SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1(1), 1–4(JANUARY 1988)

Implementing Prototype Testing Tools

DICK HAMLET

Center for Software Quality Research, Department of Computer Science,
Portland State University, Box 751, Portland, OR 97207; hamlet@cs.pdx.edu

SUMMARY

Testing tools are software analyzers that use information from particular executions of a program as well as information about a specification and the program text itself. Research prototypes of such tools are essential to investigate the ideas they embody. Often, hand calculation is so tedious and error-prone that an investigator cannot obtain any intuition about his or her ideas without an implementation to aid in experiments. Traditionally, such tools have been implemented in conventional high-level languages (e.g., C, Pascal), a process that takes more time than a prototype should. The technology of compiler generators and logic programming, applied to the idea of self-instrumenting programs, drastically shortens the prototype cycle.

This paper describes a general method for implementing prototype tools, gives examples of several old and new testing techniques fitted into the method, and discusses the ease with which such prototypes may be changed.

KEY WORDS Instrumented programs Test coverage Testing tools

INTRODUCTION

Program testing is the art of executing software on individual input values to learn about its behavior. Both "executing" and "individual input values" are important; the art enters when the person conducting the test must pick the inputs. A computer does the executing. The results of the test executions are also important; it should not be part of the art that the person must guess at their correctness, but this is often true today. A program-testing tool is itself a piece of software, whose purpose is to aid the human tester, to automate part of the testing, to generate test input values, or to perform analysis on the test results, etc. What distinguishes a testing tool from other development-support software is the presence of program executions.

Research prototypes of program-testing tools are often essential in developing testing techniques. Systematic testing involves not only many test cases, but information from the program specification, information about the program structure, and details of the execution history. For a typical technique, the bookkeeping tasks are so extensive that hand simulation is literally impossible, and to gain understanding of the method's strengths and weaknesses requires experiments with a working test tool. Unfortunately, the difficulty of understanding a new method means that such a tool will need to change as the experiments suggest modifications.

It is common practice to implement research prototypes in a conventional programming language like Pascal (cf. ASSET[1], STAD[2]) or C (cf. TACTIC[3], ATAC[4]). No matter how "quick

Work supported by National Science Foundation grants CCR-8822869 and CCR-9110111

0038–0644/88/010001–04$7.00
Received 1 March 1988
Revised 25 March 1988

and dirty" the implementation is, the prototype takes months to write, and is as hard to debug or modify as any medium-sized program. An alternative to conventional programming is the use of a self-contained language/environment within which the test method is defined and implemented[5]. We propose a scheme with many of the advantages of the special-purpose environment, but using only three established computing technologies:

(1) *Self-instrumented programs.* Instead of monitoring test executions of the program under an interpreter, it is usually possible to outfit the program with monitoring statements in its own language, interspersed with the original program statements in such a way that when the program executes, the added statements collect needed information. The instrumented program may analyze the information at the conclusion of execution, or analysis may be performed off-line. This technique has been in use for at least 15 years[6].
(2) *Table-driven parser generators.* Compiler compilers are widely available that construct a parser for a programming language from the language grammar. (The UNIX system compiler compiler used here is a combination of the tools called *lex*[7] and *yacc*[8].) Compiler compilers usually allow some form of context-sensitive syntax-directed translation, so that as the constructed parser identifies the input source program, arbitrary actions can be performed. In a compiler, these actions may build a symbol table; when a parser is used to create self-instrumented programs, extra actions create the instrumentation statements.
(3) *A logic-programming language.* Languages like Prolog have several important advantages in creating prototype analyzers. Prolog can express facts about programs and executions in a database fashion, and can be used to query those facts interactively. The declarative Prolog programming style lends itself to describing software analysis such as testing methods. Prolog has been used in this way for a static FORTRAN analyzer[9].

We have combined these technologies to achieve very rapid generation of program-analysis tools, which are easy to change. The time from conception to running prototype is an order of magnitude less than for conventional development in a high-level language. Changes in the language analyzed by the tool are not forbidding, and many changes in the analysis algorithms are trivial to implement.

In brief, the analysis paradigm is the following:

(a) The program to be analyzed is parsed using a parser automatically generated from a grammar. During the parse, syntax-directed translation techniques are used to output a collection of Prolog facts, expressing the information needed for later static analysis.
(b) Also during the parse and using syntax-directed translation, self-instrumentation is generated for the program being analyzed. The instrumentation takes the form of statements that when executed at run-time, generate Prolog facts about the behavior of the program.
(c) The program is executed on test data, and the combined facts of (a) and (b) form a Prolog database describing program and test.
(d) Prolog fragments placed in a library describe the analysis to be performed on the collected facts (c).
(e) The user interface to the analysis system is the Prolog query mechanism, in which the library (d) is an aid to investigation of the database of (c).

The research tools described here are at the opposite pole from those currently under development within language compilers. The contrasting technique is called "compiler-integrated testing (CIT)[10]"; in it, a compiler is modified to create the testing tool. For research prototypes, success of this method depends on a compiler that already contains most of the routines needed for analysis. The author developed such a tool for mutation testing in 1975[11], based on the sound software-engineering in the SIMPL-T compiler[12]. The DAISTS testing tool for data abstractions was also constructed in this way[13]. These tools were built into SIMPL-T compiler variants because we believed that was the quickest, easiest way to obtain a research prototype. SIMPL-T attempted very little flow analysis, so we did not seriously consider incorporating data-flow. With the release of the GNU C compiler[14], which does include flow analysis, CIT prototype tools for data flow can be constructed[15]. CIT tools can be efficient, and they are easier to make at "industrial strength," if the compiler is itself robust. And, as in the SIMPL-T examples of the 1970s, CIT may be the best software engineering solution to the problem of prototyping a tool. However, a CIT tool is even harder to modify than stand-alone tools. Even a small change to the analysis method means redesign and implementation, and depends critically on the quality of the original compiler modification and its documentation. The CIT tool is also at the mercy of the original compiler writers; should they decide to make changes or fix bugs, the CIT tool must follow along. Thus although CIT is the better choice for a tool that is to be used in practice, our method is superior for research into tools themselves.

PROTOTYPING METHOD

We first experimented with the prototyping method using a subset of the C language, standard UNIX compiler-compiler tools, and Prolog. In this section we explain the method by describing the construction of a prototype analyzer for one kind of dataflow test coverage.

Dataflow testing—DU pairs and paths

Structural testing criteria in which control-flow points of a program must be "covered" by tests are as old as programming. The most common of these are *statement coverage,* and *branch coverage,* in which tests are required to force execution of all program statements, and all branches, respectively. Recent interest has centered on so-called "dataflow" criteria, which require the testing of program paths defined by the usage of program variables[16]. For example, a *DU path* ("DU" abbreviates "Definition-Use") for a variable X is a path that starts where X is given a value, and ends where X is used, without X being set again along the path. In the "all-uses" dataflow coverage criterion, test data must force execution of some DU path (if there is one or more) between each definition-use pair, for all program variables. Technically, we consider a "pessimistic" all-uses, ignoring the infeasible-path problem. That is, we identify potential DU pairs and paths solely from the static connectivity of the program statements. It may happen that the meaning of the actual conditional statements in the program makes it impossible to execute some of these potential DU paths. (The alternative of defining away infeasible paths[17] is not available to the tool builder, since identifying them in general is an unsolvable problem.)

The all-uses criterion will be used to illustrate the construction of a prototype testing tool.

Prolog facts and queries

Logic programming, particularly in the widely-available language Prolog[18], is ideal for writing software-analysis programs. Because Prolog may be relatively unfamiliar to those who usually work with imperative languages, here and in a following section we describe the features we use. This treatment is neither complete nor precise, and should be skipped by those familiar with Prolog.

Prolog is a declarative language, and its simplest construction is the “fact.” A fact has the appearance of a conventional procedure call with constant actual parameters (integers, certain strings, and lists of these are available types), followed by a period. For example,

```
sample_fact(1,aaa,[1,2]).
```

is a fact. Intuitively, it is thought of as naming a relationship (`sample_fact`) that holds among the parameters (integer 1, string “aaa” and a list with integer elements 1 and 2 in this case). The fact name is called a *predicate*.

A collection of facts is sometimes called a Prolog “database,” expressing the conjunction of these facts. Prolog *queries* can be used to interactively inquire about a database. The interactive prompt is “`?-`”, usually written in a narrative to identify queries. The form is similar to that of a fact, but “variable” parameter values are used to obtain output (they begin with a capital letter). For example:

```
?- sample_fact(X,aaa,[1,2]).
```

is a query, and given the fact above, it will produce the result:

```
X = 1
```

If there are other `sample_fact`s, this query might also yield other values of `X`. The interactive user requests these by typing a semicolon.

All of the arguments may be specified, e.g.:

```
?- sample_fact(1,aaa,[1,2]).
```

with the result `yes`. Or:

```
?- sample_fact(2,aaa,[1,2]).
```

with the result `no` (if the database contained just the single fact above). Or, more arguments may be variables, so that

```
?- sample_fact(X,Y,Z).
```

will give

```
X = 1
Y = aaa
Z = [1,2]
```

as one result.

Analyzing a sample program

It is easiest to describe the prototype all-uses analyzer using a simple program shown in Figure 1. The program is intended to solve quadratic equations given their coefficients, printing either the roots, or a message identifying them as real or complex, depending on an input flag. Three coefficients and a flag value are read repeatedly until an end of file is encountered. Line numbers have been added to the program for reference convenience. Hereafter this program will be called “the quadratic program.”

```
main()
{
  /*Solve quadratic a*x^2 + b*x + c = 0 for x.
    If qual is nonzero, identify the roots
    as real or not; otherwise, print the roots */
float a,b,c,r1,r2,im,j,d,t;
int qual;
while (scanf("%f %f %f %d", &a,&b,&c,&qual)>0) {
  d = b*b - 4*a*c;
  t = 0;
  im = 0;
  if(d<0)
    im = sqrt(-d);
  else
    t = sqrt(d);
  r1 = (-b+t)/2*a;
  r2 = (-b-t)/2*a;
  j = im/2*a;
  if(qual)
    if(j==0)
      printf("Real root(s).\n");
    else
      printf("Complex roots.\n");
  else
    printf("%f+i%f and %f-i%f\n", r1,j,r2,j);
  }
}
```

Figure 1. Sample program 'quadratic'

Static facts

When the parser identifies grammatical rules deriving a program, it can perform actions to record static information about the program's construction. For example, whenever a statement is recognized, it can be numbered and the possible flow of control recorded. The Prolog predicate `edge(A,B)` is defined to hold when it is possible for control to pass from statement `A` to statement `B`. Similarly, when variable `V` acquires a value in statement `D`, or is used in statement `U`, the predicates `def(V,D)` and `use(V,U)` hold respectively*. In the

* For technical reasons, Rapps and Weyuker[16] distinguish a "c-use" (computation use) from a "p-use" (predicate use). The latter occurs in conditional expressions that influence control flow. A p-use is defined as extending to both alternatives of the conditional, technically by connecting the p-use with the edges leading from the conditional. Thus in the code fragment

```
31  if (X>0)
32    printf("positive")
33  else
34    printf("not")
```

the p-use of `X` is taken to be on the edges 31-32 and 31-34, not at statement 31 as intuition indicates. We have chosen not to use this somewhat peculiar definition of p-use, because it seems counterintuitive in a number of ways, and the technical advantages quadratic program, the facts generated when lines 18 and 19 are parsed are:

```
def(j,18).
use(im,18).
use(a,18).
edge(18,19).
use(qual,19).
```

The actions that generate these facts are easy to place in the parser; for example, a `def` fact is created when an assignment is recognized.

This static information is adequate to calculate the possible DU paths in the program. For example, 9-10-11-12-15 is one such path for `d`.

Instrumented program segment

The parser echoes the input program, but with instrumentation modifications. Most instrumentation can be placed between the original statements, and the actions to place it occur when a statement is parsed. A few constructions require a more elaborate treatment. In the following instrumented portion of the sample program, the lighter type indicates added instrumentation:

```
20  if ( (!fprintf(fileptr,"step(%d,%d,20).\n",
        xseq++,xstmt)) && (xstmt=20) && (j==0 ))  {
21    printf("Real root(s).\n");
      fprintf(fileptr,"step(%d,%d,21).\n",xseq++,xstmt);
      xstmt=21;
      }
```

The variables introduced in the instrumentation are assumed not to occur in the input program; their added declarations and initializations are not shown. The instrumentation code shown was designed by a novice; a better design is described below. When the instrumentation code is executed, it keeps track of control flow by printing Prolog facts to a file. The complication in the `if` statement is necessitated by the short-circuit evaluation many C compilers perform; if the instrumentation were not embedded in this way it might be missed.

Run-time facts

The facts generated when the instrumented program runs depend on the input data. For example, the input data at the left below produces results as shown at the right:

```
1 2 1 0         -1.000000+i0.000000 and -1.000000-i0.000000
```

(That is, the equation $x^2 + 2x + 1{=}0$ has two real roots $x{=}-1$.) The instrumentation generates the following facts from the above input data:

```
step(0,0,8).
step(1,8,9).
step(2,9,10).
step(3,10,11).
```

are seldom of practical importance. This choice is further discussed below in the section on dependency-chain coverage.

```
step(4,11,12).
step(5,12,15).
step(6,15,16).
step(7,16,17).
step(8,17,18).
step(9,18,19).
step(10,19,25).
step(11,25,8).
step(12,8,26).
step(13,26,27).
```

(The first parameter of the `step` predicate is an execution sequence counter, which will be explained in the section on analysis to follow.)

It is easy to see from these dynamic facts that the DU path 9-10-11-12-15 for `d` has been executed. To decide if enough DU paths have been executed to satisfy the all-uses criterion, however, requires more bookkeeping than is easy to do by hand. What's needed is a way to search the database for patterns, which Prolog provides.

Prolog rules

In addition to facts, Prolog allows the definition of general relationships among its predicates. However, only one restricted form of logical assertion is allowed: "IF there are values to make some predicate(s) hold, THEN another predicate holds." The syntax is abbreviated and written backwards; for example:

```
pred1(X) :- pred2(X,3), pred3(aaa,X).
```

This has the intuitive meaning: "For each `X`, IF `pred2` holds of `X` and 3, AND `pred3` holds of 'aaa' and `X`, THEN `pred1` holds of `X`." Facts are the special case of a rule in which the right side is missing, e.g.,

```
pred(5).
```

could be read, "(IF no conditions, i.e., TRUE, THEN) `pred` holds of 5."

Analysis of test coverage

Prolog rules can be used to automate the testing analysis. For example, the definition:

```
pos_path(X,Y,[X,Y]) :- edge(X,Y).
pos_path(Beg,End,[Beg|T]) :- pos_path(Mid,End,T),
                              edge(Beg,Mid).
```

defines the existence of a path in a program from statement `Beg` to statement `End`. (The list `[Beg|T]` has head `Beg` and tail the list `T`.) These rules define `pos_path` to hold in two cases: each `edge` establishes a path between its two statements; and, any sequence of connected edges is a path. This form of inductive definition is very common in Prolog, and is used to calculate the path list (third argument) given the end points (first two arguments).

For example, consider the program fragment:

```
45  printf("Entering loop");
46  for (Count=1; Count<=10; Count++) {
47    if (Count==1)
48      X = 0;
49    else
50      X = CalcIt(X);
51    }
```

Analysis of this fragment would yield (among others):

```
?- pos_path(45,50,P).
P = [45,46,47,50]
```

A similar predicate cannot be used with the `step` facts, because it would find false paths. If the fragment above is executed, a similar predicate with `step` in place of `edge` would also list 45-46-47-50. However, 45-46-47-50 has not been executed; it results from connecting the execution of statement 50 following 47 the *second* time through the loop with the execution of statement 47 following 46 following 45 the *first* time through the loop. (In fact, the path 45-46-47-50 is *infeasible* and can never be executed.) The sequence parameter of `step` was added to address just this difficulty of false paths. A predicate that uses it to define paths actually executed is:

```
x_path(S,X,Y,[X,Y]) :- step(S,X,Y).
x_path(Seq,Beg,End,[Beg|T]) :- step(Seq,Beg,Mid),
                               NextSeq is Seq+1,
                               x_path(NextSeq,Mid,End,T).
```

(The built-in `is` operator is the Prolog way of expressing a numerical relationship between variables.) The predicate expresses the information that an executed path is one in which the sequence numbers are consecutive.

The `x_path` definition, along with the static and dynamic facts generated by the parser and the instrumented program respectively, constitutes a very simple prototype test tool, one that can help a programmer with the bookkeeping of seeing which paths have been executed by a set of test data. Along with `pos_path`, the programmer can ask about path coverage, for example, in the quadratic program with the data given above:

```
?- pos_path(12,25,P), not(x_path(_,12,25,P)).
```

is a query that asks if any path starting at the test on `d` and ending with printing the roots has not been executed, and the response

```
P = [12,13,16,17,18,19,25]
```

identifies one such path, for the case of complex roots. (Prolog includes a built-in "unsafe negation;" it is unsafe because it does not always agree with true logical negation. In rules like the above, `not` is safe. Using "_" for the sequence parameter in `x_path` prevents Prolog from printing that value which is uninteresting for this query.)

Even in this trivial case, the advantages of the Prolog component can be seen: (1) no data structures need be implemented to store information from either parse- or run-time, and (2) the analysis code is straightforward and extremely simple.

A dataflow analyzer for DU paths is not much more difficult to construct. A DU path begins with a def and ends with a use, no other defs intervening. The Prolog below first defines a path containing no defs (except perhaps at the very end), then a potential DU path as one of these following a def and ending with a use:

```
def_clear(Var,X,X,[X]).
def_clear(Var,Beg,End,[Beg|T]) :- Beg=/=End, edge(Beg,Mid),
                                  not(def(Var,Beg)),
                                  def_clear(Var,Mid,End,T).
pos_DU_path(V,Def,Use,[Def|T]) :- def(V,Def), use(V,Use),
                                  Def=/=Use, edge(Def,Sec),
                                  def_clear(V,Sec,Use,T).
```

Finally, let

```
not_x_DU(Var,T) :- pos_DU_path(Var,Beg,End,T),
                   not x_path(_,Beg,End,T).
```

The `not_x_DU` predicate finds all potential DU paths that have not been executed. For very simple programs (without multiple paths for a single DU pair), it also constitutes an all-uses analysis. However, all-uses requires covering only *some* path for a DU pair, not all paths. To more correctly report a coverage failure, only the def and use locations should be given, since the analyzer cannot know which (if any) of the potential infinity of paths between these locations should be singled out for attention by the tester.

All-uses analysis can be accomplished by finding each def-use pair, deciding if there is at least one potential def-clear path between them (but not by finding all such paths), and then finding if any def-clear path was executed between them (not necessarily the potential path that was identified). The Prolog for this is:

```
all_use_missed(V,S,E) :- def(V,S), use(V,E),
                         pos_DU_path(V,S,E,_),
                         not x_dcpath(_,V,S,E,_,_).
```

where

```
x_dcpath(S,V,X,Y,[X,Y],NS) :- step(S,X,Y), not(def(V,Y)),
                              NS is S+1.
x_dcpath(S,V,B,E,[B|T],LS) :- step(S,B,M), not(def(V,M)),
                              NS is S+1,
                              x_dcpath(NS,V,M,E,T,LS).
```

For the quadratic program with the single input (as above)

```
1 2 1 0
```

the all-uses analysis produces:

```
?- all_use_missed(Var,Beg,End).
Var = d
Beg = 9
```

```
End = 13 ;
Var = t
Beg = 10
End = 16 ;
Var = t
Beg = 10
End = 17 ;
Var = im
Beg = 13
End = 18 ;
Var = j
Beg = 18
End = 20
```

These uncovered DU pairs show that the single test case does not involve a complex root, nor does it try both possibilities for `qual`. If the tester needs more information, it can be obtained from some of the Prolog already written. For example, to investigate what is required to cover the def-10,use-16 pair for `t`:

```
?- pos_DU_path(t,10,16,Path).
Path = [10,11,12,13,16]
```

and retry shows this to be the only possibility. The tester then knows that an additional test must make `d` negative (to force execution of statement 13). The test set

```
1 2 1 0        -1.000000+i0.000000 and -1.000000-i0.000000
1 0 1 1        Complex roots.
```

(which includes the earlier test and shows output to the right), should suffice, and indeed when the analysis is repeated with this data:

```
?- all_use_missed(V,B,E).
no
```

The test data has attained all-uses coverage.

In this section we have displayed the entire implementation of an all-uses test analyzer, which amounts to less than one page of Prolog code. Figure 2 shows the organization of the prototype tool.

Assessing a prototype tool

The prototype all-uses analysis tool described in the previous section was constructed as a proof-of-concept. Most of it was written by two undergraduates[19] working full-time for about two months, including the time to learn about parser generation and logic programming (a C-subset grammar was available).

The parser actions that generate static facts and create the instrumented program amounted to about 1000 C statements. (About 200 were devoted to reconstructing the input program from the parsed version.) This code is very straightforward, since it is located at points identified by

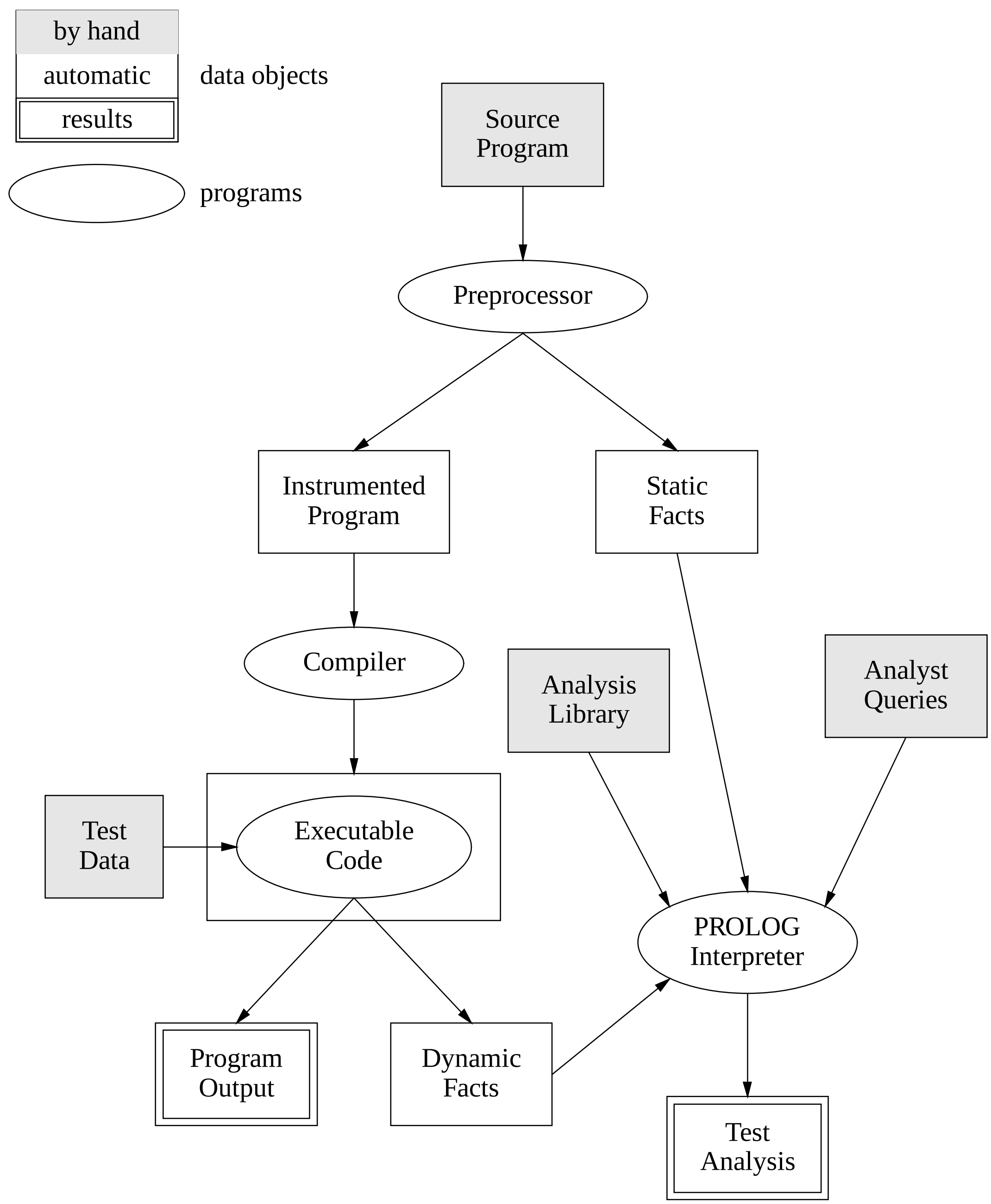

Figure 2. Block diagram of a prototype tool

the grammar. Thus for example, when an assignment statement is recognized, the actions are to reproduce the statement using the tokens and strings on the yacc stack, to generate static `edge` and `def` facts, and to produce instrumentation output that later generates a `step` fact. Similarly, recognizing an expression triggers generating one or more `use` facts.

DU-path analysis required only a handful of Prolog predicates to support queries listing uncovered DU pairs, and to further aid the tester in attaining coverage. Even the most complicated of these predicates has a simple inductive form using only a few terms.

The programming tasks separate nicely. Since the interface from the parser to Prolog is readable text, and the instrumented program and its output are also readable, samples of these can be created by hand to test the Prolog analysis routines. Thus the yacc and Prolog programming can be carried out in parallel.

These statistics show that judged against prototyping standards, the method is a success. Similar conventional implementation efforts in Pascal or C have required several times as long using experienced programmers, and produced an order of magnitude more code. The simplicity of our analysis tools is even more advantageous when modifications are considered, as discussed below.

The prototype tools are very fragile, however. The Prolog query interface requires its user to use a precise syntax correctly, and has only the "Nancy Reagan error message": when anything is wrong it just says, "`no`". For example, if the user forgets an argument to `all_use_missed` or misspells the predicate name, the `no` error response will appear to mean that all the DU pairs have been covered. In addition, even correct queries may not be useful. The path predicates like `pos_DU_path` construct their results using linear search of the static and dynamic facts, backtracking to recover from poor choices. Since the number of paths may grow exponentially in the program size, these computations may use far too much time and storage, even when analyzing only modest programs. We later argue that even the most fragile of prototypes is useful, and that the trade-offs that give away efficiency for prototyping ease are the right ones in a research or education setting.

A more robust implementation

We successfully used the prototype described above, and a number of variants constructed in the same way[20]. Its quick and dirty implementation does not seriously interfere with application by its designers. However, its implementation, which was begun as a learning experience for students, has deficiencies beyond those inherent in the approach, which make the system difficult for an outsider to use:

1) Only a subset of C was considered, and that subset grew in a haphazard way in response to the needs of experimentation. Analysis of even toy programs might encounter a hole in the subset, requiring a work-around or changes to the preprocessor.
2) The action code in the parser was not modular; actions taken for entirely different purposes were all scrambled together in one place. When changes were required, it was necessary to understand all of the code before anything could be successfully modified.
3) The system was awkward to run, requiring its user to remember and correctly name file collections to successive processors, and to repeat the entire run when only a change in test data was made.

We therefore designed and constructed an improved tool-generation system to address these deficiencies. The subset of C analyzed is large, and every attempt has been made to allow for modification by users, without requiring much understanding of the existing code.

The improved system was used to replicate the DU-path analyzer of Section 2. This analyzer handles full ANSI C, except:

a) It is blind to array and pointer usage. That is, it treats these entities as if they were simple variables, which is quite wrong for dataflow analysis. Other much more ambitious systems are also deficient in treating aggregates (e.g., ATAC[4]). Aggregates are an open research topic[3], which we are investigating using our prototyping technique[20].
b) Unions are not supported.
c) Although routines may be separately compiled, and library or other external routines may be included with a program to be analyzed, only the portion comprising a main program is analyzed.
d) Some cases of the pre/postfix operators like "++" are not correctly instrumented.

In allowing more of the C language we do not imply that the tools created can scale up to handle "real" software. The limitations on required space and time are still present. However, if an experimenter devises a clever (but limited) example program, we want the generated tools to handle the example without modifications to work around missing features of C.

Compiler-generator tools like yacc are awkward to use for preprocessing as we are doing. There is no explicit provision for echoing the source program, and no way to cleanly separate different aspects of the processing so that they can be incrementally incorporated. The newyacc parser generator[21] is designed to address these deficiencies. It has the ability to repeatedly process a parse tree, performing distinct actions on each traversal, and to echo the source at any level.

The prototype-construction system was made available in June, 1994, under a National Science Foundation software-capitalization grant. The distribution is directed at researchers who would make their own enhancements following our example. For information, contact the author by e-mail.

In the sequel, we use the more robust implementation. However, two details are suppressed:

(1) Some C constructions allow a single statement to perform multiple actions (often through use of the assignment operator "=" in expressions). Thus line numbers are not adequate to represent control points. A "logical statement number" was defined, and used in all facts, along with new facts giving the correspondence between logical and actual line numbers. Since this complication does not occur in examples here, we continue to pretend that the line numbers themselves are used.
(2) Even the limited block structure of C allows syntactically similar variable values to actually refer to distinct values. "Logical variable names" were created and used in all facts, along with a new fact giving the line number and syntactic form corresponding to these created names. We continue to pretend that the variable names in the program are actually used, and avoid confusion by choosing simple examples.

The actual complications introduced in the Prolog code by using program quantities in queries, but computing in "logical" terms, is very small.

CHANGING A PROTOTYPE TESTING TOOL

Although it is advantageous to be able to construct prototypes rapidly, it is even more important to be able to make rapid changes, particularly for testing tools. Because the extensive bookkeeping of execution monitoring is nearly impossible to do by hand, it is not until a prototype is available that a researcher discovers the most obvious flaws in a new technique. Then changes are required, and larger changes when basic concepts have to be altered. A second issue is portability of the tool. Its evaluation may require a different computer platform than that used for development; however, a far more difficult "portability" problem arises if the analyzed source language changes.

Our prototyping technique addresses issues of change as follows:

Platform. All components of the system are written in C. The programming tools themselves are usually available without porting.

Source language analyzed. When it makes sense to analyze two different source languages, it is because they are not too dissimilar in structure. They may then have similar grammars. Most existing test technology is wedded to conventional languages with expressions, assignments, explicit control flow, etc. A pair of these languages, say Pascal and C, are different in many details, but similar enough that the majority of the parser action code may be straightforwardly converted from one to the other. It is estimated that about 20% of the actions creating static facts and instrumentation for the DU-path analysis of Section 2 would have to change if the analyzed language were changed to Pascal, for example. No changes would be required in the Prolog analysis code.

Analysis method. A change in the testing technique being analyzed can involve two kinds of change in the prototype. First, the Prolog analysis code will change. The second kind of change can be more difficult: the parser actions may need to change if additional static or dynamic facts are needed to support a changed analysis. We can minimize parser changes by creating an adequate basis of facts. For example, the complete control structure of a program does not require more information than indicated above. The newyacc preprocessor generator makes it easy to change static facts in a modular way; dynamic facts are more difficult to separate within the preprocessor. The next sections describe particular changes in analysis methods, and report on the difficulty of making them.

Dependency-chain coverage

Recent descriptions[22,23] have been given for a kind of dataflow coverage that has been in informal use without tool support since the 1960s. The technique extends all-uses coverage to handle the situation in which DU paths connect to convey information, perhaps through more than one variable. Imagine a DU path for variable X in which the final use occurs in a statement assigning a value to Y, connected to a Y-DU path where the final Y-use is a Z-def, and so on. The composite path composed of DU sections joined by a shared variable, which Campbell[22] calls a *dependency chain,* should be tested just as DU pairs should be tested, since the dependency chain represents a programmer's use (at the end of the chain) of information established at the beginning and passed along. An important special case occurs when information is passed from a variable to itself. An assignment like

```
X += 1;
```

always interrupts a DU path for X and begins another; viewed as part of a dependency chain, the old value use of X is passed to the new value def of X.

Campbell considered only dependencies that occur in assignments, where the transfer of information is obvious. We first thought to extend to dependency chains the somewhat peculiar definitions of Rapps and Weyuker[16] involving “p-uses” in conditional expressions. However, except in the simplest cases, such a definition is difficult to frame, and of dubious intuitive value. We therefore decided to continue no chains from a p-use to subsequent defs. In most cases, the point is moot, because the conditional variable(s) are also used in explicit assignments, thus establishing chains beyond a conditional. Our decision is sometimes intuitively incorrect; we view these as a tradeoff for the cases in which the Rapps and Weyuker definitions create infeasible DU paths (and similar dependency chains) that intuitively should not be considered†. The choice between the definition of Rapps and Weyuker and the one we chose is not a clear one. Our prototyping scheme could be used to investigate the difference. Only small changes in the Prolog analysis predicates are required. A high school student familiar with the Prolog code was assigned to start over with the definition of Rapps and Weyuker, keeping careful track of the time required to redo the “all-uses” analysis predicates. He reported 25 minutes for coding and debugging.

For dependency chains, a coverage criterion similar to all-uses for DU paths can be defined. We will call it *all-chains*. To satisfy the all-chains criterion, test data must force execution of some chain for each pair consisting of a def (of one variable) and a subsequent use (of a perhaps different variable), where some chain does connect the pair.

In the hierarchy of path-testing methods that includes DU paths[16], dependency-chain coverage lies between all-paths and all-uses (it is incomparable to “all-DU-paths”). (We ignore changes in the hierarchy below all-uses created by the choice of how to treat p-uses.) All-chains is related to data-context ideas[2] and to program slices[24], but is narrower, closer to all-uses coverage than either of these ideas.

Altering an all-uses coverage tool to one that measures dependency-chain coverage is not difficult. No new static or dynamic facts are needed, and the analysis predicates for DU paths can be used to define a predicate for dependency chains. The tester can almost construct the new tool while analyzing a particular program. Although the following scenario is too pat to be real, it could happen that dependency-chain analysis was reinvented in response to deficiences of all-uses analysis for some particular case, the new analysis predicate written, and the analysis performed on the spot, using the same instrumented run that suggested the idea.

A dependency chain is an appropriately joined sequence of DU paths, expressed in Prolog as:

```
pos_dep(V1,V1,S1,S2,Ch) :- pos_DU_path(V1,S1,S2,L),
                            append([V1|L],[V1],Ch).
pos_dep(V1,V2,S1,S2,Ch) :- pos_DU_path(V1,S1,SM,P),
                            def(VM,SM),
```

† The Rapps and Weyuker definitions can create intuitively incorrect DU paths when a condition is repeated. For example, in

```
37  if(X<0)
38    neg = X*X;
39  printf("Ready.");
...
52  if(X<0)
53    printf("modulus: %f",sqrt(neg));
```

the infeasible path 37-39-...-52-53 is not intuitively a DU path for `X`, since the programmer has designed it to be impossible; Rapps and Weyuker would classify it as a DU path. Our definition, on the other hand, does not correctly handle

```
if(X==0) found = 1;
else found = 0;
```

because the flag `found` is dependent on the conditional variable `X`, and a chain should continue from `X` through `found` to subsequent uses of `found`, but does not.

```
                                pos_dep(VM,V2,SM,S2,[VM|[SM|Y]]),
                                append([V1|P],[VM|Y],Ch).
```

This predicate (pos_dep) displays dependency chains with embedded variables. For example, in the quadratic program:

```
?- pos_dep(b,j,8,25,Chain).
Chain = [b,8,9,d,10,11,12,13,im,16,17,18,j,19,25,j]
```

shows a chain running through the whole program.

The pos_dep predicate is more complicated than any of those presented so far. It works by pasting together DU paths in the final argument. (append places its second-parameter list at the end of its first-parameter list to create the third-parameter value.)

The analysis predicate for all-chains coverage is constructed in the same way as the similar all_use_missed:

```
all_chain_missed(V1,V2,S,E) :- def(V1,S), use(V2,E),
                                 not x_dep(_,V1,V2,S,E,_,_),
                                 pos_dep(V1,V2,S,E,_).
x_dep(Se1,V1,V1,S,E,C,SeL) :- x_DU_path(Se1,V1,S,E,L,SeL),
                                append([V1|L],[V1],C).
x_dep(Se1,V1,V2,S,E,C,SeL) :- x_DU_path(Se1,V1,S,SM,P,SeM),
                                def(VM,SM),
                                x_dep(SeM,VM,V2,SM,E,
                                   [VM|[SM|Y]],SeL),
                                append([V1|P],[VM|Y],C).
x_DU_path(S,V,B,E,P,LS) :- def(V,B), use(V,E),
                             x_dcpath(S,V,B,E,P,LS).
```

It is in general more difficult to attain all-chains coverage than all-uses coverage. For the quadratic program of Section 2.3, the test data of Section 2.8 that attains all-uses coverage does not attain all-chains coverage because seven chain pairs are missed, the first being:

```
?- all_chain_missed(V1,V2,Start,Stop).
V1 = b
V2 = j
Start = 8
Stop = 25
```

All seven chains are best displayed using another built-in Prolog predicate setof to collect the results:

```
?- setof([V1,B,E,V2],all_chain_missed(V1,V2,B,E),S).
S = [[b,8,25,j],[c,8,25,j],[d,9,25,j],[im,11,20,j],
       [im,13,25,j],[t,10,25,r1],[t,10,25,r2]]
```

In the course of investigating these deficiencies in all-chains coverage, the predicate pos_dep is useful, as in Section 2.8 the similar pos_path was useful. The path we used for illustration:

```
?- pos_dep(b,j,8,25,Chain).
Chain = [b,8,9,d,10,11,12,13,im,16,17,18,j,19,25,j]
```

cannot have been executed, since doing so would have eliminated the first missed chain. The tester might notice that there is a similar chain beginning with `a`:

```
?- pos_dep(a,j,8,25,Chain).
Chain = [a,8,9,d,10,11,12,13,im,16,17,18,j,19,25,j]
```

Because the same path is involved (8-9-10-11-12-13-16-17-18-19-25), this `a`-chain has also not been covered. Why didn't it show up in the response to the `all_chain_missed` query? The user of production software asks questions like this to understand a program and its testing; the developer of a research prototype asks them because an anomaly suggests that there may be a problem with the tool under development. There *is* a problem, but in the concept of "all chains" rather than its implementation. There are two intuitively appealing ways to define all-chains coverage by analogy to all-uses coverage, and they are the same only in simple cases. Here the research prototype tool is doing its job of informing its developer when an idea has not been thought through. The possible definitions for all-chains are:

1) A def at M and a use at N (where some path connects M to N) is covered if any chain between M and N is executed. (This is the "obvious" definition we implemented.)
2) Two chains between M and N are distinct if they involve a different sequence of intermediate variables (including those at M and N). A def-use pair is covered iff all distinct chains between them are covered.

The definition 2) may be less obvious, but it has a claim to be intuitively correct, because different variables are considered to constitute distinct DU pairs, even for the same path.

In the case of the `a`-chain anomaly that was encountered while trying the prototype, we can confirm that definitions 1) and 2) differ by asking for more information on `a`-chains between 8 and 25:

```
?- pos_dep(a,_,8,25,Chain).
Chain = [a,8,9,d,10,11,12,13,im,16,17,18,j,19,25,j] ;
Chain = [a,8,9,10,11,12,13,16,17,18,j,19,25,j] ;
Chain = [a,8,9,10,11,12,13,16,17,r2,18,19,25,r2] ;
Chain = [a,8,9,10,11,12,13,16,r1,17,18,19,25,r1] ;
Chain = [a,8,9,10,11,12,15,16,17,18,j,19,25,j] ;
Chain = [a,8,9,10,11,12,15,16,17,r2,18,19,25,r2] ;
Chain = [a,8,9,10,11,12,15,16,r1,17,18,19,25,r1] ;
Chain = [a,8,9,d,10,11,12,15,t,16,17,r2,18,19,25,r2] ;
Chain = [a,8,9,d,10,11,12,15,t,16,r1,17,18,19,25,r1]
```

These chains are all distinct, but only two paths are involved: 8-9-10-11-12-13-16-17-18-19-25 and 8-9-10-11-12-15-16-17-18-19-25. In the test cases, one path was taken and the other was not:

```
?- x_path(_,8,25,P).
P = [8,9,10,11,12,15,16,17,18,19,25]
```

(the only possibility). Hence according to definition 2), the chains passing through line 13 (the first four listed above), with the variable sequences `a-im-j`, `a-j`, `a-r2`, and `a-r1` have not been covered.

The developer needs to decide which definition is appropriate. To change the implementation to capture 2) instead of 1) is very easy, so experiments can be conducted if necessary. We decided to stick with definition 1).

Returning to the seven uncovered chain pairs that were listed above for definition 1), it can be seen that to cover these chains requires at least displaying a solution to an equation with roots that are not real, and printing the "Real root(s)" message. The last two inputs in the cumulative data set:

```
1 2 1 0         -1.000000+i0.000000 and -1.000000-i0.000000
1 0 1 1         Complex roots.
1 0 2 0         0.000000+i1.414214 and 0.000000-i1.414214
1 -1 -6 1       Real root(s).
```

do these two things, and with this data, all-chains coverage is achieved:

```
?- all_chain_missed(V1,V2,Start,Stop).
no
```

(In this case there is no difference between definition 1) and definition 2); hence the situation that will expose a difference is very fragile, and might not be noticed at all in "production" use of a tool.)

In these examples, we have used test tools as naive testers in practical situations often use them. First, we made haphazard choices of data, then when coverage was inadequate, we followed the tool's analysis to find new test points that would satisfy it. This usage pattern illustrates the abilities of our prototypes, but it is probably not a wise use of a coverage-analysis tool. An apt analogy would be a novice programmer responding to compiler syntax-error messages by making haphazard changes in the program, trying to silence the compiler. The result would be a program that compiles without warnings, but is unlikely to execute as required. A better response to syntax messages is to examine the specification and alter the program to do what is intended, with attention to the part flagged by the compiler. The same can be said for practical testing. A coverage failure should prompt study of the specification, to find a test case that will improve the coverage, but only by testing some neglected aspect of the program's functionality. Coverage is no more related to the real adequacy of tests than a syntax error is to a program's meaning. Brian Marick[25] has convincingly described the wise use of test analyzers.

There are many variants of dataflow coverage, and any new one can be tried with the same ease as dependency chains. Lest the reader think that adapting an existing tool is easy no matter how it is constructed, we suggest problems that might arise in a conventional implementation. Data structures, introduced in conventional tool implementations by necessity, would cause most of the trouble. A DU-path tool needs to calculate and store DU paths, but it probably will not have provided for storing sequences of such paths as dependency chains require. (And if the alternate definition 2) above is used, these paths will also have to be compared.) Even if the main data structure has been implemented as an abstract data type, making change as easy as possible, the effort required to modify it would not be trivial. Another level of iteration must also be added to the analysis code to calculate the path chains. Successfully making such changes in a medium-sized program is a process requiring great care and a measure of good luck; in any case it is not done on the fly when the tester has a new idea. Yet that is precisely what our prototyping scheme allows.

Data-coverage testing

Most systematic, program-based testing techniques are based on control-flow coverage. It is apparent to anyone using these methods that their flaw lies in a lack of "data coverage." Thus in (say) all-uses testing, a testset does cause all DU pairs to be exercised, and therefore satisfies the requirements of the method; but, when the necessary paths are taken, program variables do not assume crucial values, and so some bug slips by the test. The trouble with control-flow methods is that too often they explore the control possibilities with trivial data, and thus the "coverage" is spurious. It can even happen that a more demanding control criterion (e.g., all-uses coverage as opposed to branch coverage) is *less* effective in exposing bugs because a person preoccupied with forcing execution of a difficult path is led to choose trivial data on that path.

Mutation testing was originally an attempt to fill the data-coverage gap in statement testing, by forcing "expression coverage" at each control point[11]. Intuitively, it is harder to hide defects from mutation than from control-flow techniques. If in fact this is true, it supports data coverage as a measure of a test's quality.

A plausible theory of test quality can be based on data-coverage[26]. At each control point in a program, there is a range of possible internal states, i.e., the sets of values all internal variables can assume at that point during any possible execution. Any program defect must manifest itself in an erroneous program state of this kind. That is, if a program contains a defect, then on some possible execution, at some control point, some variable value will be wrong. A testset is "good" according to this theory if its coverage of program states is high. The problem with such a theory lies in determining the possible state values within a program; the determination is usually difficult, and in general an unsolvable problem. Thus we cannot hope to mimic other structural testing tools that report *deficiencies* in coverage.

We can, however, measure state coverage for a given testset. A person may then decide that the reported coverage is inadequate, and try to improve it, ideally in the way suggested by Marick[25]. The differences between using state coverage and control-flow coverage techniques are: (1) There is no easily computed standard of 100% coverage to be attained; but in compensation, (2) State coverage is plausibly connected to the detection of defects.

To create a prototype data-coverage analyzer, dynamic information must be captured to allow the calculation of execution states. This requires adding program instrumentation in the parser of our tool-building system. The new dynamic facts generated require new analysis predicates, not closely related to the existing ones. Thus the change from all-uses analysis to datastate analysis is less trivial than the one to all-chains described above. In a conventional tool implementation, a good case could be made for starting over from scratch, perhaps reusing some components from the existing analyzer. In our prototype tool, the parser changes amounted to a few hours work (less than 5% of the code was changed), and the Prolog predicates displayed below took about a day to write and test.

The new facts needed are the values of program variables at each control point during execution. This information is captured in the Prolog fact

```
value(Seq,Var,Value,Line)
```

that holds when `Var` takes on `Value` at `Line`. The counter `Seq` is used to tie each value into a particular execution chain defined by `step` facts, avoiding erroneous use of a value from another path, as described previously. To collect the `value` facts at runtime, the parser inserts extra instrumentation very similar to that generating the `step` facts, but only for those statements that give values to variables.

In this section we use the example of the quadratic program with the test data that covers all-chains:

```
1 2 1 0        -1.000000+i0.000000 and -1.000000-i0.000000
1 0 1 1        Complex roots.
1 0 2 0        0.000000+i1.414214 and 0.000000-i1.414214
1 -1 -6 1      Real root(s).
```

Its first few statements:

```
 8  while (scanf("%f %f %f %d", &a,&b,&c,&qual)>0) {
 9    d = b*b - 4*a*c;
10    t = 0;
```

when executed with this data generate:

```
step(0,0,8).
value(0,a,1.0,8).
value(0,b,2.0,8).
value(0,c,1.0,8).
value(0,qual,0.0,8).
step(1,8,9).
value(1,d,0.0,9).
step(2,9,10).
value(2,t,0.0,10).
```

Direct queries of the dynamic database can now yield information about some program runtime states. For example,

```
?- value(_,d,Value,Line).
```

has one response:

```
Line = 9
Value = 0.0
```

Or, a direct query could show that some particular states do not occur:

```
?- value(_,d,N,9), N>50.
no
```

In the interest of minimizing the number of facts generated, `value` instrumentation was added only on statements that change a value, and only for the value changed. A technique similar to that used to calculate execution paths in Section 2.8 can be used to establish state values throughout the program. To calculate the state value at any control point, we search along an execution path in reverse until the assignments are found that establish that state. The Prolog rules are:

```
xvalue(T,Var,N,Stmt) :- value(T,Var,N,Stmt), step(T,B,Stmt).
xvalue(T,Var,N,Stmt) :- step(T,B,Stmt), T1 is T-1,
                        xvalue(T1,Var,N,B),
                        not def(Var,Stmt).
values(Var,Stmt,Vs) :- bagof(X,xvalue(T,Var,X,Stmt),Vs).
```

The `xvalue` predicate agrees with `value` at the point of assignment, but at other control points the execution trace provided by `step` is used to work back to the immediately preceding assignment. The `values` predicate accumulates the state set for a single variable at a control point as a list (using a built-in predicate `bagof`). For example, in the quadratic program, the query:

```
?- values(d,12,S_d_12).
```

gives the result:

```
S_d_12 = [0.0,-4.0,-8.0,25.0]
```

To measure state coverage without displaying every value requires summarizing the data in some way. For example, the following predicate calculates the range of a variable:

```
range(Var,Stmt,[]) :- values(Var,Stmt,[]).
range(Var,Stmt,[Lo,Hi]) :- values(Var,Stmt,T),
                           min(T,Lo), max(T,Hi).
```

(where `min` and `max` compute the largest and smallest elements in a list). Applied to the data above:

```
?- range(d,12,R).
R = [-8.0,25.0]
```

Similarly, we can calculate other statistical parameters for the set of state points, to measure test quality in terms of state coverage. The code for standard deviation is:

```
stdev([],undefined).
stdev([X],0).
stdev(L,Sig) :- mean(L,M), devsqs(L,M,L2), sum(L2,S,N),
                N>1, Sig is sqrt(S/(N-1)).
```

The predicates `mean`, `devsqs`, and `sum` compute the average of a list, a list of squares, and the sum and cardinality of a list, respectively.

A predicate that combines range and standard deviation calculations in a single list is:

```
datacov(Var,Stmt,[Num,R,Sg]) :- values(Var,Stmt,S),
                                length(S,Num),
                                range(Var,Stmt,R),
                                stdev(S,Sg).
```

where `length` finds the length of a list. For example,

```
?- datacov(d,12,State).
State = [4,[-8.0,25.0],14.8633]
```

That is, four d values reached statement 12, in the range [-8,25], with σ = 14.9.

The test data above achieves all-chains coverage, the most complex of the dataflow techniques described above. To illustrate the point that data coverage under such techniques may be trivial, consider the variable a being used to compute a real root (line 17). The analysis predicate reveals that the coverage there is poor:

```
?- datacov(a,17,R).
R = [4,[1.0,1.0],0.0]
```

That is, although four a values reach statement 17, they are all 1, so there is no variation (σ = 0). Similar queries might find other statements with trivial coverage. In fact, it is easy to write a predicate to find them all:

```
notested(Var,L) :- values(Var,L,[]).
notested(Var,L) :- values(Var,L,S), stdev(S,0.0).
```

Predicate notested finds statements where the datastate for a variable includes 0 or 1 value. It is not surprising that in the example, notested(a,L) lists every statement L. After some experimentation to find a query that is well related to a deficiency in datastate testing, the following emerged:

```
?- pos_dep(V,_,S,25,_), notested(V,S).
```

That is, we look for a variable whose value doesn't vary at the beginning of a dependency chain, the chain ending with the statement that prints the roots. The result identifies three such variables: a at line 8, t at line 10, and im at line 11. The reader can possibly think of other, sharper queries. Here the prototype tool is being used to define a new kind of testing, which might be called "initialized-variation coverage."

If the tester attempts to gain better initialized-variation coverage for a, for example by adding the data point (with expected output):

```
2 4 2 0        -1.000000+i0.000000 and -1.000000-i0.000000
```

indeed initialized-variation for a is attained, but the result is a test failure:

```
2 4 2 0        -4.000000+i0.000000 and -4.000000-i0.000000
```

Conventional debugging techniques quickly show that the fault is in lines 16 and 17, where the programmer has mistaken the precedence for the "/" and "*" operators, and multiplied by a instead of dividing by it. The example is contrived, but it illustrates the danger of using trivial values (here, quadratic coefficients of only 1) in testing driven by control-flow methods, and the efficacy of a new coverage method, which the prototype tool supports.

SUMMARY AND SUGGESTIONS FOR FUTURE WORK

The combination of technologies we have described achieves rapid generation of program-analysis tools, which are easy to change.

Much information of interest for program analysis can be obtained directly from the static and run-time facts generated by the system. However, should significantly different analysis be required, performing it in Prolog is a large saving over conventional-language implementation. It is easy to write queries on the fly, to investigate issues that arise while an experiment is being conducted. These issues may arise from mistakes in the algorithms being implemented, or from more subtle, ill-considered definitions; or, they may involve new methods suggested by properties of a particular example.

This paper has been limited to a description of the prototyping technique. To make the ideas clear we used a very simple sample program and minimized discussion of the testing techniques implemented. We emphasized the ease of creating and modifying test tools, and did not examine their performance. The new techniques of dependency-chain and state-coverage testing were introduced as applications of the prototyping method.

Performance of tools is always an issue, but less so for exploratory research and in the classroom. Poor performance (or worse, inability to handle large programs at all) is rightly blamed for the failure of good ideas to move from research labs into industrial use. Certainly Prolog facts seem a poor medium for storing and searching a large volume of information efficiently. But because analysis is performed interactively and incrementally by the human tester, and when mistakes are found that person often restarts the analysis, the time saved by creating unstructured data may balance time wasted in linear searches. For some testing methods constraint logic programming[27] may achieve performance comparable to imperative-language implementations. Some logic programs lend themselves to automatic parallel decomposition, which would solve a difficult problem with high payoff[28].

ACKNOWLEDGEMENTS

A number of people made important contributions to this project. Andy Babbitt and Shari Powell did most of the "proof of concept" implementation, and responded to many requests for changes as several research projects pulled in different directions on the software. Bruce Gifford was involved from the start of the project, answering questions, helping with coding difficulties in yacc and Prolog, and devising examples. Loren Brown, Andy Bass, Tim McCormick, and Quinton Tormanen helped to clean up the mess that a research prototype becomes. Alex Schonfeld converted most of the original troff document to LaTeX.

REFERENCES

1. P. G. Frankl and E. J. Weyuker, 'A data flow testing tool', *Proceedings SoftFairII, Software Development Tools, Techniques, and Alternatives*, San Francisco, 1987, pp. 46–53.
2. J. Laski, 'Data flow testing in stad', *J. Systems Software*, **12**, 3–14, (January 1990).
3. T. J. Ostrand and E. J. Weyuker, 'Data flow-based test adequacy analysis for languages with pointers', *Proc. Symposium on Software Testing, Analysis, and Verification (TAV4)*, Victoria, BC, October 1991, pp. 74–86.
4. J. R. Horgan and S. London, 'Data flow coverage and the c language', *Proc. Symposium on Software Testing, Analysis, and Verification (TAV4)*, Victoria, BC, October 1991, pp. 87–97.
5. G. Kotik and L. Markosian, 'Automating software analysis and testing using a program transformation system', *Proc. Symposium on Software Testing, Analysis, and Verification (TAV3)*, Key West, December 1989, pp. 75–85.

6. L. G. Stucki, ‘Automatic generation of self-metric software’, *Proceedings IEEE Symposium on Computer Software Reliability*, New York, 1973.
7. M. E. Lesk and E. Schmidt, *Lex - a lexical analyzer generator*, UNIX Programmer’s Manual, v. 2, 1983.
8. S. C. Johnson, *Yacc: yet another compiler compiler*, UNIX Programmer’s Manual, v. 2, 1983.
9. W. Harrison, ‘Pdss: a programmer’s decision support system’, *Data and Knowledge Engineering*, **4**, 115–123, (January 1989).
10. R. A. DeMillo, E. W. Krauser, and A. P. Mathur, ‘Compiler-integrated program mutation’, *Proc. Computer Software and Applications Conf. (COMPSAC)*, Tokyo, September 1991, pp. 351–356.
11. R. G. Hamlet, ‘Testing programs with the aid of a compiler’, *IEEE Trans. on Software Eng.*, **3**, 279–290, (July 1977).
12. V. R. Basili and A. J. Turner, ‘Iterative enhancement: a practical technique for software development’, *IEEE Trans. on Software Eng*, **1**, 390–396, (December 1975).
13. J. D. Gannon, P. R. McMullin, and R. G. Hamlet, ‘Data abstraction implementation, specification, and testing’, *TOPLAS*, **3**, 211–223, (July 1981).
14. R. M. Stallman, *Using and porting GNU CC*, Free Software Foundation, Inc., Cambridge, MA, 1990.
15. M. J. Harrold, B. Malloy, and G. Rothermel, ‘Efficient construction of program dependence graphs’, *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, MA, June 1993, pp. 160–170.
16. S. Rapps and E.Weyuker, ‘Selecting software test data using data flow information’, *IEEE Trans. Software Eng*, **11**, 367–375, (April 1985).
17. P. G. Frankl and E. J. Weyuker, ‘An applicable family of data flow testing criteria’, *IEEE Trans. on Software Eng*, **14**, 1483–1498, (October 1988).
18. W. Clocksin and C. Mellish, *Programming in Prolog*, Springer, New York, 1984.
19. A. Babbitt and S. T. Powell, ‘Building prototype testing tools’, *Proc. Eighth Pacific Northwest Software Quality Conference*, Portland, OR, October 1990, pp. 264–280.
20. D. Hamlet, B. Gifford, and B. Nikolik, ‘Exploring dataflow testing of arrays’, *Proc. Int. Conf. on Software Engineering (ICSE)*, Baltimore, MD, May 1993, pp. 118–129.
21. J. M. Purtilo and E. L. White, ‘Using program adaptation techniques for test coverage in ada programs’, *Proc. Eighth Pacific Northwest Software Quality Conference*, Portland, OR, October 1990, pp. 281–294.
22. J. N. Campbell, *Data-flow Analysis of Software Change*, Master’s thesis, Oregon Graduate Institute, Beaverton, OR, November 1990.
23. S. Ntafos, ‘An evaluation of required element testing strategies’, *Proc. Int. Conf. on Software Engineering (ICSE)*, Orlando, FL, March 1984, pp. 250–256.
24. M. Weiser, ‘Program slicing’, *IEEE Trans. Software. Eng.*, **10**, 352–357, (July 1984).
25. B. Marick, ‘Experience with the cost of different coverage goals for testing’, *Proc. Ninth Pacific Northwest Software Quality Conference*, Portland, OR, October 1991, pp. 147–164.
26. R. G. Hamlet, ‘Probable correctness theory’, *Info. Proc. Letters*, **25**, 17–25, (April 1987).
27. J. Jaffar and J-L. Lassez, ‘Constraint logic programming’, *POPL*, Munich, January 1987.
28. B. Choi, A. Mathur, and B. Pattison, ‘Pmothra: scheduling mutants for execution on a hypercube’, *Proc. Symposium on Software Testing, Analysis, and Verification (TAV3)*, Key West, December 1989, pp. 58–65.