates a C program instrumented with probe statements that will collect that information, that can be compiled by any C compiler. Users can build variants of the instrumenter by supplying user trace routines that generate the desired type of instrumentation. We have implemented versions of the user trace routine that generate lists of statements or branches executed, and a version that generates information from which complete execution paths can be regenerated.

Metrics analyzer

When we use `Aristotle`, we may require statistics about the analysis information that is gathered by various tools. We might want to know, for example, the number of executable statements, the sizes of graphs, the numbers of statements of a certain type in a program, or the number of procedures that contain certain constructs. To facilitate collection of this information, we implemented a metrics analyzer. The analyzer uses access routines to retrieve raw data about a program from the database; depending on the type of information that the user requests, the analyzer computes and outputs metrics in various formats.

Test information management tools

Instrumented programs output trace information that shows, for a particular test, which objects (e.g., statements or branches) in a program were traversed by that test. More useful for many applications is information that lists, for each test in a test suite, the objects that were executed by that test. We call this a *test history*; `Aristotle` provides tools that manage this information.

To construct test history information, we developed a *test history builder*. This routine takes a directory of test trace files as input, reads these files, and creates a test history file that reports the information.

To support applications that require *edge trace* information, rather than construct another code instrumenter to record this information, we constructed an `Aristotle` tool that reads a test history that contains branch trace information, uses access routines to read the control-flow graphs for the traced program, and propagates branch coverage information around the flow graphs, obtaining edge coverage information.

Trace files and test history files do not reside in the `Aristotle` database; however, access to these files is handled similarly: access routines manage all accesses to the test information in trace and history files, viewers print the information as formatted output.

Regression test selector

When a program is modified, we use regression testing to validate the modified code. To reduce the cost of regression testing, we prefer to reuse tests from existing test suites whenever possible. *Regression test selection* algorithms select, from existing test suites, tests that should be rerun on a modified program. We have developed a regression test selection algorithm [24] that uses control-flow information about a program P and a modified version, P', to select tests in the test suite for P that execute code that has been modified for P'.

We implemented our regression test selection algorithm as an `Aristotle` tool. To use the regression test selector, users create a test database that contains tests of a software product. Following product release, and prior to the beginning of the next test cycle (when time for regression testing becomes a critical factor), users instrument their program and execute existing tests, collecting test trace information about each test. At regression test time, users invoke `Aristotle` to build control-flow graphs for the original and modified versions of the program. Our regression test selector traverses these graphs, and uses test trace information to identify tests that execute modified code.

Data-flow tester

Data-flow testing is a white-box approach that uses the data-flow information about a program to guide the selection of a test suite. `Aristotle`'s data-flow tester lets a user measure data-flow coverage of a C program for a test set T. The data-flow tester facilitates data-flow testing for single procedures or interacting procedures, and accounts for aliasing due to reference parameters and single-level pointers.

We implemented the data-flow testing tool using abstract execution. On invocation with a program P, the tool produces an instrumented version of P and a data-flow tester for P. When the system executes the instrumented version of P, it produces a small trace of the significant events that occurred during P's execution with some test t in T. The data-flow tester reads the significant events file and determines the data-flow coverage achieved by t. Users can provide additional tests until the desired coverage is achieved.

Database Access Routines

`Aristotle` collects program-analysis data from parsers and program-analysis tools, and stores the data in a database. In designing our database and database access protocols, we had two major goals. First, we wanted to create an interface between the database and the tools that could be modified as system requirements change. Second, we wanted to implement an initial database quickly, but later be able to replace it with a more sophisticated database, without being forced to rewrite our tools. To meet these goals, we developed a library of database access routines that serve as the sole means of access to the database. *Read access* routines retrieve data from the database and return it to calling routines. *Write access* routines receive data from calling routines and deposit the data into the database. Tools access the database by making calls to the appropriate access routines. To facilitate creation of new database access routines, we created common library routines and templates. `Aristotle` can be converted to work with a new database system by creating a new library of access routines and linking tools with that library; parsers and tools do not need to be modified to facilitate such a conversion.

We implemented `Aristotle`'s current database as a collection of files. For a given source file, `file.[lang-type]`, where `[lang-type]` is currently "c" or "java", access routines create assorted files named `file.[lang-type].suffix`, where `suffix` distinguishes the type of information contained in the file. Information in files is typically organized on a per-function basis; for any function in `file.[lang-type]`, access routines 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 or her own database directory, or users may share a database directory. At present, however, database access routines do not provide concurrency control.

User Interface

`Aristotle` tools can be accessed from the command line; thus, users can write scripts that invoke sequences of tools. We have used this approach in cases where we needed to batch-process large numbers of source files for purpose of experimentation.

To provide more user-friendly access to `Aristotle` tools, however, we have also created a menu interface. `Aristotle`'s menu interface is a simple, text-based interface, that lets users run most of `Aristotle`'s tools, specifying options that tailor tool behavior. For the occasional or novice user, the menus simplify use of the system, obviating the need to remember tool names and their command line interfaces.

The `Aristotle` menus invoke tools by using the Unix `system` command, passing as a parameter the command line invocation of the tool; that command creates a child process that execs the shell to execute the string sent as a parameter to the command. By this approach, we avoid having to link the menus with tools as callable subroutines. Moreover, developers working on new versions of tools can force the menus to use their versions by making the location of those versions appear earlier in their path. A configuration file lets users further tailor the behavior of the text menus.

Viewing Tools

`Aristotle` uses table viewers and graph viewers to display analysis information. Table viewers 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 and reaching definitions information, and dominator information. Table viewers display results to the screen, or write them to files named by the user.

`Aristotle`'s graph viewers provide graphical views of control-flow graphs, control-dependence graphs, data-dependence graphs, program-dependence graphs, and dominator or postdominator information for individual procedures. Users of the graph viewers 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, the graph viewers use database access routines to obtain the specified graph from the database, format that data for display by the `XVCG` [15] graph viewing tool, and invoke that tool.

EXPERIENCE WITH ARISTOTLE

From our experience with the construction and use of `Aristotle`, we have made a number of discoveries that will guide our ongoing work on the system, and may assist others who wish to build similar systems.

- `Aristotle` facilitates construction of, and experimentation with, software engineering tools. Students in our graduate-level software engineering courses have been able to construct and experiment with new tools by using `Aristotle` as a platform for development. For example, students have implemented data-flow analyzers, interprocedural graph constructors, and metrics analyzers that use analysis data computed by existing analysis tools. Graduate-level researchers have also been able to implement and experiment with new software engineering techniques by using `Aristotle` as a platform for development. Our regression test selection tool, described above, was implemented in one week, once the prerequisite `Aristotle` tools were available.

  One reason for `Aristotle`'s success in this area is simply that it provides core functionality that is prerequisite for other tools. However, we believe that `Aristotle`'s data-centered architecture is also responsible for this success; the architecture decouples tools from each other, and lets tool builders focus on their particular algorithms or functionality without worrying about interfaces with other tools. Furthermore, by separating functionality, `Aristotle` facilitates construction and comparison of various versions of tools, that could not fairly be compared if they combined functionality.

- `Aristotle` facilitates incremental development and prototyping. We have been able to create initial prototypes of components, and later replace them with improved versions, without impacting other tools – again because of the data-centered architecture. For example, our initial versions of tools that implement intraprocedural data-flow analysis, intraprocedural control-dependence analysis, and code instrumentation have been replaced by more efficient and more robust versions of the tools without requiring modifications to tools that interact with these new versions.

- `Aristotle`'s data-centered architecture may impact tool performance. When a tool depends on results of prior tools, a data-centered architecture forces computations to perform more file I/O than would an architecture that chained tools together directly. This impact has not, thus far, been noticeable, in part because we are not yet analyzing huge systems. In general, the flexibility we gain by the data-centered architecture outweighs efficiency losses.

Nevertheless, because we may wish to be able to reduce the file I/O impact, we have designed many existing tools, and are designing new tools, as callable subroutines. We then write driver routines that call database read access routines, call the analysis subroutines, and call database write access routines. This layered tool structure creates tools that can communicate directly with each other using calls, if required.

- `Aristotle`'s limited-functionality tools are more suitable for our purposes than tools that perform multiple analysis tasks. We constructed some of our earlier tools such that they performed multiple functions. For example, in addition to computing control-flow, local data-flow, and symbol table information, our first parser also computed dominator and control-dependence information. Combining functionality can improve efficiency by reducing file I/O in the data-centered architecture; however, simple independent tools are easier to implement and maintain, and facilitate comparative empirical studies of performance.

- Creating a general and robust program analysis infrastructure that functions across arbitrary software systems in supported languages requires much time and effort. Although generality and robustness are ultimate goals for `Aristotle`, these qualities are not prerequisites for integrating new tools into the system, or for obtaining significant experimental results. Instead, we need only create a system that functions correctly on our repository of artifacts. As we collect new artifacts, we discover faults in the system, and correct the system to process those artifacts. In this fashion, the system becomes more robust over time.

- Using the GNU C compiler, `gcc`, as a basis for `Aristotle` has presented more disadvantages than advantages. Because we had access to `gcc`, we utilized its parser to build the `Aristotle` C parser. At first, this approach appeared to give us a quick way to gather the required information about programs. However, as `Aristotle` began to evolve, we discovered several drawbacks to using `gcc`. First, `gcc` is not well documented, and its complex code presents a steep learning curve. Second, we cannot easily keep up with `gcc` releases. Attempting to do so distracts us from more worthwhile efforts; however, failing to do so creates difficulties when underlying hardware outpaces a given `gcc` release and necessitates migration to a later release. Finally, `gcc` also generates executable code. Our analysis system does not require this functionality; thus, code generation serves as an unwanted side-effect, increasing the size of our system, and reducing system portability.

- `Aristotle`'s flat-file database will hinder the processing of large amounts of data; it is also cumbersome for handling simple queries. A well-designed database management system could improve both storage of, and access to, analysis data.

ONGOING AND FUTURE WORK

Our experience with `Aristotle` and our development of new program-analysis-based software engineering techniques has motivated our ongoing work on `Aristotle`, and led us to set some goals for future work.

C Parser. We are currently planning the design and implementation of a new parser for C. This parser will compute the same information as the current tool. However, it will not be integrated into `gcc`. This new design will make `Aristotle` more portable across operating systems, and greatly reduce the size of the system.

Interprocedural analyzers. We are currently designing an interprocedural data-flow analyzer, which will gather information about definitions and uses of variables that reach across procedure boundaries, and account for global variables and procedure parameters. The interprocedural data-flow analyzer will use local and intraprocedural data- and control-flow information to create an interprocedural representation of the interacting procedures.

We are currently constructing an interprocedural control-dependence analyzer that will compute control dependencies between statements in different procedures, and a procedure-level control-dependence analyzer that will compute control dependencies between interacting procedures.

Database management system. We are researching database management systems, with the aim of replacing the current flat-file database with such a system.

Instrumenters for C and Java. We are writing a code instrumenter for Java; this instrumenter will rely on libraries of generic abstract-syntax-tree walking routines and probe routines to maximize portability to other languages. We will evaluate our approach by using the routines to write an improved C instrumenter.

Web-based menus. We are investigating possible designs for a web-based `Aristotle` menu interface, that will enable users to invoke `Aristotle` functionality remotely over the Web.

Object-oriented migration. We are designing new components and incrementally replacing system components with object-oriented components written in C++ and Java.

Acknowledgments

This work was partially supported by grants from Microsoft, Inc. and by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University and Ohio State University. Other current and past members of the Aristotle Research Group have worked on the design and implementation of `Aristotle`: Adam Capes, Madhu Chetuparambil, Ning Ci, Angela Holland, Jim Jones, Sudha Krisnhamurthy, Chaitanya Kodeboyina, Loren Larsen, John Lloyd, David Nedved, Philip O'Connor, Melanie Page, Sujatha Sathi, Manvinder Singh, Laurie Smith, Michael Smith, Ryan Stradling, Kanupriya Tewary, Rama Tumula, and Amar Yalavarthy.

References

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 348–357, September 1993.

[2] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, March 1996.

[3] T. Ball and J. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[5] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 41–50, November 1992.

[6] P. Devanbu, D. Rosenblum, and A. Wolf. Generating testing and analysis tools with Aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] P. E. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[9] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 299–308, November 1992.

[10] M.J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):442–460, July 1996.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[12] R. Kadia. Issues encountered in building a flexible software development environment. In *Proceedings of the 5th Symposium on Software Development Environments*, pages 169–179, December 1992.

[13] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[14] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–54, May 1983.

[15] I. Lemke and G. Sander. *Visualization of Compiler Graphs*, May 1994.

[16] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *8th International Parallel Processing Symposium*, pages 75–84, April 1994.

[17] J. Mylopoulos, M. Stanley, K. Wong, M. Bernstein, R. DeMori, G. Ewart, K. Kontogiannis, E. Merlo, H. Müller, S. Tilley, and M. Tomic. Towards an integrated toolset for program understanding. In *Proceedings of CASCON '94*, pages 19–31, November 1994.

[18] S.C. Ntafos. An evaluation of required testing element strategies. *Proceedings of the 7th International Conference on Software Engineering*, pages 250–256, March 1984.

[19] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, March 1984.

[20] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[21] D. J. Richardson, T. O'Malley, C. T. Moore, and S. L. Aha. Developing and integrating ProDAG in the Arcadia environment. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 109–119, December 1992.

[22] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 358–367, September 1993.

[23] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

[24] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), August 1996.

[25] K. Tewary and M. J. Harrold. Fault modeling using the program dependence graph. In *IEEE International Symposium on Software Reliability '94*, pages 126–135, November 1994.

[26] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[27] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tsend, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, December 1994.